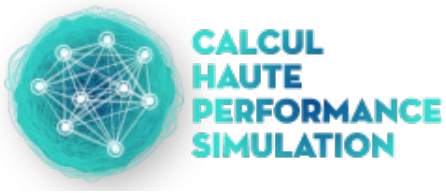

Projet de Programmation Numérique
TASKIFICATION

Encadré par : Jean-Baptiste Besnard
Réalisé par :
Karim SMAIL
Sofiane BOUZAHER
Asma KREDDIA
Atef DORAI

11 janvier 2020

Sommaire

1 Introduction **2**
- 1.1 Titre du Sujet . . . 2
- 1.2 Mots Clefs . . . 2
- 1.3 Description Générale . . . 2
- 1.4 Objectif Général . . . 2
  - 1.4.1 Pourquoi le parallélisme ? . . . 2

2 Concepts Clef **3**
- 2.1 Définition d'un Compilateur . . . 3
- 2.2 Objectif du Projet . . . 3
- 2.3 Définitions Utiles . . . 3

3 Fonctions pures - Fonctions impures **4**
- 3.1 Définition d'une fonction pure : . . . 4
- 3.2 L'avantage principal d'une fonction pure : . . . 4
- 3.3 La fonction floor . . . 5
- 3.4 La fonction max(resp.min) . . . 5

4 Définition d'une fonction impure **5**
- 4.1 Non localité des paramètres . . . 5
- 4.2 Référence externe : . . . 5
- 4.3 Entrées-Sorties . . . 5
  - 4.3.1 L'effet de bord de la fonction f ici est de modifier la valeur de la variable non local : 6
  - 4.3.2 à cause de la mutation d'une variable statique locale : . . . 6
  - 4.3.3 à cause de la mutation d'un argument mutable de type référence : . . . 6
  - 4.3.4 à cause de la mutation d'un flux de sortie . . . 6

5 Présentation de CLANG **6**
- 5.1 Définition de L'infrastructure clang - LLVM : . . . 6
  - 5.1.1 le Frontend : . . . 7
  - 5.1.2 les Passes : . . . 7
  - 5.1.3 le backend : . . . 7
  - 5.1.4 CLANG (appelé Driver de compilation) : . . . 7
  - 5.1.5 OPT : . . . 7
  - 5.1.6 LLC : . . . 7
  - 5.1.7 LLVM-AS et LLVM-DIS : . . . 7
  - 5.1.8 LLI : . . . 8
- 5.2 Introduction a l'AST . . . 8

1 Introduction

Le but de notre projet est la mise en place d'un plugin dans le compilateur CLANG (LLVM) pour identifier les appels de fonctions candidats à la « taskification » (fonctions pures).
Pour ce faire, il faudra :

— mettre en place la structure de base d'un plugin clang.
— Définir ce qui distingue les fonctions candidates à identifier.
— Implémenter ce support dans le plugin ;
— Le valider sur un cas de parallélisme producteur / consommateur.

1.1 Titre du Sujet

Analyse Statique Pour la Classification des Procédures Candidate à la « Taskification »

1.2 Mots Clefs

LLVM, Analyse statique, compilation, MPI + X, parallélisation automatique

1.3 Description Générale

Les architecture hybrides convergées à venir posent la question des modèles de programmation. En effet MPI depuis l'avènement des architectures many-core a dû être combiné avec du paralslélisme intra-noeud en OpenMP (MPI + X). Le mélange de ces modèles se traduit nécessairement par une complexité accrue de l'expression des codes de calcul. Dans ce travail nous proposons de prendre cette tendance à contre-pied en posant la question de l'expression de tâche de calcul en pur MPI. Les étudiants se verront fournir une implémentation de Remote Procedure Calls (RPC) implémentés en MPI, le but du travail et de détecter quelles fonctions sont éligibles à la sémantique RPC statiquement lors de la phase de compilation (c.a.d. les fonction dites « pures » : indépendantes du tas, des TLS, etc ...). Le travail visera le compilateur LLVM dans lequel une passe sera rajoutée pour lister l'ensemble des fonctions éligibles à la sémantique RPC. Pour exemple, une implémentation d'un algorithme de cassage de mot de passe en MPI sera fournie avec pour but sa conversion en RPC producteur/consommateur (github.com/besnardjb/MPI_Brute/) avec l'outil.

1.4 Objectif Général

Le but d'un programme est d'exécuter une tâche. Pour réaliser celle-ci, on donne à l'ordinateur une liste d'instructions qu'il va effectuer. Il existe plusieurs manières de traiter ces instructions, parmi ces manières, on trouve la programmation parallèle.

1.4.1 Pourquoi le parallélisme ?

L'exécution de certaines fonctions d'un programme de manière parallèle, nous permet un gain de temps d'exécution du programme, ce qui veut dire rendre le programme plus performant qu'avant, mais ça ne marche pas avec toutes les fonctions de tous les programmes, Donc les fonctions qui seront exécuter de manière parallèle doit être connu à la compilation du programme.

2 Concepts Clef

2.1 Définition d'un Compilateur

Un compilateur est un programme qui transforme un code source (écrit dans un langage de programmation de haut niveau d'abstraction) en un code objet (écrit dans langage de programmation de bas niveau) afin de créer un programme exécutable par une machine.

2.2 Objectif du Projet

Le but est la mise en place d'un plugin dans le compilateur CLANG (LLVM) pour identifier les appels de fonctions candidats à la « taskification » (fonction pures) , Pour ce faire, il faudra mettre en place la structure de base d'un plugin clang ,définir ce qui distingue les fonctions candidates à identifier , et enfin implémenter ce support dans le plugin et Le valider sur un cas de parallélisme producteur / consommateur.

2.3 Définitions Utiles

LLVM (Low Level Virtual Machine) est une infrastructure de compilateur conçue pour l'optimisation du code à la compilation (c'est une infrastructure qui ne contient pas des outils nécessaires pour compiler du code source C ou C++ mais uniquement des outils d'optimisations et de génération de codes machine à partir d'un format intermédiaire).

CLANG-LLVM : la structure générale de cette infrastructure à l'échelle microscopique est constitué d'une façon similaire à tout compilateur moderne .

Analyse statique : définie comme étant l'ensemble des méthodes utilisées pour obtenir des informations sur le comportement d'un programme lors de son exécution sans réellement l'exécuter.

Analyse dynamique (dynamic program analysis) : contrairement à l'analyse statique, est une analyse qui nécessite l'exécution du programme et étudie son comportement +les effets de son exécution sur son environnement.

Compilation : définie comme l'ensemble des étapes qui transforment un code (. C) en un code objet(. O). cela se fait à l'aide d'un programme appelé 'Compilateur '.

MPI : Message Passing interface, est une norme conçue pour le passage de messages entre ordinateurs distants ou dans un ordinateur multiprocesseur (donc c'est un moyen de transfert de message créer pour obtenir de meilleures performances), Elle est devenue un standard de communication pour des nœuds exécutant des programmes parallèles sur des systèmes à mémoire distribuée. Elle définit une bibliothèque de fonctions, utilisable avec les langages C, C++ et Fortran.

Open MPI (MPI+X) : (une bibliothèque MPI) :est une bibliothèque de projet qui sert à combiner l'expertise, les techniques et les ressources de toute la communauté du calcul haute performance afin de créer la meilleure bibliothèque MPI disponible.

La parallélisation automatique : est une étape de la compilation d'un programme qui consiste à transformer un code source écrit pour une machine séquentielle en un exécutable parallélisé pour ordinateur à un multiprocesseur symétrique.

Un multiprocesseur symétrique (ou symmetric shared memory multiprocessor -SMP) est une architecture parallèle son but est de multiplier les processeurs identiques sur un ordinateur, pour augmenter la puissance de calcul.

Remonte Procédure Calls (RPC) : definie comme etant un moyen de communication qui permet de faire appeller des procedures sur un ordinateur a distance en utilisant un serveur d'application.

Mémoire partagée : (communication interprocessus) : peut etre expliquer comme etant le segment de memoire permettant l'acces de plusieurs processus (l'acces des differents processus a la memoire partagée est assuré par la synchronisation .

Mémoire distribuée : Un système distribué est défini comme étant un ensemble des ressources physiques et logiques géographiquement dispersées et reliées par un réseau de communication dans le but de réaliser une tâche commune. Cet ensemble donne aux utilisateurs une vue unique des données du point de vue logique.

Open Mp (Open multi-processing) : OpenMP est une bibliothèque supportée par plusieurs langages (C,C++ et Fortran) disponible sur plusieurs plate-formes (Linux, Windows, OS X, ...). OpenMP regroupe des directives de compilation et des fonctions. Le compilateur CLANG LLVM supporte OpenMP .

3 Fonctions pures - Fonctions impures

3.1 Définition d'une fonction pure :

une fonction pure ne dépend pas et ne modifie pas l'état de variables hors de sa portée.

En pratique, cela signifie qu'une fonction pure retourne toujours le même résultat avec des paramètres identiques.

Son exécution ne dépend pas de l'état du système.

C'est-à-dire elle possède les propriétés suivantes :

1. Sa valeur de retour est la même pour les mêmes arguments (pas de variation avec des variables statiques locales, des variables non locales, des arguments mutables de type référence ou des flux d'entrée).
2. Son évaluation n'a pas d'effets de bord .

En informatique, une fonction est dite à effet de bord (effet secondaire)si elle modifie un état en dehors de son environnement local, c'est-à-dire a une interaction observable avec le monde extérieur autre que retourner une valeur.

Par exemple, les fonctions qui modifient une variable locale statique, une variable non locale ou un argument mutable passé par référence.

les fonctions qui effectuent des opérations d'entrées-sorties ou les fonctions appelant d'autres fonctions à effet de bord.

Souvent, ces effets compliquent la lisibilité du comportement des programmes et/ou nuisent à la réutilisabilité des fonctions et procédures.

3.2 L'avantage principal d'une fonction pure :

— l'appel à cette fonction avec les mêmes paramètres renverra toujours le même résultat.
— On simplifie également la mise en place des tests automatiques, ce qui sécurise notre application.
— Les fonctions pures ont pour avantage d'être prédictibles.

Ce qui permet de les tester plus facilement et surtout de mettre leur résultat en cache pour ne pas avoir à refaire le calcul pour des valeurs qu'on a déjà traitées.

Les fonctions pures sont souvent utilisées pour générer d'autres fonctions. Dans ce cas, elles sont appelées "Higher Order Functions" ou "Fonctions de rang supérieur".

Note : Une fonction de rang supérieur peut ne pas être une fonction pure.

Les fonctions arithmétiques sont l'archétype des fonctions pures.

Les fonctions suivantes sont pures :

3.3 La fonction floor

Cette fonction retourne la valeur entière d'un nombre, soit l'entier le plus proche inférieur ou égal au nombre.

Voici un exemple montrant une utilisation plus classique de cette fonction :

double floor(double value) ;

3.4 La fonction max(resp.min)

```
double MAX(double X, double Y)
{
if (X>Y) return X;
else return Y;
}
```

4 Définition d'une fonction impure

Une fonction impure est une fonction qui peut avoir des effets de bords(elle peut aussi accidentellement ne pas en avoir).

le résultat de la fonction peut dépendre du contexte, et son exécution peut le modifier .

une fonction de comptage (qui rend le nombre de fois où elle a été appelée, nécessitant donc la modification d'une variable externe) ,ou les fonctions NOW du paquetage STANDARD qui rendent l'heure qui est dans le monde simulé, et donc une valeur différente a chaque appel, sont des fonctions impures.

Les fonctions C suivantes sont impures car elles ne vérifient pas la propriété 1 ci-dessus :

4.1 Non localité des paramètres

à cause de la variation de la valeur de retour avec une variable non locale

```
int f() {
    return x;
}
```

4.2 Référence externe :

à cause de la variation de la valeur de retour avec un argument mutable de type référence

```
int f(int* x) { // la fonction f renvoie un pointeur sur entier
    return *x; // l'adresse de x est bien un pointeur sur entier
}
```

4.3 Entrées-Sorties

à cause de la variation de la valeur de retour avec un flux d'entrée

```
int f(int x) {
    x = 0;
    scanf("%d, &x);
    return x;\\
}
```

Les fonctions C suivantes sont impures car elles ne vérifient une des propriétés ci-dessus :

4.3.1 L'effet de bord de la fonction f ici est de modifier la valeur de la variable non local :

```
void f() {
    ++x;
}
```

4.3.2 à cause de la mutation d'une variable statique locale :

```
{
    static int i=0;

    i++;

}
```

4.3.3 à cause de la mutation d'un argument mutable de type référence :

```
void f(int* a) {
     ++*a;
}
```

4.3.4 à cause de la mutation d'un flux de sortie

```
void f()
{
    printf("Hello.\n");
}
```

5 Présentation de CLANG

5.1 Définition de L'infrastructure clang - LLVM :

LLVM est une collection de compilateurs modulaires ,réutilisables et de technologies de chaîne d'outils.D'une façon macroscopique, elle est construite d'une manière similaire à tout compilateur moderne, elle ne contient pas les outils nécessaires pour compiler du code source C ou C++ mais uniquement des outils d'optimisation et de génération de codes machines à partir d'un format intermédiaire. Pour mieux comprendre l'ensemble des étapes de compilation par cette infrastructure, on doit définir en détails :

5.1.1 le Frontend :

C'est le premier bloc de tout compilateur, son objectif est de valider que le programme est syntaxiquement et sémantiquement correct puis de le traduire vers une représentation intermédiaire (IR pour Intermediate Représentation) l'un des objectifs de cette représentation intermédiaire étant de simplifier le travail des autres blocs qui ne peuvent pas travailler avec la complexité de code source C ou encore pire C++.

5.1.2 les Passes :

Sont en charge d'analyser et/ou de transformer l'IR en optimisant certaines choses tout en préservant la sémantique du code. Son objectif était très souvent la maximisation des performances du code (par exemple en jouant sur la taille du code).

5.1.3 le backend :

Est en charge de transformer l'IR vers du code machine pour une architecture donnée.

Afin d'intervenir aux différents niveaux de la chaine de compilation, plusieurs outils vont venir intervenir tels que :

5.1.4 CLANG (appelé Driver de compilation) :

quand on dit clang on dit frontend C/C++ de l'infrastructure LLVM En tant que driver de compilation, l'outil clang peut-être arrêté à différents niveaux de la chaîne de compilation, et pour mieux comprendre ce point je vous montre des exemples d'utilisation de l'outil clang :

Cas 1 : clang -S -emit-llvm -o test.ll test.c => dans ce cas clang est utilisé comme étant un frontend c'est-à-dire générer IR textuel à partir du code source

Cas 2 : clang -o test.bin test.s => clang est utilisé comme assembleur et linker

5.1.5 OPT :

cet outil permet d'appliquer un ensemble de passes LLVM, l'entrée d'OPT est un fichier au format IR (bit code ou textuel) et la sortie produite est également un fichier au format IR (bit code ou textuel).

Pour le choix des passes a appliqué, l'outil OPT peut-être utilisé avec les options usuelles -O1, -O2, -O3 (si aucune des options -Ox n'est pas spécifiée, OPT n'applique aucune passe).

On peut aussi spécifier individuellement les passes que nous souhaitons appliquer et on prend comme exemple la commande suivante :

opt -S -mem2reg -constprop -o test-after-cp.ll test.ll telque mem2reg et constprop sont les passes qu'on a choisi.

5.1.6 LLC :

Il permet de compiler du code au format LLVM IR (bitcode ou textuel) vers du code assembleur pour une architecture donnée.

5.1.7 LLVM-AS et LLVM-DIS :

llvm-as et llvm-dis permettent respectivement de passer du format LLVM IR textuel au format LLVM IR bitcode et inversement.

5.1.8 LLI :

Cet outil permet d'exécuter du code au format LLVM IR (bitcode ou textuel) non pas en le compilant vers du code machine mais en l'interprétant directement.

5.2 Introduction a l'AST

Au cours de la compilation avec Clang on passe par 3 étapes : Frontend qui prend en charge la partie analyse, ie la décomposition du code source en morceaux selon une structure grammaticale, le résultat sera transformé dans un target program par le Backend,cette étape est appelée synthèse, tout en passant par une étape intermédiaire d'optimisation entre le Frontend et le Backend.
Code source –> Frontend(Analyse)–>optimizer–>Backend(synthèse)–> machine code .
Dans notre projet on s'intéresse de Frontend qui est responsable d'analyser le code source, vérifier s'il contient des erreurs et enfin le transformer en ABSTRACT SYNTAX TREE (AST).
L'AST est une représentation structurée de la syntaxe du code que l'on analyse.
L'AST est en fait une structure en C++ assez complexe de classes avec héritages.
Exemple de L'AST :
soit la fonction suivante :

```
int add(int a, int b) {
    return a + b;
}
```

L'AST correspondant a la fonction présidente :
obtenu par la commande ($clang - Xclang - ast - dump - fsyntax - only code_source.cpp$)

```
             command_block
                /       \
    function_decl       function_impl
         |                  /     \
   function_name=add  return_stm  math_expresion
      /        \                       |
return_type    parameters             operator '+'
   |              |                     /     \
name "int"     separator ','        Var_value  var_value
                 /       \             |           |
          param_decl    param_decl    "a"         "b"
           /    \         /       \
    type_decl name "a"  type_decl  name "b"
        |                  |
    name "int"          name "int"
```

Références

[1] Jonas Devlieghere. Understanding the clang ast. https ://jonasdevlieghere.com/understanding-the-clang-ast, Mis en ligne le 31 Dec 2015.

[2] The Clang Team. Clang 10 documentation. https ://clang.llvm.org/docs/ClangPlugins.html.