Simulation d'exécution de programmes parallèles

Alain Ketterlin (alain@unistra.fr)

TER – Automne 2018-2019

Le but de ce travail est de simuler l'exécution de programmes parallèles, et de visualiser leurs traces. Le langage parallèle utilisé comprend des constructions pour la création d'activités parallèles, et pour limiter la portée du parallélisme. Il permet également de définir des objets de synchronisation de portée limitée syntaxiquement, et comporte des instructions de synchronisation. Pour ce qui concerne la synchronisation, la puissance et la liberté offertes conduisent fréquemment à des inter-blocages au cours de l'exécution. Ce travail se place dans le cadre d'une recherche visant à caractériser ces inter-blocages, afin de les détecter le plus tôt possible, à savoir, éventuellement, lors de la compilation du programme.

Le travail consistera à :

1. écrire un analyseur syntaxique (et partiellement sémantique) pour un langage parallèle raisonnablement expressif, et définir une structure générique d'arbre syntaxique abstrait ;
2. visualiser sous diverses formes le programme lui-même, ou différentes simplifications du programme ;
3. simuler l'exécution du programme (éventuellement avec interaction de l'utilisateur), et capturer des traces d'exécutions ;
4. visualiser les traces d'exécutions afin de mettre en lumières les éventuels problèmes d'inter-blocage.

Un langage parallèle

Les constructions dédiées au parallélisme sont inspirées de celles fournies par le langage X10 (cf. `x10-lang.org`). Un programme X10 démarre son exécution au sein d'une « activité » principale. N'importe quelle activité peut créer une nouvelle activité parallèle avec la construction :

```
async { ... }
```

L'activité créatrice continue son activité après l'instruction `async`, alors que l'activité créée exécute le corps de l'instruction puis se termine.

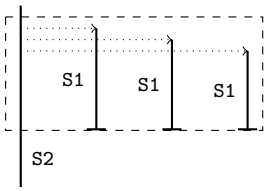
La construction `finish` permet d'attendre la fin de *toutes* les activités créées dans un fragment de programme. Une activité exécutant une instruction de la forme :

```
finish { ... }
```

exécute le corps du bloc (en créant potentiellement de nouvelles activités), mais ne poursuit l'exécution *après* `finish` que lorsque les activités créées par le bloc sont terminées.

Voici par exemple un programme qui lance trois activités en parallèle (exécutant chacune un appel de la fonction `S1`), attends leur fin, puis appelle la fonction `S2` :

```
finish
    for i in 0..3
        async S1(i);
S2()
```

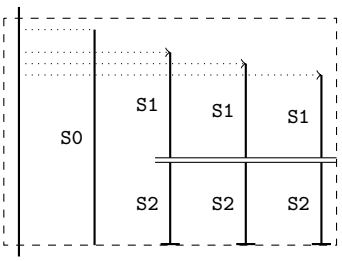


Les règles gouvernant la présence d'accolades dans le code sont les même que celles de C/C++/Java/... Notez que le langage devra également prévoir des boucles et éventuellement des instructions conditionnelles, mais que ces instructions pourront dans un premier temps être très simples (comme ci-dessus).

Synchronisation

La synchronisation d'activités parallèles est basées sur la notion d'horloge (*clock* dans la suite), qui est en fait une barrière de synchronisation : toutes les activités associées à une clock doivent se donner rendez-vous en un point du programme. Une clock est créée par une instruction `finish`, et est utilisable par une activité qui la mentionne explicitement dans son instruction de création. Par exemple :

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



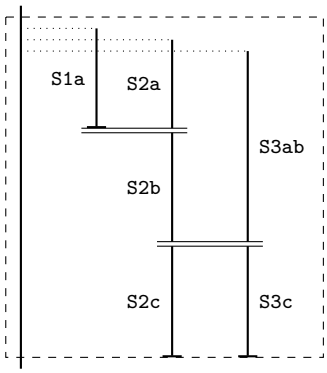
Ce fragment commence par créer une clock `clk1` pour la durée de l'instruction `finish`. Il lance ensuite une activité exécutant `S0()` : cette activité ne participe pas à la synchronisation basée sur `clk1`. Ensuite, il lance trois activités distinctes, qui toutes annoncent utiliser `clk1`, ce qu'elles font avec l'instruction spéciale `advance`. Donc, les trois activités se donnent rendez-vous entre leur appel à `S1(i)` et leur appel à `S2(i)` : aucun appel à `S2(i)` n'a lieu tant que tous les appels à `S1(i)` ne sont pas terminés. Enfin, le bloc `finish` ne se termine pas tant qu'il existe une activité (soit une des trois, soit celle exécutant `S0()`).

Notez que ce langage est intrinsèquement plus expressif que les langages de programmation habituels. Par exemple, OpenMP permet l'utilisation d'une barrière (explicite, via la directive `#barrier`) dans différentes constructions. Par contre, il ne définit pas ce qui se passe en cas d'imbrication de telles constructions. (Les règles qui gouvernent l'imbrication de blocs `finish` sont précises, mais il est impossible de les expliciter ici.)

Il est possible d'utiliser plusieurs clocks simultanément. Le principe est le suivant : `finish` ajoute des clocks aux clocks dans la portée desquelles il apparaît, alors que `async` indique

la liste des clocks qu'il compte utiliser. Par exemple :

```
finish clk1, clk2 {
    async clk1 {
        S1a();
        advance clk1;
    }
    async clk1, clk2 {
        S2a();
        advance clk1;
        S2b();
        advance clk2;
        S2c();
    }
    async clk2 {
        S3ab();
        advance clk2;
        S3c();
    }
}
```



Ce fragment comporte deux étapes de synchronisation : une première sur `clk1`, une second sur `clk2`. Avant la première synchronisation, `S1a()`/`S2a()`/`S3ab()` peuvent s'exécuter en parallèle. La première synchronisation (sur `clk1`) s'assure que `S1a()` et `S2a()` sont terminées (mais pas nécessairement `S3ab()`). Après cette synchronisation, la première activité s'arrête, et les appels `S2b()`/`S3ab()` peuvent s'exécuter en parallèle : la seconde étape de synchronisation (sur `clk2`) s'assure que ces deux appels se terminent. Après cette seconde synchronisation, les appels de `S2c()`/`S3c()` peuvent s'exécuter en parallèle. Enfin, le bloc `finish` ne sera terminé que quand ces deux appels seront terminés.

La multiplicité des clocks évoque un autre langage très utilisé en parallélisme (essentiellement lorsque la mémoire est distribuée) : MPI, qui permet aussi l'utilisation de plusieurs barrières de synchronisation via la notion de communicateur. La différence majeure de notre mini-langage avec ce schéma classique est la possibilité de varier dynamiquement les activités associées à une clock donnée.

Visualisation

Pour diverses raisons, on a besoin de visualiser un programme parallèle utilisant plusieurs clocks. Le premier objectif du stage est de proposer une visualisation du code d'un programme, similaire à celles proposées ci-dessus en face des fragments de code. Il faut pour cela deux éléments.

Le premier élément (incontournable) est un analyseur syntaxique (et sémantique pour les portées de clocks et éventuellement les autres variables). Le résultat de l'analyse doit être un arbre syntaxique abstrait facilement manipulable, non seulement pour la visualisation, mais également pour d'autres opérations (par exemple la transformation de programmes). L'arbre de syntaxe abstrait servira aussi de base à la simulation.

Le second élément est un algorithme qui dessine un programme, et met en lumière d'une part le parallélisme, et d'autre part les synchronisations entre activités. Toutes les proposi-

tions sont acceptables, à partir du moment où les informations essentielles sont présentes. Le format de sortie importe peu, il est même recommandé de produire un résultat dans un langage graphique à base de positions et de lignes, lequel pourra ensuite être interprété de différentes façons.

Simulation

Dans un second temps, on souhaite pouvoir simuler des programmes écrits dans notre mini-langage. Il faut pour cela définir une « machine virtuelle » capable d'interpréter un programme, probablement donné sous la forme d'un arbre abstrait – il n'est pas question ici de définir un « bytecode ». Le but principal de l'interpréteur est de pouvoir explorer, à tout moment, la liste des activités en vol, l'état de toutes les clocks, et toute autre information pertinente. En particulier, en cas d'inter-blocage, il doit être possible de connaître exactement la liste des activités impliquées, et pour chacune d'elle de disposer d'une description du moment où elle a été créée, des clocks qu'elle peut utiliser, etc. Il doit aussi être possible de connaître l'état de chaque clocks, en particulier le nombre de rendez-vous déjà réalisés, etc.

Par exemple, pour le fragment suivant :

```
finish c1, c2, c3 {
    async c1, c2 { S1a(); advance c1; S1b(); advance c2; S1c(); }
    async c2, c3 { S2a(); advance c2; S2b(); advance c3; S2c(); }
    async c1, c3 { S3a(); advance c3; S3b(); advance c1; S3c(); }
}
```

Il doit être possible de visualiser l'état de toutes les activités et toutes les clocks au moment de l'inter-blocage, qui intervient lors de la première instruction `advance` de chaque activité.