DLST L1 — Projet 1
Université Grenoble Alpes — INF202 2019/2020

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

Objectif Le but de ce projet est d'utiliser les notions sur les relations dans le cadre de la programmation parallèle. Il faudra analyser les instructions d'un programme séquentiel (les instructions sont exécutées les une après les autres) pour le transformer en programme parallèle (certaines instructions sont exécutées en même temps, sur un processeurs multi-cœurs). Pour cela il faudra définir une relation d'ordre entre les instructions pour déterminer quelle instruction doit être exécutée avant quelle autre.

Descritif Le projet comporte deux fichiers :
- `tp_operation.py` dans lequel il faut coder une serie d'opérations élémentaires sur les relations.
- `tp_relation.py` dans lequel il faut coder une representation des programmes puis utiliser le fichier précédant pour construire l'ordre d'exécution des instructions.

Vous pouvez réutiliser les fonctions du première fichier pour la seconde partie du projet.

1 Représentation des programmes

1.1 Synthaxe d'un programme (simplifié)

Pour ce projet, on considère qu'un programme est un ensemble d'instructions destinées à être exécutées par une machine. On supposera que chaque instruction est l'affectation d'une variable, dont la valeur est calculée par la somme d'une constante et de la valeur d'autres variables opérandes. Une instruction est donc de la forme :

```
x0 := C + x1 + x2 + ...
```

Exemple Le programme `Progamme_1` est composé de 4 instructions définissant 4 variables `x0, x2, x4, x7` :

```
      Progamme_1 :
I0 :    x0 := 91 ;
I1 :    x2 := 59 + x0 ;
I2 :    x4 := 100 + x0 + x0 + x0 ;
I3 :    x7 := 11 + x0 + x2 + x4 ;
```

inline]version liste de liste (abandonnée)

Cette simplification permet de représenter un programme sous la forme d'un tableau qu'on notera `repr_prog` (représentation d'un programme), tel que :

— Chaque ligne du tableau correspond a une ligne du programme.
Ainsi pour l'indice i, l'instruction I_i correspond à la ligne `repr_prog[i]`.
Et il y aura autant de lignes que d'instructions.

— Chaque ligne comprend 5 colonnes :
`repr_prog[i][0]` est le numéro de la variable affectée,
`repr_prog[i][1]` est la valeur de la constante, et
`repr_prog[i][j]` ($j \in \{2, 3, 4\}$) sont les numéros 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 `Programme_1` est :

```
repr_prog1 = [  [ 0,   91,   -1,   -1,   -1] ,
                [ 2,   59,    0,   -1,   -1] ,
                [ 4,  100,    0,    0,    0 ] ,
                [ 7,   11,    0,    2,    4 ] ]
```

Par exemple, la signification de la première ligne de `repr_prog1` est :
— Colonne 0 : `0`, indice de la variable affectée, c-à-d `x0` ;
— Colonne 1 : `91`, valeur de la constante ;
— Colonne 2, 3, 4 : `-1`, il n'y a pas de variable opérande.
La signification de la deuxième ligne du `repr_prog1` est :
— Colonne 0 : `2`, indice de la variable affectée, c-à-d `x2`.
— Colonne 1 : `59`, valeur de la constante.
— Colonne 2 : `0`, indice de la première variable opérande, c-à-d `x0`.
— Colonne 3, 4 : `-1`, il n'y a qu'une seule variable opérande.
Après l'exécution des deux premières lignes de `repr_prog1`, nous avons `x0 = 91` et `x2 = 59 + 91 = 150`.

1.2 Préliminaires

On représente donc les programmes par la liste d'instructions `repr_prog`.

```
          Programme_2 :
I_0 :   x3 := 1 ;
I_1 :   x7 := 3 + x3 ;
I_2 :   x5 := 0 + x7 ;
I_3 :   x0 := 11 + x5 ;
```

```
          Programme_3 :
I_0 :   x0 := 1 ;
I_1 :   x1 := 3 ;
I_2 :   x2 := 0 ;
I_3 :   x3 := 11 ;
```

Tâches

1.1 Coder les tableaux `repr_prog_1`, `repr_prog_2`, `repr_prog_3`, comme des variables globales.

1.2 Récupérer la fonction pour afficher un tableau de tableau faite dans le TP précédant.

Pour la suite du projet, vous devrez tester les fonctions à implanter sur les programmes fournis.

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. Autrement dit, on veut attribuer à chaque cœur une instruction à exécuter à un temps donné.

Le modèle d'exécution parallèle considéré est le suivant :
— la mémoire du processeur est partagée par tous les cœurs ;
— l'exécution de l'ensemble est synchrone ;
— il y a 5 cœurs disponibles.

La propriété d'exécution synchrone signifie tous les cœurs exécutent leur instruction dans un même temps. À chaque temps, chaque cœur considère une unique instruction : il lit les valeurs contenus dans les variables, calcule la somme, puis écrit le résultat dans la variable affectée. La limitation du nombre de cœurs permet de simplifier le projet. On supposera que, pour tous les programmes considérés, 5 cœurs est plus de suffisant.

On représentera l'attribution des instructions aux cœurs par un tableau qu'on nommera `exec_par` (exécution parallèle) dans lequel chaque ligne correspond à un temps de l'exécution et chaque colonne à un cœur. La valeur de `exec_par[t][c]` est donc l'indice de l'instruction exécutée au temps t par le cœur c. Si un cœur ne fait pas de calcul, la case associée prendra la valeur `-1`.

Exemple

```
exec_par2 = [  [ 0,  -1,  -1,  -1,  -1] ,
               [ 1,  -1,  -1,  -1,  -1] ,
               [ 2,  -1,  -1,  -1,  -1] ,
               [ 3,  -1,  -1,  -1,  -1] ]

 exec_par3 = [  [ 0,  1,  2,  3,  -1] ]
```

Pour le `Programme_2`, notons que l'instruction I_i dépend de l'instruction I_{i-1} (pour $i \in \{1, 2, 3\}$). Le programme ne peut donc pas être parallélisé et les instructions sont exécutés une à la fois, dans l'ordre.

Pour le `Programme_3`, les instructions I_i (pour $i \in \{0, 1, 2, 3\}$) sont indépendantes.Toutes les instructions peuvent être exécutées en parallèle.

Il y a 3 situations forçant une dépendance entre deux instructions I_i et I_j. Dans chaque cas, l'ordre des instructions I_i et I_j doit être préservé. On le traduit par $I_i < I_j$.

```
exec_par2 = [  [ 0,  -1,  -1,  -1,  -1] ,
               [ 1,  -1,  -1,  -1,  -1] ,
               [ 2,  -1,  -1,  -1,  -1] ,
               [ 3,  -1,  -1,  -1,  -1] ]
```

— Lorsqu'une instruction (noté I_i) nécessite le résultat d'une autre :

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

— Lorsque les deux instructions modifient la même variable :

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

— Une instruction modifie une variable utilisée par une autre :

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

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 `repr_prog` comme indiqué ci-dessus) on analyse le programme et on produit une première relation de *dépendance* (dépendance immédiate entre les instructions). La *dépendance* est représentée par sa matrice qu'on nommera `R_dep`. Les indices de lignes et de colonnes correspondent aux indices des instructions (dans l'ordre du programme séquentiel) ; ainsi `R_dep[i][j] = 1` signifie $I_i < I_j$

Exemple Pour le `Programme_ 2` ci-dessus on obtient la relation `R_dep2` et pour le `Programme_3` la relation `R_dep3`.

```
R_dep2 = [  [ 0,  1,  0,  0, ] ,
            [ 0,  0,  1,  0, ] ,
            [ 0,  0,  0,  1, ] ,
            [ 0,  0,  0,  0, ] ]

R_dep3 = [  [ 0,  0,  0,  0] ,
            [ 0,  0,  0,  0] ,
            [ 0,  0,  0,  0] ,
            [ 0,  0,  0,  0] ]
```

Ensuite, on souhaite prendre en compte les dépendances indirectes ; autrement dit, on rend la relation transitive. On calcule donc la clôture transitive de *dépendance* qu'on appelle relation de *précédence* et qui est représentée par sa matrice . On remarque que la *précédence* est une relation d'ordre stricte (antiréflexive, antisymétrique, transitive).

Exemple Pour les `Programme_2` et `Programme_3` on doit obtenir :

```
R_prec2 = [  [ 0,  1,  1,  1] ,
             [ 0,  0,  1,  1] ,
             [ 0,  0,  0,  1] ,
             [ 0,  0,  0,  0] ]
```

```
R_prec3 = [  [ 0,  0,  0,  0] ,
             [ 0,  0,  0,  0] ,
             [ 0,  0,  0,  0] ,
             [ 0,  0,  0,  0] ]
```

Tâches

2.1 Coder le tableau `R_dep1` comme une variable globale.

2.2 Coder la fonction `const_dep(repr_prog)` (construire la *dépendance*) qui prend comme argument la représentation d'un programme (`repr_prog`) et retourne la relation *dépendance* (`R_dep`). Pour cela, parcouror le programme en cherchant les dépendances immédiates entre chaque couple d'instructions. Par convention, la relation *dépendance* n'est pas réflexive.

2.3 Coder la fonction `const_prec(R_dep)` (construire la *précédence*) qui prend comme argument la relation *dépendance* (`R_dep`) et retourne la relation d'ordre strict *précédence* (`R_prec`). Pour cela, utiliser calculer la clôture transitive (à partir des fonctions vu dans le TP précédent).

3 Attribution des instructions aux cœurs

On rempli maintenant le tableau `exec_par`. Le tableau `exec_par` associe les instructions aux cœurs. Pour rappel, `exec_par[i][j]=k` signifie que l'instruction I_k sera exécutée sur le cœur j au temps i. Pour remplir `exec_par`, on utilise une stratégie d'exécution « dès que possible » (*A.S.A.P. : As Soon As Possible*) ; on place l'instruction I_j dès que toutes les instructions I_k la précédant($I_k < I_j$) ont été placées.

On commence par définir un tableau `attrib` (instructions déjà attribuées) initialisé à `False` pour chaque instruction. `attrib[i] = True` signifie que l'instruction i est déjà attribuée à un cœur.

L'algorithme pour construire `exec_par` est le suivant :

- On initialise le temps d'exécution courante à 0.
- Tant que `attrib` contient des éléments `False`, on répète :
  - On liste toutes les instructions pour lesquelles toutes les instructions précédentes sont déjà attribuées ;
  - On les place au temps d'exécution courant dans `exec_par`, et on modifie les valeurs correspondantes dans `attrib` ;
  - On incrémente le temps d'exécution courant.

Tâches

3.1 Coder `affich_attrib(attrib)` (afficher les instructions attribuées) qui affiche la liste `attrib` en imprimant `1` pour `True` et `0` pour `False`.

3.2 Coder la fonction qui remplit le tableau `exec_par` en utilisant la relation de *précédence*.

3.3 Coder `affich_exec(exec_par)` qui affiche le tableau `exec_par`.

4 Exécution du programme

On effectue maintenant l'exécution parallèle du programme, en respectant les dépendances des instructions, c.à.d., en parcourant le tableau `exec_par` ligne par ligne. Pour cela on défini un tableau `mem` (mémoire) qui contient l'état de la mémoire à chaque temps de l'exécution. Pour chaque temps de l'exécution, le tableau `mem` contient une ligne (donc autant de ligne que `exec_par`) plus une ligne qui correspond à l'état initial. Pour chaque variable, `mem` comporte 3 colonnes contenant respectivement :

— le numéro de la variable,
— un booléen indiquant si la variable est initialisée ou non,
— la valeur courante de la variable.

Exemple Par exemple, le `Programme_2` nécessite 4 temps d'exécution et définit les variables `x3, x7, x5, x0`, donc `mem_2` à $4+1$ lignes et 4×3 colonnes. Pour le `Programme_3`, `mem_2` à $1+1$ lignes et 4×3 colonnes.

`mem_2` :
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,

`mem_3` :
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 `calcul_mem(exec_par)` (calculer la trace de la mémoire) qui prend comme argument le tableau `exec_par`et génère le tableau `mem`.

4.2 Coder `affich_mem(mem)` qui prend comme argument le tableau `mem` et affiche chaque ligne dans le format suivant :
— chaque ligne commence par son numéro (-1 pour l'état initial),
— les noms des variables sont écrits suivit des valeurs respective (? pour les variable non initialisées),
— une variable dont la valeur est modifiée au temps courant est signalée par `[[ x ]]`.
Par exemple :

```
-1:   x0: ? x1: ?
0:    [[ x0: v0 ]] x1: ?
1:    x0: v0 [[ x1: v1 ]]
```

4.1 Exécution Séquentielle

On implante ici une option pour une exécution séquentielle. Pour cela il faut faire en sorte que toute paire d'instructions soit en relation (dans un sens ou dans l'autre). Autrement dit, il faut transformer la *précédence* en une relation d'ordre stricte et totale qu'on appelle relation *totale* représentée par sa matrice qu'on nommera `R_tot`.

Tâches

5.1 Coder `const_tot(R_prec)` (construire la relation *totale*), qui génère la relation *totale* (`R_tot`) de la bonne taille.