Parallélisation maximale automatique

Projet pratique en Systèmes d'exploitation

sergiu.ivanov@univ-evry.fr

1 Objectif

Développer une libraire en Python pour automatiser la parallélisation maximale de systèmes de tâches. L'utilisateur doit pouvoir spécifier des tâches quelconques, interagissant à travers un ensemble arbitraire de variables, et pouvoir :

1. obtenir le système de tâches de parallélisme maximal réunissant les tâches en entrée,
2. exécuter le système de tâches de façon séquentielle, tout en respectant les contraintes de précédence,
3. exécuter le système de tâches en parallèle, tout en respectant les contraintes de précédence.

2 Réalisation

Cette section précise les consignes du projet et donne quelques suggestions. Vous pouvez vous écarter de ces suggestions du moment où toutes les fonctionnalités finales énumérées dans la section précédente sont respectées.

2.1 Bibliothèque

Pour ce projet, vous devez produire une *bibliothèque* : un fichier proposant les fonctionnalités sous forme de classes et de fonctions publiques. Le mode de travail recommandé serait donc de créer le fichier contenant la réalisation des fonctionnalités, et d'avoir un fichier de test à côté que solliciterait ces fonctionnalités.

Pour nous donner une image complète de votre travail, vous nous rendrez les deux fichiers à la fin du projet.

2.2 Tâches

Votre bibliothèque peut être contenue dans un seul fichier `.py`, par exemple `maxpar.py`. Elle doit proposer à l'utilisateur des outils pour définir des tâches. Vous pouvez par exemple déclarer la classe suivante :

```
class Task:
    name = ""
    reads = []
    writes = []
    run = None
```

Les significations des champs sont les suivantes :

— `name` : le nom de la tâche, unique dans un système de tâche donné ;

— `reads` : le domaine de lecture de la tâche ;

— `writes` : le domaine d'écriture de la tâche ;
— `run` : la fonction qui déterminera le comportement de la tâche.

Une utilisation de cette classe peut ressembler au code suivant :

```
X = None
Y = None
Z = None

def runT1():
    global X
    X = 1

def runT2():
    global Y
    Y = 2

def runTsomme():
    global X, Y, Z
    Z = X + Y

t1 = Task()
t1.name = "T1"
t1.writes = ["X"]
t1.run = runT1

t2 = Task()
t2.name = "T2"
t2.writes = ["Y"]
t2.run = runT2

tSomme = Task()
tSomme.name = "somme"
tSomme.reads = ["X", "Y"]
tSomme.writes = ["Z"]
tSomme.run = runTsomme
```

Ce code définit 3 objets de la classe `Task` : `t1`, `t2` et `tSomme`, et leur associe les noms, les domaines de lecture et d'écriture, ainsi que les fonctions donnant leurs comportements (`runT1`, `runT2` et `runTsomme`). Pour exécuter ces trois tâches à la main, on peut utiliser le code suivant :

```
t1.run()
t2.run()
tSomme.run()
print(X)
print(Y)
print(Z)
```

2.3 Systèmes de tâches

Un ensemble d'objets de la classe `Task` définie dans la section précédente ne forme pas un système de tâches, car aucune information sur la précédence n'est disponible directement dans un tel ensemble. Les conditions de Bernstein permettent l'identification des paires de tâches interférentes mais il est impossible de savoir en utilisant ces conditions dans quel ordre ces tâches doivent être exécutées.

La procédure de construction d'un système de tâches sera donc paramétrée par les deux objets suivants :

1. une liste de tâches (objets de la classe `Task`),

2. un dictionnaire[1] donnant les contraintes des précédence de départ, c'est-à-dire le système de tâches de départ pour l'algorithme de parallélisation maximale.

Le dictionnaire de précédence contiendra, pour chaque nom de tâche, les noms des tâches par lesquelles elle doit être précédée.

Supposons par exemple que vous réalisiez la construction d'un système de tâches dans le constructeur[2] d'une classe `TaskSystem`. Dans le cas des trois tâches `t1`, `t2` et `tSomme` définies dans la section précédente, cette construction peut s'écrire de la façon suivante :

```
s1 = TaskSystem([t1, t2, tSomme], {"T1": [], "T2": ["T1"], "somme": ["T1", "T2"]})
```

Le dictionnaire de précédence passé en deuxième argument au constructeur précise que la tâche dont le nom est `"T2"` doit s'exécuter après celle qui a le nom `"T1"`, dans le cas où ces deux tâches doivent être ordonnées. De la même façon, la tâche dont le nom est `"somme"` doit s'exécuter après les deux autres tâches, si jamais un ordonnancement est nécessaire.

L'objet `s1` issu de cette procédure de construction doit être un système de tâches de parallélisme maximal. La classe `TaskSystem` doit réaliser au moins les méthodes suivantes :

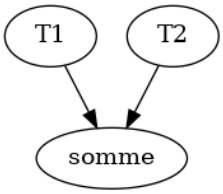
- `getDependencies(nomTache)` : pour un nom de tâche donné, renvoyer la liste des noms des tâches qui doivent s'exécuter avant la tâche `nomTache` selon le système de parallélisme maximal,
- `runSeq()` : exécuter les tâches du système de façon séquentielle en respectant l'ordre imposé par la relation de précédence,
- `run()` : exécuter les tâches du système en parallélisant celles qui peuvent être parallélisées selon la spécification du parallélisme maximal.

2.4 Validation des entrées

Les entrées fournies à la procédure de construction du système de tâches peuvent avoir des défauts : les noms des tâches peuvent être dupliqués, le dictionnaire de précédence peut contenir des noms de tâches inexistantes, peut ne pas être suffisamment complet pour le problème de minimisation donné, peut correspondre à un système de tâches indéterminé, etc. Réalisez un ensemble de vérifications de validité des entrées, en affichant des messages d'erreur détaillés.

2.5 Affichage du système de parallélisme maximal

Rajoutez à votre librairie une fonction qui permettrait d'afficher graphiquement le graphe de précédence du système de parallélisme maximal construit. Si, par exemple, vous réalisez cette fonctionnalité sous forme de méthode `draw` de la classe `TaskSystem`, l'appel `s1.draw()` peut produire l'image suivante :



1. Les dictionnaires, aussi connus sous le nom de table, hash table, map ou hash map dans d'autres langages de programmation, sont des tables associatives, e.g. `https://realpython.com/python-dicts/`.

2. Le constructeur est la méthode `__init__` d'une classe et décrit les actions qui doivent être exécutées à la création de ses objets. Vous pouvez voir un exemple ici : `https://www.w3schools.com/python/gloss_python_class_init.asp`.

2.6 Test randomisé de déterminisme

La vérification du déterminisme par les conditions de Bernstein n'est valable que si les domaines de lecture et d'écriture déclarent correctement l'ensemble des variables lues et écrites par la tâche. Oublier une variable dans cette déclaration peut entraîner une fausse conclusion de déterminisme.

Pour pallier partiellement ce problème, rajoutez à votre bibliothèque la possibilité de tester différentes exécution parallèles du système de tâches avec des jeux de valeurs aléatoires pour les variables. Si pour le même jeu de valeurs deux exécutions parallèles du système produisent des résultats différents, il n'est pas déterminé. Ce test randomisé de déterminisme peut se présenter sous la forme d'une méthode `detTestRnd()` dans la classe `TaskSystem`.

2.7 Coût du parallélisme

Rajoutez à votre bibliothèque la possibilité de comparer les temps d'exécutions séquentielle (`runSeq()`) et parallèle (`run()`) du systèmes de tâches de parallélisme maximal. Cela peut se présenter sous la forme d'une méthode `parCost()` dans la classe `TaskSystem` qui lancera le système des deux manières et qui affichera les différences des temps d'exécution.

Il est connu que les premières exécutions peuvent être plus lentes que les suivantes à cause du temps additionnel nécessaire pour préparer divers caches. Prenez ce fait en compte à la mesure des temps d'exécution. Pour augmenter la qualité des conclusions, exécutez le système plusieurs fois et affichez la moyenne des temps d'exécution.

3 Parenthèse : les variables globales

Ce projet met en avant l'utilisation des variables globales : en effet, la communication entre les tâches se fait uniquement par ce biais. L'utilisation des variables globables est généralement une très mauvaise idée, puisqu'elles engendrent des interférences implicites entre des sous-programmes. Cependant, il est assez fréquent d'avoir des ressources partagées en accès concurrent – une base de données, par exemple –, situation qu'illustrent les variables globales dans ce projet.

4 Organisation du travail et évaluation

Le projet sera évalué de deux façons : sur le code soumis et sur l'exposé de 10 minutes. Le travail en *binôme ou trinôme* est conseillé. Le travail en monôme est déconseillé mais accepté. Le travail dans des groupes de $>$ **3 personnes est interdit**.

4.1 Évaluation du code rendu

Voici la grille d'évaluation proposé à votre chargé de TD :

Fonctionnalité	**Points**
Construction du système de parallélisme maximal	5
Exécution séquentielle	3
Exécution parallèle	4
Affichage	2
Test randomisé de déterminisme	3
Coût du parallélisme	3
Total	20

Votre chargé de TD pourra adapter le barème au niveau global du groupe.

Votre code doit être *soigneusement commenté*. Sans aller jusqu'à expliquer chaque ligne de code individuellement, vos commentaires doivent rendre compte de votre compréhension : pourquoi faites-vous telle ou telle action, pourquoi la faire de cette façon et non d'une autre, etc.

4.2 Évaluation de l'exposé

À la dernière séance de TD, vous ferez un exposé de **10 minutes**, lors duquel vous expliquerez votre solution, les difficultés que vous aurez rencontrées, les façon de les résoudre, etc. Votre compréhension de l'utilité de la parallélisation automatique sera évaluée également.

La limite de temps est stricte. Chaque minute prise au-delà des 10 imparties sera pénalisée d'un **retrait d'un point** par minute à la note sur 20.

Dans le cas d'un écart flagrant d'investissement, l'enseignant·e peut donner des notes différentes aux membres d'un groupe.