Solution labo #2 : Nombre d'inversions dans un tableau avec parallélisme récursif

1 Extraits du fichier `nb_inversions.rb`

```
#
# Methode sequentielle recursive.
#
def nb_inversions_rec( seuil )
  nb_inversions_rec_ij( 0, size-1, seuil )
end

def nb_inversions_rec_ij( i, j, seuil )
  return nb_inversions_seq_ij(i, j) if j - i <= seuil

  m = (i + j) / 2

  nb1 = nb_inversions_rec_ij(i, m, seuil)

  # Notez, dans l'appel pour n2, le recouvrement (i, m)/(m, j).
  #
  # Ce recouvrement permet de traiter l'element a la frontiere m,
  # Autrement, on aurait pu/du ecrire:
  #   nb2 = nb_inversions_rec_ij( m+1, j, seuil )
  #   nb1 + nb2 + (self[m] <= self[m+1] ? 0 : 1)
  nb2 = nb_inversions_rec_ij(m, j, seuil)

  nb1 + nb2
end
#
```

```ruby
#
# Methode parallele recursive avec utilisation de PRuby.pcall.
#
def nb_inversions_pcall( seuil )
  nb_inversions_pcall_ij( 0, size-1, seuil )
end

def nb_inversions_pcall_ij( i, j, seuil )
  return nb_inversions_seq_ij(i, j) if j - i <= seuil

  m = (i + j) / 2

  nb1 = nb2 = nil
  PRuby.pcall( lambda { nb1 = nb_inversions_pcall_ij(i, m, seuil)
                lambda { nb2 = nb_inversions_pcall_ij(m, j, seuil)

  nb1 + nb2
end

#
```

```ruby
#
# Methode parallele recursive avec utilisation de PRuby.future.
#
def nb_inversions_future( seuil )
  nb_inversions_future_ij( 0, size-1, seuil )
end

def nb_inversions_future_ij( i, j, seuil )
  # Note: Il est possible de ne generer qu'un seul future!
  return nb_inversions_seq_ij(i, j) if j - i <= seuil

  m = (i + j) / 2

  nb1 = PRuby.future { nb_inversions_future_ij(i, m, seuil) }
  nb2 = nb_inversions_future_ij(m, j, seuil)

  nb1.value + nb2
end

#
```

```ruby
#
# Methode diviser pour regner "trichotomique"
# => decomposition en trois sous-problemes.
#
def nb_inversions_rec3( seuil )
  nb_inversions_rec_ij( 0, size-1, seuil )
end

def nb_inversions_rec3_ij( i, j, seuil )
  nb_elements = j - i + 1
  return nb_inversions_seq_ij(i, j) if nb_elements <= [seuil, 3]

  # On a necessairement 3 elements ou plus !
  i2 = i + nb_elements / 3
  i3 = i +  2 * nb_elements / 3

  nb1 = PRuby.future { nb_inversions_rec3_ij(i, i2-1, seuil) }
  nb2 = PRuby.future { nb_inversions_rec3_ij(i2-1, i3-1, seuil) }
  nb3 = nb_inversions_rec_ij(i3-1, j, seuil)

  nb1.value + nb2.value + nb3
end
```

2 Temps d'exécution obtenus avec `nb_inversions_bm.rb`

Note : Moyenne de trois exécutions

Temps sur `japet`

```
TAILLE=1000 ruby nb_inversions_bm.rb
[1000] (  1) seq:           :    0.001  1.00
[1000] (  1) rec            :    0.010  0.13
[1000] ( 10) rec            :    0.002  0.80
[1000] (100) rec            :    0.001  1.33
[1000] (500) rec            :    0.001  2.00
[1000] (  1) future         :    0.477  0.00
[1000] ( 10) future         :    0.038  0.04
[1000] (100) future         :    0.004  0.36
[1000] (500) future         :    0.001  2.00
[1000] (  1) pcall          :    0.744  0.00
[1000] ( 10) pcall          :    0.072  0.02
[1000] (100) pcall          :    0.006  0.24
[1000] (500) pcall          :    0.001  1.33

TAILLE=5000 ruby nb_inversions_bm.rb
[5000] (  1) seq:           :    0.003  1.00
[5000] (  1) rec            :    0.024  0.14
[5000] ( 10) rec            :    0.004  0.91
[5000] (100) rec            :    0.002  1.43
[5000] (500) rec            :    0.002  2.00
[5000] (  1) future         :    1.850  0.00
[5000] ( 10) future         :    0.139  0.02
[5000] (100) future         :    0.015  0.22
[5000] (500) future         :    0.004  0.83
[5000] (  1) pcall          :    4.114  0.00
[5000] ( 10) pcall          :    0.369  0.01
[5000] (100) pcall          :    0.031  0.11
[5000] (500) pcall          :    0.007  0.45
```

```
TAILLE=10000 ruby nb_inversions_bm.rb
[10000] (  1) seq:           :    0.006  1.00
[10000] (  1) rec            :    0.029  0.21
[10000] ( 10) rec            :    0.004  1.38
[10000] (100) rec            :    0.003  1.80
[10000] (500) rec            :    0.003  2.00
[10000] (  1) future         :    3.798  0.00
[10000] ( 10) future         :    0.365  0.02
[10000] (100) future         :    0.031  0.19
[10000] (500) future         :    0.008  0.78
NoMethodError: undefined method `+' for nil:NilClass
  nb_inversions_pcall_ij at /home/tremblay_gu/INF5171/Programmes/Ruby/Inversions
  nb_inversions_pcall_ij at /home/tremblay_gu/INF5171/Programmes/Ruby/Inversions
make: *** [bm] Erreur 1
```

Temps sur ma machine Linux

```
TAILLE=1000 ruby nb_inversions_bm.rb
[1000] (  1) seq:           :    0.003  1.00
[1000] (  1) rec            :    0.014  0.19
[1000] ( 10) rec            :    0.002  1.14
[1000] (100) rec            :    0.001  4.00
[1000] (500) rec            :    0.000  8.00
[1000] (  1) future         :    0.118  0.02
[1000] ( 10) future         :    0.011  0.24
[1000] (100) future         :    0.001  2.00
[1000] (500) future         :    0.000  8.00
[1000] (  1) pcall          :    0.270  0.01
[1000] ( 10) pcall          :    0.018  0.15
[1000] (100) pcall          :    0.001  2.00
[1000] (500) pcall          :    0.000   Inf

TAILLE=5000 ruby nb_inversions_bm.rb
[5000] (  1) seq:           :    0.002  1.00
[5000] (  1) rec            :    0.015  0.11
[5000] ( 10) rec            :    0.001  1.25
[5000] (100) rec            :    0.001  2.50
[5000] (500) rec            :    0.001  2.50
[5000] (  1) future         :    0.484  0.00
[5000] ( 10) future         :    0.018  0.09
[5000] (100) future         :    0.002  0.83
[5000] (500) future         :    0.000  5.00
[5000] (  1) pcall          :    0.790  0.00
[5000] ( 10) pcall          :    0.040  0.04
[5000] (100) pcall          :    0.005  0.33
[5000] (500) pcall          :    0.001  1.25
```

```
TAILLE=10000 ruby nb_inversions_bm.rb
[10000] (  1) seq:           :    0.008  1.00
[10000] (  1) rec            :    0.018  0.42
[10000] ( 10) rec            :    0.002  3.29
[10000] (100) rec            :    0.002  3.83
[10000] (500) rec            :    0.002  3.83
[10000] (  1) future         :    0.714  0.01
[10000] ( 10) future         :    0.062  0.12
[10000] (100) future         :    0.005  1.64
[10000] (500) future         :    0.001  5.75
ThreadError: unable to create new native thread
              initialize at org/jruby/RubyThread.java:441
                     new at org/jruby/RubyThread.java:390
            pcall_thread at /home/tremblay_g/.rvm/gems/jruby-1.7.25/gems/pruby-0
              each_index at org/jruby/RubyArray.java:1672
            pcall_thread at /home/tremblay_g/.rvm/gems/jruby-1.7.25/gems/pruby-0
                   pcall at /home/tremblay_g/.rvm/gems/jruby-1.7.25/gems/pruby-0
  nb_inversions_pcall_ij at /home/tremblay_g/INF5171/Programmes/Ruby/Inversions/
  nb_inversions_pcall_ij at /home/tremblay_g/INF5171/Programmes/Ruby/Inversions/
make: *** [bm] Erreur 1
```

3 Temps pour appeler une méthode vs. créer/activer un *thread* avec JRuby

Question : Si on compare le temps requis pour faire un appel de méthode direct vs. un appel via un *thread*, combien de fois plus long est l'appel via un *thread*?

```ruby
# Programme Ruby.
require 'benchmark'

N = ARGV[0] ? ARGV[0].to_i : 100_000

# Methode simple qu'on va appeler.
def inc( x )
  x + 1
end

# Methode qui execute un des appels qu'on veut mesurer.
def executer( bm, nom_methode )
  GC.start; GC.disable

  bm.report( nom_methode )do
    N.times { yield }
  end

  GC.enable
end

puts "Temps pour faire #{N} appels..."

Benchmark.bm(30) do |bm|
  x = 0
  executer bm, "d'une methode simple" do
    x = inc( x )
  end

  x = 0
  executer bm, "d'un Thread style future" do
    x = Thread.new { x + 1 }.value
  end
end
```

= MAC BOOK ==========

```
Ruby MRI
--------
Temps pour faire 100_000 appels...
                                      user     system      total        real
d'une methode simple              0.020000   0.000000   0.020000 (  0.014103)
d'un Thread style future
```

```
JRuby
-----
Temps pour faire 100_000 appels...
                                      user     system      total        real
d'une methode simple              0.200000   0.010000   0.210000 (  0.067000)
d'un Thread style future
```

= MAC BOOK ==========

Ruby MRI

```
--------
Temps pour faire 100_000 appels...
                                     user     system      total        real
d'une methode simple             0.020000   0.000000   0.020000 (  0.014103)
d'un Thread style future         1.290000   2.780000   4.070000 (  3.078120)
```

=> Thread 218 fois plus long que methode!?

JRuby

```
-----
Temps pour faire 100_000 appels...
                                     user     system      total        real
d'une methode simple             0.200000   0.010000   0.210000 (  0.067000)
d'un Thread style future        16.530000   9.610000  26.140000 ( 17.413000)
```

=> Thread 259 fois plus long que methode!?

```
Ruby MRI
--------
Temps pour faire 100_000 appels...
                                     user     system      total        real
d'une methode simple             0.020000   0.000000   0.020000 (  0.015892)
d'un Thread style future
```

```
JRuby
-----
Temps pour faire 100_000 appels...
                                     user     system      total        real
d'une methode simple             0.320000   0.010000   0.330000 (  0.048000)
d'un Thread style future
```

= JAPET ==========

```
Ruby MRI
--------
Temps pour faire 100_000 appels...
                                     user     system      total        real
d'une methode simple             0.020000   0.000000   0.020000 (  0.015892)
d'un Thread style future         2.350000   5.230000   7.580000 (  6.809706)
```

=> Thread 428 fois plus long que methode!?

```
JRuby
-----
Temps pour faire 100_000 appels...
                                     user     system      total        real
d'une methode simple             0.320000   0.010000   0.330000 (  0.048000)
d'un Thread style future       246.240000  19.970000 266.210000 ( 51.394000)
```

=> Thread 1000 fois plus long que methode!?

= Linux/CentOS ==========

```
Ruby MRI
--------
Temps pour faire 100_000 appels...
                                     user     system      total        real
d'une methode simple             0.010000   0.000000   0.010000 (0.007983)
d'un Thread style future
```

```
JRuby
-----
Temps pour faire 100_000 appels...
                                     user     system      total        real
d'une methode simple             0.200000   0.010000   0.210000 ( 0.058000)
d'un Thread style future
```

= Linux/CentOS ==========

```
Ruby MRI
--------
Temps pour faire 100_000 appels...
                                     user     system      total        real
d'une methode simple             0.010000   0.000000   0.010000 (0.007983)
d'un Thread style future         0.720000   0.850000   1.570000 (1.195668)
```

=> Thread 150 fois plus long que methode!?

```
JRuby
-----
Temps pour faire 100_000 appels...
                                     user     system      total        real
d'une methode simple             0.200000   0.010000   0.210000 ( 0.058000)
d'un Thread style future        10.460000   1.990000  12.450000 ( 5.559000)
```

=> Thread 96 fois plus long que methode!?

4 Temps pour appeler une méthode vs. créer/activer un *thread* avec Java

```
class Compteur {
  private int x;

  void reset() { x = 0; }
  void inc() { x += 1; }
}

...
Compteur o = new Compteur();

// Appels de methodes.
tempsDebut = System.nanoTime();
o.reset();
for ( int i = 0; i < N; i++ ) {
  o.inc();
}
tempsFin = System.nanoTime();
System.out.format( ... );

// Creation de threads.
tempsDebut = System.nanoTime();
o.reset();
for ( int i = 0; i < N; i++ ) {
  Thread t = new Thread( () -> { o.inc(); } );
  t.start();
  try { t.join(); } catch( Exception e ){};
}
tempsFin = System.nanoTime();
System.out.format( ... );
```

= Linux/CentOS ==========

```
$ javac -Xlint:unchecked -classpath . MethodeVsThread.java

$ java -ea -classpath . MethodeVsThread 100000
Temps pour 100000 appels de methodes (ms)
     2,4
Temps pour 100000 creation/activation de threads (ms)
  2235,2
```

= JAPET ==========

```
$ javac -Xlint:unchecked -classpath . MethodeVsThread.java

$ java -ea -classpath . MethodeVsThread 100000
Temps pour 100000 appels de methodes (ms)
     4,9
Temps pour 100000 creation/activation de threads (ms)
 10471,1
```