Table des matières

Chapitre 8

Méthodologie pour la programmation parallèle et patrons d'algorithmes

8.1 Introduction

Ce chapitre présente tout d'abord une «méthodologie» — en fait, plutôt une «heuristique» — pour la programmation parallèle en mémoire partagée — donc avec *threads* — qui s'inspire de l'approche PCAM présentée par I. Foster [Fos95] :
`http://wotug.org/parallel/books/addison-wesley/dbpp/`

Ce chapitre introduit ensuite les principaux patrons d'algorithmes parallèles présentés par Sottile, Mattson, et Rasmussen [SMR09].

8.2 Approche PCAM

L'approche **PCAM** de Foster a été conçue plus particulièrement pour un modèle de programmation parallèle *par échanges de messages* — donc en mémoire distribuée, avec des *processus* plutôt que des *threads*. Toutefois, oOn peut quand même s'en inspirer pour la programmation parallèle avec *threads* communiquant par *variables partagées*.

8.2.1 Objectifs

Les objectifs (parfois contradictoires) qui sont visés par l'approche PCAM sont les suivants :

- Réduire les coûts de synchronisation et communication entre les *threads*.
- Distribuer le travail de façon la plus uniforme possible entre les *threads* — on dit aussi «équilibrer la charge» entre les *threads* (et processeurs) (*load balancing*).
- Conserver une flexibilité quant à la capacité de *dimensionnement* du programme (*scalability*) — donc, si on ajoute des processeurs, alors on devrait pouvoir augmenter l'accélération.

8.2.2 Étapes de l'approche PCAM

Les principales étapes de l'approche PCAM sont les suivantes :

1. *P*artitionnement : On identifie *tout le parallélisme disponible*. À cette étape, on ne soucie pas de la granularité. En fait, on vise même à obtenir des tâches **de très fine granularité** — donc aussi parallèle que possible.
2. *C*ommunication : On identifie les liens et dépendances entre les diverses tâches.
3. *A*gglomération : On procède au regroupement des tâches de fine granularité pour obtenir des tâches de plus forte granularité.
4. *M*apping : On associe les diverses tâches résultantes 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** (granularité aussi fine que nécessaire en fonction du problème), sans tenir compte des ressources ou contraintes de la machine — en d'autres mots, on suppose une machine «idéale» avec des ressources illimitées.

   Stratégies typiques = parallélisme de données, de résultat, diviser-pour-régner récursif, etc. Voir section 8.5

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

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

3. Si nécessaire, on **agglomère** (sur la base des dépendances) un groupe de tâches de fine granularité, de façon à produire des tâches de plus grosse granularité, ce qui aura aussi pour effet de réduire le nombre de tâches. Par exemple : regroupement par blocs d'éléments adjacents, par groupes d'éléments distribués cycliquement, d'un vecteur ou d'une matrice, etc.

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

   (a) Association via parallélisme récursif :

   Si l'algorithme sous-jacent est naturellement récursif, on peut alors utiliser du *parallélisme récursif*, où chaque instance récursive traite une sous-tâche.

   Pour limiter le nombre de *threads* et augmenter la granularité, on peut utiliser un *seuil de récursion*, qui indique quand passer d'une solution parallèle récursive à une solution séquentielle (récursive ou itérative). Ce seuil peut être fondé sur la taille du sous-problème à traiter ou sur le niveau de la récursion — après avoir atteint une certaine profondeur dans l'arbre de récursion parallèle, on cesse de créer des *threads*.

   (b) Association statique :

   Si toutes les tâches sont connues de façon statique — c'est-à-dire que le fait d'i.e., exécuter une tâche ne génère pas de nouvelles tâches — et si toutes les tâches requièrent sensiblement la même quantité de travail (sensiblement le même temps pour les traiter), alors on peut utiliser une association *statique* tâche/*thread*, i.e., **on lance un *thread* pour chaque tâche.**

(c) Association dynamique :

Si *i*) le travail requis pour les différentes tâches varie grandement d'une tâche à une autre ou si *ii*) le nombre de tâches varie dynamiquement (exécuter une tâche peut en créer de nouvelles), alors on peut utiliser une association *dynamique* tâche/*thread* : on crée un nombre fixe de *threads*, qui vont exécuter une tâche à la fois en fonction de leur disponibilité — en utilisant l'option «`dynamic: k`» ou un «sac de tâches»pour représenter les tâches à traiter.

Dans un tel cas, il est **important** que le nombre de tâches soit **(nettement) plus grand** (grosso modo, au moins *un ordre* de grandeur plus grand) que le nombre de *threads*... sinon la distribution du travail entre les *threads* pourrait ne pas être uniforme.

Le pseudocode 8.1 présente, dans un pseudocode informel, une heuristique pour la conception d'un programme parallèle avec *threads*.

```
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*.

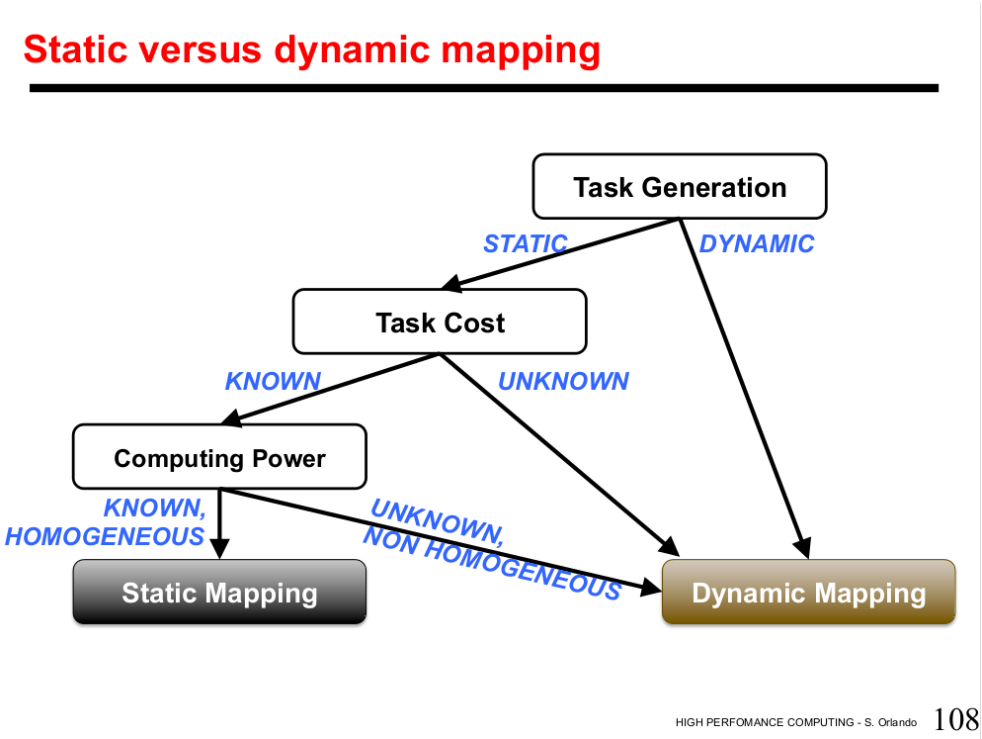


Figure 8.1: Arbre de décision pour choisir entre une stratégie statique vs. dynamique d'association (*mapping*) des tâches aux *threads*.
Source : `http://www.dais.unive.it/~calpar/`

La Figure 8.1 présente un «d'arbre de décision» pour déterminer si une approche **statique** d'association des tâches aux *threads* est appropriée ou si une approche **dynamique** est préférable... ou même nécessaire (première branche droite).

Une approche **statique** est généralement plus efficace parce qu'elle génère moins de surcoûts de communication et de synchronisation. Toutefois, une approche statique n'est appropriée que si les conditions suivantes sont satisfaites :

1. Toutes les tâches peuvent être identifiées de façon statique, c'est-à-dire, aucune nouvelle tâche n'est générée en cours d'exécution.
2. Le coût de chacune des tâches est connu.
3. Les coûts des diverses tâches sont sensiblement uniformes **et** les unités de traitement possèdent toutes la même puissance de traitement.

Les deux dernières conditions font en sorte qu'on pourra être (relativement) assuré que les unités de traitement termineront (presque) toutes en même temps, donc sans qu'il n'y ait d'unité qui attend à ne rien faire que les autres unités aient terminé.

8.4 Effet de la granularité sur les performances

Dans cette approche de programmation parallèle, un point important à garder à l'esprit est le suivant. À l'étape d'*Agglomération*, on regroupe des tâches *de granularité (possiblement) «très fine»* — tâches obtenues à l'étape de *Partitionnement* — pour générer des tâches de granularité «plus grossière». On fait de tels regroupements pour réduire les coûts associés à la gestion d'un trop grand nombre de petites tâches.

Par contre, il est important aussi de tenir compte que si on réduit le nombre de tâches de façon trop importante et qu'on obtient *un très (trop!?) petit nombre de tâches de forte granularité*, *alors les performances peuvent se dégrader* ☹

La figure 8.2 illustre l'effet typique de la granularité des tâches d'un programme parallèle sur le temps d'exécution, alors que la figure 8.3 illustre l'effet sur les *performances* (e.g., accélération) de ce programme — un temps d'exécution plus petit implique de meilleures performances!

Un tel comportement des performances (ou même d'autres facteurs) avec courbe en U inversé *est fréquent*, et ce pas uniquement pour la taille des grains. Par exemple, pour plusieurs problèmes, lorsqu'on garde la taille du problème fixe, au début, lorsqu'on augmente le nombre de processeurs, le temps d'exécution peut diminuer. Toutefois, si on continue d'augmenter le nombre de processeurs, le temps d'exécution va se stabiliser, puis commencer à augmenter lorsque le nombre de processeurs devient trop grand ☹

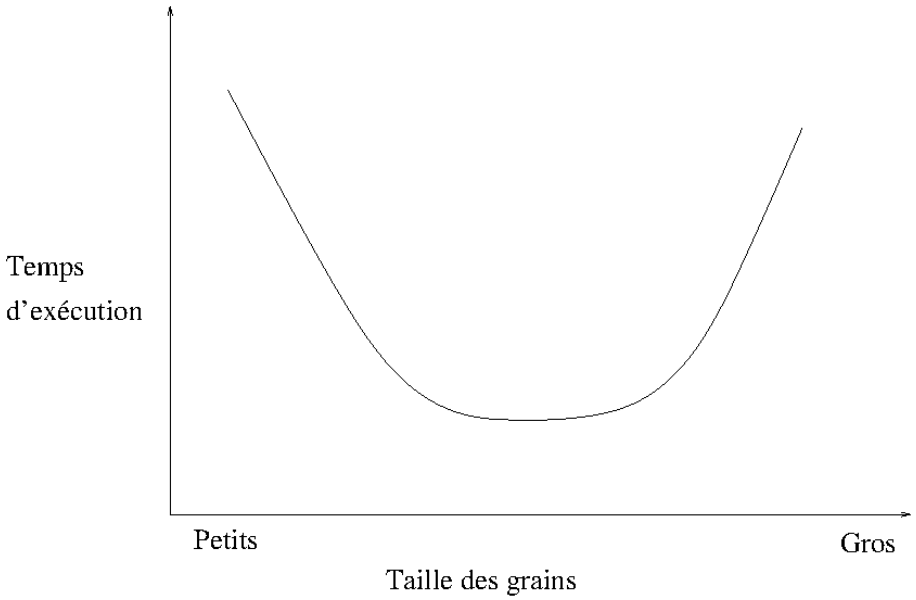


Figure 8.2: Courbe illustrant l'effet général de la taille des grains (des tâches) sur le temps d'exécution.

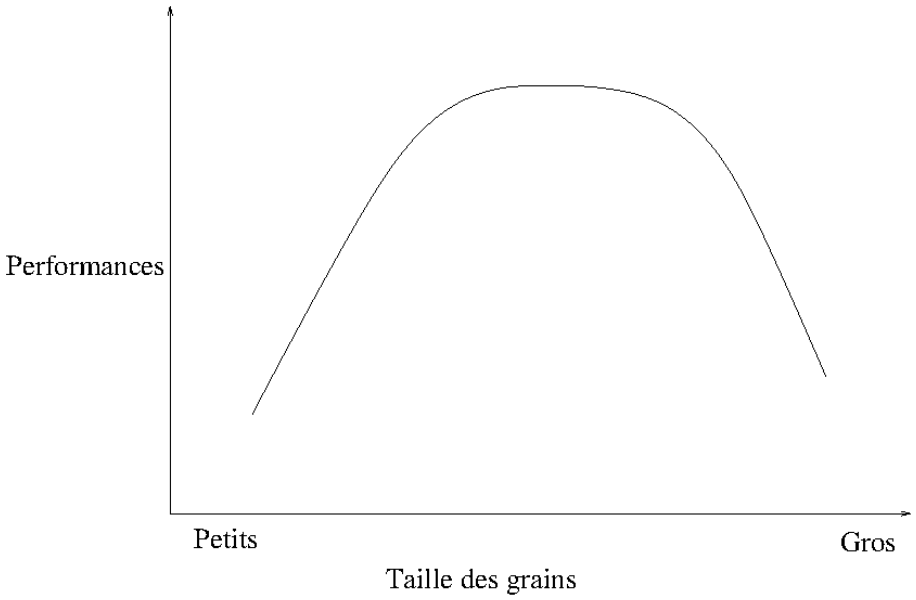


Figure 8.3: Courbe illustrant l'effet général de la taille des grains (des tâches) sur la performance.

8.5 Patrons d'algorithmes parallèles

Divers auteurs ont présenté des «patrons» pour développer des algorithmes parallèles [Lea00, SSRB00, MSM05, SMR09].

Dans ce chapitre, nous allons faire un survol des patrons présentés dans «*Introduction to Concurrency in Programming Languages*» [SMR09]. Il s'agit ici de «**patrons** de conception», donc de **stratégies de conception d'algorithmes** — et non pas des constructions de programmation parallèle : un patron **d'algorithme** peut souvent être mis en oeuvre par plusieurs patrons **de programmation**. Ces stratégies nous permettent, lors de l'étape Pde la méthode PCAM, d'identifier les tâches qui peuvent s'exécuter en parallèle.

Un algorithme parallèle implique nécessairement le fait d'exécuter plusieurs calculs en même temps. On peut classer les algorithmes parallèles en trois (3) grandes catégories, selon la façon dont les calculs à faire en parallèle sont identifiés :

8.5.1 **Parallélisme de tâches** (*task parallelism*) : on dispose d'un *grand nombre de tâches* pouvant s'exécuter en parallèle.

8.5.2 **Parallélisme de données** (*data parallelism*) : on dispose d'une *grande quantité de données* sur lesquelles on applique la même série d'opérations sur plusieurs éléments de données.

8.5.3 **Parallélisme de flux** (*flow parallelism* ou *stream parallelism*) : on dispose d'un *flux de données* qu'on doit traiter, avec un traitement décomposé en plusieurs *étapes* — comme une chaîne de montage.

8.5.1 Parallélisme de tâches

Dans certains systèmes complexes, l'architecture du système est *naturellement* composée d'un grand nombre de composants relativement indépendants qui interagissent entre eux. Dans un tel cas, chaque composant peut être associé à une tâche indépendante s'exécutant de façon concurrente et parallèle.

La figure 8.4 illustre l'architecture d'un système pour la modélisation du *climat*.[1] La parallélisation de ce système pourrait se faire en associant une ou plusieurs tâches à chacune des composantes.

[1] À ne pas confondre avec la météo. Alors que la modélisation de la météo est un processus à court terme (quelques jours), la modélisation du climat est un processus *à long terme* (années, décennies, etc.), qui implique un nombre de facteurs beaucoup plus grand que la météo.

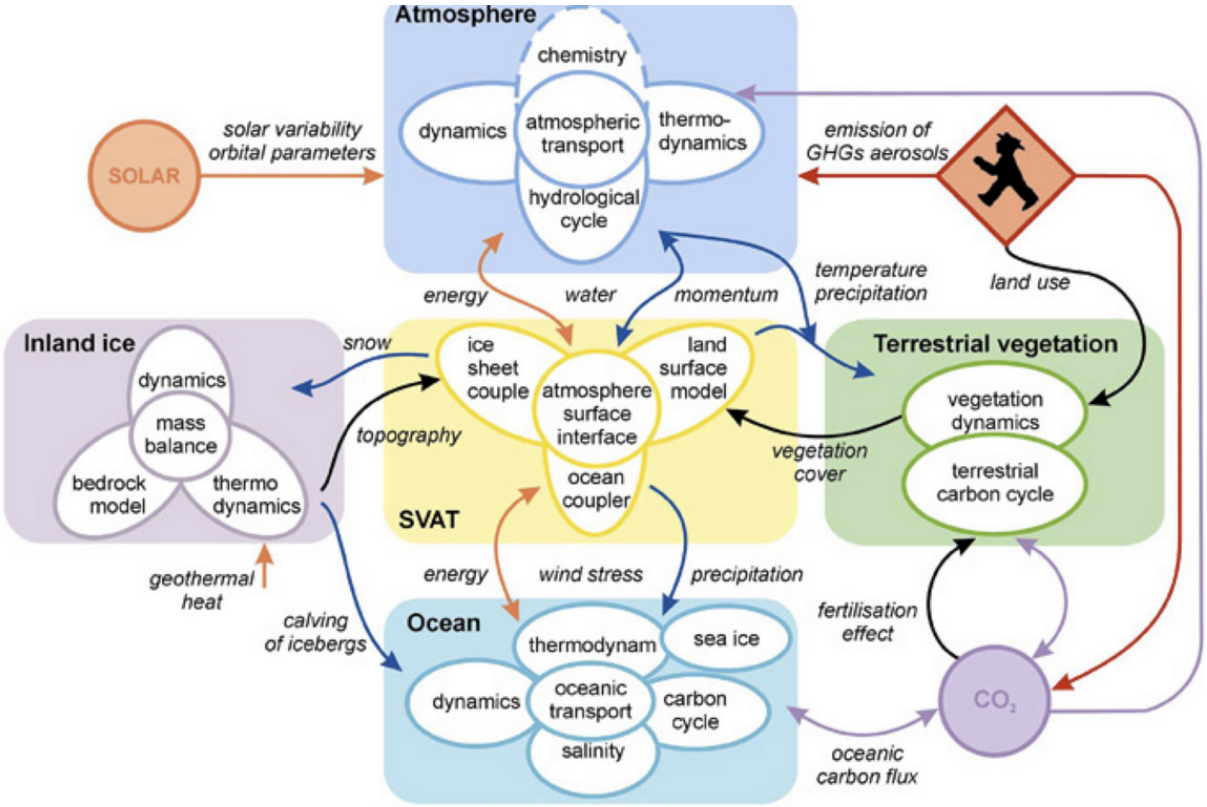


Figure 8.4: Architecture d'un système pour la modélisation du climat (source : `http://www.cs.toronto.edu/~sme/PMU199-climate-computing/pmu199-2012F/coupled-architecture.jpg`).

Parallélisme «embarrassant»

Certains problèmes peuvent facilement être décomposés en un grand nombre de parties indépendantes, donc où chaque partie peut s'exécuter en parallèle, *sans aucune* contrainte ou *dépendance*. On parle alors de parallélisme «**embarrassant**» (*embarassingly parallel*).

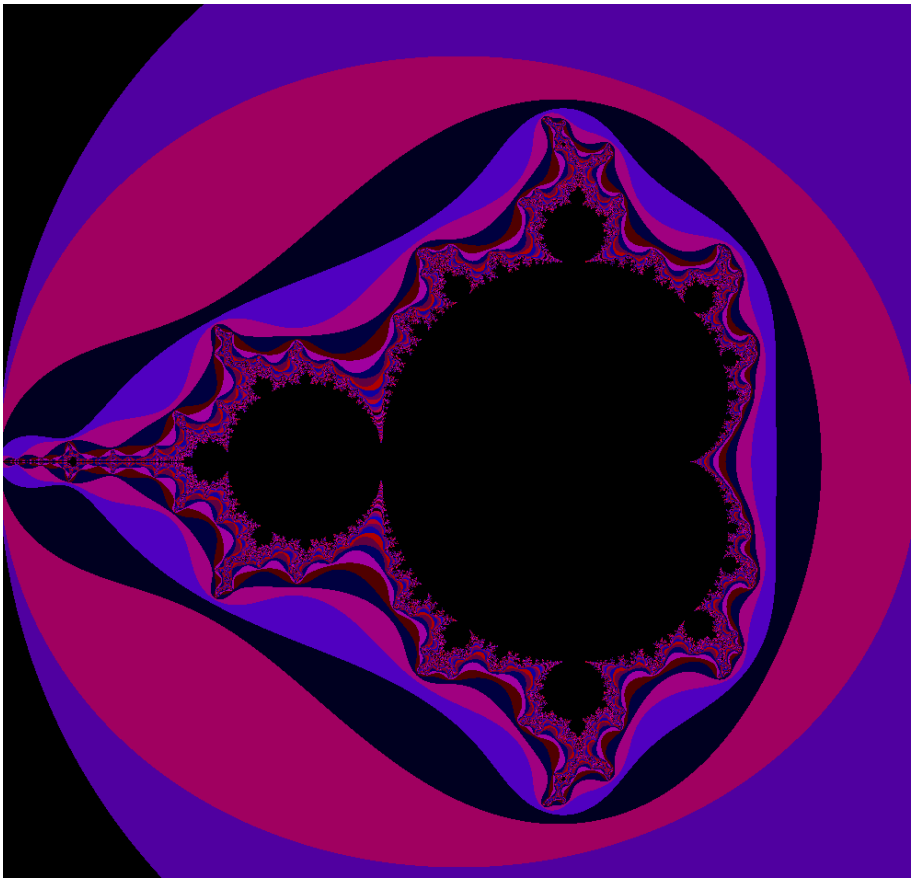


Figure 8.5: Représentation graphique de l'ensemble de Mandelbrot.

La figure 8.5 présente une représentation graphique de l'ensemble de Mandelbrot, image contenant 1 Mega pixels (1000×1000). La couleur associée à chaque pixel *dépend uniquement du coordonnée du pixel*. Ceci implique que **tous les pixels peuvent être calculés en parallèle**.

Lorsqu'on présente des algorithmes ou programmes parallèles, on utilise souvent des *graphes de dépendances de tâches* pour illustrer les diverses tâches à exécuter et les dépendances — de données ou de contrôle — qui existent entre ces diverses tâches. Plus le nombre de dépendances est élevé, plus les possibilités d'exécution parallèle sont restreintes.

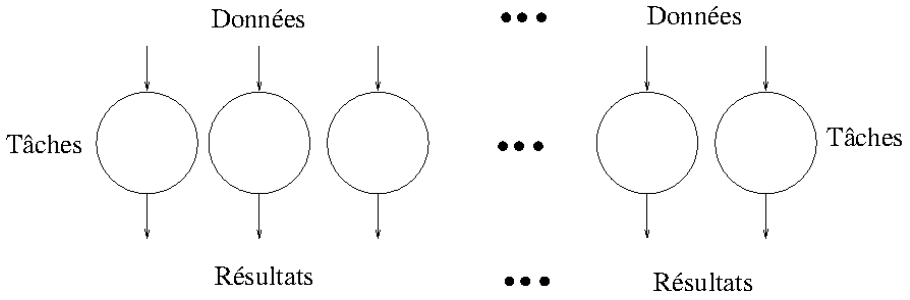


Figure 8.6: Graphe de dépendances de tâches pour un problème avec parallélisme embarrassant.

Dans un problème avec parallélisme embarrassant, le graphe de tâches possède typiquement l'allure de la figure 8.6. Dans le cas du calcul de l'ensemble de Mandelbrot, en théorie, chaque point pourrait être associé à une tâche distincte traitée par un *thread* indépendant. Toutefois, ce ne serait pas nécessairement une solution idéale :

- Parce que créer et gérer une tâche parallèle et un *thread* peut être relativement coûteux. Pour la figure 8.5, utiliser une tâche parallèle et un *thread* pour chaque pixel impliquerait d'avoir *un million de* threads*!*

- Lorsqu'on a un grand nombre de tâches indépendantes qui s'exécutent vraiment en parallèle, le temps total d'exécution est alors le temps requis pour l'exécution *de la plus grosse tâche*.

  Par exemple, supposons qu'on ait un programme décomposé en 1000 tâches traités par 1000 *threads* qui s'exécutent sur 1000 processeurs. Supposons que 999 de ces tâches s'exécutent en 1 ms alors qu'une autre tâche, plus complexe, s'exécute en 100 ms. Dans ce cas, le temps total d'exécution serait de 100 ms... et pendant 99 ms, on aurait 999 processeurs qui seraient inutilisés, en attente que le dernier *thread*, donc le programme, se termine.

Dans un problème avec parallélisme embarrassant — et, en fait, pour de nombreux autres problèmes — il est important soit que les tâches soient de même taille, soit que l'on utilise un mécanisme qui permette de sélectionner de façon dynamique les tâches à exécuter, pour éviter que des processeurs soient sous-utilisés. C'est ce qu'on appelle le problème de *répartition de la charge* et de *répartition des tâches*. Ces questions ont été abordées dans les sections qui précèdent — étape A (*agglomération*) et étape M (*mapping*).

Parallélisme «semi-embarrassant»

Un problème avec parallélisme «**semi-embarrassant**», comme pour le cas précédent, peut être décomposé en un grand nombre de tâches qui sont indépendantes, *mais pas tout à fait complètement* : les tâches interagissent, mais de façon limitée, et *souvent uniquement vers la fin de l'exécution des tâches.*

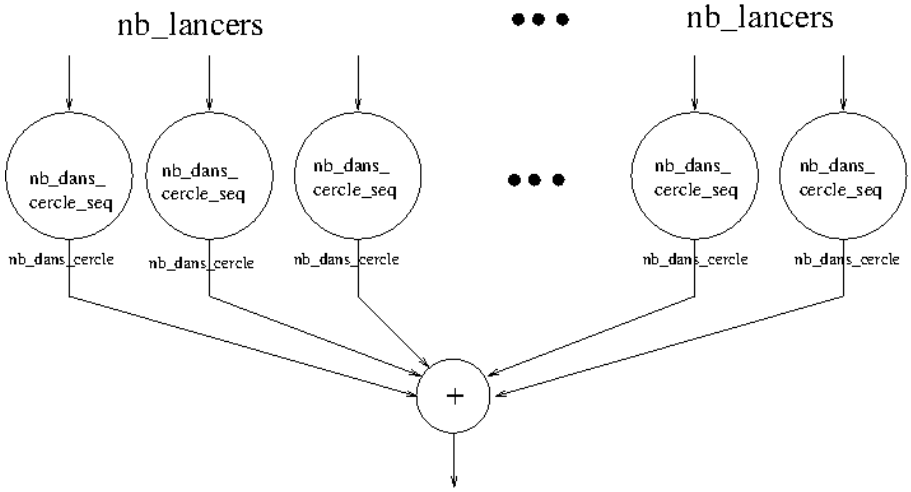


Figure 8.7: Graphe de dépendances de tâches pour un problème avec parallélisme semi-embarrassant — approximation de π par une méthode Monte Carlo.

Un problème relativement simple ayant cette propriéte est celui visant à estimer la valeur de π à l'aide d'une méthode de Monte Carlo : voir section **??** (p. **??**). Le graphe de dépendances des tâches ressemble alors à celui de la figure 8.7 : le gros du travail — la simulation des lancers — se fait de façon complètement indépendante par chacun des *threads*, et ce n'est qu'à la toute fin qu'il suffit d'additionner les résultats calculés par chacun d'entre eux!

Parallélisme récursif

De nombreux problèmes peuvent être résolus par un algorithme récursif, et ce en utilisant l'approche «diviser-pour-régner» — voir sections **??** et **??** pour plus de détails. L'approche diviser-pour-régner peut être décrite comme suit :

```
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 parle d'une approche «diviser-pour-régner **dichotomique**» lorsqu'on décompose le problème en deux (2) sous-problèmes.L'algorithme récursif a alors l'allure du Pseudocode 8.2.

```
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 (décomposition en deux (2) sous-problèmes).

Un algorithme récursif s'obtient alors facilement lorsque les deux sous-problèmes sont **disjoints** — donc indépendants : il suffit d'exécuter *en parallèle* les deux appels à `resoudre_dpr_2`! Dans un tel cas, on obtient alors un graphe de dépendances tel que celui présenté à la Figure 8.8 — beaucoup de parallélisme donc, en fait, *souvent trop!*

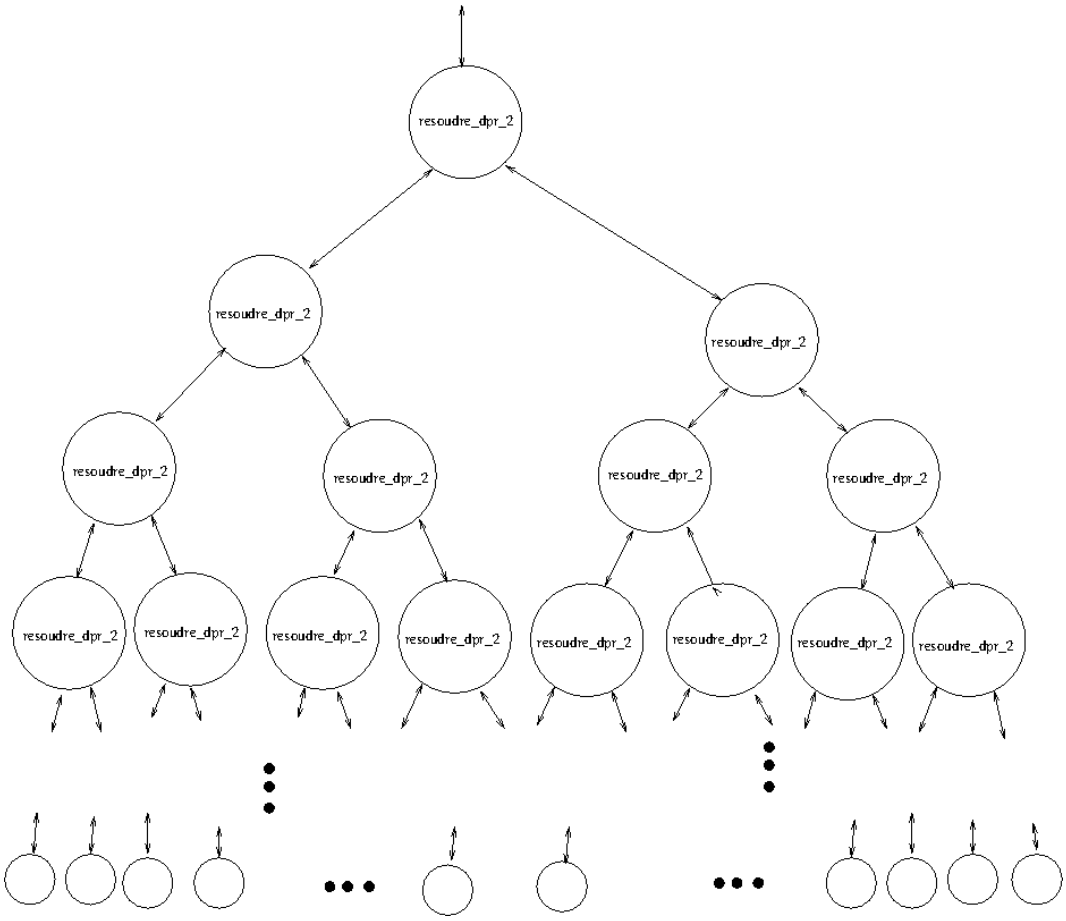


Figure 8.8: Graphe de dépendances de tâches pour un problème avec parallélisme récursif dichotomique — appels récursifs à `resoudre_dpr_2`.

Patron de conception d'algorithme vs. patron de programmation

Bien qu'il existe souvent une **correspondance** assez directe entre un patron de **conception** et un patron de **programmation**, **ceci ne signifie pas que ce patron de programmation soit nécessairement** la meilleure solution... mais plusieurs patrons de programmation peuvent aussi être plus utilisés

Exemple : Un problème pourrait avoir une solution naturellement récursive, donc suggérant un patron d'algorithme «diviser-pour-régner» avec récursion. Par contre, une mise en oeuvre avec du parallélisme de style *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, comme on l'a vu dans un exercice fait précédememnt, il est tout à fait possible de mettre en oeuvre une stratégie diviser-pour-régner... **sans utiliser de récursion** — par exemple, avec un *pool de threads* et un sac de tâches.

8.5.2 Parallélisme de données

Lin et Snyder [LS09] 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***.

Dans les premières machines parallèles de type SIMD — *Single Instruction, Multiple Data* — une telle forme de parallélisme était le fondement même des calculs parallèles — en fait, la seule forme exploitable de parallélisme. Toutefois, une particularité de ces machines, qu'on ne cherche pas à reproduire lorsqu'on développe des algorithmes avec parallélisme de données, est le fait que dans une machine SIMD, c'est *exactement la même* **instruction** qui s'exécute sur tous les processeurs en même temps, de façon synchrone.

Dans les machines modernes où le parallélisme de données est aussi le fondement des calculs — par exemple les GPU — le modèle SIMD a été assoupli et chaque unité d'exécution va exécuter le même segment de code — souvent appelé *kernel* — donc la même procédure. On se rapproche donc d'un modèle SPMD — *Single Program,*

Multiple Data — utilisé dans plusieurs langages de programmation modernes, par exemple, MPI [Pac97, Qui03], que nous ne traiterons pas dans ce cours

C'est donc plus ce dernier modèle, plus souple, que l'on va tenter d'exploiter lorsqu'on développe des algorithmes parallèles fondés sur le parallélisme de données.

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

Nous avons déjàéjà vu, à la section **??**, l'approche de parallélisme de données fondée sur les *applications* et les *réductions* parallèles — le modèle avec les opérations *Map* et *Reduce*.

Dans ce modèle, 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, i.e., on part d'une collection (de données) pour produire une collection (résultat) :

```
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

Une stratégie intéressante et souvent utile dans le contexte du parallélisme de donnée, stratégie proposée par Carriero & Gelernter [CG89], est celle du **parallélisme de résultat**. Dans cette approche, on identifie le parallélisme *en partant du produit final*, i.e., en partant du résultat désiré et en tentant de le décomposer en morceaux indépendants. On attribue ensuite à chaque travailleur la tâche de produire un **morceau du résultat** final.

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.*

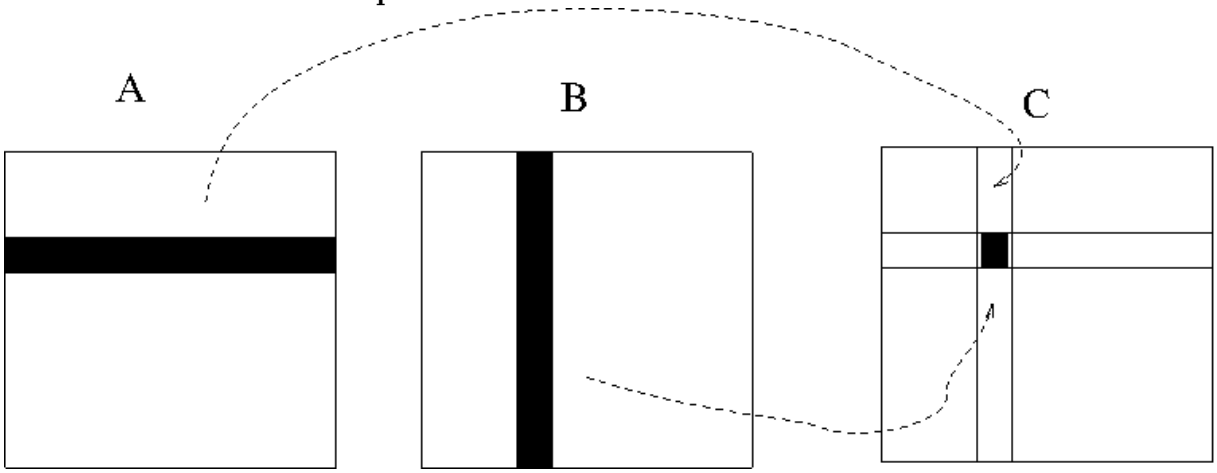
Quelques exemples qui utilisent du parallélisme de données ou de résultat :

1. Programme Ruby **??** : Fait la somme de deux tableaux. Chaque position du tableau à calculer représente une tâche indépendante.
2. Programme PRuby **??** : Fait le produit de deux matrices. Le calcul de chaque élément de la matrice résultat (produit scalaire ligne/colonne) représente une tâche indépendante.

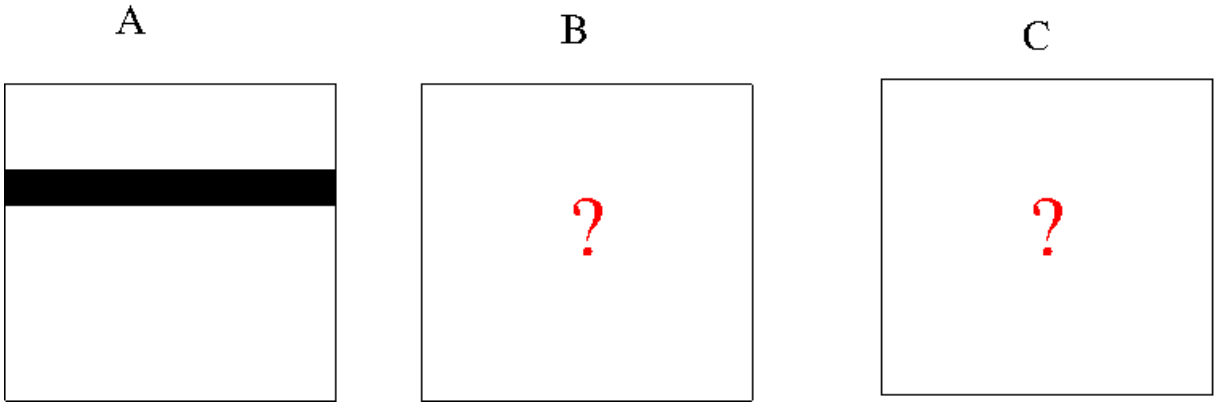
Rappel :

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

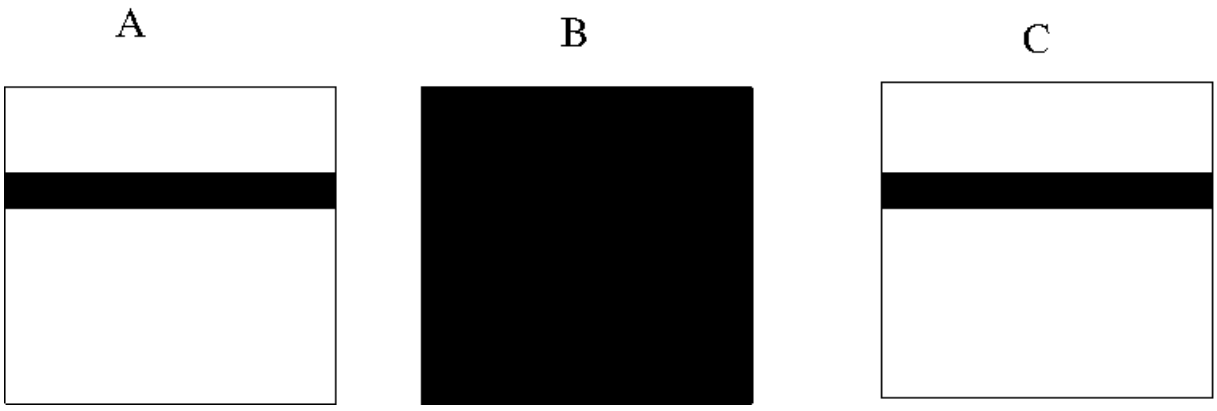
Illustration des dépendances de calcul :



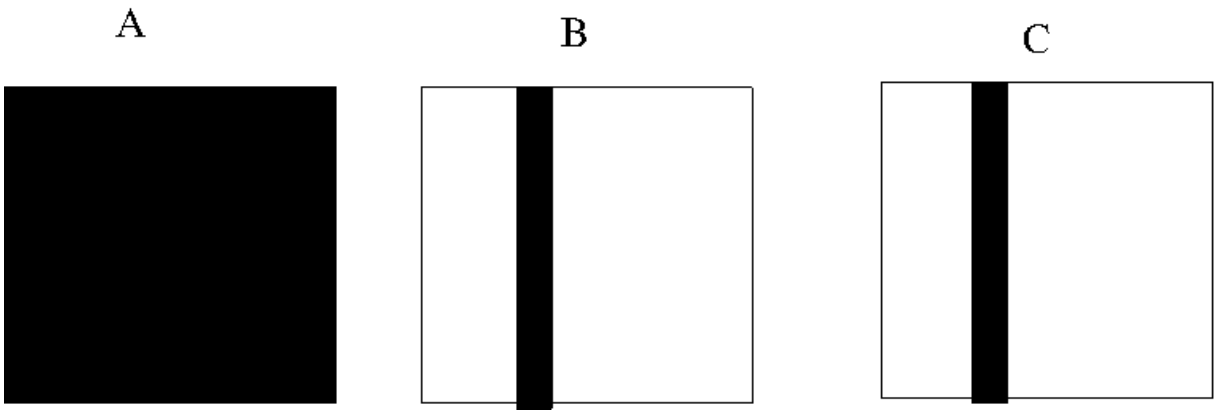
Parallélisme de données avec une ligne de A par *thread*


A
B
C
?
?

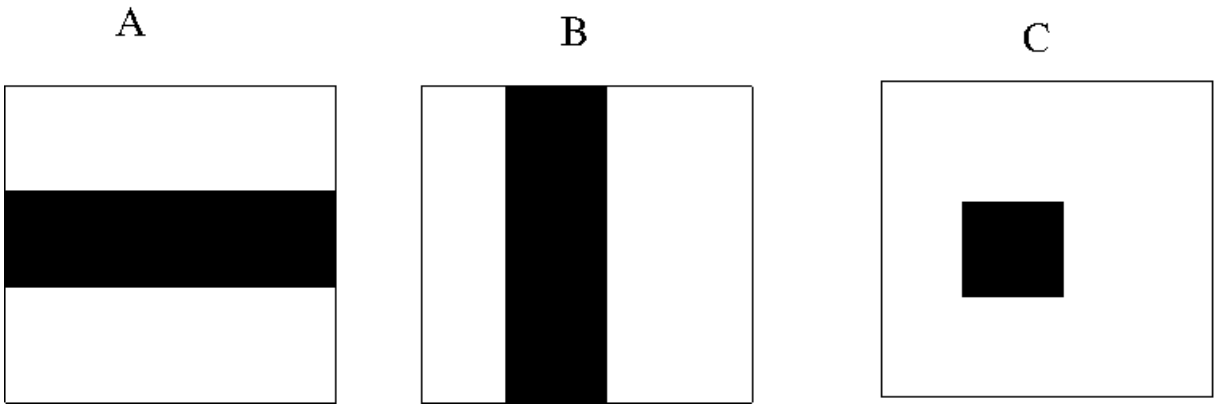
Parallélisme de données avec une ligne de A par *thread*



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



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



Évidemment, dans chacun de ces cas, bien qu'à l'étape de partitionnement on puisse obtenir *une tâche par item du résultat*, on peut ensuite produire un programme plus performant en regroupant (agglomérant) ces tâches de granularité fine :

- Programme Ruby **??** : Utilisation d'un `pmap` avec «`static:`» (implicite).
- Programme Ruby **??** : Utilisation d'un `pmap` avec «`dynamic:`» explicite ou d'un `TaskBag`.
- Programme PRuby **??** : Utilisation d'un `peach` avec «`static:`» implicite pour les lignes, et d'un `each` pour les colonnes — donc agglomération par blocs de lignes adjacents.

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.

Décomposition géométrique

La motivation de ce patron est la suivante, patron aussi appelé *décomposition du domaine* :

> *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 — Chapitre **??**, sur la diffusion de la chaleur dans un cylindre.

Structures de données récursives

Ce patron est utilisé lorsque la structure de données à traiter est *récursive*, par exemple, un arbre. Il s'agit d'une forme de parallélisme récursif, mais où la récursion est **guidée** (déterminée) **par la structure de données** elle-même.

Par exemple, soit un arbre binaire complet pour lequel on veut faire la somme des champs `valeur`, arbre ayant la structure suivante :

- Une `Feuille` comporte un champ `valeur` ;
- Un `Noeud` a toujours deux enfants, `gauche` et `droite`, mais n'a pas de champ `valeur`.

On obtient alors facilement un algorithme parallèle récursif tel qu'illustré dans le Programme Ruby 8.1.

Voici un petit exemple d'arbre avec deux noeuds internes et trois feuilles crées et manipulés avec les méthodes du Programme Ruby 8.1 :

```
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
```

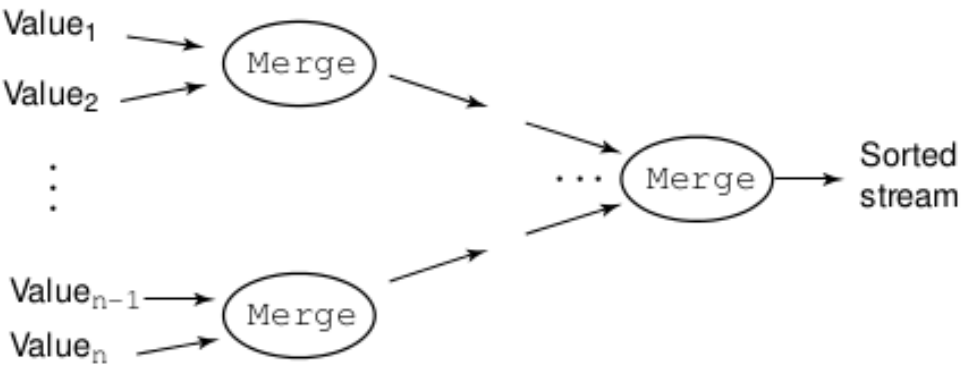
Programme Ruby 8.1 Utilisation du parallélisme récursif pour parcourir un arbre binaire et calculer la somme des valeurs des feuilles.

```
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
```

8.5.3 Parallélisme de flux

Le parallélise de flux a été présenté à la section **??**, donc nous ne le présenterons pas en détail à nouveau.

Soulignons toutefois que certains auteurs parlent aussi dans ce cas de parallélisme de type «*producteur–consommateur*» [And00] : le programme est composé d'un ensemble de processus qui communiquent entre eux de façon uni-directionnelle, les processus étant généralement organisés sous forme d'un pipeline — linéaire ou plus complexe, e.g., un arbre ou un graphe dirigé acyclique (DAG), un réseau de tri par exemple : voir figure 8.9.



A sorting network of **Merge** processes.

Figure 8.9: Un réseau de tri avec un ensemble de processus de type producteur et consommateur (source : [And00]).

D'autres auteurs présentent aussi ce patron sous le nom de **parallélisme de spécialistes** [CG89, CG90] : on identifie diverses tâches **spécialisées** requises pour effectuer le travail global. Ensuite, on associe à chaque travailleur **une** de ces tâches spécifiques, en tentant le plus possible de les faire traiter en parallèle. Les divers travailleurs exécutent donc des tâches tout à fait *distinctes* les unes des autres — pas seulement en termes des données manipulées ou des attributs de la tâche, comme dans un sac de tâches, mais même en termes *du type de tâche effectué*. Un exemple :

les processus `depaqueter`, `changer_exposant` et `paqueter` dans le problème de Jackson **??**, qui font toutes des choses très différentes.

8.5.4 Arbre de décision pour choisir le patron le plus approprié

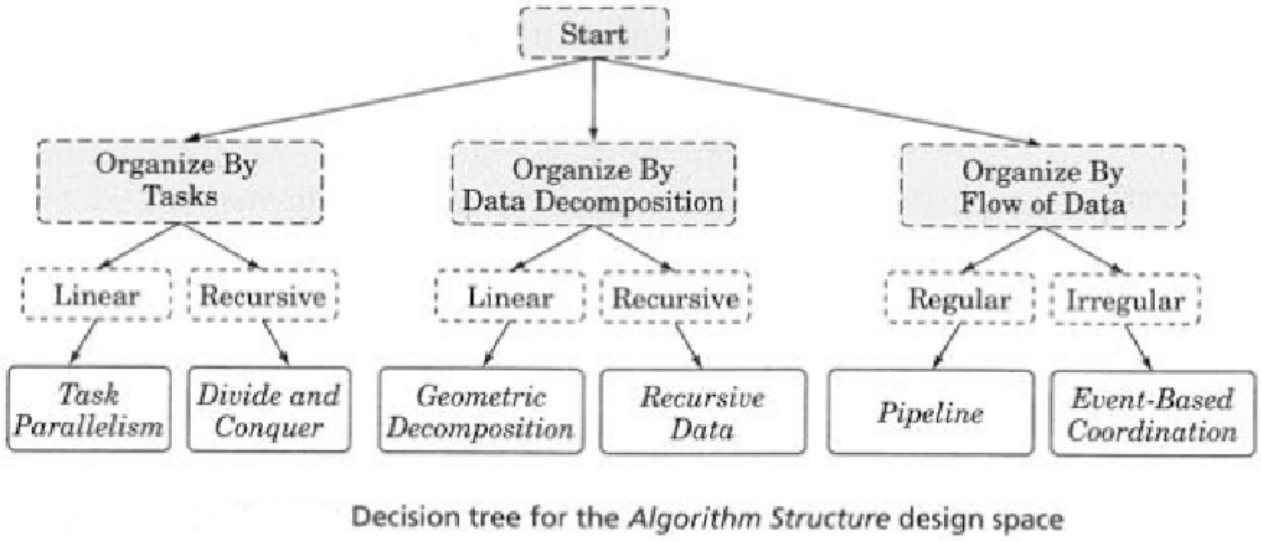


Figure 8.10: Arbre de décision pour choisir le patron le plus approprié.

La figure 8.10 (tirée de [SMR09]) présente un «arbre de décision» pour déterminer le patron d'algorithmes les plus approprié pour un problème. Encore une fois, il s'agit d'une *heuristique*, pas d'un *algorithme* exact et précis.

Références

[And00] G.R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, Reading, MA, 2000. [QA76.58A57 2000].

[CG89] N. Carriero and D. Gelernter. How to write parallel programs—a guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, Sept. 1989. [Tiré de [ST95]].

[CG90] N. Carriero and D. Gelernter. *How to Write Parallel Programs—A* First *Course*. MIT Press, 1990.

[Fos95] I. Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995. `http://www-unix.mcs.anl.gov/dbpp`.

[Lea00] D. Lea. *Concurrent Programming in Java—Design Principles and Patterns (Second Edition)*. Addison-Wesley, 2000.

[LS09] C. Lin and L. Snyder. *Principles of Parallel Programming*. Addison Wesley, 2009.

[MSM05] T.G. Mattson, B.A. Sanders, and B.L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 2005.

[Pac97] P.S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufman Publ., 1997.

[Qui03] M.J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2003.

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

[SSRB00] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-oriented software architecture Vol. 2—Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Ltd., 2000.

[ST95] D.B. Skillicorn and D. Talia. *Programming Languages for Parallel Processing*. IEEE Computer Society Press, 1995.