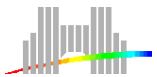
Master Informatique Fondamentale - M1 Compilation

Ordonnancement d'instructions

Paul Feautrier

ENS de Lyon
Paul.Feautrier@ens-lyon.fr
perso.ens-lyon.fr/paul.feautrier

1er mai 2007

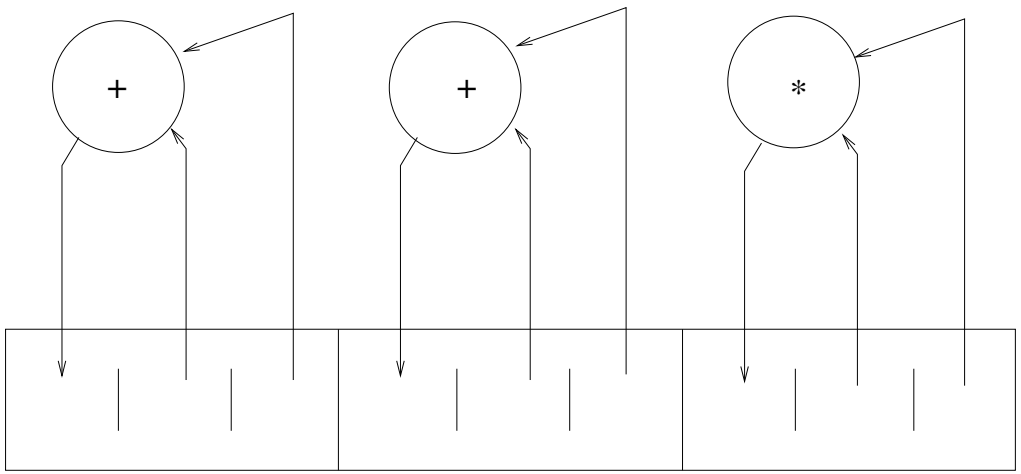


Qu'est-ce que l'ordonnancement

- Tous les processeurs modernes sont paralléles. Ce parallélisme peut être :
  - invisible : le programmeur peut raisoner comme si le processeur était séquentiel. Exemple : le Pentium
  - visible : le programmeur ou le compilateur doit exploiter le parallélisme. Exemple : l'Itanium
- Dans un ordinateur parallèle, la durée totale dépend de l'ordre des instructions
- Ordonnancer, c'est trouver le meilleur ordre possible

Mécanismes, I

Processeur à instructions larges (VLIW) : dans chaque instruction il y a une tranche par unité fonctionelle indépendante.



Le compilateur est responsable du remplissage des tranches de l'instruction.

Mécanisme, II

Ordinateur superscalaire ; le processeur lit plusieurs instructions à la fois (fenètre de 2 à 4 instructions). Il lance l'exécution des instructions

- dont les opérandes sont disponibles
- qui peuvent être exécutées par un opérateur disponible

Parallélisation dynamique par le matériel, mais le compilateur peut aider.

Un exemple

```
    float x, y, z, t;
    float a, b, c, d;

1:  y = z+t;
2:  x = x+y;
3:  b = c*d;
4:  a = a*b;
```

- Un processeur ayant un additioneur et un multiplieur en un cycle, et une fenètre d'activation de 2 positions.
- Le programme prend 4 cycles
- Si on interchange 2 et 3, le programme prend deux cycles
- Est-ce permis ?

Les données du problème, I

Deux instructions sont indépendantes si on peut les intervertir sans changer le résultat du programme.

```
// pas indépendantes
  x = 1;
  x = 2;
  print(x);
//---------------
  x = 1;
  y = 2
// pas indépendantes
  y = 2*y;
  x = y*y;
  print(x);
// indépendantes
  x = 1;
  y = 2;
```

En pratique, deux instructions sont dépendantes si elles partagent une variable et si l'une au moins la modifie

Les données du problème, II

- Chaque instruction utilise une (ou plusieurs) ressource(s) : additioneur, multiplieur, port mémoire, etc.
- Chaque instruction dure un certain nombre de cycles : son résultat n'est disponible que lorsqu'un certain nombre de cycles se sont écoulés.
- Une ressource normale est occupée pendant toute la durée de l'instruction
- Une ressource *pipelinée* peut accepter une opération par cycle, mais le résultat n'est disponible qu'au bout de plusieurs cycles.

Construction du graphe de dépendance

- Pour chaque instruction i, collecter l'ensemble $R(i)$ des valeurs lues, et l'ensemble $M(i)$ des valeurs modifiées.
- Il y a une dépendance de i vers j si i estexécutée avant j et si l'un des trois ensembles $M(i) \cap R(j)$, $R(i) \cap M(j)$ ou $M(i) \cap M(j)$ est non vide.
- Le calcul est facile car les ensembles en question sont de petite taille
- La complexité est cependant quadratique en la taille du programme.

Variations sur le graphe de dépendance

- On peut ne tenir compte que des dépendances Producteur-Consommateur : $M(i) \cap R(j)$ non vide. Les autres dépendances doivent être éliminées par renommage, soit à la compilation (forme SSA) soit à l'exécution (algorithme de Tomasulo)
- On peut introduire des dépendances de contrôle : une instruction interne à une conditionnelle ne peut être exécutée que quand on connait le résultat du test
- on peut aussi les ignorer, soit parce que l'on travaille par Bloc de Base, soit par *spéculation*

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;
$L :=$ toutes les ressources;
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

Lemme de Brent.

- 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.

Algorithme 0/1

Une solution exacte.

- On calcule une borne supérieure L de la durée de l'ordonnancement (par exemple par un algorithme de liste).
- Pour chaque instruction S, on introduit L variables 0 / 1, X_{St} telles que $X_{St} = 1$ ssi l'exécution de S commence à l'instant $0 \leq t < L$. Pour simplifier, on suppose que la durée de toutes les instructions est 1 cycle
- Contraintes :
  - Chaque instruction n'est exécutée qu'une fois : $\sum_t X_{St} = 1$.
  - Si R est l'ensemble des instructions utilisant un type de ressource dont il existe N_R exemplaires, alors $\sum_{S \in R} X_{St} \leq N_R$
  - La date de lancement de S est $\theta(S) = \sum_t tX_{St}$. Si S précède T dans le GD, $\theta(S) + 1 \leq \theta(T)$.
  - La fonction à minimiser est $\max_S \theta(S)$.
- 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.

Ordonnancement global, le problème

Comme l'algorithme de liste est un algorithme de simulation, il est impossible de l'exécuter en présence de boucles (on ne connait pas en général le nombre de tours).

- On va essayer d'étendre l'algorithme au cas d'un DAG.
- On ne traitera que les corps des boucles les plus internes (qui sont des DAGs) par des techniques de type déroulage.
- Pour des solutions plus évoluées, voir le cours de M2.

Ordonnancement d'un DAG

Qu'est-ce que le graphe de dépendance d'un DAG ?

- Les tests lisent des variables (mais en général n'en modifient pas), il faut en tenir compte dans le calcul des dépendances
- En principe – voir plus loin – on ne peut pas intervertir une instruction et le test qui la commande. On doit introduire des *dépendances de contrôle*
- Enfin, il n'y a pas de dépendance entre deux instructions situées dans les branches opposées d'un test, parce qu'elles ne sont jamais présentes toutes les deux dans une même exécution du programme.

Algorithme d'ordonnancement, I

Première solution : on exécute l'algorithme de liste normalement.

- On obtient un code où toutes les instructions sont exécutées, même si elles sont dans la branche fausse d'un test
- Pour rétablir la sémantique du programme, il faut garder chaque instruction par le résultat du (ou des) test(s) qui la commande(nt).
- Comme le test a été ordonnancé avant l'instruction (dépendance de contrôle), la valeur de la garde est disponible
- Le code résultant ne peut être exécuté efficacement que sur un processeur à prédicats (Itanium).

Algorithme d'ordonnancement, II

On ordonnance indépendemment les deux branches d'un test.

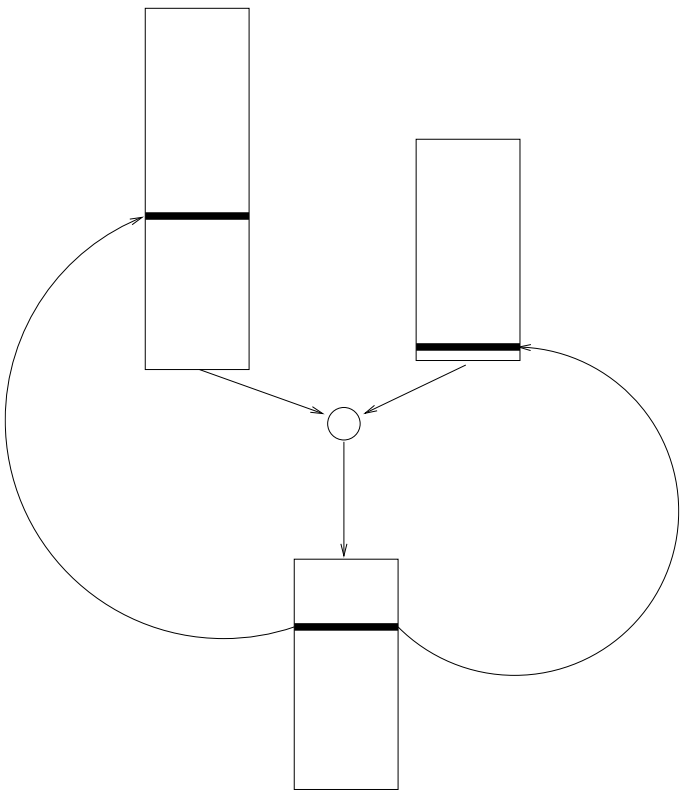
- Quand on rencontre un test, on le génère, puis l'ordonnanceur se duplique (mécanisme analogue à *fork*), et chaque copie traite une branche
- Quand on termine un test, l'un des ordonnateurs se termine et l'autre se réinitialise, ce qui a pour effet d'empécher des instructions de remonter au dessus du point de jonction.

Ces deux méthodes ont l'inconvénient de trop contraindre l'ordonnancement.

Trace Scheduling

- On détermine le chemin le plus probable dans le DAG (la trace) soit par analyse du programme, soit par profilage
- La trace est un bloc de base, que l'on ordonnance de façon standard, mais sans tenir compte des points de jonction.
- On détermine une deuxième trace dans ce qui reste du programme (cette trace peut comporter plusieurs blocs de base) et on l'ordonnance
- Et on continue jusqu'à épuisement du DAG.

Déplacement de code, clonage



Si une instruction est remontée au dessus d'un point de jonction, le code devient incorrect.

- Pour corriger, il faut dupliquer l'instruction en cause dans l'autre branche du test avant de l'ordonnancer.
- Pour éviter cette opération, il faut transformer le DAG en arbre par clonage (un arbre n'a pas de points de jonction).

Ordonnancement de boucles, I

Les méthodes précédentes ne peuvent pas être appliquées à une boucle :

- Chaque instruction peut être exécutée plusieurs fois
- Le graphe de dépendance n'est plus un DAG

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

Première méthode : ordonnancer le corps de la boucle.

- Le corps de la boucle la plus interne est un DAG
- Mais le corps de boucle peut ne pas saturer les unités disponibles

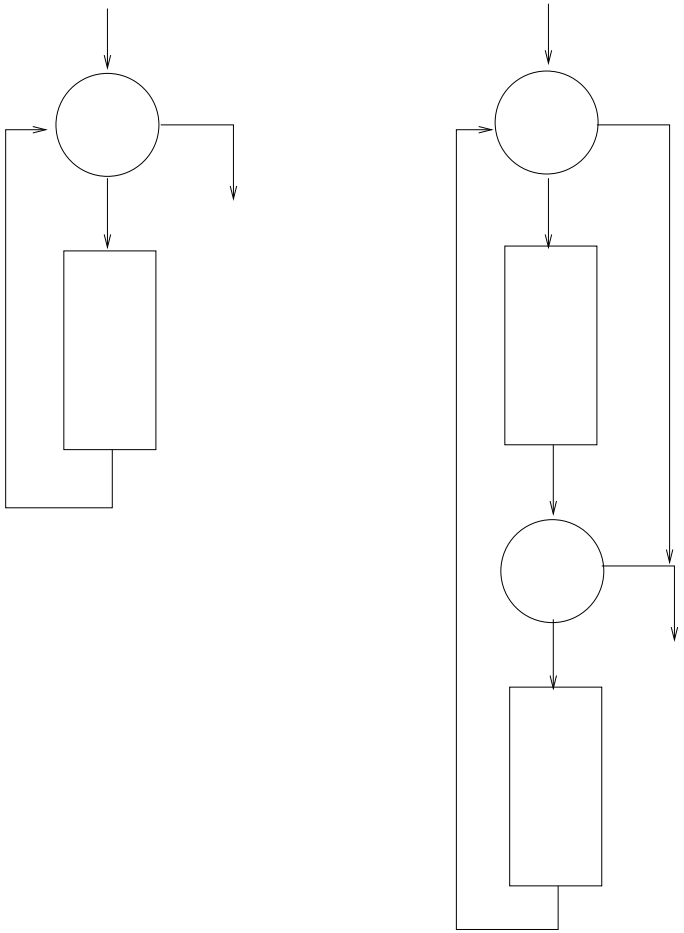
Ordonnancement de boucles, II

Dérouler la boucle

- Possible seulement si le nombre de tour est connu d'avance
- La boucle devient un DAG, et dans certains cas un bloc de base, que l'on ordonnance par les méthodes précédentes

Ordonnancement de boucles, III

Déroulage partiel : on exécute les itérations de la boucle deux par deux, ou trois par trois, etc.



- On peut ajuster le degré de déroulage pour obtenir les perfomances désirées
- En contrepartie, la taille du code augment considérablement

Ordonnancement à l'exécution

- Certains processeurs (haut de gamme) sont capables d'ordonnancer le code pendant l'exécution (processeurs Out Of Order)
- Le processeur calcule dynamiquement le graphe de dépendance (uniquement sur les regsitres)
- Il applique ensuite directement l'algorithme de liste
- L'ordonnancement se fait sur une fenètre de quelques dizaines d'instructions.
- L'ordonnancement est interrompu par la présence d'un test
- Le processeur peut parfois renommer les registres pour éliminer les dépendances autres que Producteur-Consommateur.