Solution labo #4 : Palindromes

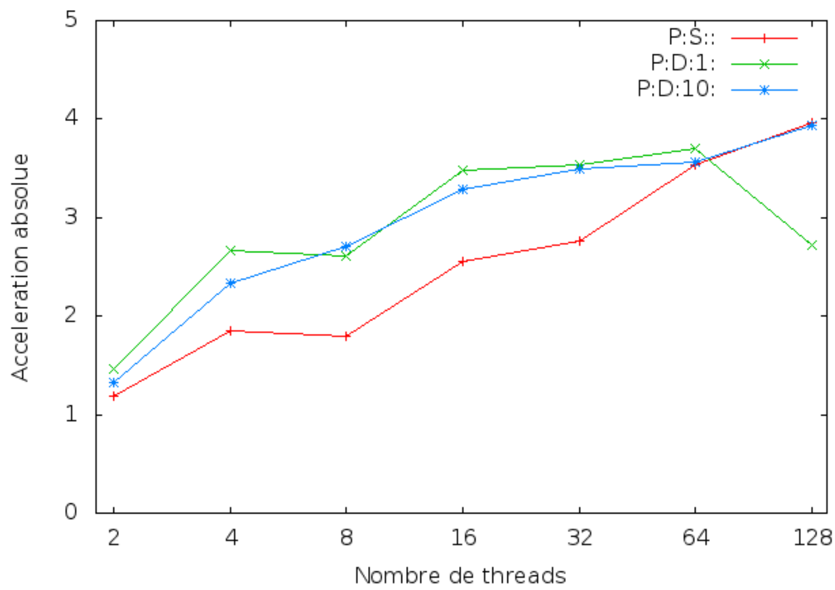
1 Première version

Extraits du fichier `palindromes.rb`

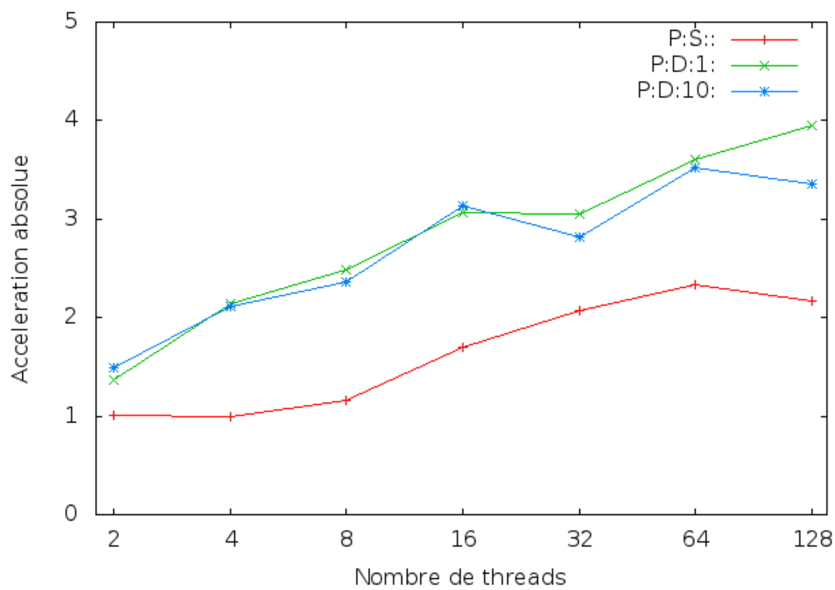
```
class String
  def palindrome?
    i, j = 0, size-1
    while i < j
      i += 1 while self[i] !~ /[a-zA-Z]/ && i < j
      j -= 1 while self[j] !~ /[a-zA-Z]/ && i < j
      return false unless self[i].upcase == self[j].upcase

      i, j = i+1, j-1
    end

    true
  end
end

def palindromes_seq?( *chaines )
  chaines.map(&:palindrome?)
end

def palindromes_par_static?( taille_tache, *chaines )
  chaines.pmap(static: taille_tache, &:palindrome?)
end

def palindromes_par_dynamic?( taille_tache, *chaines )
  chaines.pmap(dynamic: taille_tache, &:palindrome?)
end
```

Graphes d'accélération

Même quantité de travail pour toutes les tâches



Quantité de travail variable entre les tâches



Dans le premier *benchmark*, le travail à faire est le même pour toutes les chaines, puisque c'est toujours la même chaine qui est retournée par la méthode `generer_chaine`. Pour un grand nombre de *threads*, il y a moins de surcoûts de synchronisation que dans l'approche dynamique, et c'est ce qui explique pourquoi l'approche statique s'améliore, alors que l'approche dynamique avec un seul item par tâche empire (trop de synchronisations).

Dans le deuxième *benchmark*, le travail à faire pour les 80 pour cent des chaines présentes au début du tableau... est trivial (chaine vide). Avec la répartition statique, rapidement plusieurs des *threads* n'ont plus rien à faire, ce qui rend le programme moins efficace. Par contre, l'approche dynamique permet de mieux répartir le travail à faire entre les *threads*. Mais on remaque que, là aussi, lorsqu'un seul item est traité à la fois et qu'on utilise un grand nombre de *threads*, les performances se dégrandent parce qu'il y a trop de synchronisations.

2 Deuxière version

Voici une deuxième version de la méthode `String#palindrome?`

```
def palindrome?
  ch = gsub(/[^a-zA-Z]/, '')
  ch.upcase!

  ch == ch.reverse
end
```

Initialement, je n'ai pas utilisé cette version car le fait de créer deux chaines intermédiaires — la première produite par `gsub`, l'autre produite par `reverse` — me semblait inefficace. **Sauf que j'avais tort!**

> *"We should forget about small efficiencies, say about 97% of the time:* **Premature optimization is the root of all evil.***"*
>
> *D. Knuth/C.A.R. Hoare*

1. **The first rule of optimization**: *Don't.*
2. **The second rule of optimization**: *Don't... yet.*
3. **The third rule of optimization**: *Profile before optimizing*

Soit le programme *benchmark* suivant.

```
require 'benchmark'
require 'palindromes'

class String
  def palindrome1
    i, j = 0, size-1
    while i < j
      i += 1 while self[i] !~ /[a-zA-Z]/ && i < j
      j -= 1 while self[j] !~ /[a-zA-Z]/ && i < j
      return false unless self[i].upcase == self[j].upcase

      i, j = i+1, j-1
    end

    true
  end

  def palindrome2
    ch = gsub(/[^a-zA-Z]/, '')
    ch.upcase!

    ch == ch.reverse
  end
end

NB_FOIS = 1000

Benchmark.bm do |bm|
  bm.report("palindrome1") do
   NB_FOIS.times { UN_TRES_LONG_PALINDROME.palindrome1 }
  end
  bm.report("palindrome2") do
    NB_FOIS.times { UN_TRES_LONG_PALINDROME.palindrome2 }
  end
end
```

Résultats :

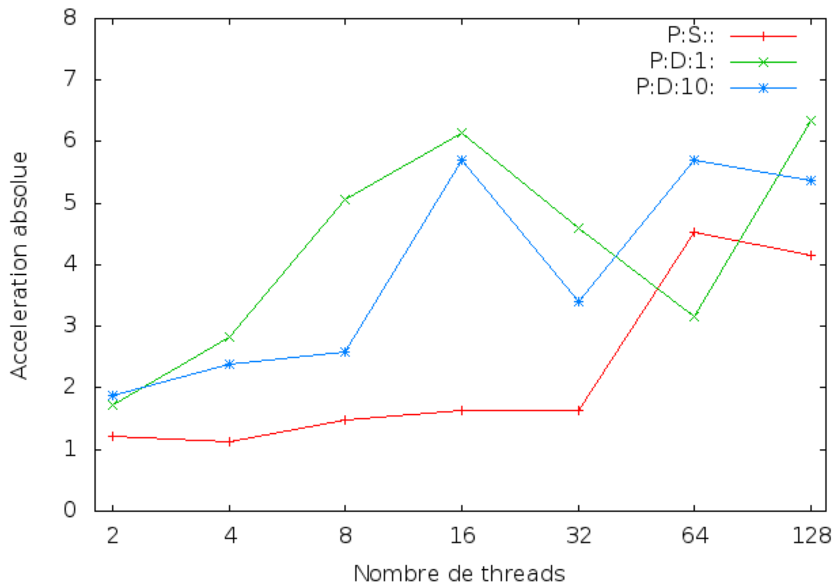
```
             user       system     total        real
palindrome1  2.230000   0.060000   2.290000 (   1.155000)
palindrome2  0.610000   0.010000   0.620000 (   0.281000)
```

Donc, la solution **simple** est la plus performante, quatre (4) fois plus rapide
⇒ KISS

Pourquoi : coût du *pattern-matching* répétitif ☹

Nouveaux graphes d'accélération

Même quantité de travail pour toutes les tâches



Quantité de travail variable entre les tâches