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 utilisée pour définir une mesure de **similarité** entre chaînes (de caractères, de symboles, de gènes, etc.).

Différentes distances ont été introduites, par ex., traitement et analyse de textes, analyse de protéines et de génomes en bio-informatique, et même correction de dictées musicales.

L'idée est de déterminer le nombre **minimum** d'opérations qui doivent être appliquées sur la première chaîne pour obtenir la deuxième.

```
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.

```ruby
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
```

```
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", "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**.

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.

9.1.3 Une formalisation du problème avec des équations de récurrence

Soit deux chaînes `ch1` et `ch2`.

Notons $D(i,j)$ le **coût minimal** pour transformer `ch1[0...`i`]` en `ch2[0...`j`]` `(*)`.

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

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

On suppose des coûts identiques (unitaires) :

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

(*) 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 appels récursifs

= on effectue les calculs **de façon ascendante**

$\Rightarrow$ 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

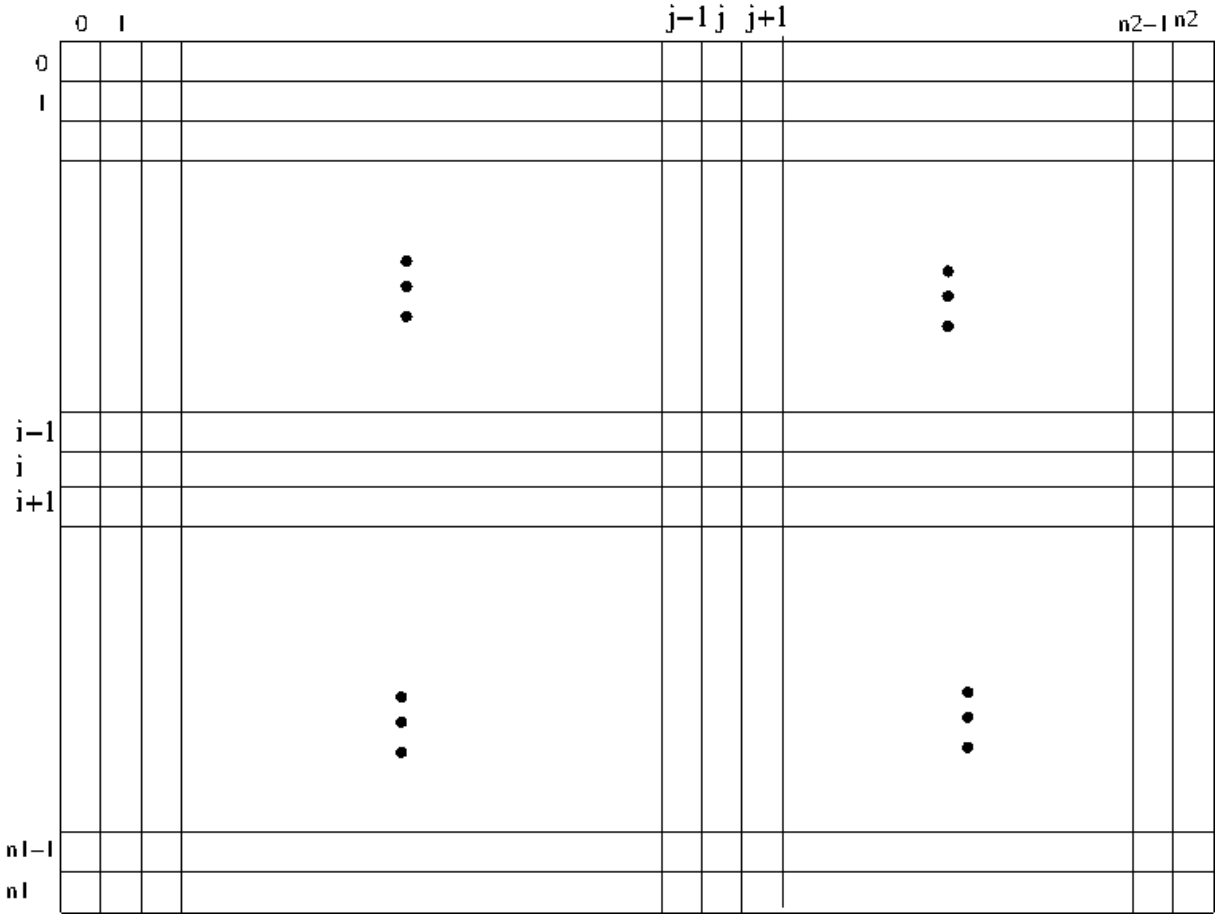
Figure 9.5: Contenu de la matrice `d` à la fin de la méthode `distance` pour `ch1 = "surgery"` et `ch2 = "survey"`.

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

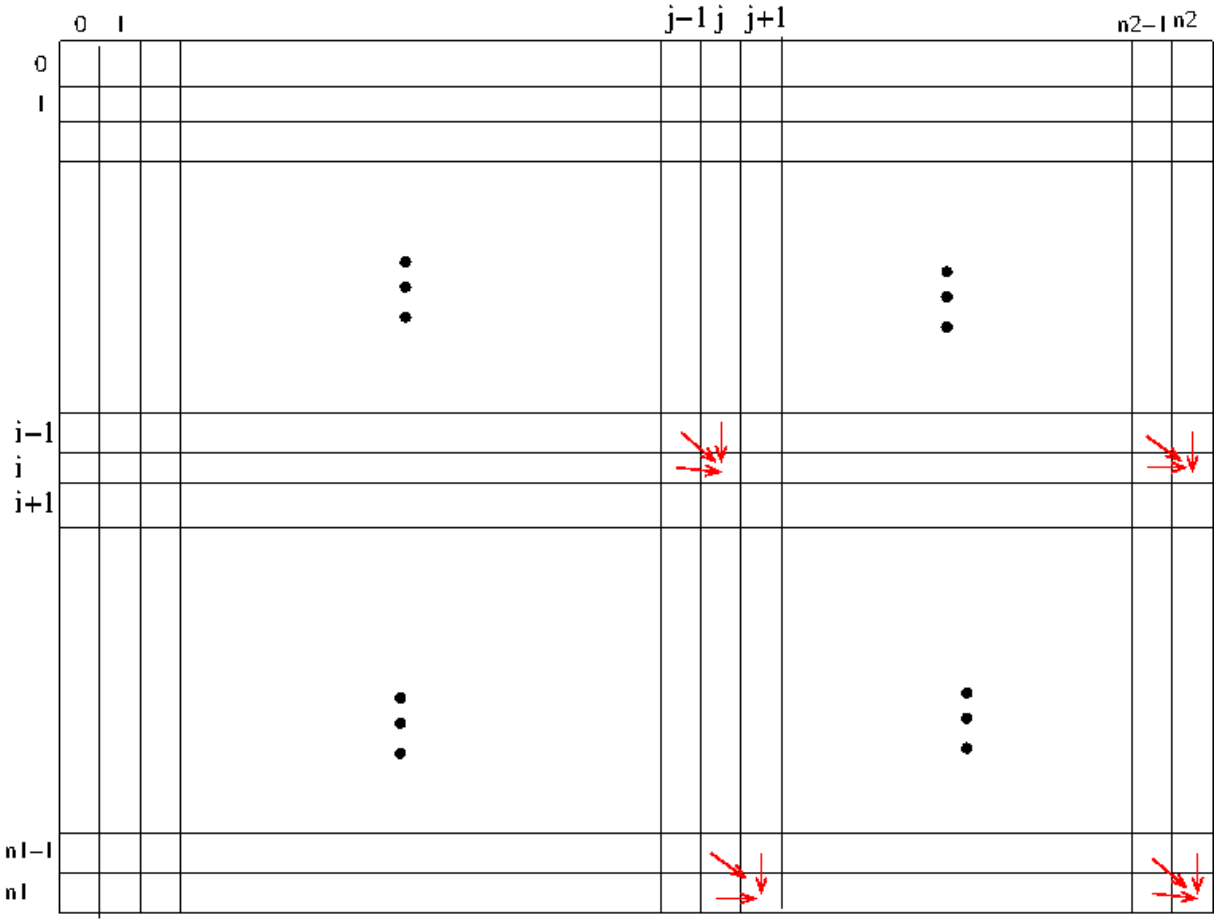
1. Quelles sont les tâches les plus fines de la version séquentielle?
2. Quelles sont les dépendances entre ces tâches?

Exercice 9.2: Tâches et dépendances

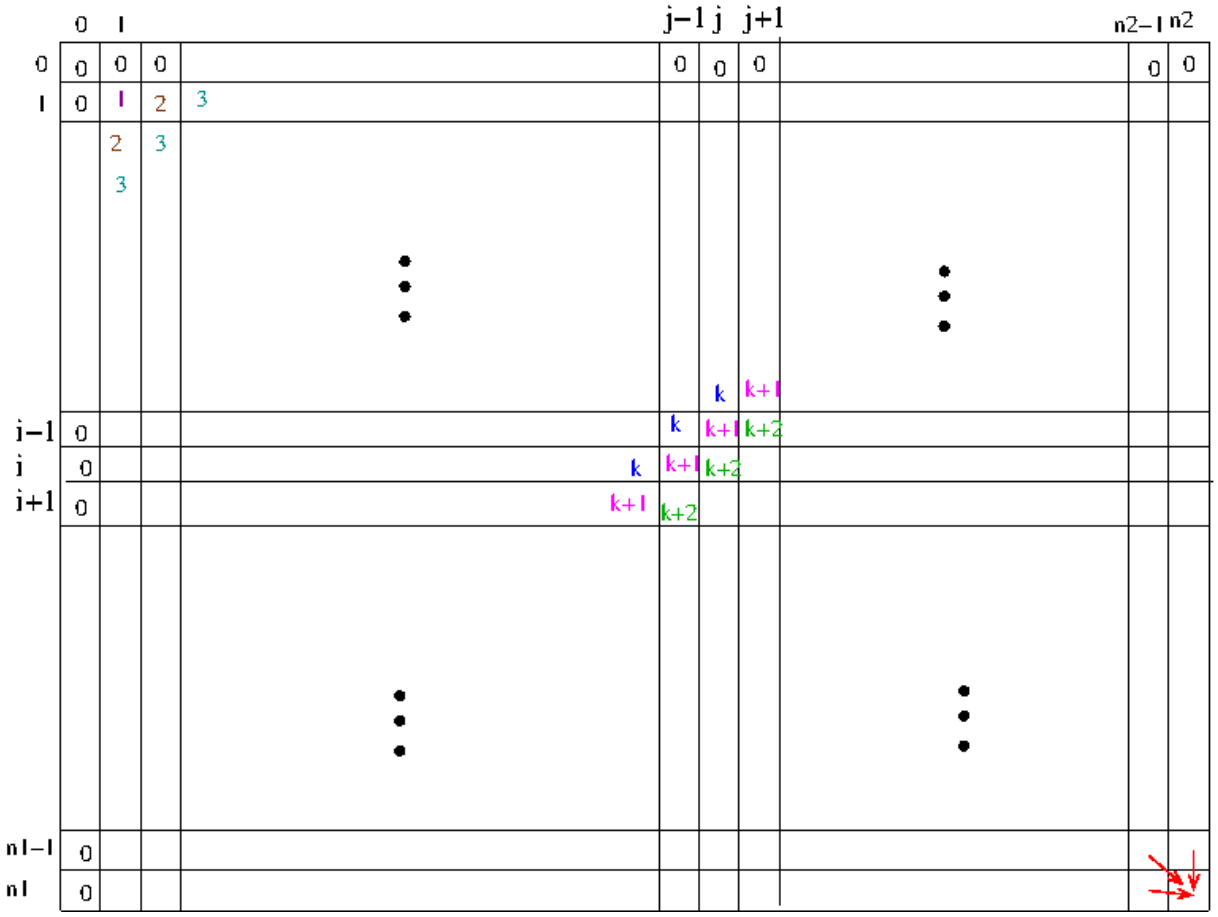
Parallélisme de résultat : Chaque entrée de la matrice peut être vue comme une tâche indépendante



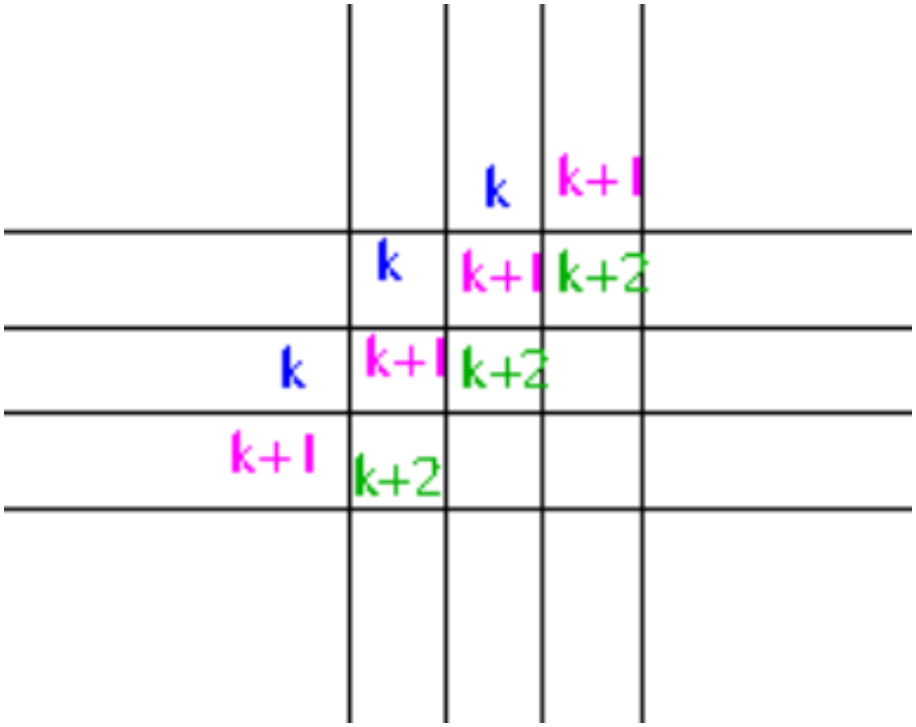
Parallélisme de résultat : Si on ignore la première ligne et la première colonne, la position `[i,j]` dépend des positions `[i-1,j]`, `[i,j-1]` et `[i-1,j-1]`



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, radioactivité, évolution de populations, etc.

Certaines de ces équations peuvent être résolues de façon *analytique*.

Par contre, de nombreuses équations **ne peuvent pas** être résolues de cette façon.

Dans ce dernier cas, on utilise des **approximations numériques**.

9.2.2 Le problème et sa solution

Description du problème

Soit un cylindre métallique, initialement à une température t. Les extrémités sont **maintenues** à une température constante.

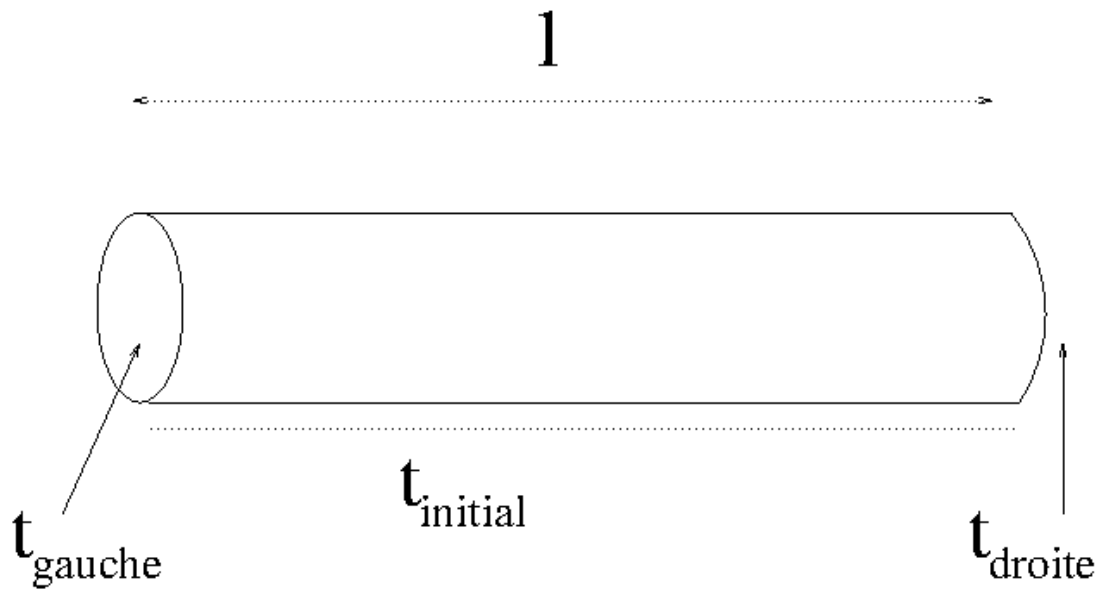


Figure 9.6: Cylindre à l'état initial.

On veut déterminer comment sera **distribuée** la température.

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

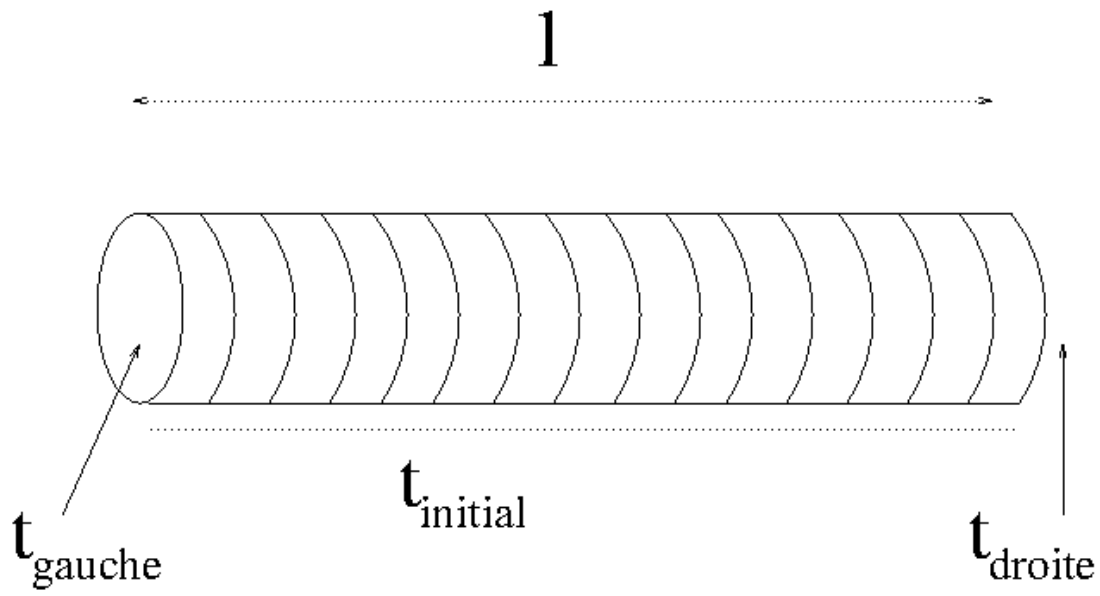


Figure 9.7: Cylindre à l'état initial mais **discrétisé**.

On va **simuler** l'évolution de la température en fonction du temps.

Plus spécifiquement, on doit simuler tant un **espace continu** que le **temps continu**, donc on va discrétiser l'espace et le temps.

Représentation numérique

L'état de la température = vecteur de réels.

- Durant un (court) laps de temps, la chaleur se propage uniquement entre segments **adjacents** ;
- Le segment gauche a la même influence que le segment droit.
- L'influence d'un voisin est proportionnelle au **gradient** de température.

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

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

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.

$T_0 \;=$	1.00	0.00	0.00	0.00	0.00	10.00

$T_0 =$	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_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_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_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_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_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_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

.

.

.

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

.

.

.

Quelques remarques

Effet de l'augmentation du nombre de points

Dans l'exemple précédent, supposons qu'on utilise 60 segments :

T_0	$=$	1.00	0.00	0.00	0.00	$\ldots$	0.00	0.00	0.00	10.00

.

.

.

T_{33}	$=$	1.00	0.86	0.73	0.61	$\ldots$	6.08	7.28	8.60	10.00

Le point fixe ne serait atteint qu'après **plusieurs centaines** d'itérations.

9.2.4 Application de l'approche PCAM de Foster

Partitionnement

Notons par $T_t[i]$ la valeur du point $T[i]$ au temps t.

Supposons que nous ayons 6 points ($i = 0, \ldots, 5$) et que nous voulons calculer pour $t = 0, 1, 2, \ldots, 9$.

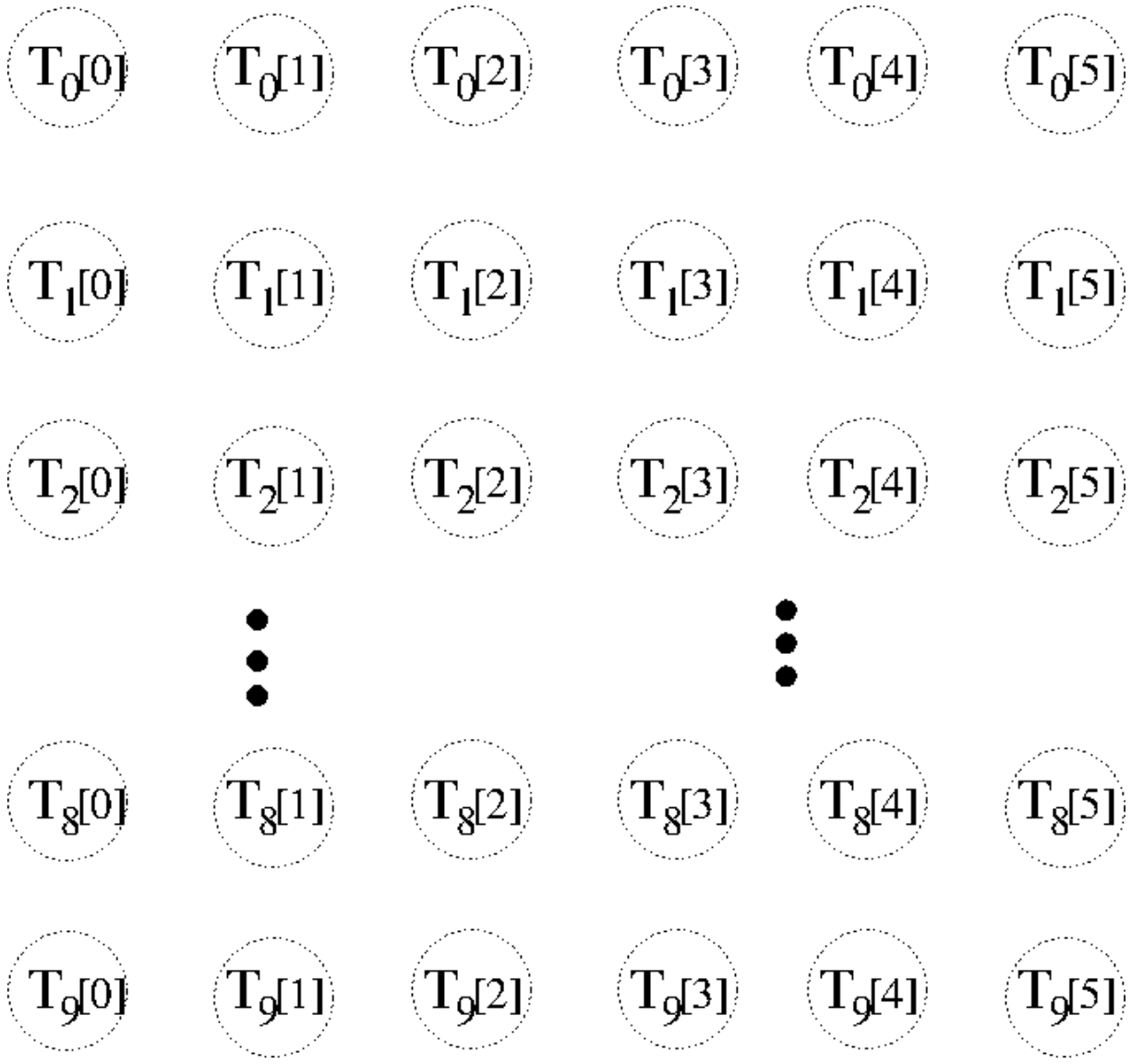


Figure 9.8: Les différentes tâches de granularité fine.

Communications

Il faut identifier les dépendances entre les différentes valeurs (tâches) pour trouver comment les regrouper..

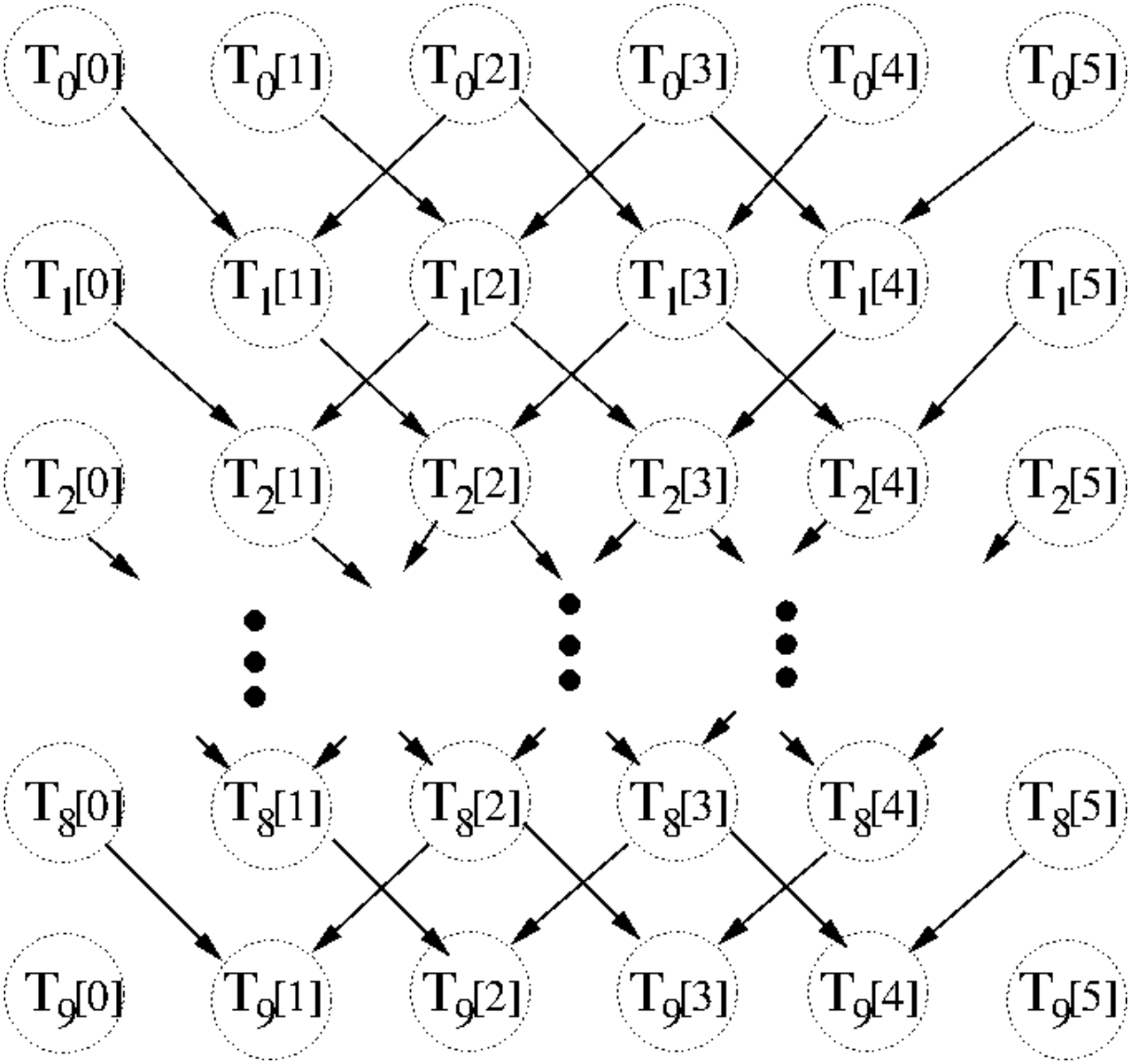


Figure 9.9: Dépendances des tâches de granularité fine.

Agglomération

Agglomération **temporelle** : on regroupe tous les «moments dans le temps» d'un point donné.

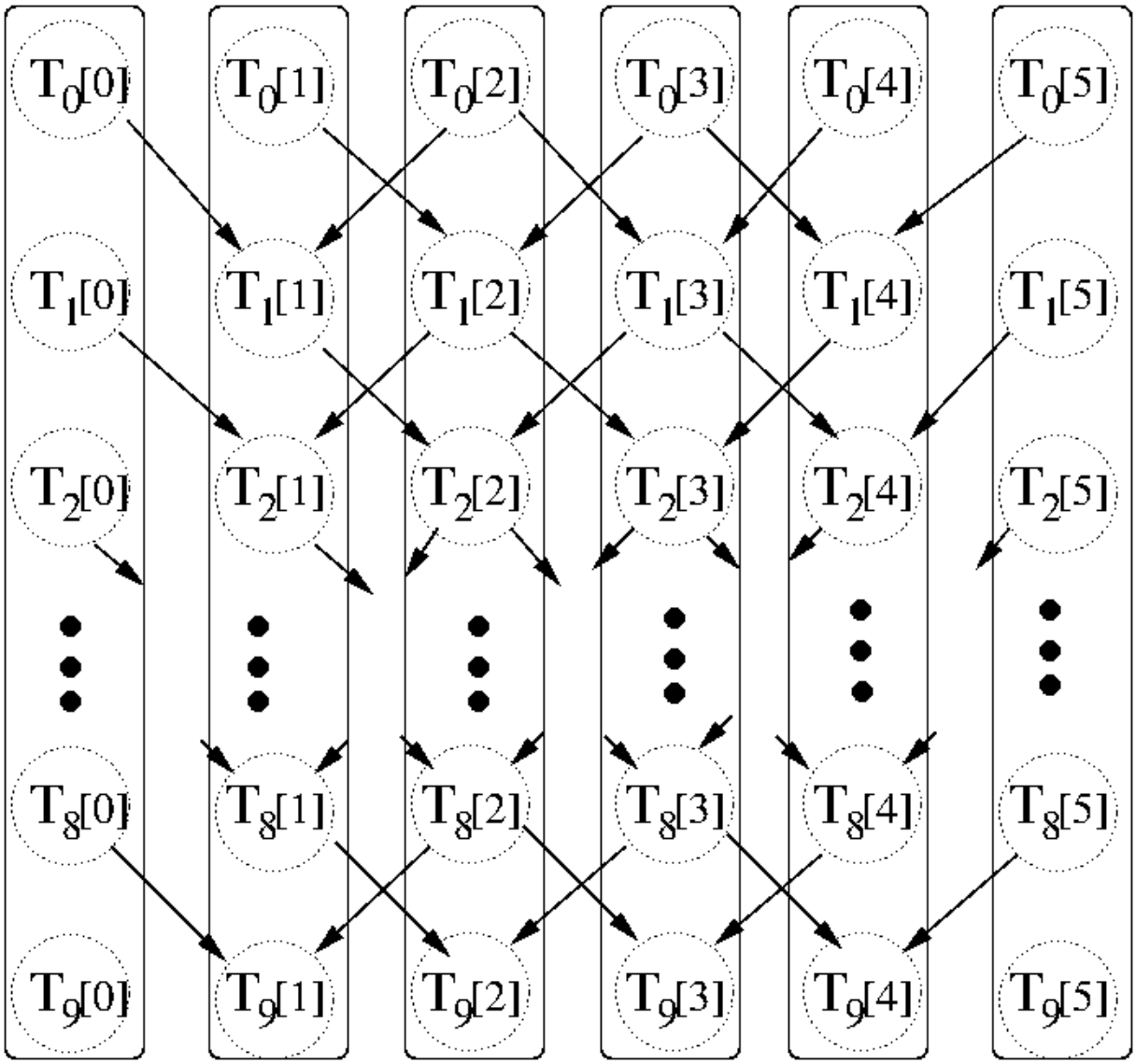


Figure 9.10: Dépendances des tâches lorsqu'une tâche est pour un seul et unique point (segment) — agglomération temporelle.

Agglomération **spatiale** : on regroupe des points voisins les uns des autres.

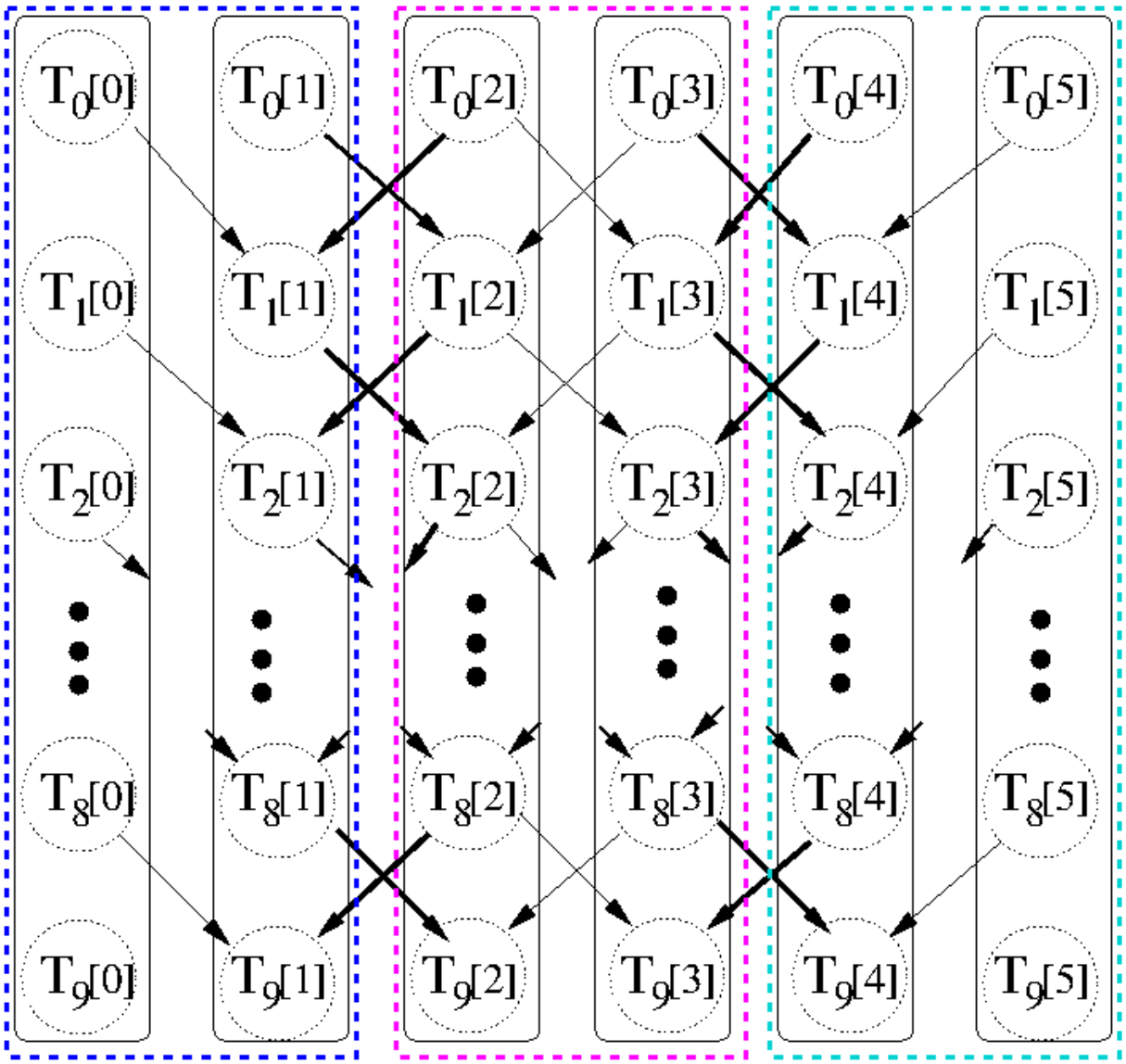


Figure 9.11: Dépendances des tâches lorsqu'une tâche est pour un groupe de points (segments) — agglomération spatiale.

Mapping

Association *statique* entre tâches et *threads* — parce que le travail est le même!

9.2.5 Accélération de la convergence

L'équation utilisée plus haut est dite **de Jacobi** :

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

On peut accélérer la convergence en utilisant **la nouvelle valeur du voisin gauche**.

On utilise alors l'équation dite **de Gauss–Seidel** :

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

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

Désavantage de Gauss-Seidel : pas parallélisable

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

Mais, on peut la paralléliser, en effectuant les calculs **en deux (2) passes**.

On divise les points en deux sous-groupes :
les rouges (pairs) vs. les **noirs** (impairs).

Pour le calcul à un temps t :
1. On calcule les points rouges.
2. On calcule les points **noirs**.

	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 manipuler des polynomes comportant **un (très!) grand nombre** de coefficients.

L'opération la plus importante est **le produit**.

Remarque : Le produit de polynomes est une version **simplifiée** de divers problèmes.

Soit p et q deux polynomes, 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

Supposons $n \leq m$.

La somme de deux polynomes est :

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

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

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

Examinons un exemple simple :

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

Une première façon d'effectuer le produit :

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

9.3.4 Une représentation des polynomes

Représentation des polynomes =
tableau (`Array`) des coefficients :

$$p(x) = p_0 + p_1 x + p_2 x^2 + \ldots + p_{n-1} x^{n-1} + p_n x^n$$

$$q(x) = q_0 + q_1 x + q_2 x^2 + \ldots + q_{m-1} x^{m-1} + q_m x^m$$

`p = [`p_0`,` p_1`,` $\ldots$`,` p_{n-1}`,` p_n`]`

`q = [`q_0`,` q_1`,` $\ldots$`,` q_{m-1}`,` q_m`]`

- Représentation **normalisée** des polynomes $\Rightarrow$ le coefficient à droite **n'est jamais 0**.
  Exception : $p(x) = 0 \Rightarrow$ `p = [0]`
- Taille pour un polynome de degré $n = n + 1$.

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

Approche s'inspirant de la méthode «manuelle» :

```
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 paralléliser en utilisant `pmap` et `preduce`.

Question : Peut-on paralléliser cet algorithme?

Question : Quels sont les avantages ou désavantages?

☺ On peut facilement paralléliser le premier `map`.

☹ On parcourt plusieurs fois le polynome `q` — une fois par coefficient de `p`.

☹ La réduction des produits intermédiaires peut se faire en parallèle — en temps logarithmique — sauf que les objets additionnés sont complexes, car ce sont des polynomes de degré $n, \ldots, m$.

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

Une autre façon de représenter le produit ci-haut :

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

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

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 représentation en termes d'indices :

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

Donc, de façon générale :

$$p(x) = \sum_{k=0}^{n} p_k x^k$$

$$q(x) = \sum_{k=0}^{m} q_k x^k$$

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

Avec r_k 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 approche :

```
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?

☹ Impossible de paralléliser les boucles, à cause des conflits d'accès — plusieurs `i` et `j` distincts peuvent avoir la même somme (même `k`) donc plusieurs *threads* pourraient tenter de mettre à jour la même position du tableau `r`.

☺ Complexité «quadratique» intéressante :

$$O(n \times m)$$

Donc : $O(n^2)$ si $n = m$.

9.3.8 Produit séquentiel : 3^e version

Une troisième approche :

```
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?

☺ Facilement parallélisable au niveau de la boucle externe, puisque chaque itération est indépendante — ne travaille que sur `r[k]`.

☹ Mise en oeuvre plus ou moins efficace, car elle parcourt l'ensemble des indices de `p` et `q` pour chaque item du résultat:

$$O(n \times m \times (n + m))$$

Donc : $O(n^3)$ si $n = m$.

9.3.9 Produit séquentiel : 4^e version

Semblable à la version précédente, mais...

- Utilise un style fonctionnel
- Introduit une méthode auxiliaire `coefficient` plus efficace pcq. n'a pas deux boucles complètes
- 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`.