

INF 560 Calcul Parallèle et Distribué Cours 8

Eric Goubault et Sylvie Putot

Ecole Polytechnique

9 février 2015



PLAN DU COURS

- Analyse systématique de dépendances
- Transformations de boucles
- Algorithme d'Allen et Kennedy

... dans une perspective de parallélisation automatique.

- Revenons à la mémoire partagée... et aux PRAM (CREW)!
- Problème important: paralléliser les nids de boucle
 - les boucles représentent souvent l'essentiel du temps de calcul
 - leur régularité rend l'optimisation plus facile
 - parallélisation par analyse des flots de données
 - méthodes systématiques et automatisables d'analyse de dépendance et de transformations de programmes préservant la sémantique du programme
 - les compilateurs pour langages parallèles utilisent ces techniques issues de la parallélisation automatique
- La récursion est plus difficile à exploiter car moins régulière
- Déjà comprendre ce qui est intrinsèquement séquentiel, de ce qui peut être calculé en parallèle

LOI D'AMDAHL

- Soit un programme qui s'exécute séquentiellement en temps T
- Et s fraction d'exécution séquentielle, au mieux $1 - s$ en parallèle
- Sur p processeurs, l'accélération du calcul complet par rapport à un calcul séquentiel sera au maximum de,

$$Acc = \frac{T}{T_p} = \frac{1}{s + \frac{1-s}{p}}$$

(maximum de $\frac{1}{s}$)

Conséquence: même si on parallélise 80 pourcent d'un code (le reste étant séquentiel), on ne pourra jamais dépasser, quelle que soit la machine cible, une accélération d'un facteur $1/0.2 = 5!$

- Par une succession de transformations.
- Assurer la validité de ces transformations = analyse de programme préalable
- Repose sur la description du flot des données, dont on déduit les contraintes d'ordre d'exécution.
- Reconnaissance de formes: on raisonne sur la géométrie du programme

Nid de boucles = ensemble de boucles imbriquées

EXEMPLE

```
for ( i=1; i<=N; i++) {
  for ( j=i; j<=N+1; j++)
    for ( k=j-i; k<=N; k++) {
      S1;
      S2;
    }
  for ( r=1; r<=N; r++)
    S3;
}
```

Ceci n'est pas un nid de boucle parfait:

- nid de boucle parfait: toutes les instructions sont englobées par les mêmes boucles
- S1 et S2 sont englobées par les boucles i, j et k
- alors que S3 est englobée par les boucles i et r

VECTEURS D'ITÉRATION

- Les instances des itérations de n boucles parfaitement imbriquées sont représentées par un vecteur de dimension n , dont les composantes décrivent les valeurs des indices = vecteur d'itérations
- On note une instance de l'instruction S à l'itération I , $S(I)$
- Généralement les domaines d'itération sont constitués d'entiers dans des polyèdres.

EXEMPLE

Sur l'exemple précédent:

- Pour S3 on a un vecteur d'itération en (i, r) dont le domaine est $1 \leq i, r \leq N$
- Pour S1 on a un vecteur d'itération en (i, j, k) dont le domaine est $1 \leq i \leq N, i \leq j \leq N+1$ et $j-i \leq k \leq N$.

ORDRE SÉQUENTIEL D'EXÉCUTION

Définit l'ordre d'exécution "par défaut":

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

- Les différentes instances d'un nid de boucle sont exécutées en respectant l'ordre lexicographique des vecteurs d'itération
- Au sein d'une instance, les instructions sont exécutées en suivant l'ordre textuel (le texte du programme)
- Pour les nids de boucles non-parfaits, les domaines d'itérations sont incomparables a priori, mais il suffit de "compléter" les vecteurs d'itération de façon cohérente: $I \rightarrow \tilde{I}$

- On veut mettre en parallèle certaines instructions: une partie de l'ordre séquentiel est à respecter absolument, une autre pas (permutation possible d'actions)
- On va définir un ordre partiel (Bernstein), ordre minimal à respecter pour produire un code sémantiquement équivalent à l'ordre initial
- En fait, l'ordre séquentiel va être une *extension* de l'ordre partiel de Bernstein
- Cet ordre partiel est défini à partir de 3 types de dépendances de données: flot, anti et sortie.

Dépendance de données entre $S(I)$ et $T(J)$ si ils accèdent au même emplacement mémoire, et que l'un au moins d'accès est en écriture:

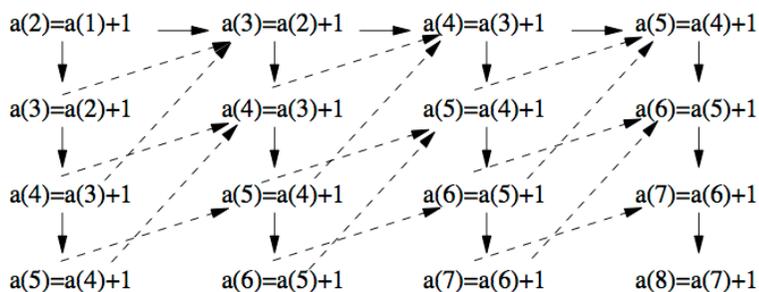
- Toujours dirigées par l'ordre séquentiel: dépendance de $S(I)$ vers $T(J)$ si $S(I) <_{seq} T(J)$
- S et T pas nécessairement distincts
- Dépendance de flot (de $S(I)$ vers $T(J)$) si un emplacement mémoire est en écriture pour $S(I)$ et en lecture pour $T(J)$
- Dépendance anti: lecture pour $S(I)$ et écriture pour $T(J)$
- Dépendance de sortie: écriture pour $S(I)$ et pour $T(J)$

```
// dep de flot:      // dep anti:      // dep de sortie:
a = ... // S        ... = a // S        a = ... // S
... = a // T        a = ... // T        a = ... // T
```

EXEMPLE

```
for (i=1; i<=N; i++)
  for (j=1; j<=N; j++)
    a(i+j) = a(i+j-1)+1;
```

Des dépendances de flot, anti et de sorties:



CALCUL DES DÉPENDANCES: ACCÈS À UNE CASE COMMUNE

- Supposons que $S(I)$ et $T(J)$ accèdent au même tableau a (S en écriture, T en lecture)

```
for ( ... )
  a[f(I)] = ... // S(I)
  ...     = a[g(J)] // T(J)
```

- L'accès au tableau est commun pour les vecteurs I et J si $f(I) = g(J)$
- Si f et g sont des fonctions affines à coefficients entiers: peut se tester en temps polynomial
- Dans le cas général, on va utiliser des calculs/heuristiques approchées
 - l'approximation doit toujours être pessimiste: on peut perdre du parallélisme mais pas négliger une dépendance et engendrer un programme faux
 - par exemple on va souvent ignorer que les solutions trouvées doivent être entières

- Résolution de systèmes d'égalités et d'inégalités affines
- Chercher une dépendance de flot par exemple revient à ajouter en plus la contrainte $S(I) <_{seq} T(J)$
- Calculer de façon hiérarchique sur les indices de boucle
- Donne des graphes de dépendances très redondants
- Chercher les dépendances *directes* (ce qui reste après élimination des dépendances qui peuvent être reconstituées par transitivité) : plus compliqué
 - par programmation linéaire, problèmes d'optimisation dans les entiers → algorithme de Fourier-Motzkin, la référence est l'Omega Test de Pugh utilisé dans `petit`
 - ici il n'est pas possible d'approximer

```
for ( i=1; i<=N; i++)
  for ( j=1; j<=N; j++)
    a(i+j) = a(i+j-1)+1; // S(i, j)
```

DÉPENDANCE DE FLOT

- On cherche dépendance écriture $S(i', j')$ vers lecture $S(i, j)$
- Alors $f(I) = f(i', j') = i' + j'$ et $g(J) = g(i, j) = i + j - 1$
- $f(I) = g(J) \Leftrightarrow i' + j' = i + j - 1$
- $S(i', j') <_{seq} S(i, j) \Leftrightarrow ((i' \leq i - 1) \text{ ou } (i = i' \text{ et } j' \leq j - 1))$

Dépendance de flot directe:

$$\max_{<_{seq}} \{(i', j') \mid (i', j') <_{seq} (i, j), i' + j' = i + j - 1, 1 \leq i, i', j, j' \leq N\}$$

Solution:

$$\begin{cases} (i, j - 1) & \text{si } j \geq 2 \\ (i - 1, j) & \text{si } j = 1 \end{cases}$$

EXEMPLE: ANTIDÉPENDANCE

ANTIDÉPENDANCE

- On cherche dépendance lecture $S(i', j')$ vers écriture $S(i, j)$
- Alors $f(I) = f(i', j') = i' + j' - 1$ et $g(J) = g(i, j) = i + j$
- $f(I) = g(J) \Leftrightarrow i' + j' - 1 = i + j$
- $S(i', j') <_{seq} S(i, j) \Leftrightarrow ((i' \leq i - 1) \text{ ou } (i = i' \text{ et } j' \leq j - 1))$

Antidépendance directe:

$$\max_{<_{seq}} \{(i', j') \mid (i', j') <_{seq} (i, j), i' + j' - 1 = i + j, 1 \leq i, i', j, j' \leq N\}$$

Solution (pour $i \geq 2, j \leq N - 2$):

$$(i - 1, j + 2)$$

APPROXIMATION DES DÉPENDANCES

Le calcul précédent permet de savoir si deux instructions engendrent des opérations en dépendance:

GRAPHE DE DÉPENDANCE ÉTENDU (GDE):

- Sommets: instances $S_i(I)$, $1 \leq i \leq s$ où s est le nombre d'instructions du programme et $I \in D_S$
- Arcs: $S(I) \rightarrow T(J)$ pour chaque dépendance

Mais ce graphe de dépendances détaillé est trop complexe en général: on recherche des représentations synthétiques, plus ou moins approchées

QUELQUES DÉFINITIONS

- Ensemble des paires de dépendances entre S et T :

$$\{(I, J) \mid S(I) \rightarrow T(J)\} \subseteq \mathbf{Z}^{n_S} \times \mathbf{Z}^{n_T}$$

- Ensemble de distances de ces dépendances:

$$\{(\tilde{J} - \tilde{I}) \mid S(I) \rightarrow T(J)\} \subseteq \mathbf{Z}^{n_S, T}$$

Sur l'exemple précédent:

- anti: lecture $a(i+j-1)$ - écriture $a(i+j)$:
 $(i', j') = (i-1, j+2)$ ensemble de distance $\{(1, -2)\}$
- flot: écriture $a(i+j)$ - lecture $a(i+j-1)$: $(i', j') = (i, j-2)$
ou $(i', j') = (i-1, j)$ ensemble de distance $\{(1, 0), (0, 1)\}$
- sortie: écriture $a(i+j)$ - écriture $a(i+j)$:
 $(i', j') = (i-1, j+1)$ ensemble de distance $\{(1, -1)\}$
- Problème: on ne peut généralement calculer le GDE à la compilation! (de toutes façons, est trop gros!)

GRAPHE DE DÉPENDANCE RÉDUIT (GDR)

GDR= il y a dépendance entre deux instructions s'il y a au moins une dépendance entre deux instances de ces instructions

- Sommets: les instructions S_i ($1 \leq i \leq s$) et non plus leurs instances
 - Sur l'exemple précédent, N^2 sommets dans le GDE, 1 sommet dans le GDR
- Arcs: $e : S \rightarrow T$ si il existe au moins un arc $S(I) \rightarrow T(J)$ dans le GDE
- Etiquette: $w(e)$ décrivant un sous-ensemble D_e de $\mathbf{Z}^{n_S, T}$ = surapproximation des ensembles de distances du GDE

Le tri topologique du GDR permet d'avoir une idée des portions séquentielles, et des portions parallélisables.

QUELQUES ABSTRACTIONS

Représentations (ou abstractions) classiques des ensembles de distance en étiquettes du GDR:

- Niveaux de dépendance (GDRN)
- Vecteurs de direction (GDRV)

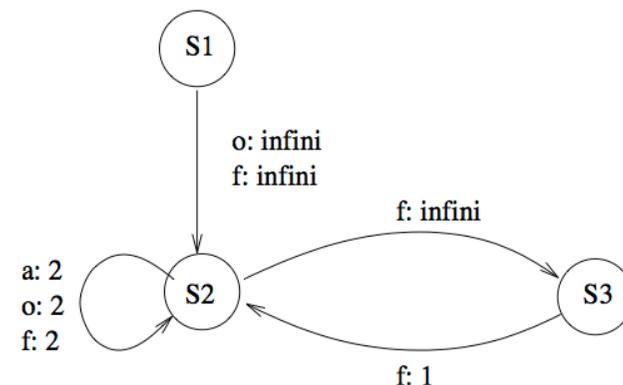
GRAPHE DE DÉPENDANCES RÉDUIT PAR NIVEAUX (GDRN)

- Une dépendance entre $S(I)$ et $T(J)$ est boucle indépendante si elle a lieu lors d'une même itération des boucles englobant S et T
- Sinon elle est portée par la boucle...
- Etiquetage en conséquence, de $e : S \rightarrow T$ du GDR:
 - $l(e) = \infty$ si $S(I) \rightarrow T(J)$ avec $\tilde{J} - \tilde{I} = 0$
 - $l(e) \in [1, n_S, T]$ si $S(I) \rightarrow T(J)$, et la première composante non nulle de $\tilde{J} - \tilde{I}$ est la $l(e)$ -ème composante

EXEMPLE

```

for (i=2; i<=N; i++) {
  S1: s(i) = 0;
  for (j=1; j<=i-1; j++)
    S2: s(i) = s(i)+a(j, i)*b(j);
  S3: b(i) = b(i)-s(i);
}
    
```

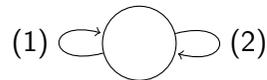


Les deux exemples suivant ont le même GRDN:

```

for ( i=1; i<=N; i++)
  for ( j=1; j<=N; j++)
    a(i, j) = a(i, j-1) + a(i-1, j);

for ( i=1; i<=N; i++)
  for ( j=1; j<=N; j++)
    a(i, j) = a(i, j-1) + a(i-1, N);
    
```



Alors que:

- Exemple 1: plus long chemin de dépendances de longueur $2N$, du parallélisme (diagonales) exploitable, qui ne sera pas détecté par l'algorithme d'Allen et Kennedy qui raisonne sur le GDRN
- Exemple 2: un chemin de dépendance de longueur N^2 ($a_{2,1}, \dots, a_{2,n}, a_{3,1}, \dots, a_{3,N}, \dots, a_{N,N}$) pas parallélisable

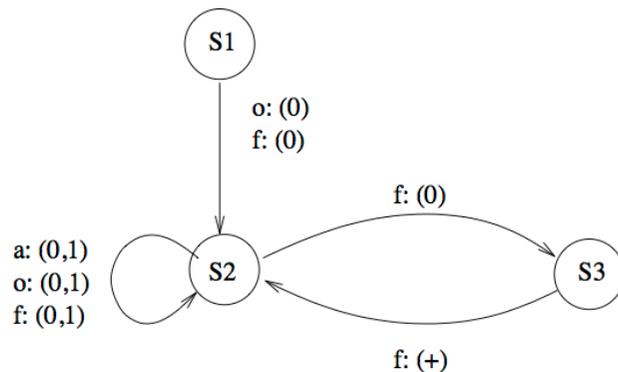
Sont des abstractions des ensembles de vecteurs de distance:

- on note $z+$ pour une composante si toutes les distances sur cette composante ont au moins la valeur z
- on note $z-$ pour une composante si toutes les distances sur cette composante ont au plus la valeur z
- on note $+$ à la place de $1+$, $-$ à la place de $-1-$
- on note $*$ si la composante peut prendre n'importe quelle valeur
- on note z si la composante a toujours la valeur z

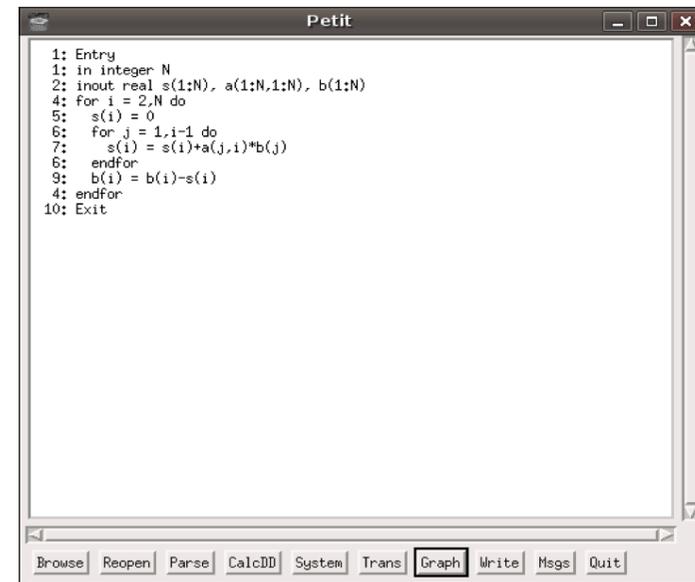
GDRV = GDR annoté par ces vecteurs de direction

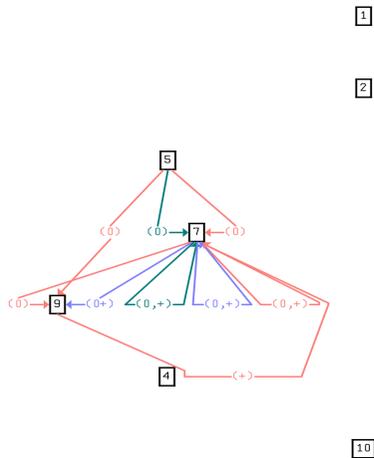
```

for ( i=2; i<=N; i++) {
  S1: s(i) = 0;
  for ( j=1; j<=i-1; j++)
    S2: s(i) = s(i)+a(j, i)*b(j);
  S3: b(i) = b(i)-s(i);
}
    
```



S3 vers S2: dépendances de flot $b(i')$ écrit en S3 sera lu en S2 pour $i \geq i' + 1$, donc vecteur de direction $+$ sur le niveau 1.





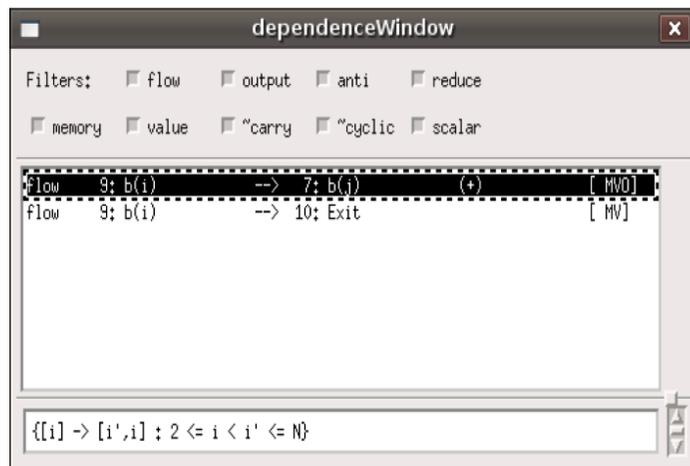
Calcule un GDRV, avec en bleu les dépendances anti, en vert de sortie, et en rouge les dépendances de flot.

```

Petit
1: Entry
1: in integer N
2: inout real s(1:N), a(1:N,1:N), b(1:N)
4: for i = 2,N do
5:   s(i) = 0
6:   for j = 1,i-1 do
7:     s(i) = s(i)+a(j,i)*s(j)
8:   endfor
9:   s(i) = b(i)-s(i)
4: endfor
10: Exit
    
```

Flow 9: b(i) --> 7: b(j) (+) [MW0]

Faire apparaître (Cycl) les dépendances sur un tableau



Ouvrir les dépendances correspondantes pour faire apparaître les vecteurs d'itération

PRINCIPE

Une transformation d'un programme qui préserve le graphe de dépendances étendu fournit un programme équivalent au programme initial

TRANSFORMATIONS ET PARALLÉLISATION DE BOUCLES

- Une boucle peut être parallélisée si il n'y a aucune dépendance entre ses itérations
- On peut alors paralléliser un nid de boucle niveau par niveau: une boucle for de profondeur k est équivalente à une boucle forall (parallèle) si elle ne contient pas de dépendances de niveau k
- Des transformations valides peuvent faire apparaître de nouvelles opportunités de parallélisation

TRANSFORMATIONS ET PARALLÉLISATION DE BOUCLES

ALGORITHME D'ALLEN ET KENNEDY (BASÉ SUR LE GDRN)

- Distribution de boucles

ALGORITHME DE LAMPORT (BASÉ SUR LE GDRV)

- Torsion de boucles
- Inversion de boucles
- Permutation de boucles

- Egalement réduction/fusion, déroulement, etc.

DISTRIBUTION DE BOUCLES

Principe de la distribution: une boucle contenant au moins deux instructions est équivalente à la séquence de deux boucles s'il n'existe pas de cycle de dépendance dans le GDR

EXEMPLE

```
for (i=0 ; i<N ; i++) {  
  S1: s += a[i];  
  S2: b[i] = c[i] + d[i];  
}
```

Pas de dépendances entre S1 et S2, la distribution est donc possible (on peut même échanger les boucles sur S1 et S2), et la boucle sur S2 est parallèle:

```
for (i=0 ; i<N ; i++)  
  S1: s += a[i];  
forall (i=0 ; i<N ; i++)  
  S2: b[i] = c[i] + d[i];
```

DISTRIBUTION DE BOUCLES: AUTRES CONFIGURATIONS

EXEMPLE

Dépendance de flot S1 -> S2

```
for (i=0 ; i<N ; i++) {  
  S1: a[i] = ...;  
  S2: ... = a[i-1];  
} devient  
for (i=0 ; i<N ; i++)  
  S1: a[i] = ...;  
for (i=0 ; i<N ; i++)  
  S2: ... = a[i-1];
```

EXEMPLE

Anti-dépendance S2 -> S1: réordonnancement

```
for (i=0 ; i<N ; i++) {  
  S1: a[i] = ...;  
  S2: ... = a[i+1];  
} devient  
for (i=0 ; i<N ; i++)  
  S2: ... = a[i+1];  
for (i=0 ; i<N ; i++)  
  S1: a[i] = ...;
```

FUSION DE BOUCLES

A l'inverse, on peut vouloir opérer la transformation inverse de la distribution, si cela n'empêche pas du parallélisme

EXEMPLE

```
for (i=1; i<=N; i++)  
  D[i]=E[i]+F[i];  
for (j=1; j<=N; j++)  
  E[j]=D[j]*F[j];
```

devient:

```
for (i=1; i<=N; i++)  
{  
  D[i]=E[i]+F[i];  
  E[j]=D[j]*F[j];  
}
```

Cela permet une vectorisation et donc une réduction du coût des boucles (dans tous les cas, on économise sur le contrôle et on améliore la localité).

- Point de départ = GDRN
- Cycles de dépendances = composantes fortement connexes du graphe
- Transformations autorisées = distribution de boucles pour réduire les dépendances
- On entoure chaque composante fortement connexe d'une distribution de la boucle la plus externe
- Cette boucle est séquentielle s'il existe dans la composante une dépendance de profondeur la boucle
- Remplacer certaines boucles for par des boucles forall
- En opérant de façon itérative sur la profondeur/niveau de boucle
- Les instructions ne sont pas modifiées

On appelle l'algorithme suivant avec $k = 1$ sur le GDRN G :

ALLEN-KENNEDY(G, k)

- Supprimer dans le GDRN G toutes les arêtes de niveau strictement inférieur à k
- Calculer les composantes fortement connexes (CFC) de G
- Pour tout CFC C dans l'ordre topologique:
 - Si C est réduit à une seule instruction S sans arête, alors générer des boucles parallèles dans toutes les dimensions restantes (i.e. niveaux k à n_S) et générer le code
 - Sinon, $l = l_{min}(C)$ (niveau minimal de dépendance des arêtes de C)
générer des boucles parallèles du niveau k au niveau $l - 1$, et une boucle séquentielle pour le niveau l .
Allen-Kennedy($C, k = l + 1$).

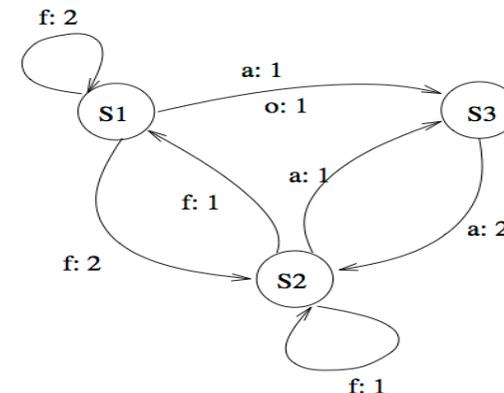
EXEMPLE

```

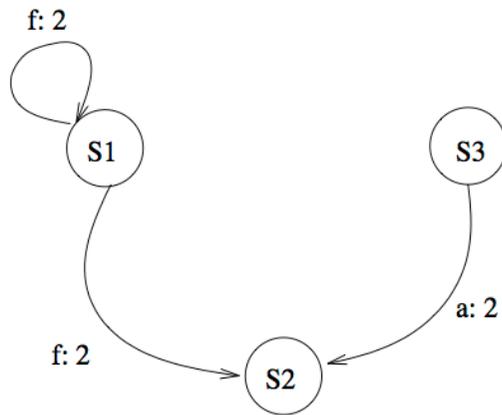
for (i=1; i<=N; i++)
  for (j=1; j<=N; j++) {
    S1: a(i+1, j+1) = a(i+1, j)+b(i, j+2);
    S2: b(i+1, j) = a(i+1, j-1)+b(i, j-1);
    S3: a(i, j+2) = b(i+1, j+1)-1;
  }
    
```

- Flot $S1 \rightarrow S1$, variable a , distance $(0, 1)$,
- Flot $S1 \rightarrow S2$, variable a , distance $(0, 2)$,
- Flot $S2 \rightarrow S1$, variable b , distance $(1, -2)$,
- Flot $S2 \rightarrow S2$, variable b , distance $(1, 1)$,
- Anti $S1 \rightarrow S3$, variable a , distance $(1, -2)$,
- Anti $S2 \rightarrow S3$, variable a , distance $(1, -3)$,
- Anti $S3 \rightarrow S2$, variable b , distance $(0, 1)$,
- Sortie $S1 \rightarrow S3$, variable a , distance $(1, -1)$.

EXEMPLE



- Le GDRN est fortement connexe et a des dépendances de niveau 1
- La boucle sur i (niveau 1) sera donc séquentielle
- On enlève maintenant les dépendances de niveau 1...



- S2 et S3 forment chacun une CFC sans arête: on peut les distribuer et obtenir des boucles parallèles
- S1 contient une arête de niveau 2: on ne pourra pas paralléliser cette boucle

Ci-dessous le code généré correspondant à l'algorithme avec énumération des CFC dans l'ordre topologique (S1 et S3 ont une dépendance vers S2, d'où réordonnancement des boucles intérieures: en plus de la distribution)

```

for (i=1; i<=N; i++) {
  for (j=1; j<=N; j++)
    S1: a(i+1, j+1) = a(i+1, j)+b(i, j+2);
  forall (j=1; j<=N; j++)
    S3: a(i, j+2) = b(i+1, j+1)-1;
  forall (j=1; j<=N; j++)
    S2: b(i+1, j) = a(i+1, j-1)+b(i, j-1);
}
  
```

D'AUTRES TRANSFORMATIONS DE BOUCLES

TRANSFORMATIONS UNIMODULAIRES (LAMPART)

Transformations qui changent de façon bijective le vecteur d'itérations des boucles en définissant $I' = TI$ où T est une matrice unimodulaire, ie à coeff entiers de déterminant égal à + ou -1. Combinaisons de:

- Permutations/échanges de boucles
- Inversions de boucles (parcourir les boucles en sens inverse)
- Torsions/rotations de boucles

PERMUTATION/ÉCHANGE DE BOUCLES

Dans un nid de boucle parfait, une boucle parallèle peut toujours être amenée en position la plus interne (par exemple pour être vectorisée)

EXEMPLE

```

for (i=2; i<=N; i++)
  for (j=2; j<=M; j++)
    A[i, j]=A[i, j-1]+1;
  
```

devient,

```

for (j=1; j<=M; j++)
  A[1:N, j]=A[1:N, j-1]+1;
  
```

Les critères pour examiner si on peut échanger deux boucles sont inclus dans ceux qui permettent de dire si la boucle externe est parallèle.

EXAMPLE

```
forall (j=1; j<=N; j++)
  forall (k=1; k<=N; k++)
  ...
```

devient,

```
forall (i=1; i<=N*N; i++)
  ...
```

Cela permet de changer l'espace d'itérations (afin d'effectuer éventuellement d'autres transformations).

- Parallélisation polyédrique: transformation qui regroupe autant de dépendances que possible sur la boucle extérieure
- Correspond à un changement de base; "front d'ondes"

EXAMPLE

Aucune boucle n'est parallèle a priori

```
for (i=1; i<=N; i++)
  for (j=1; j<=N; j++)
    a[i, j] = (a[i-1, j] + a[i, j-1]) / 2;
```

En faisant une rotation de l'espace d'itérations de 45 degrés, $k = (i + j)/2$, $l = (j - i)/2$, on obtient:

```
for (k=2; k<=N; k++)
  forall (l=2-k; l<=k-2; l+=2)
    a[(k-1)/2, (k+1)/2] = (a[(k-1)/2-1, (k+1)/2] +
                           a[(k-1)/2, (k+1)/2-1]) / 2;

for (k=1; k<=N; k++)
  forall (l=k-N; l<=N-k; l+=2)
    a[(k-1)/2, (k+1)/2] = (a[(k-1)/2-1, (k+1)/2] +
                           a[(k-1)/2, (k+1)/2-1]) / 2;
```

Transformation toujours légale:

EXAMPLE

```
for (i=1; i<=100; i++)
  A[101-i] = A[i];
```

devient

```
forall (i=1; i<=50; i++)
  A[101-i] = A[i];
forall (i=51; i<=100; i++)
  A[101-i] = A[i];
```

- Une boucle séquentielle avec dépendances est transformée en deux boucles qui peuvent chacune être parallélisées (dans chaque boucle pas d'accès à une case commune)
- Mais la 1ère boucle doit être effectuée avant la 2nde

Pour utiliser le parallélisme d'instruction

EXAMPLE

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

devient,

```
for (i=1; i<=99; i=i+2)
{
  A[i] = B[i+2] * C[i-1];
  A[i+1] = B[i+3] * C[i];
}
```

Remarque: optimisations du compilateur ... peuvent compliquer l'analyse de dépendances...

- Il y a une relation entre la taille de la mémoire et le degré de parallélisme
- Si n calculs doivent être exécutés en parallèle, il faut n cellules mémoire pour qu'il n'y ait pas de dépendances
- Sinon on ne peut pas exploiter le parallélisme

EXEMPLE

Expansion de scalaire

```

for (i=1; i<=N; i++) {
  t = a[i];
  a[i] = b[i];
  b[i] = t;
}
    
```

devient

```

for (i=1; i<=N; i++) {
  tt[i] = a[i];
  a[i] = b[i];
  b[i] = tt[i];
}
    
```

- On élimine le cycle d'anti-dépendance dans le GDR
- La boucle peut ensuite facilement être distribuée/vectorisée

```

for (i=1; i<=N; i++) {
  s = 0.0;
  for (j=1; j<=N; j++)
    s += a[i][j]*b[j];
  c[i] = s;
}
    
```

Pas de parallélisme car un seul accumulateur c .

```

forall (i=1; i<=N; i++) {
  c[i] = 0.0;
  for (j=1; j<=N; j++)
    c[i] += a[i][j]*b[j];
}
    
```

Maintenant, la boucle sur i est parallèle...

EXEMPLE

Expansion de noeud

```

for (i=1; i<=N; i++) {
  a[i] = x[i+1]+x[i];
  x[i+1] = b[i] + t;
}
    
```

devient

```

for (i=1; i<=N; i++) {
  S1: xx[i] = x[i+1];
  S2: a[i] = xx[i]+x[i];
  S3: x[i+1] = b[i] + t;
}
    
```

- Le cycle dans le GDR d'origine est éliminé
- La boucle peut ensuite facilement être distribuée/vectorisée en réordonnant S1, S3, S3

Phase de *remontée* après une décomposition LU:

Soit donc à résoudre le système triangulaire supérieur

$$Ux = b$$

avec,

$$U = \begin{pmatrix} U_{1,1} & U_{1,2} & U_{1,3} & \cdots & U_{1,n} \\ 0 & U_{2,2} & U_{2,3} & \cdots & U_{2,n} \\ & & & \cdots & \\ 0 & 0 & \cdots & 0 & U_{n,n} \end{pmatrix}$$

et $U_{i,i} \neq 0$ pour tout $1 \leq i \leq n$.

REMONTÉE...

On procède par “remontée” c’est à dire que l’on calcule successivement,

$$x_n = \frac{b_n}{U_{n,n}}$$

$$x_i = \frac{b_i - \sum_{j=i+1}^n U_{i,j} x_j}{U_{i,i}}$$

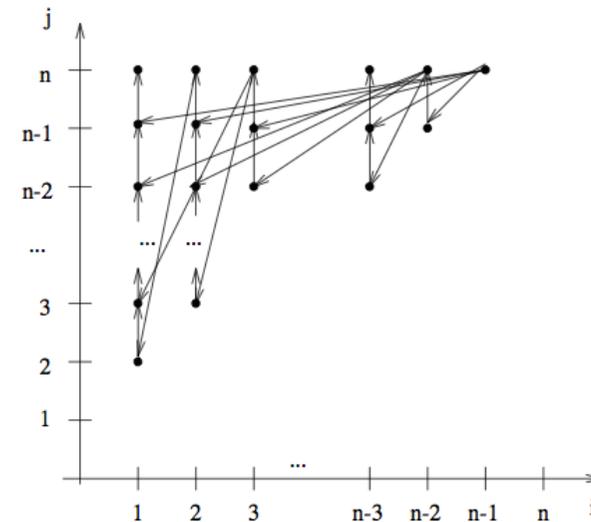
pour $i = n - 1, n - 2, \dots, 1$.

ALGORITHME SÉQUENTIEL

```
x[n]=b[n]/U[n,n];
for (i=n-1; i >= 1; i--) {
  x[i]=0;
  for (j=i+1; j <= n; j++)
    x[i]=x[i]+U[i,j]*x[j];
  x[i]=(b[i]-x[i])/U[i,i];
}
```

GRAPHE DE DÉPENDANCES ÉTENDU

Dépendances de flot:



PARALLÉLISATION

En faisant une rotation et une distribution de boucles:

```
forall (i=1; i <= n-1; i++) // H'
  x[i]=b[i];
x[n]=b[n]/U[n][n]; // T'
for (t=1; t <= n-1; t++) { // H
  forall (i=1; i <= n-t; i++)
    x[i]=x[i]-x[n-t+1]*U[i][n-t+1]; // L
  x[n-t]=x[n-t]/U[n-t][n-t]; // T
}
```

Le ratio d'accélération est d'ordre $\frac{n}{4}$ asymptotiquement:...

COÛT SIMD

- En ne regardant que la partie H du code (la seule vraiment importante asymptotiquement), et on a:
 - un coût de 2 pour L, un coût de 1 pour rapatrier $x[n-t+1]$
 - les calculs se recouvrent dans le forall, on ne compte donc pas la boucle sur i,
 - un coût égal de 1 pour T
 - donc un coût de $4(n-1)$ pour H en sommant sur t
- A cela, il faut ajouter le temps pour H' égal à 1, et encore 1 pour T'.
- Donc un coût total de $4n-2$.
- A rapporter à un coût séquentiel de l'ordre de n^2

- Semaine prochaine: géométrie du parallélisme et tolérance aux pannes (Eric Goubault)