DLST L1 — Travaux Pratiques
Université Grenoble Alpes — INF202 2017/2018

Projet 1: relation d'ordre et optimisation d'exécution

L'objectif du projet est placer au mieux les instructions d'un programme sur un processeurs multi-coeurs à l'aide d'un algorithme qui construit une relation d'ordre entre instructions et qui place sur les coeurs les instructions minimales pour la relation d'ordre

1 Les programmes et leur exécutions séquentielles

Dans toute la suite, on note v le nombre de variables, i le nombre d'instructions et ℓ le nombre de lignes du tableau placement.

L'objet d'entrée est une liste d'instructions appelée Dprog.
D'une manière générale chaque instruction est écrite sous la forme

variable := constante + autres_opérandes

c'est-à-dire une expression est une somme dont les opérandes sont

- une constante entière
- jusqu'à 3 variables

Cette forme permet de les représenter dans un tableau à 6 colonnes, indexé par le numéro d'instruction.
La première colonne donne le nombre de colonnes significatives, compris entre 3 et 6 (chaque instruction peut comporter entre 1 et 4 opérandes.
La seconde colonne donne le numéro de la variable affectée, par exemple 3 pour x3
La troisième colonne est la constante en premier opérande de la somme.
Les 3 dernières colonnes donnent des numéros de variables, ou un entier arbitraire non significatif. Par exemple l'instruction ci-dessus est codée par 5, 4, 52, 13, 0, -1. La valeur -1 est arbitraire, car le 5 au début indique que seules les 5 premières valeurs sont significatives.

2 Relations de dépendance

L'objectif est de paralléliser l'exécution d'une séquence d'instructions en faisant en sorte d'obtenir un résultat identique à l'exécution séquentielle. Le modèle d'exécution parallèle considéré est le suivant :

- la mémoire du processeur est partagée par tous les coeurs
- l'exécution de l'ensemble est synchrone

Cette dernière propriété signifie qu'à chaque temps, chaque coeur considère une unique instruction

```
var := constante + autres_opérandes
```

il lit les entiers contenus les variables de autres_opérandes, calcule leur somme additionnée à la constante, puis écrit le résultat dans la variable en membre gauche.
Pour simplifier le problème on considère que l'on n'a pas de limitation sur le nombre de coeurs disponibles.

Pour un D-programme séquentiel de n instructions, il n'est pas utile d'avoir plus de n coeurs. Cela correspondrait au cas où toutes ses instructions peuvent s'effectuer simultanément. Par exemple :

$$\begin{array}{l} \text{x0} = 1 \\ \text{x1} = 3 \\ \text{x2} = 0 \\ \text{x4} = 1 \end{array} \longrightarrow \; x0 = 1 || x1 = 3 || x2 = 0 || x4 = 1$$

En revanche : d'autres programmes ont besoin de rester au moins partiellement séquentiels

$$\begin{array}{l} \text{x0} = 1 \\ \text{x1} = 3 + \text{x0} \\ \text{x2} = 2 + \text{x0} \\ \text{x3} = 1 + \text{x0} \end{array} \longrightarrow \; x0 = 1; x1 = 4; x2 = 3; x4 = 2$$

Il y a trois situations forçant une dépendance entre deux instructions i et j :

1. Une instruction modifie une variable utilisée par une autre.

   i: y:=x
   j: x:=1

   L'ordre des instructions i et j doit doit être préservé. Puisque dans le programme séquentiel, i est avant j, on le traduit par $i < j$.

2. Lorsqu'une instruction nécessite le résultat d'une autre.

   i: x:=1
   j: y:=x

   L'ordre des instructions i et j doit doit être préservé. On le traduit par $i < j$.

3. Lorsque les deux instructions modifient la même variable.

i: x:=1
j: x:=2

L'ordre des instructions i et j doit doit être préservé. On le traduit par $i < j$.

Notez que les 3 situations indiquées correspondent à des comparaisons immédiates. Or on peut avoir, par exemple, $i < j$ et $j < k$, causant indirectement $i < k$.

Étant donné une entrée (un D-programme codé sous forme de tableau comme indiqué ci-dessus) on analyse le D-programme et produit un premier tableau *Avant* représentant la relation de dépendance immédiate. On applique la convention décrite ci-dessous.
On utilisera une fonction *progDep(prog)* qui prend comme arguments le tableau *Dprog* et retourne la relation d'ordre *Avant*.

Pour cela on parcours le programme en cherchant les dépendances immédiates pour chaque instruction et en remplissant *Avant*.

Ensuite, on calcule la clôture transitive de *Avant*.
Pour cela, on codera les fonctions compositions, union et identité, et on calculera

$$\bigcup_{k=0}^{i} \text{avant}^k$$

Maintenant, on remplis le tableau placement avec une exécution A.S.A.P. : pour cela, on construit un tableau de booléen *Placee* de taille $i \times i$ initialisé à Faux.
On répète, tant que ce tableau est non vide (Bonus : trouver une manière fiable d'exécuter ce test sans avoir à relire le tableau en entier à chaque fois),
lister toutes les instructions sans pré-requis dont la valeur "fixée" est fausse.
Les écrire sur une ligne de *Placement*
Fixer leur valeurs à Vrai dans *Placee*.

Placement est une liste de listes, tels que les listes internes ne sont pas toutes de la même taille.

3 Anneau pour ce projet

Vous devrez avoir dans votre réponse un préambule

```
#anneau

(fonctions de l'anneau)

#relations
```

dans lequel vous récupérez les fonctions de l'anneau01, c.a.d. $\mathbb{Z}$

4 Exécution du Programme

L'exécution du D-programme aura lieu en deux étapes. Dans un premier temps, nous considérons l'exécution séquentielle selon le tableau *Dprog* donné comme entrée initiale.
Ensuite, on calcule le tableau *Placement* selon les règles présenté ci-dessus et on exécute les instructions de façon parallèle selon les dépendances des instructions, c.a.d., nous allons parcourir le tableau *Placement* en exécutant les instructions sur chaque ligne.

Dans le deux cas on a besoin d'un tableau *Memoire* qui contiens l'état de la mémoire après l'exécution de chaque instruction. Sa taille dépend de la forme d'exécution.
Pour l'exécution séquentielle le tableau aura une taille $i \times 3v$.
Pour l'exécution parallèle sa taille sera $\ell \times 3v$.
Après avoir exécuté une instruction, on rempli la ligne correspondant dans le format suivant:

| 0 | bvaleur | valeur | 1 | bvaleur | valeur | ... | v | bvaleur | valeur |

Où les valeurs du tableaux alternent entre les variables, un entier fixé à 0 si la valeur est inconnue et 1 sinon, et les valeurs.

Pour l'affichage de l'exécution, on va créer trois fonctions:

1. *affiche1(R)*, qui affiche une relation. Les 0 sont représentés par " ", et les 1 par " $<$ ".
2. *affiche2(T)* qui affiche le tableau *Placement* en remplissant les -1 par " ".
3. *affiche3(M)* qui prend comme argument le tableau *Memoire* et imprime chaque ligne dans le format suivant:

   1: x0: ? x1: ?

   2: [[x0: value]] x1: ?

   3: x0: value [[x1: value]]

   Chaque ligne commence par son numéro et on re-écrit les variables en ajoutant la lettre x pour représenter la variable courante. Nous utilisons "?" pour représenter les valeurs inconnu dans cet état de la mémoire et le valeurs qui vient de changer dans cette exécution seront indiquées entre [[ ]].

4.1 Sortie et impression

Pour l'exécution séquentielle, il faut afficher le tableau *Dprog*, ensuite le tableau *Memoire* dès que l'exécution est terminé.

Pour l'exécution parallèle, il faut afficher le tableau *Dprog*, le tableau *Avant* et les tableaux *Placement* et *Memoire*.

Bonus : Aviez vous vraiment besoin de calculer la clôture de Avant ?
Si non, et si vous ne le faite pas dans le projet, prouvez que ce n'était pas nécessaire.