Simulation d'exécution de programmes parallèles

Travail d'étude et de recherche.
Rapport.

Olga Pigareva
M1 SIL

16 mai 2019

Table des matières

1 La présentation de la problématique dans son contexte

1.1 Introduction

Le sujet de mon travail d'étude et de recherche aborde un domaine d'actualité dans le monde numérique, la programmation parallèle. La parallélisation de tâches et de processus permet d'augmenter la performance et faire l'exécution d'un programme beaucoup plus rapide. C'est pourquoi la programmation parallèle reste un sujet très actuel, même si elle était inventée il y a déjà plusieurs années.

La programmation parallèle comprend la résolution d'un problème simultanément par plusieurs processus. Cela implique l'utilisation d'une seule ressource par plusieurs processus, ce qui nécessite une synchronisation. De son côté, la synchronisation peut amener à un interblocage qui peut avoir les conséquences indésirables (catastrophiques) pour un programme.
Mon travail fait partie d'une recherche visant à caractériser ces interblocages.
Dans les parties suivantes, j'expliquerai ce que signifient ces termes.

1.2 Programmes parallèles

Les programmes parallèles permettent de créer les activités qui peuvent exécuter les instructions indépendamment. Souvent ces activités ont besoin d'acceder aux mêmes ressources. Pour ne pas avoir des résultats imprévus lors de modification de données par les processus différents, les ressources sont synchronisées. Cela veut dire que les processus peuvent acceder aux données chacun à son tour. Pour attendre la libération de données ou la fin d'instructions d'un processus, un programme parallèle utilise les horloges qui sont les barrières de synchronisation.

1.3 Interblocage

Le problème d'interblocage apparaît quand chaque processus attend un autre pour se terminer et finalement aucun peut acceder à la ressource. Par conséquent, le programme reste bloqué.

Par exemple, sur la Figure 1, le processus P1 attend la fin de processus P2 pour acceder à R1. Et P2, de son côté, attend la fin de P1 pour acceder à R2. Tous les deux processus restent bloqués.

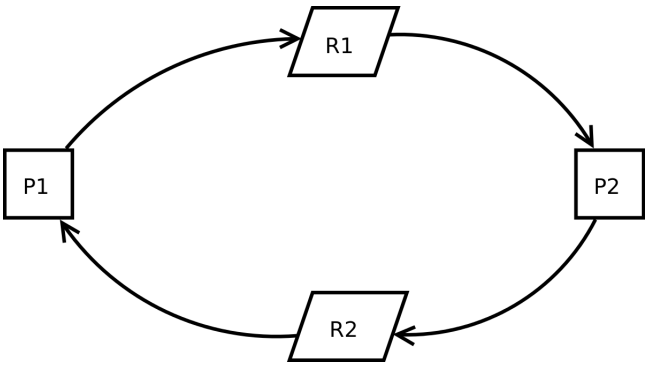


FIGURE 1 – Exemple d'interblocage

Souvent pour les programmes avec plusieurs processus et plusieurs ressources, il est difficile de contrôler l'absence d'interblocages, surtout avec la liberté et puissance offertes par la synchronisation. D'où l'intérêt de reconnaître ces interblocages automatiquement le plus tôt possible, à savoir, éventuellement, lors de la compilation du programme.

1.4 Problème et objectifs

Est-il possible de définir un interblocage pendant l'étape de compilation ?

Pour répondre à cette question, l'idée est de commencer par créer une version simplifiée d'un langage parallèle en prenant un langage X10 comme un exemple. Le compilateur doit reconnaître juste les instructions qu'on a besoin pour créer plusieurs activités parallèles (boucles, conditions, création de nouveaux processus et horloges).
L'exemple de tel programme est représenté par la Figure 2.

```
finish clk1 {
    async S0();
    for i in 0..3
        async clk1 {
            S1(i);
            advance clk1;
            S2(i)
        }
}
```

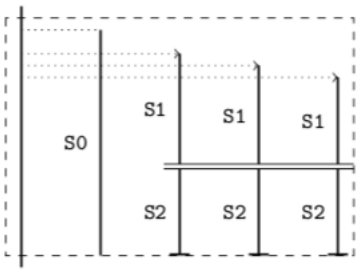


FIGURE 2 – Exemple d'un programme parallèle

Après avoir implémenté un langage qui permet de créer plusieurs activités parallèles avec la synchronisation, il doit être possible de visualiser l'état de toutes les activités au moment d'interblocages. Pour cela, il faut créer une machine virtuelle qui va être capable de simuler l'exécution de programmes parallèles et donner une liste d'activités.

2 Choix d'implémentation et différentes étapes de réalisation du projet

Ci-dessous je vais décrire les principales parties de mon travail et préciser le choix d'implémentation.

2.1 Le langage

Le langage expérimental dont on a besoin est une version simplifiée (avec les instructions de base) d'un langage parallèle de programmation. Sa syntaxe se ressemble à celui du langage parallèle X10, un exemple du programme est représenté sur la Figure 3.

Voici les instructions principales du langages dont on a besoin dans le cadre de cette recherche :

Boucles et Tests :

Pour avoir une possibilité de créer plusieurs activités, le langage doit savoir reconnaître les boucles et vérifier les conditions.
Les seules variables dans ce langage sont les compteurs de boucles.
La syntaxe est la suivante :

- **for i = 0 .. 8** - répéter une instruction pour i de 0 à 8
- **if (i < 7)**

Les instructions de programmation parallèle :

- **finish (clock1, ...)** - créer un block avec les horloges qu'on peut utiliser pour synchroniser les activités
- **async (clock1, ...)** - créer une activité qui peut dépendre d'une clock (p.e. clock1 ici)
- **advance clock1** - créer une barrière de synchronisation sur une clock (clock1 ici)

Les horloges :

Chaque instruction peut contenir une horloge (une clock), car c'est un point important dans la programmation parallèle. Les activités sont indépendantes mais peuvent être enregistrées avec une ou plusieurs horloges communes.

Les horloges créées par *finish* peuvent être utilisées que à l'intérieur de ce bloc. Pour utiliser une horloge dans une nouvelle activité, elle doit être précisée par *async*, cela peut être également une liste de clocks.
Une barrière de synchronisation *advance* peut utiliser une clock à la fois.

Il est important de mentionner qu'à tout moment du programme on sait exactement quelles sont les clocks visibles.

les instructions :

Les instructions ne sont pas importantes dans la recherche, c'est pourquoi seulement les instructions basiques, comme 'S2 ;', sont présentes.

```
finish (cl1, cl2) {
    async (cl2)  {
        for i = 1 .. 2{
            advance cl2;
            S1;
        }
        for j = 0 .. (i/2)
            if (c > 2){
                S3;
            }
    }
    S1;
    advance cl2;
}
```

FIGURE 3 – Exemple d'un programme parallèle

2.2 Le compilateur

Les trois parties importantes de la compilation ce sont : un analyseur lexical, un analyseur syntaxique, qui produit en sortie un arbre syntaxique abstrait, et un générateur de code intermédiaire.

2.3 Grammaire

```
program -> list_stmt
stmt -> instruction ';'
        |   FOR ID '=' range block
        |   IF '(' affine ')' block
        |   IF '(' affine ')' block ELSE block
        |   FINISH '(' clocks ')' block
        |   FINISH block
        |   ASYNC '(' clocks ')' block
        |   ASYNC block
        |   ADVANCE ID ';'

list_stmt -> list_stmt stmt | stmt
instruction -> ID
clocks -> clocks ',' ID
        |  ID
        |   ε
block -> '{' list_stmt '}'
        |   stmt
range -> affine RANGE affine
affine -> affine '+' affine
  |   affine '-' affine
  |   affine '/' affine
  |   affine '*' affine
  |   affine COMP affine
  | '(' affine ')'
  | ID
  | NUMBER
```

2.4 Construction d'un arbre syntaxique abstrait

L'analyseur syntaxique produit comme résultat un arbre syntaxique abstrait, un AST à la suite, qui est utilisé plus tard pour effectuer une analyse sémantique et générer de code intermédiaire.

La structure d'un AST contient 12 types de nœuds - un nœud par instruction :

— id

— number

— basic

— advance

— finish

— async

— operation

— assignment

— for

— range

— statements

— if

Chaque nœud peut avoir au maximum trois fils.

Il est possible de visualiser un arbre syntaxique abstrait généré grâce à une fonction *ast_print(ast,indent)*.

Le résultat est présenté sur la Figure 4 où on peut remarquer que l'hiérarchie de l'arbre est obtenue à l'aide d'indentation.

```
finish (cl1,cl2) {
    async (cl1) {
        advance cl1;
        S1;
    }
    advance cl1;
    S2;
}
```

```
rendering AST
statements
finish
    cl2 cl1
    statements
    async
        cl1
        statements
        advance cl1
        S1
    advance cl1
    S2
```

FIGURE 4 – Exemple d'un résultat de la fonction *ast_print(ast,indent)*

2.5 Vérification sémantique

Après construire un arbre, les vérifications sur la sémantique du programme sont effectuées.

L'analyse sémantique comprend :

une détection de l'utilisation de variables non initialisées - comme dans le langage les seules variables sont les compteurs de boucles, il ne peuvent pas être utilisés avant cette boucle.

une détection des horloges utilisées en dehors de leur portée - l'activité n'est pas inscrite à une clock ne peut pas l'utiliser.

L'analyseur sémantique parcours récursivement un AST et compare les variables et les horloges rencontrées avec le contenu de la table de symboles.
Une clock est supprimée de la table de symboles à chaque fois quand elle est en dehors de la portée d'une instruction. De cette façon, l'instruction comme async ou advance ne peut pas utiliser une horloge qui n'est pas visible pour elle.

2.6 Machine virtuelle

Ayant un langage avec les instructions parallèles, il est possible de choisir entre une machine virtuelle parallèle et une machine virtuelle séquentielle pour simuler l'exécution du programme.

Etant donné que le but principal de l'interpréteur est de pouvoir produire la liste des activités en cours et pas la performance du programme, j'ai décidé d'utiliser un principe de l'exécution séquentielle. Cela veut dire, que seulement une activité est exécutée à la fois.

Ainsi, la machine virtuelle l'exécutera chaque activité **pas à pas** jusqu'au moment qu'elle sera bloquée et seulement après cela la prochaine activité dans une pile va s'exécuter.

2.7 Les structures de données de la machine virtuelle

Pour capturer des traces d'exécutions, on garde l'information principale sur un programme, telle que toutes les clocks créées, toutes les activités inscrites avec ces clocks etc.

Voici la structure de la machine virtuelle :

state :

- **ready** - une liste d'activités prêtes à être exécutées
- **running** - une activité en cours d'exécution

activity :

- **id** - l'identifiant d'une activité
- **program counter** - le compteur ordinal, que l'activité incrémente après la lecture d'une instruction et les instructions de saut écrivent dedans.
- **finish stack** - une pile de pointeurs sur les structures de finish englobants
- **registered with** - une liste d'horloges avec lesquelles l'activité est enregistrée

finish :

- **id** - l'identifiant d'un finish
- **clocks** - les horloges créées par ce finish
- **activities** - les activités qui sont dans le bloc du finish
- **is waiting** - un flag qui indique si ce finish est en attente de la terminaison de ces activités

clocks :

- **id** - l'identifiant d'un clock
- **registered** - la pile d'activités enregistrées avec ce clock
- **blocked** - la pile d'activités bloquées par ce clock

2.8 L'exécution du code intermédiaire

Le code intermédiaire est généré à partir d'un arbre de syntaxe abstrait.

Il est possible d'exécuter directement un AST sans générer le code, mais comme la machine virtuelle pas à pas a besoin de pouvoir retourner en arrière quand elle fait une boucle ou un changement d'activité, c'est plus logique d'avoir le code avec un compteur ordinal que garder un chemin de l'arbre pour chaque instruction.

Ci-dissous une spécification sur l'exécution des instructions principales de programme.

FINISH :

- créer une nouvelle structure finish
- la rajouter dans une pile de finish de l'activité en cours

CLOCK CREATE :

- créer une nouvelle structure clock
- ajouter l'activité en cours dans la pile 'registered' enregistrées avec ce clock
- ajouter ce clock dans une pile de clocks de l'activité en cours
- rajouter ce clock dans une pile de clocks de finish de l'activité en cours

END FINISH :

si la pile d'activités créées dans ce finish n'est pas vide, alors :

- mettre l'état 'is waiting' de finish à vrai
- pour chaque clock créée par ce finish enlever l'activité en cours de la pile 'registered' de ce clock
- enlever l'activité en cours de la pile d'activités prêtes à être exécutées

sinon mettre l'état de finish 'is waiting' à faux et détruire les clocks créés par ce finish

ASYNC :

- créer une nouvelle activité avec le compteur du programme égal à label suivant et le finish englobant dans la pile de finish
- ajouter cette activité dans 'ready' et dans une pile d'activités créées par le finish

CLOCK REGISTER :

- rajouter le clock dans une pile de clocks de l'activité en cours
- ajouter l'activité en cours dans la pile 'registered' de ce clock

END ASYNC :

- enlever l'activité en cours de la pile 'registered' pour tous les clocks avec lesquels elle était enregistrée
- enlever l'activité en cours de la pile d'activités du finish englobant
- enlever l'activité en cours de la liste 'ready'

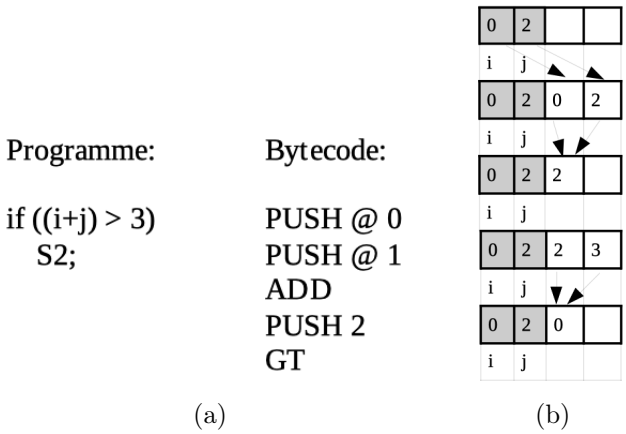
ADVANCE :

- rajouter l'activité en cours dans la pile 'blocked' du clock
- enlever l'activité en cours de la liste 'ready'
- si les piles 'blocked' et 'registered' du clock sont identiques, déplacer les activités qui étaient dans la pile 'blocked' dans la liste 'ready'

2.9 Evaluation des expressions arithmétiques

L'évaluation des expressions arithmétiques s'effectue à l'aide d'une pile. Plus précisément, on mets les résultats intermédiaires dans la pile et la manipule pour faire le calcul. Si le programme a plusieurs variables (plusieurs compteurs de boucles) leurs valeurs sont stockés sur les positions concrètes de la pile (sous les premiers indices) et sont mis à jour à chaque itération.

Sur la Figure 5 on peut observer une schéma qui montre le principe du calcul.



(a) (b)

FIGURE 5 – Exemple d'évaluation des expressions à l'aide de la pile

Pour exécuter ces expressions les instructions spéciales sont présentes dans le code intermédiaire, telles que les instructions :

— **PUSH @ 0** - ajouter une nouvelle variable à l'adresse 0 de la pile

— **PUSH 3** - empiler un entier 3

— **ADD, MULT, DIV, SUB** - effectuer une opération affine entre les deux valeurs qu'ils sont dans la pile

— **LT, GT, EQ, NE,. . .** - effectuer un comparaison entre les deux valeurs qu'ils sont dans la pile

Le principe de l'évaluation à l'aide de la pile est assez simple et n'est pas en priorité pour ce recherche, c'est pourquoi seulement le code intermédiaire pour les expressions arithmétiques est implémenté dans la version courante du projet.

3 Conclusion

Le but de mon travail d'étude et de recherche était de simuler l'exécution de programmes parallèles et d'avoir la possibilité de détecter l'interblocage. Dans son cadre un langage de programmation parallèle a été créé basé sur le langage déjà existant X10 mais seulement avec les instructions nécessaires pour la suite de la recherche.

Pour cela l'analyseur lexical et l'analyseur syntaxique étaient implémentés avec comme résultat un arbre de syntaxe abstrait. Le code intermédiaire est généré en parcourant un AST. La machine virtuelle séquentielle réalisée, parcours pas à pas ce code intermédiaire et est capable d'afficher l'état global de chaque activité avec l'état d'horloges créées.

Grâce à ce travail il est possible maintenant de détecter un interblocage et analyser l'état de la machine virtuelle pour savoir les horloges et les activités qui ont provoqué cet interblocage.

Pour aller plus loin, on pourrait faire une visualisation interactive de l'état d'exécution d'un programme parallèle.

Références

[1] Alain Ketterlin. *La présentation de sujet de TER.*

[2] Agarwal Shivali, Barik Rajkishore, Sarkar Vivek, Shyamasundar R.K. *May-happen-in-parallel analysis of X10 programs.* Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2007), PPOPP.

[3] Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove *X10 Language Specification.* Version 2.6.2

[4] Composing Programs : Parallel Computing, `https://composingprograms.com/pages/48-parallel-computing.html`