Solution devoir # 1

1 Fichier `classificateur.rb`

1.1 Méthodes `classifier_les_points`

```
def classifier_les_points_seq( points, representants )
  points.each do |p|
    p.groupe = groupe_le_plus_pres( p, representants )
  end
end

def classifier_les_points_par_fj_fin( points, representants )
  PRuby.pcall( 0...nb_points,
               lambda do |k|
                 points[k].groupe = groupe_le_plus_pres( points[k], representant
               end
               )
end


def classifier_les_points_par_fj_adj( points, representants )
  executer_par_fj_adj(nb_points) do |k|
    points[k].groupe = groupe_le_plus_pres( points[k], representants )
  end
end


def classifier_les_points_par_fj_cyc( points, representants )
  executer_par_fj_cyc(nb_points) do |k|
    points[k].groupe = groupe_le_plus_pres( points[k], representants )
  end
end
```

```ruby
def classifier_les_points_par_sta( points, representants )
  points.peach( static: @taille_tache ) do |p|
    p.groupe = groupe_le_plus_pres( p, representants )
  end
end


def classifier_les_points_par_dyn( points, representants )
  points.peach( dynamic: @taille_tache ) do |p|
    p.groupe = groupe_le_plus_pres( p, representants )
  end
end


#
```

```ruby
#
# Trouve le (numero du) groupe le plus pres d'un point en fonction
# des representants des divers groupes.
#
# @param [Point] point le point a analyser
# @param [Array<Vector>] representants les representants des divers groupes
# @return [Fixnum] le numero du groupe le plus pres
#
def groupe_le_plus_pres( point, representants )
  dist_0 = point.distance( representants[0] )

  (1...representants.size).reduce( [0, dist_0] ) do |g_et_dist_min, r|
    g_min, dist_min = g_et_dist_min
    dist = point.distance( representants[r] )
    if dist < dist_min
      [r, dist]
    else
      g_et_dist_min
    end
  end
    .first
end
```

1.2 Méthodes nouveaux_representants

```ruby
def nouveaux_representants_seq( nb_groupes, points )
  (0...nb_groupes).map do |gr|
    representant_pour_groupe( gr, points )
  end
end


def nouveaux_representants_par_fj_fin( nb_groupes, points )
  (0...nb_groupes)
    .map { |gr| PRuby.future { representant_pour_groupe( gr, points ) } }
    .map(&:value)
end

def nouveaux_representants_par_fj_adj( nb_groupes, points )
  representants = Array.new(nb_groupes)

  executer_par_fj_adj(nb_groupes) do |gr|
    representants[gr] = representant_pour_groupe( gr, points )
  end

  representants
end


def nouveaux_representants_par_fj_cyc( nb_groupes, points )
  representants = Array.new(nb_groupes)

  executer_par_fj_cyc(nb_groupes) do |gr|
    representants[gr] = representant_pour_groupe( gr, points )
  end

  representants
end

#
```

```ruby
def nouveaux_representants_par_sta( nb_groupes, points )
  (0...nb_groupes).pmap( static: @taille_tache ) do |gr|
    representant_pour_groupe( gr, points )
  end
end


def nouveaux_representants_par_dyn( nb_groupes, points )
  (0...nb_groupes).pmap( dynamic: @taille_tache ) do |gr|
    representant_pour_groupe( gr, points )
  end
end

#
```

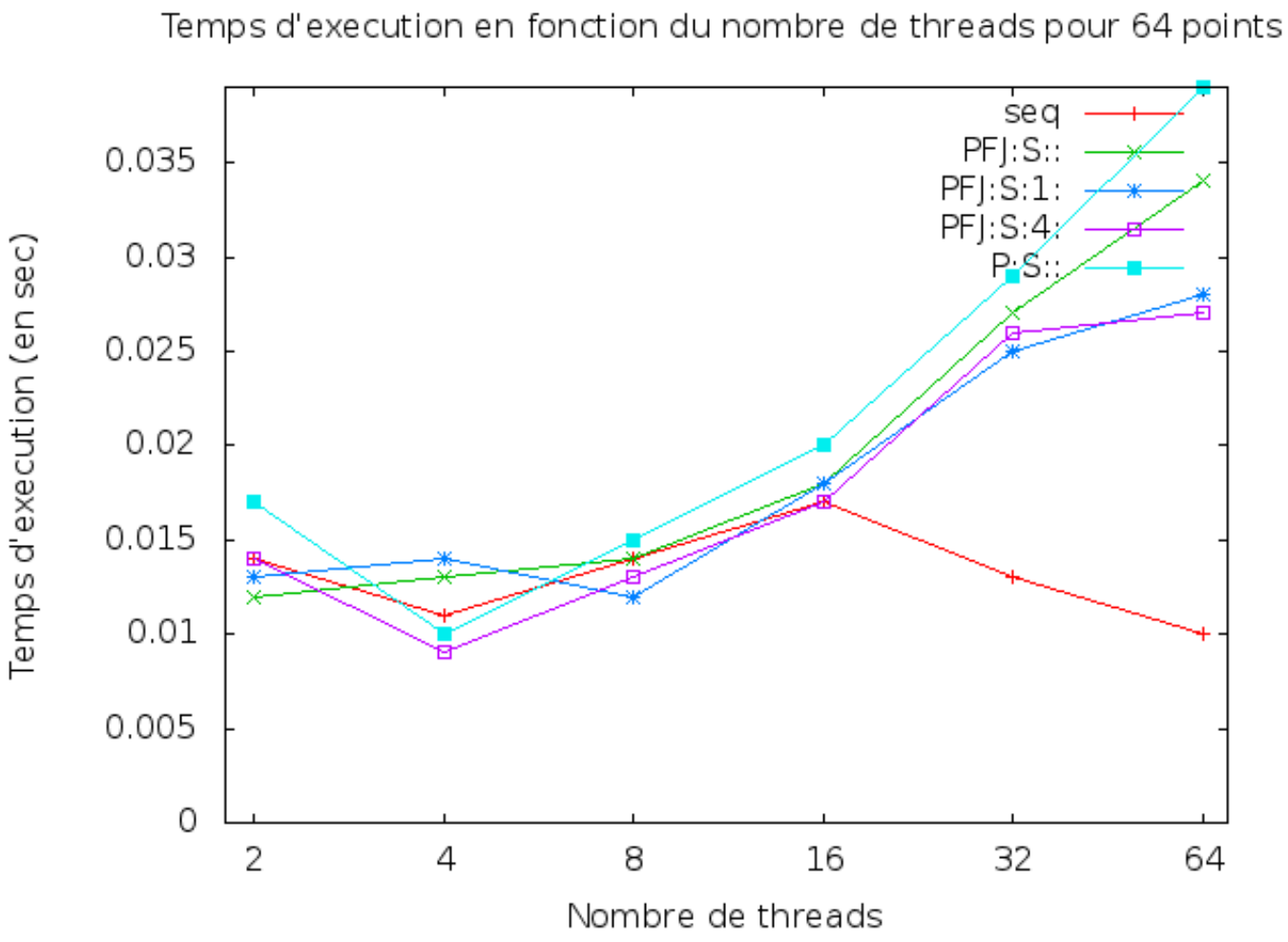
```ruby
# Calcule la position moyenne des points qui sont dans le groupe gr.
def representant_pour_groupe( gr, points )
  taille = points.first.position.size
  zerov = Vector.elements( Array.new(taille) { 0.0 } ) # Vecteur zero approprie.

  nb_points, somme = points.reduce( [0, zerov] ) do |nb_et_somme, p|
    if p.groupe == gr
      nb, somme = nb_et_somme
      [nb + 1, somme + p.position]
    else
      nb_et_somme
    end
  end

  somme / nb_points
end
```

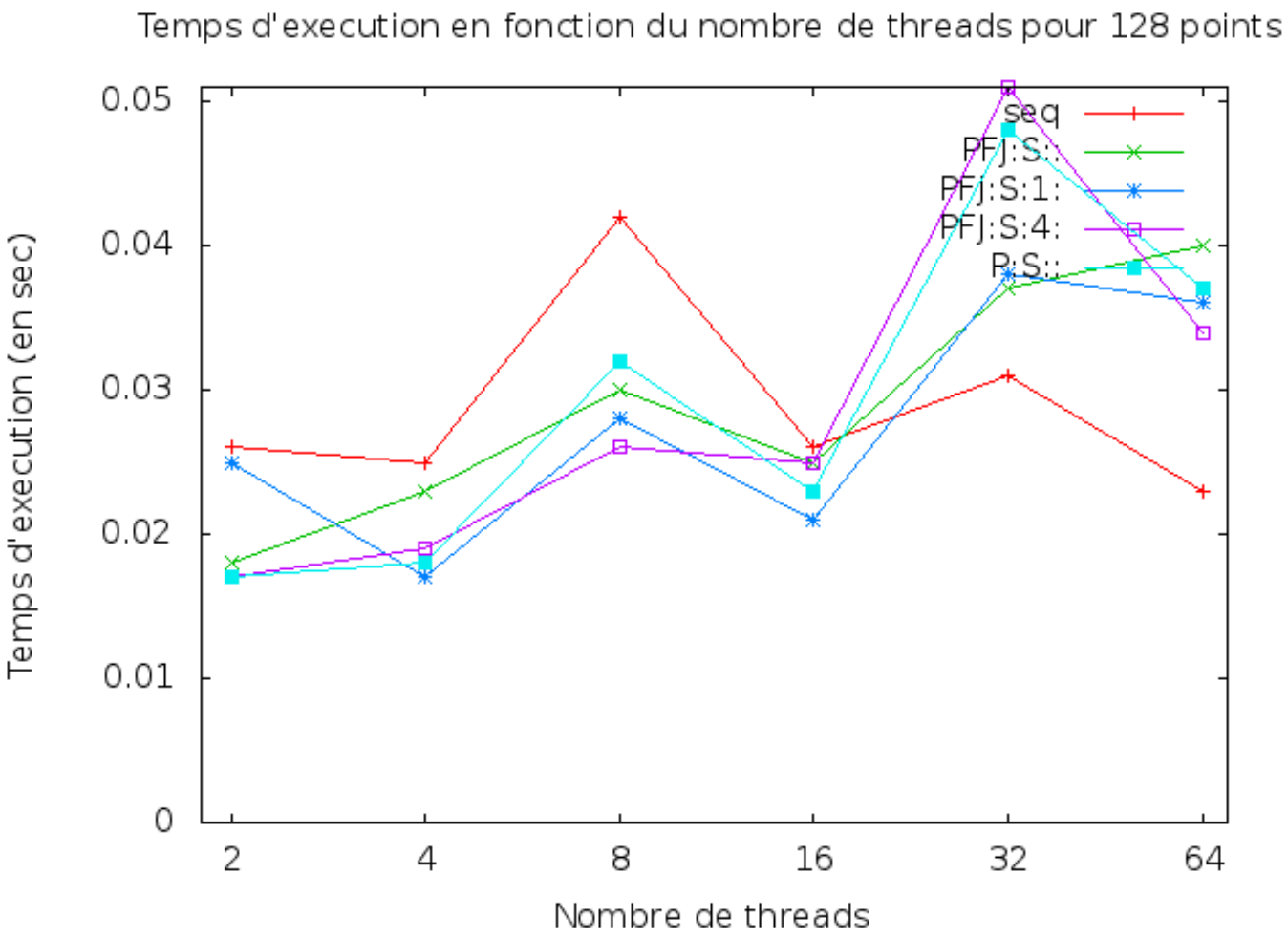
1.3 Autres méthodes auxiliaires, utilisées par les autres méthodes précédentes

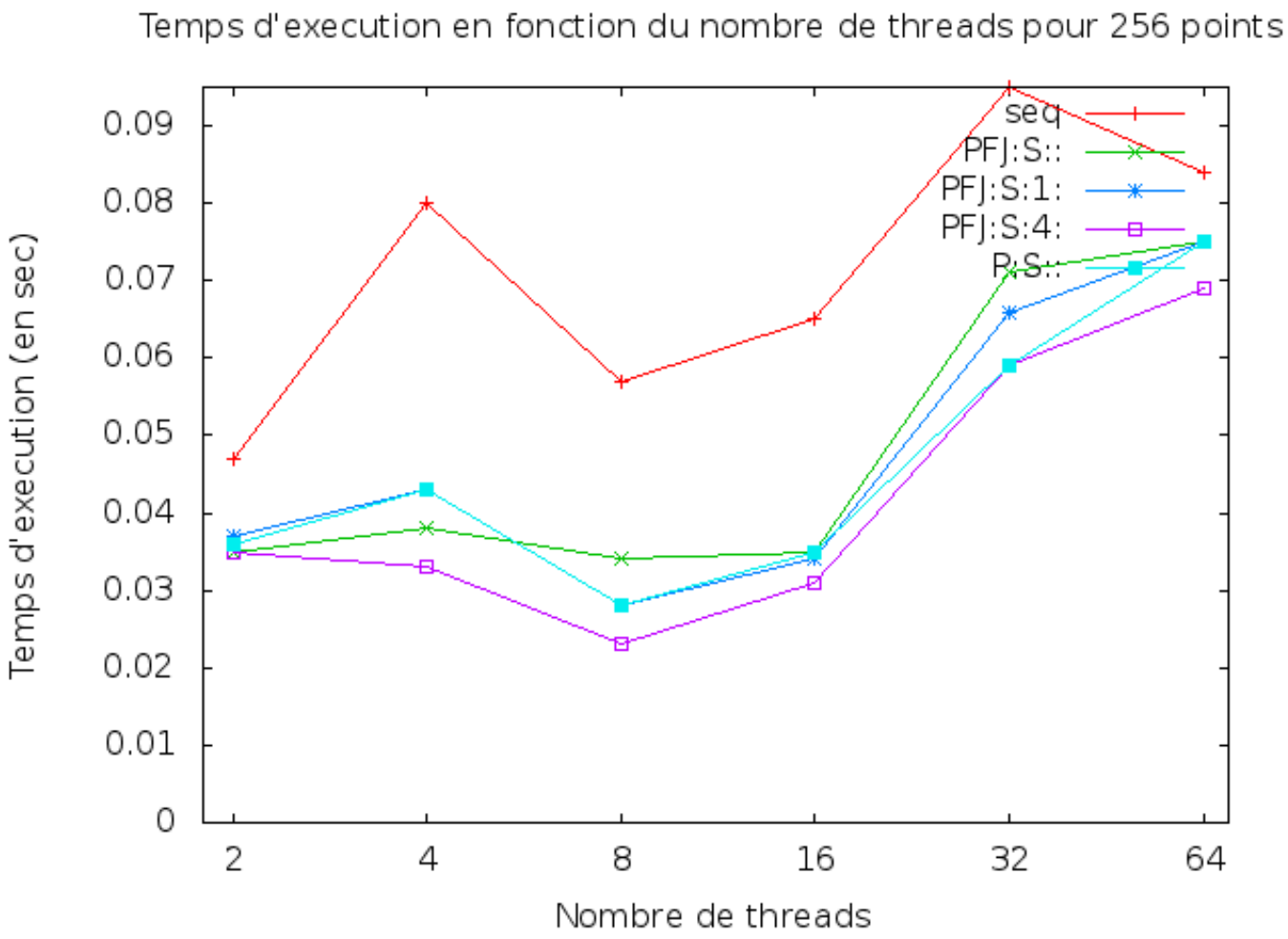
```
#
# Execute un bloc de code de facon parallele via une distribution
# par bloc d'elements adjacents.
#
def executer_par_fj_adj( nb_elements )
  nb_thr = [nb_threads, nb_elements].min

  PRuby.pcall( 0...nb_thr,
               lambda do |num_thread|
                 bornes_de_tranche(num_thread, nb_elements, nb_thr).each do |k
                   yield( k )
                 end
               end
               )
end

#
# Execute un bloc de code de facon parallele via une distribution
# cyclique par bloc de taille_tache elements.
#
def executer_par_fj_cyc( nb_elements )
  nb_thr = [nb_threads, nb_elements].min
  incr = nb_thr * taille_tache

  PRuby.pcall( 0...nb_thr,
               lambda do |num_thread|
                 premier_index = num_thread * taille_tache
                 (premier_index...nb_elements).step(incr).each do |b_inf|
                   b_sup = [b_inf+taille_tache-1,nb_elements-1].min
                   (b_inf..b_sup).each do |k|
                     yield( k )
                   end
                 end
               end
               )
end
```

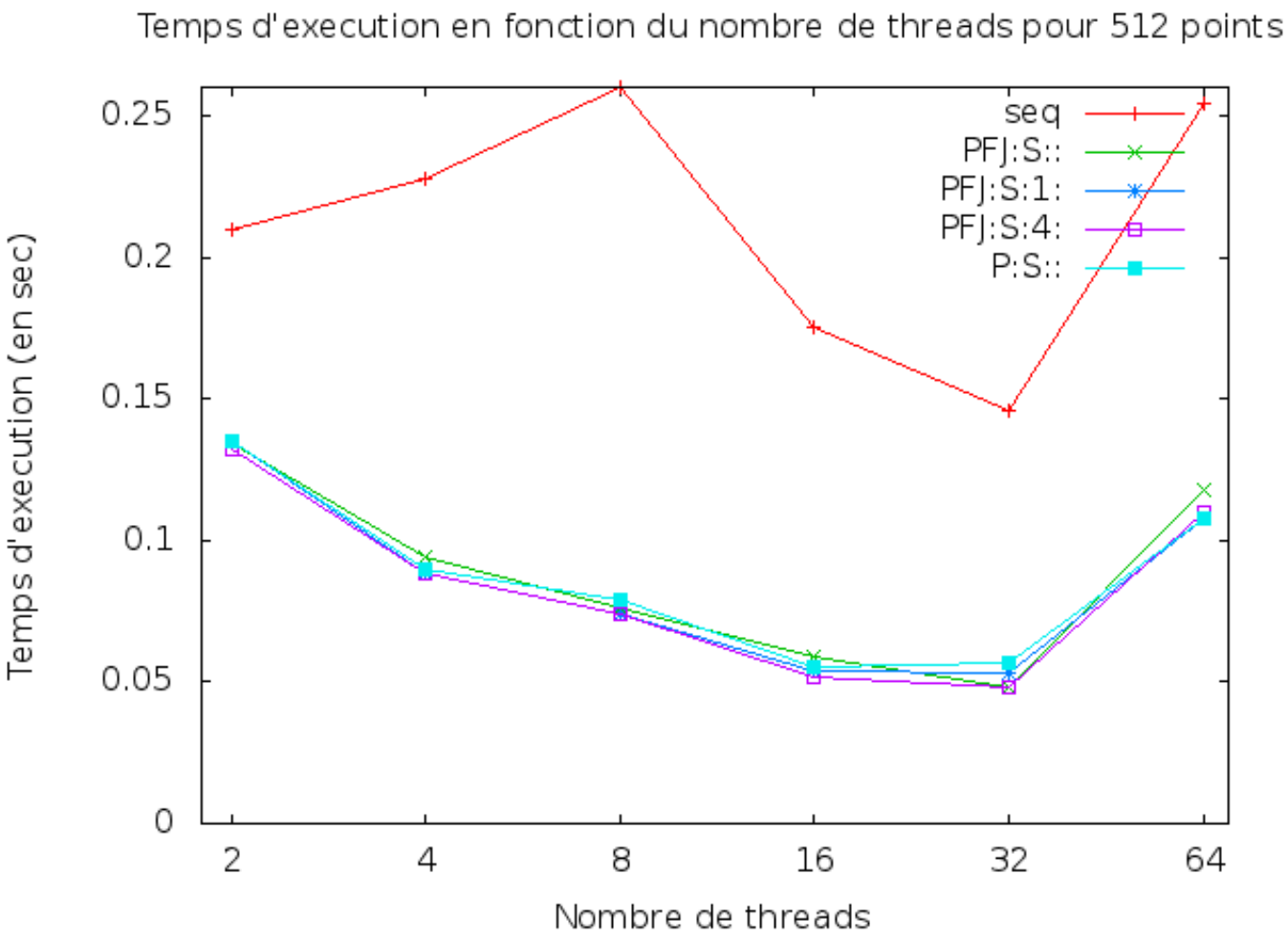
```
#
# Determine les bornes qui vont *uniformiser* le plus possible la
# repartition des elements quand le nombre n'est pas divisible de
# facon exacte par le nombre de threads.
#
def bornes_de_tranche( num_thread, n, nb_threads )
  nb_min_par_thread = n / nb_threads
  nb_en_trop_a_distribuer = n % nb_threads

  b_inf = num_thread * nb_min_par_thread + [num_thread, nb_en_trop_a_distribue
  b_sup = (num_thread+1) * nb_min_par_thread + [num_thread+1, nb_en_trop_a_dis

  b_inf..b_sup
end
```

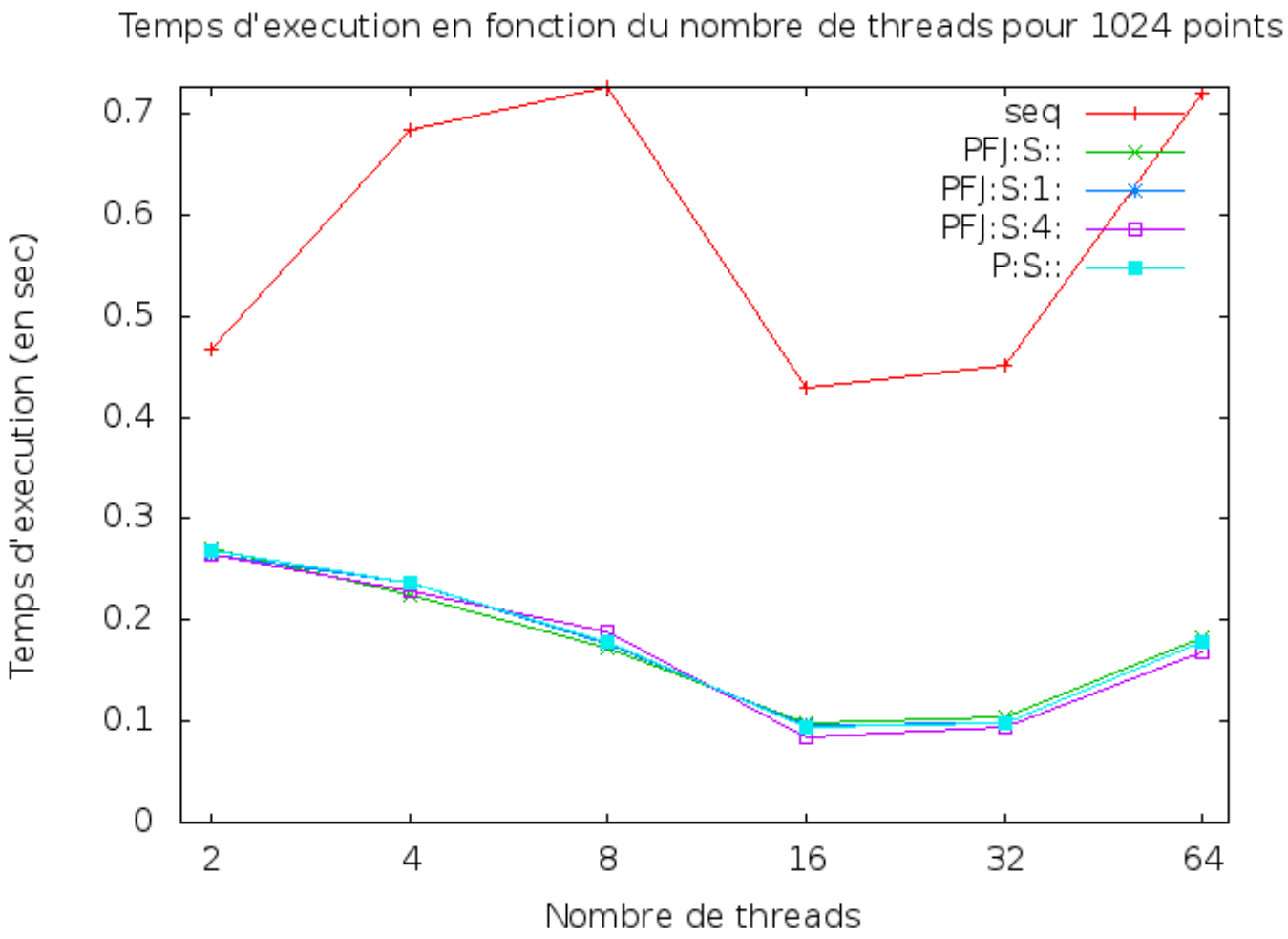
2 Graphes de temps d'exécution

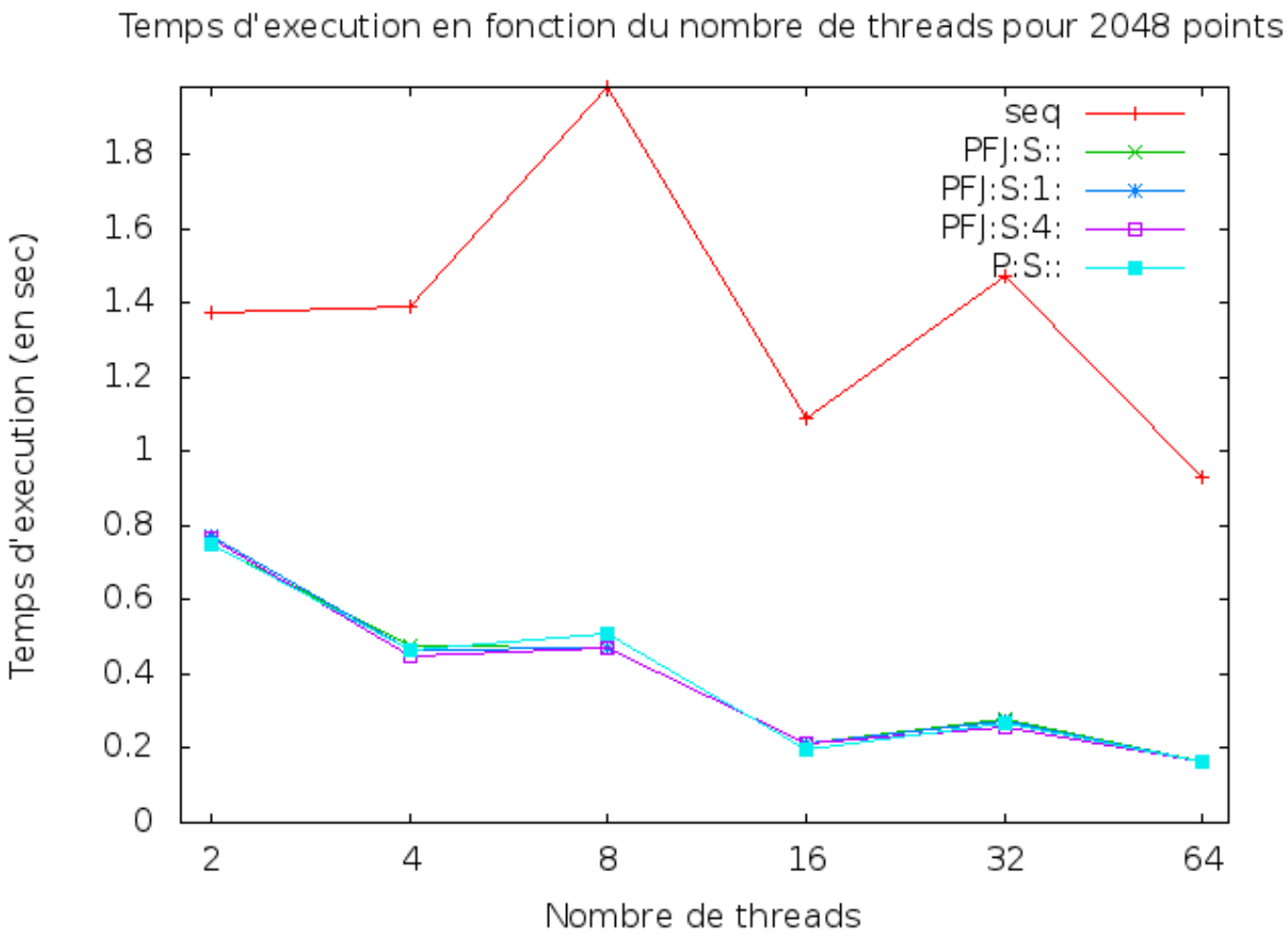



Temps d'execution en fonction du nombre de threads pour 128 points
Temps d'execution (en sec)
0.05
0.04
0.03
0.02
0.01
0
2
4
8
16
32
64
Nombre de threads
seq
PFJ:S:
PFJ:S:1:
PFJ:S:4:
P:S:

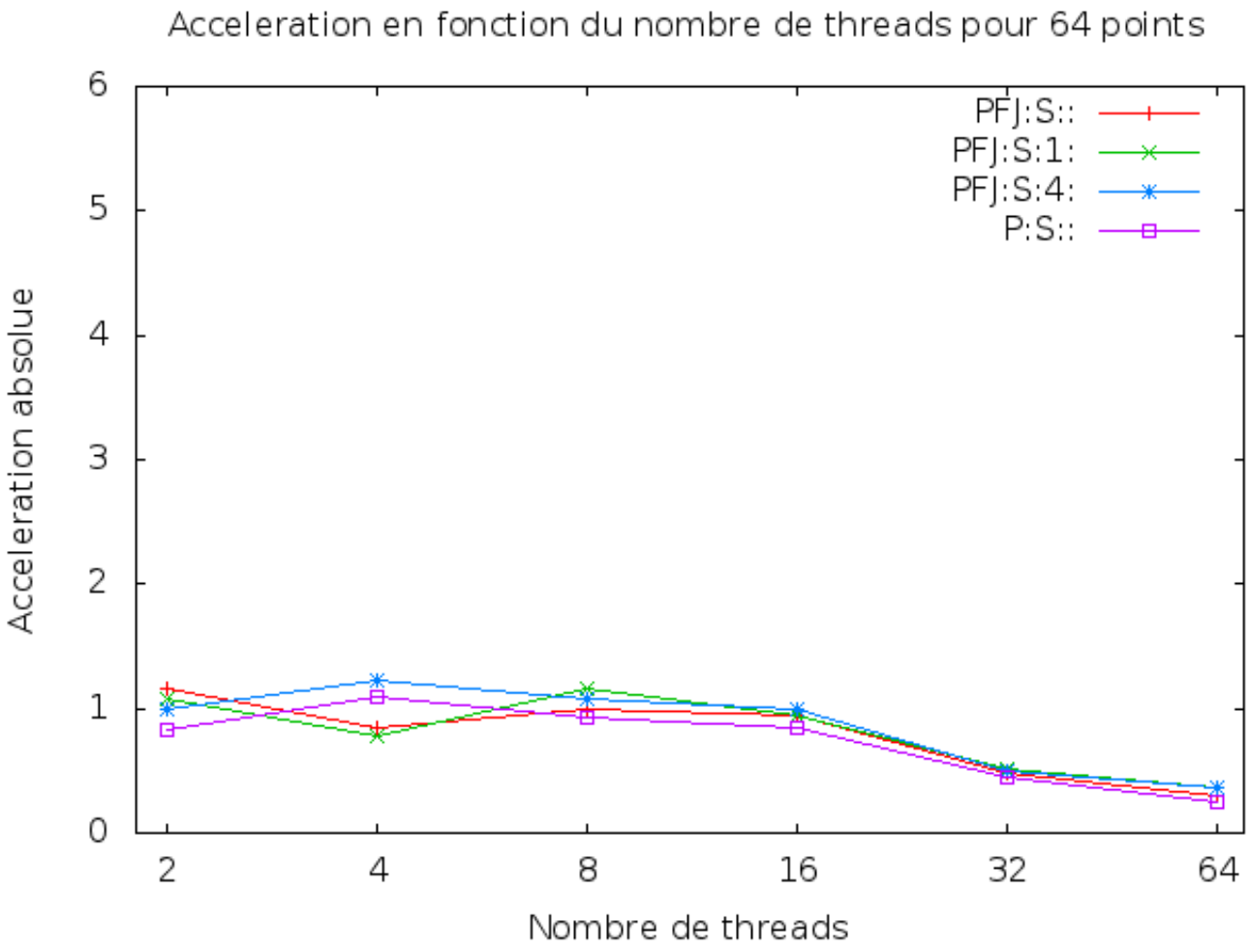

Temps d'execution en fonction du nombre de threads pour 256 points
Temps d'execution (en sec)
0
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0.09
2
4
8
16
32
64
Nombre de threads
seq
PFJ:S::
PFJ:S:1:
PFJ:S:4:
P:S::

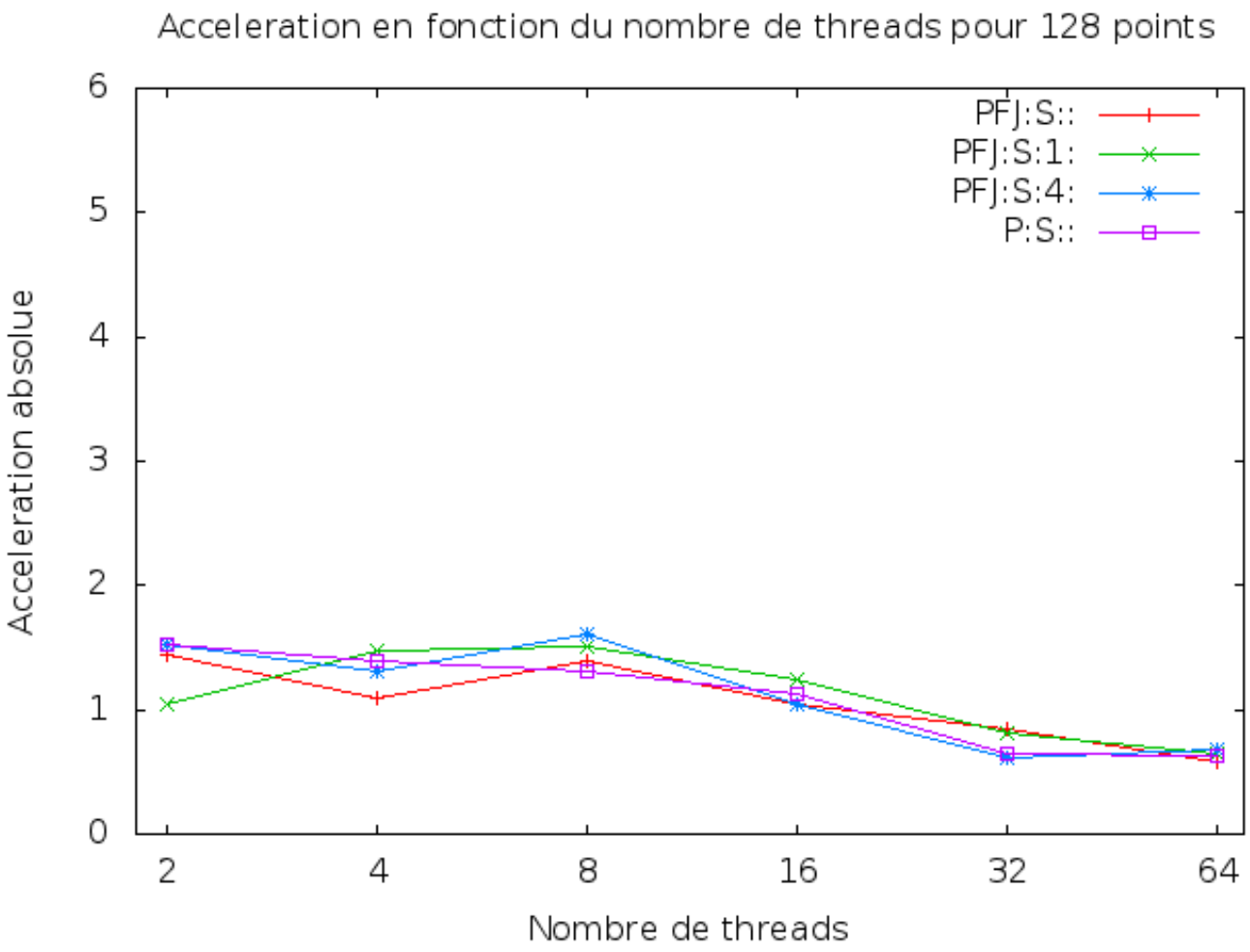

Temps d'execution en fonction du nombre de threads pour 512 points
Temps d'execution (en sec)
0.25
0.2
0.15
0.1
0.05
0
2
4
8
16
32
64
Nombre de threads
seq
PFJ:S::
PFJ:S:1:
PFJ:S:4:
P:S::

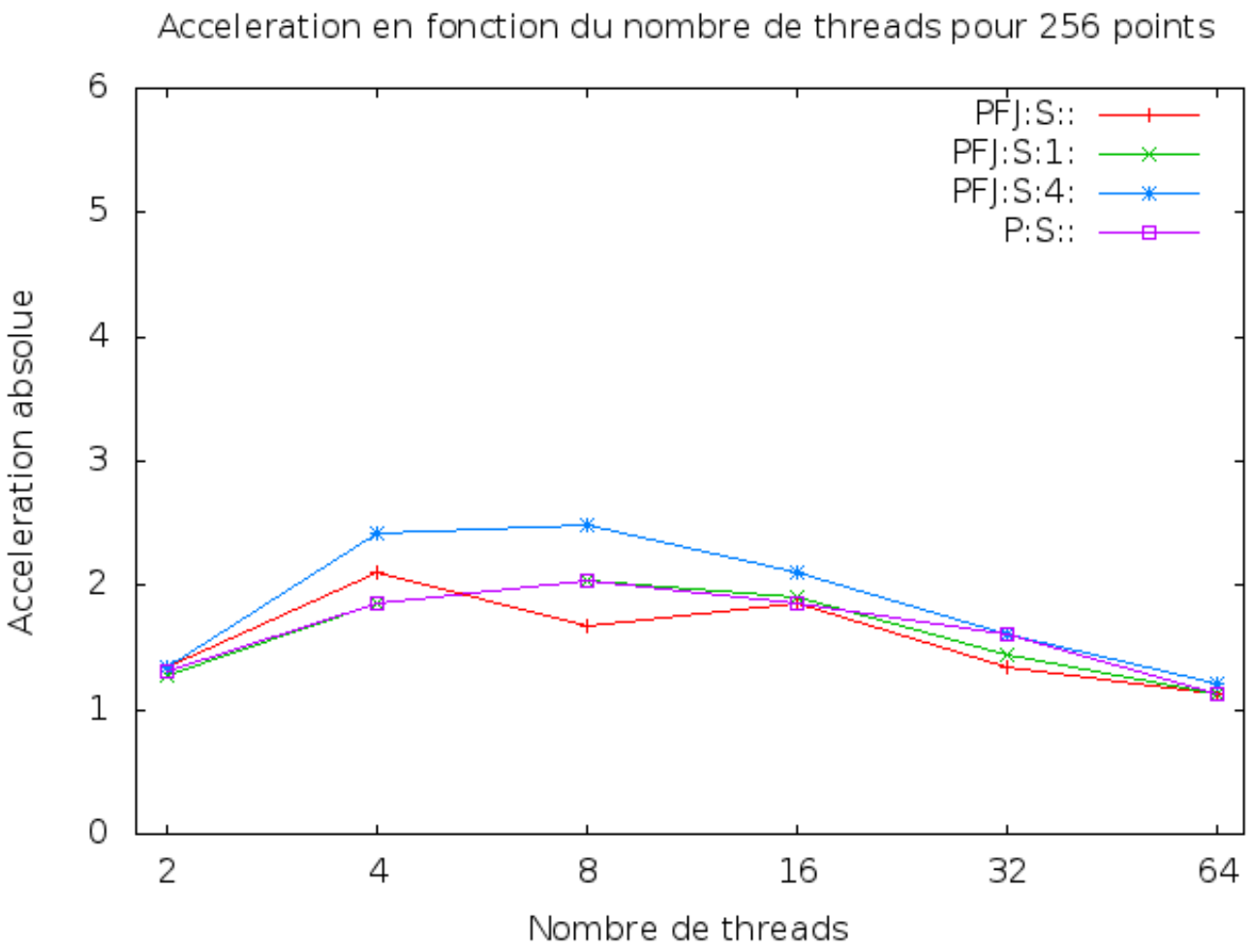

Temps d'execution en fonction du nombre de threads pour 1024 points
Temps d'execution (en sec)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
2
4
8
16
32
64
Nombre de threads
seq
PFJ:S::
PFJ:S:1:
PFJ:S:4:
P:S::

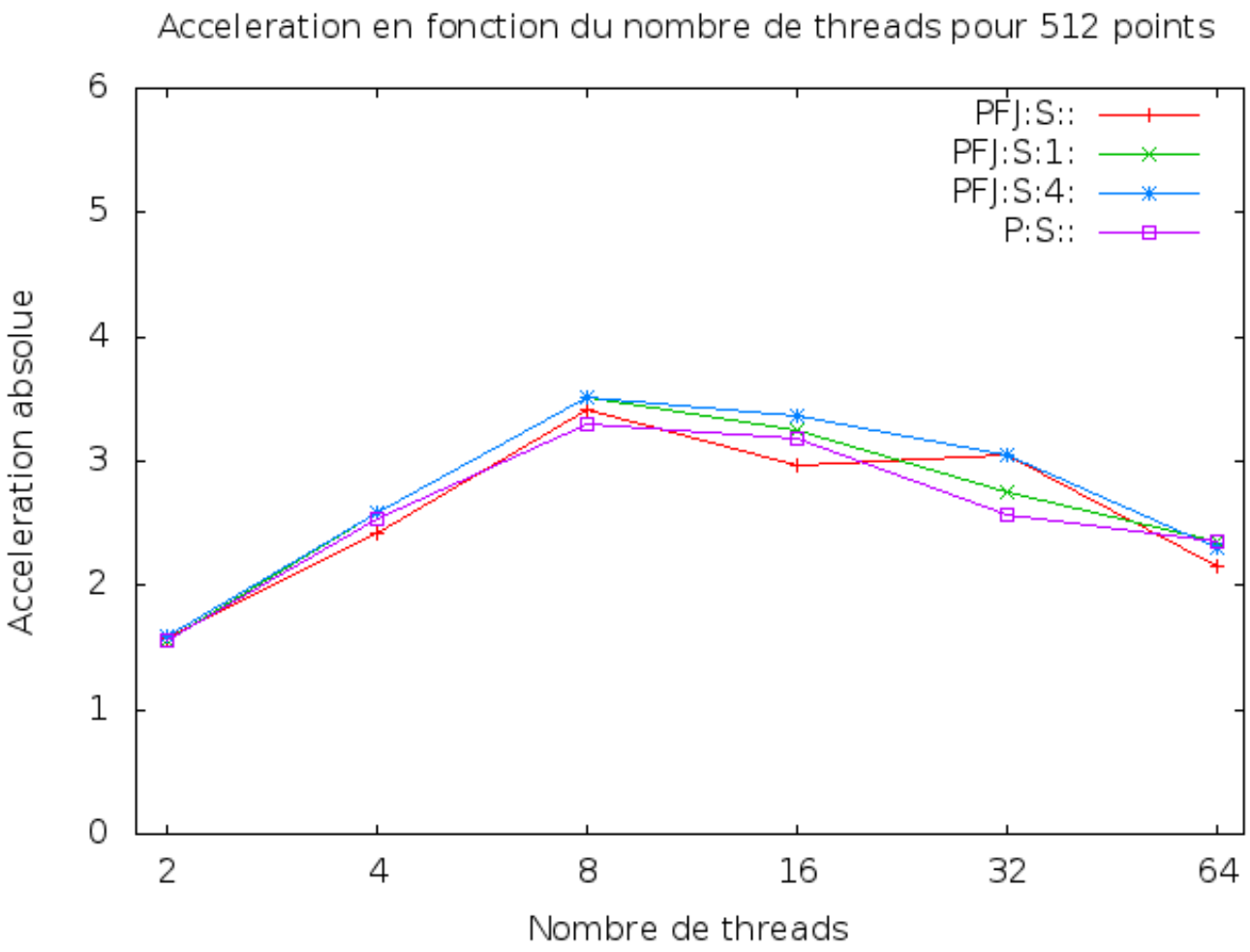

Temps d'execution en fonction du nombre de threads pour 2048 points
Temps d'execution (en sec)
Nombre de threads
seq
PFJ:S::
PFJ:S:1:
PFJ:S:4:
P:S::
0
0.2
0.4
0.6
0.8
1
1.2
1.4
1.6
1.8
2
4
8
16
32
64

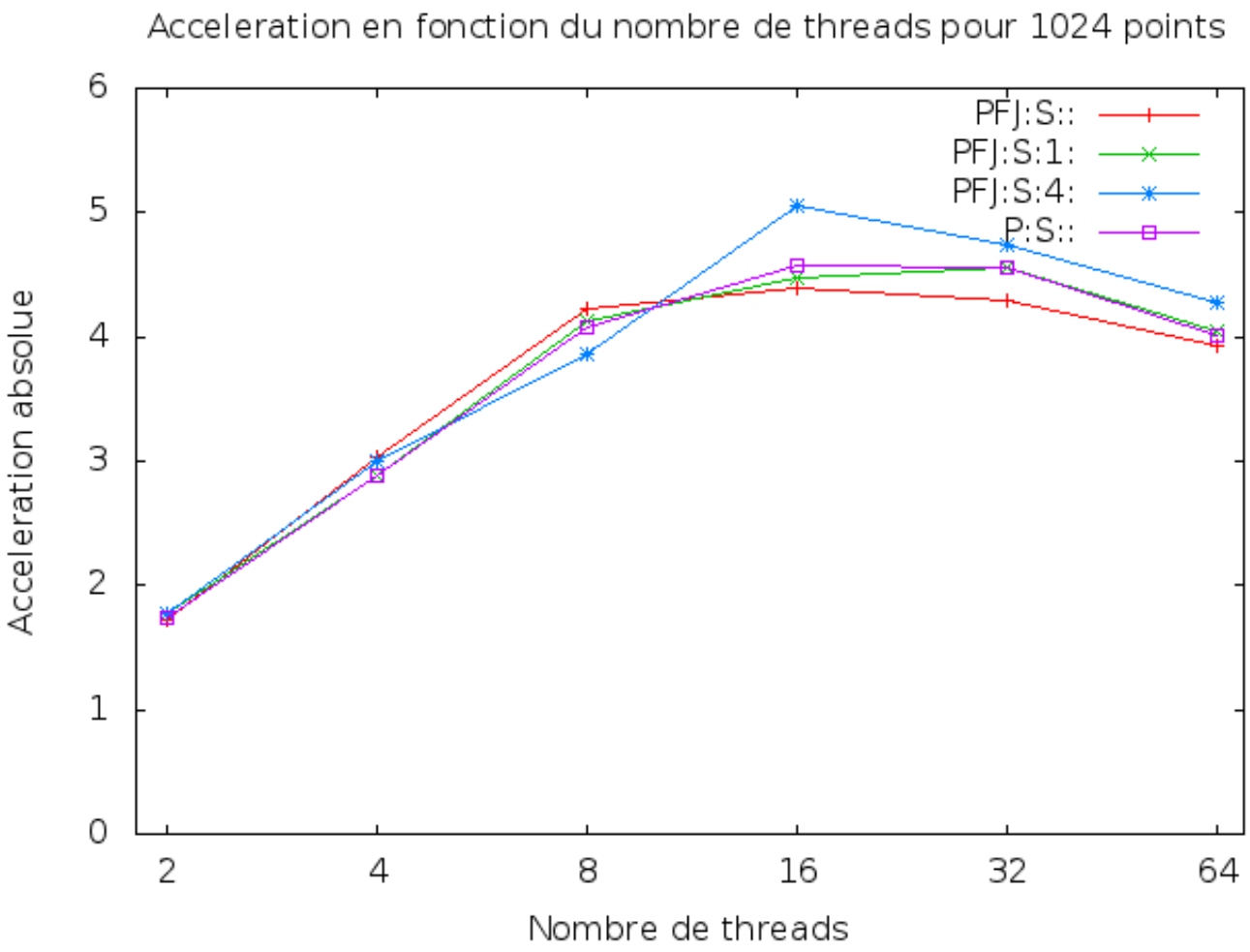
3 Graphes d'accélération

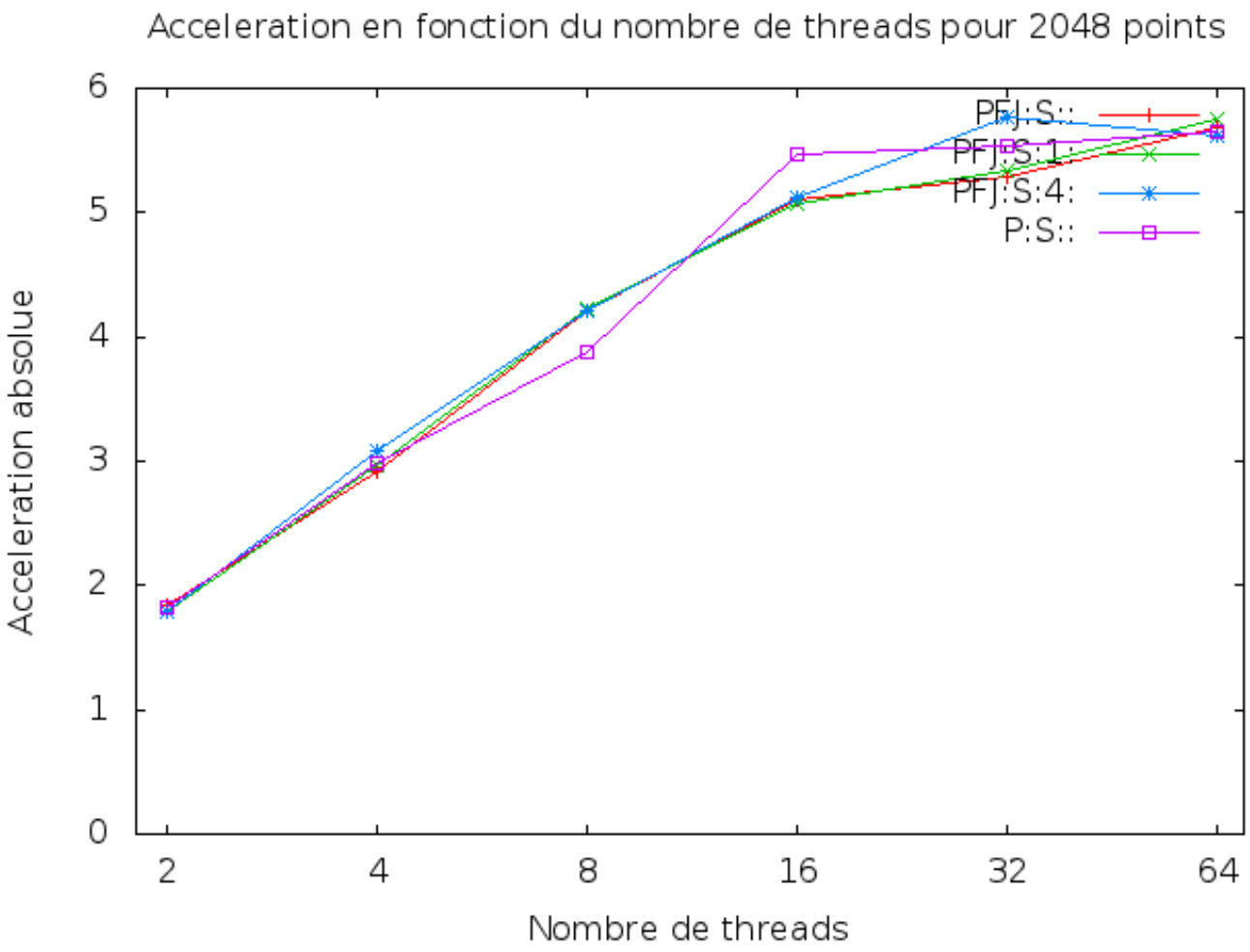



Acceleration en fonction du nombre de threads pour 128 points
Acceleration absolue
Nombre de threads
PFJ:S::
PFJ:S:1:
PFJ:S:4:
P:S::
0
1
2
3
4
5
6
2
4
8
16
32
64


Acceleration en fonction du nombre de threads pour 256 points
Acceleration absolue
Nombre de threads
PFJ:S::
PFJ:S:1:
PFJ:S:4:
P:S::
0
1
2
3
4
5
6
2
4
8
16
32
64


Acceleration en fonction du nombre de threads pour 512 points
Acceleration absolue
Nombre de threads
PFJ:S::
PFJ:S:1:
PFJ:S:4:
P:S::
0
1
2
3
4
5
6
2
4
8
16
32
64


Acceleration en fonction du nombre de threads pour 1024 points
Acceleration absolue
Nombre de threads
PFJ:S::
PFJ:S:1:
PFJ:S:4:
P:S::
0
1
2
3
4
5
6
2
4
8
16
32
64


Acceleration en fonction du nombre de threads pour 2048 points
Acceleration absolue
Nombre de threads
0
1
2
3
4
5
6
2
4
8
16
32
64
PFJ:S::
PFJ:S:1:
PFJ:S:4:
P:S::