Parallélisation automatique

Plan

1. Analyse de dépendances
2. Algorithme d'Allen et Kennedy
3. Transformations unimodulaires

1 Analyse de dépendances

Nids de boucles

– Nids de boucles imbriquées
  $\rightarrow$ meilleure source potentielle de parallélisme dans les programmes
– Structure régulière + code exécuté un grand nombre de fois
  $\rightarrow$ optimisations de compilation sophistiquées

– *Détection du parallélisme :*
  identification de certaines boucles comme étant parallèles

– *Compréhension du parallélisme :*
  détection des dépendances responsables de la séquentialité intrinsèque de certaines parties du code.

Vecteur d'itération

```
DO i=1, N
  DO j=i, N+1
    DO k=j-i, N
      S1
      S2
    ENDDO
  ENDDO
  DO r=1, N
    S3
  ENDDO
ENDDO
```

- Pas 1 pour chaque compteur de boucle
- Itérations de n boucles parfaitement imbriquées $\rightarrow$ *vecteur d'itération* à n composantes

- *Domaine d'itération* = ensemble de toutes les valeurs possibles du vecteur d'itération
- Les boucles i et r définissent un vecteur d'itération bi-dimensionnel (i, r) dont le domaine d'itération est l'ensemble des points entiers du carré $1 \leq i, r \leq N$
- Les boucles i, j, et k définissent un vecteur d'itération tri-dimensionnel (i, j, k) dont le domaine d'itération est l'ensemble des points entiers tels que $1 \leq i \leq N$, $i \leq j \leq N+1$, et $j - i \leq k \leq N$.

Nids de boucles et ordre séquentiel

- $S(I)$ instance de S exécutée à l'itération I.
- Instances d'une instruction exécutées suivant l'ordre $<_{seq}$
- Nid parfait $\rightarrow$ ordre lexicographique des vecteurs d'itération :

$$S(I) <_{seq} S(J) \Leftrightarrow I <_{lex} J$$

- Cas général : deux instances $S(I)$ and $T(J)$ de deux instructions S et T englobées par $n_{S,T}$ boucles communes
  1. I vecteur d'itération de taille n_S, $\tilde{I}$ les $n_{S,T}$ premières composantes
  2. J vecteur d'itération de taille n_T, $\tilde{J}$ les $n_{S,T}$ premières composantes
  3. Code C englobé par les $n_{S,T}$ boucles les plus externes
  4. Si $\tilde{I} <_{lex} \tilde{J}$, alors $C(\tilde{I}) <_{seq} C(\tilde{J})$, d'où $S(I) <_{seq} T(J)$.
  5. Si $\tilde{J} <_{lex} \tilde{I}$, alors $C(\tilde{J}) <_{seq} C(\tilde{I})$, d'où $T(J) <_{seq} S(I)$.
  6. Si $\tilde{J} = \tilde{I}$, même itération des boucles englobantes,// $S(I) <_{seq} T(J)$ ssi $S <_{text} T$
- Résumé :

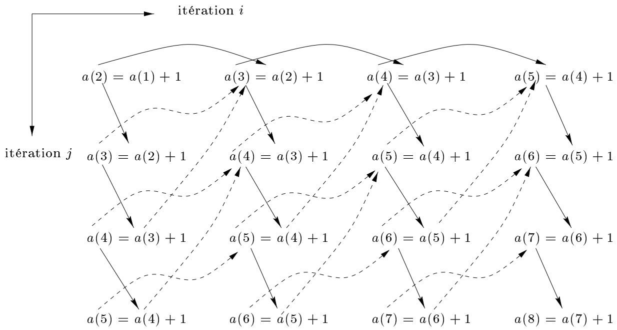
$$S(I) <_{seq} T(J) \Leftrightarrow (\tilde{I} <_{lex} \tilde{J}) \text{ ou } (\tilde{I} = \tilde{J} \text{ et } S <_{text} T)$$

- Exemple :
  $S_1(I) <_{seq} S_2(J) \Longleftrightarrow I <_{lex} J$ ou $I = J$ ($S_1 <_{text} S_2$ est vrai)
  $S_3(i, r) <_{seq} S_1(i', j, k) \Longleftrightarrow i < i'$ ($S_3 <_{text} S_1$ est faux).

Dépendances de données

- Ordre partiel $\Rightarrow$: ordre minimal à respecter pour produire un code sémantiquement équivalent au code initial
- $<_{seq}$ extension de $\Rightarrow$
- Instances d'instruction *dépendantes* : comparables par $\Rightarrow$

```
DO i=1, N
   DO j=1, N
     a(i+j) = a(i+j-1) + 1
   ENDDO
ENDDO
```



- Trois types de dépendances : flot, anti, et sortie
- Dépendances de données (en opposition aux dépendances de contrôle) : deux calculs accèdent au même emplacement mémoire, en lecture ou en écriture.

Flot, Anti, et Sortie

- *Dépendance de données* entre $S(I)$ et $T(J)$ si accès même emplacement mémoire, l'un au moins de ces accès en écriture
- Dépendance dirigée selon l'ordre séquentiel : de $S(I)$ vers $T(J)$ si $S(I) <_{seq} T(J)$
  1. **Flot** l'emplacement mémoire commun est une écriture pour $S(I)$, une lecture pour $T(J)$, et n'est pas accédé en écriture entre $S(I)$ et $T(J)$, au sens de l'ordre séquentiel
  2. **Anti** l'emplacement mémoire commun est une lecture pour $S(I)$, une écriture pour $T(J)$, et n'est pas accédé en écriture entre $S(I)$ et $T(J)$
  3. **Sortie** $S(I)$ et $T(J)$ sont deux accès en écriture consécutifs à un même emplacement mémoire.
- On note $S(I) \Rightarrow T(J)$ une dépendance de $S(I)$ vers $T(J)$
- Tout ordre d'exécution qui respecte l'ordre partiel $\Rightarrow$ conduit à un code sémantiquement équivalent
- Détection du parallélisme : déterminer de tels ordres d'exécution, qui révèlent un parallélisme maximal.

Calcul (approché) des dépendances

$$S(I):\ a(f(I)) = \ldots$$

$$T(J):\ \ldots = a(g(J))$$

1. Accès commun :
$$f(I) = g(J)$$
2. Vecteurs d'itération valides :
$$I \in \mathcal{D}_S \quad \text{et} \quad J \in \mathcal{D}_T$$
3. Dépendance de flot (directe ou transitive) :
$$S(I) <_{seq} T(J)$$

- Fonction d'accès affines + domaines d'itération $\mathcal{D}_S$ et $\mathcal{D}_T$ polyèdres → techniques de programmation linéaire en nombres entiers
- Recherche de dépendances *directes* : trouver l'opération d'écriture (dans la case mémoire lue par $T(J)$) *la plus récente* → minima ou maxima lexicographiques dans des polyèdres.

Exemple

```
DO i=1, N
  DO j=1, N
    a(i+j) = a(i+j-1) + 1
  ENDDO
ENDDO
```

- **Recherche d'une dépendance de flot directe**
- Ecriture $S(i',j')$ sur $a(i'+j')$ et lecture $S(i,j)$ sur $a(i+j-1)$
- Equation $f(I) = g(J)$:
$$i' + j' = i + j - 1$$
- Equation $S(i',j') <_{seq} S(i,j)$:
$$(i' \leq i-1) \text{ ou } (i = i' \text{ et } j' \leq j-1)$$
- Résoudre le problème lexicographique
$$\max_{<_{seq}}\{(i',j') \mid (i',j') <_{seq} (i,j),\ i'+j' = i+j-1,\ 1 \leq i,i',j,j' \leq N\}$$
- Solution $(i, j-1)$ si $j \geq 2$ et $(i-1, j)$ si $j = 1$.
- Vecteurs de distance $(1,0)$ et $(0,1)$.

- **Anti-dépendance directe** de $S(i,j)$ vers $S(i',j')$
- Résoudre le problème lexicographique
$$\min_{<_{seq}}\{(i',j') \mid (i,j) <_{seq} (i',j'),\ i'+j' = i+j-1,\ 1 \leq i,i',j,j' \leq N\}$$
- Solution $(i+1, j-2)$ si $j \geq 3$ et $i \leq N-1$
- Vecteur de distance $(1,-2)$.

Graphe de dépendance étendu GDE

– Sommets $\{S_i(I) \mid 1 \leq i \leq s \text{ et } I \in \mathcal{D}_{S_i}\}$ (s nombre d'instructions)
– Une arête $S(I) \Rightarrow T(J))$ pour chaque dépendance
– Ensemble des paires de dépendance entre S et T :

$$\{(I, J) \mid S(I) \Rightarrow T(J)\} \subset \mathbb{Z}^{n_S} \times \mathbb{Z}^{n_T}$$

n_S (resp. n_T) nombre de boucles englobant S (resp. T)

– Ensemble de distance de ces dépendances

$$\{(\tilde{J} - \tilde{I}) \mid S(I) \Rightarrow T(J)\} \subset \mathbb{Z}^{n_{S,T}}$$

$n_{S,T}$ nombre de boucles englobant à la fois S et T

– **Une dépendance est toujours orientée selon l'ordre séquentiel**
– Vecteur de distance toujours lexico-positif
– Si $S(I) \Rightarrow T(J)$, alors $(\tilde{J} - \tilde{I}) \geq_{lex} \mathbf{0}$;
– De plus, si $S \geq_{text} T$, alors $(\tilde{J} - \tilde{I}) \neq \mathbf{0}$.

Graphe de dépendance réduit

– On ne peut pas calculer le GDE à la compilation :

1. Certains paramètres non instanciés
2. Calcul exact des accès mémoire impossible
3. Taille prohibitive

– *Graphe de dépendance réduit* GDR : sommets = instructions
– Arête $e = (S, T) : S \to T$ dans le GDR s'il existe au moins une paire (I, J) telle que $S(I) \Rightarrow T(J)$
– Chaque arête $e = (S, T)$ du GDR a une étiquette $w(e)$ qui décrit une sur-approximation des ensembles de distance

$$\text{Si } S(I) \Rightarrow T(J) \text{ (dans le GDE) alors}$$
$$\exists e = (S, T) \text{ (dans le GDR) telle que } (\tilde{J} - \tilde{I}) \in D_e$$

– Approximation des dépendances :

1. Niveaux de dépendance
2. Vecteurs de direction

Niveaux de dépendance

- Une dépendance entre deux instances $S(I)$ et $T(J)$ est
  - *boucle-indépendante* si elle a lieu au cours d'une même itération, i.e. si $\tilde{I} = \tilde{J}$
  - *portée par la boucle* si elle a lieu entre deux instances qui correspondent, pour au moins l'une des boucles englobantes, à des valeurs différentes du compteur de boucle.
- Une arête $e = (S, T)$ du GDR est étiquetée par la valeur du niveau $l(e) \in [1 \ldots n_{S,T}] \cup \{\infty\}$:
  - $l(e) = \infty$ si $S(I) \Rightarrow T(J)$ avec $\tilde{J} - \tilde{I} = \mathbf{0}$.
  - $l(e) \in [1 \ldots n_{S,T}]$ si $S(I) \Rightarrow T(J)$ et la première composante non nulle de $\tilde{J} - \tilde{I}$ est la $l(e)$-ème composante.

- *Graphe de dépendance réduit par niveaux*, ou GDRN
- $l(e) \leq n_{S,T} \rightarrow$ dépendance portée par la boucle de niveau $l(e)$
- $l(e) = \infty \rightarrow$ dépendance boucle-indépendante.

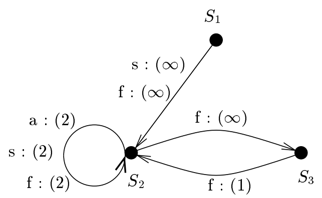
Vecteurs de direction

```
DO i=1, N
   DO j=1, N
     a(i+j) = a(i+j-1) + 1
   ENDDO
ENDDO
```

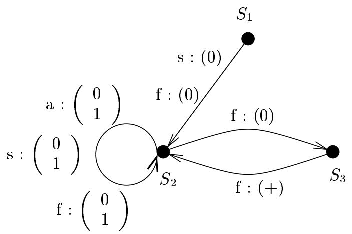
- Trois paires de références au tableau a induisent une dépendance :
  1. *lecture de* $a(i+j-1)$-*écriture de* $a(i+j)$, ensemble de distance $\{(1, -2)\}$
  2. *écriture de* $a(i+j)$-*lecture de* $a(i+j-1)$, ensemble de distance $\{(1, 0), (0, 1)\}$
  3. *écriture de* $a(i+j)$-*écriture de* $a(i+j)$, ensemble de distance $\{(1, -1)\}$
- Dépendance **uniforme** : ensemble des vecteurs de distance fini (dont la taille ne dépend pas des paramètres du domaine)
- Sinon, représentation par un vecteur de dimension $n_{S,T}$, appelé *vecteur de direction*
- Composantes prises dans $\mathbb{Z} \cup \{*, +, -\} \cup (\mathbb{Z} \times \{+, -\})$
- i-ème composante = approximation des i-èmes composantes de tous les vecteurs de distance possibles :
  $z+$ (resp. $z-$) si toutes les i-èmes composantes sont plus grandes (resp. plus petites), au sens large, que $z \in \mathbb{Z}$
  $*$ si les i-èmes composantes peuvent prendre n'importe quelle valeur ;
  z si la dépendance est uniforme dans cette dimension, avec pour unique valeur $z \in \mathbb{Z}$
- On écrit $+$ (resp. $-$) plutôt que $1+$ (resp. $(-1)-$).

Exemple (nid de boucles de Petersen)

DO i=2, N
  S_1 : s(i) = 0
  DO j=1, i-1
    S_2 : s(i) = s(i) + a(j,i) $*$ b(j)
  ENDDO
  S_3 : b(i) = b(i) - s(i)
ENDDO



GDRN (niveaux)

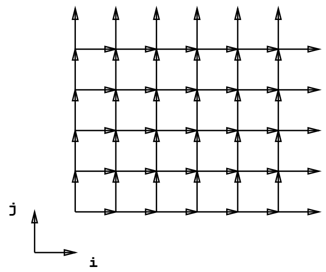


GDR avec vecteurs de direction

Limitations des abstractions de dépendance

– Noyau SOR :

DO i=1, N
  DO j=1, N
    a(i,j) = a(i,j-1) + a(i-1,j)
  ENDDO
ENDDO



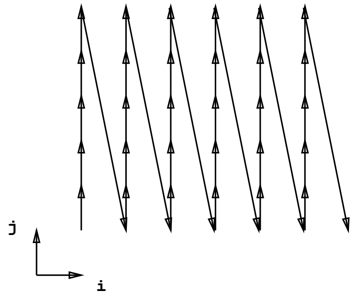
GDE du noyau SOR

GDRN et GDR avec vecteurs de direction pour le noyau SOR

– Les niveaux de dépendance 1 et 2 sont équivalents aux vecteurs de direction $(+,*)$ et $(0,+)$

```
DO i=1, N
  DO j=1, N
    a(i,j) = a(i,j-1) + a(i-1,N)
  ENDDO
ENDDO
```

– Même GDRN que le noyau SOR
– Le GDE contient un chemin de dépendance de longueur N^2



– Rien ne sert d'utiliser une détection de parallélisme trop sophistiquée si on l'applique sur une approximation rudimentaire !
– Niveaux de dépendance $\rightarrow$ abstraction adaptée à l'algorithme d'Allen et Kennedy
– Vecteurs de direction $\rightarrow$ bon outil pour les transformations unimodulaires de Lamport.

2 Algorithme d'Allen et Kennedy

Principe
– Abstraction = niveaux de dépendance
– Code initial = nid de boucles quelconque, parfait ou non
– Imbrication globale des boucles modifiée par l'algorithme
– Ojectif : pour chaque instruction, détecter le nombre maximal de boucles parallèles qui l'englobent

Distribution de boucle

Proposition Toute instance d'une instruction (ou bloc d'instructions) S peut être exécutée avant toute instance d'une instruction T si on n'a jamais $T(J) \Rightarrow S(I)$, i.e. par définition de $T \rightarrow S$, s'il n'y a pas de dépendance de T vers S.

DO i=..., ...	DO i=..., ...	DO i=..., ...	DO i=..., ...
S_1	S_1	S_2	S_2
S_2	ENDDO	ENDDO	S_1
ENDDO	DO i=..., ...	DO i=..., ...	ENDDO
	S_2	S_1	
	ENDDO	ENDDO	
(a)	(b)	(c)	(d)

- $S_1 \to S_2$ mais il n'y a pas de dépendance de S_2 vers S_1. La distribution de boucle conduit au code (b)
- $S_2 \to S_1$ mais il n'y a pas de dépendance de S_1 vers S_2. La distribution de boucle conduit au code (c)
- Il n'y a pas de dépendance entre S_1 et S_2. Les quatre codes sont valides
- $S_1 \to S_2$ et $S_2 \to S_1$. La distribution de boucle n'est pas valide. Le code initial (a) est valide, bien sûr, de même que le code (d) si toutes les dépendances sont portées par la boucle.

Distribution de boucle maximale

1. Calcul des composantes fortement connexes (CFC) du GDR
2. Tri topologique des CFC : numérotation des CFC de telle sorte que pour toute arête $S \to T$, S appartient à une CFC C_i et T à une CFC C_j avec $i \leq j$.
3. Distribution des boucles autour des CFC : on duplique les structures de boucles autour de chaque CFC. Les instructions d'une même CFC sont conservées dans le même ordre textuel, et les codes correspondant aux différentes CFC sont ordonnés selon l'ordre topologique.

Algorithme générique

- Marquer "parallèles" autant de boucles que possible pour chaque instruction
- Boucle **parallèle** : toutes les itérations de la boucle peuvent être exécutées simultanément et dans n'importe quel ordre, i.e. s'il n'y a pas de dépendance portée par la boucle
- Une boucle de profondeur k est parallèle s'il n'y a pas de dépendance de niveau k dans le GDRN du code englobé par la boucle
- Notation **DOPAR** et **DOSEQ**
- **DOPAR** n'est pas la boucle FORALL mais la boucle INDEPENDENT de High Performance Fortran (HPF).

- Idée : utiliser la distribution de boucle pour diminuer le nombre d'instructions à l'intérieur d'une boucle, et partant réduire le nombre de dépendances potentielles
- Après distribution, la détection du parallélisme sera menée indépendamment dans chaque CFC du GDRN
- La boucle la plus externe sera marquée **DOPAR** dans certaines composantes et **DOSEQ** dans d'autres
- L'algorithme se poursuit récursivement.

Algorithme

- n_S nombre de boucles englobant une instruction S
- GDRN $G \to l_{\min}(G)$ niveau minimal d'une arête de G :

$$l_{\min}(G) = \min\{l(e) \mid e \in G\}$$

- Appel initial : $k = 1$, G GDRN du code à paralléliser.

ALLEN-KENNEDY(G, k)

1. Supprimer dans G toutes les arêtes de niveau $< k$.
2. Calculer les composantes fortement connexes (CFC) G.
3. **Pour chaque** CFC C dans l'ordre topologique **faire**

   Si C est réduit à une seule instruction S, sans arête,

   Alors

   Générer des boucles **DOPAR** dans toutes les dimensions restantes, i.e. niveaux k à n_S, et générer le code pour S.

   Sinon

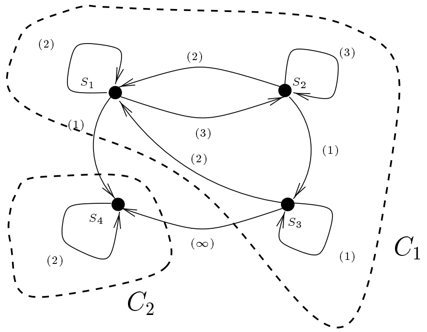
   (a) Soit $l = l_{\min}(C)$.

   (b) Générer des boucles **DOPAR** boucles du niveau k au niveau $l - 1$, et une boucle **DOSEQ** pour le niveau l.

   (c) Appeler ALLEN-KENNEDY(C, $l + 1$).

Exemple complet

```
DO i=..., ...
  DO j=..., ...
    DO k=..., ...
      S1
      S2
    ENDDO
    S3
    S4
  ENDDO
ENDDO
```



(a) Pseudo-code (b) GDRN correspondant

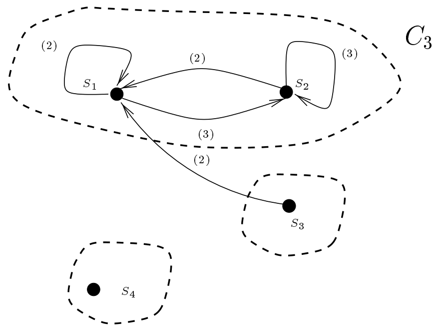
```
DOSEQ i=..., ...

  ALLEN-KENNEDY(C1, 2)

ENDDOSEQ
DOPAR i=..., ...
  DOSEQ j=..., ...

    ALLEN-KENNEDY(C2, 3)

  ENDDOSEQ
ENDDOPAR
```

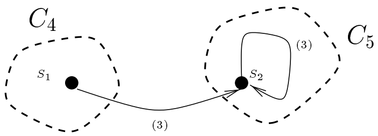


Code et GDRN avant le deuxième appel récursif

```
DOSEQ i=..., ...
  DOPAR j=..., ...
    S3
  ENDDOPAR
  DOSEQ j=..., ...

    ALLEN-KENNEDY(C3, 3)

  ENDDOSEQ
ENDDOSEQ
DOPAR i=..., ...
  DOSEQ j=..., ...
    S4
  ENDDOSEQ
ENDDOPAR
```



Code et GDRN avant le troisième appel récursif

```
DOSEQ i=..., ...
  DOPAR j=..., ...
    S3
  ENDDOPAR
  DOSEQ j=..., ...
    DOPAR k=..., ...
      S1
    ENDDOPAR
    DOSEQ k=..., ...
      ALLEN-KENNEDY(C4, 4)
    ENDDOSEQ
  ENDDOSEQ
ENDDOSEQ
DOPAR i=..., ...
  DOSEQ j=..., ...
    S4
  ENDDOSEQ
ENDDOPAR
```

```
DOSEQ i=..., ...
  DOPAR j=..., ...
    S3
  ENDDOPAR
  DOSEQ j=..., ...
    DOPAR k=..., ...
      S1
    ENDDOPAR
    DOSEQ k=..., ...
      S2
    ENDDOSEQ
  ENDDOSEQ
ENDDOSEQ
DOPAR i=..., ...
  DOSEQ j=..., ...
    S4
  ENDDOSEQ
ENDDOPAR
```

Code avant et après le dernier appel récursif

3 Transformations unimodulaires

Définition et validité

- Nids parfaits (toutes les instructions sont englobées par n boucles)
- Dépendances uniformes
- Nouvel ordre d'énumération des points du domaine d'itération
- Nouveau vecteur d'itération $I' = TI$, T matrice unimodulaire
- Matrice unimodulaire : matrice entière carrée de déterminant ± 1
- Corps du nid de boucles atomique, GDR réduit à un seul sommet
- Dépendances boucle-indépendantes toujours préservées
- Dépendances portées par certaines boucles $\rightarrow$ conditions de validité.

Matrices unimodulaires

- I_n matrice identité de taille n
- $E_{i,j}$ matrice carrée de taille n : coefficients tous nuls sauf $e_{i,j} = 1$
- Trois types de matrices unimodulaires *élémentaires* :
  1. $R_i = I_n - 2E_{i,i}$, égale à son inverse
  2. $P_{i,j} = I_n - E_{i,i} - E_{j,j} + E_{i,j} + E_{j,i}$, égale à son inverse
  3. Pour $i \neq j$, $S_{i,j}(k) = I_n + kE_{i,j}$, d'inverse $S_{i,j}(-k)$
- Toute matrice unimodulaire peut être décomposée en produit de matrices unimodulaires élémentaires.

Transformations de boucles classiques

– Avant transformation le domaine d'itération est scanné par le vecteur d'itération

$$(I_1, \ldots, I_n)$$

– *Permutation de boucle.* Après transformation par la permutation σ, nouveau vecteur d'itération

$$(I_{\sigma(1)}, \ldots, I_{\sigma(n)})$$

Echange de la i-ème et de la j-ème boucle : matrice $P_{i,j}$

– *Inversion de boucle.* Inverser la i-ème boucle revient à la parcourir en sens inverse. Nouveau vecteur d'itération

$$(I_1, \ldots, I_{i-1}, -I_i, I_{i+1}, \ldots, I_n)$$

Transformation correspondante R_i

– *Torsion de boucle.* Tordre la boucle I_j d'un facteur (entier) k à l'aide la boucle I_i (avec $i < j$) $\rightarrow$ nouveau vecteur d'itération

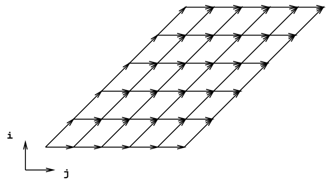
$$(I_1, \ldots, I_{j-1}, I_j + kI_i, I_{j+1}, \ldots, I_n)$$

Transformation correspondante $S_{i,j}(k)$.

Noyau SOR

```
DO i=1, N
   DO j=1, N
     a(i,j) = a(i,j-1) + a(i-1,j)
   ENDDO
ENDDO
```


```
DO j=2, 2N
   DOPAR i=max(1,j-N), min(N,j-1)
     a(i,j-i) = a(i-1,j-i) + a(i,j-i-1)
   ENDDOPAR
ENDDO
```



Code parallèle et GDE (avec $N = 6$) pour le noyau SOR

Ré-écriture du nid

(a)	(b)	(c)

DO $I \in \mathcal{D}$, I entier
  $S(I)$
ENDDO
	DO $I'=T(I)$, $I \in \mathcal{D}$, I entier
	  $S(I)$
	ENDDO
		DO $I' \in T(\mathcal{D})$, I' entier
		  $S(T^{-1}(I'))$
		ENDDO

Code avant et après une transformation unimodulaire T

Validité

- $S_i(I) \Rightarrow S_j(J)$ dans le code initial $\Rightarrow S_i(I) <_{seq} S_j(J)$ dans le code transformé
- $S_i(I)$ calculée à l'itération $T(I)$ et $S_j(J)$ à l'itération $T(J)$
- Les dépendances boucle-indépendantes ($I = J$ and $S_i <_{text} S_j$) sont préservées automatiquement
- Pour une dépendance de vecteur de distance non nul, garantir $T(I) <_{lex} T(J)$, i.e. $T(J - I) >_{lex} \mathbf{0}$

Proposition T est valide ssi $d \neq \mathbf{0} \Rightarrow T(d) >_{lex} \mathbf{0}$.

La méthode de l'hyperplan

- Nid de boucles à dépendances uniformes
- D ensemble des m vecteurs de dépendance
- Chacun vecteur de dépendance est lexicographiquement positif et non nul
- On veut T t.q. $d \in D \Rightarrow T(d) >_{lex} \mathbf{0}$
- Idée : transformer tous les vecteurs de dépendance de manière à ce que leur *première* composante soit positive
  $\rightarrow$ toutes les dépendances de niveau 1, portées par la boucle la plus externe
  $\rightarrow$ $n-1$ boucles parallèles englobées par une boucle séquentielle
  $\rightarrow$ *méthode de l'hyperplan* de Lamport
- Soit s la première ligne de T, on veut $sd \geq 1$ pour tout vecteur de dépendance d
- Toutes les instances d'instruction $S(I)$ telles que $sI = cte$ peuvent être exécutées en parallèle
- Hyperplan affine (ou front d'onde) normal à s
- Le flot des calculs progresse selon la direction s
- Le vecteur s est appelé *vecteur de temps*.

Théorème Etant donné un ensemble D de m vecteurs lexicographiquement (strictement) positifs, il existe un vecteur entier s tel que $sd \geq 1$ pour tout $d \in D$.

Exemple

```
DO i=1, N
   DO j=1, N
     a(i+j) = a(i+j-1) + 1
   ENDDO
ENDDO
```

– Quatre vecteurs de dépendance uniformes :

$$\{(1,-1),(1,-2),(1,0),(0,1)\}$$

– Vecteur de temps $s=(3,1)$

– Forme normale de Hermite :

$$s=(3,1)=(1,0)\begin{pmatrix}3&1\\1&0\end{pmatrix}\text{ et }T=\begin{pmatrix}3&1\\1&0\end{pmatrix}=\begin{pmatrix}0&1\\1&0\end{pmatrix}\begin{pmatrix}1&0\\3&1\end{pmatrix}$$

– T correspond à une torsion de boucle de facteur 3, suivi d'un échange de boucle

– Code après transformation :

DO j=4, 4N
   DOPAR i= max(1, $\lceil\frac{j-N}{3}\rceil$), min(N, $\lfloor\frac{j-1}{3}\rfloor$)
     a(j-2i) = a(j-2i-1)+1
   ENDDOPAR
ENDDO

Recherche du vecteur de temps optimal

– Nid de boucles parfait de profondeur n, bloc d'instructions atomique

– m vecteurs de dépendance uniformes (entiers et lexicographiquement strictement positifs)

– $D=(d_1,\ldots,d_m)$ *matrice de dépendance*

– Domaine d'itération : ensemble des points entiers du polyèdre

$$J=J(A,b)=\{x\in\mathbb{Z}^n \mid Ax\le b\}$$

A matrice de taille $l\times n$, b vecteur à l composantes

– Vecteurs de temps à composantes rationnelles : pour $s\in\mathbb{Q}^n$,

$$T_s(p)=\lfloor sp\rfloor+c_s \text{ où } c_s=-\min_{p\in J}\lfloor sp\rfloor$$

– Ordonnancement valide si $s\in S(D)=\{s\in\mathbb{Q}^n \mid sD\ge \mathbf{1}\}$

– Latence de l'ordonnancement

$$\begin{aligned}L_s &= 1+\max\{T_s(p)\mid p\in J(A,b)\}-\min\{T_s(p)\mid p\in J(A,b)\}\\ L_s &= 1+\max\{\lfloor sp\rfloor-\lfloor sq\rfloor \mid p,q\in J(A,b)\}\end{aligned}$$

– Latence minimale $L_{\min}=\min\{L_s \mid s\in S(D)\}$.

Programmation linéaire (en rationnels)

- $L_s^* = \max\{s(p-q) \mid p, q \in J(A,b)\}$
- $L_{\min}^* = \min\{L_s^* \mid s \in S(D)\}$
- $L_{\min} \leq L_{\min}^* + 2$
- Chercher $s \in S(D)$ qui minimise L_s^*, avec

$$L_s^* = \max\{s(p-q) \mid Ap \leq b,\ Aq \leq b\}$$

- $L_{\min}^*$ solution d'un problème de min-max :

$$L_{\min}^* = \min\{\max\{s(p-q) \mid Ap \leq b,\ Aq \leq b\} \mid sD \geq \mathbf{1}\}$$

- Théorème de dualité de la programmation linéaire :

$$\begin{aligned} L_s^* &= \max\{s(p-q) \mid Ap \leq b,\ Aq \leq b\} \\ L_s^* &= \min\{(s_1+s_2)b \mid s_1 \geq \mathbf{0},\ s_2 \geq \mathbf{0},\ s_1 A = s,\ s_2 A = -s\} \end{aligned}$$

- Problème d'optimisation :

$$L_{\min}^* = \min\{(s_1+s_2)b \mid s_1 \geq \mathbf{0},\ s_2 \geq \mathbf{0},\ s_1 A = s,\ s_2 A = -s,\ sD \geq \mathbf{1}\}$$

- Problème linéaire en b

Exemple de Peir et Cytron

```
DO i = 0, n
  DO j = 0, n
    a(i,j) = b(i,j-6) + d(i-1,j+3)
    b(i+1,j-1) = c(i+2,j+5)
    c(i+3,j-1) = a(i,j-2)
    d(i,j-1) = a(i,j-1)
  ENDDO
ENDDO
```

- Aucune des deux boucles ne peut être marquée comme parallèle
- Il y a du parallélisme ...bien caché dans ce code
- Domaine d'itération $J = J(A, Nb)$ avec

$$A = \begin{pmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{pmatrix} \quad b = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

- Matrice de dépendance

$$D == \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 2 & -6 & 5 & 1 & -4 \end{pmatrix}.$$

– Soit $s = (a, b)$, $s_1 = (a_1, b_1, c_1, d_1)$ et $s_2 = (a_2, b_2, c_2, d_2)$.
– Problème d'optimisation :

$$\begin{cases} sD \geq \mathbf{1} : 2b \geq 1, a - 6b \geq 1, a + 5b \geq 1, b \geq 1, a - 4b \geq 1 \\ s_1 A = s : -a_1 + b_1 = a, -c_1 + d_1 = b \\ s_2 A = -s : -a_2 + b_2 = -a, -c_2 + d_2 = -b \\ s_1 \geq 0 : a_1 \geq 0, b_1 \geq 0, c_1 \geq 0, d_1 \geq 0 \\ s_2 \geq 0 : a_2 \geq 0, b_2 \geq 0, c_2 \geq 0, d_2 \geq 0 \\ \min (s_1 + s_2) b : b_1 + b_2 + d_1 + d_2 \end{cases}$$

– Méthode du simplexe de Maple
$\rightarrow$ $s = (7, 1)$, $s_1 = (0, 7, 0, 1)$ et $s_2 = (7, 0, 1, 0)$
$\rightarrow$ Latence optimale pour le domaine $J(A, Nb)$:

$$N \times (7 + 0 + 1 + 0) = 8N.$$

Pipeline logiciel

Plan

1. Formulation du problème
2. Rappels sur l'ordonnancement de graphes de tâches
3. Résolution avec infinité de ressources

4 Formulation du problème

Software pipelining

– Ordonnancer un ensemble de tâches génériques
– Exécution un grand nombre de fois (coeur d'une boucle externe)
– Ré-ordonner pour utiliser au mieux les ressources matérielles.

Exemple de référence

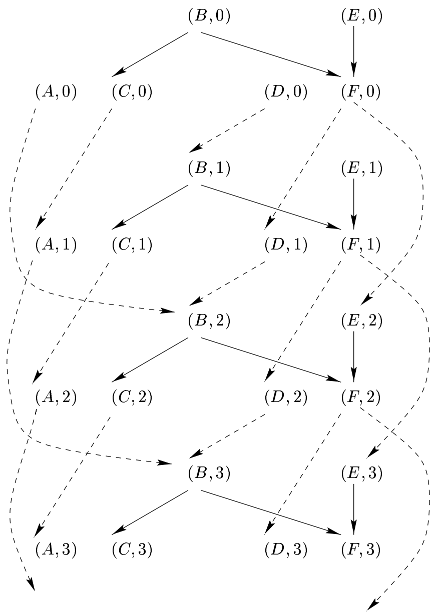
```
DO k=0, N-1
    (A) : a(k) = c(k-1)
    (B) : b(k) = a(k-2) * d(k-1)
    (C) : c(k) = b(k) + 1
    (D) : d(k) = f(k-1)/3
    (E) : e(k) = sin(f(k-2))
    (F) : f(k) = log(b(k) + e(k))
ENDDO
```

– 6 instructions A, B, C, D, E, et $F \rightarrow$ tâches génériques
– N itérations, N paramètre de valeur inconnue
– (v, k) instance de la tâche générique v à l'itération k, délai $d(v)$.

Contraintes de précédence

– GDE de sommets $\{(v, k) \mid v \in V,\ 0 \leq k < N\}$
– $V = \{A, B, C, D, E, F\}$
– Structure régulière des dépendances

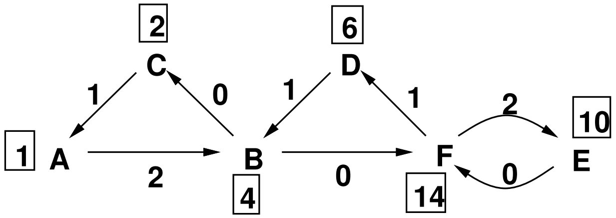
1. de niveau ∞ (trait plein)
2. de niveau 1 (pointillé).



Graphe de dépendance réduit

- $G = (V, E, d, w)$ GDR
- $d : V \to \mathbb{N}^*$: délai de chaque tâche générique
- $w : E \to \mathbb{N}$: distance de dépendance de chaque arête
- Ordonnancement : $\sigma : V \times \mathbb{N} \to \mathbb{N}$
- Contraintes de dépendance :

$$\forall e = (u, v) \in E,\ \forall k \geq 0,\ \sigma(v, k + w(e)) \geq \sigma(u, k) + d(u)$$

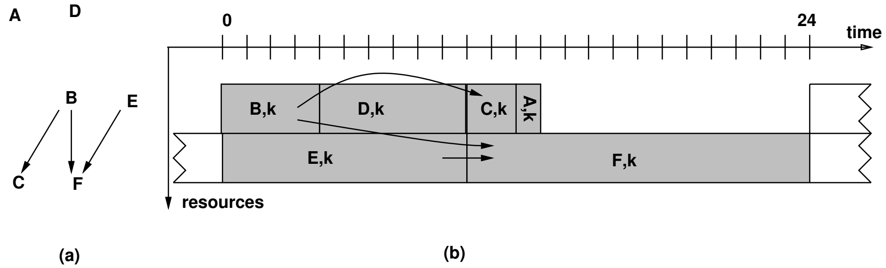


Ordonnancements cycliques

- $\sigma(v, k) = c_v + \lambda k$ avec $c_v \in \mathbb{N}$ et $\lambda \in \mathbb{N}$
- σ ordonnancement périodique, de période λ
- σ ordonnance des "tranches" de longueur λ
- A l'intérieur de chaque tranche, une et une seule instance de chaque tâche générique est initiée.

Retour à l'exemple de référence

- $A(G)$ sous-graphe du GDR avec seulement les arêtes de poids nul
- $A(G)$ caractérise toutes les dépendances de niveau ∞
- $A(G)$ est acyclique
- Ordonnancement de liste pour $A(G)$
- Chemin critique σ_a avec 2 processeurs

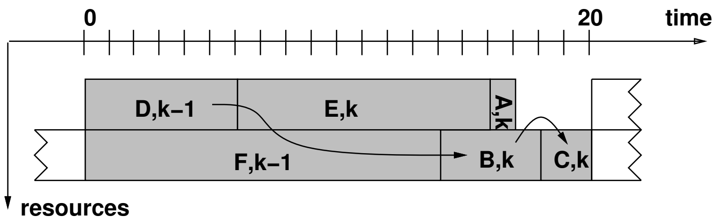


- $\lambda = 24$: latence de l'ordonnancement de liste σ_a
- $\sigma(v, k) = \sigma_a(v) + \lambda k$
- Contraintes de dépendance satisfaites :
  1. de niveau ∞ : imposées par σ_a ;
  2. de niveau 1 : choix de λ

Parenthèse : meilleure solution

- $\sigma(v, k) = c_v + \lambda k$, où $\lambda = 20$

Tâche générique v	A	B	C	D	E	F
Constante c_v	16	14	18	20	6	20



- Validité de cet ordonnancement :
  - Vérification directe pour chacune des 8 arêtes de dépendance dans le GDR
  - Lire le poly :-)

Contraintes de ressources

- Chaque ressource ne peut exécuter qu'une opération à la fois
- Opérations atomiques (une fois initiée, ne peut être interrompue)
- Avec p ressources identiques :

$$\forall t \in \mathbb{N}, \ |\{(v, k) \mid t - d(v) < \sigma(v, k) \leq t\}| \leq p$$

- Cas général : ressources de différents types (additionneurs et multiplieurs), ou pipelinées (une opération peut être initiée à chaque top).

Bornes inférieures pour λ

- Indépendamment du nombre de ressources, $M \geq w(\mathcal{P})$ pour tout chemin $\mathcal{P}$ dans le graphe
- Si on dispose de p ressources, $M \geq \frac{\text{Seq}}{p}$, où Seq est le temps séquentiel, i.e. la somme des durées de toutes les tâches
- Pour un cycle $\mathcal{C}$, $\rho(\mathcal{C})$ est le *rapport* durée sur distance $\frac{d(\mathcal{C})}{w(\mathcal{C})}$.
- Pour tout ordonnancement σ de temps de cycle λ_σ :

$$\text{pour tout cycle } \mathcal{C}, \ \lambda_\sigma \geq \rho(\mathcal{C}) \qquad (1)$$

$$\text{si } p \text{ ressources sont disponibles, } p\lambda_\sigma \geq \sum_{v \in V} d(v) \qquad (2)$$

5 Rappels sur l'ordonnancement de graphes de tâches

Graphes de tâches

- Ordonnancement d'un système de tâches $G = (V, E, w)$: fonction $\sigma : V \longrightarrow \mathbb{N}^*$ avec $\sigma(u) + w(u) \leq \sigma(v)$ pour toute arête $e = (u, v) \in E$.
- Soit $G = (V, E, w)$ un système de tâches. Il existe un ordonnancement ssi G est sans cycle
- Avec p processeurs $\rightarrow$ contraintes de ressource : fonction d'allocation alloc : $V \longrightarrow \mathcal{P}$, où $\mathcal{P} = \{1, \ldots, p\}$ avec

$$\mathrm{alloc}(T) = \mathrm{alloc}(T') \Rightarrow \begin{cases} \quad \sigma(T) + w(T) \leq \sigma(T') \\ ou \;\; \sigma(T') + w(T') \leq \sigma(T) \end{cases}$$

- Latence $\mathrm{MS}(\sigma, p)$: temps d'exécution total

$$\mathrm{MS}(\sigma, p) = \max_{v \in V}\{\sigma(v) + w(v)\} - \min_{v \in V}\{\sigma(v)\}$$

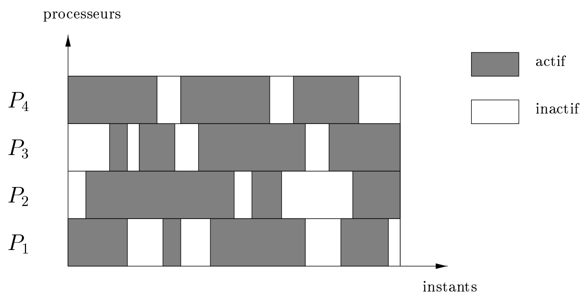
- $\mathrm{Pb}(\infty)$: déterminer un ordonnancement de latence minimale avec ressources illimitées
- $\mathrm{Pb}(p)$: déterminer un ordonnancement de latence minimale avec p processeurs.

Propriétés

- Soit $G = (V, E, w)$ un DAG et σ a ordonnancement pour G avec p processeurs. Alors $\mathrm{MS}(\sigma, p) \geq w(\Phi)$, pour tout chemin Φ de G.

- Efficacité $e(\sigma, p) == \frac{\mathrm{Seq}}{p \times \mathrm{MS}(\sigma,p)}$, où $\mathrm{Seq} = \sum_{v \in V} \; w(v)$
  Alors

$$0 \leq e(\sigma, p) \leq 1$$



Complexité de Pb(∞)

– Pour $v \in V$, PRED(v) prédécesseurs immédiats de v
– Pour $v \in V$, SUCC(v) successeurs immédiats de v
– $v \in V$ est un sommet d'entrée si PRED(v) $= \emptyset$.
– $v \in V$ est un sommet de sortie si SUCC(v) $= \emptyset$.
– Niveau supérieur $tl(v)$ (top level) : plus grand poids d'un chemin menant d'un sommet d'entrée à v, en excluant le poids de v.
– Niveau inférieur $bl(v)$ (bottom level) : plus grand poids d'un chemin menant de v à un sommet de sortie, incluant le poids de v

– Calcul en traversant le graphe :

$$tl(v) = 0 \text{ si } v \text{ sommet d'entrée}$$

$$tl(v) = \max\{tl(u) + w(u); u \in \text{PRED}(v)\} \text{ sinon}$$

– De même, $bl(v) = \max\{bl(u); u \in \text{SUCC}(v)\} + w(v)$ (et $bl(v) = w(v)$ pour un sommet de sortie v).

Proposition Soit $G = (V, E, w)$ un DAG. L'ordonnancement libre σ_{free} défini par

$$\sigma_{free}(v) = tl(v), \forall v \in V$$

est un ordonnancement optimal de G
$\rightarrow$ Pb(∞) peut être résolu en temps $O(|V| + |E|)$.

Résolution de Pb(p)

– NP-complétude de Pb(p)
– Heuristiques de liste : à chaque instant, on alloue autant de tâches que possible aux processeurs disponibles
– Toute stratégie *ne laissant pas délibérément un processeur inactif* est à un facteur deux de l'optimal.

Garantie de performance

– $G = (V, E, w)$ DAG, σ un ordonnancement pour G
– $v \in FREE(\sigma, t)$ (libre au temps t) si son exécution n'a pas encore commencé ($\sigma(v) \geq t$), mais tous ses prédécesseurs ont été exécutés au temps t ($\forall\, u \in \text{PRED}(v),\ \sigma(u) + w(u) \leq t$)
– Ordonnancement de liste : à tout instant t,
 si $|FREE(\sigma, t)| = r \geq 1$,
 si q processeurs sont disponibles,
 alors on débute l'exécution de $\min(r, q)$ tâches libres.

Théorème Soit $G = (V, E, w)$ un DAG avec p processeurs. Soit σ un ordonnancement de liste quelconque pour G. Soit $\text{MS}_{opt}(p)$ la latence d'ordonnancement optimal pour G. Alors

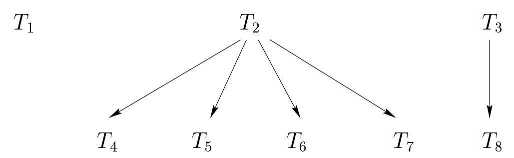
$$\text{MS}(\sigma, p) \leq (2 - \frac{1}{p})\text{MS}_{opt}(p)$$

Implémentation d'un ordonnancement de liste

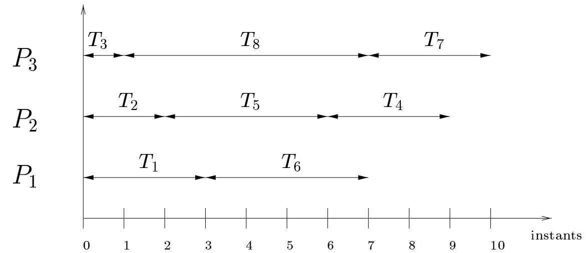
1. *Initialisation :*
   (a) Calculer le niveau de priorité de toutes les tâches.
   (b) Constituer la file de priorité comme étant la liste des tâches libres (tâches sans prédécesseurs) triées en ordre de priorité décroissante.
   (c) Soit t l'instant courant : $t = 0$.
2. *Tant qu'il reste des tâches à exécuter* :
   (a) Ajouter les nouvelles tâches libres, s'il y en a, à la file de priorité. Si l'exécution d'une tâche se termine à l'instant t, supprimer cette tâche de la liste des prédécesseurs de chacun de ses successeurs. Ajouter les tâches dont la liste des prédécesseurs est devenue vide à la file.
   (b) S'il y a q processeurs disponibles et r tâches dans la file, supprimer les $\min(q, r)$ premières tâches de la file et les exécuter ; si T est l'une de ces tâches, poser $\sigma_{cp}(T) = t$.
   (c) Incrémenter t.

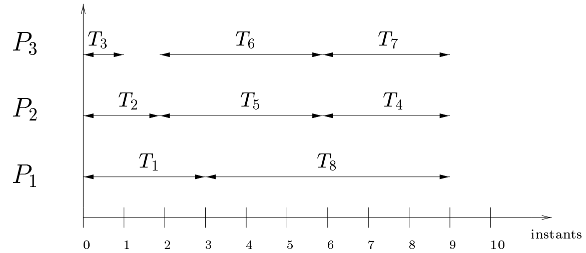
Chemin critique

- *Chemin critique* d'une tâche : son niveau inférieur
- Utilisé pour déterminer les niveaux de priorité entre tâches



Tâches	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8
Poids	3	2	1	3	4	4	3	6
Chemin critique	3	6	7	3	4	4	3	6





Ordonnancement optimal

6 Résolution avec infinité de ressources

Outils
– Graphes à potentiel
– Algorithme de Bellman-Ford
– Lire le poly !

Ordonnancement optimal à ressources illimitées
– Graphe $G = (V, E, d, w)$
– d définit les poids des sommets, w ceux des arêtes
– Ordonnancement $\sigma(v, k) = \lambda k + c(v)$
– On cherche λ minimal

– **Théorème**

$$\lambda_\infty = \max\{\rho(\mathcal{C}) \mid \mathcal{C} \text{ cycle élémentaire de } G\}$$

(et $\lambda_\infty = 0$ si G est acyclique)
– Constante $c(v)$: plus long chemin de s à v dans le graphe $G'_\lambda = (V', E', w'_\lambda)$:

1. Sommets de G'_λ : ajouter à V un nouveau sommet s :

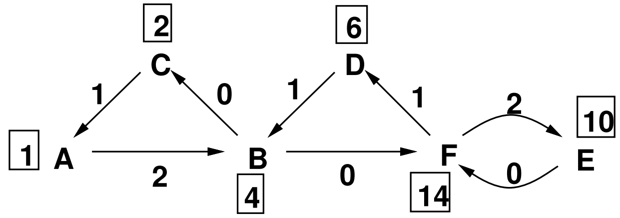
$$V' = V \cup \{s\}.$$

2. Arêtes de G'_λ : pour chaque sommet $v \in V$ ajouter à E une arête de s à v : $E' = E \cup (\{s\} \times V)$.
3. Poids des arêtes de G'_λ :

$$w'_\lambda(e) = 0 \text{ si } e = (u, v) \in E' \setminus E$$

$$w'_\lambda(e) = d(u) - \lambda w(e) \text{ sinon.}$$

Retour à l'exemple



– Trois cycles élémentaires

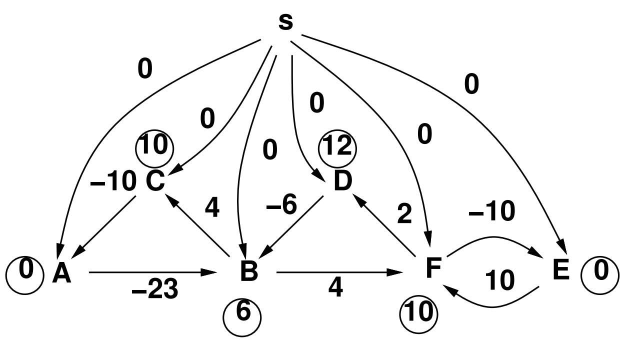
– $\rho(C_1) = \frac{7}{3}$ pour $C_1 : A \to B \to C \to A$

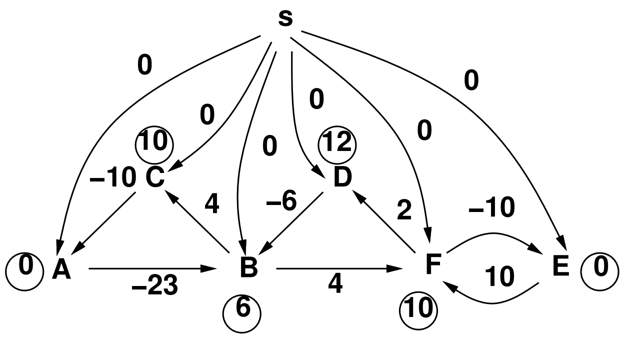
– $\rho(C_2) = \frac{24}{2}$ pour $C_2 : B \to F \to D \to B$

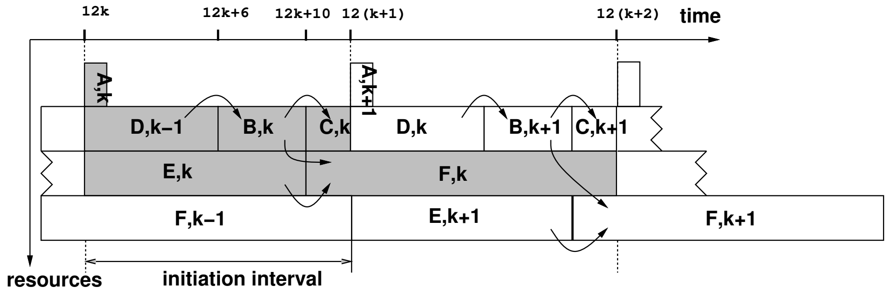
– $\rho(C_3) = \frac{24}{2}$ pour $C_3 : E \to F \to E$

– Donc $\lambda_\infty = 12$

Calcul des plus longs chemins dans le graphe G'_{12} :







Quatre processeurs sont nécessaires :

Instant de calcul	λk	$\lambda k{+}6$	$\lambda k{+}10$	$\lambda k{+}12$	$\lambda k{+}18$	$\lambda k{+}22$
Processeur P_1	(A, k)			$(A, k+1)$		
Processeur P_2	$(D, k{-}1)$	(B, k)	(C, k)	(D, k)	$(B, k{+}1)$	$(C, k{+}1)$
Processeur P_3	(E, k)		(F, k)			
Processeur P_4				$(E, k{+}1)$		$(F, k{+}1)$