DLST L1 — Projet 1
Université Grenoble Alpes — INF202 2018/2019

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

Objectif L'objectif du projet est de 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 Représentation des programmes

1.1 Définition

Un programme informatique est un ensemble d'instructions destinées à être exécutées par une machine. Nous écrirons chaque instruction sous la forme

$$\text{variable} := \text{constante} + \text{autres variables}$$

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

- une constante entière
- aucune, une, deux ou trois variables

Exemple Le programme *"Progam 1"* est composé de 4 instructions définissant 4 variables x0, x2, x4 et x7 :

Program 1 :
I_0 : x0 := 91 ;
I_1 : x2 := 59 + x0 ;
I_2 : x4 := 100 + x0 + x0 + x0 ;
I_3 : x7 := 11 + x0 + x2 + x4 ;

Cette forme permet de représenter un programme sous la forme d'un tableau qu'on appellera **Dprog**, tel que :

- L'indice de ligne correspond au numéro de l'instruction. Il y aura donc autant de lignes que d'instructions.
- Chaque ligne comprend 5 colonnes : l'indice de la variable affectée, la valeur de la constante, et les indices des 3 autres variables opérandes. Si l'expression utilisent moins de 3 variables opérandes, les cases associées prendront la valeur -1.

Exemple La représentation de *Program 1* en un Dprog :

```
Dprog1 = [   [ 0,   91,   -1,   -1,   -1],
             [ 2,   59,    0,   -1,   -1],
             [ 4,  100,    0,    0,    0],
             [ 7,   11,    0,    2,    4]
]
```

Par exemple, la représentation de la ligne d'indice 1 du Dprog est :
— Colonne 1 : 2, indice de la variable affectée x2.
— Colonne 2 : 59, valeur de la constante.
— Colonne 3 : 0, indice de la 1ère variable opérande.
— Colonne 4 et 5 : -1, il n'y a qu'une seule variable opérande.

1.2 Préliminaires

Nous allons représenter les programmes par liste d'instructions Dprog. Nous noterons L le nombre d'instructions du programme.

Tâches

1.1 Coder une fonction *readProgram(filename)* qui permet de lire le fichier nommé *"filename"* contenant un tableau tel que spécifié et qui produit une liste de listes python (Dprog).

1.2 Coder une fonction *printProgram(listoflists)* qui prend en argument une liste de liste python et affiche à l'écran le programme comme montré ci-dessus (exemple *Program 1*).

1.3 Tester la lecture et l'affichage des fichiers *"program1.txt"*, *"program2.txt"*, *"program3.txt"* et *"program4.txt"*.

Pour la suite du projet, vous pourrez tester les fonctions à implémenter sur les programmes fournis *"programi.txt"*, $i \in \{1, 2, 3, 4\}$.

2 Relations de dépendance

L'exécution séquentielle d'un programme consiste à exécuter une instruction à la fois, dans l'ordre d'apparition dans le programme. L'objectif du projet 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 itération, chaque coeur considère une unique instruction : il lit les entiers contenus dans les variables existantes, calcule leur somme additionnée à la constante, puis écrit le résultat dans la variable affectée. Pour simplifier le problème on considère que l'on n'a pas de limitation sur le nombre de coeurs disponibles.

Nous écrirons la *liste d'exécution* du programme :

Program 2 :
I_0 : x3 := 1 ;
I_1 : x7 := 3 + x3 ;
I_2 : x5 := 0 + x7 ;
I_3 : x0 := 11 + x5 ;

comme :

Liste exécution *Program 2* :
line 0 = | 0 | * | * | * | * |
line 1 = | 1 | * | * | * | * |
line 2 = | 2 | * | * | * | * |
line 3 = | 3 | * | * | * | * |

où chaque ligne indique l'indice des instructions qui sont exécutées.
Notons que l'instruction I_i dépend de l'instruction I_{i-1}, $i \in \{1, 2, 3\}$. Le programme ne peut donc pas être parallélisé et les instructions sont exécutés une à la fois, dans l'ordre.
Pour le programme suivant,

Program 3 :
I_0 : x0 := 1 ;
I_1 : x1 := 3 ;
I_2 : x2 := 0 ;
I_3 : x3 := 11 ;

la liste d'exécution du programme est

Liste exécution *Program 3* :
line 0 = | 0 | 1 | 2 | 3 | * |

car toutes les instructions peuvent être exécutées en parallèle.

Il y a trois situations forçant une dépendance entre deux instructions I_i et I_j :

1. Lorsqu'une instruction (noté I_{index}) nécessite le résultat d'une autre.

   I_i : x :=1
   I_j : y :=x

   L'ordre des instructions i et j doit doit être préservé. On le traduit par $I_i < I_j$.
2. Lorsque les deux instructions modifient la même variable.

   I_i : x :=1
   I_j : x :=2

   L'ordre des instructions i et j doit doit être préservé. On le traduit par $I_i < I_j$.
3. Une instruction modifie une variable utilisée par une autre.

   I_i : y :=x
   I_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 $I_i < I_j$.

Remarque Notez que les 3 situations indiquées correspondent à des comparaisons immédiates. Or on peut avoir, par exemple, $I_i < I_j$ et $I_j < I_k$, causant indirectement $I_i < I_k$.

Étant donné une entrée (un programme codé sous forme de tableau comme indiqué ci-dessus) on analyse le programme et on produit un premier tableau

Before représentant la relation de dépendance immédiate entre les instructions consécutives. On applique la convention décrite ci-dessous.

Pour le *Program 2* ci-dessus on obtient la relation *Before Program 2* :

	0	1	2	3
0	*	<	*	*
1	*	*	<	*
2	*	*	*	<
3	*	*	*	*

Pour le *Program 3* ci dessus on obtient la relation *Before Program 3* :

	0	1	2	3
0	*	*	*	*
1	*	*	*	*
2	*	*	*	*
3	*	*	*	*

Tâches

2.1 On utilisera une fonction *progDep(listoflists)* qui prend comme argument la liste de listes (Dprog) produite en lisant un fichier et retourne la relation d'ordre *Before* (liste de listes booléenne, ou numpy array). Pour cela on parcourt le programme en cherchant les dépendances immédiates entre chaque deux instructions consécutives. En cas de dépendance immédiate, on associe la valeur 1, sinon la valeur 0. Par convention, la relation *Before* n'est pas réflexive.

2.2 Ensuite, on calcule la clôture transitive de *Before* que nous noterons *After*. Coder les fonctions python vues en TD calculant la clôture transitive (voir exemples ci-dessous).

2.3 Coder *afficheRelation(R)*, qui affiche une relation R (*Before* ou *After*). Les 0 sont représentés par " * " , et les 1 par " < ".

Exemples On doit obtenir *After Program 2* :

	0	1	2	3
0	*	<	<	<
1	*	*	<	<
2	*	*	*	<
3	*	*	*	*

et *After Program 3* (cas particulier ou *After* est égale à *Before*, car déjà transitive) :

	0	1	2	3
0	*	*	*	*
1	*	*	*	*
2	*	*	*	*
3	*	*	*	*

Exécution Séquentielle

Pour faire une exécution séquentielle du programme, il suffit de générer une Relation qui contient des *True* dans la partie supérieure de la diagonale.

Tâches

2.4 Coder *relationSequentielle(L, R)*, qui génère une relation de taille LxL correspondant à l'exécution séquentielle. Cette fonction utilise la clôture transitive R (*After*).

3 Placement à partir de la clôture transitive

Maintenant on remplira un tableau de placement (*Placement*). Le tableau *Placement* est le tableau qui associe les instructions aux processeur. Une ligne représente une étape d'exécution d'instructions sur des multiples processeurs. Chaque colonne représente un processeur qui exécutera une (ou aucune) instruction. Si *Placement*$[i][j]=I_k$ alors l'instruction I_k sera exécuter sur le processeur j à l'étape i. Pour remplir le tableau de placement, nous utiliserons une stratégie d'exécution A.S.A.P (As Soon As Possible). On place l'instruction I_j à la première occasion possible. C'est à dire, dès que toutes les instructions I_k tel que $I_k < I_j$ ont été placées.

On commence par définir un tableau *Placee* de taille N, ou N est le nombre d'instructions totale du programme, initialisé à *False*. *Placee* nous informera des instructions du programme exécutées et celles qui sont en attente.

L'algorithme pour construire le tableau de placement des instructions sur les coeurs à chaque étape d'exécution du programme est le suivant :

- On initialise l'étape d'exécution courante à 0.
- Tant que *Placee* contient des éléments *False*, on répète :
  - On liste toutes les instructions **sans prérequis** dont la valeur dans *Placee* est *False*
  - On les place sur la ligne courante d'exécution de *Placement*, et on modifie la valeur à *True* dans *Placee*.
  - On incrémente l'étape courante d'exécution

On rappelle que le nombre de coeurs disponible n'est pas limité, le nombre de colonnes de *Placement* n'est donc pas le nécessairement le même d'une étape d'exécution à une autre. *Placement* est donc une liste de listes, telle que les listes internes ne sont pas toutes de la même taille.

Tâches

3.1 Coder affichePlacee(*Placee*) qui affiche la liste *Placee* en imprimant 1 pour True et 0 pour False.

3.2 Coder la fonction qui remplit le tableau *Placement* en utilisant la clôture transitive (*After*).

3.3 Coder affichePlacement(*Placement*) qui affiche le tableau *Placement*.

4 Exécution du programme

Nous allons exécuter le programme défini par un objet d'entrée fourni dans un fichier. Après lecture (Sec 1), calcul des relations de dépendance (Sec 2), et le placement dans l'exécution (Sec 3), nous exécuterons les instructions de façon parallèle selon les dépendances des instructions, c.à.d., nous allons parcourir le tableau *Placement* en exécutant les instructions sur chaque ligne.

Pour l'exécution correcte du programme, nous avons besoin d'un tableau *Memoire* qui contient l'État de la mémoire après l'exécution de chaque instruction. La taille du tableau *Memoire* est $(l+1) \times 3N$, ou l est le nombre de lignes du tableau *Placement* (et le $+1$ correspond à l'état initial) et V le nombre de variables définies dans le programme. Par exemple, le *Program 2* définit les variables $(x3, x7, x5, x0)$, donc $V = 4$ et nécessite 4 lignes d'exécution, donc $l + 1 = 5$. Pour le *Program 3*, $V = 4$ et $l + 1 = 2$.

Pour chaque ligne d'exécution (donné par une ligne du tableau *Placement*), on remplit la ligne correspondant dans le tableau *Memoire* avec le format suivant :

i, bvaleur, valeur, j, bvaleur, valeur, ..., k, bvaleur, valeur ...

où les index i, j, k indiquent le nom de la variable (xi, xj, xk), le champ bvaleur vaut 0 si la variable est non initialisé et 1 si la valeur est initialisé ; le champ valeur contient la valeur courante de la variable.

Exemples Memoire

Memoire Program 2 (Taille 5x12) :

3, 0, 0, 7, 0, 0, 5, 0, 0, 0, 0, 0,
3, 1, 1, 7, 0, 0, 5, 0, 0, 0, 0, 0,
3, 1, 1, 7, 1, 4, 5, 0, 0, 0, 0, 0,
3, 1, 1, 7, 1, 4, 5, 1, 4, 0, 0, 0,
3, 1, 1, 7, 1, 4, 5, 1, 4, 0, 1, 15,

Memoire Program 3 (Taille 2x12) :

0, 0, 0, 1, 0, 0, 2, 0, 0, 3, 0, 0,
0, 1, 1, 1, 1, 3, 2, 1, 0, 3, 1, 11,

Tâches

4.1 Coder calculMemoire(Placement) qui prend comme argument le tableau Placement et génère le tableau *Memoire*.

4.2 Coder afficheMemoire(M) qui prend comme argument le tableau *Memoire* et affiche chaque ligne dans le format suivant :

-1 : x0 : ? x1 : ?
0 : [[x0 : value]] x1 : ?
1 : x0 : value [[x1 : value]]

Chaque ligne commence par son numéro (-1 pour l'état initial) et on réécrit les variables en ajoutant la lettre x pour représenter la variable. Nous utilisons " ?" pour représenter les valeurs non initialisés et les valeurs qui sont modifiés dans cette exécution seront indiquées entre [[ ]].

5 Rendu Final et Présentation

Vous devrez rendre les fichiers python qui permettent l'exécution suivante

python execution.py program.txt (option -s)

Le script python doit comporter une option pour une exécution du programme en "séquentiel" (-s), le mode par défaut étant l'exécution en parallèle.

Affichage

- Pour l'exécution séquentielle, il faut afficher le tableau Dprog, puis le tableau *Memoire* (fonction afficheMemoire).
- Pour l'exécution parallèle, il faut afficher le tableau Dprog, le tableau *Before*, le tableau *After* et les tableaux *Placement* et *Memoire*.

Format du rendu Les fichiers python sont à rendre dans une archive zip, à envoyer à votre référent de TP en indiquant les noms du binôme.

Mail des référents

- joanny.perret@grenoble-inp.fr
- jose-ignacio.requeno-jarabo@univ-grenoble-alpes.fr
- mouhcine.mendil@inria.fr
- sergi.pujades-rocamora@inria.fr