Table des matières

Chapitre 9

Exemples illustrant l'approche PCAM

9.1 Calcul de la distance d'édition entre deux chaines

9.1.1 Définition du problème

La **distance d'édition** est souvent utilisée pour définir une mesure de **similarité** entre chaînes (de caractères, de symboles, de gènes, etc.). Différentes distances sur diverses formes de chaînes ont ainsi été introduites, par ex., traitement et analyse de textes, analyse de protéines et de génomes en bio-informatique, et même, tout récemment, correction de dictées musicales.

L'idée maîtresse derrière la distance d'édition est de déterminer le nombre **minimum** d'opérations (par ex., insertion, suppression, ou substitution) qui doivent être appliquées sur la première chaîne pour obtenir la deuxième. Un exemple, présenté à la Figure 9.1, montre comment le mot "`surgery`" peut être transformé en "`survey`" à l'aide d'opérations de suppression, insertion ou substitution.

```
surgery
surery     -- Suppression de g
surey      -- Suppression de r
survey     -- Insertion de v

surgery
urgery     -- Suppression de s
rgery      -- Suppression de u
gery       -- Suppression de r
ery        -- Suppression de g
very       -- Insertion de v
vey        -- Suppression de r
svey       -- Insertion de s
suvey      -- Insertion de u
survey     -- Insertion de r

surgery
survery    -- Substitution de g par v
survey     -- Suppression de r
```

Figure 9.1: Trois façons différentes transformer de “`surgery`” en “`survey`”.

9.1.2 Une première solution purement récursive

Programme Ruby 9.1 Fonction récursive pour calculer la distance d'édition entre deux chaines avec un cout unitaire pour les opérations.

```
def cout_subst( c1, c2 )
  c1 == c2 ? 0 : 1
end

def distance( ch1, ch2 )
  # Cas de base
  return ch2.size if ch1.size == 0
  return ch1.size if ch2.size == 0

  # Cas recursifs
  avec_insertion =
    distance( ch1, ch2[0..-2] ) + 1

  avec_suppression =
    distance( ch1[0..-2], ch2 ) + 1

  avec_substitution =
    distance( ch1[0..-2], ch2[0..-2] ) +
                  cout_subst(ch1[-1], ch2[-1])

  [avec_insertion, avec_suppression, avec_substitution].min
end
```

La figure 9.2 présente l'arbre des appels récursifs pour un appel initial à la fonction avec «`distance( "ad", "axe" )`».

Que constate-t-on quant aux appels récursifs qui sont effectués lors du calcul de la distance d'édition entre `"ad"` et `"axe"`?

Exercice 9.1: Arbre des appels récursifs pour le calcul de distance d'édition.

```
distance( "ad", "axe" )
  distance( "ad", "ax" )
    distance( "ad", "a" )
      distance( "ad", "" )
      distance( "a", "a" )
        distance( "a", "" )
        distance( "", "a" )
        distance( "", "" )
      distance( "a", "" )
    distance( "a", "ax" )
      distance( "a", "a" )
        distance( "a", "" )
        distance( "", "a" )
        distance( "", "" )
      distance( "", "ax" )
      distance( "", "a" )
    distance( "a", "a" )
      distance( "a", "" )
      distance( "", "a" )
      distance( "", "" )
  distance( "a", "axe" )
    distance( "a", "ax" )
      distance( "a", "a" )
        distance( "a", "" )
        distance( "", "a" )
        distance( "", "" )
      distance( "", "ax" )
      distance( "", "a" )
    distance( "", "axe" )
    distance( "", "ax" )
  distance( "a", "ax" )
    distance( "a", "a" )
      distance( "a", "" )
      distance( "", "a" )
      distance( "", "" )
    distance( "", "ax" )
    distance( "", "a" )
```

Figure 9.2: Arbre des appels pour un appel initial à `distance("ad", "axe")`.

```
distance( "ad", "axe" )
  distance( "ad", "ax" )
    distance( "ad", "a" )
      distance( "ad", "" ) = 2
      distance( "a", "a" )
        distance( "a", "" ) = 1
        distance( "", "a" ) = 1
        distance( "", "" ) = 0
      distance( "a", "a" ) => 0  # =>  Cas de base
      distance( "a", "" ) = 1
    distance( "ad", "a" ) => 1
    distance( "a", "ax" )
      distance( "a", "a" )
        distance( "a", "" ) = 1
        distance( "", "a" ) = 1
        distance( "", "" ) = 0
      distance( "a", "a" ) => 0
      distance( "", "ax" ) = 2
      distance( "", "a" ) = 1
    distance( "a", "ax" ) => 1
    distance( "a", "a" )
      distance( "a", "" ) = 1
      distance( "", "a" ) = 1
      distance( "", "" ) = 0
    distance( "a", "a" ) => 0
  distance( "ad", "ax" ) => 1
  distance( "a", "axe" )
    distance( "a", "ax" )
      distance( "a", "a" )
        distance( "a", "" ) = 1
        distance( "", "a" ) = 1
        distance( "", "" ) = 0
      distance( "a", "a" ) => 0
      distance( "", "ax" ) = 2
      distance( "", "a" ) = 1
    distance( "a", "ax" ) => 1
    distance( "", "axe" ) = 3
    distance( "", "ax" ) = 2
  distance( "a", "axe" ) => 2
  distance( "a", "ax" )
    distance( "a", "a" )
      distance( "a", "" ) = 1
      distance( "", "a" ) = 1
      distance( "", "" ) = 0
    distance( "a", "a" ) => 0
    distance( "", "ax" ) = 2
    distance( "", "a" ) = 1
  distance( "a", "ax" ) => 1
distance( "ad", "axe" ) => 2
```

Figure 9.3: Arbre des appels pour un appel initial à `distance("ad", "axe")` avec indication des résultats retournés — cas de base notés par «=», cas récursifs notés par «=>».

9.1.3 Une formalisation du problème avec des équations de récurrence

Soit deux chaînes `ch1` et `ch2`. Dénotons par $D(i, j)$ le **coût minimal** pour transformer la chaîne `ch1[0...`i`]` en `ch2[0...`j`]` `(*)`, le coût étant défini par le nombre d'opérations pour passer de l'une à l'autre tel que décrit plus bas. Les équations de récurrences suivantes (équations récursives) donnent le coût pour passer de la chaîne A à la chaîne B, en supposant que les opérations possibles sont l'insertion, la suppression et la substitution.

$$
\begin{aligned}
D(0,0) &= 0 \\
D(i,0) &= D(i-1,0) + \text{coût}_{sup}(\texttt{ch1}[i-1]) \\
D(0,j) &= D(0,j-1) + \text{coût}_{ins}(\texttt{ch2}[j-1]) \\
D(i,j) &= \texttt{min} \begin{cases} D(i-1,j) + \text{coût}_{sup}(\texttt{ch1}[i-1]) \\ D(i,j-1) + \text{coût}_{ins}(\texttt{ch2}[j-1]) \\ D(i-1,j-1) + \text{coût}_{sub}(\texttt{ch1}[i-1], \texttt{ch2}[j-1]) \end{cases}
\end{aligned}
$$

$$\texttt{distance}(\texttt{ch1}, \texttt{ch2}) = D(\texttt{ch1.size}, \texttt{ch2.size})$$

Pour simplifier, on va supposer que les coûts des diverses opérations sont définis comme suit, c'est-à-dire que chaque opération a le même coût (coût unitaire) sont les mêmes pour toutes les opérations :

Suppression	$\text{coût}_{sup}(c) = 1$
Insertion	$\text{coût}_{ins}(c) = 1$
Substitution	$\text{coût}_{sub}(c, d) = (c \texttt{ == } d \texttt{ ? } 0 \texttt{ : } 1)$

(*) Note :

- `"abc"[0...0] == ""`
- `"abc"[0...3] == "abc"`

9.1.4 Une solution séquentielle avec «programmation dynamique» basée sur les équations de récurrence

Stratégie de programmation dynamique :

= on **précalcule** — on «mémorise» — dans une **table** les résultats des divers appels récursifs

= on effectue les divers calculs **de façon ascendante** — plutôt que *descendante* comme dans un algorithme purement récursif

⇒ on procède des cas de base **vers** les cas plus complexes

Programme Ruby 9.2 Fonction séquentielle non-récursive utilisant la programmation dynamique pour calculer la distance d'édition entre deux chaines avec un cout unitaire pour les opérations.

```ruby
def distance( ch1, ch2 )
  n1 = ch1.size
  n2 = ch2.size
  d = Matrice.new( n1+1, n2+1 )

  # Cas de base (couts unitaires).
  d[0,0] = 0
  (1..n1).each do |i|
    d[i, 0] = i
  end
  (1..n2).each do |j|
    d[0, j] = j
  end

  # Cas recursifs.
  ((1..n1)*(1..n2)).each do |i, j|
    d[i, j] = [ d[i-1, j] + 1,
                d[i, j-1] + 1,
                d[i-1, j-1] + cout_subst( ch1[i], ch2[j] )
              ].min
  end

  d[n1, n2]
end
```

```
      a x e
    - - - -
  | 0 1 2 3
a | 1 1 2 3
d | 2 2 2 2
```

Figure 9.4: Contenu de la matrice `d` à la fin de la méthode `distance` pour `ch1 = "ad"` et `ch2 = "axe"`.

```
      s u r v e y
    - - - - - - -
  | 0 1 2 3 4 5 6
s | 1 0 1 2 3 4 5
u | 2 1 0 1 2 3 4
r | 3 2 1 1 2 3 4
g | 4 3 2 2 1 2 3
e | 5 4 3 3 2 2 3
r | 6 5 4 4 3 2 3
y | 7 6 5 5 4 3 2
```

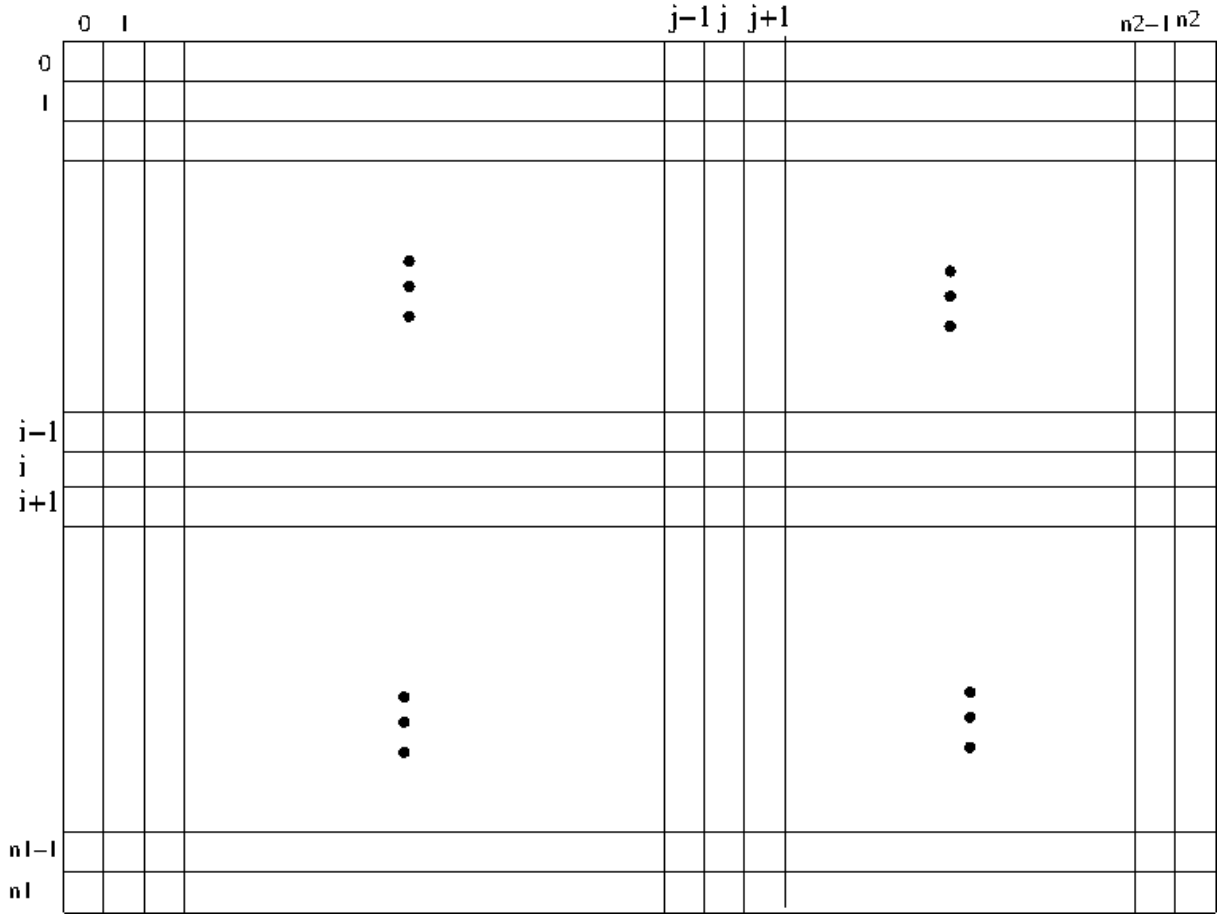
Figure 9.5: Contenu de la matrice `d` à la fin de la méthode `distance` pour `ch1 = "surgery"` et `ch2 = "survey"`.

9.1.5 Analyse des dépendances pour une version parallèle de la solution avec «programmation dynamique»

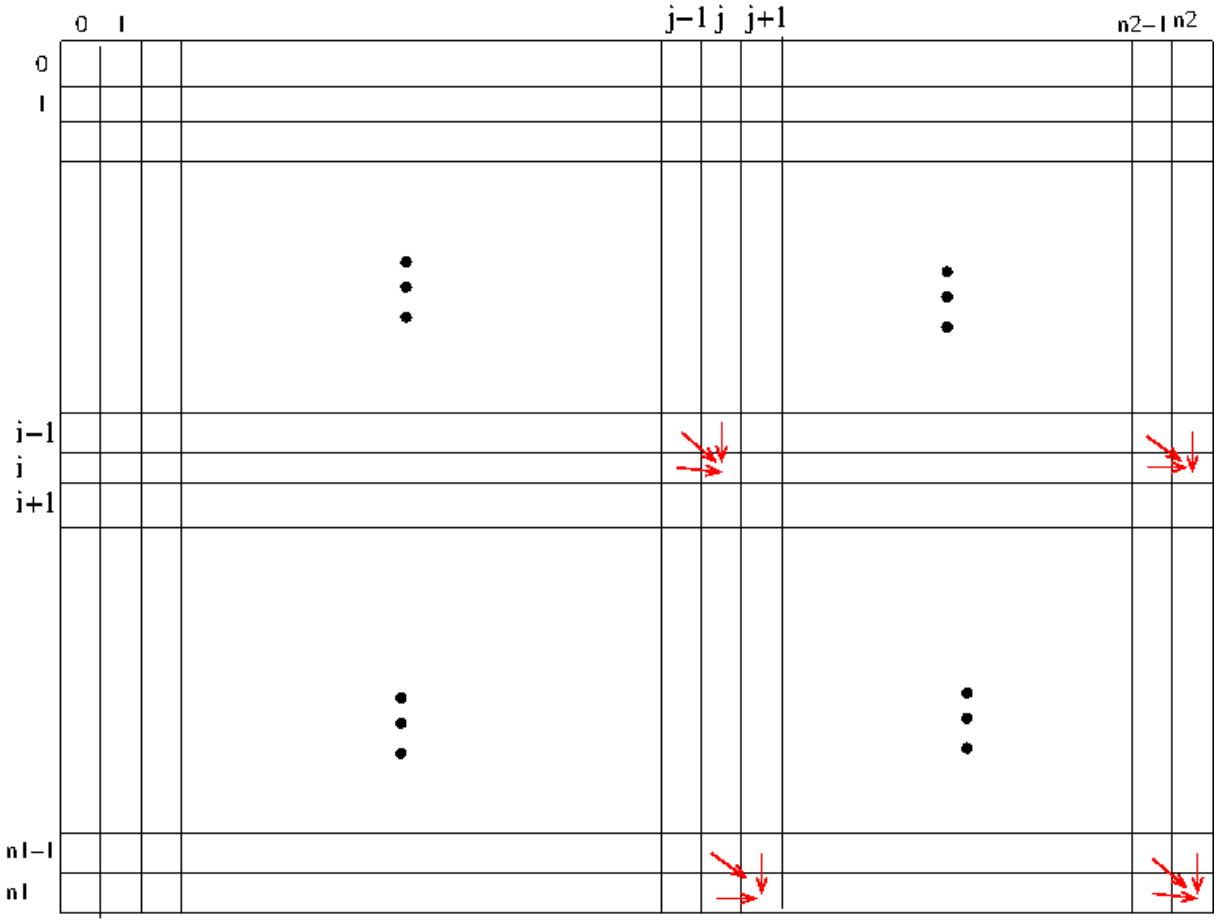
1. Quelles sont les tâches les plus fines de la version séquentielle?
2. Quelles sont les dépendances entre ces tâches?

Exercice 9.2: Tâches et dépendances

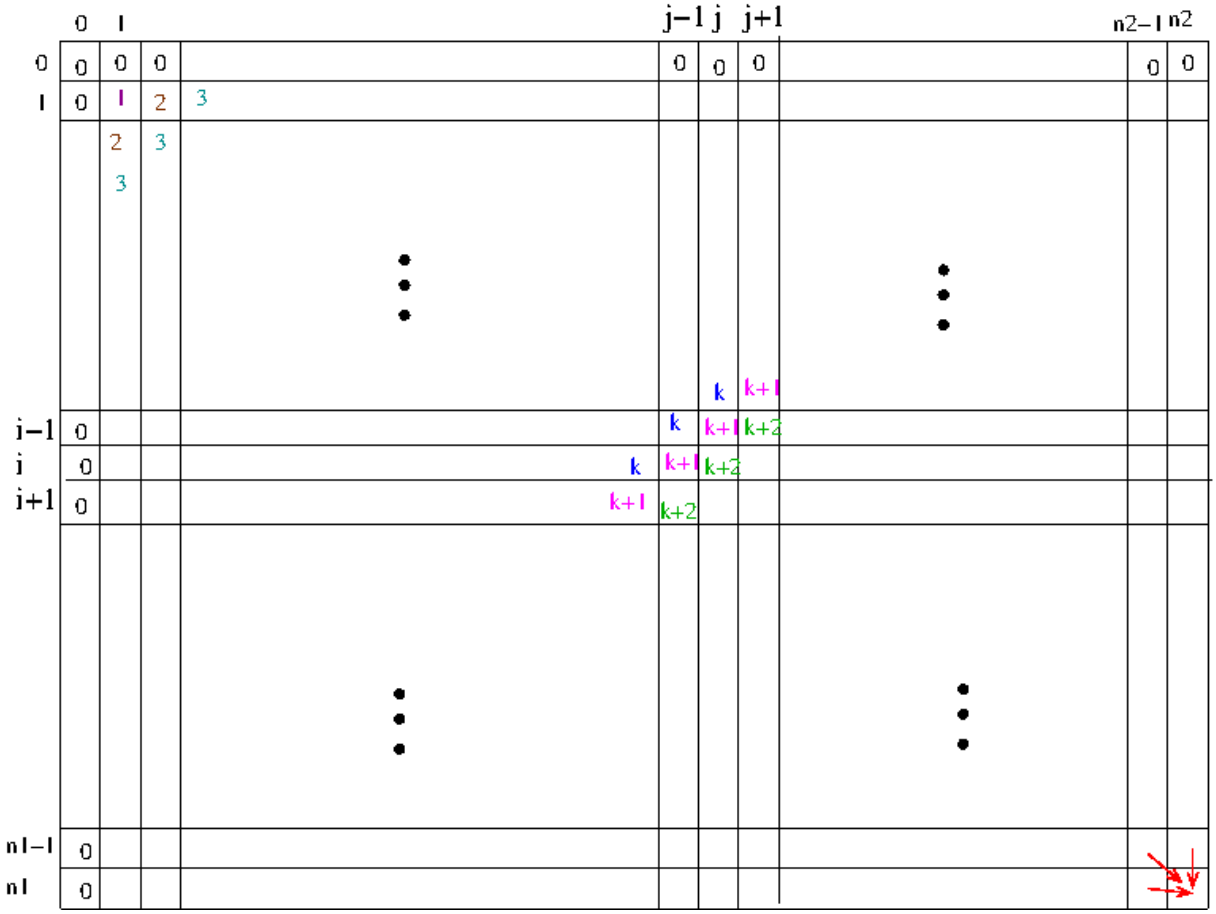
Parallélisme de résultat : Chaque entrée de la matrice peut être vue comme une tâche indépendante



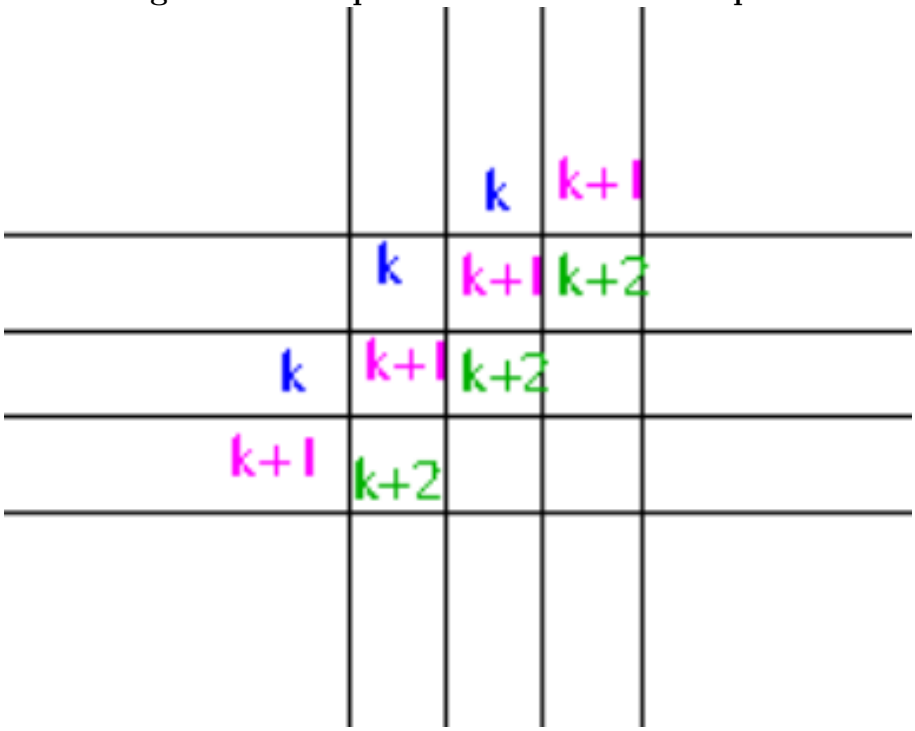
Parallélisme de résultat : Si on ignore la première ligne et la première colonne, la position `[i,j]` dépend des positions `[i-1,j]`, `[i,j-1]` et `[i-1,j-1]`



Parallélisme de résultat : Si on suppose que la première ligne et la première colonne sont évaluées au temps 0, on peut alors numéroter comme suit le moment où les éléments **pourront** être évalués :



On appelle un tel patron d'évaluation un calcul ***wavefront*** **: les éléments sur une même ligne de front peuvent être évalués en parallèle**



Programme Ruby 9.3 Fonction parallèle non-récursive utilisant la programmation dynamique pour calculer la distance d'édition entre deux chaines avec un cout unitaire pour les opérations.

Programme omis

9.2 Résolution numérique de l'équation de diffusion de la chaleur dans un cylindre

9.2.1 Introduction

De nombreux phénomènes physiques sont modélisés par des *équations différentielles* : diffusion de la chaleur, propagation d'ondes (sonores, sismiques, magnétiques), déplacement d'un pendule, radioactivité, évolution de populations animales, etc.

Certaines de ces équations peuvent être résolues de façon *analytique* (symbolique). Par contre, de nombreuses équations **ne peuvent pas** être résolues de cette façon. Dans ce dernier cas, on peut alors utiliser des **approximations numériques** des solutions recherchées.

C'est ce que nous verrons dans ce document avec un exemple *concret* simple portant sur la *diffusion de la chaleur dans un cylindre métallique*.

Nous verrons aussi comment obtenir un programme *parallèle* produisant cette solution numérique, et ce en utilisant la stratégie PCAM de Foster [Fos95].

Ces notions de base nous permettront ensuite de comprendre les différentes solutions MPD (vues en cours) pour le problème de la diffusion de la chaleur dans une grille à deux (2) dimensions.

9.2.2 Le problème et sa solution

Description du problème

Soit un cylindre métallique de longueur l,[1] Le cylindre est initialement à une température t. Les extrémités gauche et droite du cylindre sont **maintenues** à une température constante à l'aide de sources de chaleur et de thermostats. Les températures aux extrémités sont respectivement t_{gauche} à gauche et t_{droite} à droite. Quant à la température pour le reste du cylindre, on suppose qu'elle est la même partout, soit une température de $t_{initial}$. On suppose que le cylindre est isolé thermiquement sur toute sa longueur, donc on peut ignorer les pertes de chaleur. Voir figure 9.6 pour une représentation du cylindre.

On veut déterminer de quelle façon sera *distribuée* la température le long du cylindre lorsque le point d'équilibre sera atteint, donc comment la température sera diffusée des extrémités vers l'ensemble du cylindre, en tenant compte du gradient de température (allant de t_{gauche} à gauche jusqu'à t_{droite} à droite).

[1] La longueur exacte n'a aucune importance, puisqu'on peut toujours faire une mise à l'échelle. En fait, pour simplifier, on pourrait supposer une longueur unitaire, c'est-à-dire égale à 1.0.

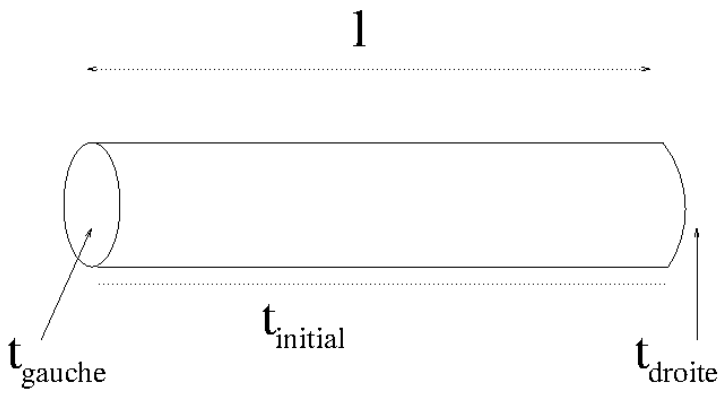


Figure 9.6: Cylindre à l'état initial.

Description générale de la solution : discrétisation de l'espace et du temps

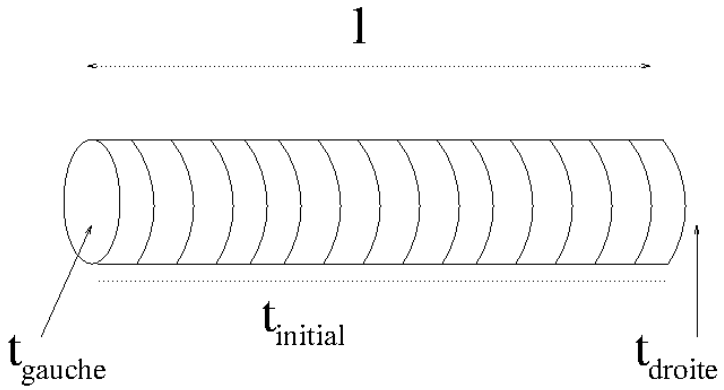


Figure 9.7: Cylindre à l'état initial mais **discrétisé** — découpé en petits segments.

Pour déterminer la distribution de la température le long du cylindre, on va *simuler* numériquement l'évolution de la température en fonction du temps. Plus spécifiquement, puisqu'on travaille avec un ordinateur et qu'on doit simuler un **espace continu** (le cylindre) que le **temps continu**, donc on va *discrétiser l'espace et le temps.* En d'autres mots, on va découper le cylindre en petits segments (discrétisation de l'espace) puis on va, de façon itérative, simuler l'avancement du temps (discrétisation du temps) en *simulant* la propagation de la chaleur entre les segments adjacents durant un «petit intervalle de temps». Idéalement, on va répéter

ce processus itératif jusqu'à ce qu'on atteigne un «point fixe», c'est-à-dire un «état stable» où il n'y a plus aucun changement.

Représentation numérique du cylindre et de la température

La seule propriéte du cylindre qui nous intéresse est sa température. Puisqu'on le découpe en petits segments adjacents, l'état de la température du cylindre peut donc être représenté par un simple vecteur de nombres réels, en supposant que chaque tranche/segment de cylindre est à température uniforme.

Pour simuler la diffusion de la chaleur entre segments, on va supposer ce qui suit, qui se veut une approximation discrète d'un processus continu :

- Durant un (court) intervalle de temps, la chaleur se propage uniquement entre deux (petits) segments *adjacents* ;
- Soit un petit segment s qui n'est pas à une des extrémités. Alors le segment à gauche de s a la même influence sur s que le segment à droite.
- L'influence de la température d'un voisin est proportionnelle au *gradient* de température, i.e., à la différence de température.

Supposons donc qu'on découpe (discrétise) le cylindre en n segments. Soit alors T un vecteur de taille n (indexé de 0 à $n-1$) qui représente la température de chacun des petits segments. Plus précisément, soit $T_t[i]$ la température du $i^{ème}$ segment au temps t — où le segment i n'est pas à l'une des extrémités. On va alors chercher à déterminer la température de ce même segment, mais cette fois au temps $t+1$. Cette température va dépendre de trois facteurs :

- De la température du segment lui-même, i.e., $T_t[i]$.
- De l'influence du segment à gauche, donc du gradient de température associé au voisin gauche, représenté par la différence suivante : $T_t[i-1] - T_t[i]$.
- De l'influence du segment à droite : $T_t[i+1] - T_t[i]$.

En se fondant sur l'hypothèse mentionnée précédemment (gauche et droite ont la même influence), on obtient alors l'équation suivante :

$$\begin{aligned} T_{t+1}[i] &= T_t[i] + \frac{(T_t[i-1] - T_t[i]) + (T_t[i+1] - T_t[i])}{2} \\ &= \frac{T_t[i-1] + T_t[i+1]}{2} \end{aligned}$$

En d'autres mots, en se fondant sur ce modèle, la température au temps $t+1$ est simplement la *moyenne* des températures des voisins adjacents! C'est ce qu'on appelle l'équation de Jacobi.

9.2.3 Un exemple concret

Soit un cylindre de longueur $l = 6$ avec $t_{gauche} = 1°C$, $t_{droite} = 10°C$ et $t_{initial} = 0°C$. On va simuler la distribution de température en décomposant le cylindre en six (6) segments de longueur unitaire. On aura alors l'évolution suivante de la distribution de température, où T_0 représente l'état initial et où les calculs sont faits avec deux chiffres de précision après le point décimal.

T_0 =	1.00	0.00	0.00	0.00	0.00	10.00
T_1 =	1.00	0.50	0.00	0.00	5.00	10.00
T_2 =	1.00	0.50	0.25	2.50	5.00	10.00
T_3 =	1.00	0.63	1.50	2.63	6.25	10.00
T_4 =	1.00	1.25	1.63	3.88	6.32	10.00

.
.
.

T_{30} =	1.00	2.74	4.59	6.39	8.19	10.00
T_{31} =	1.00	**2.80**	**4.60**	**6.40**	**8.20**	10.00
T_{32} =	1.00	**2.80**	**4.60**	**6.40**	**8.20**	10.00
T_{33} =	1.00	**2.80**	**4.60**	**6.40**	**8.20**	10.00

.
.
.

On remarque donc qu'à partir de T_{31}, on atteint un *point fixe*. En d'autres mots, la distribution de température devient stable, ne change plus du tout par la suite. C'est donc qu'on a *convergé* vers la solution finale.

Quelques remarques

Effet de l'augmentation du nombre de points

- Plus le nombre de points utilisés (le nombre de segments utilisés pour la discrétisation de l'espace) est grand — mais en autant qu'on reste dans les limites de précision de calcul — alors plus la solution numérique finale est précise. Toutefois, le temps pour arriver à un point fixe est alors plus long.

Ainsi, dans l'exemple précédent, supposons qu'on utilise 60 segments (au lieu de 6), toujours pour une précision de deux chiffres après le point. Alors :

T_0	=	1.00	0.00	0.00	0.00	...	0.00	0.00	0.00	10.00
.										
.										
.										
T_{33}	=	1.00	0.86	0.73	0.61	...	6.08	7.28	8.60	10.00

Quant au point fixe, il ne serait atteint qu'après *plusieurs centaines* d'itérations.

9.2.4 Application de l'approche PCAM de Foster

Nous allons maintenant illustrer l'approche PCAM de Foster sur le problème du cylindre, et ce en utilisant la méthode de Jacobi (première méthode la plus simple).

Partitionnement

Pour un point donné (discrétisation spatiale du cylindre), il va falloir calculer sa température au temps (discrétisation temporelle) $t = 0$, $t = 1$, $t = 2$, etc.

Notons par $T_t[i]$ la valeur du point $T[i]$ au temps t. Supposons que comme dans notre exemple précédent, nous ayons aussi 6 points ($i = 0, \ldots, 5$) et que nous voulons calculer pour $t = 0, 1, 2, \ldots, 9$.

Il nous faudra alors calculer les différentes valeurs illustrées à la figure 9.8, qui représentent donc les différentes *tâches* de granularité les plus fines.

Communications

Il nous faut ensuite identifier les dépendances entre les différentes valeurs (tâches) pour pouvoir identifier la meilleure façon de regrouper les tâches entre elles..

L'équation de Jacobi est la suivante :

$$T_{t+1}[i] \quad = \quad \frac{T_t[i-1] + T_t[i+1]}{2}$$

Les dépendances sont donc celles illustrées dans la figure 9.9.

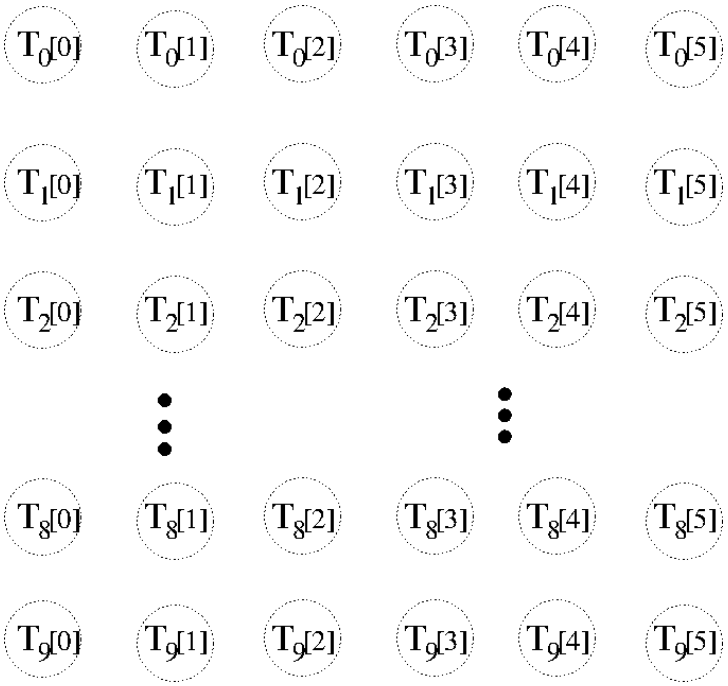


Figure 9.8: Les différentes tâches de granularité fine.

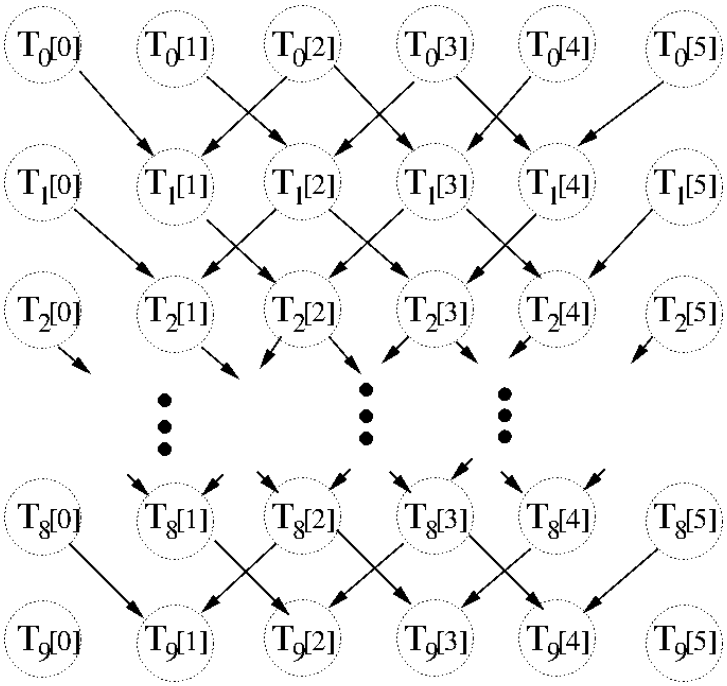


Figure 9.9: Dépendances des tâches de granularité fine.

Agglomération

À un premier niveau, on peut agglomérer les tâches de façon verticale, donc en combinant ensemble les tâches responsables de calculer, pour un point donné, les différentes valeurs dans le temps. En d'autres mots, on effectue alors une agglomération selon la dimension temporelle. On obtient alors les dépendances illustrées à la figure 9.10.

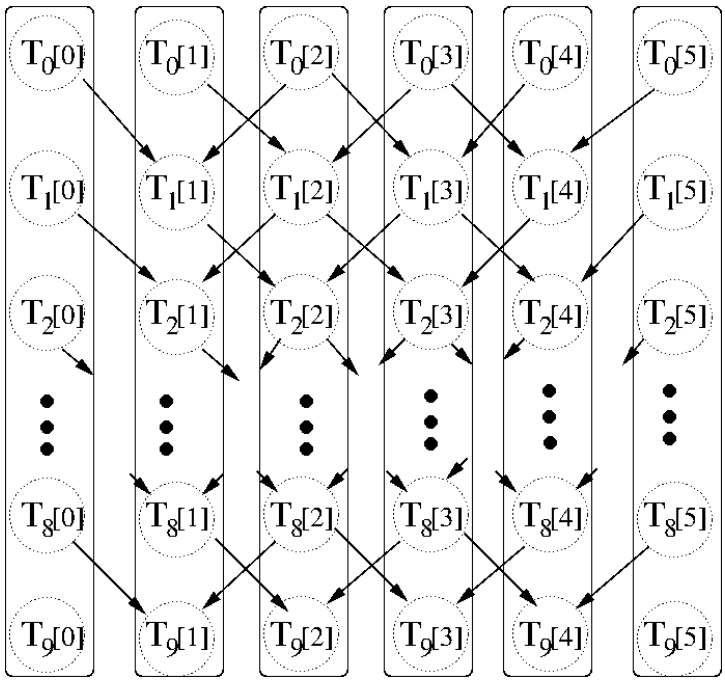


Figure 9.10: Dépendances des tâches lorsqu'une tâche est pour un seul et unique point (segment) — agglomération temporelle.

À un deuxième niveau, on peut ensuite agglomérer les tâches de façon horizontale, donc en combinant les tâches responsables de différents points adjacents. En d'autres mots, on effectue alors une agglomération selon la dimension spatiale. Si on suppose qu'on cible une machine avec trois processeurs, et qu'on veut obtenir trois processus, on obtiendrait alors les dépendances illustrées à la figure 9.11. Signalons que seules les flèches plus larges/épaisses représentent des dépendances qui donneront lieu à des communications inter-processus.

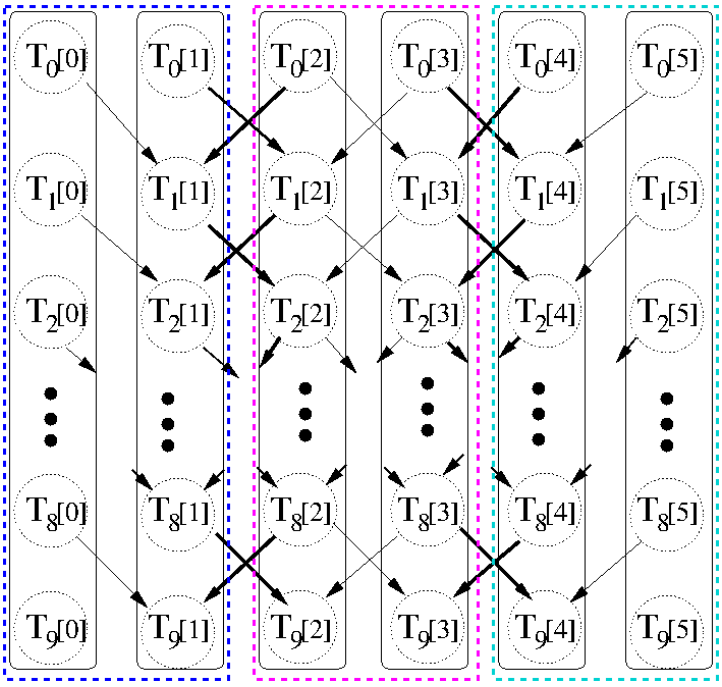


Figure 9.11: Dépendances des tâches lorsqu'une tâche est pour un groupe de points (segments) — agglomération spatiale.

Mapping

Dans les solutions que nous allons voir (en cours), nous utiliserons des associations *statiques* entre tâches et *threads*, typique des programmes SPMD (*Single Program, Multiple Data*), ou autres...

9.2.5 Accélération de la convergence

L'équation utilisée plus haut pour le calcul est celle dite *de Jacobi* :

$$T_{t+1}[i] \quad = \quad \frac{T_t[i-1] + T_t[i+1]}{2}$$

Or, lorsqu'on calcule de façon séquentielle, de gauche à droite, les différentes valeurs, on peut alors accélérer la convergence en utilisant *la nouvelle valeur du voisin gauche* plutôt que l'ancienne.

Dans ce cas, on utilise alors l'équation suivante dite *de Gauss–Seidel* :

$$T_{t+1}[i] \quad = \quad \frac{T_{t+1}[i-1] + T_t[i+1]}{2}$$

Pour notre exemple précédent (six points), on aurait alors les itérations suivantes :

T_0 =	1.00	0.00	0.00	0.00	0.00	10.00
T_1 =	1.00	0.50	0.25	0.13	5.06	10.00
T_2 =	1.00	0.63	0.38	2.72	6.36	10.00
T_3 =	1.00	0.69	1.70	4.03	7.02	10.00

.
.
.

T_{15} =	1.00	**2.80**	**4.60**	**6.40**	**8.20**	10.00
T_{16} =	1.00	**2.80**	**4.60**	**6.40**	**8.20**	10.00
T_{17} =	1.00	**2.80**	**4.60**	**6.40**	**8.20**	10.00

.
.
.

Parallélisation de la méthode de Gauss–Seidel

Le désavantage de la méthode de Gauss-Seidel est qu'elle n'est pas parallélisable à cause de la dépendence séquentielle créée par le balayage des points de la gauche vers la droite.

$$T_{t+1}[i] \quad = \quad \frac{T_{t+1}[i-1] + T_t[i+1]}{2}$$

Mais, Il est quand même possible de paralléliser cette méthode, pour profiter de la convergence plus rapide, en effectuant l'ensemble des calculs *en deux (2) passes*.

Plus précisément, on divise l'ensemble des points en deux sous-groupes, typiquement appelés les points rouges (indices pairs) vs. les points *noirs* (indices impairs). Pour le calcul à un temps t donné, on calcule tout d'abord l'ensemble des valeurs pour les points rouges — qui ne dépendent que des points noirs. Ensuite, on effectue le calcul pour les points *noirs* — qui ne dépendent que des points rouges, déjà calculés donc de temps $t+1$.

Dans notre exemple précédent avec six points, on aurait alors les valeurs suivantes pour la première itération :

	R	N	R	N	R	N
$T_0 =$	1.00	0.00	0.00	0.00	0.00	10.00
$T_1^R =$	1.00		0.00		5.00	
$T_1^N =$		0.50		2.50		10.00
$T_1 =$	1.00	0.50	0.00	2.50	5.00	10.00
$T_2^R =$	1.00		1.50		6.25	
$T_2^N =$		1.25		3.88		10.00
$T_2 =$	1.00	1.25	1.50	3.88	6.25	10.00

9.3 Opérations sur des polynomes

9.3.1 Définition du problème

On désire créer et manipuler des polynomes comportant *un (très!) grand nombre* de coefficients.L'opération la plus importante — et la plus intéressante au niveau du parallélisme — est celle consistant à faire *le produit* de deux polynomes.

Remarque : Le produit de polynomes peut être vu comme une version *simplifiée* de divers problèmes, par exemple, *i*) la manipulation d'entiers (ou de réels) «à précision infinie» — donc des entiers (ou réels) comportant autant de chiffres que nécessaire, limité seulement par la quantité de mémoire disponible ; *ii*) le produit de convolution discrète[2] utilisé en traitement de signal (*DSP = Digital Signal Processing*).

Soit p et q deux polynomesrespectivement de degré n et m :

$$\begin{aligned} p(x) &= p_0 + p_1 x + p_2 x^2 + \ldots + p_{n-1} x^{n-1} + p_n x^n \\ &= \sum_{k=0}^{n} p_k x^k \end{aligned}$$

$$\begin{aligned} q(x) &= q_0 + q_1 x + q_2 x^2 + \ldots + q_{m-1} x^{m-1} + q_m x^m \\ &= \sum_{k=0}^{m} q_k x^k \end{aligned}$$

9.3.2 Somme de deux polynomes

Sans perte de généralité, supposons que $n \leq m$. La somme de deux polynomes est alors définie simplement comme suit :

$$p(x) + q(x) = \sum_{k=0}^{n} (p_k + q_k) x^k + \sum_{k=n+1}^{m} q_k x^k$$

[2] `https://fr.wikipedia.org/wiki/Produit_de_convolution`

9.3.3 Une première façon d'effectuer le produit de deux polynomes

Comment sera défini $p(x) * q(x)$, le produit de $p(x)$ et $q(x)$?

Examinons un exemple simple, avec des valeurs spécifiques concrètes :

$$\begin{aligned} p(x) &= 2 + 3x + 4x^2 \\ q(x) &= 10 + x + 2x^2 + 5x^3 \end{aligned}$$

Une première façon d'effectuer le produit, semblable à celle vue à l'école, serait la suivante :

$$\begin{aligned} p(x) * q(x) &= 2 * (10 + x + 2x^2 + 5x^3) \\ &+ 3x * (10 + x + 2x^2 + 5x^3) \\ &+ 4x^2 * (10 + x + 2x^2 + 5x^3) \\ \\ &= 20 + 2x + 4x^2 + 10x^3 \\ &+ 30x + 3x^2 + 6x^3 + 15x^4 \\ &+ 40x^2 + 4x^3 + 8x^4 + 20x^5 \\ \\ &= 20 + 32x + 47x^2 + 20x^3 + 23x^4 + 20x^5 \end{aligned}$$

9.3.4 Une représentation des polynomes

Nous allons utiliser la représentation suivante pour des polynomes $p(x)$ et $q(x)$ respectivement de degré n et m — c'est-à-dire que nous allons utiliser un simple tableau (`Array`) des coefficients :

`p = [`p_0`, `p_1`, ..., `p_{n-1}`, `p_n`]`

`q = [`q_0`, `q_1`, ..., `q_{m-1}`, `q_m`]`

- On suppose une représentation *normalisée* des polynomes, c'est-à-dire que le coefficient le plus à droite *n'est jamais 0* — sauf pour le polynome zéro = `[0]`.

  Exception : $p(x) = 0 \Rightarrow$ `p = [0]`

- Avec une représentation normalisée, la taille du tableau (le nombre de coefficients) représentant un polynome de degré n est $n + 1$.

9.3.5 Produit séquentiel : $1^{ère}$ version

Une première façon pour «tenter» de paralléliser le produit, s'inspirant de la méthode «manuelle», pourrait reposer sur la méthode séquentielle suivante :

```
def fois( p, q )
  polys = (0...p.size).map do |k|
    ([0] * k) + q.map { |q_k| p[k] * q_k }
  end

  polys.reduce([0]) do |produit, p|
    plus( produit, p )
  end
end
```

On pourrait alors paralléliser en utilisant `pmap` et `preduce`.

Question : Peut-on paralléliser cet algorithme?

Question : Quels sont les avantages ou désavantages?

9.3.6 Une deuxième façon d'effectuer le produit de deux polynomes

Une autre façon de représenter le produit ci-haut serait la suivante, où r est le polynome résultat et dont les coefficients sont définis comme suit :

$$\begin{aligned} p(x) * q(x) &= 20 + 2x + 4x^2 + 10x^3 \\ &+ \quad 30x + 3x^2 + 6x^3 + 15x^4 \\ &+ \quad\quad 40x^2 + 4x^3 + 8x^4 + 20x^5 \\ &= 20 + 32x + 47x^2 + 20x^3 + 23x^4 + 20x^5 \end{aligned}$$

Donc :

$$\begin{aligned} r_0 &= 2 * 10 = 20 \\ r_1 &= 2 * x + 3x * 10 = 32x \\ r_2 &= 2 * 2x^2 + 3x * x + 4x^2 * 10 = 47x^2 \\ r_3 &= 2 * 5x^3 + 3x * 2x^2 + 4x^2 * x = 20x^3 \\ r_4 &= 3x * 5x^3 + 4x^2 * 2x^2 = 23x^4 \\ r_5 &= 4x^2 * 5x^3 = 20x^5 \end{aligned}$$

Note : $2 * 10 = 2x^0 * 10x^0 = 20x^0 = 20$

Une façon de présenter ces opérations, en termes d'indices et non de valeurs spécifique, est la suivante :

$$\begin{aligned} r_0 &= p_0 * q_0 = 20x^0 \\ r_1 &= (p_0 * q_1 + p_1 * q_0) * x^1 = 32x^1 \\ r_2 &= (p_0 * q_2 + p_1 * q_1 + p_2 * q_0) * x^2 = 47x^2 \\ r_3 &= (p_0 * q_3 + p_1 * q_2 + p_2 * q_1) * x^3 = 20x^3 \\ r_4 &= (p_1 * q_3 + p_2 * q_2) * x^4 = 23x^4 \\ r_5 &= (p_2 * q_3) * x^5 = 20x^5 \end{aligned}$$

Donc, de façon générale :

$$\begin{aligned} p(x) &= \sum_{k=0}^{n} p_k x^k \\ q(x) &= \sum_{k=0}^{m} q_k x^k \end{aligned}$$

$$r(x) = \sum_{k=0}^{n+m} r_k x^k$$

Avec r_k — le k-ième coefficient de r — défini comme suit :

$$r_k = \sum_{\substack{i,j \geq 0 \\ i+j=k}} p_i * q_j$$

9.3.7 Produit séquentiel : 2^e version

Une deuxième façon de «tenter» de paralléliser le produit de deux polynomes pourrait reposer sur la méthode séquentiell suivante :

```
def fois( p, q )
  nb = p.size + q.size - 1
  r = Array.new(nb){ 0 }

  p.each_index do |i|
    q.each_index do |j|
       r[i+j] += p[i] * q[j]
    end
  end
end
```

Question : Peut-on paralléliser cet algorithme?

Question : Quels sont les avantages ou désavantages?

9.3.8 Produit séquentiel : 3^e version

Une troisième façon, basée sur du *parallélisme de résultat*, pourrait reposer sur la méthode sequentielle suivante :

```
def fois( p, q )
  nb = p.size + q.size - 1
  r = Array.new(nb) { 0 }

  r.each_index do |k|
    # Calcul de r[k]
    p.each_index do |i|
      q.each_index do |j|
        r[k] += p[i] * q[j] if i+j == k
      end
    end
  end
end
```

Question : Peut-on paralléliser cet algorithme?

Question : Quels sont les avantages ou désavantages?

9.3.9 Produit séquentiel : 4^e version

Semblable à la version précédente, mais...

- Utilise un style fonctionnel
- Introduit une méthode auxiliaire `coefficient` — qui encapsule ce qui était fait par les deux boucles internes dans la version précédente, mais sans faire un parcours complet
- Complexité $O(n \times m)$

```
def fois( p, q )
  nb = p.size + q.size - 1

  (0...nb).map do |k|
    coefficient(k, p, q)
  end
end

def coefficient( k, p, q )
  exp_min = [ 0, k-q.size+1 ].max
  exp_max = [ k, p.size-1 ].min
  # assert exp_min <= exp_max

  (exp_min..exp_max).reduce(0) do |somme, i|
    somme + p[i] * q[k-i]
  end
end
```

De quelle façon peut-on paralléliser `fois`, et ce avec le plus de parallélisme possible?

Exercice 9.3: Parallélisation de `fois`.

Dans le `pmap`, de quelle façon devrait-on répartir les itérations entre les *threads*?

Exercice 9.4: Utilisation du `pmap` dans `fois`.

Références

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