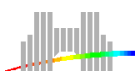
Parallélisation automatique
Méthodes élémentaires de parallélisation

Paul Feautrier

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

18 novembre 2008



PARALLELISATION D'UN BLOC de BASE

Bloc de base : définition

Une séquence d'instruction sans contrôle.

- Si la première instruction est exécutée, alors toutes le sont une et une fois
- Dans un langage structuré :

```
begin S1; S2; ... ; Sn end        { S1; S2; ... ; sN; }
```

- Dans un langage possédant des `GOTO`s, un bloc de base ne doit contenir ni test, ni `GOTO` ni étiquette, sauf en première position
- Mais on peut faire mieux :
  - Construire le graphe de contrôle (aka organigramme) et localiser les arcs séquentiels. On peut fusionner les sommets adjacents à un arc séquentiel
  - On peut agréger une construction complexe (une boucle, un appel de fonction) pourvu que l'on sache calculer les ensembles lus et modifiés
- Propriété fondamentale : dans un bloc de base, on peut identifier opérations et instructions.

GD d'un bloc de base

- Si le bloc de base ne modifie que des scalaires, le calcul du GD est trivial
- Si le bloc modifie des tableaux, il faut d'abord analyser les indices par la méthode de l'exécution symbolique
- On procède ensuite comme pour les scalaires

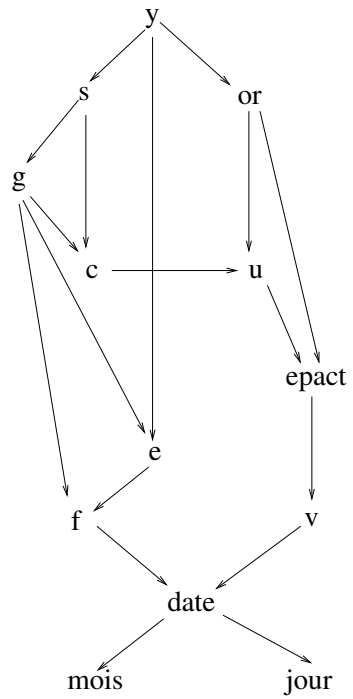
```
S1:   a[i] = 0.;
S2:   i++;
S3:   a[i] = 1.;
```

- Si i_0 est la valeur de `i` à l'entrée du bloc de base, on sait que la valeur en S1 est i_0 et la valeur en S3 est $i_0 + 1$
- Ces deux valeurs ne peuvent être égales, donc il n'y a pas de dépendance
- Dans d'autres circonstances, on peut être amené à prendre des décisions pessimistes.

Observation : le GD d'un bloc de base est *acyclique*.

Exemple

```
read(y);
s = y/100+1;
g = (3*s)/4 -12;
c = (8*s + 5)/25 - 5 - g;
or = y%19 + 1;
u = (11*or + 20 + c)%30;
epact = u + or - 11;
e = (5*y)/14 - g - 10;
f = e/g;
v = 44 - epact;
date = v + 7 - (v + f)%7;
mois = 3 + date/11;
jour = (date - 1)%31 + 1;
```



Programme adapté de J.Arsac

On remarque que le programme est en assignation unique, et donc qu'il n'a a que des dépendances de flot

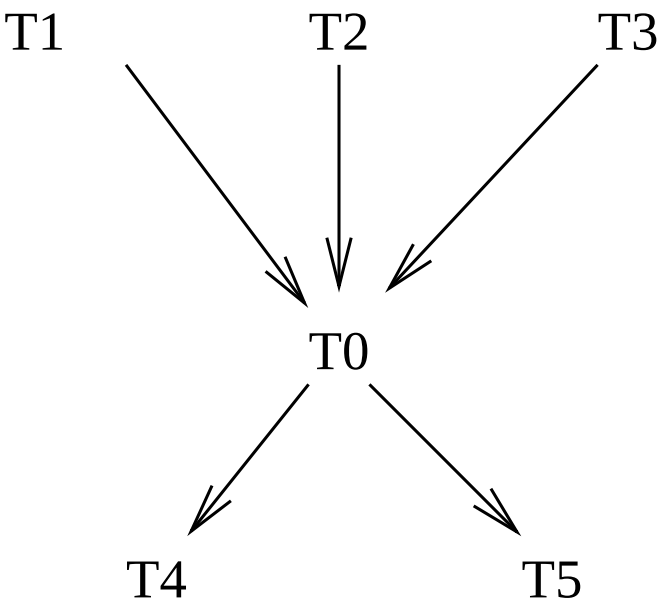
Solution I : le GD est le programme

- Chaque instruction est un processus
- On représentation les arcs du GD comme des opérations de synchronisation
- N'a d'intérêt pratique que si chaque instruction a une durée longue par rapport à une opération de synchronisation
- La taille du programme devient quadratique en le nombre d'instructions !
- Exemple I : une ligne de C et un appel de sémaphore Unix : sans intérêt
- Exemple II : une compilation et une synchronisation par destruction de fichier : intéressant.

Parallélisme au niveau du Shell

Opération de synchronisation :

```
synchro: until rm $1
         do sleep 1
         done
```




```
synchro lock10
synchro lock20
synchro lock30
task
touch lock04
touch lock05
```

Ordonnancemnt

Le principe :

- on suppose que l'on connait la durée d'exécution de chaque instruction (par exemple, toutes de la même durée)
- on connaît le nombre de processeurs
- choisir pour chaque instruction une date de lancement et un processeur sous les contraintes :
  - satisfaire les dépendances
  - n'utiliser que les processeurs disponibles
- on peut ensuite exécuter le programme au moyen d'un échéancier qui lance chaque instruction au moment voulu, ou bien écrire un programme VLIW conforme à l'ordonnancement.

Ordonnancement libre

On suppose que l'on dispose d'autant de processeurs que nécessaire

- Soit $d(k)$ la durée d'exécution de la tâche k
- La date d'activation de la tâche t est donnée par la formule :

$$\theta(k) = \max_{i \in \mathrm{Pred(k)}} \theta(i) + d(i)$$

- Mais le calcul doit être fait dans un ordre compatible avec le graphe de dépendance.

Tri topologique

Trouver un ordre total compatible avec un ordre partiel donné comme fermeture transitive d'un graphe acyclique.

- On associe à chaque sommet un compteur de prédécesseurs
- tant qu'il reste des sommets
  - On sélectionne un sommet dont le compteur est à 0
  - On décrémente les compteurs de ses successeurs
  - On émet le sommet sélectionné et on l'enlève

Il est facile de voir que si on ne trouve pas de sommet sans prédécesseur, le graphe a un cycle.

Ordonnancement sous contraintes de ressources

S'il n'y a qu'un seul type de ressource (par exemple, P processeurs généralistes), on doit assurer qu'à chaque instant, il n'y a pas plus de P taches actives.

- Le problème devient bien plus difficile
- On peut montrer qu'il est *NP-complet* dès que $P > 2$
- Mais il existe un algorithme approché efficace

Algorithme exact, I/II

- On calcule une borne supérieure L de la durée de l'ordonnancement (par exemple la somme des $d(i)$)
- Pour chaque instruction S_i, on introduit L variables 0 / 1, X_{it} telles que $X_{it} = 1$ ssi l'exécution de S_i commence à l'instant $0 \leq t < L$. Pour simplifier, on suppose que la durée de toutes les instructions est 1 cycle
- La date de lancement de S_i est $\theta(i) = \sum_t tX_{it}$.

Algorithme exact, II/II

Contraintes :

- Chaque instruction n'est exécutée qu'une fois : $\sum_t X_{it} = 1$.
- Si R est l'ensemble des instructions utilisant un type de ressource dont il existe N_R exemplaires, alors $\forall t : \sum_{S_i \in R} X_{it} \leq N_R$
- Si S_i précède S_j dans le GD, $\theta(S_i) + 1 \leq \theta(S_j)$.
- La fonction à minimiser est $\max_i \theta(S_i) + d(i)$.
- C'est un problème de programmation linéaire en nombres entiers, qui fournit directement la solution optimale, mais qui ne peut être résolu que pour des petits programmes.

Algorithme de liste, principe

Simuler le fonctionnement d'un ordonnanceur *glouton*. On gère :

- Le temps
- Les instructions en cours d'exécution, avec leur date de terminaison : e
- La liste des instructions exécutables : r
- La liste des ressources libres : L

Algorithme

$t := 0$; $e := \emptyset$;
$r :=$ les instructions sans prédécesseurs dans le graphe de dépendance;
while *il reste des instructions à ordonnancer* **do**
    Let $f :=$ *la liste des instructions qui se terminent à l'instant* t;
    libérer les ressources utilisées par les instructions de f ;
    décrémenter le nombre de prédécesseurs des successeurs des instructions de f et ajouter les instructions sans prédécesseurs à la liste r;
    foreach $i :=$*instruction de* r **do**
        if *il y a assez de ressources pour* i **then**
            calculer la date de terminaison de i et l'ajouter à e;
            décompter les ressources utilisées par i
        end
    end
    $t := t + 1$;
end

Priorités

- L'ordre dans lequel les instructions prêtes sont sélectionneées influe sur la qualité de l'ordonnancement
- On utilise un système de priorités, mais comme le problème est NP-complet, il n'y a pas de meilleur choix universel
- On se base en général sur un ordonnancement sans contraintes de ressources, qui peut se faire en temps linéaire (algorithme du tri topologique), au plus tôt ou au plus tard. Il existe de nombreuses règles de priorité :
  - priorité à l'instruction la plus longue
  - priorité à l'instruction ayant la date au plus tard la plus faible
  - priorité à l'instruction ayant la plus faible mobilité (donc priorité au chemin critique)
- Aucune de ces règles n'est uniformément la meilleure

Garantie, I

- L'algorithme de liste est *glouton* en ce sens qu'aucune ressource ne reste oisive s'il existe une instruction prête qui peut l'utiliser
- Dans le cas particulier où il n'y a qu'un seul type de ressources (par exemple des processeurs identiques) on peut montrer que le résultat de l'algorithme de liste est au plus deux fois plus mauvais que l'optimum.
- Si on associe à chaque arc la durée de l'instruction source, alors la durée du plus long chemin représente le temps d'exécution sur un ordinateur ayant une infinité de ressources, T_∞
- La somme des durées des instructions donne le temps d'exécution séquentiel, T_s.
- Soit L la durée obtenue par l'algorithme de liste, et T_{opt} le temps d'exécution optimal.

Garantie, II

- Le temps d'exécution optimal sur P processeurs ne peut être inférieur ni à T_s/P ni à T_∞ :

$$\max(T_s/P, T_\infty) \leq T_{\text{opt}} \leq L$$

- L'exécution selon l'algorithme glouton se décompose en périodes où toutes les ressources sont actives, et d'autres ou certaines ressources sont oisives. La somme des durées des périodes d'activité ne peut dépasser T_s/P
- On va montrer que $L \leq T_\infty + T_s/P$. Il en résulte :

$$L/T_{\text{opt}} \leq \frac{T_\infty + T_s/P}{\max(T_s/P, T_\infty)} \leq 2$$

Garantie, III

- Soit I_1 une instruction qui se termine à l'instant $t_0 = L$ et qui commence à l'instant $t_1 < t_0$.
- Soit I_2 l'instruction qui "libère" I_1, i.e. parmi les prédécesseurs immédiats de I_1, l'instruction qui se termine le plus tard. I_2 se termine à l'instant t_2 et a commencé à l'instant t_3.
- Ou bien $t_2 = t_1$, ou bien pendant l'intervalle $[t_2, t_1]$ tous les processeurs étaient actifs.
- On continue jusqu'à atteindre une instruction sans prédécesseur. Soit t_{2n+1} sa date de départ. Dans l'intervalle $[0, t_{2n+1}]$ tous les processeurs sont actifs.
- Donc, si on additionne les intervalles pair / impair, on trouve une durée bornée par T_s/P, et si on additionne les intervalles impair / pair, on trouve la durée d'un chemin du graphe de dépendance, bornée par T_∞. D'où le lemme.

Algorithme Inverse

On peut exécuter l'algorithme de liste "à l'envers"

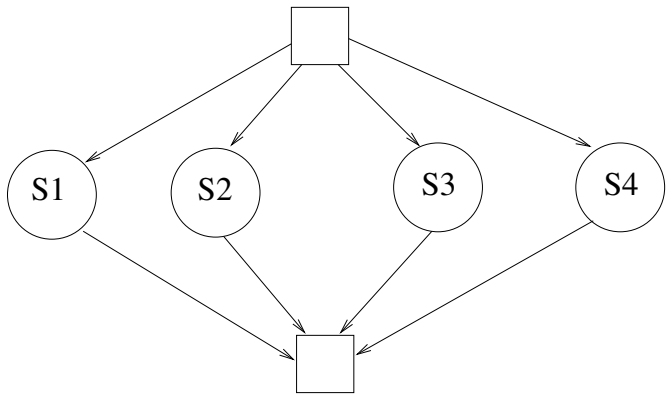
- Il suffit pour cela d'inverser le sens des arcs du graphe de dépendance.
- L'ordonnancement obtenu, exécuté en sens inverse, est un ordonnancement légal.
- On ne sait pas d'avance quel est le meilleur ordonnancement.
- Quelques compilateurs calculent les deux ordonnancements et choisissent le meilleur.

Solution III : Compilation série / parallèle

De nombreux systèmes offrent un opérateur de composition parallèle. Notations variées

```
  S1 // S2 // S3;
cobegin S1; S2; S3 coend (Algol)
{| S1; S2; S3; S4 |}     (Earth C)
  PARALLEL SECTIONS      (OpenMP)
    S1
  PARALLEL
    S2
    ...
  END PARALLEL SECTION
```

Primitives `fork` et `wait` d'Unix



- La taille du programme n'augmente pas
- Il est facile d'estimer la durée du programme par la règle *MaxPlus*

Graphes série / parallèle

Observations :

- Les instructions d'un `cobegin ... coend` ont même ensemble de prédécesseurs et même ensemble de sucesseurs
- Arc séquentiel : un arc dont la source n'a qu'un seul successeur et le but qu'un seul predécesseur. Dans le graphe d'exécution, l'arc qui représente la composition `S1 ; S2` est séquentiel
- D'où l'algorithme.

Algorithme

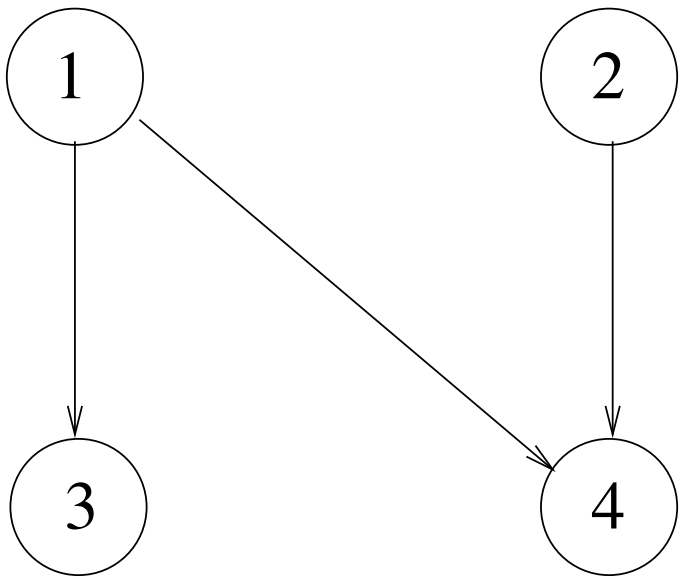
Soit $\approx$ la relation d'équivalence

$$x \approx y \Leftrightarrow \mathrm{Pred}(x) = \mathrm{Pred}(\mathrm{y}) \ \ \& \ \ \mathrm{Succ}(x) = \mathrm{Succ}(\mathrm{y}).$$

while *le GD n'est pas réduit à un point* **do**
    Construire *les classes d'équivalence de la relation* $\approx$*;*
    contracter chaque classe en un `cobegin coend`*;*
    détecter les arcs séquentiels et fusionner les deux sommets adjacents;
end

Contre-exemple

Problème : il arrive que l'algorithme ne se termine pas. Il existe des graphes qui ne sont pas série / parallèle.



Tri topologique, II

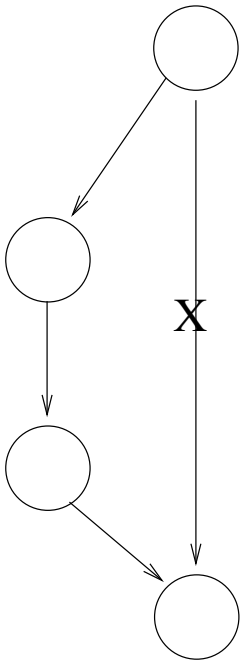
Version améliorée :

- On associe à chaque sommet un compteur de prédécesseurs
- tant qu'il reste des sommets
  - On sélectionne tous les sommets dont le compteur est à 0 et on construit un `cobegin coend`
  - On décrémente les compteurs de tous leurs successeurs
  - On enlève les sommets sélectionnés

Optimal si toutes les tâches ont la même durée et s'il y a suffisamment de processurs
Sinon, c'est au système exécutif de gérer la pénurie

Améliorations

Un arc est inutile s'il peut être reconstruit par transitivité.



- Tant qu'un point fixe n'est pas atteint
  - Pour tout arc
  - Existe-t-il un chemin de la source de l'arc vers son but ne passant pas par l'arc ?
  - Si oui, supprimer l'arc.

Peut se combiner avec l'algorithme série / parallèle. N'a aucun intérêt pour le tri topologique.

Traitement des tests

Que faire si le bloc de base contient des tests bien structurés ?

- Aggréger les deux branches du test
- Ou bien, transformer les dépendances de contrôle en dépendances ordinaires par *if conversion*

```
if(p)
   S1;
else
   S2;
```

```
boolean beta;
beta = (p == 0);
if(beta) S1;
if(!beta) S2;
```

- En matériel, les instructions S1 et S2 sont sous prédicat
- Spéculation : exécuter en parallèle le test et les deux branches ; valider les résultats de la branche sélectionnée par le test.

PARALLELISATION D'UN NID DE BOUCLES

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).

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.

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 rang p est parallèle, le terme correspondant dans l'ordre d'éxécution (qui commence par $p-1$ égalités) a disparu.
- Il ne doit donc pas y avoir de dépendance de profondeur $p-1$.
- 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 rang p est parallèle si son corps de boucle n'a aucune dépendance de profondeur $p - 1$.

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.

Exemple III

```
k = 0;                          |
for(i=0; i<n; i++){             |  for(i=0; i<n; i++)
   a[k] = 0;                    |      a[3*i] = 0;
   k += 3;                      |
}                               |
```

Séquentiel

Parallèle

TRANSFORMATIONS DE PROGRAMMES

Transformations de programme

- Les transformations de programme ont été inventées dans un but d'optimisation
- Certaines sont toujours bénéfiques : éviter de faire plusieurs fois le même calcul ou simplifier une expression algébrique
- Mais très souvent une optimisation détruit le parallèlisme (voir les exemples précédents)
- Donc, chercher à inverser les transformations optimisantes
- 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
- En principe, si on transforme un programme, il faut prouver que les deux versions sont équivalentes. On essaie de mettre cette démonstration "en facteur".

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.

Eclatement d'instruction

- Une instruction trop complexe peut être difficile à paralléliser parce qu'elle engendre beaucoup de dépendances :

```
for(i=0; i<n; i++)
    s = s + a[i]*b[i];
```

- On peut séparer la multiplication et l'addition en introduisant un temporaire :

```
for(i=0; i<n; i++){
    tmp[i] = a[i]*b[i];
    s += tmp[i];
}
```

- puis éclater la boucle et parallèliser la multiplication
- Attention, ceci ne marche que si le temporaire est un tableau
- Indispensable pour la vectorisation.

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.

Observation

Les ensembles critiques à examiner pour savoir si deux boucles peuvent être échangées :

$$\{\mathcal{M}(S,i,j) \cap \mathcal{M}(S,i',j') \mid i' < i \ \ \& \ \ j' < j\} = \emptyset$$

etc., sont inclus dans ceux qui permettent de savoir si la boucle la plus externe est parallèle :

$$\{\mathcal{M}(S,i,j) \cap \mathcal{M}(S,i',j') \mid i' < i\} = \emptyset$$

Theorem

Dans un nid de boucle parfait, une boucle parallèle peut toujours être amenée en position la plus interne, puis être vectorisée.

D'où le succés des super-ordinateurs vectoriels des années 80 jusqu'à nos jours.

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.

Expansion de scalaire

- Remplacer un scalaire modifié par un tableau indexé par le compte-tour de la boucle que l'on veut paralléliser : suppression des dépendances PP

  `tmp = ...` => `Tmp[i] = ....`

- Mais il faut reconstituer le flot des données en ajustant les lectures
- Règle : Une lecture de `tmp` *avant* la première écriture doit être remplacée par `Tmp[i-1]`, ce qui fait échouer la parallélisation
- Une lecture de `tmp` *aprés* la première écriture doit être remplacée par `Tmp[i]`
- Prendre des précautions si `tmp` est utilisé après la sortie de la boucle.

Exemple

On reprend l'exemple :

```
for(i=0; i<n; i++)
    s = s + a[i]*b[i];
```

Eclatement d'instruction :

```
for(i=0; i<n; i++){
    tmp = a[i]*b[i];
    s = s + tmp;
```

Expansion de scalaire :

```
for(i=0; i<n; i++){
    tmp = a[i]*b[i]; ==========>  Tmp[i] = a[i]*b[i];
    s = s + tmp;                  s = s + Tmp[i];
```

Il serait inutile d'expanser s. Pourquoi ?

Mise en assignation unique

- Généralise l'expansion de scalaire. On utilise la méthode de calcul des dépendances directes vue précédemment
- 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) où le nom de l'instruction est remplacé par le tableau associé
- 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 statique (la forme SSA) où on se contente d'un renommage et où le calcul des sources est approximatif.

ALGORITHME DE ALLEN ET KENNEDY

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 boucles originales
- 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 :

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

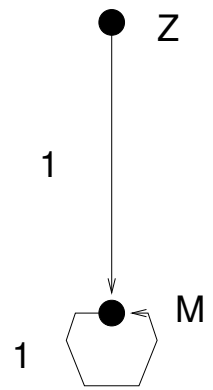
si p est inférieur au nombre de boucles englobant S et T, ou bien :

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

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 profondeur 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 descendants) ...
- ... 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.

Récursion

- Le programme ainsi obtenu satisfait les dépendances de profondeur 0 et les dépendances entres cfc. On peut donc les effacer.
- Pour construire les boucles 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 connexes.

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 sous-graphe G du GD 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, qui est parallèle si G ne comporte pas d'arc de profondeur p.
- Retourner le programme composé des H_k en séquence et entourés de leur boucle.

Complexité

- L'algorithme de Tarjan est linéaire en le nombre d'arcs
- A chaque niveau, on traite un ou plusieurs graphes dont le nombre total d'arcs est au plus égal au nombre d'arcs du GD initial
- Le nombre de niveaux est égal à la profondeur P maximum des nids de boucles
- La complexité est donc $O(P|GD|)$.