8. Méthodologie pour la programmation parallèle et patrons d'algorithmes

8.1 Introduction

1. Méthodologie — «**heuristique**» — pour la programmation avec *threads* qui **s'inspire** de l'approche PCAM de I. Foster :

`http://wotug.org/parallel/books/addison-wesley/dbpp/`

2. Patrons d'algorithmes parallèles de Sottile, Mattson, et Rasmussen.

8.2 Approche PCAM

L'approche **PCAM** a été conçue pour un modèle de programmation parallèle *par échanges de messages*.

On peut quand même s'en inspirer pour la programmation avec *threads* et *variables partagées*.

8.2.1 Objectifs

Les objectifs (contradictoires) qui sont visés :

- Réduire les coûts de synchronisation.
- Distribuer le travail de façon uniforme — «équilibrer la charge».
- Conserver une flexibilité quant à la *scalability* — si on ajoute des processeurs, on devrait augmenter l'accélération.

8.2.2 Étapes de l'approche PCAM

Les étapes de l'approche PCAM sont :

1. *Partitionnement* : On identifie *tout le parallélisme disponible.*

   Tâches **de très fine granularité** — aussi parallèle que possible.

2. *Communication* : On identifie les dépendances entre les tâches.

3. *Agglomération* : On **regroupe** les tâches pour obtenir des tâches plus grosses.

4. *Mapping* : On associe les tâches à des *threads*.

8.3 Méthodologie de programmation parallèle avec *threads* inspirée de l'approche PCAM

1. On commence par **identifier les tâches les plus fines possibles** — on suppose une machine «idéale».

Stratégies typiques (prochaine section) :

- Parallélisme de données
- Parallélisme de résultat
- Parallélisme récursif diviser-pour-régner
- Etc.

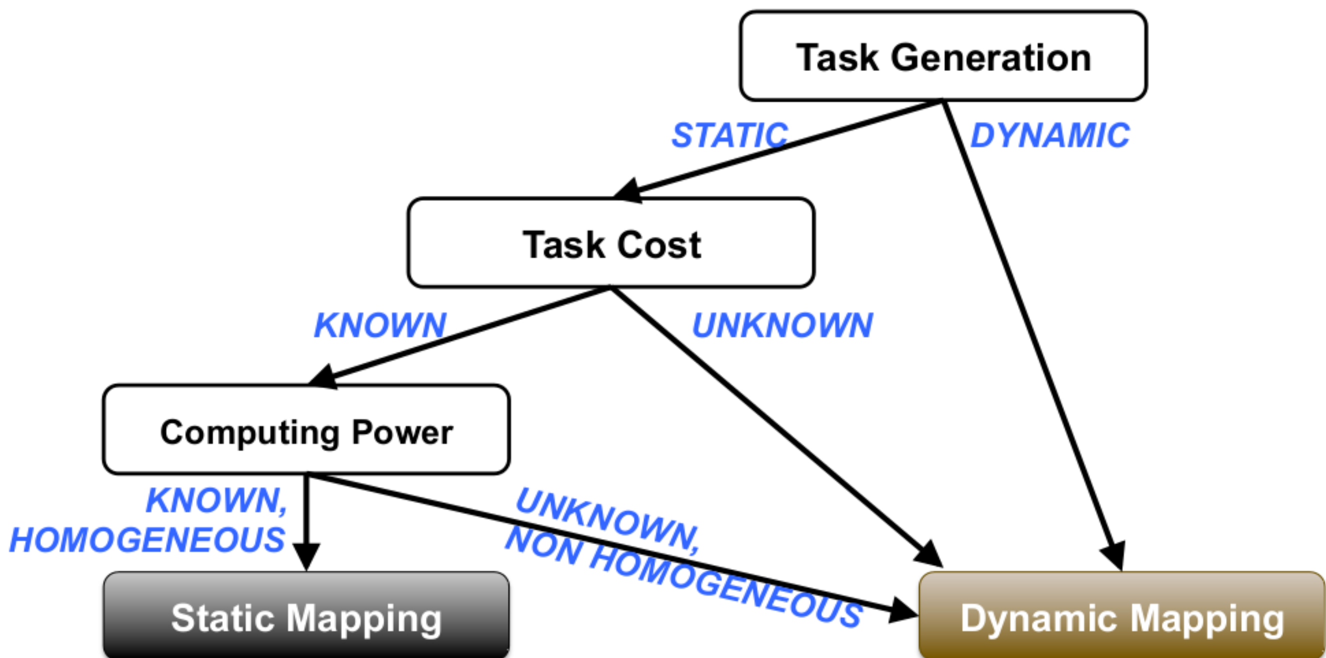
2. On **analyse les dépendances** entre les tâches

   Par exemple, on peut produire un **graphe des dépendances** entre les tâches.

3. On **agglomère** (sur la base des dépendances) un groupe de tâches pour produire des tâches plus **grossières**.

   Par exemple : regroupement par blocs d'éléments adjacents, par groupes d'éléments distribués cycliquement, etc.

4. On associe les tâches à des *threads* :



(a) Association via parallélisme récursif :
Si l'algorithme sous-jacent est récursif

Pour limiter le nombre de *threads* et augmenter la granularité : *seuil de récursion*.

(b) Association statique :

Si toutes les tâches sont connues de façon statique — **i.e., exécuter une tâche ne génère pas de nouvelles tâches** — et requièrent **le même travail**, alors association *statique* tâche/*thread*, i.e., on lance un *thread* pour chaque tâche.

(c) Association dynamique :

Si

i) le travail requis varie d'une tâche à une autre

ou si

ii) le nombre de tâches varie dynamiquement (une tâche peut en créer de nouvelles),

alors association *dynamique* tâche/*thread* : — par ex., «`dynamic: k`» ou un «sac de tâches».

Important : Il faut que le nombre de tâches soit (nettement) plus grand que le nombre de *threads*.

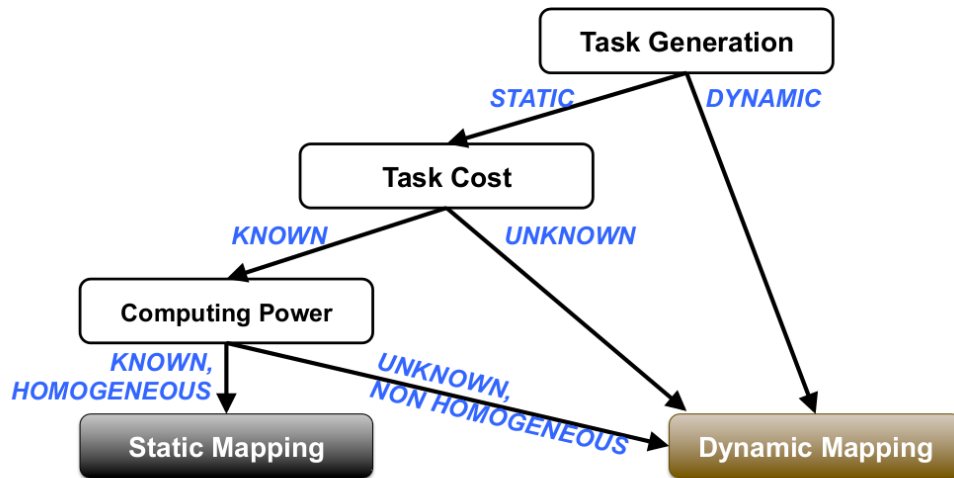


Figure 8.1: Arbre de décision pour choisir entre une stratégie statique vs. dynamique d'association (*mapping*) des tâches aux *threads*.
Source : `http://www.dais.unive.it/~calpar/`

```
DEBUT

  Décomposer le problème dans le plus grand nombre possible
      de tâches indépendantes
    - Décomposition des données (parallélisme de résultat)?
    - Décomposition fonctionnelle (parallélisme de spécialistes)?


  SI nombre de taches > nombre de processeurs ALORS

     SI les diverses taches sont similaires entre elles ALORS
       Agglomérer les taches (granularité fine ou moyenne) pour en faire des
         taches de plus grande granularité (moyenne ou grossière)
     FIN

     SI la quantité de travail est +/- la même pour chaque tache ALORS

       SI nombre de taches = nombre de processeurs ALORS
         ⇒ Association statique tache/processeur
       SINON # Nombre de taches > nombre de processeurs
         ⇒ Association statique et cyclique tache/processeur ?
         ⇒ Association dynamique tache/processeur (sac de taches) ?
       FIN

     SINON # quantité de travail variable
       ⇒ Association statique et cyclique tache/processeur ?
       ⇒ Association dynamique tache/processeur (sac de taches) ?
     FIN

  SINON # nombre de taches ≤ nombre de processeurs

     SI les données traitées sont des flux de donnees ALORS
       ⇒ Parallélisme de specialistes/de flux
     SINON
       ⇒ Pas grand chose à faire ☹
     FIN

  FIN

FIN
```

Pseudocode 8.1: Heuristique pour la conception d'un programme parallèle avec *threads*.

8.4 Effet de la granularité sur les performances

- Granularité trop fine $\Rightarrow$ Mauvaises performances parce que surcoûts élevés — création de *threads*, synchronisation, etc. ☹
- Donc…
- On agglomère des tâches pour réduire ces surcoûts
- Mais…
- Granularité trop grossière $\Rightarrow$ pas assez de parallélisme et/ou charge mal équilibrée ☹

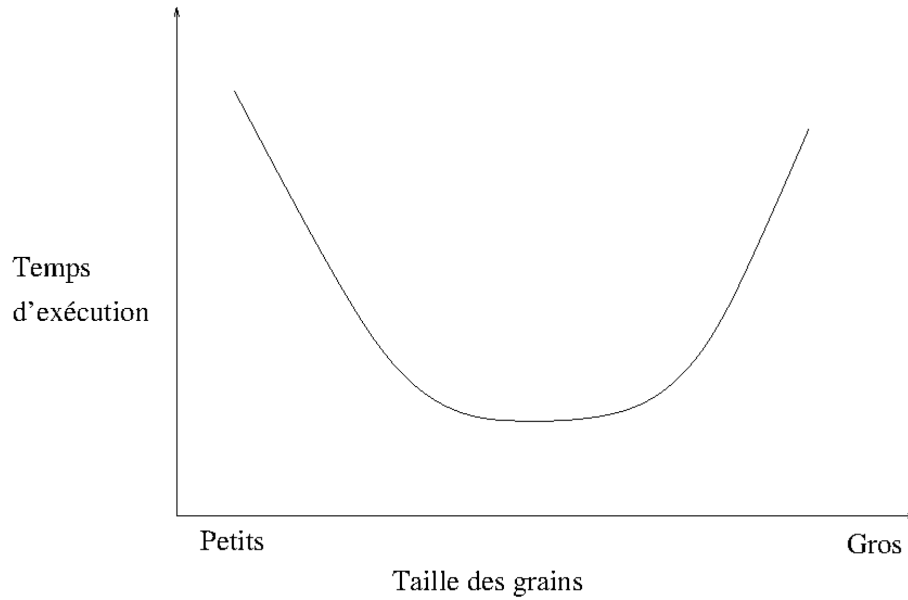


Figure 8.2: Courbe illustrant l'effet général de la taille des grains (des tâches) sur le temps d'exécution.

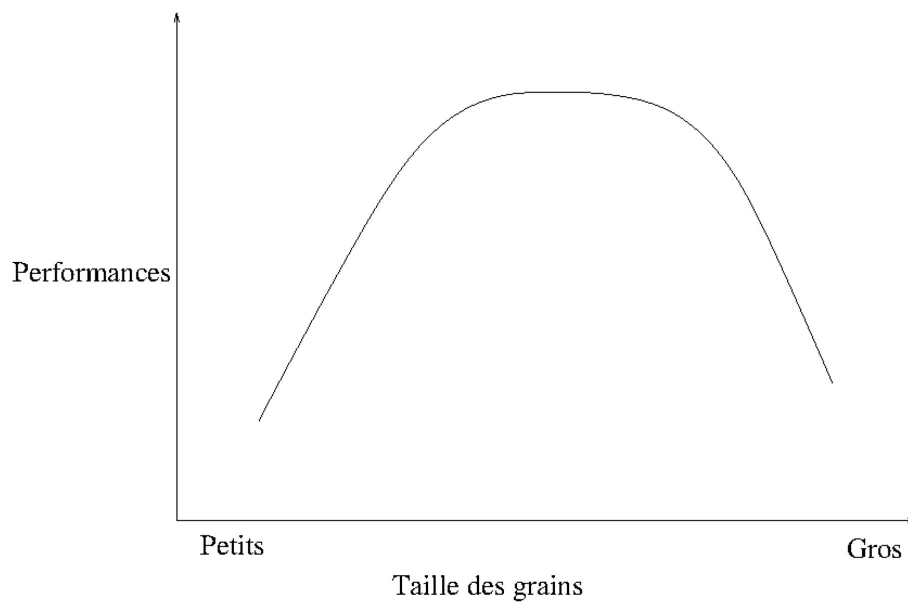


Figure 8.3: Courbe illustrant l'effet général de la taille des grains (des tâches) sur la performance.

Fait : On va retrouver ces courbes en $\cup/\cap\ldots$ dans de nombreuses situations — accélération et nombre de *threads*, efficacité et taille du problème, etc.

8.5 Patrons d'algorithmes parallèles

Patrons présentés dans «*Introduction to Concurrency in Programming Languages*»

(Sottile, Mattson et Rasmussen, 2009).

Il s'agit de «patrons de conception», donc de **stratégies de conception d'algorithmes** :
un patron **d'algorithme** peut souvent être mis en oeuvre par plusieurs patrons **de programmation**.

Ces stratégies permettent — étape P CAM — d'identifier les tâches qui peuvent s'exécuter en parallèle.

8.5.1 Parallélisme de tâches (*task parallelism*) :

grand nombre de tâches pouvant s'exécuter en parallèle.

8.5.2 Parallélisme de données (*data parallelism*) :

grande quantité de données sur lesquelles on applique la même série d'opérations.

8.5.3 Parallélisme de flux (*stream parallelism*) :

flux de données qu'on doit traiter, avec un traitement décomposé en **étapes**.

8.5.1 Parallélisme de tâches

Dans certains systèmes complexes, le système est **naturellement** composé d'un grand nombre de composants indépendants.

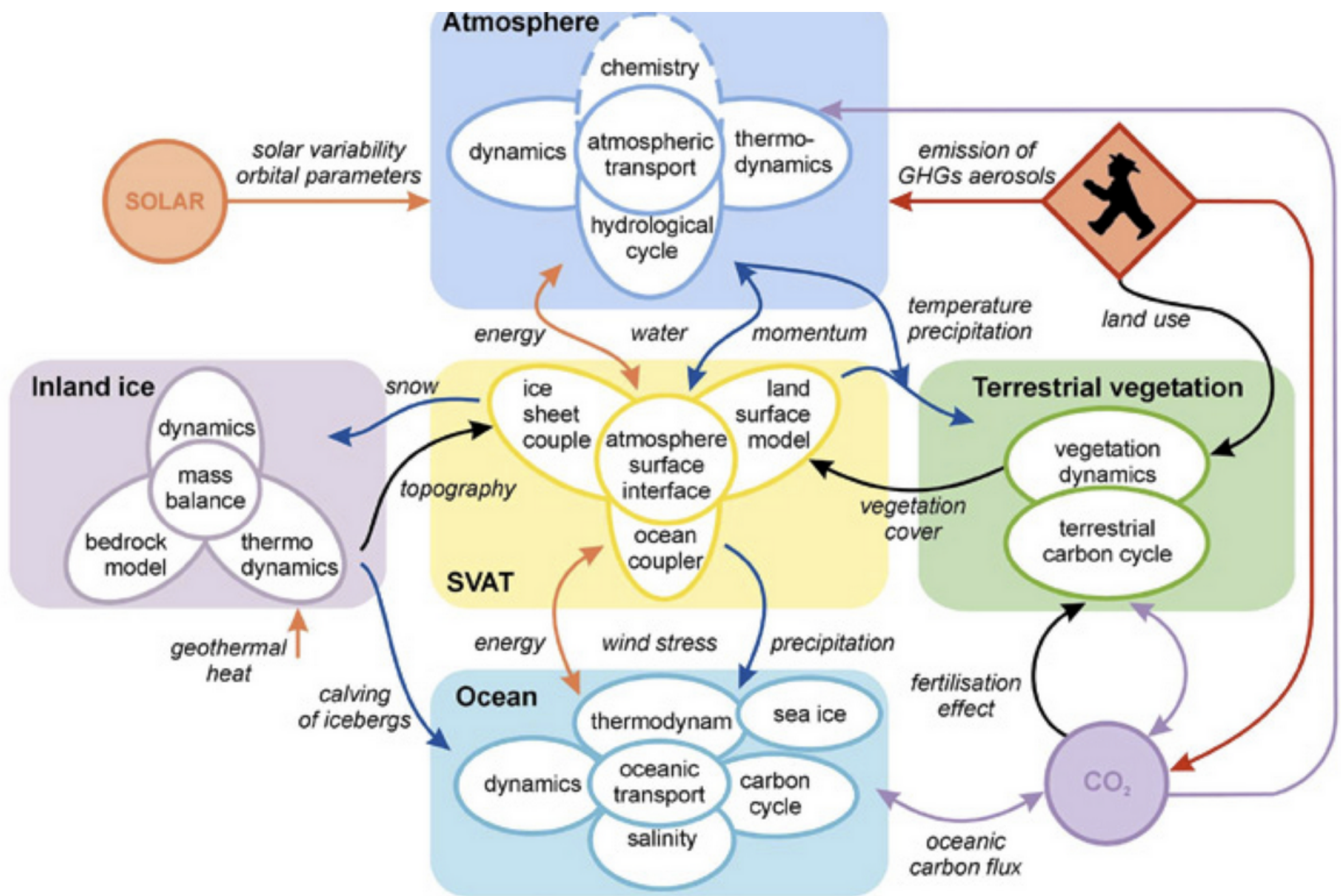


Figure 8.4: Architecture d'un système pour la modélisation du climat (source : `http://www.cs.toronto.edu/~sme/PMU199-climate-computing/pmu199-2012F/coupled-architecture.jpg`).

Parallélisme «embarrassant»

Certains problèmes peuvent être décomposés en un grand nombre de parties indépendantes, où chaque partie peut s'exécuter **sans aucune dépendance**.

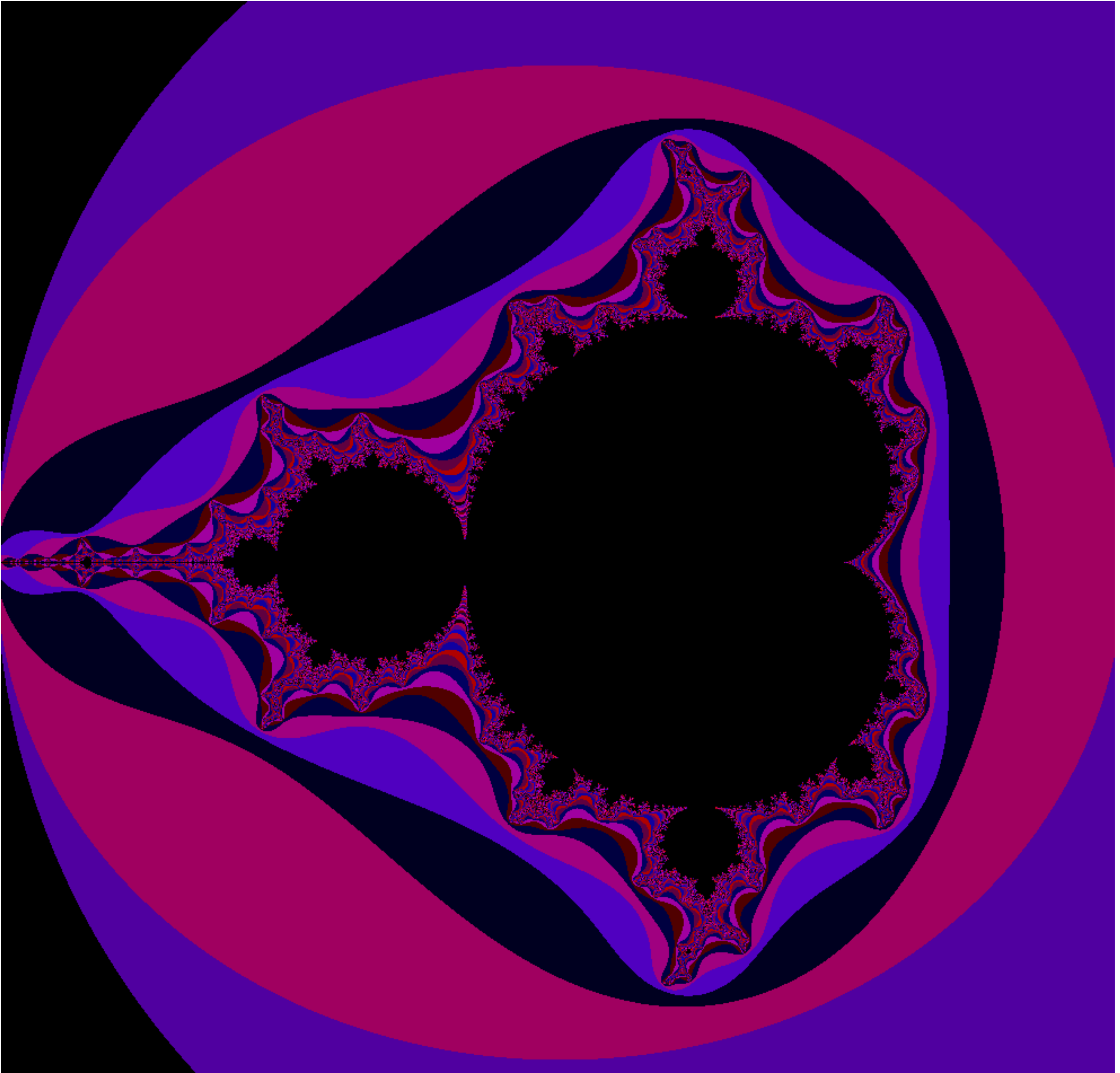


Figure 8.5: Représentation graphique de l'ensemble de Mandelbrot.

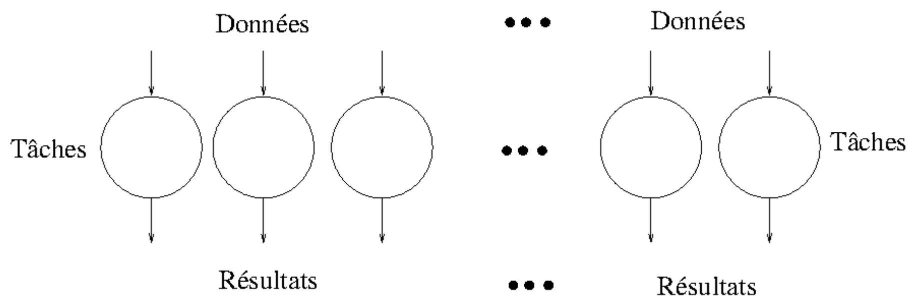


Figure 8.6: Graphe de dépendances de tâches pour un problème avec parallélisme embarrassant.

Parallélisme «semi-embarrassant»

Un problème avec parallélisme «semi-embarrassant» peut être décomposé en un grand nombre de tâches indépendantes, **mais pas tout à fait complètement** : les tâches interagissent, mais *souvent uniquement à la fin de l'exécution.*

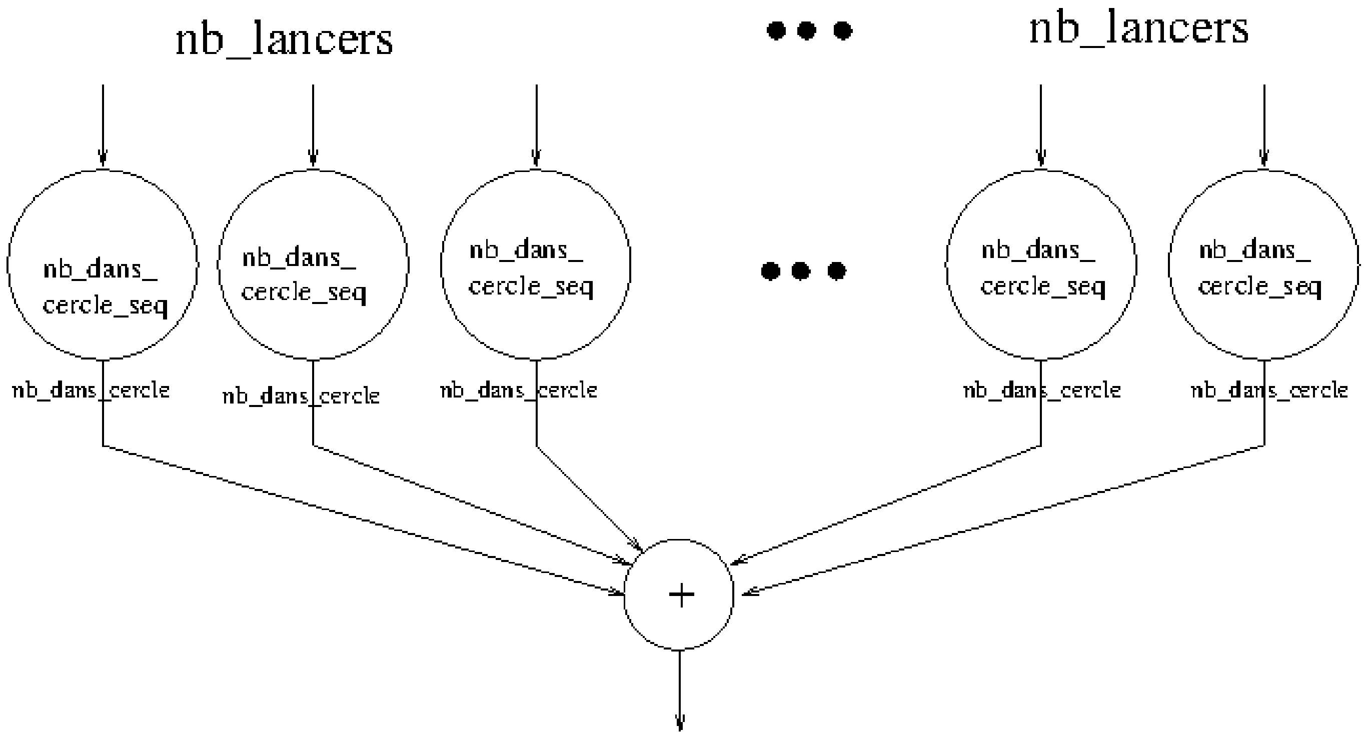


Figure 8.7: Graphe de dépendances de tâches pour un problème avec parallélisme semi-embarrassant — approximation de π par une méthode Monte Carlo.

Parallélisme récursif

De nombreux problèmes peuvent être résolus par une approche «diviser-pour-régner»
⇒ parallélisme récursif!

Approche diviser-pour-régner

```
SI le problème est simple ALORS
  On trouve la solution directement
SINON
  On décompose le problème en sous-problèmes
  On résout récursivement les sous-problèmes
  On combine les solutions des sous-problèmes
            pour obtenir la solution du problème initial
FIN
```

On a une approche «diviser-pour-régner dichotomique» lorsqu'on décompose en deux sous-problèmes.

```
FONCTION resoudre_dpr_2( probleme )
DEBUT
  SI est_simple( probleme ) ALORS
    # Cas non-récursif
    RETOURNER resoudre_probleme_simple( probleme )
  SINON
    # Cas récursifs avec deux sous-problèmes
    prob1, prob2 = decomposer( probleme )

    sol1 = resoudre_dpr_2( prob1 )
    sol2 = resoudre_dpr_2( prob2 )

    RETOURNER combiner_solutions( sol1, sol2 )
  FIN
FIN
```

Pseudocode 8.2: Pseudocode décrivant un algorithme récursif dichotomique.

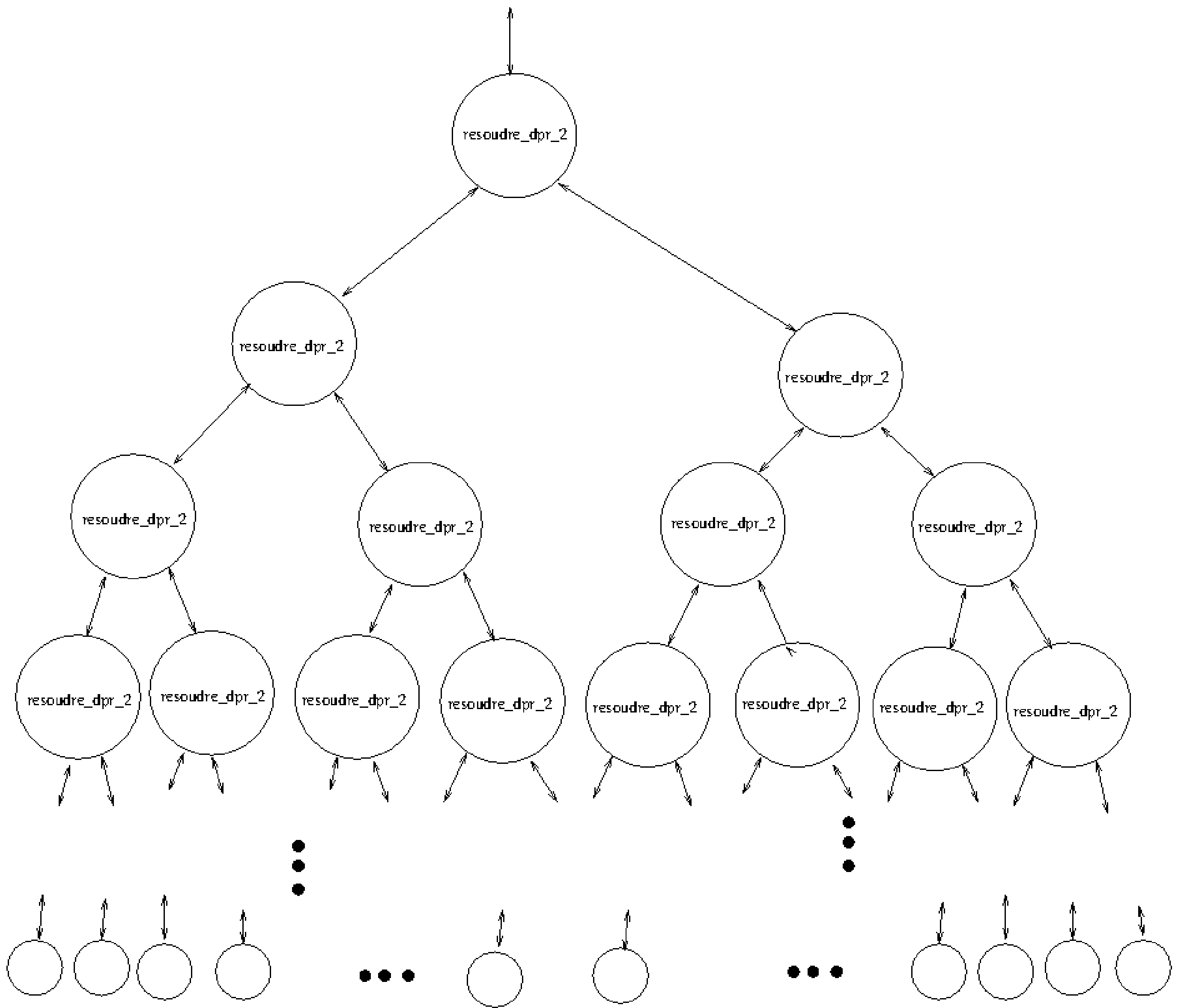


Figure 8.8: Graphe de dépendances de tâches pour un problème avec parallélisme récursif dichotomique — appels récursifs à `resoudre_dpr_2`.

Patron de conception d'algorithme vs. patron de programmation

Il existe souvent une **correspondance** directe entre patron de **conception** et patron de **programmation**... mais plusieurs patrons de programmation peuvent aussi être plus utilisés

Exemple :

Un problème pourrait avoir une solution naturellement récursive, mais du parallélisme *fork-join* pourrait ne pas convenir :

- Ce mécanisme n'est pas supporté par le langage cible ☹
- Les surcoûts associés à la création dynamique de *threads* sont trop élevés ☹
- Etc.

Mais, il est parfois possible de mettre en oeuvre une stratégie diviser-pour-régner… **sans utiliser de récursion**!

Question : Comment?

8.5.2 Parallélisme de données

Lin et Snyder définissent un calcul avec **parallélisme de données** comme suit :

> un calcul avec **parallélisme de données** en est un dans lequel le parallélisme est obtenu en exécutant, en même temps, une même opération sur différents items de données ; **la quantité de parallélisme augmente donc de façon proportionnelle à la quantité de données**.

Application parallèle et réduction, avec vs. sans construction pour le parallélisme de données

Déjà vu : parallélisme de données fondé sur les **applications** et les **réductions** parallèles.

Avec une opération style *map*, **on part d'une grande quantité de données à traiter** et on applique **en parallèle** une fonction sur cette collection :

```
def map_foo( col )
  col.pmap { |x| foo( x ) }
end
```

Dans un langage **qui ne supporte que le parallélisme de boucles**, on peut quand même réaliser un algorithme fondé sur le parallélisme de données :

```
def map_foo( col )
  res = Array.new(col.size)

  res.peach_index do |k|
    res[k] = foo( col[k] )
  end

  res
end
```

Parallélisme de résultat

Autre stratégie en lien avec le parallélisme de données = **parallélisme de résultat**.

1. On identifie le parallélisme **en partant du produit final**, i.e., en partant du résultat désiré.

2. On attribue à chaque travailleur un **morceau du résultat**.

Remarque : Le parallélisme «de résultat» est une forme de parallélisme de données

- Si on indique simplement «parallélisme de données», c'est qu'on part des données pour identifier les tâches.
- Si on indique «**parallélisme de résultat**», c'est qu'**on part du résultat à calculer pour identifier les tâches**.

Quelques exemples :

1. Somme de deux tableaux.

   De données ou de résultat?

2. Produit de deux matrices.

   De données ou de résultat?

Quelques exemples :

1. Somme de deux tableaux.

   De données ou de résultat conduisent à la même solution!

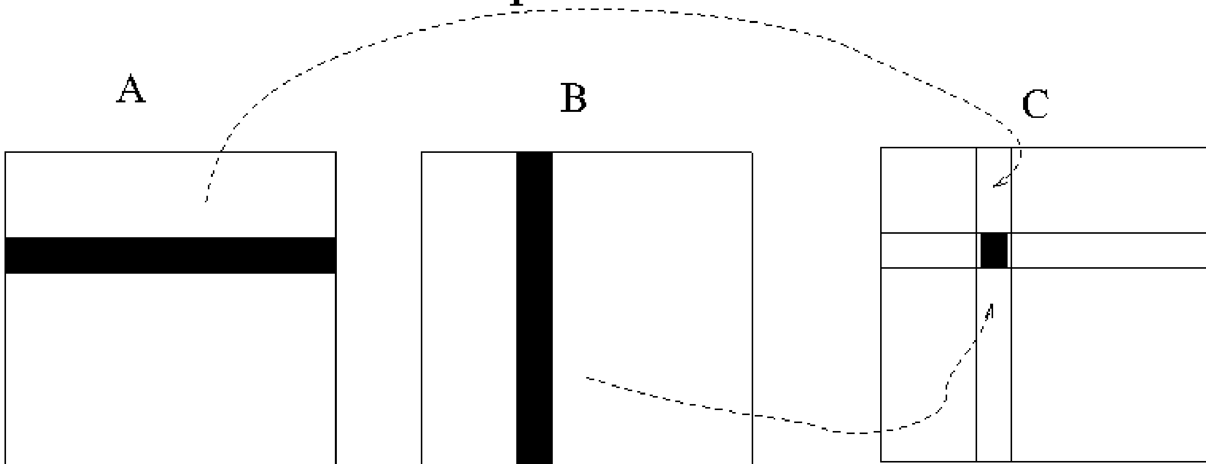
2. Produit de deux matrices.

   De données ou de résultat conduisent à la même solution en mémoire partagée, mais pas en mémoire distribuée!

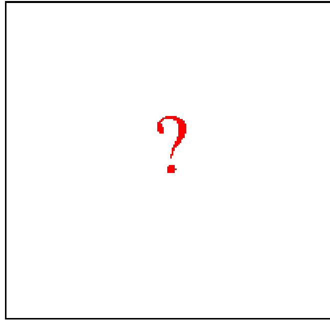
Rappel :

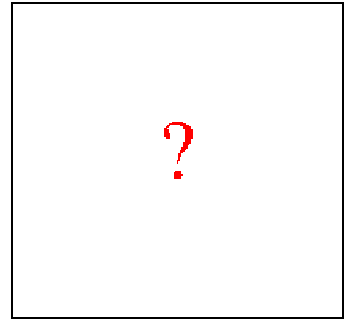
$$C[i,j] = \sum_k A[i,k] \times B[k,j]$$

Illustration des dépendances de calcul :



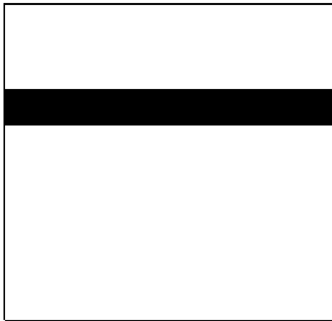
Parallélisme de données avec une ligne de A par *thread*



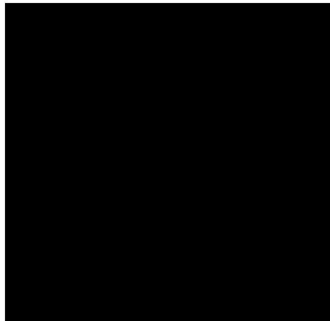


Parallélisme de données avec une ligne de A par *thread*

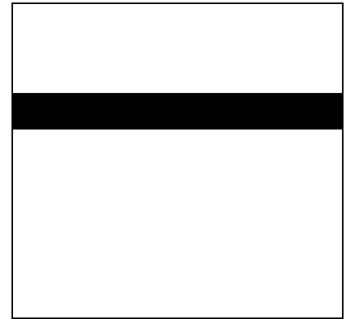
A



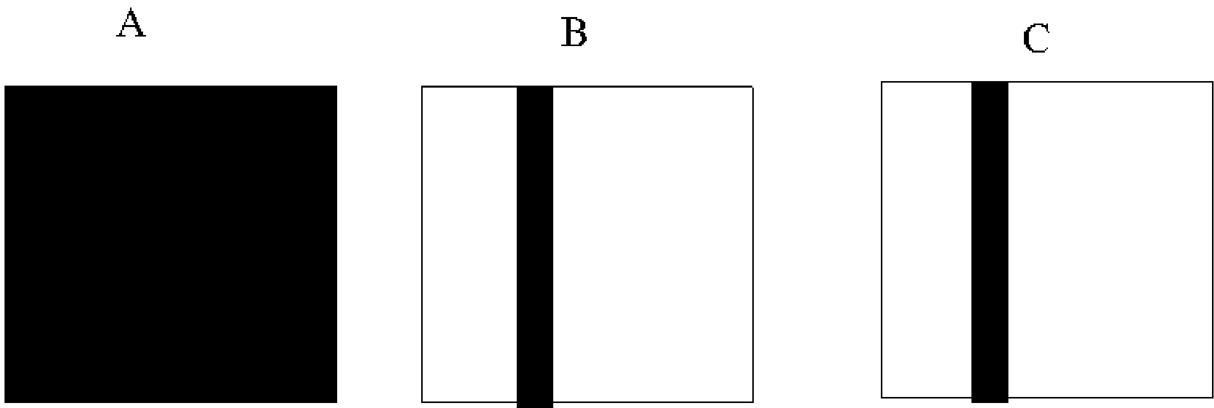
B



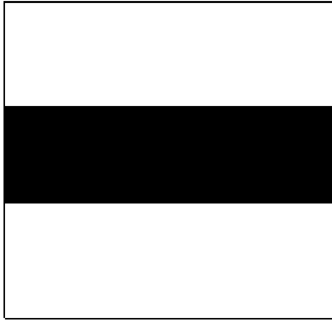
C

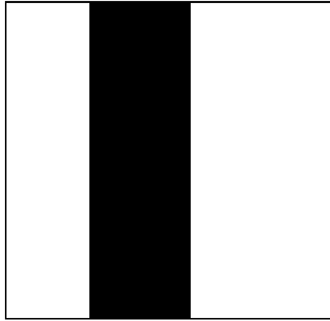


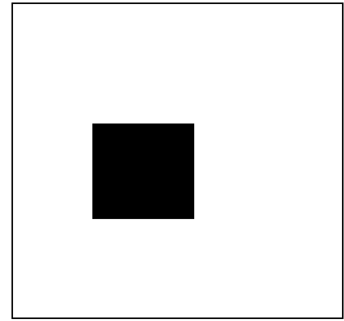
Parallélisme de données avec une colonne de B par *thread*



Parallélisme de résultat avec un bloc de C par *thread*







Décomposition géométrique

> *This pattern is used when (1) the concurrency is based on parallel updates of chunks of a decomposed data structure, and (2) the update of each chunk requires data from other chunks.*
>
> *Source :* `http://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/AlgorithmStructure/GeoDecomp.htm`

On verra un exemple dans un chapitre ultérieur — diffusion de la chaleur dans un cylindre.

On veut paralléliser la fonction `histogramme` présentée plus bas, fonction qui produit un histogramme pour les entiers d'un tableau `elems`.

Chaque nombre d'`elems` est un entier compris entre 0 et `val_max` (incl.). Les bornes du tableau résultant, e.g., `histo`, sont comprises entre 0 et `val_max` (incl.) tel que :

`histo[val]` = *nombre d'occurrences de* `val` *dans* `elems`

Exemple : soit les 12 valeurs et l'appel suivants :

```
elems == [10, 1, 3, 3, 3, 2, 9, 1, 1, 1, 3, 10]

histo = histogramme( elems, 10 )
```

Alors, après l'appel on aura :

```
histo == [0, 4, 1, 4, 0, 0, 0, 0, 0, 1, 2]
```

```
def histogramme_vide( val_max )
  Array.new(val_max + 1) { 0 }
end

def histogramme( elems, val_max )
  histogramme = histogramme_vide(val_max)

  elems.each { |x| histogramme[x] += 1 }

  histogramme
end
```

Exercice 8.1: Production d'un histogramme pour une série d'entiers bornés.

Structures de données récursives

Utilisé lorsque la structure de données à traiter est *récursive*, par exemple, un arbre.

Il s'agit d'une forme de parallélisme récursif, mais **guidée par la structure de données**.

Programme Ruby 8.1 Utilisation du parallélisme récursif pour parcourir un arbre binaire et calculer la somme des valeurs des feuilles.

```
class Arbre
end

class Feuille < Arbre
  ...

  def somme
    @valeur
  end
end

class Noeud < Arbre
  attr_reader :gauche, :droite

  ...

  def somme
    fg = PRuby.future{gauche.somme}
    fd = droite.somme

    fg.value + fd
  end
end
```

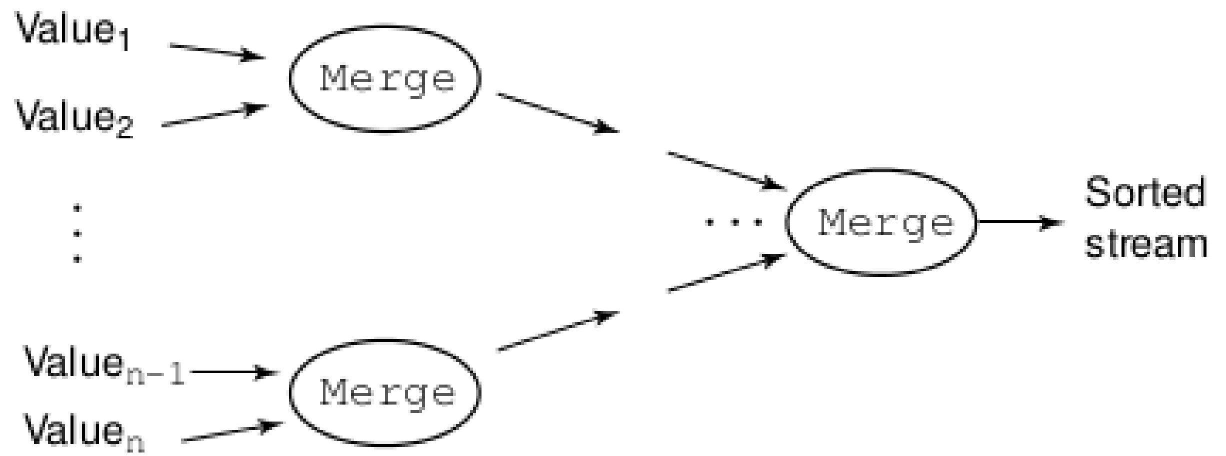
Voici un petit exemple d'arbre avec deux noeuds internes et trois feuilles crées et manipulés avec les méthodes du Programme Ruby **??** :

```
a = Noeud.new(
      Noeud.new( Feuille.new(10),
                 Feuille.new(30) ),
      Feuille.new(40)
    )

puts a.inspect # Formate pour mieux illustrer la structure!
=>

#<Noeud:0x48b67364
  @droite=#<Feuille:0x189cbd7c @valeur=40>,
  @gauche=#<Noeud:0x7bf3a5d8
            @droite=#<Feuille:0x42e25b0b @valeur=30>,
            @gauche=#<Feuille:0x39b43d60 @valeur=10>
           >
 >

puts a.somme
=>
80
```

8.5.3 Parallélisme de flux

Certains auteurs parlent plutôt de parallélisme de type «**producteur–consommateur**».

En général, les liens sont **unidirectionnels**, mais pas nécessairement sous forme d'un pipeline linéaire!



A sorting network of **Merge** processes.

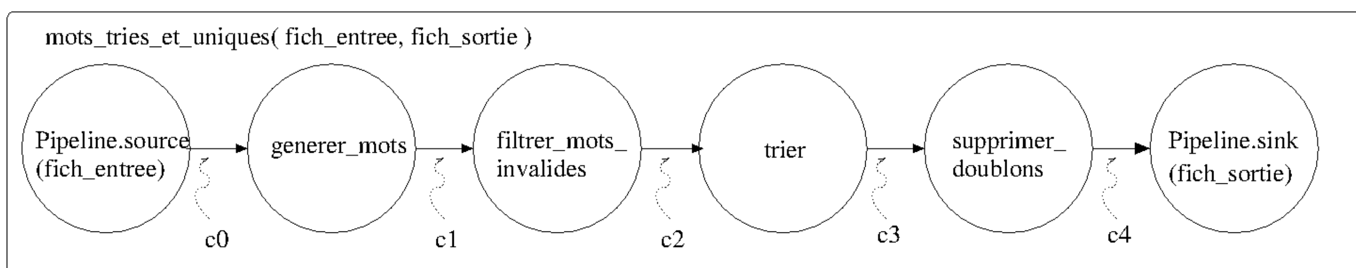
Copyright © 2000 by Addison Wesley Longman, Inc.

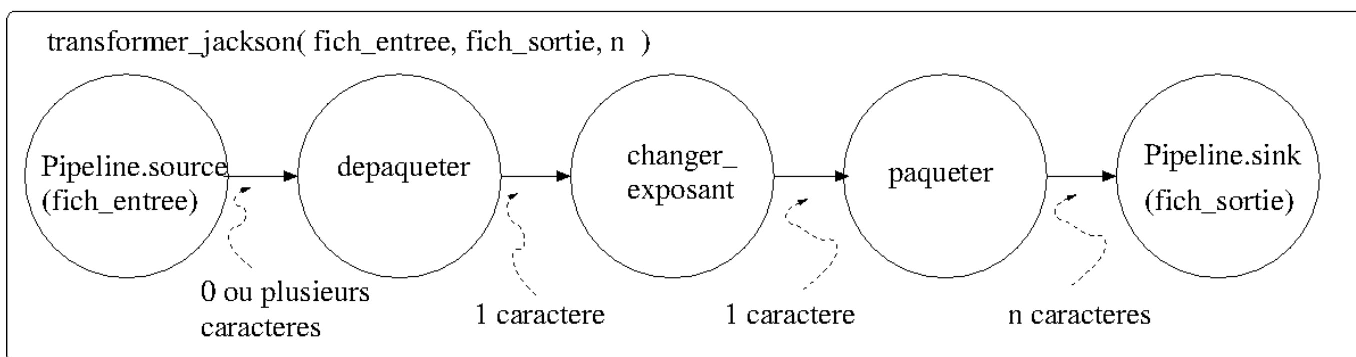
Figure 8.9: Un réseau de tri avec un ensemble de processus de type producteur et consommateur.

D'autres auteurs présentent plutôt ce patron sous le nom de **parallélisme de spécialistes** :

• on identifie diverses tâches **spécialisées** requises pour effectuer le travail global.

• on associe à chaque travailleur **une** de ces tâches.





Donc, le nombre de travailleurs est limité = profondeur du pipeline!

8.5.4 Arbre de décision pour choisir le patron le plus approprié

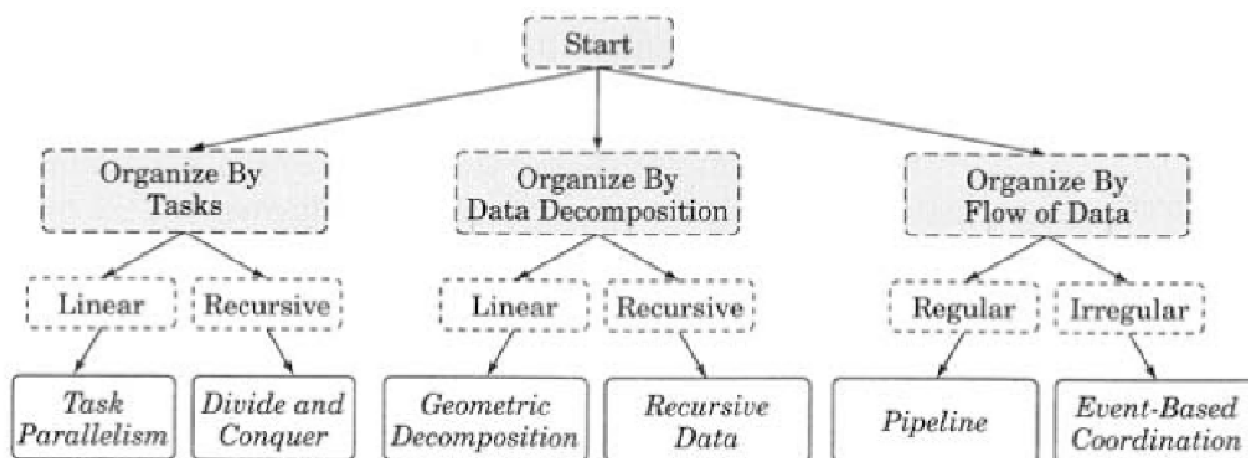


Figure 8.10: Arbre de décision pour choisir le patron le plus approprié.