5. Patrons de programmation parallèle avec PRuby

5.1 La bibliothèque `PRuby`

Cinq patrons de base de programmation parallèle :

1. Parallélisme `fork/join`
2. Parallélisme de boucles
3. Parallélisme de données
4. Parallélisme «Coordonnateur/Travailleurs»
5. Parallélisme de flux de données

Ces patrons de programmation sont représentatifs de ce qu'on retrouve dans divers langages de programmation modernes.

`PRuby` est une bibliothèque — un ***gem*** — pour la programmation parallèle en Ruby.

Un point important : `PRuby` n'est pas pour la programmation parallèle **haute performance** ☹

`PRuby` peut plutôt être vue comme une forme de **pseudocode parallèle exécutable**.

Dans d'autres chapitres, on verra d'autres langages. . . plus performants.

5.2 Parallélisme *fork–join* : `pcall` et `future`

> *fork–join model : (computer science) A method of programming on parallel machines in which one or more child processes* **branch out from the root task** *when it is time to do work in parallel, and* **end when the parallel work is done**.
>
> *Source : http://www.answers.com/topic/fork-join-model*

Mécanisme qu'on retrouve dans plusieurs langages :

- Processus Unix avec `fork` et `wait/waitpid`.
- *Threads* Posix avec `pthread_create` et `pthread_join`.
- *Threads* Java avec `start()` et `join()`.
- *Threads* Ruby avec `new` et `join`.
- *Threads* Cilk avec `spawn` et `sync`.
- Etc.

5.2.1 Factoriel

Différentes versions d'une méthode factorielle
`fact( n )` = $1 \times 2 \times 3 \times \ldots \times (n-1) \times$ `n`.

Programme Ruby 5.1 — Version récursive séquentielle... et linéaire

Programme Ruby 5.1 Méthode récursive séquentielle (linéaire) pour calculer `fact(n)`.

```
def fact( n )
  if n == 0
    1
  else
    n * fact( n - 1 )
  end
end
```

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

Exercice 5.1: Extraction de parallélisme de `fact`?

Programme Ruby 5.2 — version récursive séquentielle... et dichotomique

Programme Ruby 5.2 Méthode récursive séquentielle (dichotomique) pour calculer `fact(n)`.

```ruby
def fact( n )

  # Fonction auxiliaire:
  #   fact(n) = fact_(1, n).
  def fact_( i, j )
    return i if i == j

    mid = (i + j) / 2
    r1 = fact_( i, mid )
    r2 = fact_( mid + 1, j )

    r1 * r2
  end

  fact_( 1, n )
end
```

Programme Ruby 5.3 — version récursive parallèle avec `pcall`

Programme Ruby 5.3 Méthode récursive parallèle pour calculer `fact(n)`.

```
def fact( n )
  def fact_( i, j )
    # Cas de base = probleme trivial (1 seul element).
    return i if i == j

    # Cas recursif pour probleme plus complexe:
    #   solution parallele et recursive
    r1, r2 = nil, nil
    mid = (i + j) / 2

    PRuby.pcall( lambda { r1 = fact_(i, mid) },
                 lambda { r2 = fact_(mid + 1, j) } )

    r1 * r2
  end

  fact_( 1, n )
end
```

```
PRuby.pcall\
  lambda { r1 = fact(i, mid, seuil) },
  lambda { r2 = fact(mid+1, j, seuil) }


PRuby.pcall lambda { r1 = fact(i, mid, seuil) },
            lambda { r2 = fact(mid+1, j, seuil) }


PRuby.pcall -> { r1 = fact(i, mid, seuil) },
            -> { r2 = fact(mid+1, j, seuil) }

...
```

Programme Ruby 5.4 — version récursive parallèle avec `Ruby.pcall` et avec un seuil de récursion

Programme Ruby 5.4 Méthode récursive parallèle pour calculer `fact(n)` avec troncation de la récursion.

```
def fact( n, seuil )
  def fact_( i, j, seuil )
    # Probleme simple, mais non trivial
    #   => solution iterative sequentielle.
    return (i..j).reduce(:*) if j - i <= seuil

    # Probleme complexe =>
    #   solution recursive parallele.
    r1, r2 = nil, nil
    mid = (i + j) / 2

    PRuby.pcall( lambda { r1 = fact_(i, mid, seuil) },
                 lambda { r2 = fact_(mid + 1, j, seuil) } )

    r1 * r2
  end

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

```
>> [10, 20, 30, 40].reduce { |x,y| x + y }
=> 100

>> [10, 20, 30, 40].reduce(999) { |x,y| x + y }
=> 1099

>> [].reduce { |x,y| x + y }
=> nil

>> [].reduce(0) { |x,y| x + y }
=> 0

>> [10, 20, 30, 40].reduce(999, &:+)
=> 1099

>> [10, 20, 30, 40].reduce(999, :+)
=> 1099

>> (10..20).reduce('') { |x, y| x + ', ' + y.to_s }
=> ', 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20'
```

Figure 5.1: Exemples d'exécution de la méthode Ruby `reduce` (du module `Enumerable`).

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 )`.

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

Programme Ruby 5.5 — Version récursive parallèle avec des `futures`

Programme Ruby 5.5 Méthode récursive parallèle pour calculer `fact(n)` avec des `futures`.

```ruby
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 = PRuby.future { fact_(mid + 1, j, seuil) }

    r1.value * r2.value
  end

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

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?

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

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

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

Remarque : Voir le chapitre «Diviser pour régner».

5.2.2 Somme de deux tableaux

Méthode pour effectuer la somme de deux `Arrays`.

Programme Ruby 5.6 Méthode séquentielle itérative pour faire la somme de deux tableaux.

```
def somme_tableaux( a, b )
  DBC.require a.size == b.size # Precondition: omise ailleur

  c = Array.new(a.size)

  (0...c.size).each do |k|
    c[k] = a[k] + b[k]
  end

  c
end
```

Programme Ruby 5.7 Méthode parallèle itérative *à granularité fine* pour faire la somme de deux tableaux avec `pcall`.

```
def somme_tableaux( a, b )
  c = Array.new(a.size)

  PRuby.pcall( 0...c.size,
               lambda { |k| c[k] = a[k] + b[k] }
               )

  c
end
```

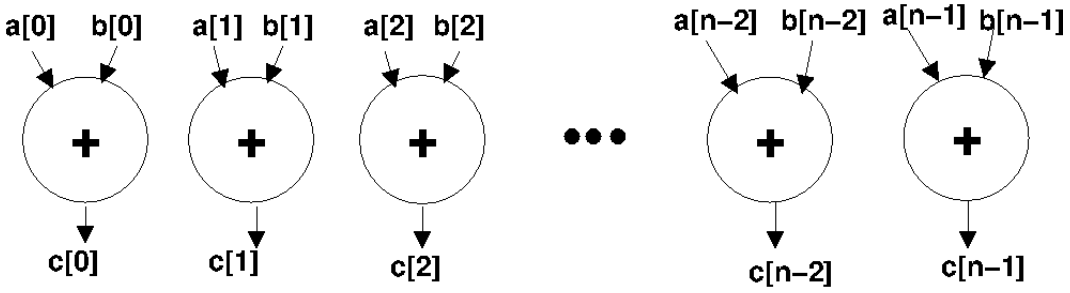


Figure 5.2: Graphe de dépendances des tâches pour le calcul parallèle de la somme de deux tableaux.

Parallélisme «embarassant»!
Parallélisme «à granularité (très!) fine»!

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

Exercice 5.5: Performances si deux gros tableaux?

Programme Ruby 5.8 Méthode parallèle itérative à granularité grossière pour faire la somme de deux tableaux avec `pcall`.

```
def somme_tableaux( a, b, nb_threads = PRuby.nb_threads )
  DBC.require a.size == b.size && a.size % nb_threads == 0

  # Les indices pour la tranche du thread no. k.
  def indices_tranche( k, n, nb_threads )
    (k * n / nb_threads)..((k + 1) * n / nb_threads - 1)
  end

  # Somme sequentielle de la tranche pour indices (inclusif)
  def somme_seq( a, b, c, indices )
    indices.each { |k| c[k] = a[k] + b[k] }
  end

  # On alloue le tableau pour le resultat.
  c = Array.new(a.size)

  # On active les divers threads,
  # en specifiant les indices de la tranche a traiter.
  PRuby.pcall( 0...nb_threads,
               lambda do |k|
                inds = indices_tranche(k, c.size, nb_threads
                somme_seq( a, b, c, inds )
               end
               )

  # On retourne le resultat.
  c
end
```

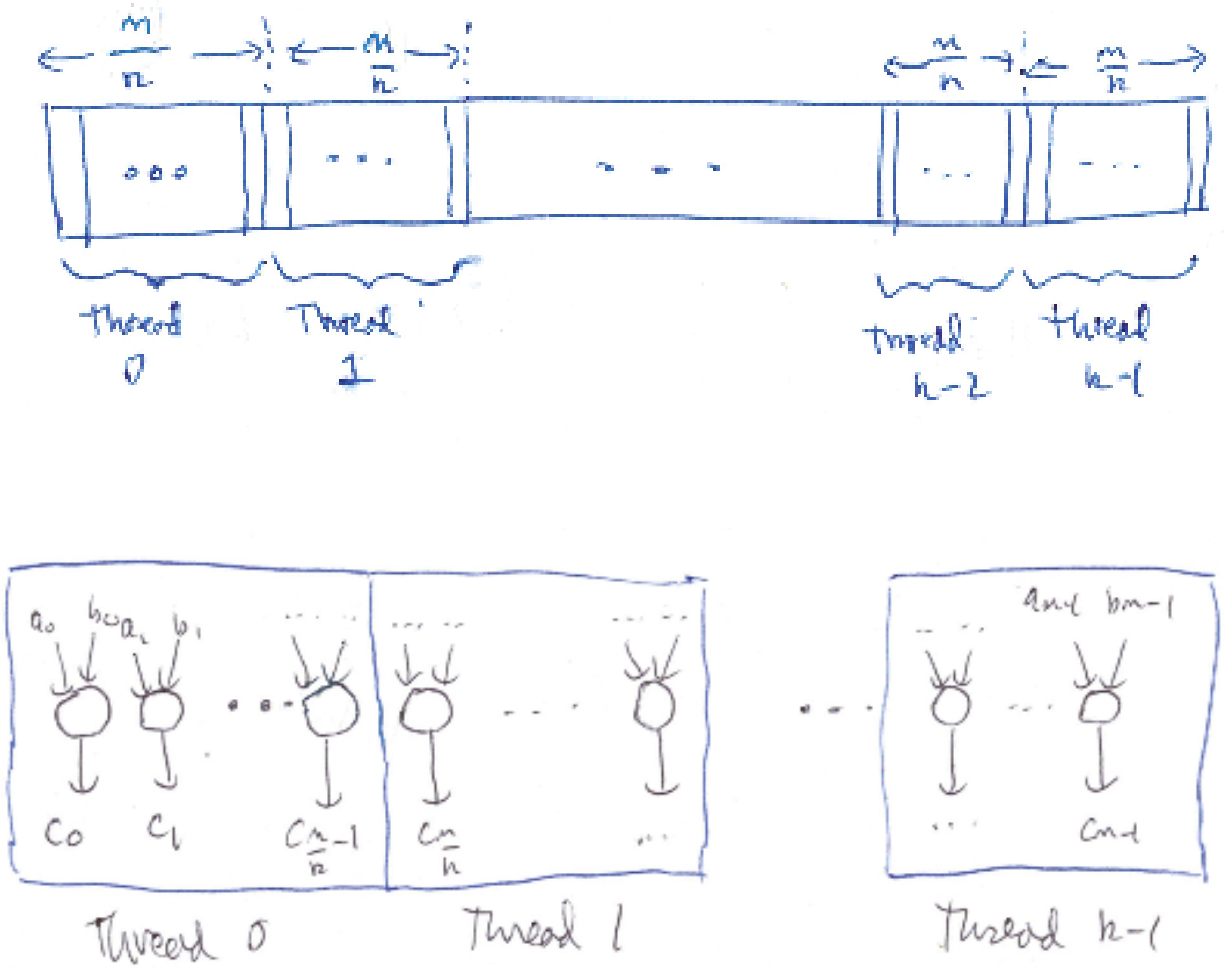


Figure 5.3: Répartition par blocs d'éléments adjacent de `n` éléments entre `k` *threads*. Chaque *thread* traite `n/k` éléments adjacents — pour simplifier, on suppose que `n` est divisible par `k`.

5.2.3 Calcul de π à l'aide d'une méthode Monte Carlo

Pour certains problèmes, dits avec «**parallélisme semi-embarrassant**», le problème peut être décomposé en un grand nombre de tâches qui sont indépendantes... mais pas tout à fait.

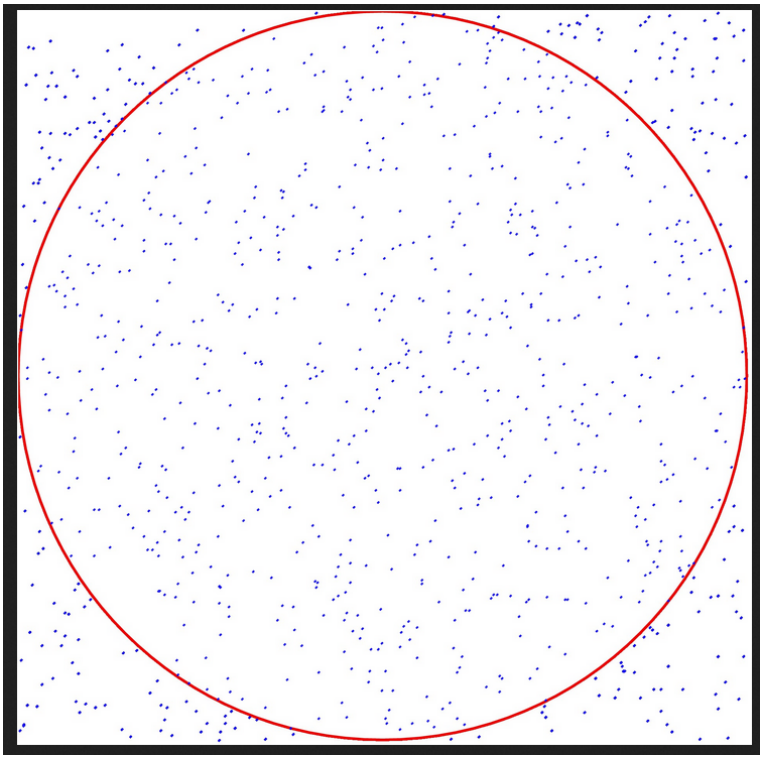


Figure 5.4: Estimation de la valeur de π à l'aide d'une méthode Monte Carlo (source : `http://i.stack.imgur.com/uYrT5.jpg`).

Le rapport entre la surface du carré et celle du cercle est défini comme suit :

- Aire du cercle (rayon $r = 1$) : $A_{cercle} = \pi r^2 = \pi$
- Aire du carré de coté 2 : $A_{carré} = (2)^2 = 4$
- Rapport cercle sur carré : $\frac{A_{cercle}}{A_{carré}} = \frac{\pi}{4}$
- Valeur de π : $\pi = 4\frac{A_{cercle}}{A_{carré}}$

$$\pi \approx 4 \times \frac{nb\_total\_dans\_le\_cercle}{nb\_de\_lancers}$$

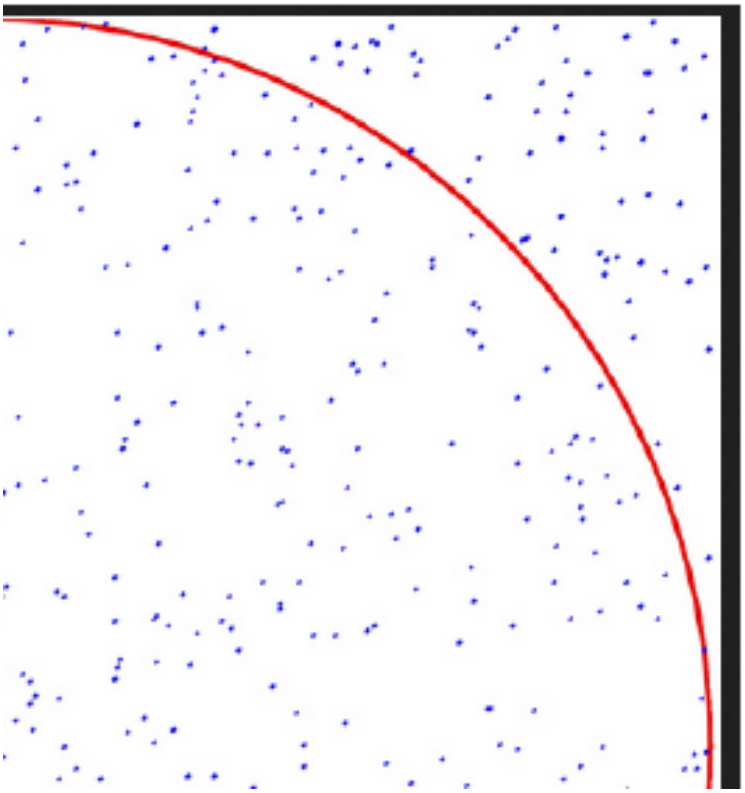


Figure 5.5: Estimation de la valeur de π à l'aide d'une méthode Monte Carlo : La méthode présentée travaille exclusivement sur le quadrant supérieur droit.

Programme Ruby 5.9 Méthode parallèle pour estimer la valeur de π à l'aide d'une méthode Monte Carlo (style **fonctionnel**).

```ruby
def nb_dans_cercle_seq( nb_lancers )
  nb = 0
  nb_lancers.times do
    # On genere un point aleatoire.
    x, y = rand, rand

    # On incremente s'il est dans le cercle
    nb += 1 if x * x + y * y <= 1.0
  end

  nb
end

def evaluer_pi( nb_lancers, nb_threads = PRuby.nb_threads
  # On active les divers threads en creant des futures.
  futures_nb_dans_cercle = (0...nb_threads).map do
    PRuby.future { nb_dans_cercle_seq(nb_lancers/nb_threa
  end

  # On recoit et on additionne les resultats des futures.
  nb_total_dans_cercle =
    futures_nb_dans_cercle
    .map(&:value)
    .reduce(&:+)

  4.0 * nb_total_dans_cercle / nb_lancers
end
```

```
>>  [10, 20, 30, 40].map { |x| 2 * x }
=>  [20, 40, 60, 80]

>>  [].map { |x| 2 * x }
=>  []

>> a = (0...10).map { |i| 10*i+1 }
=> [1, 11, 21, 31, 41, 51, 61, 71, 81, 91]

>> a.map { |x| 10 * x }
=> [10, 110, 210, 310, 410, 510, 610, 710, 810, 910]
>> a
=> [1, 11, 21, 31, 41, 51, 61, 71, 81, 91]

>> a.map! { |x| 10 * x }
=> [10, 110, 210, 310, 410, 510, 610, 710, 810, 910]
>> a
=> [10, 110, 210, 310, 410, 510, 610, 710, 810, 910]

>> -3.abs
=> 3

>> [-10, 10, 20, -2, 0].map(&:abs)
=> [10, 10, 20, 2, 0]

>> [-10, 10, 20, -2, 0].map(:abs)
ArgumentError: wrong number of arguments calling `map`
                (1 for 0)
        from (irb):6:in 'evaluate'
        ...
```

Figure 5.6: Exemples d'exécution de la méthode Ruby `map` (du module `Enumerable`).

Programme Ruby 5.10 Méthode parallèle pour estimer la valeur de π à l'aide d'une méthode Monte Carlo, dans un style plus **«impératif»**.

```ruby
def evaluer_pi( nb_lancers, nb_threads = PRuby.nb_threads
  # On active les threads en creant des futures.
  futures_nb_dans_cercle = []
  nb_threads.times do
    futures_nb_dans_cercle << PRuby.future do
      nb_dans_cercle_seq( nb_lancers / nb_threads )
    end
  end

  # On recoit les resultats des futures.
  les_nbs = []
  futures_nb_dans_cercle.each do |f|
    les_nbs << f.value
  end

  # On additionne les resultats intermediaires.
  nb_total_dans_cercle = 0
  les_nbs.each do |nb|
    nb_total_dans_cercle += nb
  end

  4.0 * nb_total_dans_cercle / nb_lancers
end
```

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*.

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

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?

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

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
```

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

5.3 Parallélisme de boucles : `peach` et `peach_index`

Une boucle `for` est dite «boucle définie».

Si les itérations sont **indépendantes** — notamment, n'écrivent pas dans les mêmes variables — alors **elles peuvent être exécutées en parallèle.**

Plusieurs langages parallèles introduisent des boucles parallèles : `parallel_for`, `forall`, `foreach`, etc.

En `PRuby` :
`peach` et `peach_index`
= variantes parallèles de `each` et `each_index`.

Exemple Ruby 5.1 Différences entre `each` et `each_index` pour les `Array` et `Range`.

```
>> [10, 20, 30].each { |x| puts x }
10
20
30
=> [10, 20, 30]

>> [10, 20, 30].each_index { |x| puts x }
0
1
2
=> [10, 20, 30]

>> (1..3).each { |x| puts x }
1
2
3
=> 1..3

>> (1..3).each_index { |x| puts x }
NoMethodError: undefined method `each_index' for 1..3:Range
...
```

5.3.1 Somme de deux tableaux

Programme Ruby 5.11 Méthode parallèle itérative *à granularité fine* pour faire la somme de deux tableaux avec `peach`.

```
def somme_tableaux( a, b, _nb_threads )
  c = Array.new(a.size)

  (0...c.size).peach( nb_threads: c.size ) do |k|
    c[k] = a[k] + b[k]
  end

  c
end
```

à granularité fine

Programme Ruby 5.12 Méthode parallèle itérative *à granularité grossière* pour faire la somme de deux tableaux avec `peach`.

```
def somme_tableaux( a, b, nb_threads = PRuby.nb_threads )
  c = Array.new(a.size)

  (0...c.size).peach( nb_threads: nb_threads ) do |k|
    c[k] = a[k] + b[k]
  end

  c
end
```

à granularité grossière

- La méthode `peach` peut recevoir divers arguments par mot-clé :
  - nombre de *threads* à utiliser pour exécuter les itérations.

```
col.peach { ... }
=
col.peach( nb_threads: PRuby.nb_threads ) { ... }
```

- Les itérations peuvent être distribuées **de différentes façons** :
  - ```
    col.peach { ... }
    =
    col.peach( static: true ) { ... }
    ```
    ```

- Si un entier `k` est spécifié, on a une répartition statique et cyclique par bloc de `k` éléments.

  Par exemple, 12 éléments à répartir entre 3 *threads* :

  - `static: true` :

t_0	t_0	t_0	t_0	t_1	t_1	t_1	t_1	t_2	t_2	t_2	t_2

  - `static: 1` :

t_0	t_1	t_2	t_0	t_1	t_2	t_0	t_1	t_2	t_0	t_1	t_2

  - `static: 2` :

t_0	t_0	t_1	t_1	t_2	t_2	t_0	t_0	t_1	t_1	t_2	t_2

  - `static: 3` :

t_0	t_0	t_0	t_1	t_1	t_1	t_2	t_2	t_2	t_0	t_0	t_0

  - `static: 4` :

t_0	t_0	t_0	t_0	t_1	t_1	t_1	t_1	t_2	t_2	t_2	t_2

- Il est aussi possible de spécifier une répartition *dynamique*.

5.3.2 Calcul de π à l'aide d'une méthode Monte Carlo

Programme Ruby 5.13 Méthode parallèle pour estimer la valeur de π à l'aide d'une méthode Monte Carlo et en utilisant un `peach`.

```
def evaluer_pi( nb_lancers, nb_threads = PRuby.nb_threads
  nb_dans_cercle = Array.new( nb_threads )

  (0...nb_threads).peach( nb_threads: nb_threads ) do |k|
    nb_dans_cercle[k] =
       nb_dans_cercle_seq(nb_lancers / nb_threads)
  end

  nb_total_dans_cercle = nb_dans_cercle.reduce(&:+)

  4.0 * nb_total_dans_cercle / nb_lancers
end
```

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}"
```

Exercice 5.9: Somme avec `peach`.

5.3.3 Produit de matrices

Programme Ruby 5.14 Méthode séquentielle pour effectuer le produit de deux matrices.

```
def produit( a, b )
  DBC.require a.nb_colonnes == b.nb_lignes

  c = Matrice.new( a.nb_lignes, b.nb_colonnes )

  (0...c.nb_lignes).each do |i|
    (0...c.nb_colonnes).each do |j|
      c[i, j] = 0
      (0...a.nb_colonnes).each do |k|
        c[i, j] += a[i, k] * b[k, j]
      end
    end
  end

  c
end
```

Programme Ruby 5.15 Méthode **parallèle** à granularité *(très!) fine* pour effectuer le produit de deux matrices.

```ruby
def produit( a, b )
  c = Matrice.new( a.nb_lignes, b.nb_colonnes )
  nbl = c.nb_lignes # Vars...pour mise en page
  nbc = c.nb_colonnes

  (0...nbl).peach(nb_threads: nbl) do |i|
    (0...nbc).peach(nb_threads: nbc) do |j|
      c[i, j] = 0
      (0...a.nb_colonnes).each do |k|
        c[i, j] += a[i, k] * b[k, j]
      end
    end
  end

  c
end
```

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?

Exercice 5.10: Nombre de *threads* pour deux matrices.

Programme Ruby 5.16 Méthode parallèle à granularité *grossière* pour effectuer le produit de deux matrices, avec répartition entre les *threads* par ligne.

```ruby
def produit( a, b )
  c = Matrice.new( a.nb_lignes, b.nb_colonnes )

  (0...c.nb_lignes).peach( nb_threads: c.nb_lignes ) do |i|
    (0...c.nb_colonnes).each do |j|
      c[i, j] = 0
      (0...a.nb_colonnes).each do |k|
        c[i, j] += a[i, k] * b[k, j]
      end
    end
  end

  c
end
```

Question : Combien de *threads* seront créés?

Programme Ruby 5.17 Méthode parallèle à granularité *encore plus grossière* pour effectuer le produit de deux matrices, avec répartition par **blocs** de lignes.

```
def produit( a, b )
  c = Matrice.new( a.nb_lignes, b.nb_colonnes )

  (0...c.nb_lignes).peach do |i|
    (0...c.nb_colonnes).each do |j|
      c[i, j] = 0
      (0...a.nb_colonnes).each do |k|
        c[i, j] += a[i, k] * b[k, j]
      end
    end
  end

  c
end
```

Question : Combien de *threads* seront créés?

Équivalents :

- `a.peach`
- `a.peach( nb_threads: PRuby.nb_threads )`
- `a.peach( nb_threads: System::CPU.count )`

Modes de répartition des matrices entre *threads*

Certains langages de programmation — Fortran 90, HPF — permettent d'exprimer *explicitement* le mode de répartition.

Figure 5.7: Distribution par bloc vs. distribution cyclique pour un tableau à 1 dimension (source : `http://www.dais.unive.it/~calpar/New_HPC_course/5_Parallel_Patterns.pdf`).

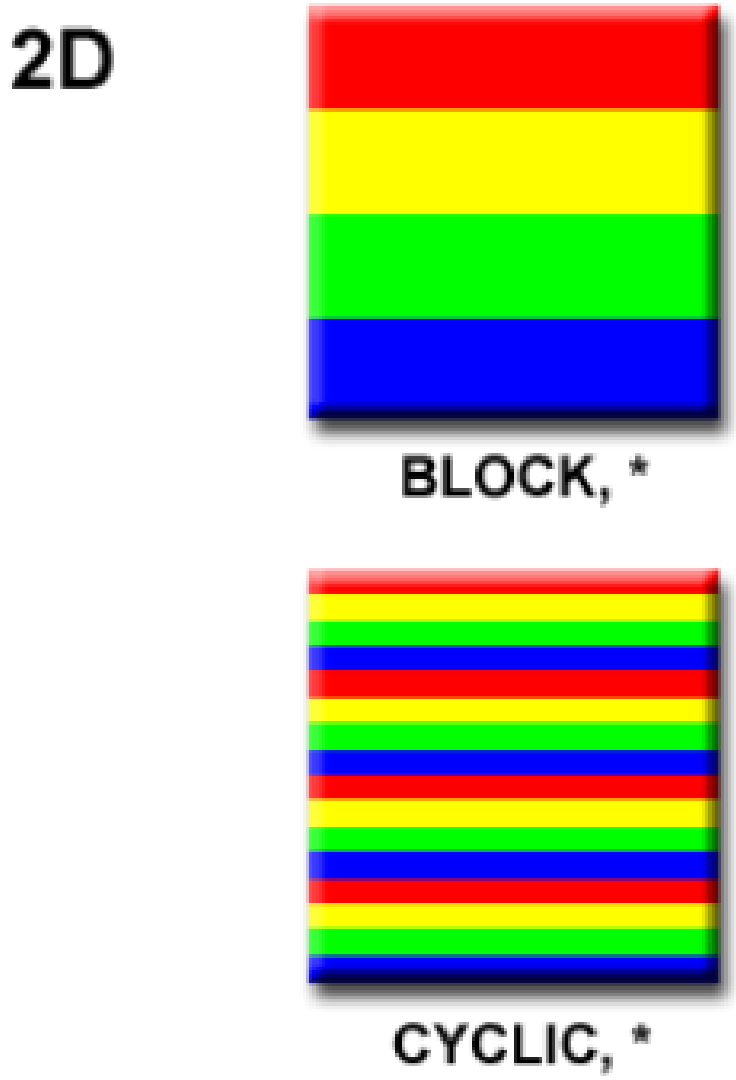


Figure 5.8: Distribution par bloc vs. distribution cyclique pour une matrice à 2 dimensions (source : `http://www.dais.unive.it/~calpar/New_HPC_course/5_Parallel_Patterns.pdf`).

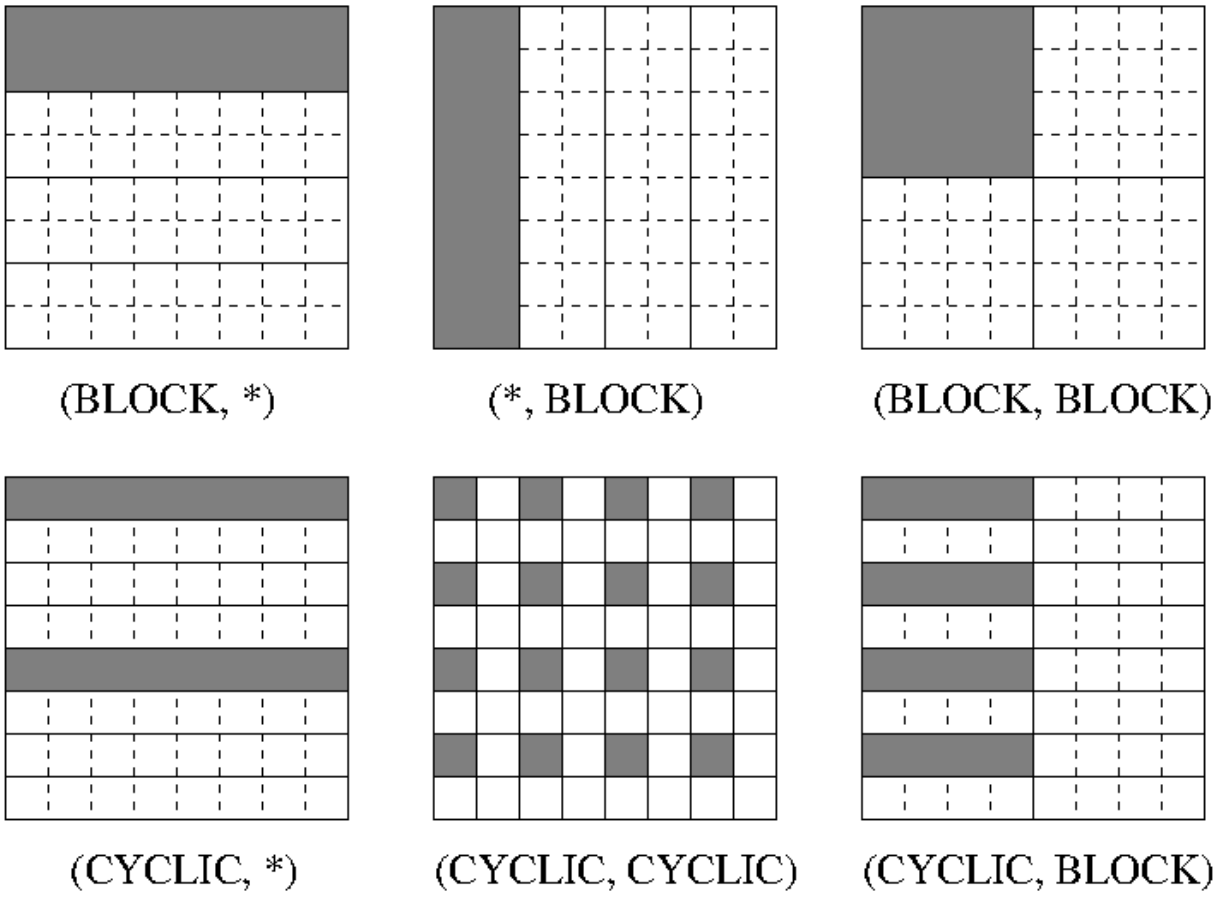


Figure 5.9: Différents types de distribution des données pour un tableau 8×8 entre quatre (4) processus en HPF.

5.4 Parallélisme de données et approche de style fonctionnel : `pmap` et `preduce`

5.4.1 Parallélisme de données

Parallélisme de données = Application d'une même opération *sur tous les éléments d'une collection.*

5.4.2 Application, réduction et préfixes

- *Application* : applique une méthode sur chacun des éléments pour obtenir **une nouvelle collection**

  Soit f une opération unaire et $C = [c_1, \ldots, c_n]$.

  L'application de f à C produira :

  $$R = [f(c_1), \ldots, f(c_n)]$$

- *Réduction* : applique une méthode *binaire* sur les éléments de C pour obtenir **une valeur généralement d'un type plus simple**.

  Soit $\oplus$ une opération binaire et $C = [c_1, \ldots, c_n]$.

  L'application de $\oplus$ sur C produit :

$$r = ((((c_1 \oplus c_2) \oplus c_3) \oplus \ldots) \oplus c_n)$$

- *Calcul de préfixes* : applique une opération binaire associative sur les éléments de C pour obtenir une autre collection de même taille :

$$R = [c_1, c_1 \oplus c_2, c_1 \oplus c_2 \oplus c_3, \ldots, c_1 \oplus \ldots \oplus c_{n-1} \oplus c_n]$$

- **Ruby** :

```
[1, 2, 3, 4].map { |x| 2 * x }
=>
[2, 4, 6, 8]

[1, 2, 3, 4].reduce(1) { |prod, n| prod * n }
=>
24
```

- **Lisp** :

```
(map 'list #'(lambda (x) (* 2 x)) '(1 2 3 4))
=>
(2 4 6 8)

(reduce #'* '(1 2 3 4))
=>
24
```

- **Haskell** :

```
map (2*) [1,2,3,4]
=>
[2,4,6,8]

foldr (*) 1 [1,2,3,4]
=>
24
```

Exemple d'exécution 5.1: Exemples d'utilisation d'opérations de style `map` et `reduce` dans divers langages de programmation : Ruby, Lisp, Haskell.

5.4.3 Application et réduction en parallèle

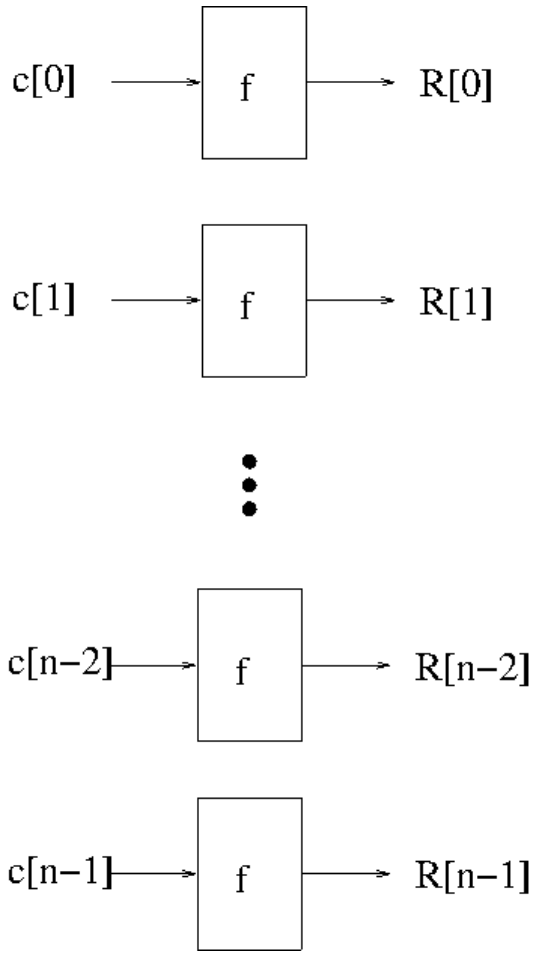
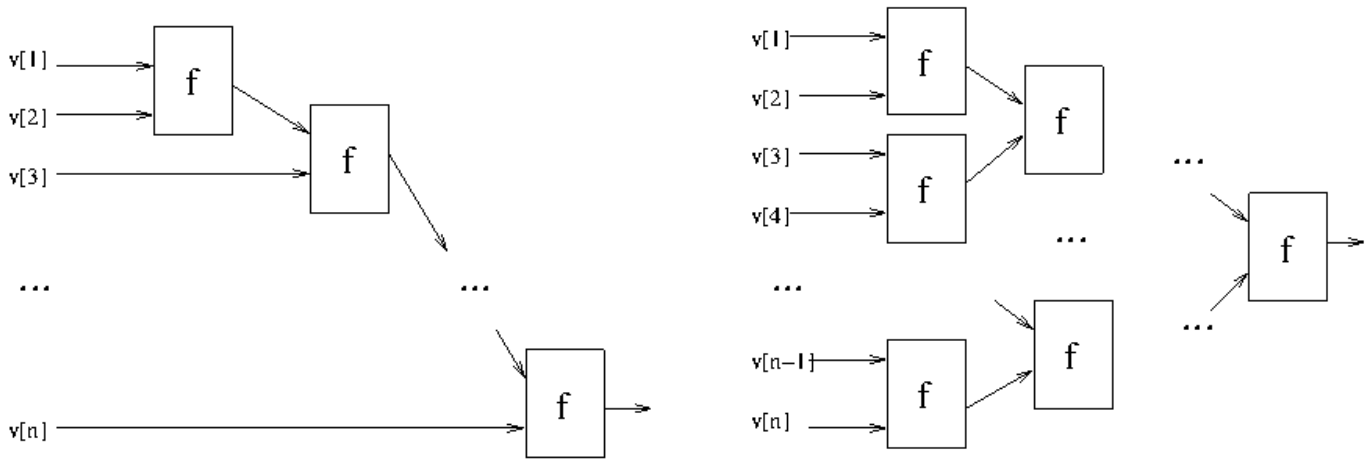


Figure 5.10: Graphe de dépendances pour une α-application.

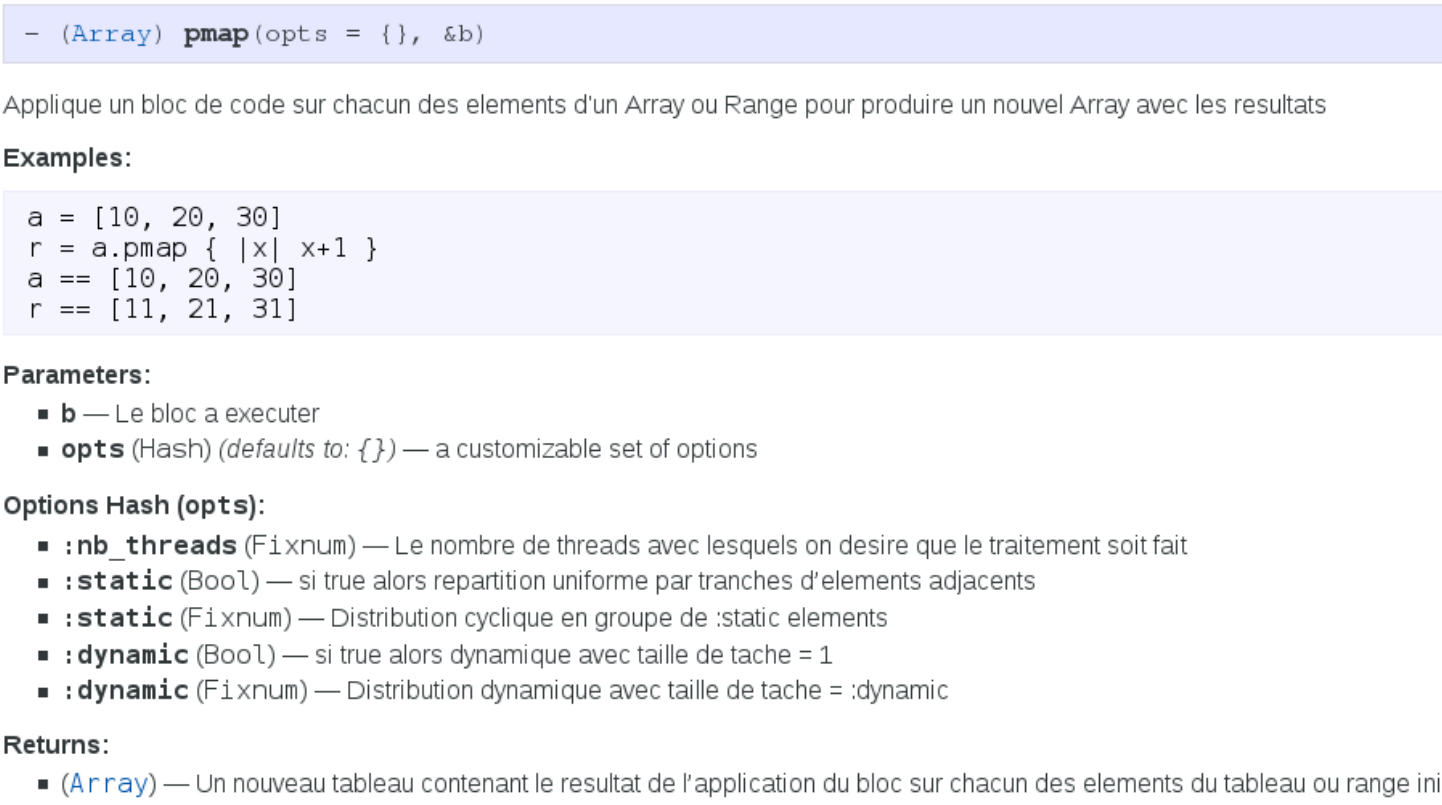


(a) Schéma béta

(b) Schéma béta–logarithmique

Figure 5.11: Graphes de dépendances pour une réduction β vs. une réduction β-logarithmique.

La Figure 5.12 présente la description de la méthode `pmap` telle qu'on la retrouve dans la documentation en ligne de la bibliothèque `PRuby`.



```
- (Array) pmap(opts = {}, &b)
```

Applique un bloc de code sur chacun des elements d'un Array ou Range pour produire un nouvel Array avec les resultats

Examples:

```
a = [10, 20, 30]
r = a.pmap { |x| x+1 }
a == [10, 20, 30]
r == [11, 21, 31]
```

Parameters:

- **b** — Le bloc a executer
- **opts** (Hash) *(defaults to: {})* — a customizable set of options

Options Hash (opts):

- **:nb_threads** (Fixnum) — Le nombre de threads avec lesquels on desire que le traitement soit fait
- **:static** (Bool) — si true alors repartition uniforme par tranches d'elements adjacents
- **:static** (Fixnum) — Distribution cyclique en groupe de :static elements
- **:dynamic** (Bool) — si true alors dynamique avec taille de tache = 1
- **:dynamic** (Fixnum) — Distribution dynamique avec taille de tache = :dynamic

Returns:

- (Array) — Un nouveau tableau contenant le resultat de l'application du bloc sur chacun des elements du tableau ou range ini

Figure 5.12: Documentation de la méthode `pmap`.

5.4.4 Somme de deux tableaux

Programme Ruby 5.18 Méthode parallèle pour effectuer la somme de deux tableaux avec `pmap`.

```
def somme_tableaux( a, b, nb_threads = PRuby.nb_threads )
  (0...a.size).pmap( nb_threads: nb_threads ) do |k|
    a[k] + b[k]
  end
end
```

5.4.5 Calcul de π à l'aide d'une méthode Monte Carlo

Programme Ruby 5.19 Méthode parallèle pour estimer la valeur de π à l'aide d'une méthode Monte Carlo avec `pmap`.

```
def evaluer_pi( nb_lancers, nbt = PRuby.nb_threads )
  nb_dans_cercle = (0...nbt).pmap(nb_threads: nbt) do
    nb_dans_cercle_seq( nb_lancers / nbt )
  end

  nb_total = nb_dans_cercle.reduce(&:+)

  4.0 * nb_total / nb_lancers
end
```

Programme Ruby 5.20 Méthode parallèle pour estimer la valeur de π à l'aide d'une méthode Monte Carlo avec `preduce`.

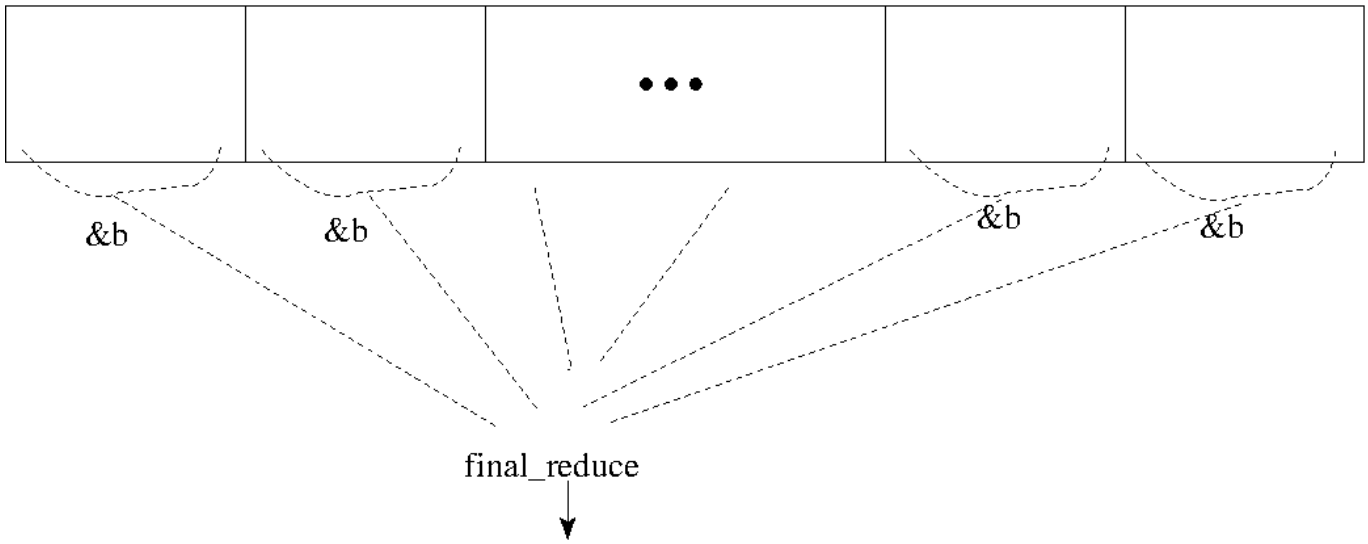
```
def evaluer_pi( nb_lancers, nbt = PRuby.nb_threads )
  total = (0...nbt)
            .preduce(0, nb_threads: nbt) do |nb, _numt|
    nb + nb_dans_cercle_seq( nb_lancers / nbt )
  end

  4.0 * total / nb_lancers
end
```

5.4.6 Factoriel

Programme Ruby 5.21 Méthode parallèle pour calculer `fact(n)` avec `preduce`.

```
def fact( n, nbt )
  (1..n).preduce( 1, nb_threads: nbt ) do |prod, k|
    prod * k
  end
end
```

5.4.7 Forme générale de `preduce`




```
- (T, Fixnum) preduce(val_initiale, opts = {}, &b)
```

Parameters:

- **`val_initiale`** — Valeur initiale a utiliser, qui devrait etre l'element neutre si l'operation est cumulative (+, *, etc.)
- **`b`** — Le bloc de code a executer sur chacun des elements d'une tranche
- **`opts`** (Hash) *(defaults to: {})* — a customizable set of options

Options Hash (`opts`):

- **`:nb_threads`** (`Fixnum`) — Le nombre de threads avec lesquels on desire que le traitement soit fait
- **`:final_reduce`** (`Symbol`, `Proc`) — L'operateur binaire a utiliser pour la reduction finale des resultats intermedia

Returns:

- (T, `Fixnum`) — La valeur finale reduite. Si self.class == Array<T> alors return.class == T sinon return.class = Fixn

Requires:

- Le bloc recoit deux arguments... et devrait etre associatif
- La fonction final_reduce recoit deux arguments... et devrait etre associative

Figure 5.13: Documentation de la méthode `preduce`.

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`?

Exercice 5.11: Méthode `pmax` pour la classe `Ensemble`.

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]
  ```

Exercice 5.12: Méthode `pselect` sur une collection de type `Array`.

Examples:

```
a = [10, 20, 30]
r = a.preduce(0) { |x, y| x+y }
r == 60
a == [10, 20, 30]

r = a.preduce(23) { |x, y| [x, y].max }
r == 30
r = a.preduce(99) { |x, y| [x, y].max }
r == 99

a = [11, 2, 30, 40, 40, 39, 38, 5, 6]
r = a.preduce(0, nb_threads: 3, final_reduce: :+) do |m, x|
  [m, x].max
end
r == 30 + 40 + 38
```

Figure 5.14: Documentation de la méthode `preduce` (suite).

5.5 Parallélisme style «Coordonnateur/Travailleurs» : `peach/pmap` dynamique et `TaskBag`

Deux grandes façons de créer des *threads* et de leur attribuer des tâches à traiter :

- **Création dynamique** de *threads* : Avec `pcall` ou `future`.

  Question : Quel est le désavantage de cette approche?

- **Répartition statique du travail** entre un nombre fixe de *threads* : Avec `peach/peach_index` ou avec `pmap/preduce`.

  Question : Quel est le désavantage de cette approche?

- Créer des *threads* de façon récursive est coûteux.

  ET

  Si on veut utiliser du parallélisme à granularité grossière avec un nombre fixe de *threads*, c'est plus compliqué pour répartir le travail entre les *threads*.

- Facile de créer un nombre fixe de *threads* et de répartir le travail de façon grossière.

  Sauf que pour que les performances soient bonnes, il faut que chaque «gros» morceau nécessite (plus ou moins) le même temps de traitement de traitement.

Une répartition statique entre un nombre fixe de *threads* ne fonctionne bien *que si le travail à effectuer par chacun des* threads *prend le même temps.*

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**?

Exercice 5.13: Temps d'exécution et accélération pour des tâches de temps variable.

Ce qu'il faut = approche qui permet d'utiliser **un nombre fixe et limité de *threads***, créés en une seule fois, tout en permettant une **répartition plus égale** du travail à faire.

Une telle approche est possible avec une **répartition dynamique des tâches**.

5.5.1 Traitement d'une série de fichier avec `wc` et `pmap`

Sur Unix, `wc` donne le nombre de lignes, de mots et de caractères dans un fichier texte :

```
$ cat foo.txt
1
22 22
333 333 333
4444 4444 4444 4444

$ wc foo.txt
 4 10 40 foo.txt
```

Programme Ruby 5.22 Version parallèle de la méthode `wc` appliquée à une liste de fichiers avec `pmap`.

```ruby
# Compte le nombre de mots dans une ligne.
def nb_mots( ligne )
  ligne.strip.split(/\s+/).size
end

# Version Ruby de wc qui traite un (1) fichier.
def wc1( fich )
  lignes = IO.readlines( fich )

  nb_lignes = lignes.size
  nb_mots   = lignes.map { |l| nb_mots(l) }.reduce(0, &:+)
  nb_cars   = lignes.map(&:size).reduce(0, &:+)

  [nb_lignes, nb_mots, nb_cars, fich]
end


# Fonction qui applique wc sur une liste de fichiers.
def wc( fichs )
  fichs.pmap { |fich| wc1( fich ) }
end
```

Effet de `split` :

```
>> 'bcaadefxyzaa'.split(/a+/)
=> ['bc', 'defxyz']

>> 'abc def xxx   y'.split(/\s+/)
=> ['abc', 'def', 'xxx', 'y']
```

Quel est le principal défaut du Programme Ruby 5.22?

Exercice 5.14: Le défaut de la solution statique pour `wc` ☹

Une solution **simple** à ce problème :

```
def wc( fichs )
  fichs.pmap( dynamic: true ) do |fich|
    wc1( fich )
  end
end
```

Équivalent à un appel avec «`dynamic: 1`».

Comportement de la répartition dynamique :

Lorsqu'un *thread* est activé, sa première action est d'obtenir le prochain élément à traiter **parmi ceux pas encore traités**.

Lorsque le *thread* termine, il tente d'obtenir le prochain élément **pas encore traité**.

Le *thread* se termine **lorsqu'il n'y a plus aucun élément à traiter**.

Il est possible d'augmenter la granularité des tâches :
«`fichs.pmap( dynamic: 5 ) {...}`»
⇒ Chaque tâche va comporter 5 noms de fichiers!

Mise en oeuvre = ***pool de* threads** (*thread pool*)

5.5.2 Approche *Coordonnateur–Travailleurs* et sac de tâches

Note : C'est ce qu'on appelle aussi une approche avec ***pool de* threads** (*thread pool*)

Dans une approche dynamique des tâches, on parle d'une approche Coordonnateur–Travailleurs :

• un des *threads* joue le rôle de **coordonnateur**, qui gère les tâches

• les autres *threads* sont des **travailleurs**, qui traitent les tâches.

Dans certains cas, l'exécution d'une tâche entraine la création d'une ou plusieurs nouvelles tâches.

On utilise alors une structure de données pour les tâches : le **sac de tâches**.

Le code exécuté par les travailleurs :

```
THREAD travailleur( sac_tâches )
  terminé ← false
  WHILE !terminé DO
    obtenir une tâche du sac_tâches # Bloquant!
    IF il restait une tâche à exécuter THEN
      exécuter la tâche obtenue... possiblement
            en générant de nouvelles tâches
    ELSE
      terminé ← true
    END
  END
END
```

5.5.3 Traitement d'une série de fichiers avec `wc` et un TaskBag

Programme Ruby 5.23 Version parallèle de la méthode `wc` appliquée à une liste de fichiers avec un `TaskBag`.

```
def wc( fichiers, taille_tache = 2 )
  nb_trvs = PRuby.nb_threads # Un travailleur par thread.

  # Les taches a mettre dans le sac.
  taches = (0...fichiers.size)
    .step( taille_tache )
    .map { |i| i..[i+taille_tache-1, fichiers.size-1].min }

  res = Array.new(fichiers.size) # Tableau des resultats.

  # On active les travailleurs et on attend qu'ils terminent
  PRuby::TaskBag.create_and_run(nb_trvs, *taches) do |sac|
    sac.each do |i_j|
      res[i_j] = i_j.map { |k| wc1(fichiers[k]) }
    end
  end

  res
end
```

- Coordonnateur =
  thread qui exécute `wc`
- Travailleurs =
  threads qui exécutent le bloc fourni à `create_and_run`

5.5.4 Méthode mystère

Programme Ruby 5.24 Méthode `mystere`.

```
def mystere( n, seuil )
  resultats = PRuby::TaskBag
                    .create_and_run( PRuby.nb_threads,
                                     1..n ) do |sac_taches|
    res = 1

    sac_taches.each do |range|
      i, j = range.first, range.last
      while j - i + 1 > seuil
        m = (i + j) / 2
        sac_taches.put (m + 1)..j
        j = m
      end
      res *= (i..j).reduce(1, :*)
    end

    res
  end

  resultats.reduce(1, :*)
end
```

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

Exercice 5.15: Méthode `mystere`.

5.6 Parallélisme de flux de données avec filtres et pipelines : `source`, «|» et `sink`

Remarque : Dans cette section, on va voir des exemples, parmi les rares cette session, de concurrence qui n'est pas en mémoire partagée.

Les *threads* vont communiquer par l'intermédiaire de **canaux de communication**, sans partager de variables/données avec les autres!

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`.

Exercice 5.16: Méthode `mystere` de la classe `Array`

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`»

Exercice 5.17: Modes de répartition des tâches.

Le parallélisme vient de ce qu'on traite des **flux de données** — *data streams* — et le traitement sur une donnée est décomposé en plusieurs étapes.

$\approx$ **chaine de montage**, pipeline d'exécution, etc.

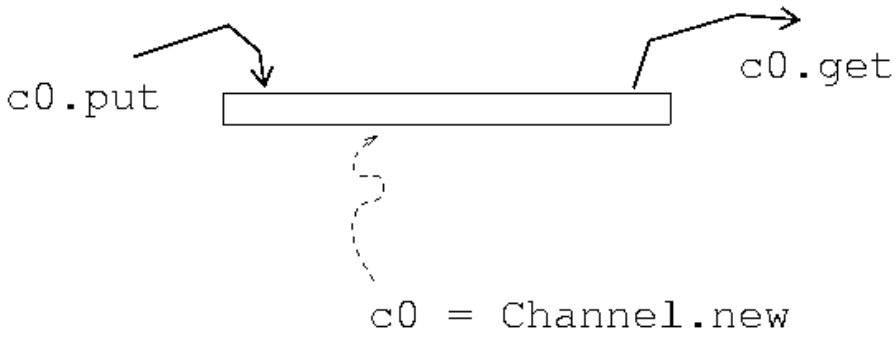
5.6.1 Unix et ses pipes

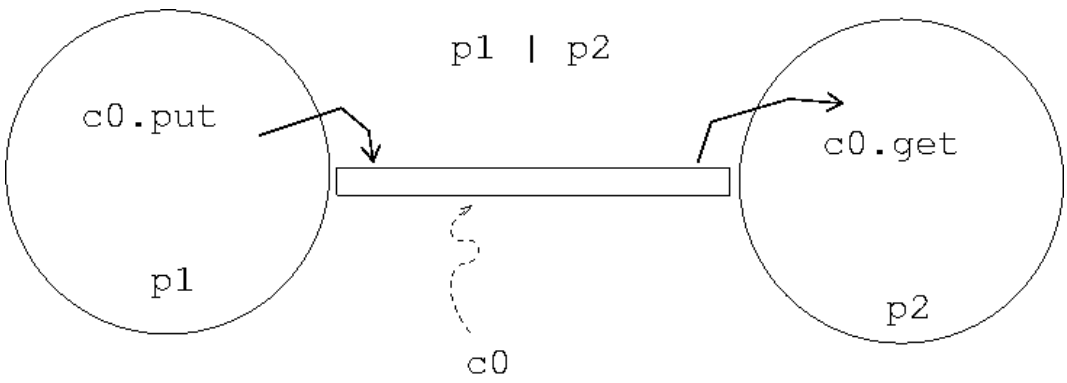
> *(i) Make each program do one thing well. [. . .]*
> *(ii) Expect* **the output of every program to become the input to another**, *as yet unknown,* **program**.
>
> *McIlroy* et al. *(cité dans)*

```
cat $1.tex |
  sed '/\\begin{figure}/,/\\end{figure}/d' |
  sed '/\\begin{table}/,/\\end{table}/d' |
  grep -v "^%" |
  tr "[~]" "[ ]" |
  tr "[\t]" "[\n]"|
  tr "[ ]" "[\n]"|
  grep -v '\\'  |
  wc -w
```

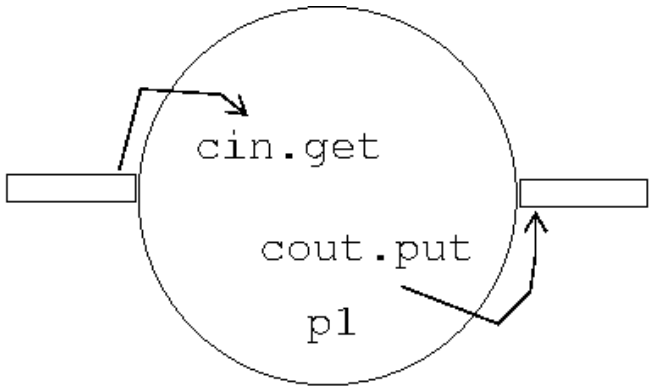
Figure 5.15: Script Unix pour supprimer des commandes LaTeX dans un fichier et compter le nombre de «vrais» mots d'un document.

5.6.2 PRuby et ses Channels




p1 | p2
c0.put
c0.get
p1
p2
c0

Un filtre est défini à l'aide d'une lambda-expression avec **deux arguments**, lesquels donnent accès **aux deux canaux** :




```
p1 = lambda { |cin, cout| ... }
```

La fin d'un flux est signalée par `PRuby::EOS`.

Cette valeur est retournée de façon **persistente**.

Cette valeur est automatiquement transmise lorsque la méthode `close` est appelée.

- (Channel) **initialize**(name = nil, max_size = 0, contents = [])

Constructeur de base.

Parameters:

- **name** (String) *(defaults to: nil)* — Le nom du canal, utilise essentiellement pour le deb
- **max_size** (Fixnum) *(defaults to: 0)* — Taille maximum du canal, i.e., nombre max. d'ele
- **contents** (Array) *(defaults to: [])* — Contenu initial du canal

- (Object) **get**

Obtient l'element en tete du canal.

Returns:

- L'element qui etait en tete du canal. Bloque si empty?

Ensures:

- L'element retourne est retire du canal, donc le canal a un element de moins

- (self) **put**(elem)

Also known as: **<<**

Ajoute un element a la queue du canal.

Parameters:

- **elem** — L'element a ajouter

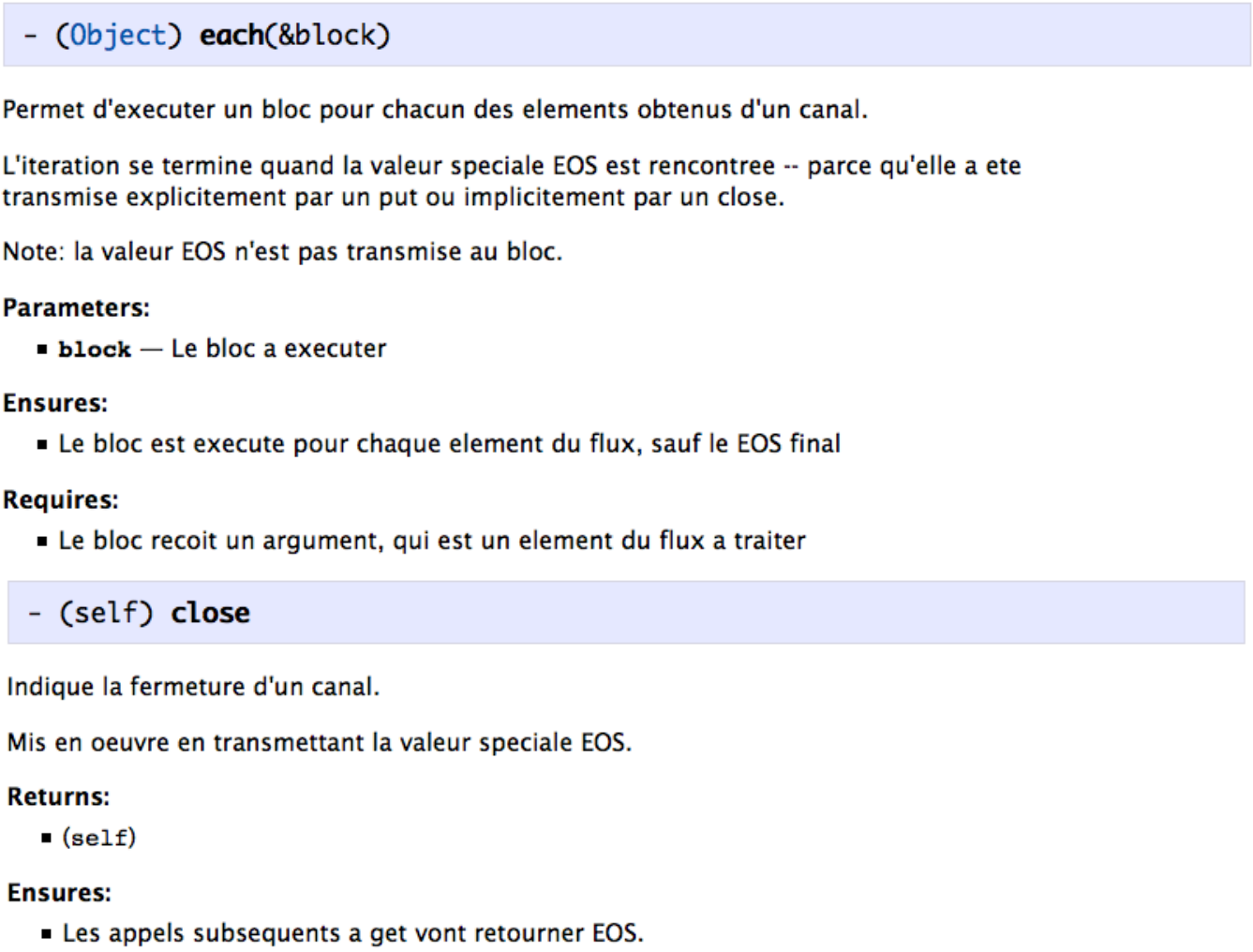
Returns:

- (self)

Requires:

- !elem.nil? && !full?

Figure 5.16: Les principales méthodes de la classe `Channel` : `initialize`, `get`, `put`, `each` et `close` ($1^{ère}$ partie).



- (Object) **each**(&block)

Permet d'executer un bloc pour chacun des elements obtenus d'un canal.

L'iteration se termine quand la valeur speciale EOS est rencontree -- parce qu'elle a ete transmise explicitement par un put ou implicitement par un close.

Note: la valeur EOS n'est pas transmise au bloc.

Parameters:

- **block** — Le bloc a executer

Ensures:

- Le bloc est execute pour chaque element du flux, sauf le EOS final

Requires:

- Le bloc recoit un argument, qui est un element du flux a traiter

- (self) **close**

Indique la fermeture d'un canal.

Mis en oeuvre en transmettant la valeur speciale EOS.

Returns:

- (self)

Ensures:

- Les appels subsequents a get vont retourner EOS.

Figure 5.16: Les principales méthodes de la classe `Channel` : `initialize`, `get`, `put`, `each` et `close` (2^e partie).

5.6.3 Tri unique des mots d'un fichier

Méthode pour identifier les mots d'un fichier texte et produire ces mots, avec un seul mot par ligne, **en ordre** alphabétique **et sans doublon**.

Semblable à «`sort -u`» sous Unix.

Programme Ruby 5.25 Méthode pour trier les mots d'un fichier, en s'assurant que chaque mot apparaît au plus une fois.

```ruby
def trier_mots_uniques( fich_entree, fich_sortie )
  generer_mots = lambda do |cin, cout|
    cin.each do |ligne|
      ligne.split( /\s+/ ).each { |mot| cout << mot }
    end
    cout.close
  end

  filtrer_mots_invalides = lambda do |cin, cout|
    cin.each { |mot| cout << mot if /^\w+$/ =~ mot }
    cout.close
  end

  trier = lambda do |cin, cout|
    # Channel definit each et inclut Enumerable
    #   => sort peut etre appele.
    cin.sort.each { |mot| cout << mot }
    cout.close
  end

  supprimer_doublons = lambda do |cin, cout|
    precedent = nil
    cin.each do |mot|
      cout << mot if mot != precedent
      precedent = mot
    end
    cout.close
  end

  (PRuby::Pipeline.source(fich_entree) |
   generer_mots |
   filtrer_mots_invalides |
   trier |
   supprimer_doublons |
   PRuby::Pipeline.sink(fich_sortie))
    .run
end
```

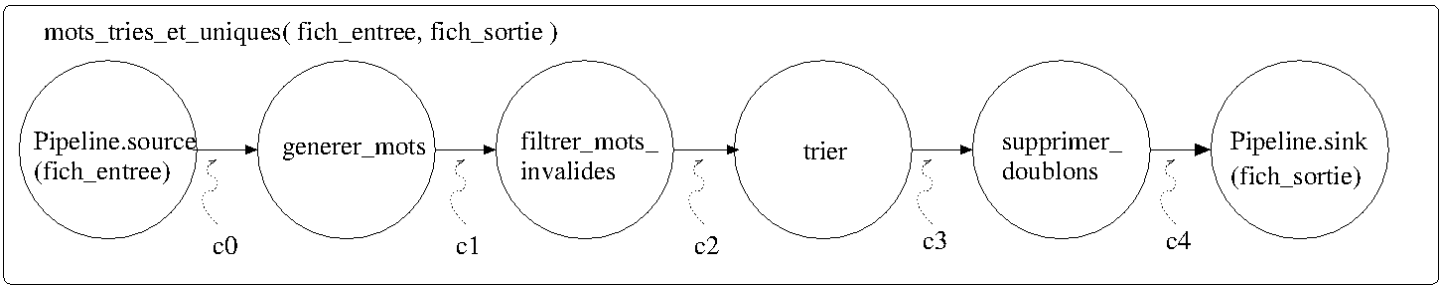


Figure 5.17: Graphe des processus et de leurs canaux pour les filtres et le pipeline du Programme Ruby 5.25.

Supposons un fichier d'entrée comme suit :

```
abc def
abc xx ! def
xx
```

Les éléments qui transiteront dans les canaux — ignore les `"\n"` :

```
c0: ['abc def', 'abc xx ! def', 'xx', EOS]

c1: ['abc', 'def', 'abc', 'xx', '!', 'def', 'xx', EOS]

c2: ['abc', 'def', 'abc', 'xx', 'def', 'xx', EOS]

c3: ['abc', 'abc', 'def', 'def', 'xx', 'xx', EOS]

c4: ['abc', 'def', 'xx', EOS]
```

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?

Exercice 5.18: Exécution parallèle, ou non, d'un pipeline.

5.6.4 Problème «de Jackson»

Certains problèmes, bien qu'ils semblent simples, peuvent être difficiles à résoudre à l'aide d'un programme séquentiel.

```
Entree:
-------
abc ** dsds cssa
ssdsx
fssfdfdfdfdfdf
s.s.**xtx*zy

Sortie:
-------
abc
^ ds
ds c
ssas
sdsx
fssf
dfdf
dfdf
dfs.
s.^x
tx*z
y
```

Figure 5.18: Exemple d'un fichier d'entrée et du fichier de sortie correspondant pour le problème de Jackson avec `n=4`.

Programme Ruby 5.26 Méthode pour le problème de Jackson, avec filtres et pipeline PRuby.

```ruby
def transformer_jackson( fich_donnees, fich_sortie, n )
  # Transforme flux de lignes en flux de caracteres.
  depaqueter = lambda do |cin, cout|
    cin.each do |ligne|
      ligne.each_char { |c| cout << c }
    end
    cout.close
  end

  changer_exposant = lambda do |cin, cout|
    cin.each do |c|
      (cin.get; c = "^") if c == "*" && cin.peek == "*"
      cout << c
    end
    cout.close
  end

  # Transforme flux de cars en flux de lignes de longueur n
  paqueter = lambda do |cin, cout|
    ligne = ""
    cin.each do |char|
      ligne << char
      (cout << ligne; ligne = "") if ligne.size == n
    end

    cout << ligne unless ligne.empty?
    cout.close
  end

  (PRuby::Pipeline.source(fich_donnees) |
   depaqueter |
   changer_exposant |
   paqueter |
   PRuby::Pipeline.sink(fich_sortie))
    .run
end
```

- (Object) **peek**

Lit l'element en tete du canal, mais sans le retirer du canal.

Returns:

- L'element en tete du canal. Bloque si empty?

Ensures:

- Aucun effe sur le contenu du canal

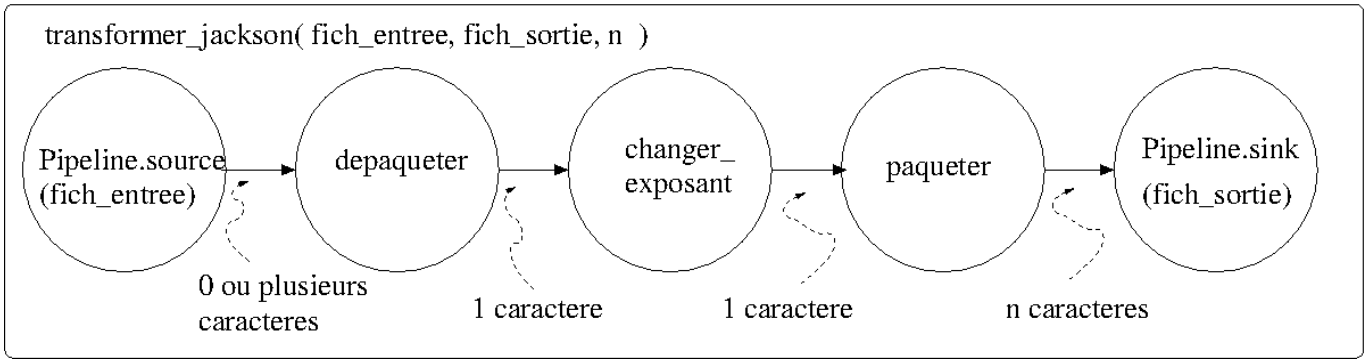


Figure 5.19: Graphe des processus et de leurs canaux pour les filtres et le pipeline du Programme Ruby 5.26.

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

Exercice 5.19: Version séquentielle de la transformation de Jackson.

5.6.5 Dénombrement des mots d'un texte avec filtres et pipelines

```
# Données à traiter
[
 "abc ghi def",
 "def abc",
 "abc"
]

# Résultat obtenu
[
 ["abc", 3],
 ["def", 2],
 ["ghi", 1]
]
```

Programme Ruby 5.27 Méthodes auxiliaires utilisées pour le dénombrement des mots d'un texte avec filtres et pipelines.

```ruby
def generer_mots
  lambda do |cin, cout|
    cin.each do |ligne|
      ligne.split( /\s+/ ).each { |mot| cout << mot }
    end
    cout.close
  end
end

def compter_occurrences
  lambda do |cin, cout|
    nb = 0 # Nb. d'occurrences vues jusqu'a present.
    cin.each do |mot|
      if mot == cin.peek
        nb += 1
      else
        cout << [mot, nb+1] # On ajoute le dernier mot vu.
        nb = 0
      end
    end
    cout.close
  end
end
```

Programme Ruby 5.28 La méthode `compter_mots` avec filtres et pipelines.

```
def compter_mots( lignes )
  resultat = []

  pipeline =
    PRuby::Pipeline.source( lignes ) |
    generer_mots |
    trier |
    compter_occurrences |
    PRuby::Pipeline.sink( resultat )
  pipeline.run

  resultat
end
```

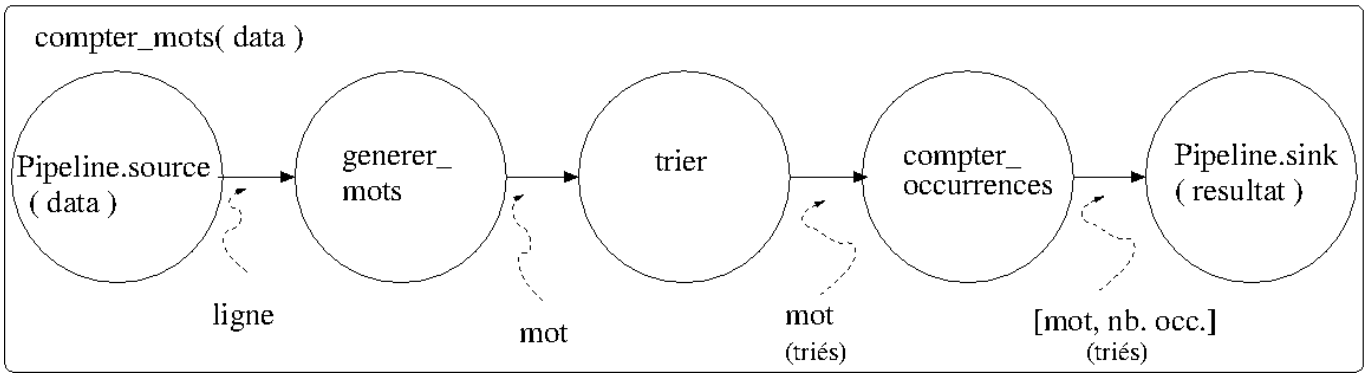


Figure 5.20: Graphe des processus et canaux pour le pipeline du Programme Ruby 5.28.

5.6.6 Pipelines avec canaux et processus explicites «à la Go»

Le langage Go

Figure 5.21: Figure tirée du site web du langage Go : `https://golang.org/`

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
```

Exercice 5.20: Programme `mystere` avec pipeline.

Exemple PRuby illustrant le style Go avec canaux et processus explicites

Programme Ruby 5.29 Un petit pipeline avec trois processus.

```ruby
# Les trois processus, qui seront organises en un pipeline lineaire:
#     ... | p1 | p2 | p3 | ...

p1 = lambda do |cin, cout|
  n = cin.get
  (1..n).each { |i| cout << i }
  cout.close
end

p2 = lambda do |cin, cout|
  cin.each { |v| cout << 10 * v }
  cout.close
end

p3 = lambda do |cin, cout|
  r = 0
  cin.each { |v| r += v }
  cout << r
  cout.close
end

# Creation des canaux.
c1, c2, c3, c4 = Array.new(4) { PRuby::Channel.new }

# Activation des processus.
p1.go( c1, c2 )
p2.go( c2, c3 )
p3.go( c3, c4 )

# Ecriture initiale dans le premier canal => amorce le flux des donnees
c1 << 10

# Reception du resultat du dernier canal.
puts c4.get # => 550
```

Dénombrement des mots d'un texte avec canaux et processus de style Go

Programme Ruby 5.30 La méthode `compter_mots` avec canaux explicites et processus de style Go.

```ruby
def compter_mots( lignes )
  # On cree les canaux requis.
  c1, c2, c3, c4 = Array.new(4) { PRuby::Channel.new }

  # On lances les processus.
  generer_mots.go(c1, c2)
  trier.go(c2, c3)
  compter_occurrences.go(c3, c4)

  # On transmet les donnees au 1er processus.
  lignes.each { |ligne| c1 << ligne }; c1.close

  # On obtient le resultat du dernier processus.
  c4.to_a
end
```

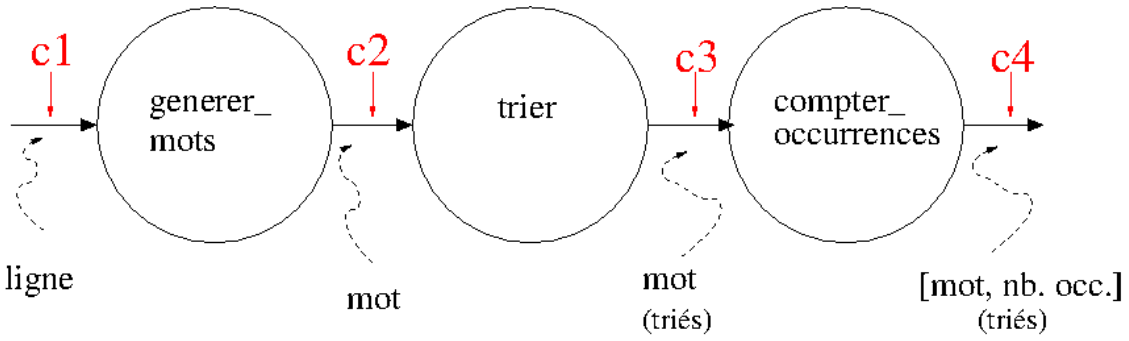


Figure 5.22: Graphe des processus Go et canaux, explicites, pour le pipeline du Programme Ruby 5.30.

5.7 Parallélisme de flux de données avec *streams* : `source`, `filter`, `map`, `group_by_key`, `sink`, etc.

Tri unique des mots d'un fichier

Programme Ruby 5.31 Méthode pour trier les mots d'un fichier, en s'assurant que chaque mot apparaît au plus une fois — version avec *streams*.

```
def trier_mots_uniques( fich_entree, fich_sortie )
  PRuby::Stream.source( fich_entree )
    .flat_map { |ligne| ligne.split( /\s+/ ) }
    .filter { |mot| /^\w+$/ =~ mot }
    .sort
    .uniq
    .sink( fich_sortie )
end
```

Exemple Ruby 5.2 Exemples pour illustrer les méthodes `flat_map`, `filter` `uniq` et `group_by_key` de la classe `PRuby::Stream`.

```ruby
# Exemple flat_map
a = [1, 0, 2, 0, 3, 4, 0]
r = [1, 1, 2, 2, 3, 3, 4, 4]

PRuby::Stream.source( a )
  .flat_map { |x| x == 0 ? [] : [x, x] }
  .to_a.must_equal r

# Exemple filter
a = [1, 2, 3, 4]
r = [2, 4]

PRuby::Stream.source( a )
  .filter { |x| x.even? }
  .to_a.must_equal r

# Exemple uniq.
a = [3, 1, 4, 2, 2, 3, 1]
r = [3, 1, 4, 2]

PRuby::Stream.source( a )
  .uniq
  .to_a.must_equal r

# Exemple group_by_key
a = [["abc", 10],
     ["abc", 22],
     ["def", 10],
     ["abc", 999],
     ["def", 222]]
r = [["abc", [10, 22, 999]],
     ["def", [10, 222]]]

PRuby::Stream.source( a )
  .group_by_key
  .to_a.must_equal r
```

Dénombrement des mots d'un texte avec des *streams*

Programme Ruby 5.32 La méthode `compter_mots` avec *streams*.

```
def compter_mots( lignes )
  PRuby::Stream.source( lignes )
    .flat_map { |ligne| ligne.split( " " ) }
    .map { |mot| [mot, 1] }
    .group_by_key
    .map { |mot, occs| [mot, occs.reduce(0, :+)] }
    .sort
    .to_a
end
```

```ruby
lignes = ["abc def ghi", "abc def", "abc"]

p PRuby::Stream.source( lignes )
    .flat_map { |ligne| ligne.split( /\s+/ ) }
    .peek { |mot| p "#{mot}" }
    .map { |mot| [mot, 1] }
    .peek { |k| p k }
    .group_by_key
    .peek { |k| p k }
    .map { |m, occs| [m, occs.reduce(0, :+)] }
    .sort
    .to_a

####################################

"abc"                                  # Debut 1er peek
"def"
"ghi"
"abc"
"def"
"abc"                                  # Fin 1er peek
["abc", 1]                             # Debut 2e peek
["def", 1]
["ghi", 1]
["abc", 1]
["def", 1]
["abc", 1]                              # Fin 2e peek
["abc", [1, 1, 1]]                      # Debut 3e peek
["def", [1, 1]]
["ghi", [1]]                           # Fin 3e peek
[["abc", 3], ["def", 2], ["ghi", 1]] # Resultat final
```

Exemple d'exécution 5.2: Exemples d'exécution d'une série d'opérations sur des *streams* illustrant l'effet de `group_by_key`.

5.A Sommaire : Comparaison de quelques approches pour la somme de deux tableaux

Programme Ruby 5.33 Méthode parallèle itérative *à granularité fine* pour faire la somme de deux tableaux avec `pcall`.

```
def somme_tableaux( a, b )
  c = Array.new(a.size)

  PRuby.pcall( 0...c.size,
               ->( k ) { c[k] = a[k] + b[k] }
               )

  c
end
```

Question : Avantages? Désavantages?

Programme Ruby 5.34 Méthode parallèle itérative pour faire la somme de deux tableaux avec `pcall`.

```ruby
# Les indices pour la tranche du thread no. k.
def indices_tranche( k, n, nb_threads )
  (k * n / nb_threads)..((k + 1) * n / nb_threads - 1)
end

# Somme sequentielle de la tranche pour indices (inclusif)
def somme_seq( a, b, c, indices )
  indices.each { |k| c[k] = a[k] + b[k] }
end

def somme_tableaux( a, b, nb_threads = PRuby.nb_threads )
  c = Array.new(a.size)

  PRuby.pcall( 0...nb_threads,
               lambda do |k|
                 somme_seq(
                   a, b, c,
                   indices_tranche(k, c.size, nb_threads)
                 )
               end
               )

  c
end
```

Question : Avantages? Désavantages?

Programme Ruby 5.35 Méthode parallèle itérative pour faire la somme de deux tableaux avec `pcall`.

```
def somme_seq_cyclique( a, b, c, num_thread, nb_threads )
  (num_thread...a.size).step(nb_threads).each do |k|
    c[k] = a[k] + b[k]
  end
end

def somme_tableaux( a, b, nb_threads = PRuby.nb_threads )
  c = Array.new( a.size )

  PRuby.pcall( 0...nb_threads,
               lambda do |num_thread|
                 somme_seq_cyclique(
                   a, b, c,
                   num_thread, nb_threads
                 )
               end
               )

  c
end
```

Question : Avantages? Désavantages?

Programme Ruby 5.36 Méthode parallèle itérative *à granularité grossière* pour faire la somme de deux tableaux avec `peach`.

```
def somme_tableaux( a, b, nb_threads = PRuby.nb_threads )
  c = Array.new(a.size)

  (0...c.size).peach( nb_threads: nb_threads ) do |k|
    c[k] = a[k] + b[k]
  end

  c
end
```

Question : Avantages? Désavantages?

Programme Ruby 5.37 Méthode parallèle pour effectuer la somme de deux tableaux avec `pmap`.

```ruby
def somme_tableaux( a, b, nb_threads = PRuby.nb_threads )
  (0...a.size).pmap( nb_threads: nb_threads ) do |k|
    a[k] + b[k]
  end
end
```

Question : Avantages? Désavantages?

Programme Ruby 5.38 Méthode parallèle pour effectuer la somme de deux tableaux avec `pmap`.

```
def somme_tableaux( a, b, nb_threads = PRuby.nb_threads )
  (0...a.size).pmap( nb_threads: nb_threads, dynamic: true )
    a[k] + b[k]
  end
end
```

Question : Avantages? Désavantages?