# Table des matières

# Chapitre 5

# Patrons de programmation parallèle avec PRuby

## 5.1 La bibliothèque `PRuby`

Ce chapitre présente cinq principaux patrons de base de la 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** — ces «constructions» pour écrire des programmes parallèles — sont représentatifs de ce qu'on retrouve dans divers langages de programmation modernes.

Dans le présent chapitre, ces patrons sont exprimés en `PRuby`. `PRuby` est une bibliothèque — un ***gem*** — pour la programmation parallèle en Ruby.

Un point important à signaler est que la bibliothèque `PRuby` n'a pas été conçue pour la programmation parallèle **haute performance** ☹ En fait, Ruby n'est pas lui-même un langage qui vise les performances du code ; Ruby vise plutôt à améliorer… les performances de la personne qui écrit le programme ☺ En ce sens, la bibliothèque `PRuby` peut donc être vue comme une forme de pseudocode parallèle exécutable permettant d'exprimer les principales approches de programmation parallèle.

Dans d'autres chapitres, on verra d'autres langages conçus spécifiquement pour la programmation parallèle.

## 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`*

Dans un programme utilisant le parallélisme de type «*fork–join*», le programme débute son exécution avec un unique *thread* — parfois appelé le «*thread* maître». Ensuite, ce premier *thread* crée *explicitement* de nouveaux *threads* — avec `fork` ou une instruction équivalente. Puis, si nécessaire, le *thread* «parent» attend que les *threads* enfants aient terminé avant de poursuivre son exécution. De plus, les *threads* «enfants» peuvent eux aussi créer de nouveaux *threads* — et aussi attendre que leurs «enfants» terminent.

Ce modèle tire son nom de la façon dont on crée un processus sur Unix — avec un appel système `fork` — et de la façon dont attend la fin d'un *thread* dans la bibliothèque de *threads* Posix — `pthread_join`. Ces termes ont ensuite été repris, parfois tels quels, parfois modifiés, dans de nombreux autres langages de programmation.

Comme on le verra, dans certains langages, tant le `fork` que le `join` — ou des opération équivalentes — sont *explicites* — par ex., avec les *threads* de base Posix, Java ou Ruby — ce qui permet de créer des structures *non imbriquées* d'appels de *threads*. Dans d'autres langages, par contre, l'équivalent du `fork` est explicite mais l'opération `join` est *implicite*, ce qui ne permet que la création *imbriquée* de *threads*. Dans ce dernier cas, on parle parfois d'une approche avec `cobegin/coend`. C'est ce style que nous allons voir en premier avec la méthode `pcall` de la bibliothèque `PRuby`.

L'approche *fork-join* est un mécanisme de base 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

Comme premier exemple, nous allons définir dDifférentes versions d'une méthode pour calculer la factorielle d'un nombre `n` — `fact( n ) =` $1\times2\times3\times\ldots\times(n-1)\times$`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
```

Le programme Ruby 5.1 présente une première version *récursive séquentielle et linéaire*. Ici, linéaire signifie qu'un seul appel récursif est effectué, *ce qui ne permet donc aucun parallélisme* ☹

| Peut-on extraire du parallélisme de cet algorithme, tel que formulé?<br>(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
```

Le programme Ruby 5.2 présente une version *récursive séquentielle et dichotomique* : dans le cas récursifs, *deux (2)* appels récursifs à `fact` sont effectués. Ces appels étant *indépendants l'un de l'autre, ils peuvent donc être exécutés en parallèle* ☺

On remarque qu'une méthode auxiliaire est introduite — `fact_`. Cette méthode assure que les sous-problèmes *sont similaires au problème initial* — pour que les appels récursifs puissent avoir des arguments équivalents à l'appel de la méthode : voir Annexe **??**.

## 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
```

Le Programme Ruby 5.3 présente une version récursive parallèle avec `pcall`. Quelques remarques :

- Plutôt qu'avoir un `if` avec une branche `then` *et* une branche `else` complexe imbriquée et indentée, on retourne immédiatement et directement le résultat approprié dans le cas de base : «`return i if i == j`».

- Les deux appels récursifs, au lieu d'être faits l'un après l'autre, sont faits *en parallèle*, et ce en utilisant la méthode `pcall`.

  Plus précisément, la méthode `pcall` de la bibliothèque `PRuby` — méthode de classe (statique) — reçoit en argument deux ou plusieurs `lambdas`. Ces différents lambdas sont alors évalués de façon concurrente — en parallèle si les ressources le permettent. *De plus, l'appel à* `pcall` *ne se termine que lorsque tous les appels des lambdas ont complété.* Un appel à la méthode `pcall` est donc *bloquant* pour le *thread* appelant et introduit une forme de *barrière de synchronisation* entre les *threads* exécutant les lambdas.

- Avant de faire les appels récursifs, on affecte les deux variables `r1` et `r2` à `nil` — on aurait aussi pu écrire «`r1 = r2 = nil`». Ces affectations sont

nécessaires pour que les variables `r1` et `r2` indiquées dans les `lambda` soient déjà déclarées et visibles, et donc puissent être modifiées pour obtenir les résultats des appels récursifs : dans un `lambda`, si une variable non-locale existe dans l'environnement, alors c'est cette variable qui est utilisée.

- Une expression `lambda` peut aussi être dénotée par le symbole «`->`». De plus, dans un appel de méthode, les parenthèses, à moins qu'il n'y ait ambiguité à cause de la précédence des opérateurs, peuvent être omises. L'appel à `pcall` aurait donc pu être écrite de différentes façons comme suit — le caractère «\» indique une continuation de l'instruction à la ligne suivante :

```
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) }

...
```

- La méthode `pcall` peut être vue comme une forme de `cobegin/codend`. Dans un langage comme MPD, on aurait écrit quelque chose comme suit — `co` = `cobegin` = `fork` et `oc` = `coend` = `join` — donc avec une syntaxe différente du `pcall` mais avec un comportement semblable :

```
co
    r1 = fact_( i, mid )
//  r2 = fact_( mid+1, j )
oc
```

## 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
```

Le Programme Ruby 5.4 présente une version légèrement modifiée de la méthode précédente. Quelques remarques :

- Dans une approche diviser-pour-régner, lorsque le problème est suffisamment «simple», on le résout directement — le «cas de base». Dans les exemples vus jusqu'à présent, on considérait qu'un problème était simple lorsqu'il était *trivial* — de taille 1, i.e., `i == j`.
- Effectuer des appels récursifs entrainent certains coûts. Ces coûts sont encore plus élevés lorsque ces appels récursifs sont effectués en parallèle, avec des *threads* ☹ (voir séance de laboratoire).
- Une façon intéressante, et relativement facile, d'améliorer les performances d'un algorithme récursif est de terminer la récursion lorsque le problème est suffisamment simple *mais sans être trivial*. C'est ce qui est fait ici : lorsque le nombre dééléments à multiplier est suffisamment petit — `j - i <= seuil`

```
>> [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`).

— alors on effectue le produit `i * (i+1) * ... * (j-1) * j` avec `reduce`, donc sans faire d'appels récursifs additionnels. Voir Figure 5.1 pour des exemples illustrant la méthode `reduce`.

| 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 présente une autre version du calcul de factoriel avec un seuil de récursion :

- Dans le Programme Ruby 5.4 (idem pour le Programme Ruby 5.3), pour que les *threads* enfants puissent retourner un résultat au *thread* parent, le parent doit déclarer une variable appropriée, laquelle est ensuite utilisée comme variable non-locale du `lambda`. Or, ceci complexifie inutilement le lancement d'un *thread* exécuté pour évaluer une expression et retourner un résultat — par opposition à un *thread* exécuté pour ses effets de bord. L'utilisation de

`future` permet de simplifier ce cas où un *thread* est utilisé pour évaluer une expression, ce qui permet de rendre la version parallèle semblable à la version séquentielle.

- Un appel à la méthode `PRuby.future` reçoit *un (1) bloc* — ou *un (1)* `lambda` — qui sera évalué en parallèle. Contrairement à un appel à `pcall`, on ne spécifie qu'un seul segment de code à évaluer en parallèle. De plus, l'appel à `future` *ne bloque pas*, mais retourne (quasi)immédiatement un `future` (un objet de classe `PRubyFuture`). Le *thread* appelant peut donc poursuivre son exécution. Lorsqu'il a besoin du résultat associé au `future`, il doit alors faire appel à la méthode `value`. C'est cette dernière qui est possiblement *bloquante* : si le bloc a terminé son exécution et a produit son résultat, alors l'appel à `value` retourne immédiatement cette valeur, sans bloquer; par contre, si le résultat n'est pas encore disponible, alors l'appel à `value` bloque jusqu'à ce que le `future` ait terminé et retourné son résultat.

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`.

### 5.2.2 Somme de deux tableaux

Dans cet exemple, nous allons définir une méthode pour effectuer la somme de deux tableaux — deux vecteurs, deux `Array`s. Le Programme Ruby 5.6 présente une version séquentielle et itérative d'une telle méthode. La méthode reçoit en arguments les tableaux `a` et `b` et retourne un tableau `c` qui représente leur somme, élément par élément (*element-wise*).

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

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

  c = Array.new(a.size)

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

  c
end
```

Quelques remarques :

- Une précondition a été indiquée pour assurer que les deux tableaux sont de même taille. Dans les versions qui suivent de cette méthode, cette précondition sera omise pour simplifier le code présenté.
- Les index d'un tableau `a` vont de `0` à `a.size-1` inclusivement. Ce sont donc sur ces index que l'on itère, avec `each` : rappelons que le `Range` `0...k` (borne exclusive) dénote les mêmes éléments que `0..k-1` (borne inclusive).

**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
```

Le Programme Ruby 5.7 présente une version parallèle utilisant `pcall`. La sommation pour chacun des éléments peut se faire de façon totalement indépendante — il s'agit d'une forme de parallélisme dite «*embarrassingly parallel*». Il est donc possible de créer un *thread* distinct pour chacun des éléments, et c'est ce qui est fait ici. Dans ce cas, on parle aussi de parallélisme à *granularité (très!) fine*, puisque chaque *thread* n'effectue qu'un tout petit calcul, exécute un tout petit nombre d'instructions. La Figure 5.2 illustre le graphe de dépendances des tâches pour des tableaux de taille `n` (variante d'un graphe vu précédemment).

Remarques additionnelles :

- L'instruction `pcall` reçoit ici deux (2) arguments :
  - Le premier argument est un `Range`, donc un intervalle d'index.
  - Le deuxième argument est une lambda-expression avec un (1) argument, qui sera une des valeurs du `Range`.
- Une telle instruction `pcall` crée *une famille de* threads, indexée par les éléments du `Range`. Avec `pcall`, on peut donc lancer l'exécution de plusieurs *threads*, soit en spécifiant deux ou plusieurs lambda *sans argument*, soit en spécifiant un `Range` et une lambda *avec un argument (entier)*.

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

**Exercice 5.5:** Performances si deux gros tableaux?

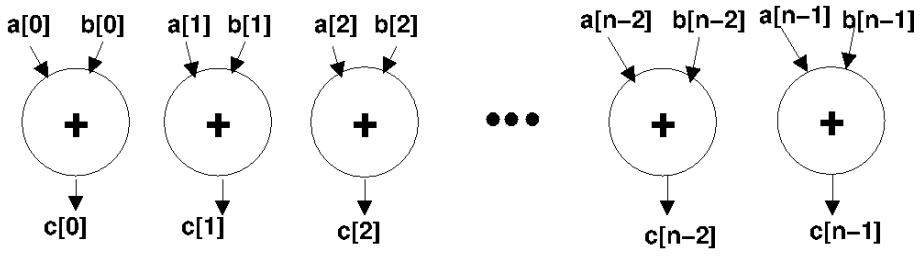



Figure 5.2: Graphe de dépendances des tâches pour le calcul parallèle de la somme de deux tableaux.

**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
```

Créer des *threads* coûte cher et en créer un pour chaque élément d'un tableau ne sera assurément pas performant. Ainsi, si l'on doit additionner 1000 éléments sur une machine avec 4 processeurs, on créera donc 1000 *threads*, dont l'exécution devra ensuite être répartie entre les 4 processeurs. Beaucoup de travail pour rien!

Pour obtenir un programme performant, il est généralement préférable d'utiliser un nombre de *threads* qui soit du même ordre de grandeur que le nombre de processeurs. Ici, cela signifie que chaque *thread* devrait prendre en charge la somme de *plusieurs* éléments, et non pas un seul.

Le Programme Ruby 5.8 présente une telle méthode, dite «*à granularité grossière*» :

- Un argument additionnel, mais optionnel, a été ajouté à la méthode `somme_tableaux`, argument permettant de spécifier combien de *threads* doivent être utilisés. Par défaut, on utilise comme nombre de *threads* la valeur `PRuby.nb_threads`, qui est égale au nombre de processeurs (ou coeurs) de la machine.

- Chaque *thread* est responsable de calculer une *tranche d'éléments adjacents*, calcul qui se fait de façon séquentielle et itérative dans la méthode `somme_seq`.

- Les indices associés à une tranche pour un *thread* sont obtenus avec la méthode `indices_tranche`, qui retourne un `Range`. Par exemple, si on a 100 éléments (`a.size == 100`) et 4 *threads* (`nb_threads == 4`), alors les tranches des différents *threads* seront réparties comme suit :

  - *thread* 0 : `0..24`
  - *thread* 1 : `25..49`
  - *thread* 2 : `50..74`
  - *thread* 3 : `76..99`

  Pour que cette façon simple de déterminer les indices à traiter fonctionne correctement, il faut que le nombre d'eléments à traiter soit un multiple du nombre *threads*. Cette condition est vérifiée dans la précondition. On verra ultérieurement comment calculer ces intervalles d'indices sans imposer cette condition. On verra aussi d'autres façons de répartir les éléments à traiter entre les divers *threads*.

  Dans le cas plus général, donc avec $k$ *threads*et $n$ éléments, voir la figure 5.3 pour la répartition résultante.

- La stratégie utilisée ici consiste donc à *répartir* un grand nombre d'éléments à traiter entre un petit nombre de *threads* en groupant ces éléments par *blocs* (*tranches*) d'éléments adjacents. Cette stratégie est fréquemment utilisée. On verra plus loin (section 5.3.1) que `PRuby` permet de l'exprimer de façon *beaucoup plus simple*.

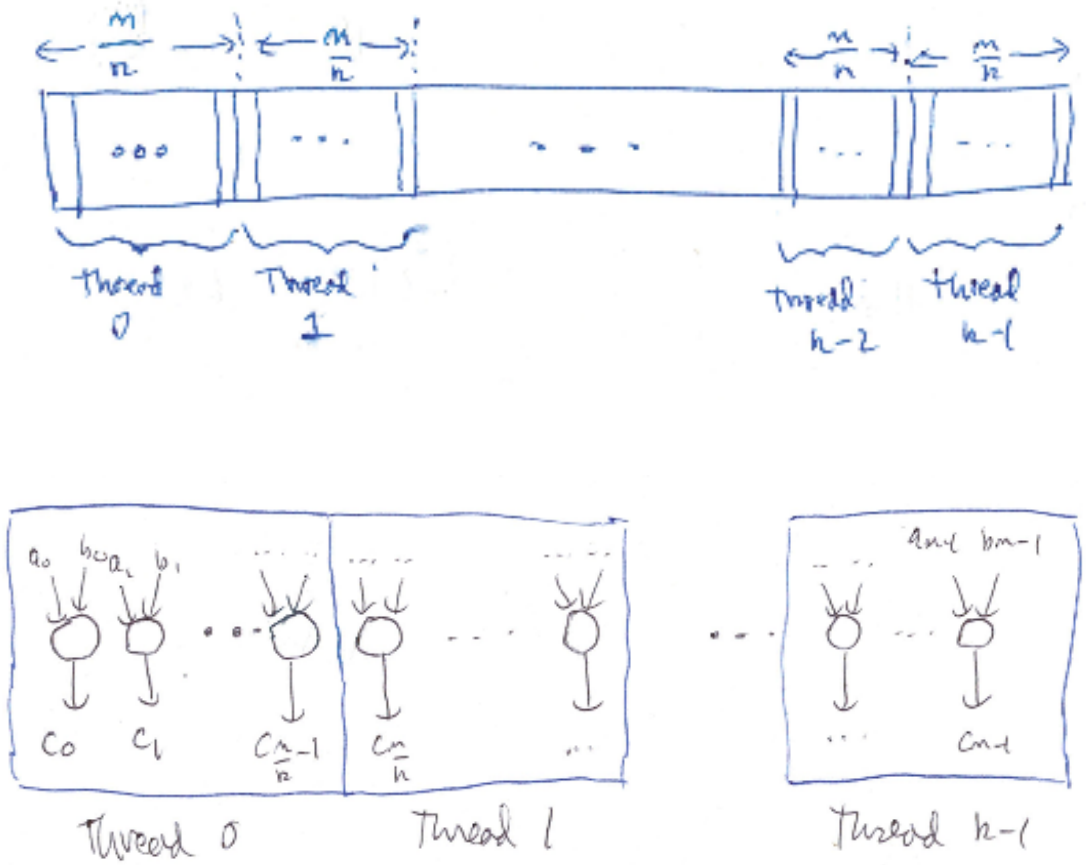



Figure 5.3: Répartition par blocs d'éléments adjacent de `n` éléments entre `k` *threads*. Chaque *thread* traite `n/k` éléments adjacents — pour simplifier, on suppose que `n` est divisible par `k`.La figure du haut présente une distribution d'un tableau, alors que la figure du bas présente le graphe de dépendances des tâches pour la somme de deux tableaux mais où un bloc de tâches est attribué à chacun des *threads*.

### 5.2.3 Calcul de $\pi$ à l'aide d'une méthode Monte Carlo

Un problème avec «parallélisme embarrassant», tel que celui de la somme de deux tableaux, peut être décomposé en un ensemble de tâches qui sont complètement indépendantes.

Pour d'autres problèmes, dits avec «**parallélisme semi-embarrassant**», le problème peut aussi être décomposé en un grand nombre de tâches qui sont indépendantes... **mais pas tout à fait** complètement : les tâches interagissent, mais de façon limitée, et souvent uniquement **vers la fin de l'exécution** des tâches.

Un problème relativement simple ayant cette propriéte est celui visant à estimer la valeur de $\pi$ à l'aide d'une méthode Monte Carlo. Cette méthode consiste à effectuer des essais «au hasard» — donc à l'aide de nombres pseudo-aléatoires — dans le but de trouver une solution au problème, solution parfois approximative mais «pas trop mauvaise».

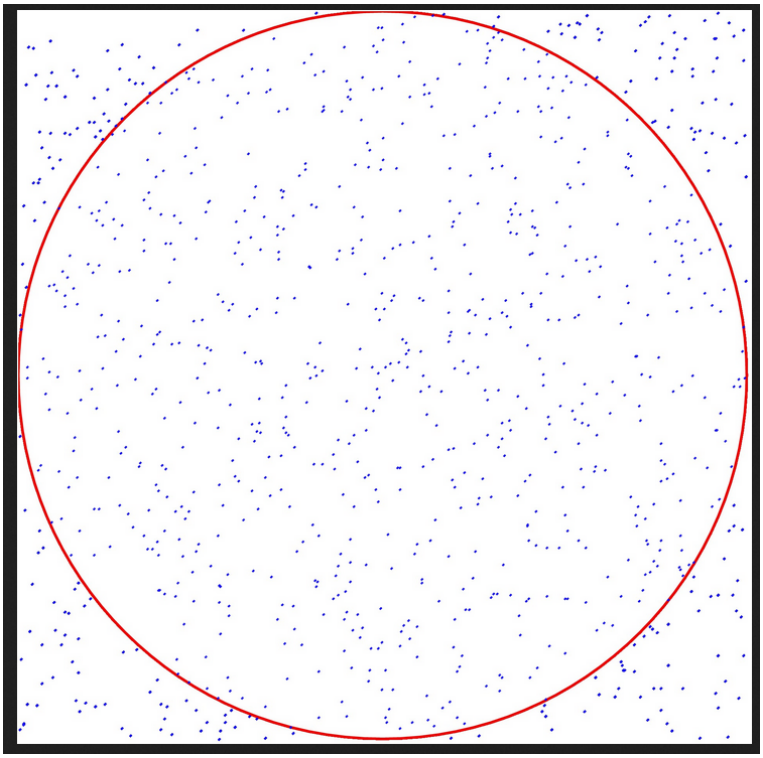


Figure 5.4: Estimation de la valeur de $\pi$ à l'aide d'une méthode Monte Carlo (source : `http://i.stack.imgur.com/uYrT5.jpg`).

La figure 5.4 illustre comment on peut estimer la valeur de $\pi$ à l'aide d'une telle méthode. Supposons qu'on ait une cible carrée de taille 2 par 2 dans laquelle est

inscrit un cercle de rayon 1.

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

- Aire du cercle de 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 $\pi$ : $\pi = 4\frac{A_{cercle}}{A_{carré}}$

Comment peut-on alors estimer la valeur de $\pi$? Supposons «qu'on lance *au hasard*» des flèchettes sur cette cible carrée. Si le nombre de fléchettes lancées est assez grand, alors on pourra estimer la valeur de $\pi$ comme suit :

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

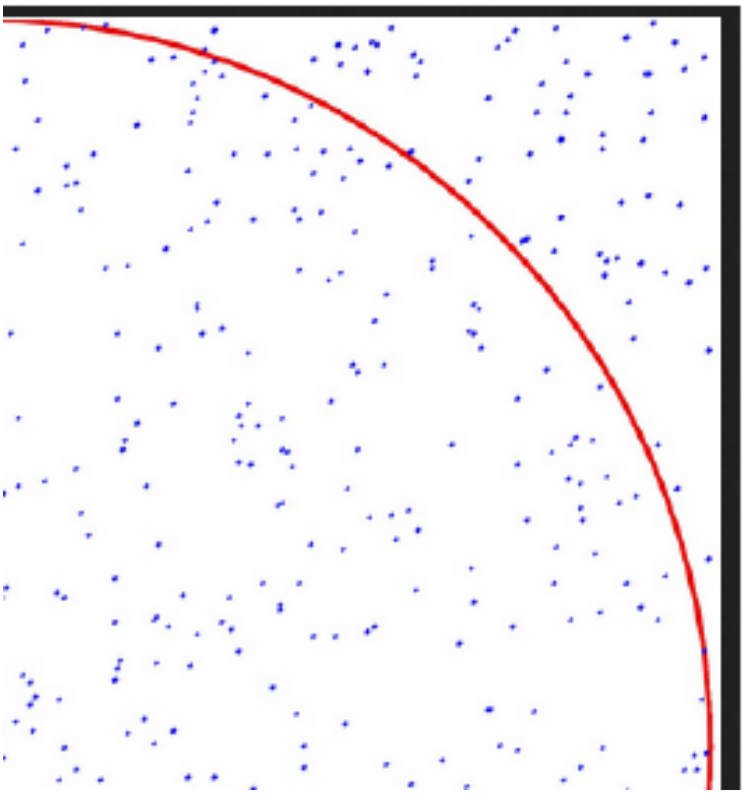


Figure 5.5: Estimation de la valeur de $\pi$ à l'aide d'une méthode Monte Carlo : La méthode présentée travaille exclusivement sur le quadrant supérieur droit.

```
>>  [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 5.9** Méthode parallèle pour estimer la valeur de $\pi$ 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_threads) }
  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
```

Le Programme Ruby 5.9 présente une mise en oeuvre parallèle avec des `futures` de cette approche, qui travaille uniquement dans le cadra supérieur droit, tel qu'illustré dans la Figure 5.5 :

- On crée `nb_threads` *threads*, et ce en créant un `future` pour chaque *thread*. Après l'exécution du premier `map`, `futures_nb_dans_cercle` est donc un tableau de `futures` — très probablement encore en cours d'exécution si le nombre de lancers est grand.
- Chaque *thread* simule le lancement de `nb_lancers/nb_threads` lancers — donc on répartit uniformément entre les *threads*les lancers à effectuer.

- Chaque *thread* — une instance de `nb_dans_cercle_seq` — retourne `nb`, le nombre de lancers effectués qui ont abouti dans le cercle.
- La méthode principale appelle la méthode `value` sur chaque `future` — donc peut bloquer en attente d'un résultat — puis effectue la somme de ces valeurs à l'aide de `reduce`.

**Programme Ruby 5.10** Méthode parallèle pour estimer la valeur de $\pi$ 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
```

Pour ceux plus familiers avec une approche impérative, le Programme Ruby 5.10 présente une version équivalente, mais dans un style plus «impératif». Soulignons toutefois qu'en Ruby, c'est le Programme Ruby 5.9 qui est considéré comme ayant «le meilleur style».

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`

Dans un langage impératif «classique» comme le langage C qui manipule principalement des tableaux, la structure de contrôle de base utilisée pour traiter ces tableaux est la boucle, plus spécifiquement la boucle `for`.

Une boucle `for` est dite «boucle définie», car on connaît le nombre d'itérations lors du lancement de la boucle — contrairement à une boucle `while`, dite «boucle indéfinie», où le nombre d'itérations dépend d'une condition déterminée en cours d'exécution.

Si les différentes itérations de la boucle `for` sont **indépendantes** les unes des autres — notamment, n'écrivent pas dans les mêmes variables — alors **elles peuvent être exécutées en parallèle.**

Pour ce faire, de nombreux langages parallèles — dont diverses variantes ou extensions de C, par ex., OpenMP/C — introduisent des boucles parallèles, sous différents noms : `parallel_for`, `forall`, `foreach`, etc.

En `PRuby`, deux méthodes sont disponibles pour exprimer le parallélisme de boucle, `peach` et `peach_index`, des variantes parallèles de `each` et `each_index`.

Avant de voir des exemples d'utilisation des versions parallèles, rappelons la différence entre `each` et `each_index` dans le cas des classes `Array` et `Range`, les deux classes pour lesquelles les variantes parallèles sont disponibles en PRuby :

- L'itérateur `each` reçoit chacun des éléments d'une collection et applique un bloc à chaque élément reçu.
- L'itérateur `each_index` reçoit chacun des indices des éléments d'une collection, et applique un bloc à chaque indice reçu. Cette méthode n'est toutefois valide que pour un objet `Array`, et non pour un `Range`.

Dans les deux cas, la valeur retournée par l'appel à la méthode est la collection elle-même, i.e., celle sur laquelle s'applique la méthode `each` ou `each_index`. L'exemple Ruby 5.1 présente quelques exemples simples.

**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
```

Le Programmme Ruby 5.11 présente une version avec parallélisme de boucle **à granularité fine** pour la somme de deux tableaux.

**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
```

Le Programmme Ruby 5.12 quant à lui présente une version avec parallélisme de boucle **à granularité grossière** pour la somme de deux tableaux. Comme on le constate, c'est une version très simple, semblable à la version séquentielle, sauf pour la spécification du nombre de *threads* à utiliser. De plus, cette version est tout à fait équivalente *à celle du Programme Ruby 5.8* quant à la répartition des éléments entre les *threads*, et ce même si elle est beaucoup plus simple. Quelques explications :

- La méthode `peach` peut recevoir divers arguments, tous optionnels, arguments qui sont spécifiés par mot-clé (*keyword arguments*) :

  - On peut spécifier le nombre de *threads* à utiliser pour exécuter les diverses itérations de la boucle.

    Par défaut, si aucune valeur n'est spécifiée, on utilise `Pruby.nb_threads` *threads* (nombre de coeurs ou processeurs de la machine), donc :

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

- Les *threads* vont se partager les diverses itérations de la boucle. Ces itérations peuvent être distribuées **de différentes façons** :

  - Par défaut — ou lorsqu'on indique simplement «`static: true`» — la répartition se fait par groupes d'éléments adjacents, comme dans le Programme Ruby 5.8 :

    ```
    col.peach { ... }
    =
    col.peach( static: true ) { ... }
    ```

  Ici, on parle d'une répartition *statique* parce qu'on sait, *dès le lancement des* threads, quelles seront les itérations traitées par chaque *thread*.

  Signalons que contrairement au Programme Ruby 5.8, le nombre d'éléments à traiter n'a pas besoin d'être divisible par le nombre de *threads*. Si le nombre d'éléments n'est pas exactement divisible, les éléments seront répartis le plus uniformément possible entre les *threads*. Par exemple, si on doit répartir 20 éléments entre 6 *threads*, les 2 premiers *threads* traiteront 4 éléments alors que les 4 *threads* suivants en traiteront 3.

- Lorsqu' une valeur numérique est spécifiée, par exemple «`static: k`», on a alors une répartition statique et cyclique par bloc de `k` éléments.

  Par exemple, si on a 12 éléments à répartir entre 3 *threads*, on aurait donc les associations suivantes — $t_i$ dans une position du tableau indique que c'est le *thread* $i$ qui traite cet élément :

  - `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* des éléments. On verra des exemples plus loin.

### 5.3.2 Calcul de $\pi$ à l'aide d'une méthode Monte Carlo

**Programme Ruby 5.13** Méthode parallèle pour estimer la valeur de $\pi$ 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
```

Le Programme Ruby 5.13 présente une autre version du calcul de $\pi$, cette fois en utilisant un `peach`. Solution simple, n'est-ce pas? Mais on verra bientôt qu'il est possible de faire *encore plus simple*!

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
```

Le Programme PRuby 5.14 présente une méthode séquentielle itérative pour calculer le produit de deux matrices. Ces matrices sont des objets de la classe `Matrice`. Voir l'URL suivant pour une description de l'API de cette classe :

`http://www.labunix.uqam.ca/~tremblay/INF5171/pruby/Matrice.html`

On remarque qu'une précondition sur la taille des matrices à multiplier a été spécifiée. La même condition s'applique évidemment aux versions parallèles, même si cette condition ne sera plus indiquée explicitement.

Le Programme PRuby 5.15 présente une version parallèle *à granularité très fine* du produit de matrices. Chaque élément du résultat — matrice `c` — peut être calculé de façon indépendante, donc avec un algorithme avec parallélisme embarrassant. Dans cette version, on lance donc un *thread* **pour chaque élément du résultat**! Clairement cette solution risque de ne pas être efficace, car un très (trop!?) grand nombre de *threads* seront lancés.

**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.

Le Programme PRuby 5.16 présente une autre version du produit de matrices, cette fois, avec un *thread* pour chaque ligne.

**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?

Finalement, le Programme PRuby 5.17 présente une autre version du produit de matrices parallèle, cette fois *à granularité grossière* :

- Un seul `peach` est utilisé, au niveau des lignes. Pour le traitement des diverses colonnes d'une ligne, on utilise plutôt un `each`, donc un traitement itératif séquentiel.
- Aucun argument n'est spécifié au `peach`, donc :
  - On utilisera un nombre de *threads* égal à `PRuby.nb_threads`, soit le nombre de processeurs de la machine.
  - On utilisera une répartition *statique adjacente* des éléments entre ces *threads* — par groupe d'éléments adjacents, comme vu précédemment.

En d'autres mots, chaque *thread* calculera *un bloc de lignes de la matrice résultat* `c`, donc la granularité des *threads* sera plus *grossière*.

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

```ruby
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?

## Modes de répartition des matrices entre *threads*

Certains langages de programmation — par exemple, Fortran 90, HPF (*High Performance Fortran*) — permettent d'exprimer *directement et explicitement* le mode de répartition des tableaux entre les *threads* ou les processus.

Les figures 5.7 et les figures 5.8–5.9 illustrent différents modes de répartition d'un tableau à une dimension (figure 5.7) et à deux dimensions (figures 5.8 et 5.9).


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`).

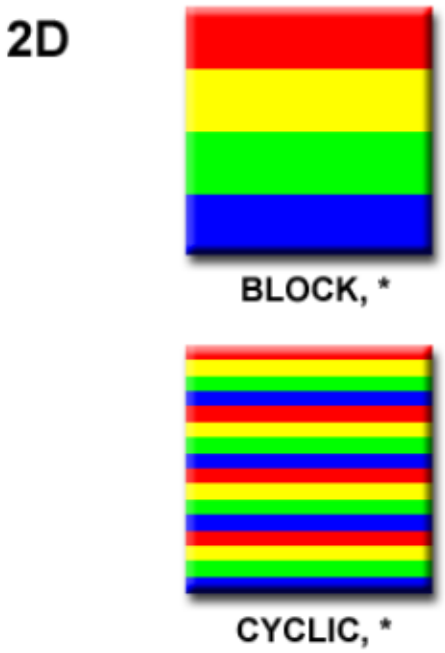



Figure 5.8: Distribution par bloc vs. distribution cyclique pour une matrice à 2 dimensions (source : `http://www.dais.unive.it/~calpar/New_HPC_course/5_Parallel_Patterns.pdf`).

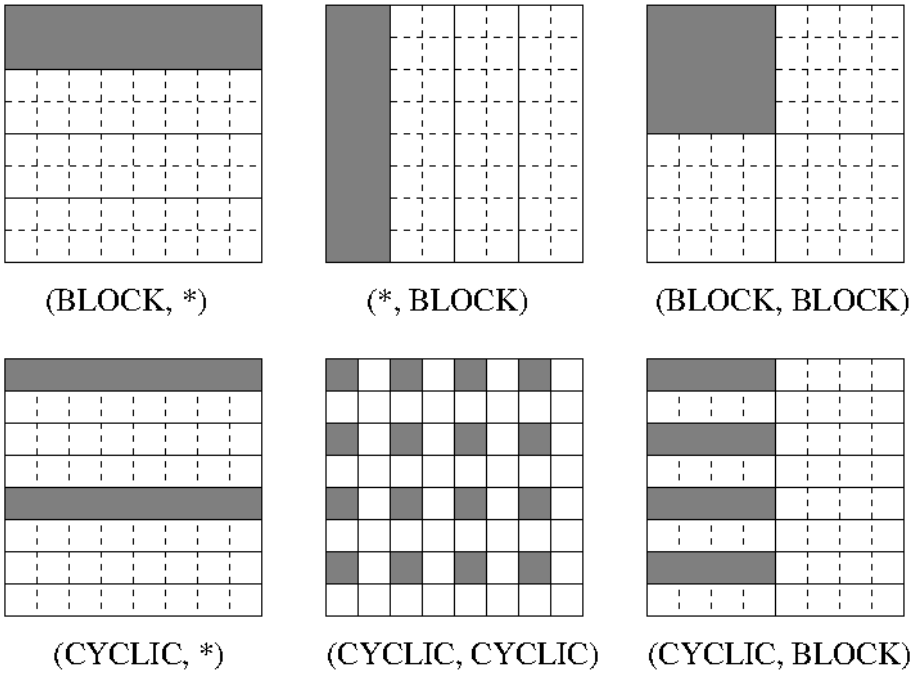



Figure 5.9: Différents types de distribution des données pour un tableau $8 \times 8$ entre quatre (4) processus en HPF. Les données pour le processus 0 sont en gris foncé (source : [Fos95]). Sur la première ligne : différents modes de répartition par blocs — dont blocs de lignes (gauche) et blocs de colonnes (centre). Sur la deuxième ligne : différents modes de répartition cyclique.

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

### 5.4.1 Parallélisme de données

Le **parallélisme de données** correspond à l'application d'une même opération *sur tous les éléments d'une collection* — collection généralement (mais pas toujours) *homogène*, c'est-à-dire dont les éléments sont tous du même type.

Habituellement, ces collections de données sont des listes (des séquences) — dans les langages fonctionnels — ou des tableaux — dans les langages impératifs plus «*mainstream*». Dans les deux cas, il s'agit de structures de données *régulières* et linéaires — par opposition aux structures de données *dynamiques* avec des pointeurs arbitraires qui donnent lieu à des arbres ou graphes (non-linéaires).

### 5.4.2 Application, réduction et préfixes

Un programme écrit dans le style «parallélisme de données» est généralement composé d'une séquence d'applications d'opérations sur des collections. Les trois principales opérations sur les collections sont les suivantes :

- *Application* : une telle opération consiste à appliquer une méthode sur chacun des éléments d'une collection pour obtenir un résultat qui est **une nouvelle collection**, de taille identique. Dans la terminologie des langages fonctionnels, on parle alors d'une opération de type `map` — certains auteurs parlent aussi d'un schéma de type *$\alpha$-notation* ou d'une *$\alpha$-application* [GUD96].

  Plus spécifiquement, soit $f$ une opération unaire (avec un seul argument) et $C = [c_1, \ldots, c_n]$ une collection de taille $n$. L'application de $f$ à la collection $C$ produira alors la collection $R$ (aussi de taille $n$) définie comme suit :

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

  Il est aussi possible de généraliser à des opérations $k$-aires — c'est-à-dire avec $k$ arguments. Soit les $k$ collections $C^1, \ldots, C^k$ toutes de taille $n$. L'application de $g$, une méthode de $k$ arguments, sur ces $k$ collections produit alors la collection $R$ de taille $n$ satisfaisant la propriété suivante :

  $$R = [g(c_1^1, \ldots, c_1^k), \ldots, g(c_n^1, \ldots, c_n^k)]$$

- *Réduction* : une telle opération consiste à appliquer une méthode *binaire* (à deux arguments) sur les divers éléments de la collection pour obtenir un résultat qui est **une valeur généralement d'un type plus simple** — par

exemple, la réduction d'une collection de nombres avec l'opérateur d'addition produirait un nombre en résultat. Dans la terminologie des langages fonctionnels, on parle alors d'un opération de type `fold` ou `reduce` — certains auteurs parlent plutôt de $\beta$-réduction [GUD96].

Par exemple, soit $\oplus$ une opération binaire et $C = [c_1, \ldots, c_n]$ une collection. L'application de $\oplus$ sur $C$ via une $\beta$-réduction produit alors le résultat $r$ définie comme suit :

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

- *Calcul de préfixes* : une telle opération applique une opération binaire associative sur les divers éléments de la collection pour obtenir un résultat qui sera une autre collection de même taille, où le $i^{ème}$ éléments de la collection résultante est obtenu par une série d'applications de l'opération binaire sur les $i$ premiers éléments. Plus précisément, soit $\oplus$ une opération binaire et $C = [c_1, \ldots, c_n]$ une collection. Le calcul de préfixes sur $C$ avec $\oplus$ produira alors le résultat $R$ satisfaisant la condition suivante, donc tel que $R_i = c_1 \oplus c_2 \oplus \ldots \oplus c_i$ :

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

Les notions d'application avec `map` et de réduction avec `reduce` ne sont pas nouvelles. Le premier langage à introduire de telles opérations fut Lisp, et ce dès les années 60. Par la suite, de nombreux langages, surtout les langages fonctionnels, ont défini de telles opérations. De nos jours, la plupart des langages définissent de telles opérations. L'exemple d'exécution 5.1 présente un exemple d'application et un exemple de réduction exprimés dans trois langages : Lisp [Ste84], Haskell [Tho96] et Ruby. De telles opérations sont aussi redevenues (très!) «à la mode» ces dernières années avec l'introduction du modèle Map/Reduce de Google [DG08], popularisé notamment grâce à Hadoop [Whi15].

### 5.4.3 Application et réduction en parallèle

Dans un langage fonctionnel[1], une méthode n'a jamais d'effets de bord — ne peut pas modifier son environnement. En d'autres mots, si on appelle deux fois la même méthode avec les mêmes arguments, on obtiendra toujours *le même résultat*, et ce sans que l'environnement ne soit modifié.

Les opérations `map` et `reduce` introduites par les langages fonctionnels ne visaient pas nécessairement la programmation parallèle. Par contre, puisque les méthodes

[1]En fait, cette affirmation n'est valide que pour les langages *purement* fonctionnels. C'est le cas d'Haskell, mais pas de Lisp ou Ruby.

- **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.

n'ont pas d'effet de bord — donc pas d'interactions entre elles — on en déduit qu'une application `map` est *naturellement parallèle* : toutes les applications de la méthode aux divers éléments de la collection *peuvent se faire en parallèle.*

Dans le cas d'une réduction, si la méthode binaire utilisée pour la réduction est *associative* — ce qui est habituellement le cas pour les opérations utilisées dans des telles $\beta$-réductions, par exemple, $+$, $*$, `MAX`, `MIN` — alors l'ordre d'évaluation n'a pas d'importance. On peut donc parenthéser cette expression de façon différente sans changer le résultat[2]. Par exemple, pour l'expression de réduction présentée plus haut avec $n = 8$, alors les deux façons suivantes de parenthéser l'expression sont équivalentes :

$$((((c_1 \oplus c_2) \oplus c_3) \oplus \ldots) \oplus c_8) = (((c_1 \oplus c_2) \oplus (c_3 \oplus c_4)) \oplus ((c_5 \oplus c_6) \oplus (c_7 \oplus c_8)))$$

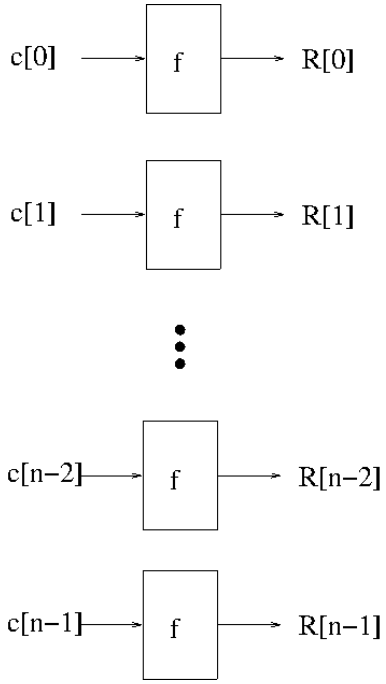
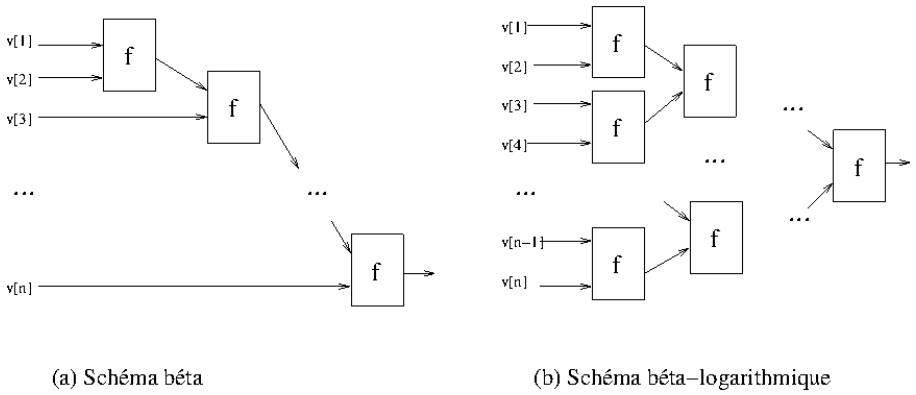



Figure 5.10: Graphe de dépendances pour une $\alpha$-application.

[2]Et ignorant aussi, dans le cas des nombres à virgule flottante, les questions d'arrondissement




(a) Schéma béta

(b) Schéma béta–logarithmique

Figure 5.11: Graphes de dépendances pour une réduction $\beta$ vs. une réduction $\beta$-logarithmique.

Alors que la première façon de parenthéser l'expression conduit à une évaluation en temps linéaire ($O(n)$) — et ce même avec une évaluation parallèle —, la deuxième façon permet une évaluation en temps *logarithmique* ($O(lg\ n)$) — d'où le terme parfois utilisé de *$\beta$-réduction logarithmique*. Voir aussi la Figure 5.11 (adaptée de [GUD96, p. 107]) — dans cette figure, $f$ est l'opération binaire, plutôt que $\oplus$.

Comme on le verra dans les exemples qui suivent, la bibliothèque `PRuby` définit des versions parallèles de ces méthodes, soit `pmap` et `preduce`, qui s'appliquent toutefois uniquement à des objets de classe `Array` ou `Range` et non à des collections arbitraires.

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
```

Le Programme Ruby 5.18 présente une version parallèle d'une méthode pour effectuer la somme de deux tableaux à l'aide de `pmap`, et ce à l'aide de parallélisme à granularité grossière, où les éléments à additionner sont répartis par blocs d'éléments adjacents entre les `nbt` *threads*— par défaut, le mode de répartition est «`static: true`». Difficile de faire plus simple!

### 5.4.5 Calcul de $\pi$ à l'aide d'une méthode Monte Carlo

**Programme Ruby 5.19** Méthode parallèle pour estimer la valeur de $\pi$ 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 $\pi$ 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
```

Dans le Programme Ruby 5.19, qui utilise un `pmap`, on constate qu'après le `pmap`, il est nécessaire de faire la sommation des résultats produits par chacun des *threads*. Il s'agit donc d'un cas typique de *réduction*. Le Programme Ruby 5.20 présente donc une nouvelle, et dernière, version parallèle de la méthode `evaluer_pi`, cette fois en utilisant `preduce`. Là aussi, difficile de faire plus simple et plus succinct!

### 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
```

Et quand on parle de solution parallèle simple — à granularité grossière: voir le Programme Ruby 5.21 pour le calcul de `fact(n)`!

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 :

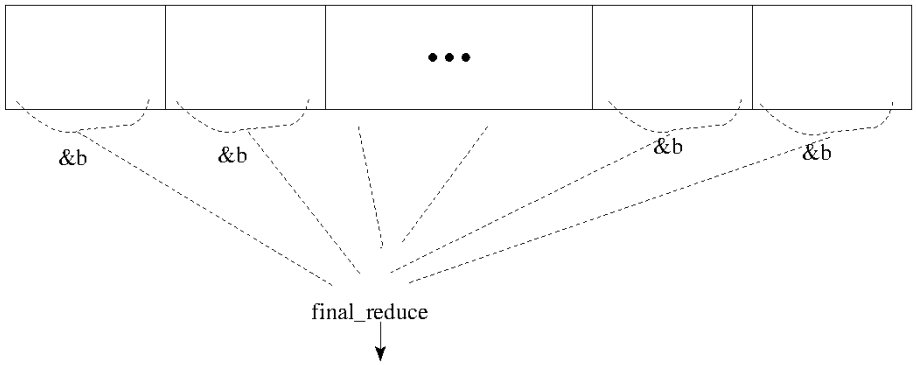
- On suppose que c'est **l'évaluation du prédicat sur un élément** qui est coûteuse (longue à exécuter).
- Faites le travail en deux passes : une première parallèle, l'autre séquentielle.
- La méthode `compact` supprime les `nil` :

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

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

### 5.4.7 Forme générale de `preduce`

Les Figures 5.13–5.14 présentent la description génerale de la méthode `preduce` telle qu'on la retrouve dans la documentation en ligne de la bibliothèque `PRuby`. On remarque qu'il est possible de spécifier une méthode finale de réduction, exprimée sous forme d'une lambda, utilisée tel qu'indiqué dans la figure suivante :




```
- (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`.

**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`

Jusqu'à présent, nous avons vu deux grandes façons de créer des *threads* et de leur attribuer des tâches à traiter :

- **Création dynamique** de *threads* : Dans cette approche, on **crée des *threads* de façon dynamique** — e.g., avec `pcall` ou avec des `future` — et chaque *thread* traite une tâche qui lui est associée au moment où la tâche est créée. Un exemple type est celui du parallélisme récursif, illustré notamment par le Programme Ruby 5.5 (p. 10).

  Un désavantage de cette approche est que le nombre de *threads* créés varie généralement *en méthode de la taille du problème*, et donc ce nombre de *threads* peut être (très!?) élevé. Et c'est le cas même si, comme dans la solution à l'exercice 5.3, on limite le nombre `future`s utilisés, et donc le nombre de *threads* créés.

- **Répartition statique du travail** entre un nombre fixe de *threads* : Dans cette approche, on crée un nombre fixe de *threads* et on répartit le travail à faire entre les *threads* de façon *statique*. Des exemples de ce type sont les Programmes Ruby 5.8 et 5.18, qui utilisent respectivement du parallélisme de boucle (`peach`) et du parallélisme de données (`pmap`), et où les éléments à traiter peuvent être répartis par blocs (d'éléments adjacents) ou de façon cyclique.

Créer des *threads* de façon dynamique peut être couteux, surtout si le nombre de *threads* est élevé. De plus, comme on l'a vu avec l'exemple du calcul récursif et parallèle de la méthode pour factoriel, il arrive souvent que le seul travail effectué par un *thread*... soit de créer d'autres *threads* et d'attendre qu'ils terminent pour combiner les solutions produites par les appels récursifs.

Les coûts élevés de création des *threads* peuvent être réduits lorsqu'on crée, *une seule fois*, un nombre fixe de *threads*. Cette approche est particulièrement efficace lorsque le nombre de *threads* est du même ordre de grandeur que le nombre de processeurs.

Malheureusement, une répartition statique entre un nombre fixe de *threads* ne fonctionne bien *que si le travail à effectuer par chacun des* threads *prend sensiblement 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 faudrait est une 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 entre les *threads*. Une telle approche est possible avec une **répartition dynamique des tâches** ainsi qu'avec l'utilisation d'une approche de type «Coordonnateur/Travailleurs».

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

Sur Unix/Linux, l'utilitaire `wc` permet de calculer le nombre de lignes, de mots et de caractères contenus dans un fichier texte. Voici un exemple :

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

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

Le fichier `foo.txt` contient donc 5 lignes, 10 mots et 40 caractères — chaque ligne se termine par un saut de ligne, donc un caractère «\n», et ces sauts de ligne sont comptés parmi les caractères.

**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
```

Le problème qu'on veut traiter est le suivant : étant donné une *liste de fichiers* (`Array`), on veut obtenir une liste de résultats équivalents à ceux produit par `wc`.

Le Programme Ruby 5.22 présente une première solution, mise en oeuvre avec `pmap` :

- La méthode `strip` permet de supprimer les blancs au début et à la fin d'une chaîne de caractères. Par exemple : `'  abc def '.strip == 'abc def'`
- Étant donnée une chaîne, la méthode `split` retourne un tableau des sous-chaînes, et ce en utilisant l'expression en argument comme délimiteur de séparation. Par exemple :

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

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

- La méthode `IO.readlines(fich)` retourne un tableau contenant toutes les lignes du fichier `fich`.
- Le résultat retourné par `wc1` est un tableau de quatre éléments, équivalent à ce qui est retourné par `wc`. On notera qu'en Ruby, les tableaux n'ont pas à être homogènes, i.e., les éléments d'un `Array` peuvent être de types divers.

Pour que le traitement se fasse de façon parallèle, un `pmap` est utilsé sur la liste des fichiers. Aucun argument n'étant spécifié, on utilisera donc `PRuby.nb_threads` *threads* — i.e., un nombre fixe de *threads* — et une répartition statique par bloc — `pmap = pmap() = pmap( static: true, nb_threads: PRuby.nb_threads )`.

| Quel est le principal défaut du Programme Ruby 5.22? |
|---|

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

Une solution simple au problème identifié dans l'exercice 5.14 consiste à définir `wc` comme suit :

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

Un appel à `pmap` qui utilise «`dynamic: true`» est équivalent à un appel avec «`dynamic: 1`». Comme dans le cas statique, on va créer un nombre fixe de *threads* — égal à `PRuby.nb_threads` si le nombre de *threads* n'est pas spécifié explicitement. La différence par rapport à `static` est que ces *threads* vont se répartir les éléments à traiter non pas au moment du lancement des *threads*, *mais en cours d'exécution*, au fur et à mesure où les éléments de la collection auront été traités.

Donc, lorsqu'un *thread* est activé, sa première action est d'obtenir le prochain élément à traiter **parmi ceux qui ne sont pas encore traités**. Lorsque le *thread* termine de traiter cet élément, il tente alors d'obtenir le prochain élément qui n'a **pas encore été traité**. Finalement, l'exécution du *thread* se termine **lorsqu'il n'y a plus aucun élément à traiter**.

Cette façon de procéder permet de mieux répartir la charge de travail entre les *threads*. Ainsi, il est possible que pendant qu'un *thread* traite un très gros fichier — donc très long à parcourir — un autre *thread* traite plusieurs petits fichiers. Au final, les chances que tous les *threads* finissent *en même temps*, ou presque, sont donc améliorées.

Il est aussi possible d'augmenter la granularité des tâches simplement en spécifiant comme valeur un entier supérieur à 1. Dans notre exemple, si on avait utilisé l'appel «`fichs.pmap( dynamic: 5 ) {...}`», alors un *thread* aurait obtenu les noms de fichiers à traiter par blocs de 5 — ou moins pour le dernier bloc lorsque le nombre total d'éléments n'est pas divisible par 5.

### 5.5.2 Approche *Coordonnateur–Travailleurs* et sac de tâches

Lorsqu'on utilise une approche avec répartition dynamique des tâches, on parle souvent d'une approche **Coordonnateur–Travailleurs** : un des *threads* joue le rôle de **coordonnateur**,dont le travail consiste à gérer les tâches et à les répartir entre les autres *threads* lesquels font le «vrai travail» — d'où le terme de **travailleurs**. Comme on utilise un nombre *fixe* de travailleurs, on crée donc un nombre fixe de *threads*.

Dans certains problèmes, l'exécution d'une tâche peut entrainer **la création d'une ou plusieurs nouvelles tâches**. On utilise alors une structure de données particulière pour représenter les tâches en attente d'exécution, qu'on appelle le **sac de tâches**.

C'est le coordonnateur — parfois un *thread* explicite, parfois simplement le *thread*-maître — qui gère les accès au sac de tâches, pour obtenir une tâche ou en ajouter une nouvelle.

Le code exécuté par les travailleurs a typiquement l'allure suivante :

```
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_travailleurs = PRuby.nb_threads

  # Les taches a mettre initialement 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_travailleurs, *taches ) do |sac_taches|
    sac_taches.each do |i_j|
      res[i_j] = i_j.map { |k| wc1(fichiers[k]) }
    end
  end

  res
end
```

Le Programme Ruby 5.23 présente une nouvelle version de la méthode `wc` appliquée à une liste de fichiers, mais réalisée cette fois avec un sac de tâches explicite — un objet de classe `TaskBag` — et un groupe explicite de *threads* travailleurs :

- On utilise un *thread* par travailleur et autant de travailleurs que de processeurs (`PRuby.nb_thereads`). Quant au coordonnateur, il est implicitement associé au *thread* qui exécute la méthode `wc`, donc qui crée le sac et active les *threads* travailleurs.

- La méthode de création d'un objet `TaskBag` — `create_and_run` — reçoit en argument explicite le nombre de travailleurs qui seront lancés et qui se partageront l'utilisation sac de tâches.

  Elle reçoit aussi en argument explicite la liste des tâches qui seront mises initialement dans le sac de tâches.

Ici, une tâche est représentée par un `Range`. Par exemple, si `fichs.size = 10` et `taille_tache = 3`, alors les tâches suivantes seront ajoutées dans le sac : `0..2, 3..5, 6..8, 9..9`.

La méthode reçoit aussi en argument (implicite) un bloc, qui représente le code qui sera exécuté par chacun des *threads*. Ce bloc reçoit deux (2) arguments : le premier argument, obligatoire, indique le sac de tâches créé par l'appel à `create_and_run` et partagé par les *threads* ; le deuxième argument, optionnel, identifie le numéro du *thread*.

- C'est la méthode `each` qui permet à chacun des *threads* d'obtenir une tâche du sac. Un *thread* donné ne recevra pas toutes les tâches, *uniquement un sous-ensemble* — en d'autres mots, les tâches vont se répartir entre les divers *threads* par l'intermédiaire du `each`, mais pas nécessairement de façon uniforme puisqu'il s'agit d'un comportement *dynamique*.

  - Si un *thread* tente d'obtenir une tâche et que le sac n'est pas vide, alors l'une des tâches est retirée du sac et retournée au travailleur — c'est un *sac*, donc l'ordre de retrait des tâches est indéterminé.
  - Si un *thread* tente d'obtenir une tâche et que le sac est présentement vide, alors le comportement dépend *de l'état des autres* threads, d'où l'importance de connaitre le nombre de *threads* qui se partagent le sac :
    * Il y a encore des *threads* actifs, i.e., qui ne sont pas bloqués en attente d'une tâche : dans ce cas, le *thread* est mis en attente... *parce qu'un des autres* threads *encore actifs pourrait ajouter une nouvelle tâche dans le sac.*
    * Le *thread* est le dernier *thread* actif : dans ce cas, il est impossible que d'autres tâches soient ajoutées au sac. Tous les *threads* bloqués doivent alors être réactivés en signalant *que le sac est définitivement vide*, qu'il n'y aura plus de tâches à traiter, et donc qu'ils peuvent — en fait, **doivent** — terminer! Dans la méthode `TaskBag#each`, cet état est signalé avec `each` qui retourne `nil` à chacun des *threads* en attente.

- Notons que dans le code du travailleur — le bloc spécifié lors de l'appel à `create_and_run` —, une tâche obtenue est un `Range`, affecté à la variable `i_j`. Dans l'exemple, on voit que le résultat du `map` — un `Array` — est affecté directement *à la tranche appropriée du tableau de résultat* :

  ```
  res[i_j] = i_j.map { ... }
  ```

Cet exemple permet d'illustrer le fonctionnement d'un sac de tâches, mais son utilisation n'est pas strictement nécessaire, puisqu'un simple `pmap` avec `dynamic`

aurait été suffisant. C'est le cas parce que toutes les tâches sont créées une fois pour toute par le coordonnateur, au moment de la création du sac lorsque les travailleurs sont lancés. En d'autres mots, l'exécution d'une tâche n'entraîne pas la création de nouvelles tâches.

### 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
```

Le Programme Ruby 5.24 présente une méthode `mystere` qui utilise un `TaskBag` et où l'ajout de nouvelle tâches se fait vraiment de façon dynamique — une nouvelle tâche peut être ajoutée au sac (avec `TaskBag#put`) *pendant la traitement d'une tâche*.

Plus précisément, au début de la méthode `mystere`, le coordonnateur ajoute *une seule et unique tâche* dans le sac — la tâche `1..n`. Par la suite, ces sont les travailleurs qui ajoutent de nouvelles tâches de la forme `(m+1)..j` (dans le `while`).

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

**Exercice 5.15:** Méthode `mystere`.

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.

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

La prochaine structure de programmation parallèle de `PRuby` que nous allons introduire permet le «*parallélisme de flux*», aussi appelée parallélisme avec «*filtres et pipelines*».

Dans cette approche, le parallélisme vient de ce qu'on traite des **flux de données** — des séries de données, des *data streams* — et que le traitement à effectuer sur une donnée d'un flux est décomposé en une série d'étapes indépendantes, comme dans une chaine de montage.

Un endroit où on trouve cette forme de parallélisme, même si on ne s'en rend pas toujours compte, est au niveau matériel des ordinateurs. Comme on l'a vu précédemment, les ordinateurs modernes exécutent les instructions d'un programme à l'aide d'un *pipeline d'instructions* — une «chaîne de montage» pour l'exécution des instructions. Une instruction donnée va donc nécessiter plusieurs cycles pour s'exécuter, puisqu'elle doit parcourir l'ensemble de la chaîne de montage. Par contre, à un instant donné, on aura plusieurs instructions en cours d'exécution, chacune à un étage distinct du pipeline, d'où le parallélisme — dans ce cas, on parle parfois de «parallélisme temporel».

Plus spécifiquement, l'approche que nous allons présenter dans cette section s'inspire à la fois des *pipes* Unix — quant à la façon de créer les pipelines avec l'opérateur «`|`» — mais aussi du langage `Go`[3]— quant à la syntaxe pour manipuler les canaux de communication (lecture, écriture et fermeture).

### 5.6.1 Unix et ses pipes

Un contexte où les flux de données, filtres et pipelines sont fréquents — pour ne pas dire omniprésents — est lorsqu'on programme sous Unix et qu'on utilise des *pipes* — le fameux symbole «`|`». Voici ce qu'expriment certains auteurs sur la philosophie qui sous-tend l'approche Unix :

> *(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**. *Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.*
>
> *McIlroy* et al. *[MPT78] (cité dans [Ray04])*

[3] `https://golang.org/`

> *Unix tradition strongly encourages writing programs that read and write simple, textual, stream-oriented, device-independent formats. Under classic Unix, as many programs as possible are written as simple filters, which take a simple text stream on input and process it into another simple text stream on output.*
>
> *Despite popular mythology, this practice is favored not because Unix programmers hate graphical user interfaces. It's because if you don't write programs that accept and emit simple text streams, it's much more difficult to hook the programs together.*
>
> *E. Raymond [Ray04]*

```
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.

La Figure 5.15 présente un exemple d'utilisation de processus et de *pipes* Unix. Ce script sert à éliminer les commandes d'un fichier LaTeX dans le but de compter le nombre de «vrais» mots contenus dans le texte — donc sans compter les commandes LaTeX.[4] Les diverses commandes Unix utilisées dans ce script sont les suivantes (descriptions produites avec `man`) :

- `sed` : "*The* `sed` *utility is a stream editor that reads one or more text files, makes editing changes according to a script of editing commands, and writes the results to standard output.*
- `grep` : "*The* grep *utility searches files for a pattern and prints all lines that contain that pattern.*"

[4] LaTeX est un langage et système pour composer des documents, fondé sur l'utilisation de TeX. Les commandes LaTeX sont indiquées par des identificateurs qui débutent par un «\», par exemple, `\section{...}`, `\title{...}`, `\begin{...}`, etc.

Note : Rôle de l'option "`-v`" de `grep` : "*Print all lines except those that contain the pattern.*

- `tr` : "*The* `tr` *utility copies the standard input to the standard output with substitution or deletion of selected characters.*".
- `wc` : "*The* `wc` *utility reads one or more input files and, by default, writes the number of newline characters, words and bytes contained in each input file to the standard output.*".

  Note : Rôle de l'option "`-w`" de `wc` : "*Count words delimited by white space characters or new line characters.*"

La tâche à effectuer est réalisée en la décomposant en plusieurs sous-tâches plus simples et en combinant simplement ces sous-tâches à l'aide de *pipes* :

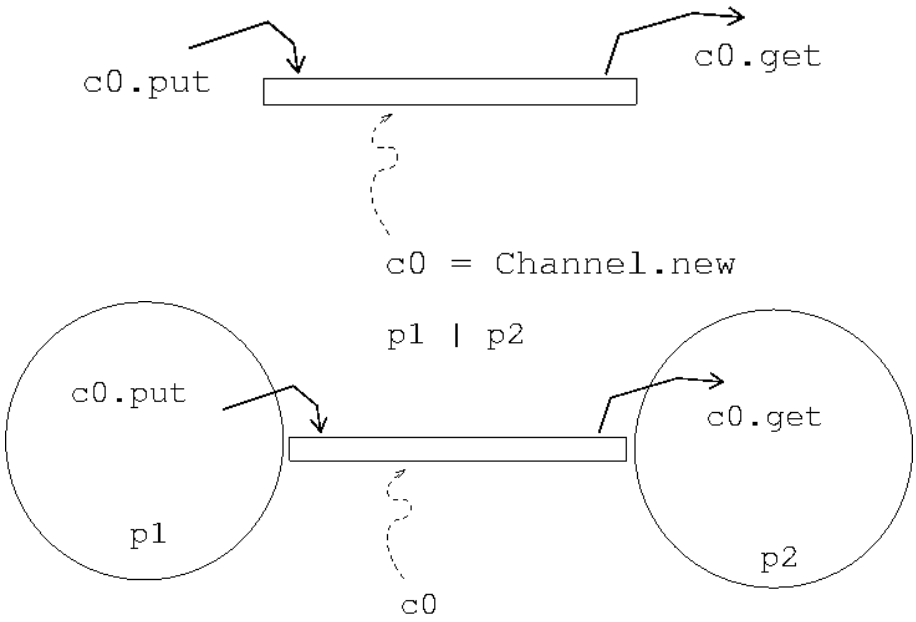
- On *filtre* — au sens de «supprimer» — les blocs de lignes qui forments les figures ou tables.
- On filtre les lignes ne contenant que des commentaires (débutant avec «%»).
- On transforme les espaces insécables («~») en espaces blancs ordinaires.
- On transforme les caractères de tabulation et blancs en sauts de lignes — donc on met un mot (groupe de caractères non blancs) par ligne.
- On filtre (supprime) toutes les lignes contenant une commande LaTeX.
- On compte le nombre de mots résultants.

Dans ce script, on utilise donc une forme de décomposition «diviser-pour-régner» — on décomposer un problème complexe en sous-problèmes plus simples, on résoud les sous-problèmes, puis on combine les solutions aux sous-problèmes. La différence avec les exemples vus précédemment pour le parallélisme récursif est que cette décomposition **ne se fait pas de façon récursive!**

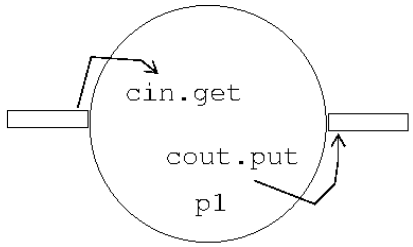
Sous Unix, l'utilisation de *pipes* implique aussi que toutes ces commandes pourraient s'exécuter *de façon concurrente et parallèle.* Ainsi, pendant qu'une commande traite un ligne présente sur son flux d'entrée, une autre commande pourrait être en train de traiter une ligne, différente, de son propre flux d'entrée.

### 5.6.2 PRuby et ses Channels

En PRuby, le parallélisme de filtres et pipelines s'exprime à l'aide de filtres spécifiés par des **lambda-expressions**, interconnectés par des canaux de communication de classe Channel. Ces canaux sont illustrés dans les figures ci-bas.




Un filtre est défini à l'aide d'une lambda-expression, laquelle doit recevoir **deux arguments**. Ce sont ces arguments qui donnent accès **aux deux canaux** associés au filtre — le canal d'entrée, typiquement dénoté par `cin`, et le canal de sortie, typiquement dénoté par `cout` :




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

La figure 5.16 présente les principales méthodes associées à la classe `Channel`. Soulignons que la fin d'un flux de données est signalée par la valeur `PRuby::EOS`. Cette valeur est retournée de façon **persistente**, i.e., si un appel à `get` retourne `PRuby::EOS`, alors tous les appels subséquents vont continuer à retourner `PRuby::EOS`. Soulignons aussi que cette valeur est automatiquement transmise sur un *canal de sortie* lorsque la méthode `close` est appelée sur un tel canal.

```
- (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

Le Programme Ruby 5.25 définit une méthode, avec filtres et pipelines, pour identifier tous les mots d'un fichier texte et produire en sortie un fichier contenant ces mots, mais avec un seul mot par ligne, **en ordre** alphabétique **et sans doublon**. Il s'agit donc d'un programme pour réaliser une fonctionnalité semblable à celle de «`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
```

Lorsqu'on veut créer et exécuter un pipeline en `Pruby`, on doit tout d'abord

créer un objet `Pipeline`. La façon la plus simple pour ce faire est d'utiliser des lambda-expressions *connectées* avec l'opérateur «`|`».

Généralement, pour communiquer avec l'environnement, il faut aussi utiliser les constructeurs `Pipeline.source` et `Pipeline.sink`, qui permettent de créer l'étage initial du pipeline — sans canal d'entrée — et l'étage final — sans canal de sortie.

Une fois un objet `Pipeline` créé, on lance alors son exécution avec `run`. Le *thread* qui appelle `run` sera alors bloqué jusqu'à ce que tous les étages du pipeline aient terminé — donc jusqu'à ce que tous les canaux aient été fermés et que le signal de fin de flux — `PRuby::EOS` (*End Of Stream*) — ait été transmis à travers tous les étages du pipeline.[5]

Un filtre — un étage d'un pipelineà l'exception du premier (source) ou du dernier (puits = *sink*) — est donc un processus représenté par une `lambda` expression, laquelle reçoit en argument deux (2) objets de type `Channel`, donc *deux flux de données* par l'intermédiaire desquels le processus peut communiquer avec son voisin en amont ou son voisin en aval — voisin gauche ou voisin droite, si on visualise le pipeline de la gauche vers la droite :

1. Le canal d'entrée, sur lequel le processus peut appeler la méthode `get`, qui lui permet de recevoir une donnée du voisin gauche.

   Signalons que lorsque la valeur retournée par `get` est `PRuby::EOS`, alors c'est que le flux d'entrée est terminé. Ce n'est pas une erreur d'appeler à nouveau `get` ; par contre, tous les appels subséquents retourneront `PRuby::EOS`.

   Notons aussi qu'il est généralement possible d'éviter de tester explicitement la fin de canal en utilisant plutôt la méthode `each`, qui permet d'itérer sur chacun des éléments dans le style propre à Ruby.

2. Le canal de sortie, sur lequel le processus peut appeler la méthode `put`, dans le but de transférer un élément au voisin droite. Un synonyme pour cette méthode est l'opérateur «`<<`».

Dans notre exemple, la tâche globale est décomposée en six (6) sous-tâches plus simples, six processus qui pourront s'exécuter de façon concurrente et qui s'échangeront des données et se synchroniseront par l'intermédiaire des canaux qu'ils partagent :

- `PRuby::Pipeline.source` : Cette méthode crée un processus, *sans canal d'entrée*, qui reçoit en argument un nom de fichier (ou un `Array`) et qui émet sur son canal de sortie les lignes — l'une après l'autre — puis qui ferme le canal

[5] Il est aussi possible de ne pas attendre/bloquer. Pour ce faire, il suffit d'appeler `run` avec l'argument `:NO_WAIT`.

de sortie lorsque la fin du fichier est rencontrée, ce qui a pour effet d'émettre `PRuby::EOS` sur ce canal de sortie.

- `generer_mots` : Reçoit sur son canal d'entrée des lignes de texte et émet sur son canal de sortie les mots – les suites de caractères sans espace blanc — contenus sur chacune des lignes.
- `filtrer_mots_invalides` : Reçoit sur son canal d'entrée des suites de caractères non blancs, et n'émet sur le canal de sortie que ceux qui contiennent uniquement des lettres.
- `trier` : Reçoit sur son canal d'entrée une suite de mots et émet sur le canal de sortie ces même mots, mais triés.
- `supprimer_doublons` : Reçoit sur son canal d'entrée une suite de mots *triés* et n'émet sur le canal de sortie que la première occurrence d'un mot lorsqu'un mot apparaît plusieurs fois de suite (mots consécutifs, puisque triés).
- `PRuby::Pipeline.sink` : Crée un processus qui reçoit sur son canal d'entrée une suite d'éléments et les écrit dans un fichier dans l'ordre reçu (ou les ajoute dans un `Array`).

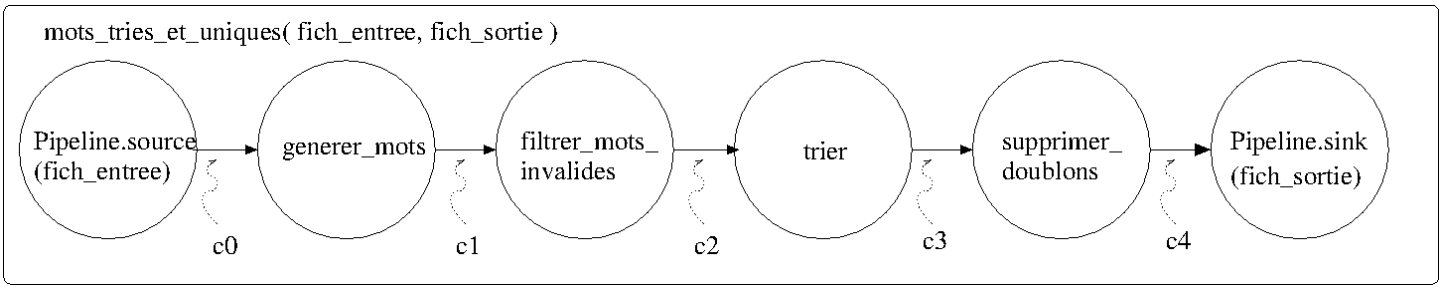



Figure 5.17: Graphe des processus et de leurs canaux pour les filtres et le pipeline du Programme Ruby 5.25.

La Figure 5.17 illustre la structure du pipeline pour le Programme Ruby 5.25 : le premier processus n'a pas de canal d'entrée — c'est une source — alors que le dernier n'a pas de canal de sortie — c'est un puits.

Un exemple pour mieux expliquer le rôle, et le contenu, des divers canaux de la Figure 5.17 est comme suit.

Supposons qu'on ait fichier d'entrée contenant les lignes suivantes :

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

Les éléments qui transiteront dans chacun des canaux seront alors les suivants — indiqués sous forme d'une liste, en ignorant les caractères de saut de ligne, l'élément à la position 0 indiquant le premier élément à transiter sur le canal, etc — 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»

Dans l'exemple qui précède, l'utilisation de filtres et pipelines permet de bien modulariser la solution, et ce en utilisant une approche diviser-pour-régner non récursive, où chaque sous-tâche est clairement identifiée.

Certains problèmes, bien qu'ils semblent simples à première vue, peuvent être difficiles à résoudre à l'aide d'un programme séquentiel, comme l'a montré M. Jackson il y a longtemps [Jac75, Jac83].

Voici un exemple simple d'un tel problème, que nous appelerons le problème «de Jackson», qui a cette propriéte. Le problème est de transformer le contenu d'un fichier texte comme suit :

- On reçoit en entrée un fichier texte formé de lignes de caractères *de longueur possiblement variable.*
- On veut produire en sortie un fichier texte, avec les mêmes caractères, y compris les blancs, mais où toutes les lignes sont exactement de longueur `n` — sauf peut-être la dernière ligne.
- La seule différence au niveau des caractères émis est que lorsqu'on rencontre deux caractères «`*`» consécutifs dans le flux d'entrée, alors on doit les remplacer par le caractère unique «`^`».

La Figure 5.18 présente un exemple de fichier d'entrée et du fichier de sortie résultat, et ce pour `n=4`.

```
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.

```
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
```

Le Programme PRuby 5.26 présente une solution avec filtres et pipeline, où la tâche essentielle (donc, on ignore `source` et `sink`) est décomposée entre trois (3)

sous-tâches :

- `depaqueter` : Ce processus transforme simplement un flux de lignes en un flux de caractères. Le canal d'entrée contient donc des lignes de longueur arbitraire, alors que le canal de sortie contient une suite de caractères.
- `changer_exposant` : Ce processus examine les caractères et remplace les doublons «`**`» par l'unique caractère «`^`». Signalons que le code de cette lambda-expression est relativement simple car il utilise `chan.peek`. Cette méthode permet de *voir* le prochain élément du canal d'entrée, mais sans le retirer du canal. Donc, si on fait plusieurs appels consécutifs à `peek` sans faire de `get`, on obtiendra toujours le même caractère — ce qui n'est évidemment pas le cas pour `get`, qui «consomme» le caractère!
- `paqueter` : Ce processus reçoit un flux de caractères et les accumule dans un tampon (`ligne`) jusqu'à ce que le tampon contienne exactement `n` éléments, et alors il émet ces lignes sur le canal de sortie. Lorsque la fin du flux est rencontrée, la ligne possiblement incomplète en cours de construction est aussi émise *avant de fermer le canal de sortie* (et donc avant de propager le signal `PRuby::EOS`).

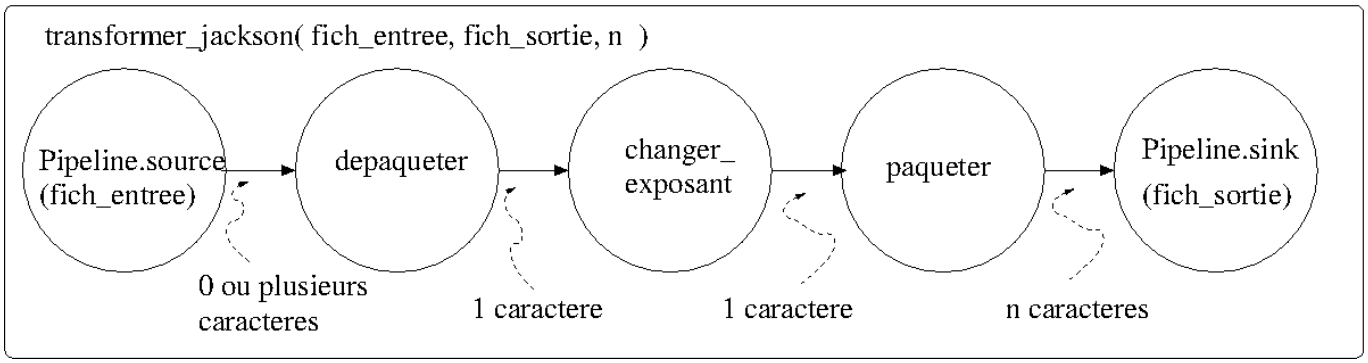



Figure 5.19: Graphe des processus et de leurs canaux pour les filtres et le pipeline du Programme Ruby 5.26.

La Figure 5.19 illustre la structure du pipeline pour le Programme Ruby 5.26 et indique aussi le contenu des canaux — plus précisément, de combien de caractères est formé *chacun des éléments émis/reçus* sur chacun des canaux.

(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

Le Programme Ruby 5.28 (avec les méthodes auxiliaires du Programme Ruby 5.27) présente un pipeline qui permet de produire la liste, en ordre alphabétique, des mots d'un texte avec le nombre d'occurrences de chacun des mots. Comme dans l'exemple précédent, pour simplifier, on suppose qu'un mot est simplement une suite de caractères non blancs. La Figure 5.20 donne la représentation graphique de ce pipeline.

Voici un petit exemple simple de données et du résultat obtenu :

```
# 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
```

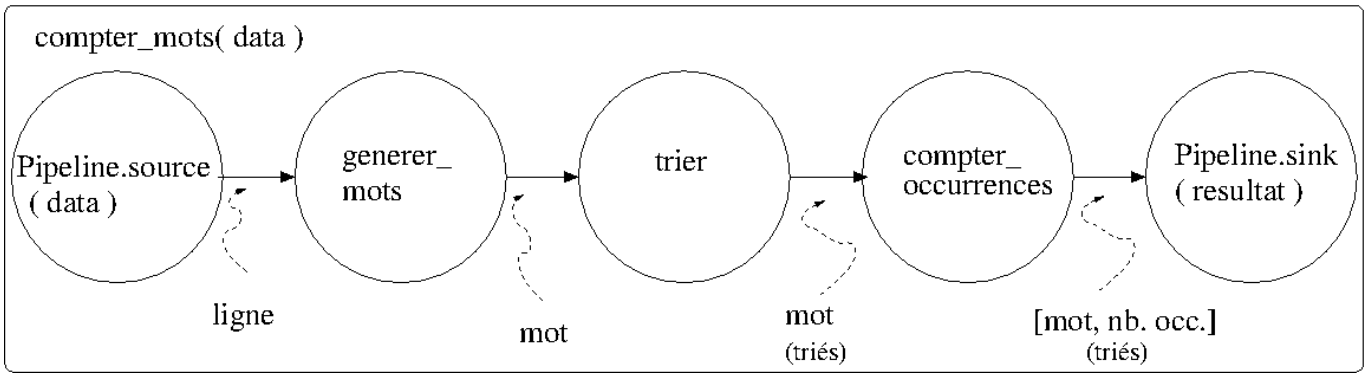



Figure 5.20: Graphe des processus et canaux pour le pipeline du Programme Ruby 5.28.

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.

### 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/`

> **Go** est un langage de programmation compilé et **concurrent** inspiré de C et Pascal. Ce langage a été développé par Google à partir d'un concept initial de Robert Griesemer, Rob Pike et Ken Thompson.
>
> Le mot clé **go** permet à un appel de fonction de s'exécuter **en concurrence** avec le thread courant. Ce code exécuté en concurrence se nomme une **goroutine** [.]
>
> Les goroutines communiquent entre elles par **passage de messages**, en envoyant ou en recevant des messages sur des **canaux**.
>
> Source : `https://fr.wikipedia.org/wiki/Go_%28langage%29`

### Exemple PRuby illustrant le style Go avec canaux et processus explicites

**Programme Ruby 5.29** Un petit pipeline avec trois processus.

```
# 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
```

Le Programme Ruby 5.29 illustre la création d'un petit pipeline (linéaire) comportant trois processus et quatre canaux (`c1, c2, c3` et `c4`). Tant les canaux que les processus sont créés explicitement, dans un style semblable à ce qu'on écrirait dans le cadre du langage `Go`.[6]

[6] `https://golang.org/`

- La méthode `PRuby::Channel.new` crée un nouveau canal. Par défaut, ce canal est non borné — i.e., il n'y a aucune limite sur le nombre d'éléments pouvant être ajoutés dans le canal (= il n'est jamais plein!).

- La méthode `go`, appliquée à une lambda-expression, lance un nouveau *thread* pour évaluer cette lambda-expression, laquelle reçoit en argument les canaux de communication à utiliser indiqués comme arguments à la méthode `go`.

  De façon typique, dans un pipeline linéaire, on transmet deux canaux pour un processus filtre — canal d'entrée + canal de sortie — et un seul canal pour une source (canal de sortie) ou un puits (canal d'entrée).[7]

  Par contre, pour une topologie plus complexe — e.g., arbre, graphe, cyclique ou non —, un processus créé avec `go` pourrait utiliser plus de deux canaux.

- Le pipeline construit dans l'exemple est semblable à celui qui serait obtenu avec l'utilisation de l'opérateur «`|`» : `...| p1 | p2 | p3 | ...`

  Lorsqu'on manipule des pipelines dans le style Unix, la création des canaux et leur association aux processus est *implicite* — les canaux sont créés et liés aux processus par la bibliothèque PRuby, lors de l'appel à la méthode «`|`». Par contre, dans le style Go, on doit créer explicitement les canaux et les transmettre explicitement aux processus (via la méthode `go`) pour les lier.

[7] Notons que, dans notre exemple, il n'y a pas de source ou puits explicite, et ce pour simplifier la présentation du code.

**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
```

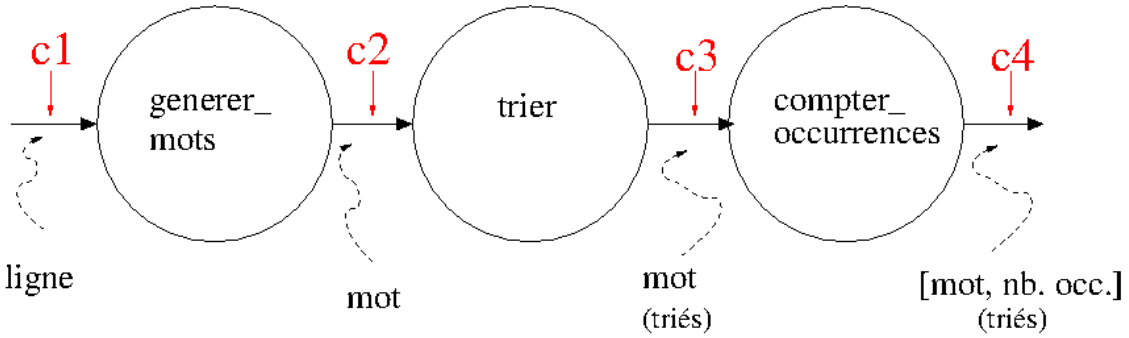



Figure 5.22: Graphe des processus Go et canaux, explicites, pour le pipeline du Programme Ruby 5.30.

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

### Tri unique des mots d'un fichier

Une autre approche de parallélisme de flux est celle avec *streams*, introduite il y a longtemps dans certains langages fonctionnels [AS85, MS+85], puis reprise plus récemment par d'autres langages, notamment Java 8.0 avec les *streams* du paquetage `java.util.stream`[8] ou certains langages de traitement de données massives et leurs collections parallèles [KKWZ15].

**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
```

Le Programme Ruby 5.31 présente une méthode pour trier les mots d'un fichier, en s'assurant que chaque mot apparaît au plus une fois — donc une méthode avec un effet identique à celui de la méthode du Programme Ruby 5.25 (p. 69).

Ici, cette méthode utilise les *streams* de la bibliothèque PRuby pour obtenir l'effet désiré, et ce en utilisant aussi du parallélisme de flux.

Quelques explications :

- Un *stream* est une forme de collection sur laquelle plusieurs des méthodes associées aux collections Ruby peuvent être utilisées — donc, plusieurs des méthodes du module `Enumerable`. Certaines de ces méthodes produisent un résultat qui est lui-même un *stream* — e.g., `map`, `filter`, `reject`, etc. — alors que d'autres produisent un résultat d'un autre type — e.g., `reduce`.

- Une caractéristique importante d'un *stream* est qu'il s'agit typiquement d'une collection **potentiellement infinie — donc non bornée (*unbounded*)**

[8] `https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html`

**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
```

— **produite de façon incrémentale.** Des exemples de tels flux de données sont des données en temps réel, par exemple, informations de réseaux de senseurs, paquets de traffic Internet, données financières (*on-line financial trading*), fichiers de journalisation (*log files*), etc.

Pour plusieurs méthodes s'appliquant à un *stream* et produisant un *stream* en résultat, la collection d'entrée n'a pas besoin d'être disponible dans son ensemble pour produire le *stream* en sortie. Les éléments peuvent être traités au fur et à mesure où ils deviennent disponibles, et les valeurs du *stream* résultant sont donc produites et émises au fur et à mesure. Les méthodes s'exécutent donc de façon **incrémentale**. En fait, lorsque le *stream* est effectivement non borné, la collection dans son ensemble **ne peut jamais être disponible**.

- La méthode `PRuby::Stream.source` reçoit en argument un nom de fichier (ou une collection ordinaire, par exemple, un `Array`) et génère un *stream* à partir des lignes de ce fichier (ou des éléments de la collection).

- L'exemple Ruby 5.2 illustre quatre des méthodes de la classe `PRuby::Stream` utilisées dans les exemples qui suivent, soit les méthodes `flat_map`,[9] `filter`, `uniq` et `group_by_key`. Les trois premières méthodes peuvent, tel qu'indiqué plus haut, être évaluées de façon incrémentale sur les éléments du *stream* d'entrée.

  Par contre, la méthode `group_by` — de même que la méthode `sort` (non illustrée dans les exemples) — ne possède pas cette propriéte, puisque pour commencer à produire le résultat, il est nécessaire de connaitre *tous les éléments du flux d'entrée*.[10]

  Soulignons que la méthode `group_by_key` *i*) prend en argument une collection *de paires* (tableaux de taille 2) et *ii*) produit en sortie aussi une collection de paires, mais d'une forme différente : dans la collection de sortie, la clé (le $1^{er}$ champe) est *unique* et la valeur associée est la collection des valeurs (le $2^{e}$ champ) de la collection d'entrée associées à cette clé.

- La méthode `sink`, comme le nom l'indique, est un puits, donc permet de **collecter** les éléments du *stream*. Lorsque l'argument, comme ici, est un `String`, la chaine est interprétée comme un nom de fichier et les éléments du

[9] Voici ce qu'indique la documentation de la méthode `flat_map` : «Applique une méthode (un bloc) sur chacun des elements du stream d'entree. La méthode produit en sortie un `Array` d'elements (contenant 0, 1 ou plusieurs elements), lesquels `Array`s sont ensuite concatenes dans le stream de sortie.»

[10] De plus, la méthode `sort` ne peut donc être exécutée que si le *stream*d'entrée est borné et elle ne commencera à émettre un résultat (le plus petit élément) que lorsque la fin du *stream* aura été rencontrée.

*stream* sont émis dans le fichier. Un *stream* peut aussi être transformé en une collection «normale», par exemple, en un `Array` en utilisant la méthode `to_a` (voir Exemple Ruby 5.2). Dans ce cas, là aussi le *stream* doit être borné.

- Dans la mise en oeuvre de la bibliothèque PRuby, une série de méthodes de manipulation de *streams* telle que celle présentée dans le Programme Ruby 5.31 est en fait **une forme de pipeline avec filtres et canaux**. Chaque méthode qui génère ou traite un *stream* est un *thread* indépendant, qui communique avec ses voisins par l'intermédiaire de canaux de communication (`PRuby::Channel`). C'est le canal, non borné, qui contient les éléments du *stream*. C'est ce qui explique qu'on a bien, dans Programme Ruby 5.31, du parallélisme de flux comportant plusieurs *threads* actifs travaillant de façon concurrente, ou parallèle.

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

Le Programme Ruby 5.32 présente une série de *streams* permettant de produire la liste, en ordre alphabétique, des mots d'un texte avec le nombre d'occurrences de chacun des mots. Il s'agit donc d'une méthode équivalente à celle présentée dans le Programme Ruby 5.28, mais beaucoup plus succincte puisqu'une grande la complexité est en grande partie dissimulée dans les méthodes, complexes, telles que `flat_map` et, surtout, `group_by_key`.

**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
```

L'exemple d'exécution 5.2, qui utilise à trois endroits la méthode `peek` pour générer une trace d'exécution, illustre la façon dont les *streams* sont transformés à travers les différentes étapes de traitement.

```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`.

```
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 ) do |k|
    a[k] + b[k]
  end
end
```

**Question** : Avantages? Désavantages?

# Références


[AS85] H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, McGraw-Hill Book Co., Cambridge, Ma., 1985.

[DG08] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[Fos95] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995. `http://www-unix.mcs.anl.gov/dbpp`.

[GUD96] M. Gengler, S. Ubéda, and F. Desprez. *Initiation au parallélisme—Concepts, architectures et algorithmes*. Masson, 1996. [QA76.58G45].

[Jac75] M.A. Jackson. *Principles of Program Design*. Academic Press, 1975.

[Jac83] M.A. Jackson. *System Development*. Prentice-Hall, 1983.

[KKWZ15] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia. *Learning Spark*. O'Reilly, 2015.

[MPT78] M.D. McIlroy, E.N. Pinson, and B.A. Tague. Unix time-sharing system forward. *The Bell System Technical Journal*, 57(6–2), 1978.

[MS+85] J.R. McGraw, S. Skedzielewski, et al. SISAL: Streams and Iteration in a Single Assignment Language—Language reference manual version 1.2. Technical Report M-146, Lawrence Livermore National Laboratory, 1985.

[Ray04] E.S. Raymond. *The Art of UNIX Programming*. Addison-Wesley, 2004.

[Ste84] G.L. Steele Jr. *Common LISP: The Language*. Digital Press, 1984.

[Tho96] S. Thompson. *Haskell—The Craft of Functional Programming*. Addison-Wesley, 1996.

[Whi15] T. White. *Hadoop—The Definitive Guide (4th Edition)*. O'Reilly, 2015.