3. Concepts de base de la concurrence et du parallélisme

3.1 Processus vs. *thread*

Tant les processus que les *threads* représentent des **instances de code en cours d'exécution** — donc une notion **dynamique**.

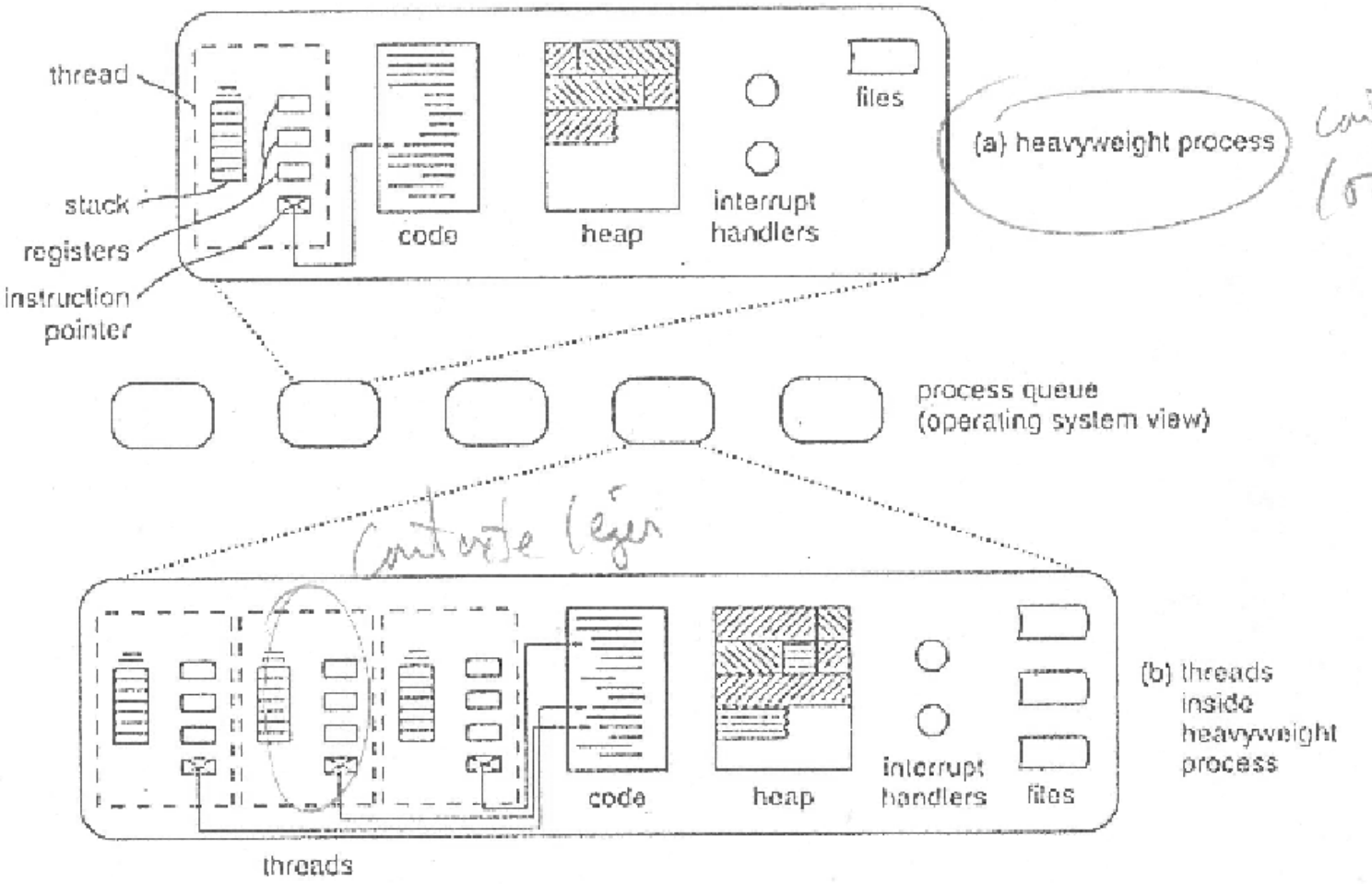


Figure 4.1
Process Organization

Figure 3.1: Processus vs. *thread* dans l'environnement Unix (source inconnue).

Mais : Certains langages — par ex., Erlang, Elixir, Go — supportent une forme de **processus légers**.

Ces processus possèdent un espace strictement privé, et communiquent par l'intermédiaire de messages.

3.2 Concurrence vs. parallélisme

In programming, concurrency is the composition of independently executing processes, while parallelism is the simultaneous *execution of (possibly related) computations.*

Concurrency *is about* **dealing** *with lots of things at once.*

Parallelism *is about* **doing** *lots of things at once.*

Source : Rob Pike, `http://blog.golang.org/concurrency-is-not-parallelism`

– **Programme séquentiel**

= Un programme qui comporte *un seul fil d'exécution* — un seul «*thread*»

⇒ Un seul doigt suffit pour indiquer l'instruction en cours d'exécution ☺

– **Programme concurrent**

= Un programme qui contient *deux ou plusieurs threads* qui *coopèrent*

⇒ Plusieurs doigts sont nécessaires pour indiquer les instructions en cours d'exécution ☺

Coopération $\Rightarrow$ Communication, échange d'information

Deux principales façons de communiquer :

- variables *partagées — principalement dans le cas de* threads
- échange de messages

Est-ce vrai qu'un seul doigt suffit pour indiquer à quel endroit on est rendu dans l'exécution d'un programme séquentiel?

Exercice 3.1: Un seul doigt suffit-il vraiment?

Différents types d'applications **concurrentes**

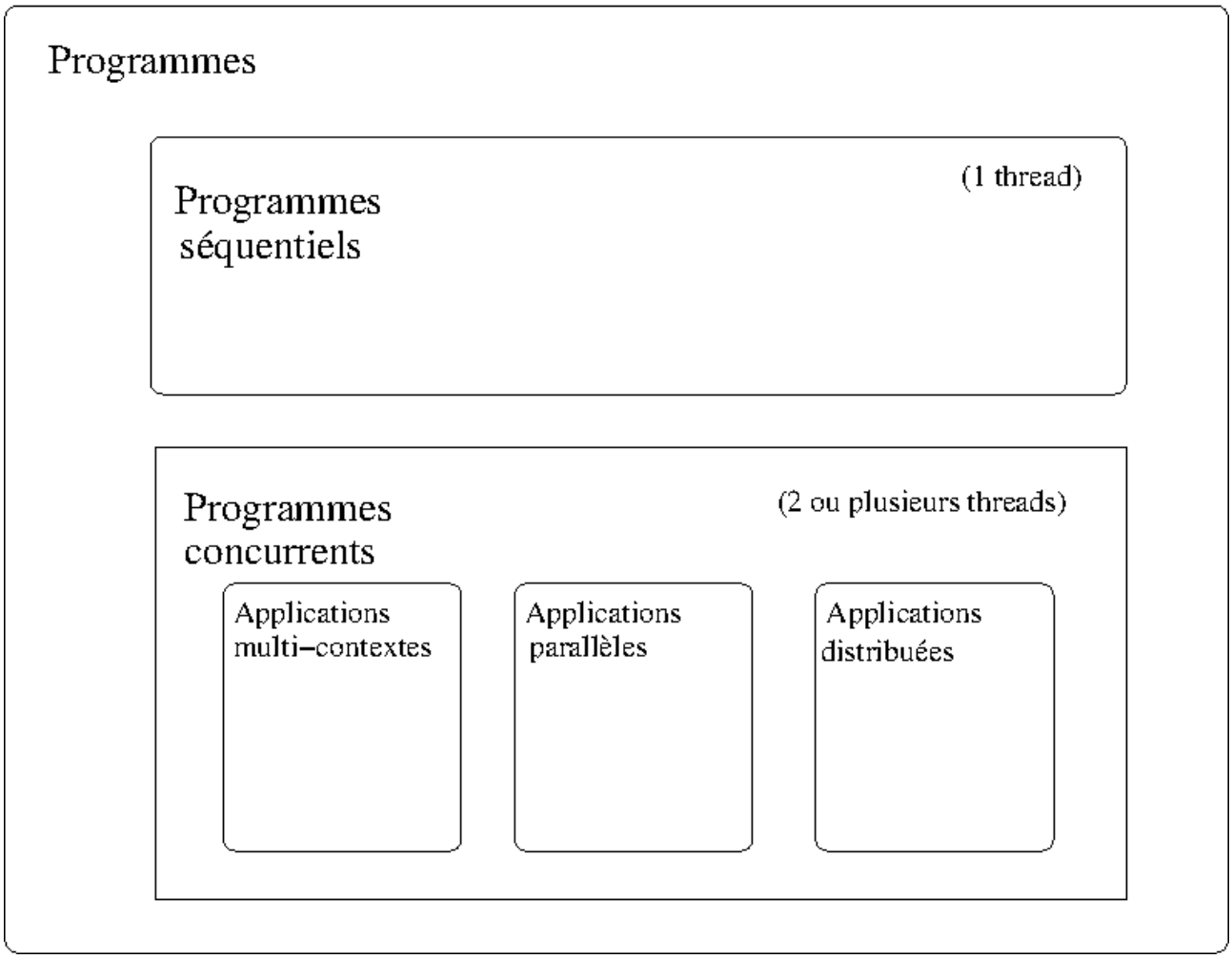


Figure 3.2: Programmes vs. programmes concurrents vs. applications multi-contextes, parallèles ou distribuées.

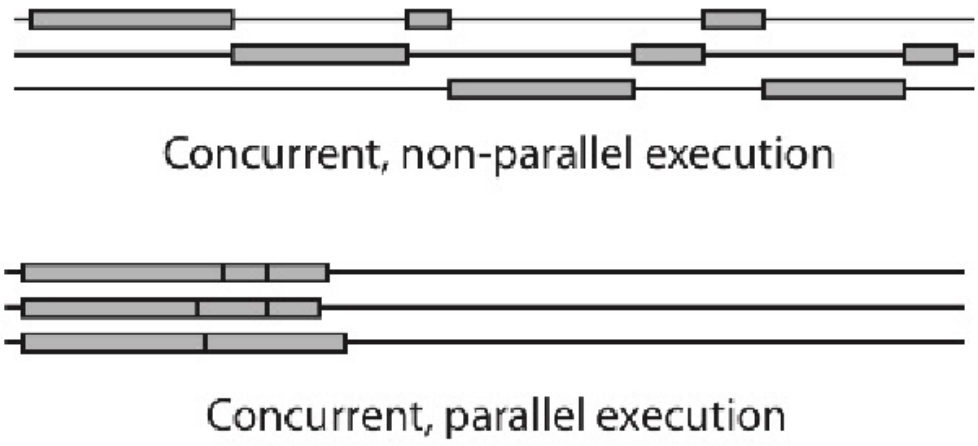


Figure 3.3: Exécution concurrente non parallèle vs. exécution concurrente et parallèle de throis *threads* : tiré de .

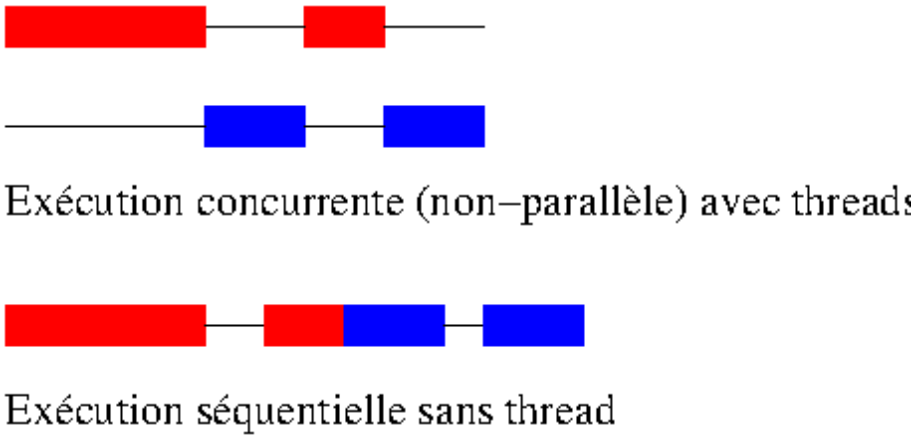


Figure 3.4: Exécution d'un programme effectuant des entrées/sortie de façon concurrente avec *threads* vs. de façon séquentielle sans *thread*.

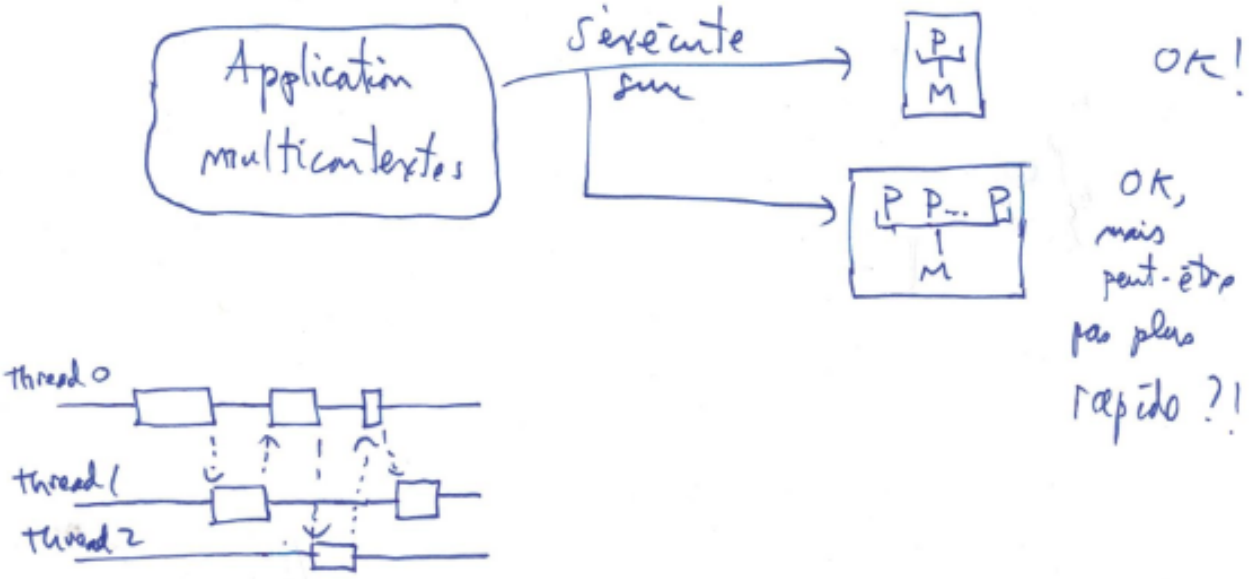


Figure 3.5: Exécution d'une application multicontextes sur une machine séquentielle vs. une machine parallèle.

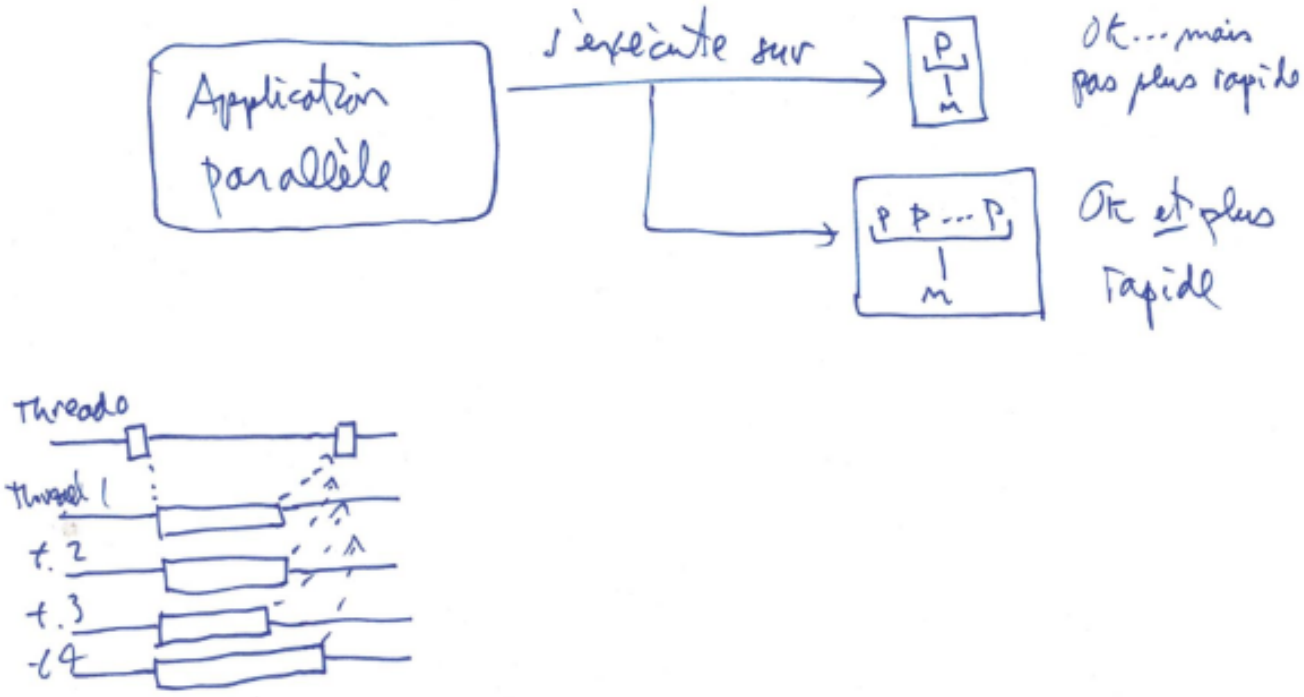


Figure 3.6: Exécution d'une application parallèle sur une machine séquentielle vs. une machine parallèle.

Autre caractérisation pour distinguer concurrence et parallélisme : *CPU-bound* vs. *IO-bound*

- un programme est *CPU-bound* si son temps d'exécution est limité par la vitesse du CPU.

  Il passe la majeure partie de son temps à utiliser le CPU donc **il s'exécuterait plus rapidement si le CPU était plus rapide**.

- un programme est *IO-bound* si son temps d'exécution est limité par la vitesse du système d'entrées/sorties.

  Il passe la majeure partie de son temps d'exécution à utiliser les E/S donc **il s'exécuterait plus rapidement si le système d'E/S était plus rapide**.

Soit deux machines multi-coeurs M_1 et M_8, ayant des configurations semblables sauf pour le nombre de coeurs : un seul (1) coeur pour M_1 et huit (8) coeurs pour M_8.

1. Soit un programme P_1 qui est *IO-bound* et qui s'exécute sur M_1 en 1.6 secondes.

   Si on exécute P_1 sur M_8, quel temps d'exécution peut-on s'attendre à obtenir?

2. Soit un programme P_2 qui est *CPU-bound* et qui s'exécute sur M_1 est 1.6 secondes.

   Si on exécute P_2 sur M_8, quel temps d'exécution peut-on s'attendre à obtenir?

Exercice 3.2: Effet d'ajouter des coeurs sur des programmes *CPU-bound* ou *IO-bound*.

3.3 Contrôle de la concurrence

3.3.1 État d'un *thread*, actions atomiques et entrelacement des instructions atomiques

Quelques définitions :

- L'état d'un *thread* est caractérisé par les variables du *thread*.

- Action atomique = examine ou change l'état d'un *thread* **de façon *indivisible***.

- Un *thread* exécute une séquence d'instructions. Chaque instruction = **une ou plusieurs *actions atomiques***.

- L'effet d'un programme est obtenu par l'entrelacement des actions **atomiques** des *thread*s.

Programme Ruby 3.1 Petit programme Ruby avec deux *thread*s simples.

```
PRuby.pcall(
  -> { puts "Thr1: ."
       puts "Thr1: .."
       puts "Thr1: ..."
     },
  -> { puts "Thr2: +"
       puts "Thr2: ++"
       puts "Thr2: +++"
     }
  )

puts "Fin du programme"
```

«Typiquement», l'exécution de ce programme produira le résultat suivant :

```
$ ruby entrelacement.rb
Thr1: .
Thr1: ..
Thr1: ...
Thr2: +
Thr2: ++
Thr2: +++
Fin du programme
```

Toutefois, il est aussi possible que ce programme produise plutôt...

```
$ ruby entrelacement.rb
Thr1: .Thr2: +

Thr1: ..
Thr1: ...
Thr2: ++
Thr2: +++
Fin du programme
```

3.3.2 Comment et pourquoi faire varier intentionnellement la vitesse d'exécution des *threads*

Programme Ruby 3.2 Petit programme Ruby avec deux *thread*s simples, mais avec des temps d'exécution variables.

```
def jiggle
  sleep rand / 10.0
end

PRuby.pcall(
  -> { puts "Thr1: ."
       jiggle
       puts "Thr1: .."
       jiggle
       puts "Thr1: ..."
     },
  -> { puts "Thr2: +"
       jiggle
       puts "Thr2: ++"
       jiggle
       puts "Thr2: +++"
     }
  )

puts "Fin du programme"
```

The reason that threading bugs can be sporadic, and hard to repeat, is that only a very few pathways through a vulnerable section can actually fail.

The point is to **jiggle** *the code so that threads run in different orderings at different times.*

«Clean Code», *R.C. Martin*

Pour qu'un programme concurrent soit correct, il doit produire le bon résultat **peu importe la vitesse à laquelle s'exécute les *thread*s**.

```
$ ruby entrelacement.rb
Thr1: .
Thr2: +
Thr1: ..
Thr1: ...
Thr2: ++
Thr2: +++
Fin du programme

$ ruby entrelacement.rb
Thr1: .
Thr2: +
Thr2: ++
Thr2: +++
Thr1: ..
Thr1: ...
Fin du programme

$ ruby entrelacement.rb
Thr1: .
Thr2: +
Thr2: ++
Thr1: ..
Thr1: ...
Thr2: +++
Fin du programme

$ ruby entrelacement.rb
Thr1: .
Thr2: +
Thr2: ++
Thr2: +++Thr1: ..

Thr1: ...
Fin du programme

...
```

Exemple d'exécution 3.1: Divers exemples d'exécution

3.3.3 Situation de compétition

> **Situation de compétition** : défaut dans un système informatique, caractérisé par **un résultat différent selon l'ordre dans lequel sont effectuées certaines opérations**.

Source : `http://fr.wikipedia.org/wiki/Situation_de_comp%C3%A9tition`

Programme Ruby 3.3 Programme avec deux *thread*s qui modifient une variable partagée avec l'opération «+=».

```
x = 0

PRuby.pcall(
  -> { x += 1 },
  -> { x += 2 }
  )

puts "x = #{x}"
```

Question : Quel sera le résultat imprimé par ce bout de code?

Programme Ruby 3.4 Programme avec deux *thread*s qui modifient une variable partagée avec l'opération «+=», mais avec des délais introduits entre les opérations *non-atomiques*.

```
x = 0

PRuby.pcall(
  -> { tmp = x; tmp = tmp + 1; x = tmp }, # tmp est local!
  -> { tmp = x; tmp = tmp + 2; x = tmp } # tmp est local!
  )

puts "x = #{x}"
```

```
x = 0

PRuby.pcall(
  -> { jiggle; tmp = x; jiggle; tmp = tmp + 1; x = tmp },
  -> { jiggle; tmp = x; jiggle; tmp = tmp + 2; x = tmp }
  )

puts "x = #{x}"
```

```
$ ruby competition2.rb
x = 2

$ ruby competition2.rb
x = 1

$ ruby competition2.rb
x = 3

$ ruby competition2.rb
x = 3

$ ruby competition2.rb
x = 2
```

Exemple d'exécution 3.2: Exemples d'exécution du programme Ruby 3.4.

Indépendance entre *threads* et absence de situation de compétition

Une situtation de compétition peut survenir uniquement si ces *threads* **partagent** une ressource

Il peut y avoir situation de compétition uniquement si **un des *threads* modifie** la ressource partagée.

Pour plus de détails, voir plus loin.

Soit l'algorithme suivant, où l'on suppose que la ligne retournée est `EOF` lorsque la fin de fichier est atteinte :

```
PROCEDURE trouver_motif( fich, motif )
DEBUT
  ouvrir le fichier fich
  ligne ← lire une ligne du fichier fich
  TANTQUE ligne ≠ EOF FAIRE
    ecrire ligne SI motif est present dans ligne
    ligne ← lire une ligne du fichier fich
  FIN
FIN
```

Est-il possible de paralléliser la procédure `trouver_motif`, c'est-à-dire, de faire en sorte que certaines tâches à l'intérieur de la procédure s'exécutent de façon concurrente?
On désire évidemment que les lignes du fichiers soient lues dans le bon ordre, et émises dans le bon ordre si elles satisfont le motif.
Indice : Introduire une variable auxiliaire...

Exercice 3.3: Parallélisation de la recherche de motifs dans un fichier.

Situation de compétition et résultat non-déterministe

Situation de compétition et erreur de programmation

Un programme contient une erreur associée à une *situation de compétition* si **une des exécutions possibles** conduit à un résultat erroné ☹

Donc, morale pour la correction : Si je réussis, en ajoutant des `jiggle` dans votre programme, à faire produire un résultat qui n'est pas bon, **alors c'est que votre programme n'est pas correct!**

3.3.4 Atomicité et exclusion mutuelle

Deux principales façons de modifier un programme pour éviter les situations de compétition :

- **Exclusion mutuelle**
- **Synchronisation conditionnelle**

Programme Ruby 3.5 Programme avec deux *thread*s qui modifient une variable partagée avec l'opération «+=», et ce à l'intérieur d'une section critique correctement protégée par un verrou.

```
mutex = Mutex.new

x = 0

PRuby.pcall(
  -> { mutex.synchronize { x += 1 } },
  -> { mutex.synchronize { x += 2 } }
  )

puts "x = #{x}"
```

Remarque : Les deux segments de code suivant sont équivalents :

```
mutex.synchronize { x+= 1 }
```

vs.

```
mutex.lock
x += 1
mutex.unlock
```

```
-> { x += 2 }
```

Soit le segment de code Ruby suivant qui trouve l'élément maximum parmi un tableau de nombres entiers positifs :

```
a = [10, 62, 173, 823, 32, 99, 9292, 0, 1]

m = 0
PRuby.pcall( 0...a.size,
  ->( i ) { m = a[i] if a[i] > m }
)

puts "maximum = #{m}"
```

---

1. Est-ce que ce programme est correct? Justifiez votre réponse.
2. S'il n'est pas correct, comment peut-on le rendre correct?

Exercice 3.4: Recherche parallèle de l'élément maximum d'un tableau

3.3.5 Situation d'interblocage

Programme Ruby 3.6 Petit programme Ruby illustrant une situation *potentielle* d'interblocage.

```
mut1 = Mutex.new
mut2 = Mutex.new

PRuby.pcall(
  -> { mut1.synchronize {
         mut2.synchronize {
           puts "Dans thr1"
         }
       }
     },
  -> { mut2.synchronize {
         mut1.synchronize {
           puts "Dans thr2"
         }
       }
     }
  )

puts "Fin du programme"
```

Question : Dans cet exemple, peut-on éviter l'interblocage?

3.4 Algorithmes parallèles et graphes de dépendances des tâches

3.4.1 La notion de tâche vs. les notions d'unité d'exécution et de *thread*

Pour un algorithme parallèle, la première étape consiste à identifier **toutes les *tâches possibles***, pour identifier ce qui pourrait être fait en parallèle.

Une tâche est un ensemble «d'instructions».

3.4.2 Tâches, dépendances entre tâches et graphes de dépendances

Pour paralléliser un algorithme, il faut identifier toutes les tâches possibles, mais aussi identifier **leurs dépendances**, pour déterminer **ce qui peut, ou non, se faire en parallèle.**

Pour aider à comprendre les dépendances, on peut produire un **graphe de dépendances des tâches**.

3.4.3 Un exemple de graphes de dépendances des tâches : le calcul des racines d'un polynome de 2^e degré

Un polynome $p(x)$ de 2^e degré est défini par :

$$p(x) = ax^2 + bx + c$$

Une racine de $p(x)$ est une valeur telle que $p(r) = 0$.
Pour un polynome de 2^e degré, on a deux racines:

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Code pour calculer `r1` :

```
t0 = -1 * b
t1 = b * b
t2 = 4 * a
t3 = t2 * c
t4 = t1 - t3
t5 = sqrt t4
t6 = t0 + t5
t7 = 2 * a
r1 = t6 / t7
```

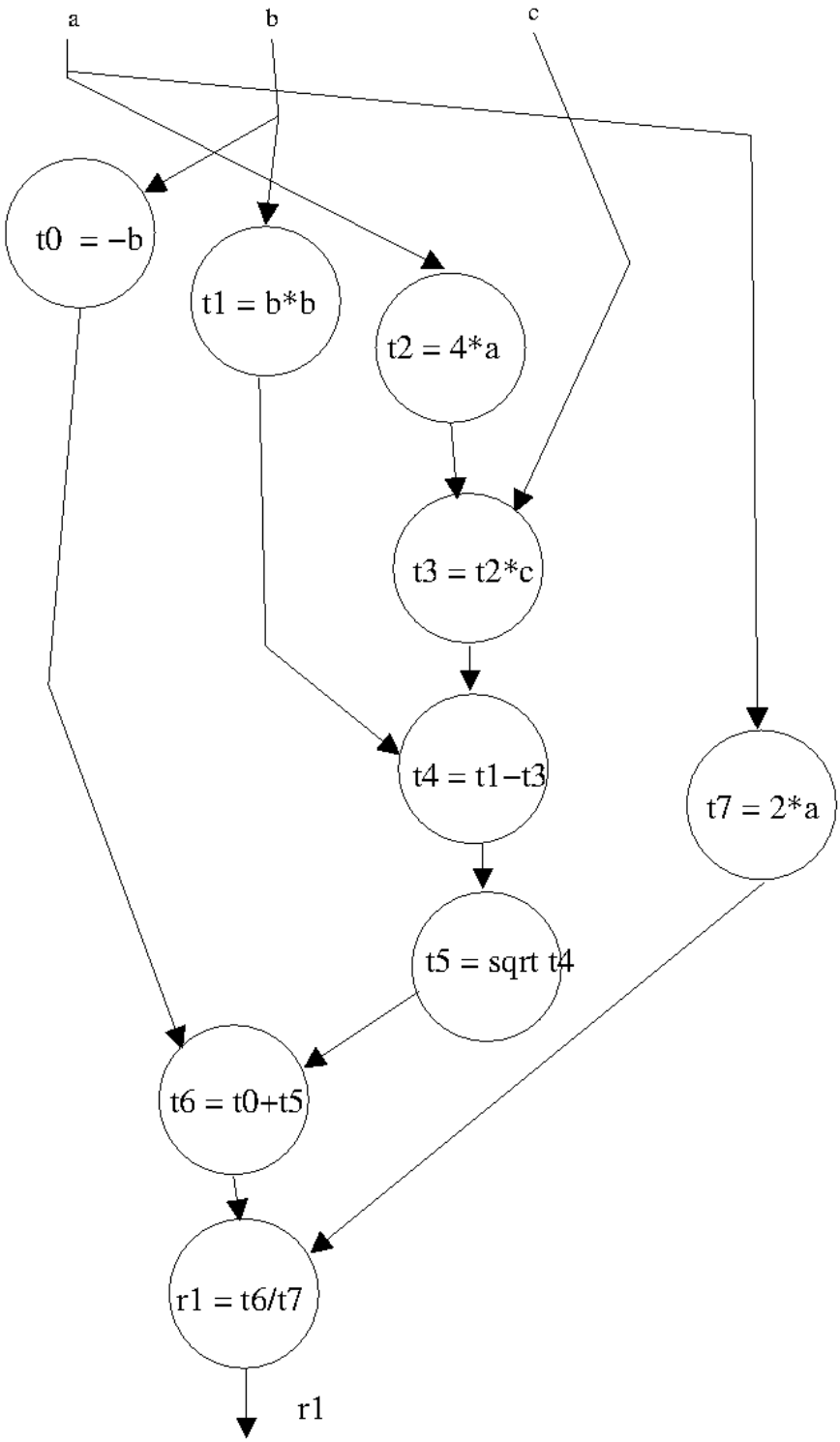


Figure 3.7: Graphe de dépendances des tâches pour le calcul de la racine r_1 d'un polynome de 2^e degré.

3.4.4 Degré de parallélisme

Le **degré de parallélisme** d'un algorithme est le nombre de tâches qui peuvent être exécutées en même temps à chaque instant.

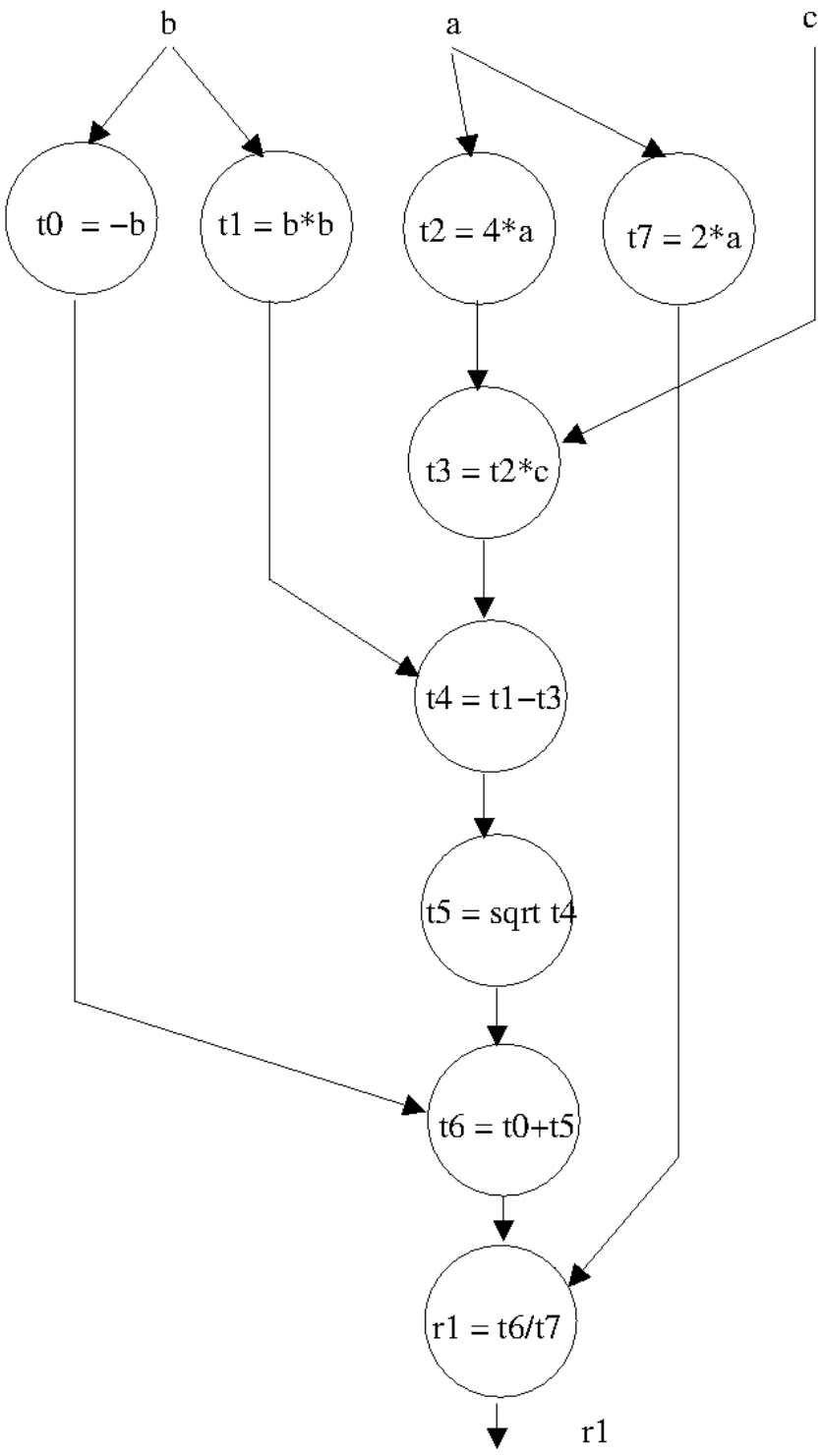


Figure 3.8: Version révisée du graphe de dépendances pour le calcul de la racine r_1 d'un polynome de 2^e degré.

Temps	Degré de parallélisme
0	4
1	1
2	1
3	1
4	1
5	1

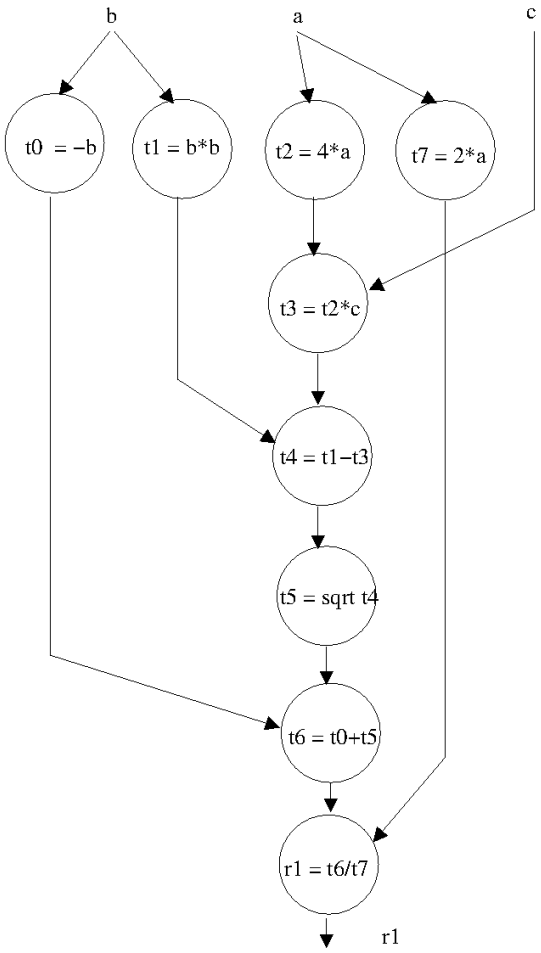
3.4.5 Longueur du chemin critique et temps d'exécution parallèle idéal

Un **chemin** entre S_1 et S_2 est une série de sommets et d'arcs qui permettent d'aller de S_1 à S_2.

La **longueur** d'un chemin est le nombre d'arcs traversés par ce chemin.

Dans un graphe de tâches, le plus long chemin de la tâche initiale à la tâche finale est appelé **le chemin critique**.

C'est cette longueur qui détermine le **meilleur temps d'exécution possible**.



Nb. d'UE	Temps min.
1	9
2	6
3	6
4	6
...	...

Temps d'exécution parallèle idéal

Le *temps d'exécution parallèle minimum* — ou **idéal** — d'un algorithme parallèle est obtenu en utilisant *autant d'unités d'exécution* que nécessaire — donc avec une machine sans limite sur le nombre d'UE.

Ce temps minimum est égal à 1+*la longueur du chemin critique* du graphe de dépendance.

Choix de la cédule d'exécution

3.4.6 Degré de parallélisme et dimensionnement (*scalability*)

Dans plusieurs problèmes, l'ajout d'UE additionnelles n'a aucun d'effet ☹

- Un *algorithme parallèle* est **dimensionnable** lorsque son degré de parallélisme augmente avec la taille du problème.

- Une **architecture** est dimensionnable si la machine continue à fournir les mêmes performances par processeur lorsque l'on accroît le nombre de processeurs.

Avoir un algorithme dimensionnable permet de résoudre des problèmes de plus grande taille sans augmenter le temps d'exécution

Question : Comment?

3.4.7 Granularité des tâches

In parallel computing, granularity means the amount of computation in relation to communication, i.e., the ratio of computation to the amount of communication.

If the granularity is too fine, the performance can suffer from the increased communication overhead. On the other side, if the granularity is too coarse, the performance can suffer from load imbalance.

Source : `http://en.wikipedia.org/wiki/Granularity`

3.4.8 Autres exemples de graphes de dépendances des tâches

Graphe de flux de données pour le calcul de la racine d'un polynome de 2^e degré

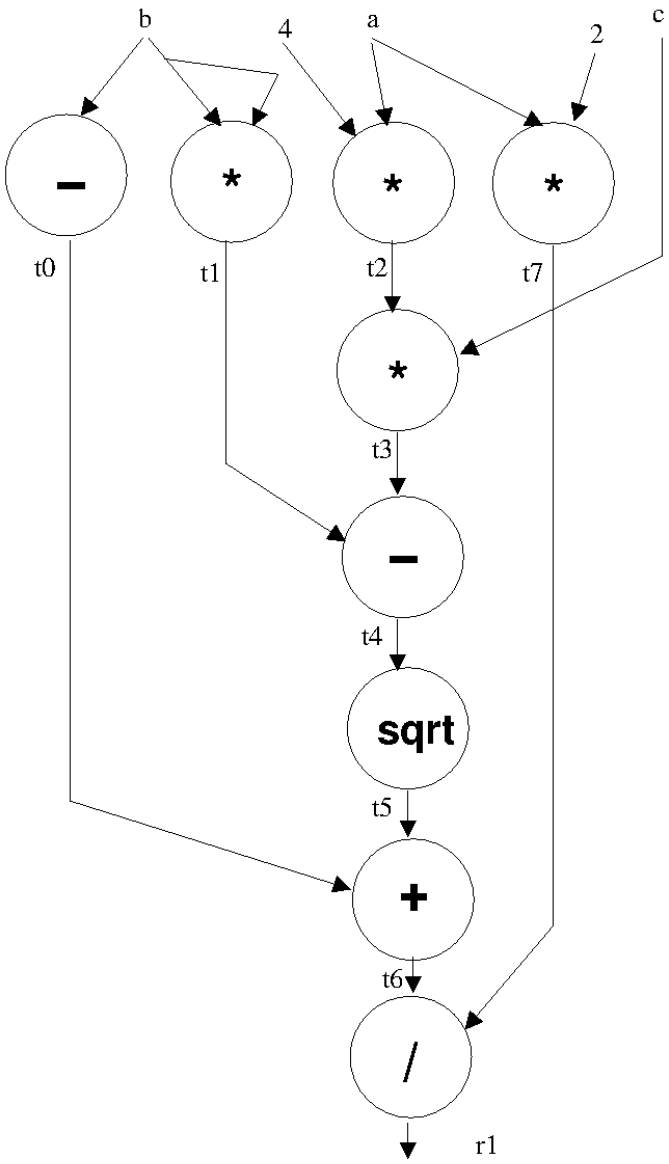


Figure 3.9: Graphe de flux de données pour le calcul de la racine r_1 d'un polynome de 2^e degré.

Somme de deux tableaux

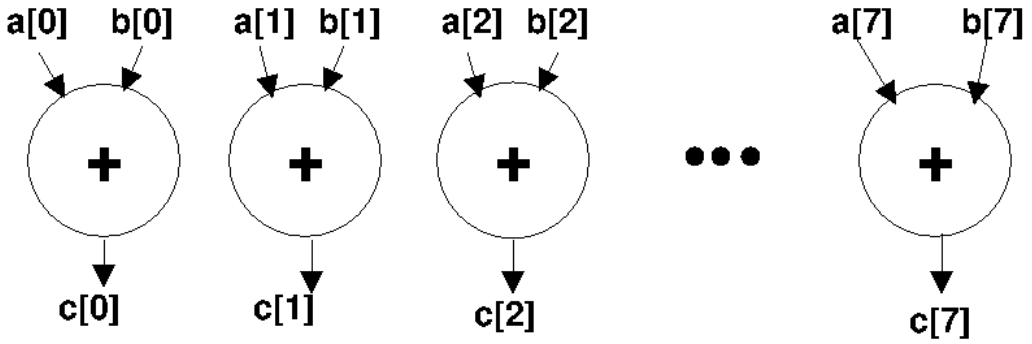


Figure 3.10: Graphe de flux de données pour un algorithme parallèle calculant la somme de deux tableaux de 8 éléments.

```
PRuby.pcall( 0..7,
   -> { |k|  c[k] = a[k] + b[k] }
)
```

Il n'y a *aucune* dépendances entre les tâches $\Rightarrow$ embarrassingly parallel.

Sommation des éléments d'un tableau

On a un tableau de 8 éléments et on veut faire la somme de ces éléments :

```
t0    = a[0] + a[1]
t1    =   t0 + a[2]
t2    =   t1 + a[3]
t3    =   t2 + a[4]
t4    =   t3 + a[5]
t5    =   t4 + a[6]
somme =   t5 + a[7]
```

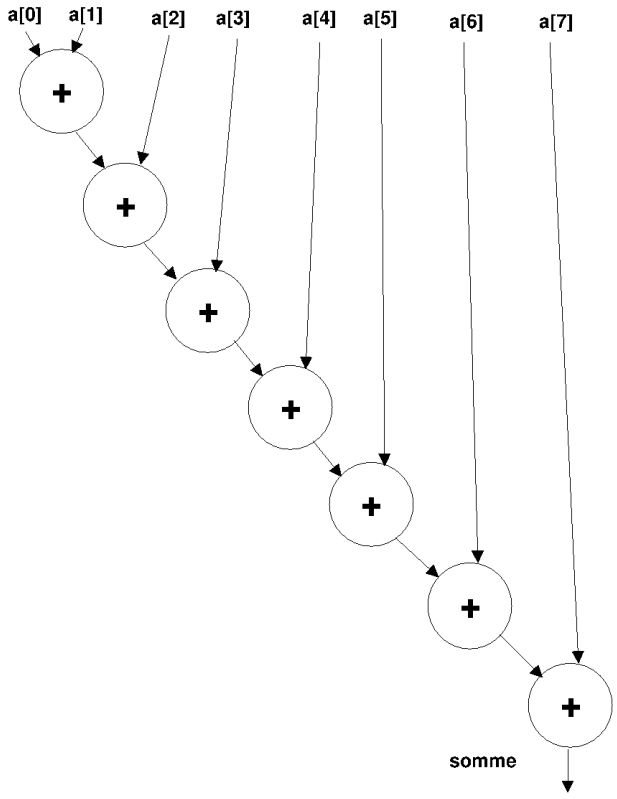


Figure 3.11: Graphe de flux de données pour un algorithme effectuant la sommation des éléments d'un tableau de 8 éléments.

```
t0    = a[0] + a[1]
t1    = a[2] + a[3]
t2    = a[4] + a[5]
t3    = a[6] + a[7]
t4    = t0   + t1
t5    = t2   + t3
somme = t4   + t5
```

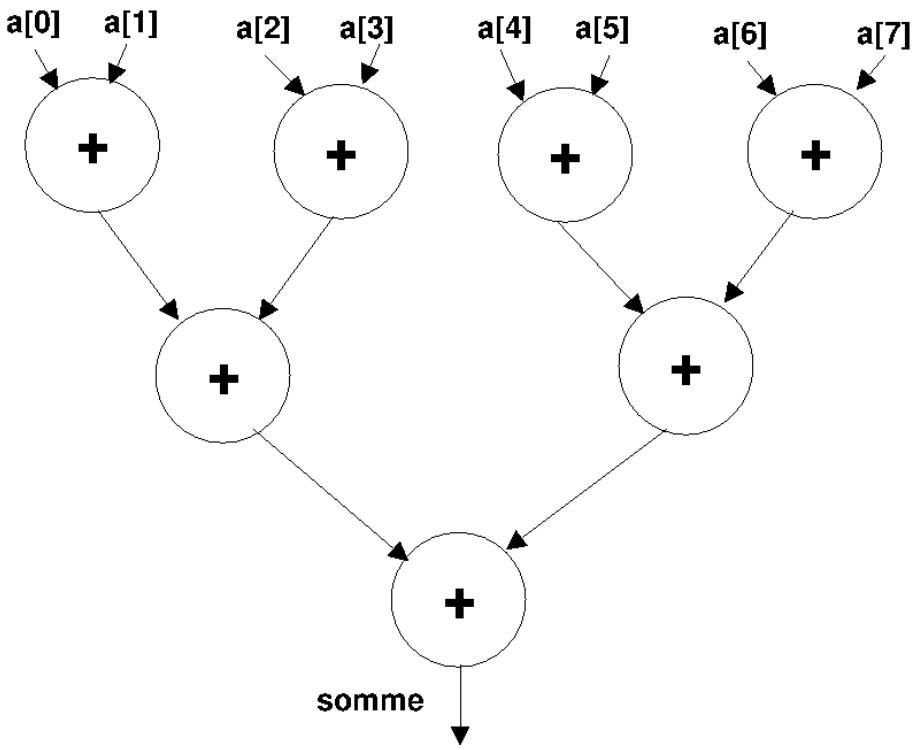


Figure 3.12: Deuxième version — parallélisable! — d'un graphe de flux de données pour un algorithme effectuant la sommation des éléments d'un tableau de 8 éléments.

Sommation récursive des éléments d'un tableau

Programme Ruby 3.7 Algorithme récursif pour effectuer la sommation des éléments d'un tableau.

```
def somme( a, i, j )
  if i == j
    a[i]
  else
    mid = (i + j) / 2

    somme(a, i, mid) + somme(a, mid+1, j)
  end
end
```

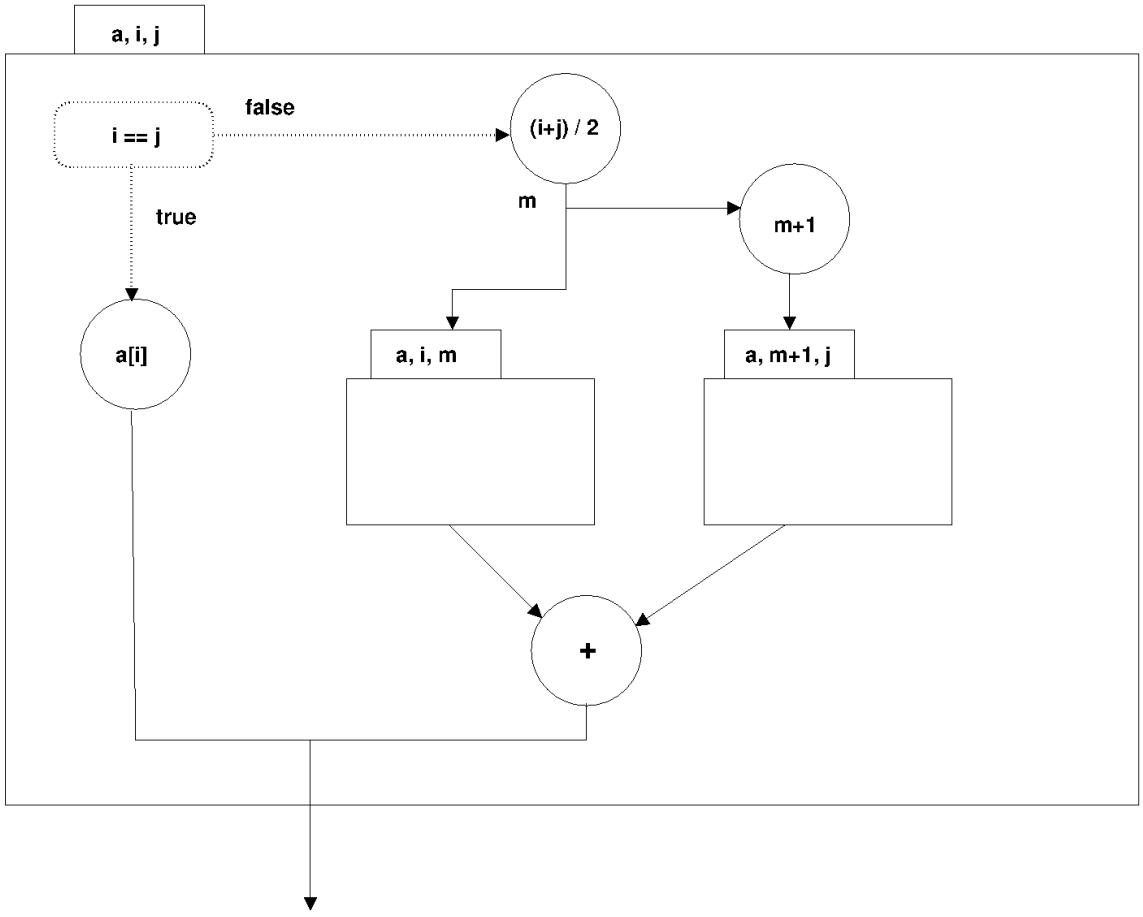


Figure 3.13: Graphe de dépendances de tâches pour le calcul récursif de la sommation des éléments d'un tableau.

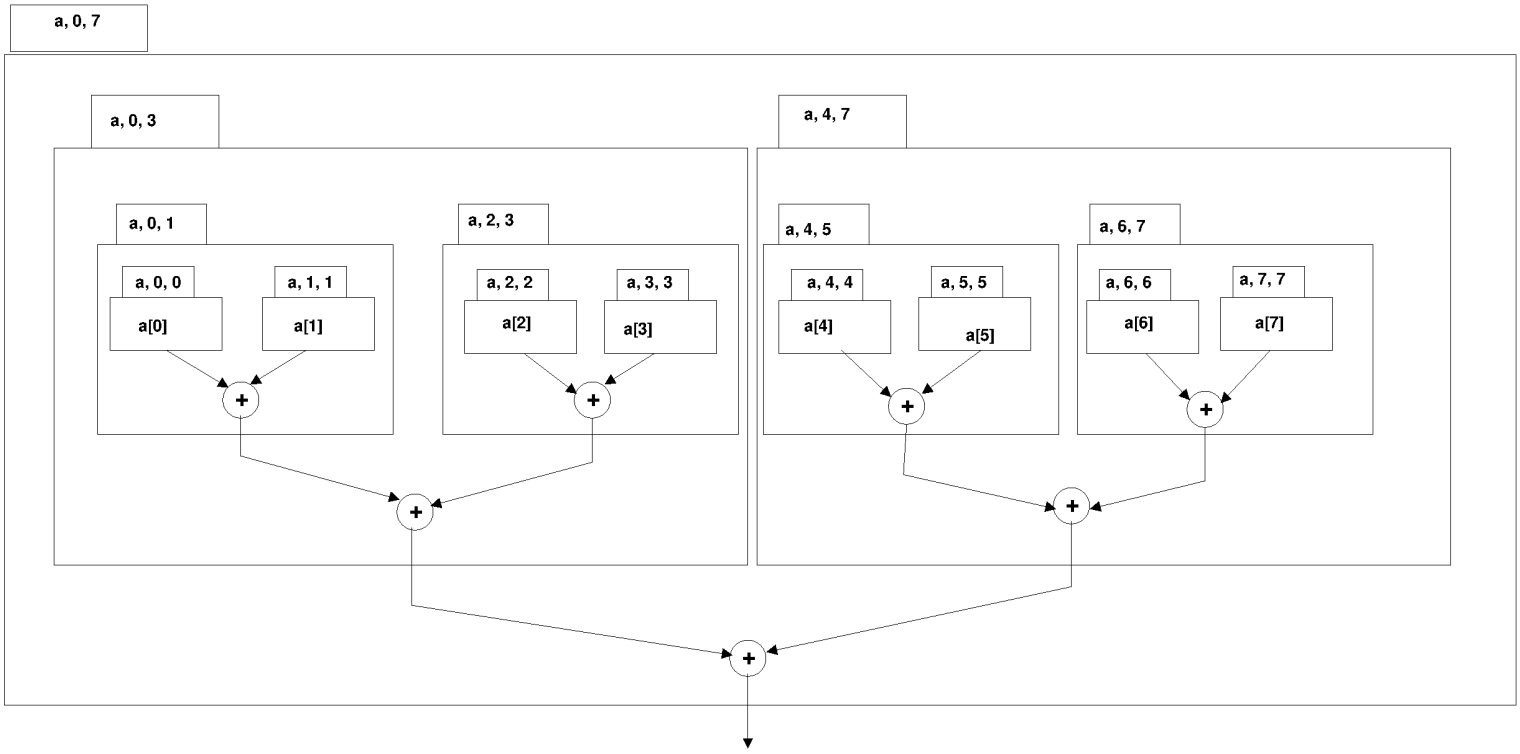


Figure 3.14: Graphe de dépendances de tâches pour le calcul récursif de la sommation des éléments d'un tableau de 8 éléments.

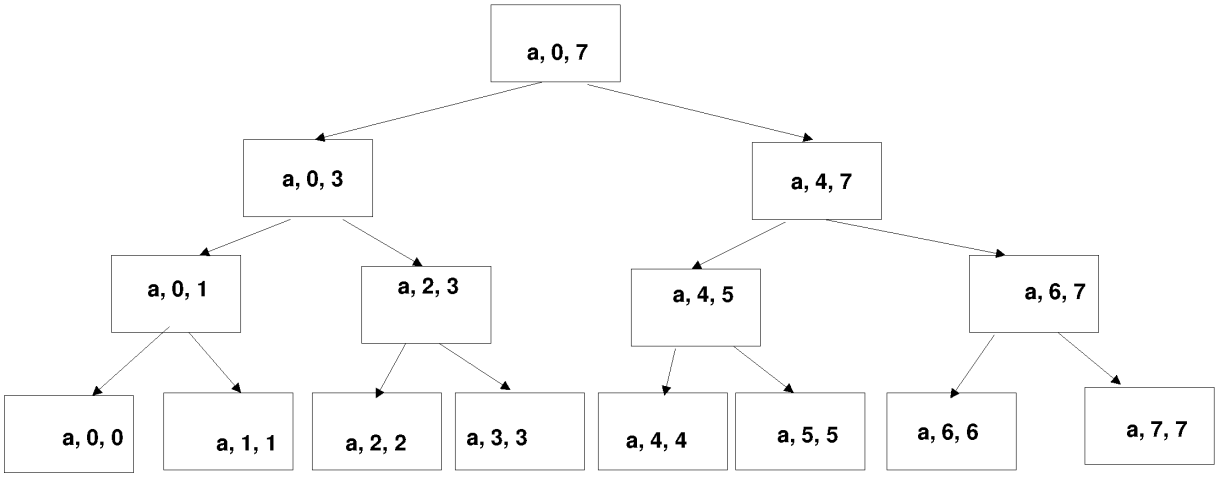


Figure 3.15: Arbre d'activations des instances de fonction pour le calcul récursif de la sommation des éléments d'un tableau de 8 éléments.

3.5 Indépendance entre *threads*

- L'**ensemble de lecture** d'une partie de programme est l'ensemble des variables **lues**, **mais non modifiées**, par cette partie de programme.

- L'**ensemble d'écriture** d'une partie de programme est l'ensemble des variables **modifiées** par cette partie de programme.

- Deux parties de programme sont **indépendantes** si l'ensemble d'écriture de chaque partie est indépendante (intersection **vide**) de l'ensemble de lecture et de celui d'écriture de l'autre partie.

Plus formellement :

- Soit $L(P)$ l'ensemble de lecture de P :

  $L(P) = \{x \mid x \text{ est lue mais non modifiée par } P\}$

- Soit $E(P)$ l'ensemble d'écriture de P :

  $$E(P) = \{x \mid x \text{ est modifiée par } P\}$$

- Soit P_1 et P_2 deux parties de programme. Ces deux parties sont indépendantes ssi :

  $$L(P_1) \cap E(P_2) = \{\}$$
  $$L(P_2) \cap E(P_1) = \{\}$$
  $$E(P_1) \cap E(P_2) = \{\}$$

3.6 Concurrence vs. parallélisme (bis) : Exemples Ruby/MRI vs. JRuby

Un problème avec parallélisme

Un problème avec concurrence

Programme Ruby 3.8 Calcul de la «somme» des éléments d'un tableau.

```ruby
def somme_seq( a )
  (0...a.size).reduce( 0 ) { |somme, k| somme + a[k]**2.0 }
end

def somme_threads( a )
  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

  def sommation_seq( a, bornes )
    bornes.reduce(0) { |somme, k| somme + a[k]**2.0 }
  end

  nb_threads = 10
  threads = (0...nb_threads).collect do |k|
    Thread.new { sommation_seq( a, bornes_tranche(k, a.size
  end

  threads.map(&:value).reduce(&:+)
end

a = Array.new( 1_000_000 ) { rand }

Benchmark.bmbm do |bm|
  bm.report( 'sequentiel' ) do
    somme_seq( a )
  end

  bm.report( 'avec threads' ) do
    somme_threads( a )
  end
end
```

Programme Ruby 3.9 Lecture et analyse d'une liste d'URIs.

```ruby
COURS = ['INF3135', 'INF3140', 'INF4110', 'INF4170', 'INF517
         'INF600A', 'INF7440', 'INF8541', 'MGL7460', 'MGL716

def titre_du_cours( cours )
  uri = "http://www.labunix.uqam.ca/~tremblay/#{cours}/index
  titre = open( uri ) do |page|
    page.detect { |ligne| /TITLE/ =~ ligne }
  end
  if titre && m = /<TITLE>(.*)<\/TITLE>/.match( titre )
    m[1]
  else
    'INCONNU'
  end
end

def titre_du_cours_seq
  COURS.collect { |cours| titre_du_cours(cours) }
end

def titre_du_cours_threads
  threads = COURS.collect do |cours|
    Thread.new { titre_du_cours(cours) }
  end
  threads.map(&:value)
end

Benchmark.bmbm do |bm|
  bm.report( 'sequentiel' ) do
    titre_du_cours_seq
  end

  bm.report( 'avec threads' ) do
    titre_du_cours_threads
  end
end
```

```
RESULTATS AVEC MRI                                          Accélération
-----------------------------------------------------------
                    user     system      total        real
sequentiel      0.250000   0.000000   0.250000 (  0.252342)
avec threads    0.240000   0.000000   0.240000 (  0.242644)  1.04

                    user     system      total        real
sequentiel      0.270000   0.000000   0.270000 (  0.264436)
avec threads    0.240000   0.000000   0.240000 (  0.241311)  1.10

                    user     system      total        real
sequentiel      0.260000   0.000000   0.260000 (  0.258922)
avec threads    0.240000   0.000000   0.240000 (  0.246041)  1.05

                    user     system      total        real
sequentiel      0.260000   0.000000   0.260000 (  0.256229)
avec threads    0.240000   0.000000   0.240000 (  0.241752)  1.06


RESULTATS AVEC JRUBY                                        Accélération
-----------------------------------------------------------
                    user     system      total        real
sequentiel      3.390000   0.260000   3.650000 (  0.322000)
avec threads    0.760000   0.020000   0.780000 (  0.103000)  3.13

                    user     system      total        real
sequentiel      3.270000   0.220000   3.490000 (  0.306000)
avec threads    1.390000   0.030000   1.420000 (  0.194000)  1.58

                    user     system      total        real
sequentiel      3.330000   0.260000   3.590000 (  0.334000)
avec threads    1.300000   0.020000   1.320000 (  0.126000)  2.65

                    user     system      total        real
sequentiel      3.190000   0.320000   3.510000 (  0.315000)
avec threads    1.470000   0.010000   1.480000 (  0.167000)  1.89
```

Figure 3.16: Temps pour diverses exécutions pour la

RESULTATS AVEC MRI — **Accélération**

	user	system	total	real	
sequentiel	0.010000	0.000000	0.010000	(0.094572)	
avec threads	0.020000	0.010000	0.030000	(0.017683)	**5.35**
	user	system	total	real	
sequentiel	0.020000	0.000000	0.020000	(0.104718)	
avec threads	0.010000	0.010000	0.020000	(0.016807)	**6.23**
	user	system	total	real	
sequentiel	0.010000	0.010000	0.020000	(0.104398)	
avec threads	0.020000	0.000000	0.020000	(0.016625)	**6.28**
	user	system	total	real	
sequentiel	0.000000	0.010000	0.010000	(0.094736)	
avec threads	0.010000	0.010000	0.020000	(0.020132)	**4.71**

RESULTATS AVEC JRUBY — **Accélération**

	user	system	total	real	
sequentiel	0.520000	0.030000	0.550000	(0.124000)	
avec threads	0.180000	0.010000	0.190000	(0.021000)	**5.90**
	user	system	total	real	
sequentiel	0.520000	0.040000	0.560000	(0.136000)	
avec threads	0.190000	0.010000	0.200000	(0.020000)	**6.80**
	user	system	total	real	
sequentiel	0.580000	0.060000	0.640000	(0.148000)	
avec threads	0.210000	0.020000	0.230000	(0.026000)	**5.69**
	user	system	total	real	
sequentiel	0.510000	0.030000	0.540000	(0.114000)	
avec threads	0.200000	0.020000	0.220000	(0.019000)	**6.00**

Figure 3.17: Temps pour diverses exécutions pour la

3.7 Exercice : Interactions entre *threads* et tableaux dynamiques

```ruby
#!/usr/bin/env ruby
#
#  Petit programme illustrant certaines
#  interactions entre threads et reallocation
#  dynamique de la taille d'un tableau.
#

NB = 20

loop do
  a = Array.new

  futures = []
  (0...NB).each do |i|
    futures << PRuby.future { a[i] = 0 }
  end

  futures.map(&:value)

  puts a.reduce(&:+)
end
```

Exercice 3.5: Qu'est-ce qui sera imprimé par ce programme?