Projet de Programmation Numérique sur Machine Parallèle "TASKIFICATION"

Dépôt Git

Encadré par : **Jean-Baptiste Besnard**

Réalisé par :

Sofiane BOUZAHER
Asma KREDDIA
Karim SMAIL
Atef DORAI

M1CHPS

13 mai 2020

Remerciements

Nous souhaitons avant tout remercier notre encadrant de projet, Jean-Baptiste BESNARD, pour le temps qu'il a consacré à nous apporter les outils méthodologiques indispensables à la conduite de cette recherche. Son exigence nous a grandement stimulés.
L'enseignement de qualité dispensé par le Master « CHPS » à également su nourrir nos réflexions et a représenté une profonde satisfaction intellectuelle, un grand merci donc aux enseignants.

Table des figures

Table des matières

1 Titre du Sujet

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

2 Introduction

Comme on a déjà vu précédemment au premier semestre l'objectif 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), l'objectif sous-jacent est l'identification du potentiel de parallélisation automatique du code en trouvant les fonctions pures et ensuite en les insérant dynamiquement dans des appels de fonctions parallèles via des Remote Procesure Call (RPC).

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.

2.1 Mots Clefs

Bibliothèque Clang LLVM, RPC,AST,Makefile ,PLugin

2.2 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 est 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.

2.3 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.

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 ceci n'est pas applicable à tous les algorithmes. Dans ce projet nous cherchons donc à développer une méthode automatisée pour localiser les portions de programme qui sont candidats à la parallélisation automatique.

3 Résumé des Travaux du Premier Semestre

Précédemment au projet effectué en semestre 1 on s'est projeté sur plusieurs points :

3.1 Fonctions pures/impures

On a commencé à définir ce qu'est une fonction pure et une fonction impure car les deux sont des fonctions candidates à la "Taskification" .
On a défini une fonction pure comme étant une fonction qui ne dépend pas et ne modifie pas l'état de variable qui n'est pas à ça portée ,son principal avantage est qu'elle est prédictible c'est à dire facile à tester [4] et on a cité quelques exemples sur les fonctions pures comme la fonction max,min et floor .
Pour les fonctions impures ce sont des fonctions a effets de bord c'est à dire elles renvoient des valeurs différentes à chaque appel . Une fonction peut être impure à cause de plusieurs raisons [8] :

- la variation de la valeur de retour avec une variable non locale.
- la variation de la valeur de retour avec un argument mutable de type référence
- la variation de la valeur de retour avec un flux d'entrée.
- la mutation d'une variable statique locale.
- la mutation d'un argument mutable de type référence.
- la mutation d'un flux de sortie.

3.2 Présentation de CLANG [6]

On a défini l'infrastructure clang - LLVM comme étant une collection de compilateurs modulaires ,réutilisables et de technologies de chaîne d'outils. Pour mieux comprendre l'ensemble des étapes de compilation par cette infrastructure,on a défini :

- le Frontend : qui est le premier bloc de tout compilateur son objectif est de voir si le programme est syntaxiquement et sémantiquement correct puis le traduire vers une présentation intermédiaire(IR) pour simplifier le travail des autres blocs.
- Les Passes : qui sont en charge d'analyser et/ou de transformer l'IR en optimisant certaines choses .l'objectif est de maximiser les performances du code.
- le backend : est en charge de transformer l'IR vers du code machine afin d'intervenir aux différents niveaux de la chaine de compilation .

On a défini aussi CLANG,OPT,LLC,LLVM-AS/LLVM-DIS et LLI comme des outils qui participent a la compilation et l'execution.

3.3 Generalité sur 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.[2]
Donc 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.

On a aussi vu plusieurs définitions utiles comme celles de l'analyse statique , l'analyse dynamique, la compilation , la mémoire partagée et la mémoire distribuée .

4 Concepts Clef

4.1 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 pure) ,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.

4.2 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).[6]

CLANG-LLVM : la structure générale de cette infrastructure à l'échelle microscopique est constitué d'une façon similaire à tout compilateur moderne .[6]

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.[5]

La parallélisation automatique : c'est l'un des stages de compilation d'un programme au niveau de lequel le code source est transformé en un exécutable parallélisé pour ordinateur à un multiprocesseur symétrique dont le but de simplifier et de réduire la durée de développement des programmes parallèles.[5]

Un multiprocesseur symétrique (ou symmetric shared memory multiprocessor -SMP) : le but principal de cette architecture parallèle est de multiplier les processeurs identiques sur un ordinateur, afin d'améliorer la puissance de calcul.[5]

Remonte Procédure Calls (RPC) : définie comme étant un moyen de communication qui permet de faire appeler des procédures sur un ordinateur a distance en utilisant un serveur d'application.[5]

Plugin Clang : Le compilateur Clang peut être étendu à l'aide d'un plugin qui permet d'exécuter des actions supplémentaires déjà définies par l'utilisateur au moment de la compilation[7], c'est a dire qu'il peut fournir des avertissements et des erreurs de compilation en plus, ou effectuer des modifications de code source.[1]

Makefile : Les Makefiles sont des fichiers, généralement appelés makefile avec m minuscule ou Makefile avec m majuscule ,ils sont utilisés par le programme make qui est un logiciel de construction automatique ,pour exécuter un ensemble d'actions.[3]

5 Description de notre Plugin :

5.1 Clang plugin :

Rappelons que le plugin est une librairie dynamique chargée par le compilateur au moment de l'exécution d'un code source. Notre plugin «impurewarnings» permet de détecter le type de fonctions dans un code source, pures ou impures, lors de la compilation.
En chargeant notre plugin lors d'une nouvelle compilation, nous attendons un WARNING pour

chaque fonction impure et une liste contenant toutes les fonctions impures.

Pour qu'un plugin fonctionne parfaitement, il doit contenir :
— un code source en cpp basé sur 3 classes :
Class **PluginASTAction**.
Class **ASTConsumer.**
Class **RecursiveASTVisitor**.
— un Makefile pour générer la bibliothèque dynamique et exécute le plugin.

5.2 Description du code source de plugin :

Ci-dessous les entêtes standard qui appartient à la documentation clang et qui sont requis pour le code :

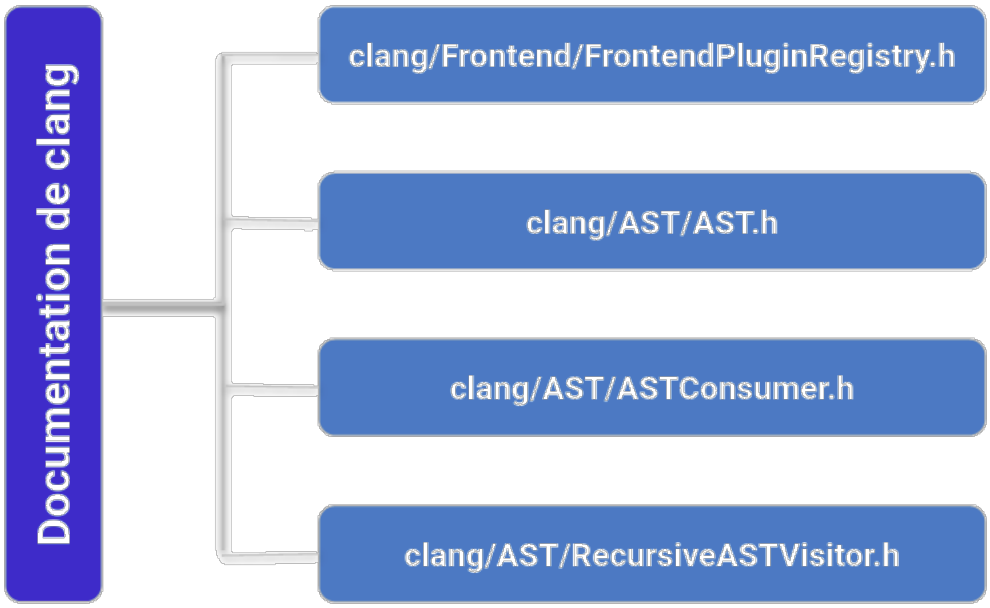

FIGURE 1 – Les Entêtes utilisés

ASTAction :
la classe PluginASTAction personnalisée est une classe de base abstraite à utiliser pour le class ASTConsumer. Il s'agit d'un point d'entrée à partir duquel nous pouvons invoquer notre ASTConsumer.

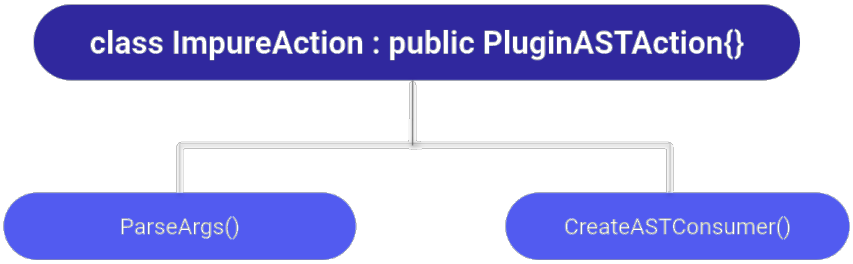

FIGURE 2 – ASTAction

Cette classe contient deux fonctions :

— **CreateASTConsumer :** cette fonction est appelée automatiquement par Clang lorsqu'on charge le plugin lors de la compilation.

— **ParseArgs :** cette fonction est indispensable pour analyser les arguments de «command line ».

ASTConsumer :

La tâche principale de l'ASTConsumer est de parcourir l'AST généré par l'analyseur du Clang, afin que notre code soit appelé lorsqu'un certain type d'élément AST a été analysé. Dans notre class ImpureExprConsumer vous trouverez 3 fonctions :

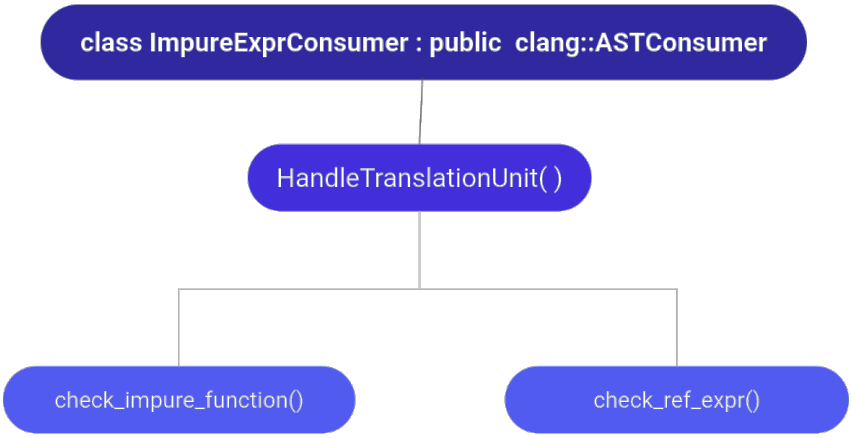

FIGURE 3 – ASTConsumer

— **HandleTranslationUnit :**cette fonction appartient a la documentation officielle du clang, et qui sera appelée chaque fois qu'on parcoure un nouvel ensemble de déclarations de niveau supérieur comme une variable globale ou une définition de fonction.

Remarque : HandleTranslationUnit est appelée qu'après la fin de l'analyse complète du fichier source, donc on a besoin d'ASTContext qui nous permet de garder la représentation de l'AST après l'analyse, ASTContext est utilisé pour obtenir le TranslationUnitDecl , qui représente l'intégralité du fichier source sous forme d'un seul Decl.

— A chaque fois qu'on traverse une nouvelle déclaration de fonction (détection d'un FunctionDecl dans l'AST), la fonction **check_impure_function** sera appelé.

— A chaque fois qu'on traverse une nouvelle déclaration d'une variable (détection d'un VarDecl dans l'AST), on l'affiche .

— A chaque fois qu'on traverse un appel vers une fonction extérieur ou une variable globale (détection d'un DeclRefExpr dans l'AST) la fonction **check_ref_expr** sera appelé.

check_impure_function : Cette fonction affiche le corps de la fonction et en même temps vérifie si la fonction fait un appel vers une autre fonction ou non, dans le cas ou c'est vrai un WARNING sera affiché.

check_ref_expr : Cette fonction vérifie si on fait un appel vers une variable globale ou non, dans le cas ou c'est vrai un WARNING sera affiché.

— Chaque fonction dans cette classe associe à un RecursiveASTVisitor, comme nous allons l'expliquer ci-dessous.

RecursiveASTVisitor :

À l'aide de deux classes précédentes on a préparé l'infrastructure de notre plugin, en parcourant la documentation officielle de RecursiveASTVisitor. On trouve de nombreuses fonctions telle que : VisitFunctionDecl , VisitStmt,... qui nous permettent de visiter tous les types de nœud dans l' AST.

Un plugin ne doit jamais appeler une fonction de visite de façon directe ,il faut toujours passer par une fonction qui appartient a la classe ASTConsumer qui appellera la fonction de visite correcte en arrière-plan, et chaque fonction de visite doit retourner 'true' pour continuer a parcourir l'AST .

Notre objectif est de détecter toutes les fonctions impures du coup on a besoin de visiter toutes les variables globales à l'aide de VisitVarDecl et toutes les fonctions à l'aide de VisitFunctionDecl , encore plus loin on doit visiter et parcourir les contenus de chaque fonction pour vérifier si on a un appel vers l'extérieur c'est-à -dire vers une variable globale ou vers une autre fonction différente de la fonction elle-même , on peut visiter tous les appels à l'aide de VisitDeclRefExpr.

Si on résume, cette classe contient 3 fonctions ,et qui sont :

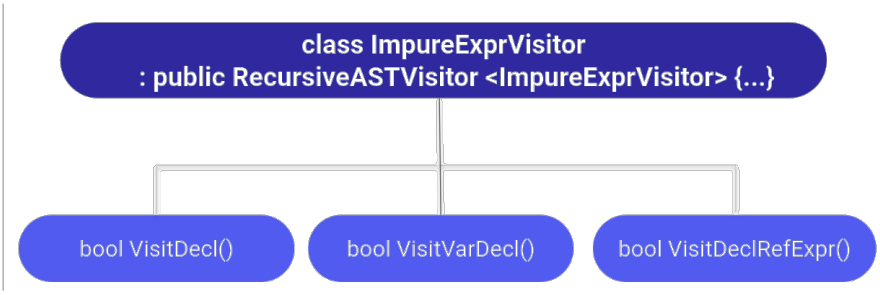

FIGURE 4 – ASTVisitor

- **VisitDecl :** qui nous permet de visiter tous les Decl ,et vérifier à l'aide de **dyn_cast** si le decl est un FunctionDecl ou non, afin de stocker toutes les fonctions dans un vector.
- **VisitVarDecl :** pour chaque VarDecl elle vérifie si la variable déclarée est globale ou pas , et la stocker dans un vector .
- **VisitDeclRefExpr :** cette fonction est appelée pour chaque DeclrefExpr dans l'AST et le stocker dans un vector .

5.3 Exécution du Plugin :

- **load :** Pour exécuter un plugin, la librairie dynamique contenant le registre de plugin doit être chargée via la commande -load, ainsi on demande à clang de charger la librairie impureplugin.so et d'utiliser le plugin nommé impurewarnings.
- **Xclang :** En clang, l'argument Xclang indique que lors de la compilation ce paramètre permet de transférer les arguments du driver vers le compilateur.

5.4 Teste de fiabilité du Plugin

On teste notre plugin à partir du fichier source qui représente un exemple d'application de notre plugin :

```
1 #include <stdio.h>
2
3 int global_variable = 99;
4
5 int pure(void)
```

```
6  {
7      return 5;
8  }
9
10 int impure()
11 {
12     global_variable = 9;
13     printf("LL\n");
14     return 0;
15 }
16
17 int main(int arcg, char ** argv)
18 {
19     pure();
20     impure();
21     return 0;
22 }
```

Et on obtient l'AST généré par notre plugin ci-dessous :

```
================
Functions
================
==== Exploring pure
ReturnStmt 0x558fca2b5490
`-IntegerLiteral 0x558fca2b5470 'int' 5
==== Exploring impure
BinaryOperator 0x558fca2b55f0 'int' '='
|-DeclRefExpr 0x558fca2b55b0 'int' lvalue Var 0x558fca2b2ba8 'global_variable' 'int'
`-IntegerLiteral 0x558fca2b55d0 'int' 9
CallExpr 0x558fca2b56b0 'int'
|-ImplicitCastExpr 0x558fca2b5698 'int (*)(const char *, ...)' <FunctionToPointerDecay>
| `-DeclRefExpr 0x558fca2b5610 'int (const char *, ...)' Function 0x558fca2a2680 'printf' 'int (const char *, ...)'
`-ImplicitCastExpr 0x558fca2b56f0 'const char *' <NoOp>
  `-ImplicitCastExpr 0x558fca2b56d8 'char *' <ArrayToPointerDecay>
    `-StringLiteral 0x558fca2b5630 'char [4]' lvalue "LL\n"
t.c:10:1: warning: impure is an impure function
int impure()
^
ReturnStmt 0x558fca2b5728
`-IntegerLiteral 0x558fca2b5708 'int' 0
==== Exploring main
CallExpr 0x558fca2b59e0 'int'
`-ImplicitCastExpr 0x558fca2b59c8 'int (*)(void)' <FunctionToPointerDecay>
  `-DeclRefExpr 0x558fca2b5980 'int (void)' Function 0x558fca2b53d0 'pure' 'int (void)'
t.c:17:1: warning: main is an impure function
int main(int arcg, char ** argv)
^
CallExpr 0x558fca2b5a60 'int'
`-ImplicitCastExpr 0x558fca2b5a48 'int (*)()' <FunctionToPointerDecay>
  `-DeclRefExpr 0x558fca2b5a00 'int ()' Function 0x558fca2b5510 'impure' 'int ()'
t.c:17:1: warning: main is an impure function
ReturnStmt 0x558fca2b5aa0
`-IntegerLiteral 0x558fca2b5a80 'int' 0
================
Global Variables
================
VarDecl 0x558fca28dc20 </usr/include/x86_64-linux-gnu/bits/libio.h:319:1, col:29> col:29 _IO_2_1_stdin_ 'struct _IO_FILE_plus':'struct _IO_FILE_plus' extern
VarDecl 0x558fca28dcf0 </usr/include/x86_64-linux-gnu/bits/libio.h:320:1, col:29> col:29 _IO_2_1_stdout_ 'struct _IO_FILE_plus':'struct _IO_FILE_plus' extern
VarDecl 0x558fca28dd78 </usr/include/x86_64-linux-gnu/bits/libio.h:321:1, col:29> col:29 _IO_2_1_stderr_ 'struct _IO_FILE_plus':'struct _IO_FILE_plus' extern
VarDecl 0x558fca29d760 </usr/include/stdio.h:135:1, col:25> col:25 stdin 'struct _IO_FILE *' extern
VarDecl 0x558fca29d7f0 </usr/include/stdio.h:136:1, col:25> col:25 stdout 'struct _IO_FILE *' extern
VarDecl 0x558fca29d880 </usr/include/stdio.h:137:1, col:25> col:25 stderr 'struct _IO_FILE *' extern
VarDecl 0x558fca2b17a8 </usr/include/x86_64-linux-gnu/bits/sys_errlist.h:26:1, col:12> col:12 sys_nerr 'int' extern
VarDecl 0x558fca2b1e10 </usr/include/x86_64-linux-gnu/bits/sys_errlist.h:27:1, col:38> col:26 sys_errlist 'const char *const []' extern
VarDecl 0x558fca2b2ba8 <t.c:3:1, col:23> col:5 used global_variable 'int' cinit
  IntegerLiteral 0x558fca2b2c10 <col:23> 'int' 99
================
DeclRefExpr
================
DeclRefExpr 0x558fca2b55b0 'int' lvalue Var 0x558fca2b2ba8 'global_variable' 'int'
==============
t.c:12:2: warning: global_variable is a global variabl
        global_variable = 9;
        ^
4 warnings generated.
```

FIGURE 5 – Résultats

6 Parallélisation via RPC

6.1 Remote Procédure Calls (RPC) principes :

Le principe est de lancer n processus qui peuvent s'exécuter en n coeur différents et de permettre aux différents processus d'échanger des traitements en procédant à des appels de fonctions à distance. On a donc une fonction qui permet d'appeler les fonctions à distance (c'est ce qu'on appelle RPC, Remote Procedure Call), il faut juste appeler MPIX_Offload qui prend en paramètres :

— Les paramètres (buffers) : c'est une liste de pointeurs.
— Leurs types : datatype .
— Le nombre de paramètres : sendcount.
— La valeur de retour : recvbuf c'est le pointeur vers la valeur de retour.
— Le type de la valeur de retour : recvtype.
— Le nom de la fonction qu'on veut appeler à distance : fname.
— Processus MPI sur lequel on veut appeler : dest.

Ce modèle d'exécution permet de s'affranchir de la contrainte de mémoire partagée, en effet en faisant appel à la fonction MPIX_Offload ce qui est intéressant est d'appeler la fonction func spécifiée dans les paramètres à distance sur un autre rang.

6.2 Le lien entre le plugin et la capacité d'appeler les fonctions à distance :

Notre plugin détecte les fonctions pures, qui sont des fonctions indépendantes aux autres fonctions, c'est-à-dire leur résultat ne dépend pas des résultats des autres fonctions (ne dépend qu'elle-même car elle répond toujours la même chose pour les mêmes paramètres d'entrée), et le plus important c'est qu'elles sont indépendantes d'où elles ont été appelées (une fonction pure appelée localement ou à distance, c'est pareil, et c'est l'intérêt de cette fonction), et l'objectif est de faire appel à ces fonctions à distance, via un exemple RPC qui est tout un système qui permet de faire ces appels en parallèle (sachant que le principe d'un compilateur est généralement de générer les fonctions d'un programme en séquentiel), cela nous apporte un gain de temps d'exécution du programme ce qui améliore ses performances.Pour cela, on a créé ce modèle RPC, qui nous permet de lancer des fonctions à distance, et la fonction MPIX_Offload est l'outil qui va nous permettre d'appeler ces fonctions à distance, via d'autres processus MPI, qui s'exécutent dans d'autres coeur.

Deux choses complémentaires à retenir :

Le plugin est l'outil qui nous permet d'identifier les fonctions qui sont candidates à être appelé à distance.

La fonction MPIX_Offload est l'outil qui va nous permettre d'appeler ces fonctions candidate à distance.

6.3 Pourquoi mettre en place un Plugin :

Parce-qu'idéalement on aimerait aller plus loin et faire la parallélisation automatique, les fonctions pures seront détectées automatiquement et seront lancées en parallèle quand ils seront exécutés plusieurs fois dans le même programme (dans une boucle par exemple).

On prend comme exemple :

```
1
2 int pure_func(int a, int b)
3 {
```

```
4       return a*b;
5   }
6
7   main :
8    //traitement
9
10   int tab[128];
11   int i;
12   for(i=0; i<128; i++)
13   {
14       tab[i] = pure_func(i, i);
15   }
```

On peut tout à fait paralléliser cet appel, puisque les éléments de la boucle sont indépendants :

— la fonction pure_func est une fonction pure, elle ne dépend que de ses paramètres .

— la valeur de retour dans tab[i] on peut les récupérer indépendamment.

Cela est équivalant à : (avec l'utilisation de openMP)

```
1    #pragma omp parallel
2   for(i=0; i<128; i++)
3    {
4       tab[i] = pure_func(i, i);
5    }
```

En revanche, nous on veut faire un programme différent, on veut le paralléliser mais pas avec des threads dans un seul processus, mais dans 128 processus différents.
Pour cela, on va utiliser la fonction MPIX_Offload, pour faire appel à pure_func dans 128 processus différents :

```
1
2   for(i=0; i<128; i++)
3    {
4       void* buffers[2] = {&i, &i};
5       MPI_Datatype type[2] = {MPI_INT, MPI_INT};
6
7       MPIX_Offload(buffers,
8                    type,
9                    2,
10                   &tab[i],
11                   MPI_INT,
12                   "pure_func",
13                   i);
14   }
15
```

Comme ça ils seront appelés les uns après les autres car MPIX_Offload est bloquant, mais si on fait :

```
1    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
2   MPI_Comm_size(MPI_COMM_WORLD, &size);
3   MPI_REQUEST req[128];
4   if (rank == 0)
5   {
6       for(i=0; i<128; i++)
```

```
7       {
8           void* buffers[2] = {&i, &i};
9           MPI_Datatype type[2] = {MPI_INT, MPI_INT};
10
11          MPIX_Ioffload(buffers,
12                        type,
13                        2,
14                        &tab[i],
15                        MPI_INT,
16                        "pure_func",
17                        (i)%size, /*pour faire appeler la function dans des
    processus deffirent autant qu'on a dans notre machine*/
18                        &req[i]);
19      }
20
21      MPI_Waitall(req, 128, MPI_STATUES_IGNORE);
22
23  }
```

Ce qui est important est que le problème dans le parallélisme en utilisant l'openMP avec mémoire partagé est qu'on est coincé dans son noeud, car il utilise les threads qui partagent la même mémoire, mais avec l'utilisation des appels dans des processus différents,on peut adresser une mémoire plus grande et donc faire des traitements plus conséquents.

7 Conclusion

En arrivant à la fin de ce projet intéressant, qui nous fournit une grande variété en matière de notions et de techniques ,et ce qui nous a aussi permis de conclure que :

1. Les fonctions pures sont des fonctions qui offrent une grande stabilité et performance lors de l'implémentation des programmes complexes comparant avec les fonctions impures .
2. Alimenter la chaine de compilation CLANG-LLVM par de nouvelles fonctionnalités trace le point de départ pour rentrer dans le monde vaste des compilateurs.
3. Intégrer des plugins au compilateur permet d'améliorer leurs performances en jouant sur la mémoire réservée et le temps d'exécution.
4. La parallélisation va RPC offre un grand gain en matière de performances, et surtout lorsque les fonctions appelées à distance sont pures.

Références

[1] Development/clang plugins. https ://wiki.documentfoundation.org/Development/Clang$_p lugins$.

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

[3] Grégory Lerbret. Introduction à makefile. https ://gl.developpez.com/tutoriel/outil/makefile/.

[4] Hassane Moustapha. Programmation fonctionelle avec javascript. https ://medium.com/hkairi/.

[5] Jimmy Wales-Larry Sanger. Wikipédia. https ://www.wikipedia.org.

[6] Manuel Selva. Introduction à l'infrastructure clang - llvm. page 19, 02 2017.

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

[8] wikipedia. Fonction pure. https ://fr.wikipedia.org/wiki/Fonction-pure.