Table des matières

Chapitre 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**. Toutefois, on établit souvent les distinctions suivantes entre processus et *thread* — voir aussi Figure 3.1 :

- **Processus** :
  - Un *processus* représente un **programme** en cours d'exécution.
  - Un processus possède un espace mémoire **privé**, indépendant de celui des autres processus.
  - Parce qu'ils ne partagent aucun espace mémoire commun, deux processus qui veulent collaborer doivent le faire en utilisant un mécanisme **d'échange de *messages*** — par exemple, pour des processus Unix, en utilisant des *pipes*.
  - Le contexte d'un processus est *lourd* : code, registres (dont SP, FP et IP[1]), pile, tas[2], fichiers, vecteur d'interruptions, etc. Créer et activer un processus — ainsi qu'effectuer un changement de contexte — est donc (très!) coûteux!

- ***Thread*** :
  - Un *thread* représente (généralement) une **fonction** (une méthode) en cours d'exécution.

[1] SP=*Stack Pointer*, FP=*Frame Pointer*, IP=*Instruction Pointer*.
[2] Tas = *heap*.

– Un processus peut contenir un ou plusieurs *threads*. Ces *threads* partagent alors la mémoire (e.g., code, tas) ainsi que diverses autres ressources.
– Parce qu'ils partagent un même espace mémoire, deux *threads* qui veulent collaborer peuvent le faire par l'intermédiaire de **variables partagées**.
– Le contexte d'un *thread* est *léger* : registres (dont IP), pile. Créer et activer un *thread* — et effectuer un changement de contexte — est donc moins coûteux que dans le cas d'un processus!

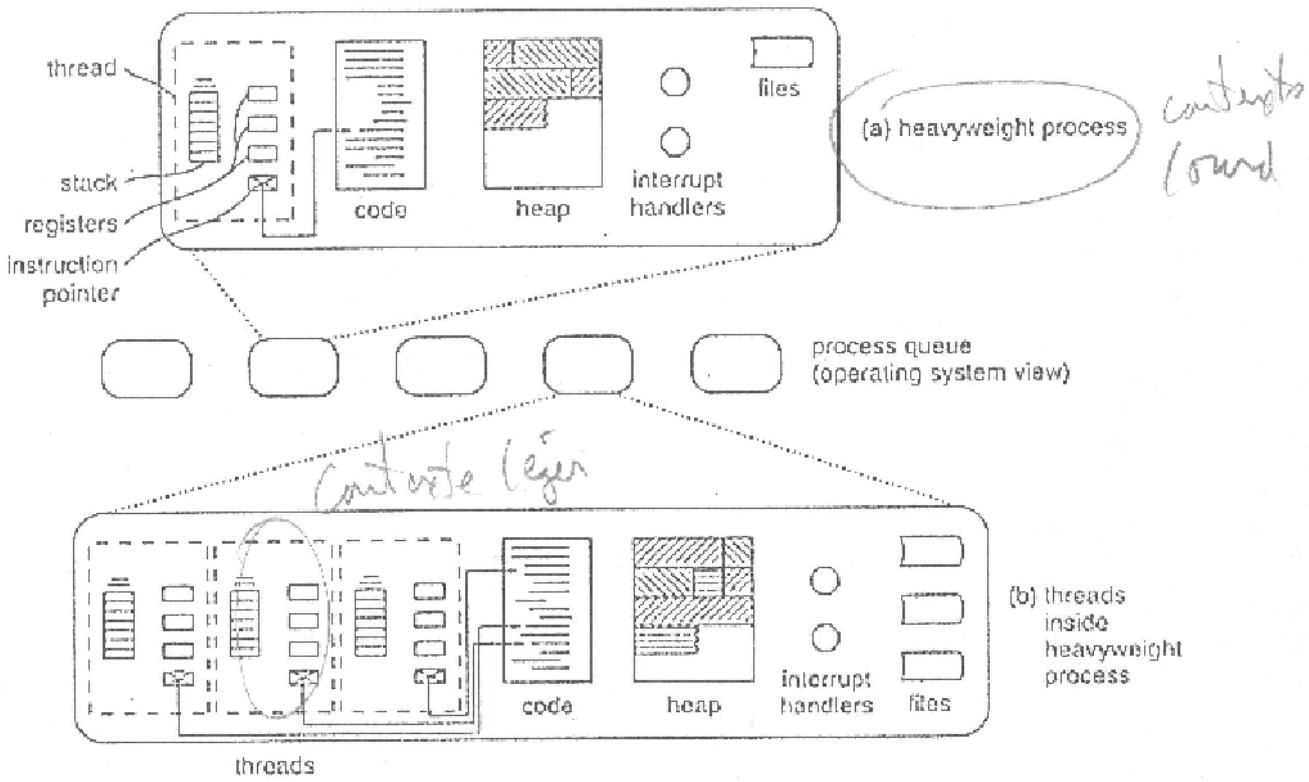

Figure 4.1
Process Organization

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

Remarques :

- Il faut bien distinguer entre le programme en tant que «code» — code source ou code compilé, une notion *statique* — et le programme en tant que «code *en cours d'exécution*» — une notion *dynamique*.

  Par exemple, un même programme Java — un même code (compilé) — peut avoir plusieurs instances différentes en cours d'exécution (exemple Unix) :

  ```
  $ javac Foo.java
  $ java Foo & java Foo & java Foo &
  ```

  La même remarque s'applique pour des *thread*s multiples lancés à partir de l'exécution d'une même fonction ou méthode.

- En fait, les processus de ces langages sont souvent même *plus légers que des* threads. Ces processus étant légers, la création et l'activation d'un tel processus — et un changement de contexte — est donc peu coûteux, souvent moins que pour des *threads*.

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 ⇒ Communication, échange d'information

Deux principales façons de communiquer :

- Par l'intermédiaire de variables *partagées* — *principalement dans le cas de* threads
- Par l'échange de messages et de signaux — par ex., via des *pipes* dans le cas de processus Unix

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 — voir Figure 3.2 :**

- Application **multi-contextes** (*multi-threaded*) = contient deux ou plusieurs *thread*s, qui peuvent ou non s'exécuter en même temps, et qui sont utilisés pour mieux organiser et structurer l'application (meilleure modularité)

  Exemples : Système d'exploitation multi-tâches, fureteurs multi-tâches, interface personne-machine vs. traitement de la logique d'affaire, serveur Web

- Application **parallèle** = chaque *thread* s'exécute sur son propre processeur (ou coeur), dans le but de résoudre plus rapidement un problème — ou pour résoudre un problème plus gros

  Exemples : Prévisions météorologiques, modélisation du climat, simulations physiques, bio-informatique, traitement graphique, etc.

- Application **distribuée** = contient deux ou plusieurs processus, qui communiquent par l'intermédiaire d'un réseau ($\Rightarrow$ délais plus longs), et ce pour répartir, géographiquement, des données et des traitements

  Exemples : Serveurs de fichiers, accès à distance à des banques de données

Les figures 3.3 à 3.6 illustrent les différences entre applications multicontextes et applications parallèles.

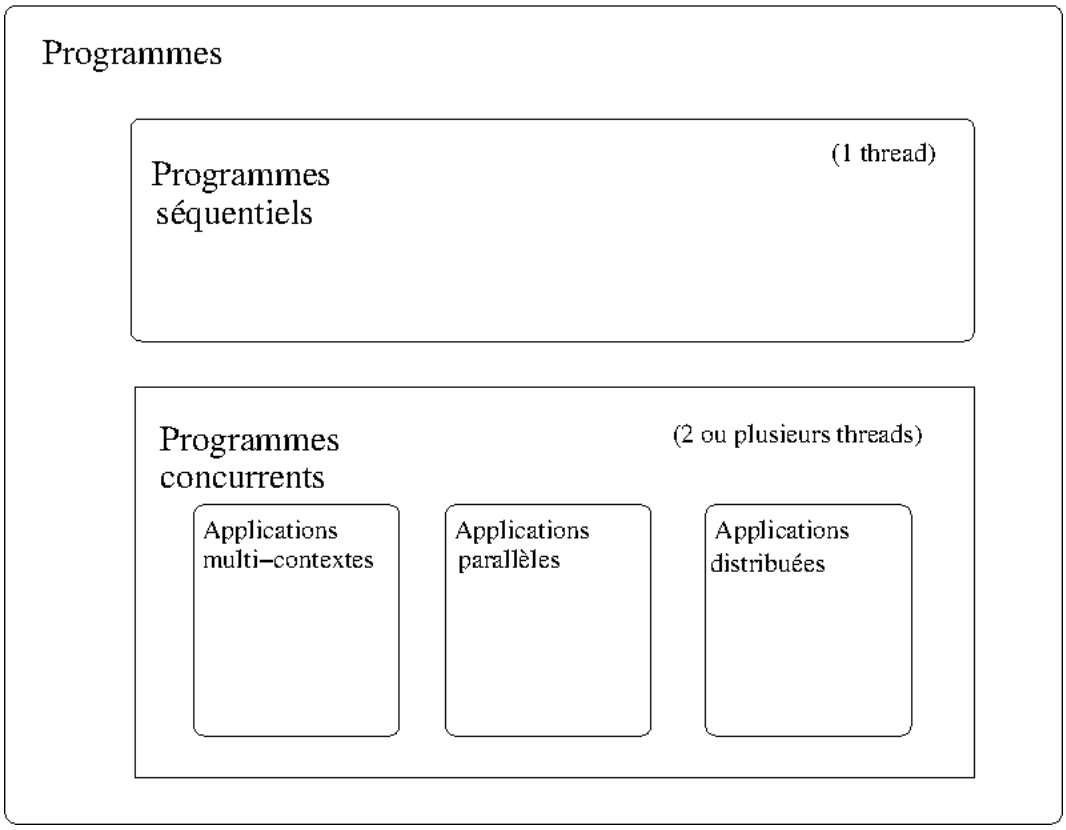

Figure 3.2: Programmes vs. programmes concurrents vs. applications multi-contextes, parallèles ou distribuées.

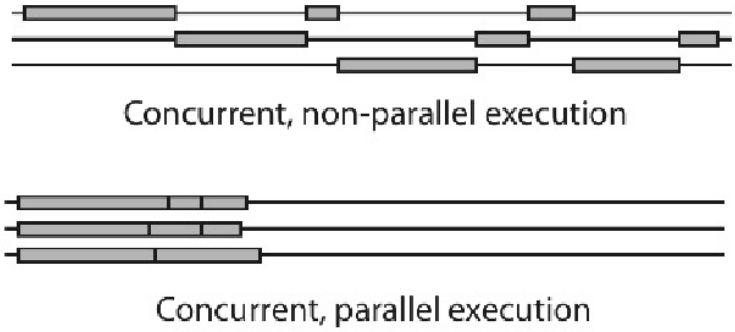

Figure 3.3: Exécution concurrente non parallèle vs. exécution concurrente et parallèle de throis *threads* : tiré de [SMR09].

À première vue, selon le diagramme de la Figure 3.3, il semblerait qu'une exécution concurrente mais non parallèle ne permette pas de réduire le temps d'exécution. Toutefois, ce n'est pas toujours ainsi!

Dans la Figure 3.3, la partie du bas suppose que chacune des tâches indiquée (chacune des lignes horizontales) peut s'exécuter sans interruption — par exemple, il s'agit d'un programme *CPU-bound* : voir plus bas.

Par contre, si l'exécution doit être interrompue pour effectuer des entrées/sorties — qui peuvent être très longues — alors une exécution avec *threads* vs. une exécution sans *thread* pourraient être telles que présentées à la Figure 3.4 : lorsqu'une opération d'E/S doit être effectuée, une exécution concurrente avec *threads* peut effectuer un changement de contexte, pour éviter d'attendre inutilement et permettre à un autre *thread* de progresser pendant que l'E/S se complète.

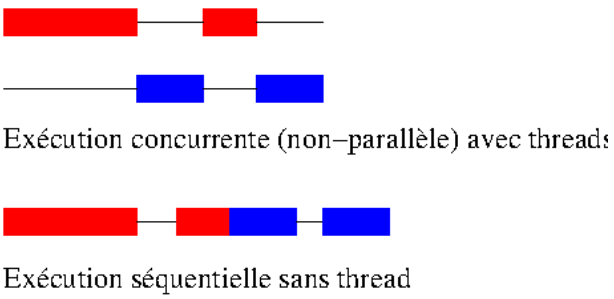

Figure 3.4: Exécution d'un programme effectuant des entrées/sortie de façon concurrente avec *threads* vs. de façon séquentielle sans *thread*.

Pour un exemple plus détaillé, voir l'exemple présenté à la Section 3.6.

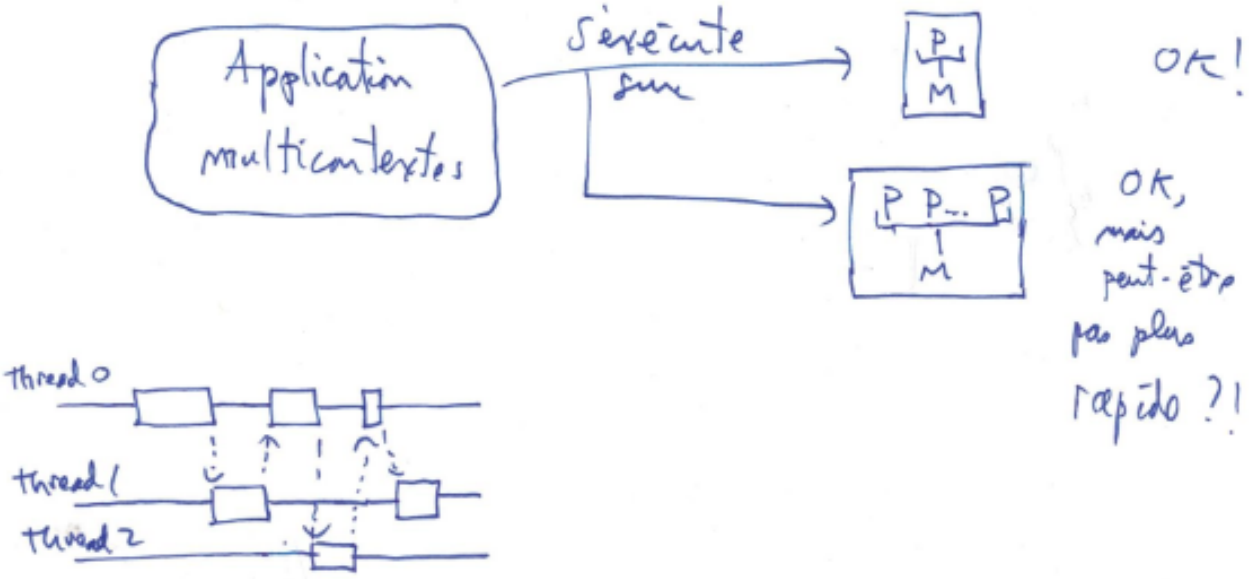

Figure 3.5: Exécution d'une application multicontextes sur une machine séquentielle vs. une machine parallèle.

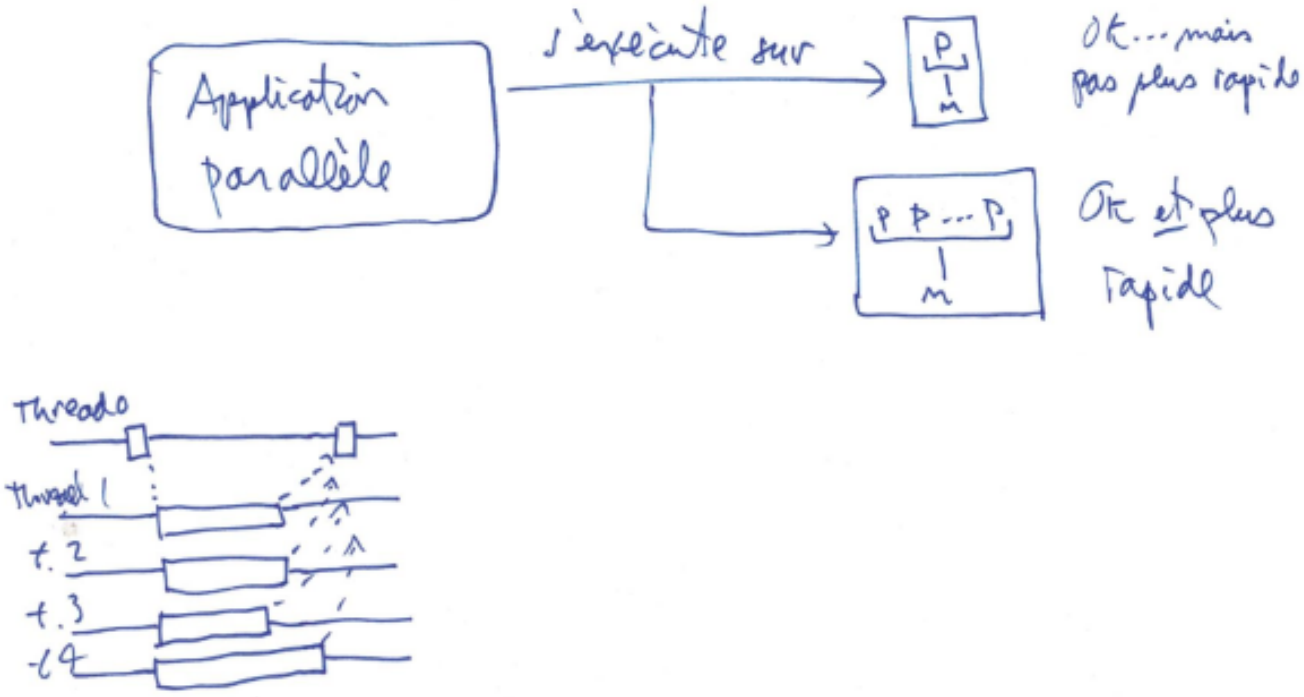

Figure 3.6: Exécution d'une application parallèle sur une machine séquentielle vs. une machine parallèle.

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

Une caractéristique intéressante qui permet de comprendre la différence entre exécution concurrente et exécution parallèle est celle de programme *CPU-bound* vs. *IO-bound* :

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

  On peut donc supposer qu'un programme *CPU-bound* passe la majeure partie de son temps d'exécution à utiliser le CPU — il est *gourmand en CPU* — et donc **il s'exécuterait plus rapidement si le CPU était plus rapide**.

- *IO-bound* : un programme est *IO-bound* si son temps d'exécution est limité (contraint) par la vitesse du système d'entrées/sorties (accès disques, accès réseaux, etc.).

  Un programme *IO-bound* passe donc la majeure partie de son temps d'exécution à utiliser les E/S — il est *gourmand en E/S* — et 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*.

Remarques additionnelles :

- Les catégories qui précèdent *ne sont pas* mutuellement exclusives. Par exemple, de nombreuses machines parallèles modernes, qu'on utilise essentiellement pour développer des *applications parallèles*, sont des *multi-ordinateurs*, donc des machines composées d'un ensemble de processeurs (avec mémoire distribuée) interconnectés *par un réseau*. On programme souvent ces machines avec des langages de programmation où l'on doit tenir compte de la distribution (principalement des données) et où les échanges d'information se font par l'envoi de messages.

- Dans le cadre du cours, nous traiterons principalement **d'applications parallèles**, puis **d'applications concurrentes**, mais pas d'applications distribuées. De plus, nous traiterons principalement de la programmation concurrente et parallèle *avec mémoire partagée*.

- Dans le cours, nous utiliserons les termes suivants :

  - **Programmation parallèle** = Programmation pour développer une application **parallèle** et l'exécuter sur machine **parallèle**.

    Dans cette forme de programmation, l'objectif sera d'obtenir un programme qui, si possible, s'exécute plus rapidement.

  - **Programmation concurrente** = Programmation pour développer une application concurrente, plus généralement une application multi-contextes, et l'exécuter sur machine arbitraire — séquentielle, multi-coeurs, multi-processeurs, etc.

    Dans ce cas, l'objectif sera typiquement de développer un programme comportant plusieurs *threads* qui **partagent des ressources** de façon correcte et efficace.

- Voir aussi l'exemple présenté à la Section 3.6.

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 valeurs des variables du *thread* à un instant donné, y compris les variables implicites, notamment le compteur d'instruction (IP).
- Action atomique = Une action qui examine ou change l'état d'un *thread* **de façon *indivisible***.
- Un *thread* exécute une séquence d'instructions. Chaque instruction peut elle-même être mise en oeuvre par une séquence d'**une ou plusieurs *actions atomiques*** associées à des instructions machines.
- L'effet de l'exécution d'un programme concurrent est obtenu par l'entrelacement des diverses actions **atomiques** effectuées par chacun des *thread*s.

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

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

puts "Fin du programme"
```

Soit le Programme Ruby 3.1. La méthode `pcall` de la bibliothèque `PRuby` — méthode de classe (statique) — reçoit en argument deux ou plusieurs `lambda`s. Ces différents lambdas sont alors évalués de façon concurrente — *en parallèle si les ressources le permettent*. 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**, introduisant une forme de *barrière de synchronisation* entre les *threads* exécutant les lambdas. «**Typiquement**», l'exécution de ce programme produira le résultat suivant :

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

Ici, c'est ce résultat qui est produit *parce que les deux* thread*s sont petits et simples*, et donc le premier *thread* peut s'exécuter rapidement et entièrement parfois même avant que le deuxième *thread* ne s'exécute.

Toutefois, il est aussi possible que, parfois, ce programme produise plutôt un résultat tel que le suivant :

```
$ 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 infrequent, sporadic, and hard to repeat, is that only a very few pathways out of the many thousands of possible 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. The combination of well-written tests and jiggling can dramatically increase the chance of finding errors.*
>
> «Clean Code», *R.C. Martin*

Pour qu'un programme concurrent soit considéré correct, il doit produire le bon résultat **peu importe la vitesse à laquelle s'exécute chacun des *thread*s**. Soit alors le Programme PRuby 3.2, une version modifiée du Programme PRuby 3.1 dans lequel on a introduit un délai arbitraire d'exécution entre chacune des instruc-

tions `puts`. Un tel délai — introduit ici par un appel à la méthode `jiggle`[3] — permet de modéliser explicitement des variations dans la vitesse d'exécution des *thread*s, ce qui est souvent utile pour déboguer un programme concurrent.

L'exemple d'exécution 3.1 (p. 19) illustre divers résultats d'exécution possibles du Programme Ruby 3.2 : les deux premiers éléments de la trace sont toujours les mêmes — à cause de l'ordre de lancement des deux *thread*s et de l'absence de `jiggle` au début du *thread* — mais ensuite l'ordre change d'une exécution à l'autre (ou presque ☺) selon la durée du délai aléatoire produit par `jiggle`. On peut noter aussi la dernière exécution, *où même les deux chaînes émises sur la sortie standard sont entrelacées* — l'une des chaînes s'imprime avant que le saut de ligne de la chaîne précédente n'ait été imprimé.

[3]Traduction de *jiggle* : secouer, se trémousser.

```
$ 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 du Programme Ruby 3.2, version avec des `jiggles` du Programme Ruby 3.1.

3.3.3 Situation de compétition

> **Situation de compétition**, plus couramment nommée *race condition*, est un défaut dans un système [...] informatique, [...] caractérisé par **un résultat différent selon l'ordre dans lequel sont effectuées certaines opérations** du système.
>
> Source : `http://fr.wikipedia.org/wiki/Situation_de_comp%C3%A9tition`

En anglais, on parle de «*race condition*» — littéralement, «condition de course». L'utilisation du terme «course» tient au fait qu'une telle situation de compétition peut survenir parce que les vitesses d'exécution des *thread*s sont imprévisibles et/ou varient (attente pour des ressources, allocation du temps processeur, etc.) et que le résultat produit par l'exécution du programme *dépend de ces vitesses d'exécution.*

La notion d'entrelacement ainsi que celle de situation de compétition dépendent de ce qui est considéré comme une «instruction atomique».

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}"
```

Soit le programme Ruby 3.3. Le résultat produit typiquement par l'exécution de ce programme sera le suivant :

```
$ ruby competition.rb
x = 3
```

Par contre, *de temps en temps*, le résultat imprimé sera `x = 2` ou `x = 1`!

Le résultat le plus typique, et attendu, est produit pour les mêmes raisons que précédemment : le premier *thread* qui est lancé s'exécute de façon simple et rapide. Par contre, ce programme *contient effectivement une situation de compétition*, qu'on peut reproduire plus facilement en le modifiant pour obtenir le programme Ruby 3.4 :

- On décompose les instructions non-atomiques en instructions plus simples. Ainsi, une instruction telle que «`x += 1`» *n'est pas une instruction atomique.*

Elle correspond en fait à l'instruction «`x = x + 1`», laquelle correspond à un séquence suivante d'instructions de bas niveau semblable à la suivante : «`tmp = x; tmp = tmp + 1; x = tmp`».[4]

- On introduit des délais arbitraires avec `jiggle` à divers endroits.

L'exemple d'exécution 3.2 donne alors des résultats possibles d'exécution, montrant clairement la présence d'une situation de compétition.

Programme Ruby 3.4 Programme avec deux *threads* 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}"
```

[4] En fait, dans une machine RISC typique, où les opérations arithmétiques se font uniquement sur les registres, on aurait les (pseudo-)instructions suivantes :

```
LOAD r1, x
r1 = r1 + 1
STORE r1, 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 entre deux *threads* peut survenir uniquement si ces *threads* **partagent** une ressource — par ex., une variable. Lorsqu'il n'y a aucun partage de ressources, il y a alors *indépendance* entre les *threads*, et donc aucun danger de situation de compétition. Dans ce cas, les deux *threads* peuvent alors s'exécuter en parallèle sans problème, sans synchronisation particulière.

Plus précisément, il peut y avoir situation de compétition uniquement si (au moins) **un des *threads* modifie** la ressource partagée. Si les deux *threads* ne font que lire la ressource, alors aucun conflit n'est possible.

Pour plus de détails, voir plus loin (section 3.5).

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

Si deux parties de programme sont *indépendantes*, alors elles peuvent être exécutées en parallèle sans qu'il n'y ait d'interférence, car il n'y a aucun danger de situation de compétition. Par contre, si deux parties de programme ne sont pas *indépendantes*, alors elles peuvent être exécutées en parallèle, soit...

- en ajoutant des opérations de synchronisation pour éviter les interférences indésirables, donc pour supprimer les situations de compétition — voir prochaine section.
- en les exécutant *telles quelles* si un résultat non-déterministe est acceptable.

Situation de compétition et erreur de programmation

On dit d'un programme concurrent qu'il contient une erreur de programmation associée à une *situation de compétition* si **une des exécutions possibles** (n'importe laquelle parmi les divers entrelacements possibles) conduit à un résultat erroné ☹

Une autre façon de présenter les choses est la suivante. Supposons que je suis en train de corriger un programme concurrent que vous avez écrit. Si je réussis, en ajoutant quelques instructions `jiggle` dans votre programme, à faire produire un résultat qui n'est pas le bon, alors c'est que votre programme n'est pas correct!

3.3.4 Atomicité et exclusion mutuelle

Il y a deux principales façons de modifier un programme concurrent pour éviter les situations de compétition, et ce en ajoutant des instructions qui vont exclure certains entrelacements — soit en rendant atomiques des séquences d'instructions, soit en retardant l'exécution d'une séquence d'instructions :

- **Exclusion mutuelle** = Permet de combiner des groupes d'instructions en *sections critiques* de façon à les rendre **indivisibles**, donc atomiques, ce qui fait que les instructions internes ne peuvent plus être entrelacées avec les instructions internes d'autres sections critiques.
- **Synchronisation conditionnelle** = Permet de retarder une action ou une instruction jusqu'à ce que l'état du système satisfasse une certaine condition.

Dans ce qui suit, nous illustrons l'exclusion mutuelle ; la synchronisation conditionnelle sera étudiée dans un chapitre ultérieur.

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}"
```

Le programme Ruby 3.5 est une version révisée du programme Ruby 3.3, mais cette fois sans situation de compétition, donc le programme imprimera toujours «`x = 3`».

Dans les deux *thread*s, la modification de `x` est effectuée à l'intérieur d'une section critique protégée par un verrou d'exclusion mutuelle — objet `mutex` instance de la classe `Mutex`. Lorsqu'on utilise un tel verrou, le bloc de code qui suit `synchronize` est assuré de s'exécuter de façon complètement atomique, sans interférence par d'autres *thread*s exécutant eux aussi des sections critiques protégées par le même verrou!

Quelques remarques additionnelles :

- Un appel tel que «`mutex.synchronize { x+= 1 }`» est en fait équivalent au segment de code suivant :

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

- La forme avec `lock`/`unlock` permet de comprendre plus en détail comment fonctionne un tel `Mutex` — qu'on appelle aussi un **verrou d'exclusion mutuelle**. Un tel verrou peut être dans l'un de deux états possibles suivants :
  - Déverrouillé : Dans cet état, un appel à `lock` met le verrou dans l'état verrouillé, et le *thread* qui appelle `lock` *poursuit immédiatement son exécution, sans bloquer.*
  - Verrouillé : Dans cet état, le *thread* qui appelle `lock` est *suspendu*, donc *reste bloqué*, en attente que le verrou redevienne libre (déverrouillé).

    Toujours dans cet état, lorsqu'un *thread* ayant acquis le verrou appelle `unlock`, alors le verrou redevient libre, i.e., déverrouillé. Si un ou plusieurs *thread*s étaient suspendus en attente du verrou, *alors un d'entre eux — et un seul! — obtiendra le verrou... et donc le verrouillera à nouveau.*

- Pour qu'un accès à une variable soit correctement protégé par une section critique atomique, il faut évidemment que tous les accès à cette variables le soient aussi. Donc, dans le programme précédent, si le deuxième *thread* du `pcall` est défini comme suit, alors le programme contiendra une situation de compétition :

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

- La description qui précède est générale, applicable à la plupart des langages de programmation modernes. Par contre, le comportement exact lors d'un `lock` ou `unlock` pourra dépendre du langage ou de la bibliothèque de *thread*s :
  - Dans les langages de haut niveau, un *thread* qui est mis en en attente d'un verrou sera **suspendu**, au sens où il ne consommera pas de temps CPU pendant son attente. On parle alors d'une **attente passive**. Par contre, des *thread*s *de très bas niveau* pourraient attendre de façon *active*, e.g., en bouclant jusqu'à ce qu'un certain bit devienne à 1.
  - Dans certains langages, une erreur sera signalée si le *thread* qui effectue l'appel à `unlock` n'est pas le même qui celui qui a fait l'appel à `lock`.

– Dans certains langages, un *thread* peut acquérir (avec `lock`) un verrou qu'il a déjà acquis mais qu'il n'a pas encore libéré — on parle alors de *verrou réentrant*. Par contre, dans d'autres langages, une telle situation conduira à un *deadlock* — voir section suivante — parce que le *thread* bloquera à cause de la non-disponibilité du verrou, déjà pris... par le même *thread*!

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

```ruby
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

Une situation d'interblocage — un *deadlock* — survient lorsque tous les *thread*s d'un programme sont bloqués en attente d'un événement ou d'une condition... mais qui ne pourra jamais survenir, par exemple, *parce que tous les* thread*s sont bloqués.*

Un exemple classique est celui d'une dépendance *cyclique* entre *thread*s — on parle alors d'un cas de *deadly embrace*.

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"
```

Le Programme Ruby 3.6 présente un exemple simple. Si on exécute ce programme, un résultat *possible* d'exécution pourrait être le suivant :

```
$ ruby interblocage.rb
Dans thr1
Dans thr2
Fin du programme
```

Par contre, dans certains cas, ... rien n'arrive, rien n'est imprimé! Pour expliquer ce dernier comportement, supposons la séquence suivante d'exécution des instructions de `thr1` et `thr2` — i.e., un entrelacement possible de l'exécution de

leurs instructions — obtenu par exemple en ajoutant «`jiggle`» dans `thr1` après l'acquisition du premier verrou :

- Le *thread* `thr1` acquiert avec succès le verrou `mut1`.
- Le *thread* `thr2` acquiert avec succès le verrou `mut2`.

  Le *thread* `thr2` tente d'acquérir le verrou `mut1`, lequel n'est pas disponible, donc `thr2` est suspendu.

  Le *thread* `thr1` tente d'acquérir le verrou `mut2`, lequel n'est pas disponible, donc `thr1` est suspendu.
- Oops! Situation d'interblocage : `thr1` attend après `thr2` (pour qu'il libère `mut2`) **et** `thr2` attend après `thr1` (pour qu'il libère `mut1`).

Dans cet exemple où deux verrous sont utilisés, la solution pour éviter un interblocage est relativement simple : il faut toujours acquérir les deux verrous *dans le même ordre*, donc interchanger les deux premières instructions de `thr2`.

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*

Lorsqu'on développe un algorithme parallèle, la première étape consiste à identifier **toutes les *tâches possibles***, pour mieux identifier ensuite ce qui pourrait être fait en parallèle.

Lorsqu'on parle de *tâche*, on parle ici d'une notion *logique*, liée au problème et à sa solution exprimée de façon abstraite, donc de niveau *algorithmique*. Une tâche est alors un ensemble d'instructions, d'opérations arithmétiques ou logiques simples, qui «fait du sens» dans le contexte de l'algorithme.

Il faut donc distinguer la notion de *tâche* de celle d'*unité d'exécution*, qui est une notion liée à l'exécution d'un programme parallèle avec un langage et une machine parallèle.

Dans ce qui suit, nous allons utiliser le terme d'«unité d'exécution» — traduction du terme *execution unit*, que nous allons parfois abrévier par UE — pour dénoter un *élément actif qui exécute du code*. Une UE pourrait donc être un des éléments suivants :

- un processeur d'une machine multi-processeurs ;
- un coeur d'une machine multi-coeurs ;
- un *thread* matériel d'une machine multi-*threaded* ;
- un *thread* d'un langage ou d'une bibliothèque de programmation parallèle ;

Ce dernier cas tient au fait que dans certains langages, les *threads* d'un programme peuvent être considérés comme des unités d'exécution, dans la mesure où on lance au départ un certain nombre de *threads*, qui exécutent ensuite les diverses *tâches* exprimés par l'algorithme et le programme.

Dans les chapitres qui suivent, nous réserverons donc la notion de *thread* pour des objets actifs créés et manipulés par le langage utilisé pour programmer notre algorithme parallèle. Dans ce qui suit, nous traiterons plutôt de tâches, au niveau logique et algorithmique.

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.**

Au niveau algorithmique, on commence par identifier les tâches les plus fines possibles (granularité aussi fine que nécessaire en fonction du problème : voir plus bas), sans tenir compte des ressources ou contraintes de la machine — en d'autres mots, on suppose une machine «idéale» avec des ressources illimitées.

Ensuite, on détermine les dépendances entre ces tâches : si une tâche T_1 doit s'exécuter avant une tâche T_2, par exemple parce que T_1 produit un résultat utilisé par T_2, alors on dit qu'il y a une dépendance (de données) de T_1 vers T_2. On peut ainsi construire un **graphe des dépendances de tâches**. Ce graphe ne permet ensuite de mieux comprendre le comportement de l'algorithme parallèle.

C'est souvent aussi à partir de ce graphe qu'on pourra développer *un programme parallèle* efficace, qui tiendra compte des ressources matérielles — par exemple, pour diminuer les coûts de synchronisation et communication, on pourra décider de combiner ensemble des petites tâches pour obtenir des tâches plus grosses. On traitera de ces questions dans un chapitre ultérieur.

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 trois coefficients a, b et c, tels que :

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

Une racine r de $p(x)$ est une valeur telle que $p(r) = 0$. Pour un polynome de 2^e degré, la formule suivante permet de trouver les racines, en supposant ici qu'au moins une racine existe ($b^2 \geq 4ac$) :

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

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

Pour cet exemple, nous allons décomposer le calcul de r_1 en une suite d'instructions simples dites «à trois adresses», soit deux opérandes, une destination pour le résultat et un opérateur — donc des pseudo-instructions machines ou du pseudo code-octet. Ce seront alors ces instructions qui représenteront nos tâches à paralléliser.
Remarque : En pratique, on ne procèra généralement pas à une décomposition aussi fine des instructions. Ici, on le fait pour illustrer plus clairement les notions de dépendances et de graphes de dépendances.

Voici un segment de code pour calculer la racine r_1 et l'affecter à la variable `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
```

La Figure 3.7 présente un graphe de dépendances pour ces *tâches* :

- Un cercle dénote une tâche. Son contenu indique l'instruction à exécuter pour cette tâche.
- Une flèche allant d'une tâche vers une autre indique que la deuxiême tâche — celle au bout de la flèche — ne peut s'exécuter que lorsque la première a

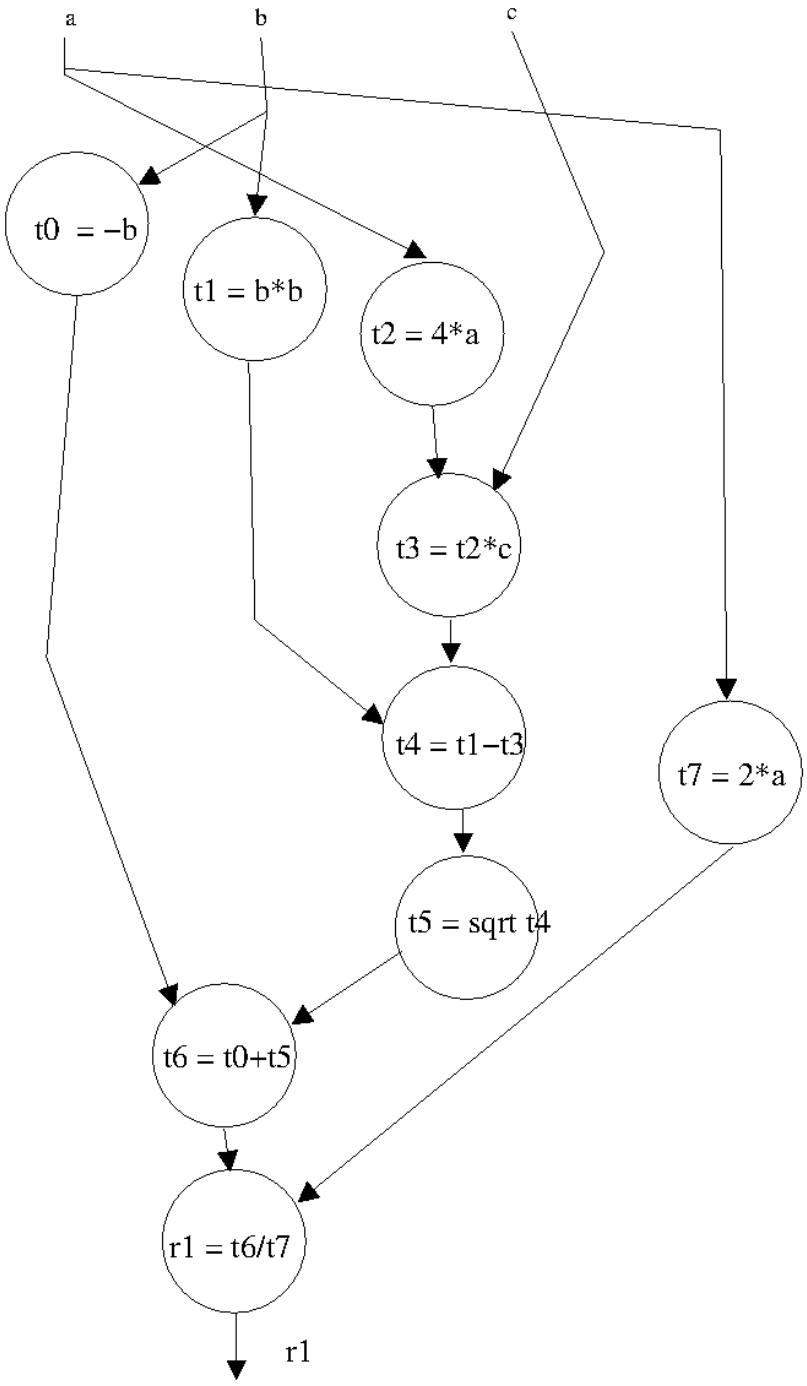

Figure 3.7: Graphe de dépendances des tâches pour le calcul de la racine r_1 d'un polynome de 2^e degré.

complété son exécution. Une dépendance impose donc un ordre d'exécution séquentiel entre ces deux tâches.

Notons qu'ici, il s'agit strictement de *dépendances de données* : une tâche T_2 — par ex., `r1 = t6/t7` — dépend d'une autre tâche T_1 — par ex., `t7 = 2*a` — parce que T_2 *utilise* le résultat calculé par T_1.

3.4.4 Degré de parallélisme

Le **degré de parallélisme** d'un algorithme parallèle à un instant donné est le nombre de tâches qui peuvent être exécutées en même temps à cet instant. Ce nombre peut évidemment varier en cours d'exécution.

La performance d'un programme parallèle dépend de son degré de parallélisme et du nombre de processeurs de la machine : si la machine possède plus de processeurs que le degré de parallélisme, alors le programme pourra s'exécuter à sa plus grande vitesse parallèle.

Pour plus facilement évaluer le degré de parallélisme pour le graphe de la Figure 3.7, on peut le modifier pour obtenir le graphe de la Figure 3.8. Il s'agit exactement des mêmes tâches et des mêmes dépendances — donc du même graphe — mais présentées de façon à ce qu'on voit bien, à un instant donné, les tâches qui sont prêtes à s'exécuter.

Le degré de parallélisme à chaque instant est donc comme suit :

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

Une autre notion intéressante pour comprendre le comportement d'un programme parallèle est celle de «*degré moyen de parallélisme*» = nombre moyen de tâches qui peuvent être exécutées en parallèle tout au long des différents moments de l'exécution. Ainsi, dans notre exemple, le degré moyen de parallélisme serait le suivant :

$$\frac{4+1+1+1+1+1}{6} = \frac{9}{6} = 1.5$$

Lorsque ce degré moyen de parallélisme est *semblable* au degré maximum, alors cela implique que les ressources de la machine pourront être utilisées de façon efficiente tout au long de l'exécution du programme. Inversement, si l'écart est grand, alors à certains instants on aura besoin d'un grand nombre de processeurs, alors qu'à d'autres moments on aura besoin de peu de processeurs — donc soit on «acquiert» un grand nombre de processeurs et plusieurs resteront parfois ou souvent inutilisés, soit on utilise moins de processeurs, ce qui augmentera le temps d'exécution puisque certaines tâches indépendanntes ne pourront pas être exécutées de façon parallèle.

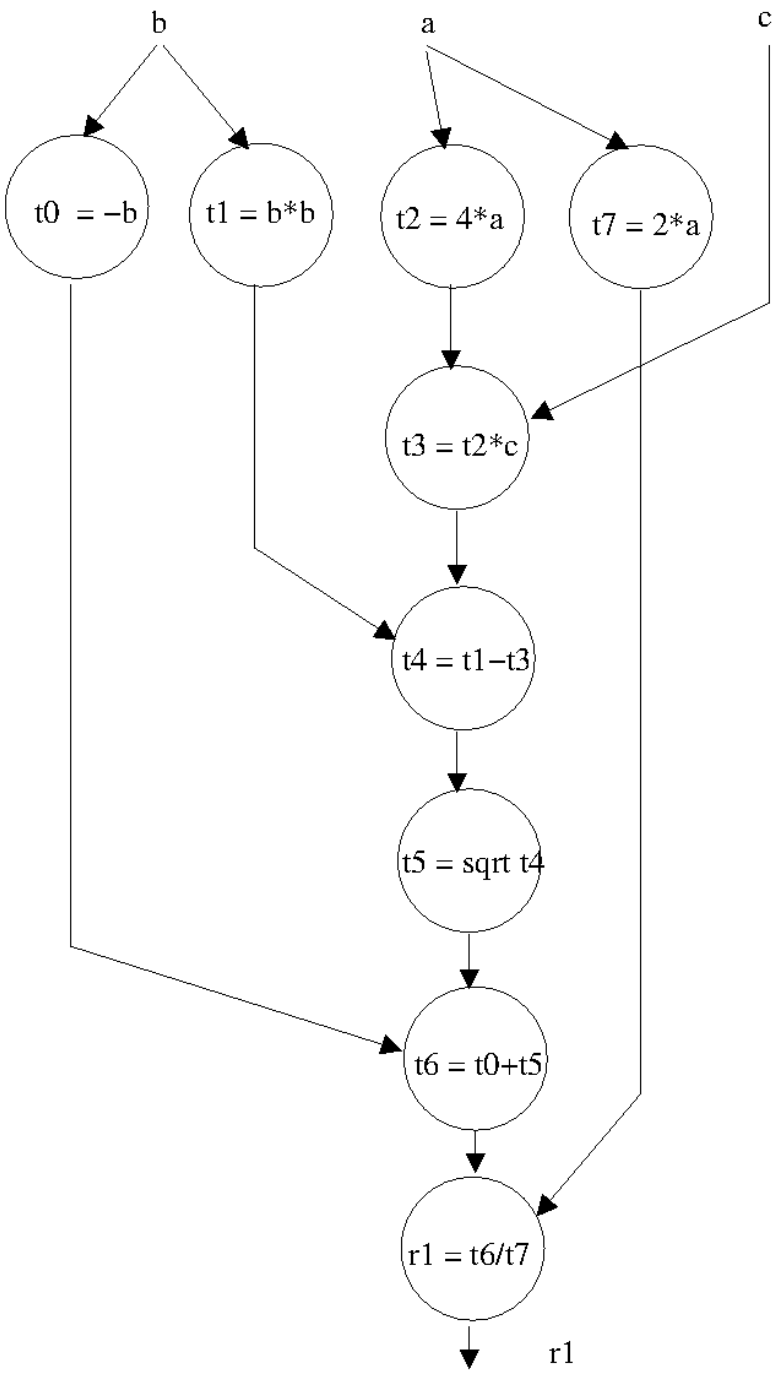

Figure 3.8: Version révisée du graphe de dépendances des tâches pour le calcul de la racine r_1 d'un polynome de 2^e degré.

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

Un **chemin** dans un graphe entre deux sommets 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 dépendances des tâches, le plus long chemin allant de la tâche initiale à la tâche finale est appelé **le chemin critique**. C'est cette longueur — dite **longueur du chemin critique** (*critical path length*) — qui détermine le **meilleur temps d'exécution possible**.

Dans notre exemple, comme on le voit bien dans la Figure 3.8, la longueur du chemin critique est de 5. En supposant que chaque tâche s'exécute en une (1) unité de temps, il faudra donc, *au mieux*, 6 unités de temps pour que l'algorithme s'exécute.

Plus exactement, le *meilleur* temps d'exécution possible varie en fonction du nombre d'unités d'exécution disponibles, comme l'indique le tableau suivant :

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

On remarque que même si le degré maximum de parallélisme est de 4, l'utilisation de 3 UE ne produira pas un meilleur temps d'exécution que si on utilise seulement 2 UE! Ceci s'explique par la longue chaîne séquentielle de dépendances — entre les calculs de `t3, t4, t5, t6` et `r1`!

Temps d'exécution parallèle idéal

Le *temps d'exécution parallèle minimum* — ou **idéal** — d'un algorithme parallèle est obtenu pouvant être obtenu en utilisant *autant d'unités d'exécution* que nécessaire — donc en supposant une machine *idéale*, sans limite sur le nombre d'UE.Ce temps minimum idéal est simplement égal à 1+*la longueur du chemin critique* du graphe de dépendance des tâches.

Choix de la cédule d'exécution

Soulignons que le temps d'exécution pour un algorithme parallèle avec un certain nombre d'UE n'est pas nécessairement unique. Pour obtenir le meilleur temps possible, il faut choisir une bonne *cédule*, i.e., faire un bon choix parmi les différentes tâches à exécuter. Pour notre exemple avec 2 UE, voici une cédule possible, où les 2 UE sont utilisées uniquement durant les trois premières unités de temps :

```
0: t0, t2
1: t1, t3
2: t7, t4
3: t5
4: t6
5: r1
```

Voici par contre une autre cédule, qui conduit à un temps d'exécution plus long :

```
0: t0, t1
1: t2, t7
2: t3
3: t4
4: t5
5: r6
6: r7
```

De façon générale, le problème de sélectionner la meilleure cédule d'ordonnacement est un problème NP-complet ☹

3.4.6 Degré de parallélisme et dimensionnement (*scalability*)

Dans plusieurs problèmes, l'ajout d'UE additionnelles n'a aucun d'effet ☹ C'est le cas pour notre exemple précédent — si on choisit une bonne cédule, l'utilisation de 3 UE n'apporte pas d'amélioration par rapport à l'utilisation de 2 UE. Un tel problème n'est donc *très parallélisable* — pas *dimensionable*.

- On dit d'un *algorithme (ou d'un programme) parallèle* qu'il est **dimensionnable** (*scalable*) lorsque son degré de parallélisme augmente au moins de façon linéaire avec la taille du problème.[5]
- On dit d'une **architecture** qu'elle est dimensionnable si la machine continue à fournir les mêmes performances par processeur lorsque l'on accroît le nombre de processeurs.

L'importance d'avoir un algorithme/programme et une machine dimensionnables vient de ce que cela permet de résoudre des problèmes de plus grande taille sans augmenter le temps d'exécution, et ce simplement en augmentant le nombre de processeurs utilisés.

[5] «*Scalability is a parallel system's ability to gain proportionate increase in parallel speedup with the addition of more processors.*» [Gra07].

3.4.7 Granularité des tâches

Une notion utile pour développer un programme parallèle efficace est celle de *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.*
>
> Fine-grained parallelism means individual tasks are relatively small in terms of code size and execution time. *The data is transferred among processors frequently in amounts of one or a few memory words.* Coarse-grained is the opposite: data are communicated infrequently, after larger amounts of computation.
>
> *The finer the granularity, the greater the potential for parallelism and hence speed-up, but the greater the overheads of synchronization and communication. In order to attain the best parallel performance, the best balance between load and communication overhead needs to be found.*
>
> *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`

Note : Dans le cas d'un programme parallèle avec mémoire partagée, on peut substituer «communication» par «synchronisation».

On discutera de la notion de granularité plus en détail lorsqu'on traitera de méthodologie de programmation parallèle.

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

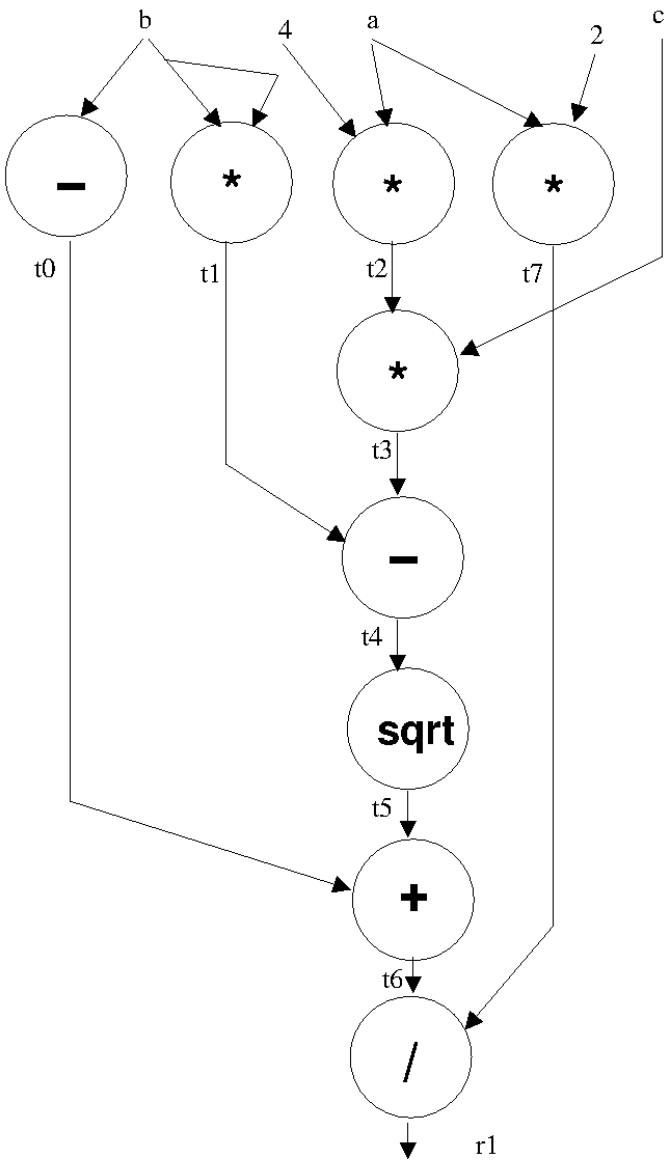

Figure 3.9: Graphe de flux de données pour le calcul de la racine r_1 d'un polynome de 2^e degré.

La Figure 3.9 présente une façon simplifiée de représenter le graphe de dépendances des tâches pour le calcul de la racine d'un polynome de 2^e degré, et ce à l'aide d'un *graphe de flux de données*.

Dans ce graphe, une tâche est simplement indiquée par l'opération arithmétique devant être exécutée — avec un opérateur unaire si la tâche ne possède qu'une seule entrée (par ex., `sqrt`), ou un opérateur binaire si la tâche possède deux entrées. Les dépendances sont alors toutes des dépendances de données.

Somme de deux tableaux

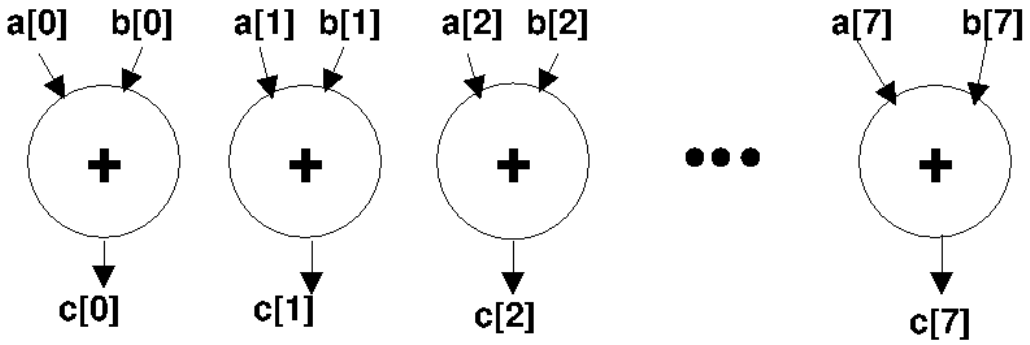

Figure 3.10: Graphe de flux de données pour un algorithme parallèle calculant la somme de deux tableaux de 8 éléments.

La Figure 3.10 présente un graphe de flux de données pour un algorithme parallèle effectuant la somme de deux tableaux de 8 éléments, algorithme pouvant être exprimé par le code PRuby suivant :

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

On constate qu'il n'y a *aucune* dépendances entre les tâches. On dit d'un tel algorithme qu'il est embarrassingly parallel.

Sommation des éléments d'un tableau

On a un tableau de 8 éléments et on veut faire la somme de ces éléments. Voici une suite d'instructions à trois adresses permettant d'effectuer ce calcul :

```
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]
```

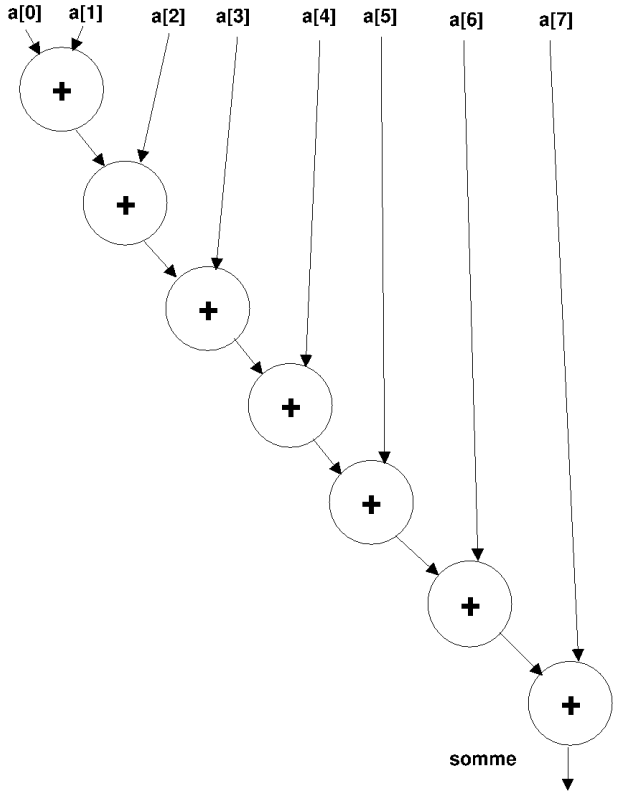

Figure 3.11: Graphe de flux de données pour un algorithme effectuant la sommation des éléments d'un tableau de 8 éléments.

La Figure 3.11 présente le graphe de flux de données associé à ce calcul. On constate que cette façon de procéder *est strictement séquentielle* — 7 unités de

temps sont nécessaires pour effectuer la somme de 8 éléments. L'algorithme est donc $O(n)$.

Voici une autre suite d'instructions à trois adresses permettant d'effectuer ce même calcul, mais de façon différente :

```
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
```

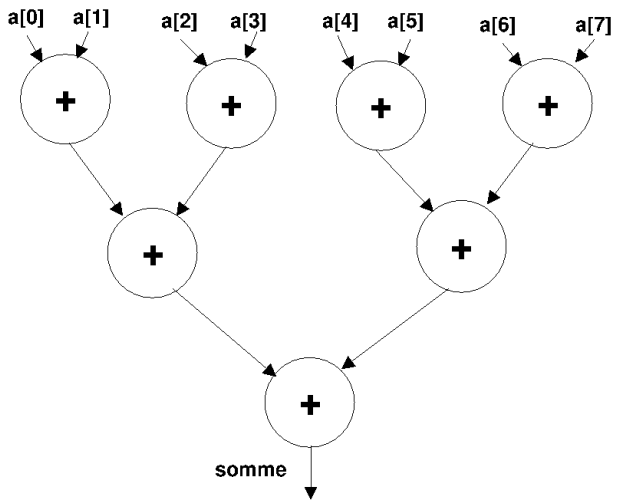

Figure 3.12: Deuxième version — parallélisable! — d'un graphe de flux de données pour un algorithme effectuant la sommation des éléments d'un tableau de 8 éléments.

La Figure 3.12 présente le graphe de flux de données associé à ce calcul. On constate que cette façon de procéder *peut se faire de façon parallèle* — — 3 unités de temps sont nécessaires pour effectuer la somme de 8 éléments. L'algorithme est $O(lg\ n)$!

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
```

Le Programme Ruby 3.7 présente un algorithme récursif pour effectuer la sommation des éléments d'un tableau.

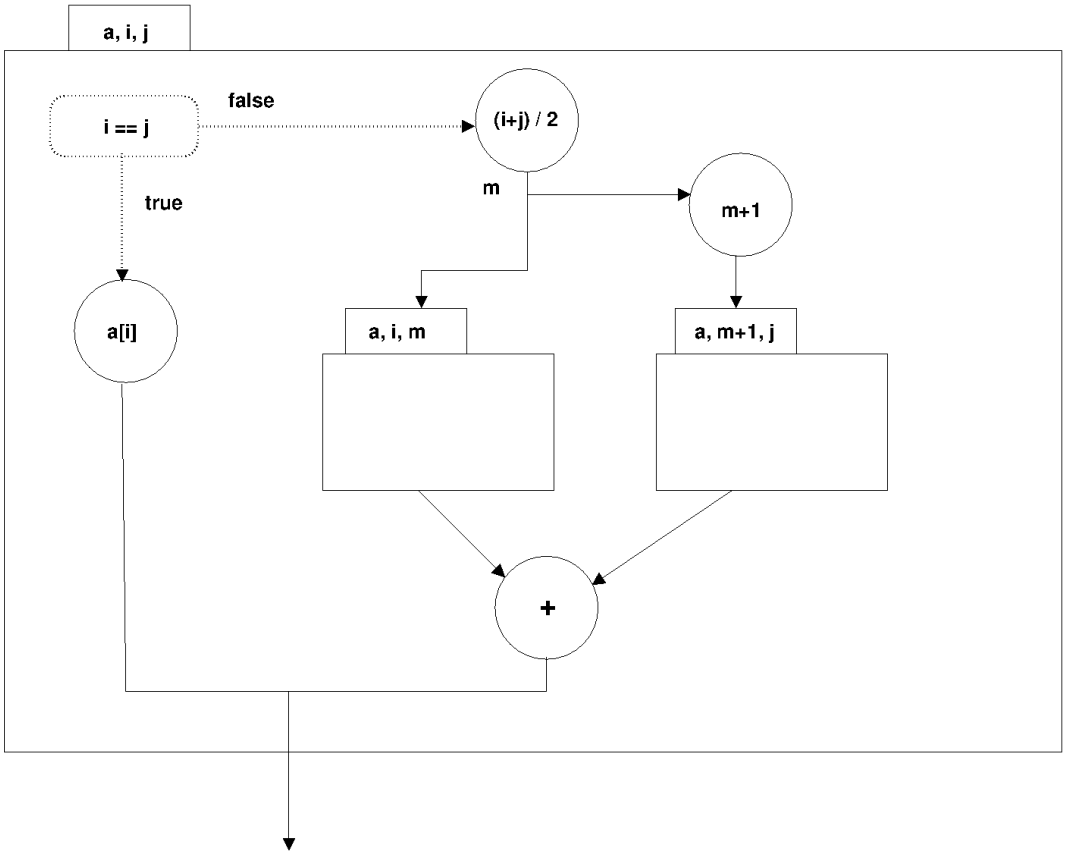

Figure 3.13: Graphe de dépendances de tâches pour le calcul récursif de la sommation des éléments d'un tableau.

La Figure 3.13 présente un graphe de dépendances pour cet algorithme récursif :

- On représente une activation de fonction avec un rectangle, les arguments de la fonction étant indiqués par un petit rectangle dans le coin supérieur gauche.
- Certaines tâches — branche `then` vs. branche `else` — sont exécutées uniquement si une certaine condition est satisfaite. Dans ce cas, on parle alors d'une *dépendance de contrôle*. Ici, ce type dépendance est indiqué par une flèche en pointillée, la tâche qui calcule la condition étant indiquée par un rectangle pointillé aux coins arrondis.

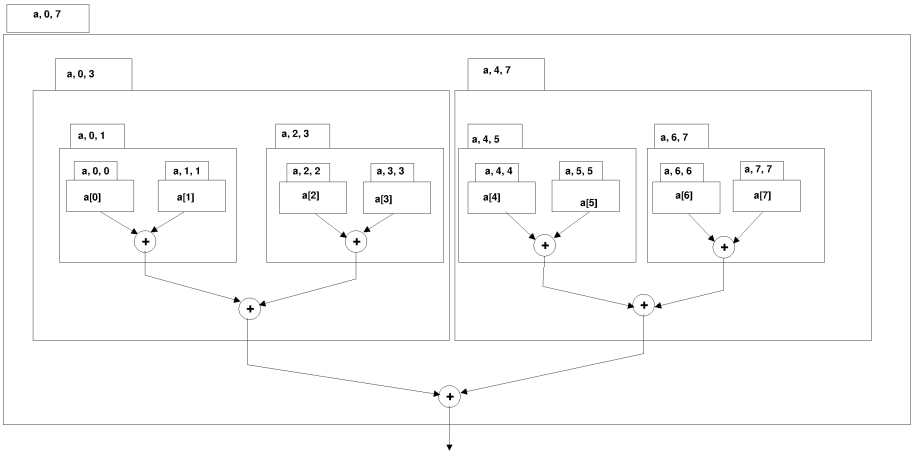

Figure 3.14: Graphe de dépendances de tâches pour le calcul récursif de la sommation des éléments d'un tableau de 8 éléments.

La Figure 3.14 présente le graphe de dépendances généré pour le cas spécifique d'une sommation d'un tableau comptant 8 éléments. On constate que la *structure* est semblable à celle du graphe de la Figure 3.12.

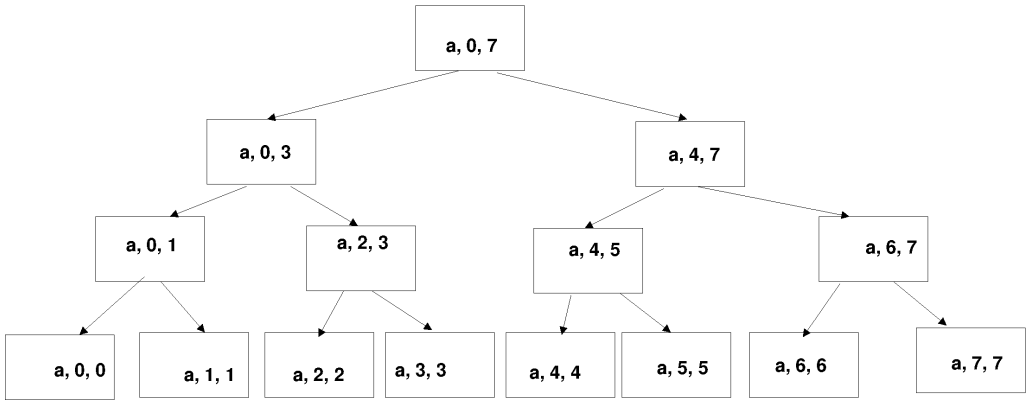

Figure 3.15: Arbre d'activations des instances de fonction pour le calcul récursif de la sommation des éléments d'un tableau de 8 éléments.

Quant à la Figure 3.15, elle présente un *arbre d'activation des fonctions* pour le calcul récursif de la somme des éléments d'un tableau comptant 8 éléments. On constate que la structure de cet arbre est elle aussi semblable à la structure du

graphe précédent, mais *renversée* et avec moins de détails : on indique uniquement les dépendances *descendantes* entre les activations de fonctions, les dépendances *ascendantes* — pour «remonter» le résultat — étant implicites : une dépendance indique donc un appel de la fonction et ensuite la réception du résultat retourné.

Dans ce qui suit, nous utiliserons souvent cette forme d'arbre d'activations pour illustrer le comportement d'algorithmes récursifs.

3.5 Indépendance entre *threads*

- L'**ensemble de lecture** (*read set*) d'une partie de programme est l'ensemble des variables *lues*, *mais non modifiées*, par cette partie de programme.
- L'**ensemble d'écriture** (*write set*) 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 (l'intersection est *vide*) tant de l'ensemble de lecture que 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 si et seulement si les trois conditions suivantes sont satisfaites :

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

  En d'autres mots, P_1 et P_2 sont indépendantes si (i) chaque partie ne modifie que des variables qui ne sont pas lues par l'autre et (ii) qu'elles écrivent dans des variables différentes.

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

Les exemples qui suivent visent à illustrer la différence entre «concurrence» et «parallélisme». Rappelons la caractérisation de R. Pike présentée plus haut :

> *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`

Dans le contexte spécifique de Ruby, une autre caractérisation intéressante est celle présentée par E. Quran :

> *Ruby* **concurrency** *is when two tasks can start, run, and complete in overlapping time periods. It doesn't necessarily mean, though, that they'll ever both be running at the same instant (e.g., multiple threads on a single-core machine). In contrast,* **parallelism** *is when two tasks literally run at the same time (e.g., multiple threads on a multicore processor).*
>
> *[...]*
>
> *Ruby concurrency without parallelism can still be very useful, though, for tasks that are IO-heavy*[6] *(e.g., tasks that need to frequently wait on the network).*
>
> *Source :* `https://www.toptal.com/ruby/ruby-concurrency-and-parallelism-a-practical-primer`

Dans ce qui suit, nous allons présenter un problème avec parallélisme dont le programme est *CPU-bound*, puis un problème avec concurrence dont le programme est *IO-bound*. Dans ce dernier cas, nous verrons que même si Ruby/MRuby ne permet pas le «vrai» parallélisme, un programme concurrent peut quand même s'exécuter plus rapidement qu'un programme séquentiel.

[6] *IO-bound.*

Un problème avec parallélisme

Le programme Ruby 3.8 présente deux versions d'une fonction pour faire la «somme» des éléments d'un tableau ; plus précisément, la fonction effectue la somme *des éléments mis au carré*... simplement pour «augmenter» un peu le temps d'exécution.

La version parallèle décompose le tableau en 10 tranches, chacune traitée par un `Thread` indépendant. Si on examine les temps requis pour diverses exécutions — figure 3.16, colonne `real` et colonne avec le **nombre en rouge** — on constate qu'avec Ruby/MRI, il n'y a *aucune accélération*, alors qu'on obtient une accélération avec JRuby. Pour une raison technique — GIL = *Global Interpreter Lock* — Ruby/MRI *ne permet* **pas** *l'exécution parallèle* ☹

Un problème avec concurrence

Le programme Ruby 3.9 présente deux versions d'une fonction pour lire et analyser des URIs, donc faire des accès au Web. La version concurrente génère un `Thread` pour chaque URI. Si on examine les temps requis pour diverses exécutions — figure 3.17, colonne `real` et colonne avec le **nombre en bleu** — on constate maintenant qu'on obtient une accélération tant dans la version Ruby/MRI que dans la version JRuby ☺ Bien que Ruby/MRI ne supporte pas l'exécution parallèle, Ruby/MRI supporte quand même l'exécution concurrente — ici, les entrées/sorties exécutées de façon asynchrone et concurrente.

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, nb_threads) ) }
  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.

```
COURS = ['INF3135', 'INF3140', 'INF4110', 'INF4170', 'INF5171',
         'INF600A', 'INF7440', 'INF8541', 'MGL7460', 'MGL7160']

def titre_du_cours( cours )
  uri = "http://www.labunix.uqam.ca/~tremblay/#{cours}/index.html"
  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 «somme» des éléments d'un tableau. (Exécution sur `japet.labunix.uqam.ca`)

```
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 lecture et l'analyse d'une liste d'URIs. (Exécution sur `japet.labunix.uqam.ca`)

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?

Références

[Gra07] J.R. Graham. Integrating parallel programming techniques into traditional computer science curricula. *Inroads — SIGCSE Bulletin*, 39(4):75–78, Dec, 2007.

[SMR09] M.J. Sottile, T.G. Mattson, and C.E. Rasmussen. *Introduction to Concurrency in Programming Languages*. Chapman and Hall/CRC, 2009.