Projet de compilation avancée : Vérification statique des séquences d'appels aux fonctions collectives MPI

Amira Akloul - Maxime Kermarquer - M2 CHPS

Table des matières

Introduction

Afin d'éviter un deadlock dans un programme MPI, les fonctions collectives doivent être réalisées dans le même ordre pour tous les processus. Le but de ce projet est de définir un pluging gcc qui analyse statiquement le code et prévient un potentiel deadlock.

Première partie

Vérification des séquences d'appels aux fonctions collectives MPI

Les fonctions collectives MPI doivent etre réalisées dans le meme ordre pour tous les processus. Si ce n'est pas le cas alors c'est un deadlock.

1) Définition d'une passe

On définit une passe qui permet de parcourir le graphe de flot de controle et les noeuds gimple. Et on vérifie si les noeuds d'appels (statments) sont des fonctions MPI.

La classe "passe" est une classe public qui hérite d'une autre classe public "gimple passe".

Dans cette classe on a une fonction **"gate"**, qui conditionne l'exécution de la passe. Et on a le coeur de la passe qui est la fonction **"exectute"** qui exécute toutes les fonctions qu'elle a dans ses instructions.

```
class my_pass : public gimple_opt_pass{
        public:
                my_pass (gcc::context *ctxt): gimple_opt_pass (my_pass_data, ctxt)
                {}
                my_pass *clone ()
                {
                        return new my_pass(g);
                }
                /* Fonction gate, qui conditionne l'exécution de la passe */
                bool gate (function *fun)
                {
                        //Conditions sur le #pragma ici
                        return true;
                }
                /* Coeur de la passe */
                unsigned int execute (function *fun)
                {
                        const char * fname = function_name(fun);
                        printf("\n--- EXECUTION DE LA PASSE ---\n");
                        printf("    Fonction analysée : %s\n",fname);
                        //Le champs "aux" nous est utile pour stocker quelle collectives MPI est dans le basic block.
                        //S'il y'en a pas le champs a la valeur LAST_AND_UNUSED_MPI_COLLECTIVE_CODE.
                        clean_aux_field(fun, LAST_AND_UNUSED_MPI_COLLECTIVE_CODE);
                        //Découpe les basic blocks qui ont plusieurs collectives MPI.
                        mpi_in_blocks(fun);
                        //Utilisation de la frontière de post-dominance pour detecter les dead-locks.
                        bitmap_head ipdf_set [LAST_AND_UNUSED_MPI_COLLECTIVE_CODE];
                        bitmap_and_pdf_it(fun, ipdf_set);
                        //Pour l'affichage du CFG
                        cfgviz_dump(fun,"", ipdf_set);
                        //On remet à 0 le champs "aux" pour les autres passes de la compilation.
                        clean_aux_field(fun, 0);
                        free_dominance_info( CDI_POST_DOMINATORS ) ;
                        return 0;
                }
};
```

FIGURE 1 – La classe pass

On définit la passe dans le plugin avec toutes les informations nécessaires. Et on l'enregistre dans le Pass_Manager avec la fonction **"register_callback"**.

```
/* Point d'entrée du plugin */
int plugin_init(struct plugin_name_args * plugin_info, struct plugin_gcc_version * version){

        /* Vérification de la version courante de GCC */
        if(!plugin_default_version_check(version, &gcc_version)) {
                fprintf(stderr, "Erreur version :\n\tPlugin incompatible avec cette version de GCC.\n");
                return 1;
        }

        //Définition de la passe
        struct register_pass_info my_pass_info;

        my_pass p(g);

        my_pass_info.pass = &p;
        my_pass_info.reference_pass_name = "cfg";
        my_pass_info.ref_pass_instance_number = 0;
        my_pass_info.pos_op = PASS_POS_INSERT_AFTER;

        //Enregistrement dans le gestionnaire de passe
        register_callback(plugin_info->base_name, PLUGIN_PASS_MANAGER_SETUP, NULL, &my_pass_info);


    return 0;
}
```

FIGURE 2 – Enregistrement de la passe dans le gestionnaire de passes

2) Définition d'une numérotation des collectives MPI

On définit une numérotation permettant de suivre la position d'une fonction collective MPI dans la séquence d'appel. ET vérifier si chaque chemin possède la même séquence d'appel, et donc que les fonctions MPI avec le même numéro doivent être les mêmes.

Pour cela on doit d'abord comparer les fonctions du code au fonctions collectives MPI qui sont dans le fichier "MPI_collectives.def".

```
/* Enum to represent the collective operations */
enum mpi_collective_code {
#define DEFMPICOLLECTIVES( CODE, NAME ) CODE,
#include "MPI_collectives.def"
        LAST_AND_UNUSED_MPI_COLLECTIVE_CODE
#undef DEFMPICOLLECTIVES
} ;

/* Name of each MPI collective operations */
#define DEFMPICOLLECTIVES( CODE, NAME ) NAME,
const char *const mpi_collective_name[] = {
#include "MPI_collectives.def"
} ;
#undef DEFMPICOLLECTIVES
```

FIGURE 3 – Renvoi du numéro de la fonction MPI

```
2 DEFMPICOLLECTIVES( MPI_INIT, "MPI_Init" )
3 DEFMPICOLLECTIVES( MPI_FINALIZE, "MPI_Finalize" )
4 DEFMPICOLLECTIVES( MPI_REDUCE, "MPI_Reduce" )
5 DEFMPICOLLECTIVES( MPI_ALL_REDUCE, "MPI_Allreduce" )
6 DEFMPICOLLECTIVES( MPI_BARRIER, "MPI_Barrier" )
```

FIGURE 4 – MPI_collectives.def

Certaines fonctions qui on été utilisées pour parcourir le cfg et vérifier les appels MPI :

enum mpi_collective_code is_mpi_call(gimple * stmt, int bb_index) : Vérifie si le statement est un appel à une collective MPI et renvoie le code de cet appel.

void read_and_mark_mpi(function *fun) : Met à jour le champs "aux" des basic blocks avec le code de l'appel à la collective MPI.

```
enum mpi_collective_code is_mpi_call( gimple * stmt, int bb_index)
{
        enum mpi_collective_code returned_code ;

        returned_code = LAST_AND_UNUSED_MPI_COLLECTIVE_CODE ;

        if (is_gimple_call (stmt))
        {
                tree t ;
                const char * callee_name ;
                int i ;
                bool found = false ;

                t = gimple_call_fndecl( stmt ) ;
                callee_name = IDENTIFIER_POINTER(DECL_NAME(t)) ;

                i = 0 ;
                while ( !found && i < LAST_AND_UNUSED_MPI_COLLECTIVE_CODE )
                {
                        if ( strncmp( callee_name, mpi_collective_name[i], strlen(mpi_collective_name[i] ) ) == 0 )
                        {
                                found = true ;
                                returned_code = (enum mpi_collective_code) i ;
                        }
                        i++ ;
                }
        }

        return returned_code;
}
```

FIGURE 5 – Fonction : is_mpi_call

Si dans un même basic block il y a deux appels MPI qu'on est entrain de vérifier ; on ne saura pas si c'est le premier ou le deuxième MPI call qui a planté. Pour avoir une meilleure analyse, on modifie le CFG en divisant le basic block au niveau de l'appel MPI.

```
/* Point d'entrée du plugin */
int plugin_init(struct plugin_name_args * plugin_info, struct plugin_gcc_version * version){

        /* Vérification de la version courante de GCC */
        if(!plugin_default_version_check(version, &gcc_version)) {
                fprintf(stderr, "Erreur version :\n\tPlugin incompatible avec cette version de GCC.\n");
                return 1;
        }

        //Définition de la passe
        struct register_pass_info my_pass_info;

        my_pass p(g);

        my_pass_info.pass = &p;
        my_pass_info.reference_pass_name = "cfg";
        my_pass_info.ref_pass_instance_number = 0;
        my_pass_info.pos_op = PASS_POS_INSERT_AFTER;

        //Enregistrement dans le gestionnaire de passe
        register_callback(plugin_info->base_name, PLUGIN_PASS_MANAGER_SETUP, NULL, &my_pass_info);


    return 0;
}
```

FIGURE 6 – Découpage d'un basic block

Les fonctions utilisées pour savoir s'il y a plusieurs appels MPI call dans un meme block et qui permettent le découpage de celui ci :

int get_nb_mpi_calls_in_bb(basic_block bb) : Renvoie le nombre d'appels MPI dans un basic block.

bool check_multiple_mpi_calls_per_bb(function *fun) : Vérifie s'il y a dans le CFG des basic blocks avec plusieurs appels MPI.

void split_multiple_mpi_calls(function * fun) : Divise les basic blocks ayant plusieurs appels au collectives MPI.

void mpi_in_blocks(function * fun) : Vérifie et découpe les basic blocks qui ont plusieurs appels MPI puis met à jour le champs "aux" des baic blocks.

3) Frontiere post dominance itérée

On utilise la notion de frontiere post dominance itérée pour trouver le noeud contenant le fork à l'origine du conflit.

Quelques fonctions utilisées pour cette partie :

void bitmap_post_dominance_frontiers (bitmap_head *frontiers, function * fun)

void bitmap_set_post_dominance_frontiers (bitmap_head node_set, bitmap_head *pdf, bitmap_head * pdf_set, function * fun)

void iterated_post_dominance(bitmap_head pdf_node_set, bitmap_head *pdf, bitmap_head * ipdf_set, function * fun)

Dans la fonction : void bitmap_and_pdf_it(function * fun, bitmap_head ipdf_set[])

- On calcule la frontière de post-dominance de tous les nœuds.
- Si un basic block contient la collective MPI i, on change la valeur du bitmap ; on regroupe les basic blocks de la collectives i ; et on cacule la pdf de cet ensemble de basic blocks.
- On calcule la frontiere de post-dominance itérée de cette ensemble de nœud. Les noeuds appartement à cet ensemble sont les basic blocks pouvant causés des deadlocks.

4) Affichage d'un warning

On affiche un warning avec toutes les informations de l'origine du problème : (Collectives MPI qui posent problème, numéro de ligne ou noeud gimple en conflit, l'origine du conflit).

```
void issue_warnings (bitmap_head ipdf_set[], function * fun)
{
        basic_block bb;
        gimple_stmt_iterator gsi;
        gimple *stmt;

        int i;

        for(i=0; i<LAST_AND_UNUSED_MPI_COLLECTIVE_CODE; i++)
        {
                if ( bitmap_count_bits( &ipdf_set[i] ) == 0 )
                {
                        continue;
                }
                printf("\n\n---------- Problems for %s ---------\n\n", mpi_collective_name[i]);

                FOR_EACH_BB_FN( bb, fun )
                {
                        if(bitmap_bit_p(&ipdf_set[i], bb->index))
                        {
                                gsi = gsi_start_bb(bb);
                                stmt = gsi_stmt(gsi);

                                printf("/!\\ /!\\ /!\\ Basic Block  %d  (line %d) might cause an issue\n", bb->index, gimple_lineno(stmt));
                        }
                }
        }
}
```

FIGURE 7 – Afichage du warning

Deuxième partie
Instrumentation dynamique du code

Une fois que le deadlock a été detécté, au lieu de le laisser se produire, on va arreter le programme.

Pour se faire, on rajoute avant les deadlock potentiels une partie de code qui vérifie si le deadlock va se produire.

On a du pour cela creer un noeud gimple et le placer avant le noeud gimple qui causait le deadlock.

Pour définir le noeud gimple, on construit un appel de fonction puis les arguments de cette fonction. Les arguments de cette fonction contiennent les informations sur l'erreur comme :

— Le nom de la collective provoquant le deadlock.

— La ligne ou se trouve la collective.

— La liste des noeuds gimple qui ont pu amené a ce deadlock.

Au final, on affiche un message d'erreur détaillé avec toutes les informations recueillies lors de l'analyse statique.

Troisième partie
Gestion des directives

On définit une directive (pragma) qui permet de sélectionner les fonctions à analyser.On peut activer l'analyse pour une ou un ensemble de fonctions.

— Pour cela on doit faire une premiere passe pour detecter les pragma.

Conclusion

Dans ce projet on a appris comment :
— Creer un plugin.
— Inserer une passe et l'enregistrer.
— Parcourir un graphe de flot de controle. Verifier si les basic blocks d'un CFG croisent une anomalie.
— Vérifier si un statment est un appel de fonction MPI.
— Trouver des forks problématiques grace à la frontiere de post dominance itérée.
— Afficher un warning pour l'utilisateur lors de la compilation si un potentiel deadlock a été detecté.
— Sortir du programme avec un message d'erreur pour faire éviter à l'utilisateur le deadlock.