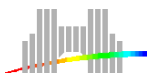
Optimisation des boucles, I/II

Compilation avancée et optimisation de programmes

Paul Feautrier

ENS de Lyon
Paul.Feautrier@ens-lyon.fr

8 novembre 2005



Importance des boucles

- Dans la majorité des programmes (calcul intensif, traitement du signal), les boucles représentent l'essentiel du temps de calcul.
- La régularité des boucles rend leur optimisation plus facile.
- L'autre source de complexité, la récursion, est plus difficile à exploiter car moins régulière (structure d'arbre contre structure de grille).

Nids de boucles

Un nid de boucles est un ensemble de boucles emboitées contenant des instructions exécutables (affectations).

- La *profondeur* d'une instruction est le nombre de boucles qui l'entourent.
- Le nid de boucles est *parfait* si toutes les instructions sont à la profondeur maximale.
- Chaque boucle a un *compte-tour* qui démare à 0, qui croit par pas de 1 et qui n'est pas modifié dans le corps de boucle. On suppose que les compte-tours ont des noms distincts.
- On distingue les boucles `do`, qui font un nombre de tours déterminé d'avance, et les boucles `while` dont le nombre de tours est imprévisible.

Vecteur d'itération

- Vecteur d'itération d'une instruction: liste des compte-tours des boucles englobantes, rangés de l'extérieur vers l'intérieur, et traité comme un vecteur (pour simplifier les noations).
- Le vecteur d'itération est un objet mathématique ; ses composantes sont des variables dont on peu changer les noms et a qui on peut donner des valeurs (numériques ou algébriques).
- Une *opération* est caractérisée par le nom de l'instruction exécutée et par la valeur courante du vecteur d'itération: $\langle S, i \rangle$.

Espace et Domaine d'itération

- Espace d'itération: $\mathbb{N}^d$, où d est le nombre de boucles englobantes.
- Domaine d'itération: l'ensemble des valeurs légales du vecteur d'itération.
- Le domaine ditération est délimité par les bornes des boucles et par des tests.
- Il peut être connu à la compilation (boucles `do`) ou seulement à l'exécution (boucles `while`).
- Il peut être paramétré.

Ordre d'exécution, I/III

Relation d'ordre

- Toute relation réflexive, symétrique et transitive.
- Ordre strict associé: $x < y \equiv x \leq y \ \ \& \ \ x \neq y$.
- Ordre total: $x \leq y \vee y \leq x$.
- Un ordre qui n'est pas total est partiel.

Ordre d'exécution, II/III

- Dans un programme séquentiel, l'ordre d'exécution des opérations, $<_{\text{seq}}$. est entièrement fixé par le texte du programme et les instructions de contrôle.
- L'ordre d'exécution est *total*. Il vaut mieux travailler avec un ordre strict: éviter de dire qu'une opération est exécutée avant elle même.
- L'ordre engendré par les boucles est l'ordre lexicographique $\ll$.

$$\begin{aligned} \langle S, i\rangle <_{\text{seq}} \langle T, j\rangle \quad &\equiv \quad i[1..N_{ST}] \ll j[1..N_{ST}] \\ &\vee \quad (i[1..N_{ST}] = j[1..N_{ST}] \ \ \& \ \ S <_{\text{text}} T) \end{aligned}$$

- $<_{\text{text}}$ l'ordre *textuel*.

Ordre d'exécution, III/III

Boucles parallèles

- Il est impossible de déterminer l'ordre d'exécution d'une boucle parallèle.

```
//for{i=0; i<n; i++)
      S;
```

- S'il y a P processeurs, chaque processeur va peut-être exécuter les itérations de P en P.
- La boucle peut également être découpée en P blocs de taille (approximativement) égale.
- Ceci revient a enlever un terme dans la formule de l'ordre lexicographique.

Dépendances, Rappel

- Dans un nid de boucle, les cellules de mémoire lues et modifiées dépendent du vecteur d'itération (directement ou indirectement): $\mathcal{R}(S, i), \mathcal{W}(S, i)$.
- Dans les cas favorables, ces ensembles se lisent directement sur le texte du programme.
- Mais il peut être nécessaire d'utiliser des méthodes d'analyse (ex. détermination de variables inductives).
- On peut se contenter de tester l'existence de dépendances ou bien construire la relation de dépendance:

$$\begin{aligned} \langle R, i\rangle \, \delta \, \langle S, j\rangle \quad &\equiv \quad \langle S, i\rangle <_{\text{seq}} \langle T, j\rangle \\ &\& \quad i \in D_S \;\; \& \;\; j \in D_T \\ &\& \quad f_S(i) = f_T(j) \neq \emptyset \end{aligned}$$

Profondeur

- Dans la définition ci-dessus, l'ordre d'exécution est une disjonction. Il en résulte que la relation de dépendance est une union.
- Il est très souvent utile de séparer les éléments de l'union. On parle alors de dépendance à la profondeur p:

$$\begin{aligned} \langle R, i\rangle \, \delta \, \langle S, j\rangle \quad &\equiv \quad i[1..p] = j[1..p] \;\; \& \;\; i[p+1] < j[p+1] \\ &\& \quad i \in D_S \;\; \& \;\; j \in D_T \\ &\& \quad f_S(i) = f_T(j) \neq \emptyset \end{aligned}$$

Tests de dépendances

Méthodes plus ou moins rapides et approchées pour décider de l'existence des dépendances. On suppose que toutes les contraintes du système sont linéaire. On ignore les contraintes non linéaires.

- Test de Banerjee: on ne prend en compte que les equations aux indices que l'on évalue aux bornes de l'intervalle d'itération.
- Test du PGCD: on vérifie que dans une équation, le pgcd des coefficients divise le terme constant.
- Test de Fourier-Motzkin: élimination successive des variables jusqu'à réduction à des inégalités numériques.
- Test Oméga: extension du test de Fourier-Motzkin pour tenir compte de l'intégrité des inconnues.
- Simplex: algorithme de programmation linéaire en rationels.
- Programmation linéaire en nombres entiers.

OPTIMISATION DE BOUCLES

- Eliminer les calculs inutiles
- Paralléliser
- Réduire la taille de la mémoire
- Améliorer la localité

Comparaisons

- Eliminer les calculs inutiles est en général une bonne optimisation, bien qu'elle fasse perdre du parallélisme :-). Cependant, il peut arriver qu'il soit plus efficace de recalculer une valeur que d'aller la chercher très loin.
- Paralléliser fait gagner un facteur P (taux de parallélisme) si le programme s'y prête (attention à la loi d'Amdhal).
- Réduire la taille de la mémoire, parce qu'une petite mémoire est plus rapide qu'une grande. Uniquement si on peut ajuster la configuration, ou si on veut éviter la pagination.
- Améliorer la localité, le plus immédiatement efficace. Facteurs d'accélération de l'ordre du rapport temps d'accès mémoire / temps d'accés cache, peut atteindre 100. Pas besoin de matériel spécial ;

Eliminer les calculs inutiles, I/II

Code Mort

- Tout calcul qui n'est jamais exécuté (analyse du graphe de contrôle), ou dont les résultats ne sont jamais utilisé.
- Si une variable n'est jamais à la source d'une dépendance de flot (PC), son calcul est inutile.
- Si une cellule de tableau n'est jamais source (au sens des dépendances directes), son calcul est inutile.

Eliminer les calculs inutiles, II/II

Invariants de boucle

- Toute quantité dont on peut montrer qu'elle reste constante pendant l'exécution de la boucle. Son calcul peut être hissé (*hoisted*) à l'extérieur de la boucle.
- Une variable est invariante dans une boucle si ses sources sont externes à la boucle et ne dépendent pas du compte-tour.
- Une expression est invariante si tous ses arguments le sont.

Un exemple plus complexe

```
for(i=0; i<n; i++)
   for(j=i+1; j<n; j++)
      for(k=i+1; k<n; k++)
G:       a[j][k] -= a[j,i]*a[i][k]/a[i][i];
```

On veut sortir la division de la triple boucle ($O(n^3)$). Source de `a[i][i]`: **if** $i = 0$ **then** $\perp$ **else** $\langle G, i-1, i, i\rangle$. Suggestion:

```
for(i=0; i<n; i++){
P: piv = 1./a[i][i];
   for(j=i+1; j<n; j++)
      for(k=i+1; k<n; k++)
G:       a[j][k] -= a[j,i]*a[i][k]*piv;
}
```

La source de `piv` dans G est bien P, et la source de `a[i][i]` est la même que dans le texte original, donc les deux programmes sont équivalents. $O(n)$ divisions.

Représentation du parallélisme

- Un programme séquentiel étant donné, trouver un programme parallèle équivalent.
- Le programme séquentiel est déterministe, le programme paralléle doit l'être aussi (sémantique par entrelacement).
- Dans un programme parallèle, l'ordre de certaines opérations n'est pas spécifié: exemples.
- L'ordre d'exécution parallèle, $<_{//}$, est *partiel*. Une exécution correspond à l'une des linéarisations possibles de l'ordre partiel. Il y en a en général beaucoup.

Théorème fondamental

Pour qu'un programme parallèle soit équivalent à un programme séquentiel, il faut et il suffit que son ordre d'exécution étende la relation de dépendance.

$$u <_{\text{seq}} v, u \,\delta\, v \Rightarrow u <_{//} v.$$

Démonstration.

Etant donné une exécution parallèle et une exécution séquentielle, on peut les ramener l'une à l'autre par permuation successive d'opérations indépendantes, donc qui commutent. □

Parallélisation d'une boucle

- L'ordre d'exécution d'une boucle parallèle est vide, et il doit étendre la relation de dépendance.
- Une boucle peut donc être parallélisée s'il n'y a aucune dépendance entre ses itérations.
- Application des tests de dépendance.

Parallélisation d'une boucle d'un nid parfait

- Si la boucle de profondeur p est parallèle, le terme correspondant dans l'ordre d'éxécution (qui commence par p égalités) a disparu.
- Il ne doit donc pas y avoir de dépendance de profondeur p.
- La parallélisation d'un nid de boucles parfait se fait donc niveau par niveau, chaque niveau étant indépendant des autres.

Parallélisation d'un nid de boucles imparfait

Exemple:

```
for(i=0; i<n; i++){
Z:  c[i] = 0.;
    for(j=0; j<n; j++)
M:      c[i] += a[i][j]*b[j];
}
```

La boucle sur i est elle parallèle ?

$$\begin{aligned} \langle Z,i\rangle <_{\text{seq}} \langle Z,i'\rangle &\equiv i < i', \\ \langle M,i,j\rangle <_{\text{seq}} \langle M,i',j'\rangle &\equiv i < i' \vee (i = i' \ \& \ j < j'), \\ \langle Z,i\rangle <_{\text{seq}} \langle M,i',j'\rangle &\equiv i < i' \vee i = i'. \end{aligned}$$

Or toute dépendance sur c comporte la clause $i = i'$.

Règle: Une boucle de profondeur p est parallèle si son corps de boucle n'a aucune dépendance de profondeur p.

Conclusion provisoire

- On dispose de méthodes simples pour trouver du parallélisme dans les boucles ...
- ... mais ces méthodes sont peu efficaces: il n'y a pas beacoup de parallélisme dans les programmes ordinaires:

Exemple I

Mauvaises habitudes:

```
for(i=0; i<n; i++){
Z:   s = 0.;
     for(j=0; j<n; j++)
M:       s += a[i][j]*b[j];
C:   c[i] = s;
}
```

Toute boucle dans laquelle un scalaire est modifié est séquentielle!

Exemple II

```
for(i=0; i<n; i++){
S: s += a[i];
A: b[i] = c[i]+d[i];
}
```

Accumuler des calculs dans la même boucle gène la parallélisation.

Transformations de programme

- On cherche à modifier le programme initial pour faire apparaître la parallélisme sans modifier les résultats finaux.
- Premier type de transformation: on exécute les mêmes opérations, mais dans un ordre différent.
- Deuxième type: on calcule les mêmes valeurs, mais on les range différemment en mémoire.

Principe

- Soit deux programmes séquentiels ayant le même ensemble d'opérations mais des ordres d'exécution (totaux) différents, $<_1$ et $<_2$.
- On considère l'ordre (partiel) $<_{//} = <_1 \cap <_2$. Si le programme correspondant est déterministe, les deux programmes d'origine sont équivalents.
- Deux opérations qui ne sont pas ordonnées par $<_{//}$ doivent être indépendantes:

$$u <_1 v \ \& \ v <_2 u \Rightarrow \neg\, u\, \delta\, v.$$

u et v forment une *paire critique*.

Eclatement de boucle

```
for(i= ....){
    S1;
    S2;
}
```

$\Longrightarrow$

```
for(i= ....)
    S1;
for(i= ....)
    S2;
```

- $\langle S_1, i\rangle <_1 \langle S_2, i'\rangle \equiv i \leq i'$,
- $\langle S_1, i\rangle <_2 \langle S_2, i'\rangle \equiv$ **true**.
- Paires critiques $\langle S_1, i\rangle, \langle S_2, i'\rangle, i' < i$: il ne doit pas y avoir de dépendance de S_2 vers S_1.
- On peut également envisager l'éclatement avec inversion.
- La boucle est insécable s'il y a *à la fois* des dépendances de S_1 vers S_2 et de S_2 vers S_1.

Exemple II

```
for(i=0; i<n; i++){
S: s += a[i];
A: b[i] = c[i]+d[i];
}
```

On trouve trois dépendances de S sur elle-même (PC, CP, PP), mais aucune dépendance de S vers A, ni de dépendance de A sur elle même. L'éclatement est donc possible, et la boucle sur A est parallèle.

```
for(i=0; i<n; i++)
S: s += a[i];
//for(i=0; i<n; i++)
A: b[i] = c[i]+d[i];
}
```

Fusion de boucles

- C'est la transformation inverse de l'éclatement de boucle. Quand il n'y a pas de parallélisme, on économise sur le contrôle et on peut améliorer la localité.
- Méthode: on effectue la fusion, et on vérifie que l'éclatement est possible.

Permutation de boucles

```
for(i= ....)                    for(j= ....)
    for(j= ...)     ⟹              for(i= ....)
       S;                              S;
```

- $\langle S,i,j\rangle <_1 \langle S,i',j'\rangle \equiv i < i' \vee (i = i' \ \& \ j < j')$,
- $\langle S,i,j\rangle <_2 \langle S,i',j'\rangle \equiv j < j' \vee (j = j' \ \& \ i < i')$.
- Paires critiques $\langle S,i,j\rangle, \langle S,i',j'\rangle, i' < i \ \& \ j' < j$.
- Intérêt: en général, le caractère parallèle “suit” la boucle. On peut ainsi adapter le code à l'architecture cible.

Exemple III

```
for(i=0; i<n; i++){
Z:  c[i] = 0.;
    for(j=0; j<n; j++)
M:      c[i] += a[i][j]*b[j];
}
```

La boucle sur i est parallèle, à gros grain. Eclater la boucle sur i, puis permuter avec la boucle sur j:

```
for(i=0; i<n; i++)
Z:  c[i] = 0.;
for(j=0; j<n; j++)
    //for(i=0; i<n; i++)
M:      c[i] += a[i][j]*b[j];
```

La dernière ligne est une opération *vectorielle*

`c[0..n-1] += a[0..n-1][j] * b[j].`

Mémoire et parallélisme

- Il y a une relation entre la taille de la mémoire et le degré de parallélisme.
- Si n opérations doivent être exécutées en parallèle, et si chacune produit un résultat, if faut n cellules de mémoire, sans quoi il y aurait des dépendances PP.
- Inversement, si les n cellules de mémoire ne sont pas prévues dans le programme, le parallélisme ne peut pas être exploité.
- On peut donc être amené a agrandir les structures de données d'un programme pour y trouver plus de parallélisme.

Mise en assignation unique

- Calculer la source de chaque référence à droite du programme.
- Remplacer chaque référence à gauche par un nouveau tableau indexé par tous les compte-tours des boucles englobantes.
- Remplacer chaque référence à droite par sa source (expression conditionelle).
- Si la source est indéfinie, laisser la référence à l'ancien tableau.

Exemple

```
for(i=0; i<n; i++){
Z:   s = 0.;
     for(j=0; j<n; j++)
M:       s += a[i][j]*b[j];
C:   c[i] = s;
}
```

- La source de `s` dans `M` est **if** $j = 0$ **then** $\langle Z, i \rangle$ **else** $\langle M, i, j-1 \rangle$.
- Les sources de `a` et `b` sont indéfinies.
- La source de `s` dans `C` est $\langle M, i, n-1 \rangle$.

```
for(i=0; i<n; i++){
    Z[i] = 0.;
    for(j=0; j<n; j++)
        M[i][j] = (j==0?Z[i]:M[i][j-1]) + a[i][j]*b[j];
    C[i] = M[i][n-1];
}
```

Expansion de la mémoire

- La mise en assignation unique supprime les dépendances CP et PP. On trouve donc plus de parallélisme.
- On peut souvent se contenter d'une expansion plus réduite (voir l'exemple) ou faire suivre la parallélisation d'une phase de contraction.
- Il existe une version réduite, l'expansion de scalaire, où on n'indexe que par le compte tour de la boucle la plus interne ...
- ... ainsi qu'une version statique (la SSA form) où on se contente d'un renommage et où le calcul des sources est approximatif.

Algorithme de Allen et Kennedy

Objectif: générer un programme adapté à un ordinateur vectoriel.

- Trouver l'éclatement maximum, parce qu'un opérateur vectoriel ne fait qu'une opération à la fois (parfois 2, chaînage).
- Trouver le parallélisme maximum.
- Placer le parallélisme dans la boucle la plus interne.

Contraintes:

- Les boucles ne sont pas modifiées, mais peuvent être dupliquées.
- Les instructions ne sont pas modifiées.
- Chaque instruction doit être entourée par les mêmes boucles.
- Ces contraintes garantissent que l'ensemble des opérations ne change pas ; seul l'ordre d'exécution est modifié.

Graphe de dépendance réduit

- Le point de départ de l'algorithme de Kennedy et Allen est le Graphe de Dépendance Réduit (GDR).
- Sommets: les instructions du programme.
- Le graphe est orienté. Chaque arc est décoré d'une *profondeur*.
- Il y a un arc de S vers T de profondeur p s'il existe une itération $\langle S, i\rangle$ et une itération $\langle T, j\rangle$ légales, en dépendances, et telles que:

$$i[1..p] = j[1..p] \ \ \& \ \ i[p+1] < j[p+1]$$

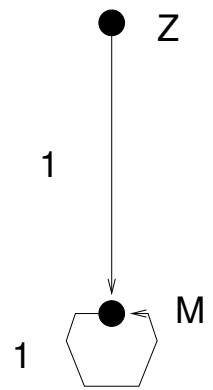
si p est inférieur au nombre de boucles englobant S et T, ou bien:

$$i[1..p] = j[1..p] \ \ \& \ \ S <_{\text{text}} T$$

dans le cas contraire.

Exemple

```
for(i=0; i<n; i++){
Z:   c[i] = 0.;
     for(j=0; j<n; j++)
M:       c[i] += a[i][j]*b[j];
}
```



Théorème Fondamental I/II

Hypothèse: principe d'ignorance Si dans le GDR il y a une dépendance de S vers T à profondeur p inférieure à la profondeur maximum, alors toutes les instances:

$$\langle S, i\rangle, \langle T, j\rangle : i[1..p] = j[1..p] \ \ \& \ \ i[p+1] < j[p+1]$$

sont en dépendance. Même hypothèse si p est la profondeux maximum.

Theorem

S'il existe dans le GDR un circuit $S_1 \rightarrow S_2, \ldots, S_n \rightarrow S_1$ alors dans le programme parallèle les instructions $S_1, \ldots, S_n$, appartiennent à une même boucle.

Théorème Fondamental II/II

Preuve Par l'absurde. On suppose que les instructions peuvent être réparties en m groupes $G_1, \ldots, G_m$, les instructions d'un même groupe étant englobées par une même boucle. On suppose les groupes classés dans l'ordre textuel, et que S_1 est dans G_1. Il ne peut y avoir de dépendance entre groupes différents qui remonte l'ordre textuel. Si on suit le cycle $S_1, \ldots, S_n$, on doit à un moment donné quitter G_1, et il est impossible d'y revenir puisqu'il faudrait remonter l'orde textuel, ce qui est une contradiction.

Composantes fortement connexes

- Appartenir à un même cycle d'un graphe est une relation d'équivalence.
- Les classes d'équivalence sont les *composantes fortement connexes* (cfc) du graphe.
- Toutes les instructions d'une cfc appartienennt à une même boucle dans le programme parallèle.
- Il existe un algorithme naïf pour trouver les cfc d'un graphe (intersection des ancètres et des decendants) ...
- ... ainsi qu'un algorithme rapide du à Tarjan.

Graphe réduit

- Il y a un arc dans le graphe des cfc entre H et K s'il existe une dépendance entre une instruction de H et une instruction de K.
- Le graphe des cfc ou graphe réduit est évidemment acyclique.
- On construit le programme parallèle en énumérant les cfc dans un ordre compatible avec le graphe réduit.
- Chaque cfc est entourée par la boucle la plus externe commune à toutes ses instructions.
- Cette boucle est séquentielle s'il existe dans la cfc une dépendance de profondeur 0, et parallèle sinon. Dans ce dernier cas, la cfc n'a qu'une seule instruction.

Récursion

- Le programme ainsi obtenu satisfait les dépendances de profondeur 0 et les dépendances entres cfc. On peut donc les effacer.
- Pour construire les boubles de profondeur 1, on appelle récursivement le même algorithme indépendemment sur chaque cfc.
- Comme on a effacé des arcs, les cfc ne sont plus nécesssairement fortement connexe.

Algorithme

codegen(L, p):

- Si la liste L ne comporte qu'une seule instruction S et si $p = N_{SS}$, alors retourner S.
- Sinon construire le graphe G dont les sommets appartiennent à la liste L et dont les arcs sont de profondeur $\geq p$.
- Former la liste des cfc de G dans l'ordre du graphe réduit: $\{G_1, \ldots, G_n\}$.
- Pour chaque k, soit $H_k =$ **codegen**$(G_k, p+1)$. Identifier la boucle entourant H_k, décider si elle est parallèle ou séquentielle.
- Retourner le programme composé des H_k en séquence et entourés de leur boucle.

Echange de boucles

Theorem

On peut toujours interchanger une boucle parallèle contenant une boucle séquentielle.

Démonstration.

En effet, l'ensemble des paires critiques est un sous-ensemble des dépendances de la boucle externe. Si celle-ci est parallèle, il n'y a pas de dépendances, donc pas de paires critiques. □

On peut donc, après exécution de l'algorithme de Allen et Kennedy, amener les boucles parallèles au niveau le plus interne.