# Analyse, optimisation, compilation

Vincent Loechner

2 octobre 2008

# Chapitre 1

# Architecture d'un compilateur

# Chapitre 2

# Analyse de dépendance

## 2.1 Modèle géométrique

### 2.1.1 Distance de dépendance

$\vec{d} = \vec{i}^T - \vec{i}^S$

### 2.1.2 Vecteur direction

```
1 for (i = 0; i < 10; i++) {
2     A[2*i] = B[i];
3     C[i] = A[i];
4 }
```

$S_2\ \delta^f_{<=}\ S_3$

[[schéma 1]]

On va se servir de cette information : $\vec{i}^S \leq \vec{i}^T$

Vecteurs direction : $<$, $>$, $=$, $\leq$, $\geq$, $\neq$, $*$ (inconnu)

Exemple (suite) :
$d = (0, 1)$
$S_3\ \delta^f_{(0,1)}\ S3$
$\rightarrow S_3\ \delta^f_{(=,<)}\ S3$

### 2.1.3 Boucles non parfaitement imbriquées

Dans ce cas, il y a un travail supplémentaire à effectuer.

Exemple :

```
1 for (i = 0; i < N; i++) {
2     B[i] /= A[i][i];
3     for (j = i + 1; j < N; j++)
4     B[j] -= A[i][j] * B[i];
5 }
```

[[schéma 2]]

$S_2\ \delta^f\ S_4$, mais on ne peut pas exprimer le vecteur de dépendance (pour l'instant).
$S_4\ \delta^f_{(1,0)}\ S_4$
$S_4\ \delta^f\ S_2$

$\Rightarrow$ Technique du « code sinking » :

```
-1 for (i = 0; i < N; i++)
 0     for (j = i; j < N; j++) {
 1         if (i == j)
 2             B[i] /= A[i][i];
 3         else /* if i >= j + 1 */
 4             B[j]  -= A[i][j] * B[i];
 5     }
```

[[schéma 3]]

$S_4\ \delta^f_{(1,0)}\ S_4$
$S_2\ \delta^f_{(=,<)}\ S_4$
$S_4\ \delta^f_{(1,0)}\ S_2$ :
$(i_0, i_0 + 1 = j_0) \rightarrow$ écriture$(B[i_0 + 1])$
$(i_0 + 1, i_0 + 1) \rightarrow$ lecture$(B[i_0 + 1)$

## 2.2 Tableaux et dépendances

On considère que les boucles sont "affines" (indices, bornes sont des fonctions affines de variables).
On considère que les accès aux tableaus sont des fonctions affines également :
$T[f(vec(i))]$
$f(vec(i)) = M \cdot vec(i) + vec(b)$

On ne saurait pas traiter des cas tels que $A[\sqrt(i)]$ ou $A[i^2]$. Mais ces cas restent rares.

### 2.2.1 Construction du système de dépendance

Inconnues : variables d'itération (ainsi que les paramètres, tels que les « N » dans les exemples précédents).

$*S_4\ \delta\ S_4$

Affecte $B[j^d]$ à une certaine itération $\begin{pmatrix} i^d \\ j^d \end{pmatrix}$.

Utilise $B[j^u]$ à $\begin{pmatrix} i^u \\ j^u \end{pmatrix}$.

On cherche le minimum en $\begin{pmatrix} i^d \\ j^d \end{pmatrix}$ de ce système.

[[schéma 4]]

### 2.2.2 Résolution du système de dépendance

$\rightarrow$ En nombres entiers.
$\rightarrow$ Égalités et innégalités.

Techniques de simplification ; par exemple, le GCD Test.

**GCD Test**

Exemple :

```
1 for (i = 0; i < N; i++)
2     for (j = 0; j < 8; i++) {
3         A[i + 2 * j] = ...;
4         ... = A[i + 4 * j - 1];
5     }
```

$S_3$ $delta^f_{(=,*)}$ $S_4$ ? (Existe-t-il une dépendance portée par la bouche j ?)

$i_d + 2 * j_d = i_u + 4 * j_u - 1$

Le GCD test consiste à prendre le PGCD de tous les coefficients des indices. Si le GCD divise la constante, alors il peut y avoir une solution.

$gcd(1, 2, -1, -4) = 1$, qui divise $-1$.

$\rightarrow$ Il peut exister une dépendance telle que $S_3$ $delta^f_{(*,*)}$ $S_4$.

Si $i_d = i_u$, $2j_d - 4j_u = -1$
$gcd(2, -4) = 2 \rightarrow$ pas de solution.
$\Rightarrow$ Il n'existe pas de dépendance telle que $S_3$ $delta^f_{(=,*)}$ $S_4$.

**Extreme value test**

```
for (i = 0; i < N; i++) {
    A[i] = ...;
    ... = A[i + 2 * N];
}
```

Il n'y a pas d'intersection entre les deux ensembles de valeurs d'indice des tableaux, donc il ne peut pas y avoir de dépendance.

Ces tests permettent d'éliminer rapidement des systèmes d'équations parfois complexes sans les résoudre entièrement.

**Solveur PLNE (problèmes linéaires en nombre entiers)**

Omega Test, Maple, etc.

### 2.2.3 Single assignment

Programmes à assignation unique.

```
for (i = 0; i < N; i++)
    B[i] = B[i - 1] / 2;
```

$\rightarrow$ Pas de dépendance.

```
1 for (i = 0; i < N; i++)
2     B = B / 2;
```

⇒ On élimine les dépendances de sortie. Il ne reste plus qu'une dépendance « écriture suivie de lecture ».
$S_2\ \delta_1^f\ S_2$
$S_2\ \delta_0^f\ S_2$

L'assignation unique consiste à ne pas réutiliser de variable autrement que par des dépendances de flot ($\delta^f$).

## 2.3 Remarques

### 2.3.1 Tests

```
if (c)
    A = B;
else
    B = C;
```

### 2.3.2 Pointeurs

Utilisés comme des tableaux → OK.
Arithmétique complexe → Pas OK !

### 2.3.3 Analyse interprocédurale

```
for (i = 0; i < N; i--)
    x = fonction(T, i);
```

→ Inline.

# Chapitre 3

# Transformations de boucles

Validité : Si $S_1\ \delta_v\ S_2 \Rightarrow T(S_1)\ \delta_{T(v)}\ T(S_2)$
→ Respecte la sémantique du programme.

## 3.1 Les transformations

### 3.1.1 Transformations simples

#### Réordonnement d'instructions

$$\begin{array}{cc} (S_1) & \\ (S_2) & \end{array} \rightarrow \begin{array}{c} (S_2) \\ (S_1) \end{array}$$

**Validité :** si $\nexists\ S_1\ \delta^*\ S_2$.
**Utilité :** instructions assembleur (pipeline, lantence mémoire, parallélisme).

#### Exemple :

```
 1 A[0] = 0;
 2 B[0] = 0;
 3 if (C > 0) {
 4     A[1] = 1;
 5     B[1] = 9;
 6 }
 7 for (i = 2; i < 100; i++) {
 8     A[i] = A[i - 2] + 2 * A[i - 1];
 9     B[i] = B[i - 1] + 2 * B[i - 2];
10 }
```

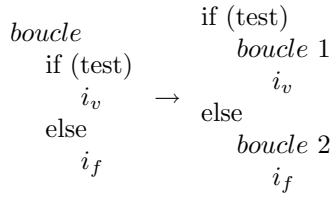
[[schéma 5]]

A : 1, 3, 4, 7, 8 ; B : 2, 3, 5, 7, 9.

Ordre quelconque ou même parallélisme entre A et B.

#### Onswitching

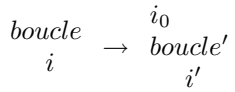
Sort un test de l'intérieur d'une boucle.




**Validité :** test indépendant de la boucle.
**Utilité :** réduit la fréquence du test.

#### Peeling (pelage)

Sort la première ou la dernière itération d'une boucle.




**Utilité :** sortir le code invariant de $i'$ et permettre d'autres transformations.
**Validité :** la boucle doit avoir plus d'une itération.
**Sinon :** il faut protéger $i_0$ avec un test.

#### Exemple :

```
for (i = 0; i < N; i++)
    A[i] = B[i] + X * Y;
```

devient :

```
T = X * Y;
A[0] = B[0] + T;
for (i = 1; i < N; i++)
    A[i] = B[i] + T;
```

#### Index set splitting

```
for (i = inf; i < sup; i++)
    ...

    ↓

for (i = inf; i < min(b, sup); i++)
    ...
for (i = max(b, inf); i < sup; i++)
    ...
```

**Validite :** $inf \leq b \leq sup$, ou ajouter des gardes (les $min()$ et $max()$ à la place de $b$).
**Utilité :** permet d'autres transformations, supprime des tests.

### 3.1.2 Fusion

Rassemble en une boucle deux boucles ayant le même nombtre d'itérations.

$$\begin{array}{l} \text{for (i...)} \\ \quad S_1 \\ \text{for (j...)} \\ \quad S_2 \end{array} \quad \begin{array}{c} \text{fusion} \\ \rightarrow \\ l \\ \leftarrow \\ \text{fission} \end{array} \quad \begin{array}{l} \text{for (...) \{} \\ \quad S_1 \\ \quad S_2 \\ \text{\}} \end{array}$$

**Validité :** $\nexists\ S_1(i)\ \delta\ S_2(j)$ avec $j < i$.
**Utilité :** localité des données.

**Exemple :**

```
for (i = 0; i < 100; i++)
    A[i] = B[i] + 1;
for (i' = 0; i' < 99; i'++)
    C[i'] = A[i' + 1] * 2;
```

$\downarrow \qquad S_2(j)\ \delta\ S_4(j)$

```
A[0] = B[0] + 1;
for (j = 0; j < 99; j++) {
    /* i = j + 1 */
    A[j + 1] = B[j + 1] + 1;
    /* i' = j */
    C[j] = A[j + 1] * 2;
}
```

### 3.1.3 Fission

C'est l'inverse.

$$\begin{array}{l} \text{for } (i \ldots) \text{ \{} \\ \quad S_1 \\ \quad S_2 \\ \text{\}} \end{array} \quad \rightarrow \quad \begin{array}{l} \text{for } (i_1 \ldots) \\ \quad S_1 \\ \text{for } (i_2 \ldots) \\ \quad S_2 \end{array} \quad \text{ou} \quad \begin{array}{l} \text{for } (i_2 \ldots) \\ \quad S_2 \\ \text{for } (i_1 \ldots) \\ \quad S_1 \end{array}$$

**Utilité :** « éloigne » les dépendances, améliorer la localité dans certains cas (beaucoup de références à de gros tableaux qui remplissent le cache).
**Validité :** $\nexists\ S_2\ \delta\ S_1,\ \nexists\ S_2\ \delta\ S_1 \Rightarrow \nexists$ cycle dans le DG entre $S_1$ et $S_2$.

**Exemple :**

```
1 for (i = 0; i < N; i++) {
2     A[i] = A[i] + B[i - 1];
3     B[i] = C[i - 1] + k;
4     C[i] = 1 / B[i];
5     D[i] = sqrt(C[i]);
6 }
```

[[schéma 6]]

```
for (i = 0; i < N; i++) {
    B[i] = C[i - 1] + k;
    C[i] = 1 / B[i];
}
for (i = 0; i < N; i++) { /* parallélisable */
```

```
    A[i] = A[i] + B[i - 1];
    D[i] = sqrt(C[i]);
}
```

### 3.1.4 Reversal (retournement)

Inverse l'ordre des itérations.

```
for (i = inf; i <= sup; i++)

for (i' = sup; i' >= inf; i--)
```

**Validité :** pas de dépendance de vecteur $\neq 0$ portée par la boucle.

**Exemple : reversal sur j**

```
for (i...)
    for (j...)
```

$\delta_{(1,2)} \rightarrow$ OK
$\delta_{(0,0)} \rightarrow$ OK
$\delta_{(0,1)} \rightarrow$ Non !

**Parallélisation :** mêmes conditions de validité.
**Utilité :** optimisations de bas niveau (il est plus facile de comparer un index à 0 qu'à une autre valeur) ; alignement des données et fusion.

### 3.1.5 Interchange

Échange de deux boucles.

**Validité :** $\nexists$ dépendance $(<,>)$, par exemple $(1,-1)$.
**Utilité :**
– contrôle le grain de parallélisme ;
– contrôle l'ordre d'accès aux tableaux.

for $(i \ldots)$ / for $(j \ldots)$ $\rightarrow$ for $(j \ldots)$ / for $(i \ldots)$

0, * -> *, 0 : ok
, 0 -> 0, * : ok
<, < -> >, > : ok
<, > -> >, < : pas ok !

```
for (i = 1; i < M; i++)
    for (j = 0; j < N; j++)
        R1 = A[j][i] = ... A[j][i - 1] ...;
```

↓

```
for (j = 0; j < N; j++)
    for (i = 1; i < M; i++)
        R1 = A[j][i] = ... A[j][i - 1] ...;
```

$d = (1,0) \rightarrow d' = (0,1) \Rightarrow$ ok

### 3.1.6 Skewing (penchage)

[[schéma 8]]

$(i, j) \rightarrow (i, j + f \cdot i)$

```
for (i...)
    for (j = inf; j < sup; j++)

for (i...)
    for (j = inf + f * i; j < sup + f * i; j++)
```

**Remarque** $f$ peut être négatif.

**Validité :** $(d_1, d_2) \rightarrow (d_1, d_2 + f \cdot d_1)$
$\nexists$ dépendance $(d_1 = 0, d_2 + f \cdot d_1 < 0)$
$(d_1 = 0, d_2 < 0) \rightarrow$ toujours valide
**Utilité :** éliminer une composante du vecteur de dépendance pour pouvoir paralléliser une boucle.

### 3.1.7 Transformations linéaires

Les matrices de fransformation sont unimodulaires ($det = \pm 1$, conserve les points entiers).

$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ interchange $\qquad \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$ reversal (externe) $\qquad \begin{pmatrix} 1 & 0 \\ f & 1 \end{pmatrix}$ skewing

$T$ : transformation unimodulaire.

**Dépendance :** $d \rightarrow T \cdot d$
**Validité :** $T \cdot d \geq 0$
$T \cdot i^T - T \cdot i^S = T \cdot (i^T - i^S) = T \cdot d$

### 3.1.8 Strip-mining

Décompose une boucle en deux parties : parcours de blocs, et interne bloc.

```
for (i = 0; i < N; i++)
    ...

    ↓

for (ib = 0; ib < N; ib += s)
    for (i = ib; i < N && i < ib + s; i++)
        ...
```

`i < N` est inutile si `N % s == 0`.

**Validité :**
$d \rightarrow (d\text{div}s, d\text{mod}s)$
$d \rightarrow (d\text{div}s + 1, -((s - d)\text{mod}s)) \rightarrow$ toujours valide
**Utilité :**

- Transformation « partielle » ensuite ;
- limiter le nombre d'itérations d'une boucle (pour l'électronique de taille fixe).

### 3.1.9 Tiling (tuilage)

→ nids de boucles.

```
for (i = 0; i < 50; i++)
    for (j = i; j < 60; j++)
        ...
```

↓

```
for (it = -15; it < 50; it += 20)
    for (jt = it; jt < 60; jt += 20)
        for (i = max(0, it); i < 50 && i < it + 20; i++)
            for (j = max(i, jt); j < 60 && j < jt + 20; j++)
```

**Validité :** si $exchange(i, j)$ valide → ok (condition suffisante mais pas forcément nécessaire). Sinon ?...
**Utilité :** localité des données.

Calcul de la taille de tuile optimale selon :
– programme (voisinage) ;
– taille du cache.
Par exemple, un algorithme de flou sur des images (un pixel est calculé à partir de ceux qui l'entourent).

## 3.2 Parallélisation

Une boucle qui ne porte pas de dépendance peut être parallélisée. En général, on essaye de paralléliser les boucles les plus externes.

## 3.3 Localité de données

[[schéma 9]]

– Localité temporelle : réutilisation de la même donnée encore dans le cache.
– Localité spatiale : accès aux données voisines en mémoire, proches dans le temps.

## 3.4 Exercice

```
for (i = 0; i < N - 1; i++)
    for (j = 0; j < M; j++)
        for (k = 1; k < L; k++)
            A[i + 1][j][k - 1] = A[i][j][k] + B[i][k];
```

1. Interchange possible ?
2. Paralléliser les deux boucles externes.
3. Améliorer la localité ?

### 3.4.1 Interchange

**Vecteurs de dépendance :**
1 – 2 – 3 : 1, 0, -1
1 – 3 – 2 : 1, -1, 0 → ok
2 – 1 – 3 : 0, 1, -1 → ok
2 – 3 – 1 : 0, -1, 1 → pas ok
3 – 2 – 1 : -1, 0, 1 → pas ok
3 – 1 – 2 : -1, 1, 0 → pas ok

### 3.4.2 Parallélisation

```
par_for (j)
  ---------------
    for (i)
        for (k)
  ---------------
```

$$\begin{pmatrix} 1 \\ -1 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 1 \end{pmatrix} \quad \rightarrow \quad \begin{matrix} i \\ k \end{matrix} \rightarrow \begin{cases} i' = i + k \\ k' = i \end{cases} \quad \rightarrow \quad \begin{matrix} i = l' \\ k = i' - k' \end{matrix}$$

```
par_for (j = 0; j < M; j++)
    par_for (i' = 1; i' <= L + N - 3; i'++)
        for (k' = max(0, i' + L + 1); k' <= min(N - 2, i' - 1); k++)
            A[k' + 1][j][i' - k' - 1] = A[k'][j][i' - k'] + B[k'][i' - k'];
```

### 3.4.3 Localité

```
for (j = 0; j < M; j++)
    for (i' = 1; i' <= L + N - 3; i'++) {
        k' = max(0, i' - L + 1);
        R = A[k' + 1][j][i' - k' - 1] = A[k'][j][i' - k'] + B[k'][i' - k'];
        for (k'++; k' <= min(N - 2, i' - 1); k'++)
            R = A[k' + 1][j][i' - k' - 1] = R + B[k'][i' - k'];
    }
```