



Implémentation d'un pugin GCC détectant des appels bloquant de collectives MPI

6 novembre 2023

Table des matières

0.1	Introduction	2
0.1.1	Problématique	2
0.1.2	Fondements du Projet	3
0.1.3	Problématique	3
0.2	Partie 1 - Vérification de la séquence d'appel aux fonctions collectives MPI	4
0.2.1	Manipulation des Blocs de Base pour les Appels MPI	4
0.2.2	Implémentation de la Visualisation CFG	5
0.2.3	Analyse des Frontières de Post-Dominance Itérée	7
0.2.4	Initialisation des Bitmaps	7
0.2.5	Construction des Frontières de Post-Dominance	7
0.2.6	Suppression des Boucles	7
0.2.7	Impression des Résultats	8
0.2.8	Intégration et Orchestration du Plugin GCC	8
0.3	Partie 2 - Gestion des directives	9
0.3.1	Gestion des Directives Pragma pour l'Analyse Statique	9
0.4	Conclusion	11

0.1 Introduction

0.1.1 Problématique

Dans l'écosystème du calcul haute performance (HPC), la gestion efficace de la communication entre processus parallèles est un défi critique. Le Message Passing Interface (MPI), un modèle de programmation standard pour le calcul parallèle, est intrinsèquement vulnérable aux deadlocks lors de l'utilisation incorrecte des fonctions collectives. Ces blocages peuvent causer des inefficacités majeures, voire l'arrêt total d'applications HPC, entraînant des pertes de temps et de ressources considérables.

Notre projet s'attaque à la problématique de la vérification statique des séquences d'appels aux fonctions collectives MPI, avec pour objectif de prévenir les deadlocks. Nous avons mis en place une méthode qui analyse statiquement le code source MPI pour s'assurer que l'ordre des appels aux fonctions collectives est correct et cohérent à travers les différents chemins d'exécution. Cette méthode est conçue pour être appliquée à la compilation, permettant ainsi de détecter les erreurs potentielles avant l'exécution du programme.

Ce rapport se concentre sur les deux premières parties de notre travail, où nous avons développé un **plugin pour l'analyse statique** au lieu de modifier directement le compilateur GCC. Cette approche présente l'avantage de simplifier considérablement le processus de vérification, puisque les plugins sont plus faciles à développer et à maintenir que les changements au cœur même du compilateur. Notre plugin examine le graphe de flux de contrôle (CFG) d'une fonction à la recherche de séquences d'appels MPI potentiellement problématiques.

Dans ce rapport, nous abordons les deux premières phases cruciales de notre projet. La première phase consiste à établir une base théorique solide pour l'analyse statique, développant un algorithme capable de parcourir le graphe de flux de contrôle (CFG) et d'identifier les séquences d'appels MPI. La deuxième phase concerne la conception et l'implémentation d'un plugin pour le compilateur qui applique cet algorithme, fournissant une solution pratique sans nécessiter une modification invasive du compilateur GCC. Ce plugin sert d'outil d'analyse préventive, en avertissant les développeurs de possibles erreurs de séquence d'appels collectives qui pourraient entraîner des deadlocks. Les détails de notre approche, de la mise en œuvre du plugin, et les résultats des tests de validation seront exposés dans les sections suivantes, démontrant ainsi l'efficacité de notre outil dans la détection proactive des erreurs dans les codes MPI.

D'après le sujet, la **première partie** de la vérification de la séquence d'appel aux fonctions collectives MPI se compose de plusieurs sous-parties importantes :

- **Passe de compilation** pour trouver les appels aux fonctions collectives MPI : Il s'agit de déterminer si chaque chemin du CFG contient la même séquence d'appels aux fonctions collectives MPI et d'émettre un avertissement à l'utilisateur si ce n'est pas le cas.
- **Définition d'une numérotation** pour suivre la position d'une fonction collective MPI : Cela implique de choisir une représentation permettant de vérifier rapidement si chaque chemin possède la même séquence d'appels.
- Utilisation de la notion de **frontière de post-dominance itérée** (PDF+) : Cela sert à retrouver le nœud d'origine du deadlock potentiel en utilisant les notions de dominance et post-dominance fournies par GCC.
- **Affichage d'un warning à l'utilisateur** : Avec les informations recueillies, un avertissement est émis pour permettre à l'utilisateur de retrouver l'origine du deadlock potentiel dans son code

0.1.2 Fondements du Projet

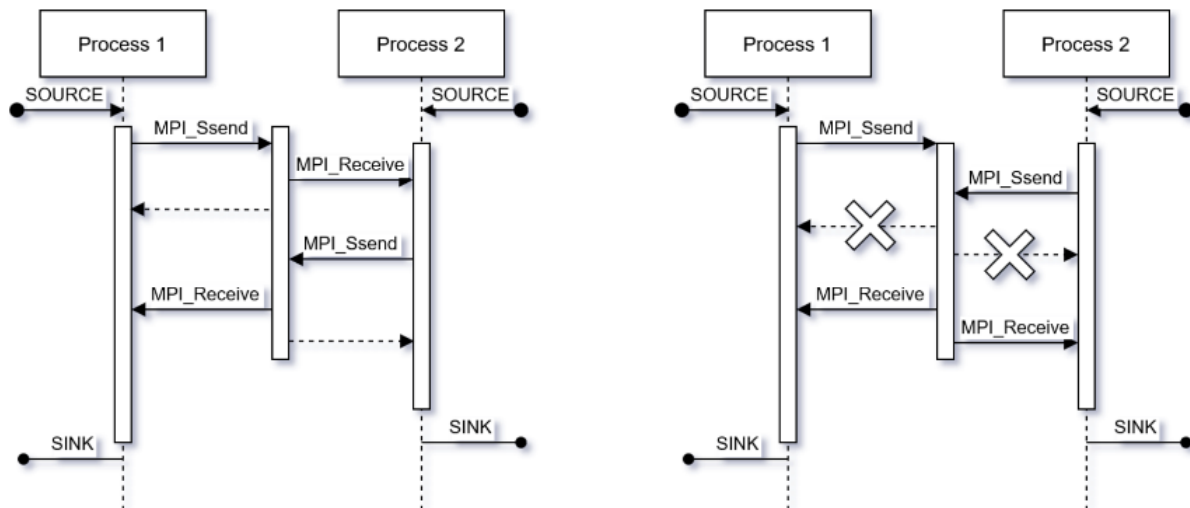
Le travail présenté dans ce rapport prend racine dans les pratiques et les principes que nous avons appliqués et affinés au cours des travaux pratiques en classe. Ces expériences éducatives ont jeté les fondements de notre compréhension du Message Passing Interface (MPI) et de la programmation parallèle, nous permettant de construire un cadre solide pour notre projet. L'outil que nous avons développé est une adaptation et une extension des concepts que nous avons explorés en classe, méticuleusement ajustés pour répondre aux exigences spécifiques et aux défis inhérents à la vérification statique des séquences d'appels aux fonctions collectives MPI. Cette évolution d'un contexte pédagogique à une application plus vaste illustre l'importance de la théorie mise en pratique, et comment l'expérience acquise en laboratoire peut être transposée à des problèmes réels et complexes dans le domaine du calcul haute performance.

0.1.3 Problématique

L'interface MPI est une norme de passage de messages standardisée conçue pour fonctionner sur des architectures de calcul parallèles. La norme MPI définit la syntaxe et la sémantique des routines de bibliothèques qui sont utiles à un large éventail d'utilisateurs écrivant des programmes de passage de messages portables en C, C++ et Fortran.

Lorsque qu'on effectue des échanges de messages entre deux processus, il faut qu'ils soient simultanément dans une situation d'envoie-écoute pour que le message soit reçu et que les deux processus puissent reprendre leur fonctionnement.

Lorsque plusieurs processus envoient des messages, on peut être dans une situation où les processus s'entrebloquent et s'attendent mutuellement, créant un blocage définitif du programme appelé **dead-lock**.



Dans notre projet, nous nous intéressons aux deadlock créés par des communications collectives.

0.2 Partie 1 - Vérification de la séquence d'appel aux fonctions collectives MPI

0.2.1 Manipulation des Blocs de Base pour les Appels MPI

La détection et la gestion appropriée des appels aux fonctions collectives MPI dans le CFG nécessitent une manipulation soignée des blocs de base. Le fichier `mpi_collectives.cpp` contient des fonctions essentielles pour cette tâche :

Détection et Séparation des Blocs de Base

Nous décidons de nommer les collectives analysées grâce à un **enum** :

```
enum mpi_collective_code print_if_mpi_coll(const gimple* stmt)
{
    if (is_gimple_call(stmt)) {
        const tree t = gimple_call_fndecl(stmt);
        const char* ident = IDENTIFIER_POINTER(DECL_NAME(t));
        for (int i = 0; i < LAST_AND_UNUSED_MPI_COLLECTIVE_CODE; i++) {
            if (strcmp(mpi_collective_name[i], ident) == 0) {
                printf("\tMPI_COLLECTIVE:_%s'_(code:_%d)\n", ident, i);
                return (enum mpi_collective_code)i;
            }
        }
    }

    return LAST_AND_UNUSED_MPI_COLLECTIVE_CODE;
}
```

La fonction **split_bb_if_necessary** parcourt les instructions à l'intérieur d'un BB à la recherche d'appels MPI.

S'il y en a plus de 2 dans un même basic block, alors on retourne true :

```
bool split_bb_if_necessary(basic_block bb)
{
    int count = 0;
    for (gimple_stmt_iterator gsi = gsi_start_bb(bb); !gsi_end_p(gsi); gsi_next(&
        const gimple* stmt = gsi_stmt(gsi);

        if (is_gimple_call(stmt)) {
            const tree t = gimple_call_fndecl(stmt);
            const char* ident = IDENTIFIER_POINTER(DECL_NAME(t));

            for (int i = 0; i < LAST_AND_UNUSED_MPI_COLLECTIVE_CODE; i++) {
                count += (strcmp(mpi_collective_name[i], ident) == 0);

                if (count >= 2) {
                    gsi_prev(&gsi);
                    gimple* prev_stmt = gsi_stmt(gsi);
                    split_block(bb, prev_stmt);
                    return true;
                }
            }
        }
    }
}
```

```

    }
    }
}

return false;
}

```

Itération et Séparation sur les Blocs de Base

split_on_mpi_collectives est une fonction qui itère sur tous les BB d’une fonction et appelle **split_bb_if_necessary** pour chaque BB. Elle marque aussi les BB avec le type d’appel MPI qu’ils contiennent pour une analyse ultérieure

```

void split_on_mpi_collectives(basic_block bb, function *fun)
{
    FOR_EACH_BB_FN(bb, fun)
    {
        bb->aux = (void*)LAST_AND_UNUSED_MPI_COLLECTIVE_CODE;
        if (split_bb_if_necessary(bb))
            printf("\tSplit the block %02d\n", bb->index);

        for (gimple_stmt_iterator gsi = gsi_start_bb(bb); !gsi_end_p(gsi);
             const enum mpi_collective_code code = print_if_mpi_coll(g
             if (code != LAST_AND_UNUSED_MPI_COLLECTIVE_CODE)
                 bb->aux = (void*)code;
        }
    }
}

```

Ces fonctions garantissent que chaque BB contient au maximum un appel MPI, ce qui simplifie grandement l’analyse des séquences d’appels. De plus, elles fournissent une base pour la génération de warnings si des séquences incohérentes sont détectées.

0.2.2 Implémentation de la Visualisation CFG

Notre plugin GCC utilise le fichier **cfgviz.cpp** pour générer des représentations visuelles du Control Flow Graph des fonctions analysées, facilitant ainsi la compréhension des flux de contrôle et la détection des séquences d’appels MPI incohérentes. Voici les détails de l’implémentation :

Génération de Noms de Fichiers

cfgviz_dump est la fonction publique appelée par le reste du plugin pour effectuer la visualisation. Elle génère le nom de fichier, ouvre le fichier de sortie, appelle **cfgviz_internal_dump** pour écrire le contenu et ferme le fichier. Les messages de journalisation aident l’utilisateur à comprendre ce que le plugin fait à ce moment.

Nettoyage des Données Auxiliaires

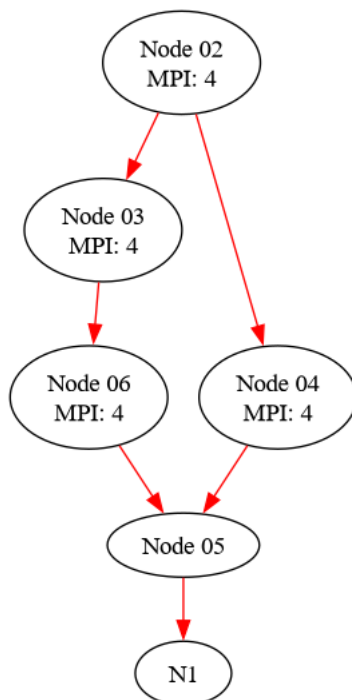
Enfin, `clear_all_bb_aux` parcourt à nouveau tous les blocs de base pour effacer les informations auxiliaires utilisées lors de la génération du CFG. Cette étape est essentielle pour éviter les interférences avec d'autres parties du plugin ou d'autres plugins qui pourraient être utilisés simultanément.

La structure des fichiers `.dot` générés permet une visualisation claire des chemins possibles dans une fonction et de l'endroit où les appels MPI sont effectués. En utilisant des outils comme Graphviz, les développeurs peuvent rapidement identifier des chemins problématiques et prendre les mesures correctives appropriées avant l'exécution du programme.

Voici un exemple du résultat obtenu après traitement de cette fonction

```
void mpi_call(int c)
{
    MPI_Barrier(MPI_COMM_WORLD);

    if(c<10)
    {
        printf("je suis dans le if (c=%d)\n", c);
        MPI_Barrier(MPI_COMM_WORLD);
        MPI_Barrier(MPI_COMM_WORLD);
    }
    else
    {
        printf("je suis dans le else (c=%d)\n", c);
        MPI_Barrier(MPI_COMM_WORLD);
    }
}
```



0.2.3 Analyse des Frontières de Post-Dominance Itérée

On peut définir la **dominance**

$X \gg Y \iff X \text{ DOM } Y$ (X apparaît sur tous les chemins de $\text{SRC} \rightarrow Y$) ($\text{SRC} \rightarrow X \rightarrow Y$)

Puis la **post dominance**

$X \gg_p Y \iff X \text{ PDOM } Y$ (X apparaît sur tous les chemins de $Y \rightarrow \text{SINK}$) ($Y \rightarrow X \rightarrow \text{SINK}$)

i.e (X est un passage obligé en dessous (destination d'un chemin depuis) de Y)

L'analyse PDF+ est fondamentale pour localiser les points de convergence des chemins d'exécution dans le CFG, ce qui permet d'identifier les endroits où les chemins divergents avec des appels MPI peuvent se rejoindre. Le fichier `frontiers.cpp` met en œuvre cette analyse en utilisant les structures de données et algorithmes fournis par GCC.

0.2.4 Initialisation des Bitmaps

La fonction **bitmap_init** crée et initialise un tableau de bitmaps pour chaque bloc de base du CFG actuel. Chaque bitmap est utilisé pour enregistrer les blocs de base qui post-dominent un autre bloc de base, ce qui est crucial pour l'analyse PDF+.

On s'inspire des fonctions définies dans **bitmap.h** dans le code source de GCC afin d'adapter **calculate_dominance_info**. On implémente d'abord le type stockant les **frontiers**

```
bitmap_initialize (bitmap head, bitmap_obstack *obstack CXX_MEM_STAT_INFO)
{
    head->first = head->current = NULL;
    head->indx = head->tree_form = 0;
    head->padding = 0;
    head->alloc_descriptor = 0;
    head->obstack = obstack;
    if (GATHER_STATISTICS)
        bitmap_register (head PASS_MEM_STAT);
}
```

0.2.5 Construction des Frontières de Post-Dominance

Ainsi à l'aide du CFG exprimé en bitmaps, l'idée est de changer

- `preds` en `succs`
- `CDI_DOMINATORS` en `CDI_POST_DOMINATORS`
- `ENTRY_BLOCK_PTR_FOR_FN` en `EXIT_BLOCK_PTR_FOR_FN`

Ainsi **post_dom_frontiers** calcule les frontières de post-dominance pour chaque bloc de base. Si un bloc de base a plusieurs successeurs, cette fonction détermine le post-dominant immédiat et ajoute les blocs de base intermédiaires à la frontière de post-dominance du bloc en cours d'analyse.

0.2.6 Suppression des Boucles

La fonction **removeloop_cfg2** utilise un algorithme de **parcours en profondeur** pour nettoyer la configuration du CFG en supprimant les boucles. Cette étape simplifie l'analyse en préparant le CFG pour une évaluation plus directe des séquences d'appels MPI.

0.2.7 Impression des Résultats

Après le calcul des frontières de post-dominance et le nettoyage du CFG, des fonctions d'impression sont utilisées pour afficher les résultats. Cela fournit une sortie en console qui peut être utilisée pour vérifier les résultats de l'analyse PDF+.

Ces fonctions jouent un rôle critique dans la détection de configurations qui pourraient mener à des deadlocks, en marquant les points de convergence dans le CFG où les appels MPI doivent être examinés pour assurer une séquence cohérente.

0.2.8 Intégration et Orchestration du Plugin GCC

Le fichier `plugin.cpp` intègre les fonctionnalités définies dans les autres fichiers source et gère leur exécution au sein du pipeline de compilation GCC. Il définit la structure et le comportement de notre passe de compilation personnalisée :

Définition de la Compatibilité et de la Passe de Compilation

Au début du fichier, la variable `plugin_is_GPL_compatible` assure la compatibilité avec la licence GPL de GCC. La structure `my_pass_data` initialise les métadonnées pour notre passe de compilation personnalisée, y compris son nom et son positionnement dans le pipeline de passes de GCC.

Classe de la Passe de Compilation Personnalisée

La classe `my_pass` hérite de `gimple_opt_pass` et définit les méthodes pour le gate (portail) et l'exécution de la passe. La méthode `gate` permet de vérifier si la passe doit être exécutée pour une fonction donnée, et `execute` implémente la logique principale de notre analyse.

Analyse et Avertissements PDF+

Dans `execute`, nous invoquons les fonctions :

- pour séparer les blocs de base sur les appels MPI (`split_on_mpi_collectives`)
- calculer les informations de post-dominance (`calculate_dominance_info`)
- générer les frontières de post-dominance (`post_dom_frontiers`). Ces étapes sont essentielles pour identifier les chemins d'exécution problématiques.

Nettoyage des ressources

Avant de terminer la passe, le plugin nettoie les structures de données (`free_dominance_info`, `clear_all_bb_aux`) pour éviter les fuites de mémoire et les interférences avec d'autres passes.

0.3 Partie 2 - Gestion des directives

Dans la deuxième partie du projet, nous mettons l'accent sur l'intégration et l'utilisation des directives `pragma` pour contrôler le processus d'analyse statique. L'objectif est de fournir aux développeurs une interface pour spécifier quelles fonctions doivent être analysées par le plugin, renforçant ainsi la précision et la pertinence de l'analyse. Les directives `pragma` permettent une granularité fine dans l'analyse, en focalisant l'attention de l'outil sur les points spécifiques du code qui sont les plus susceptibles de contenir des erreurs de séquence d'appels MPI.

Le fichier **`pragma.cpp`** implémente le support des directives `pragma`, permettant aux utilisateurs de marquer les fonctions pour l'analyse

0.3.1 Gestion des Directives Pragma pour l'Analyse Statique

Enregistrement de la Passe et Gestion des Pragmas

La fonction **`plugin_init`** est le point d'entrée du plugin, où la passe personnalisée est enregistrée dans le système de passes de GCC (**`register_callback`**). Elle gère également l'ajout de la gestion des pragmas MPI (**`my_callback_mpicoll_register`**), permettant d'influencer le comportement du plugin selon les directives de l'utilisateur

Stockage des Fonctions Marquées

La liste **`pragma_instrumented_functions`** est déclarée pour stocker les noms des fonctions que l'utilisateur a spécifiées pour l'instrumentation. Cette liste est utilisée pour conserver une trace des fonctions qui doivent être analysées par le plugin.

Interprétation de la Directive Pragma

La fonction **`pragma_mpicoll_check`** est appelée par GCC lorsqu'il rencontre une directive `pragma` **`PragmaProjetCA mpicoll_check`**.

Elle analyse la syntaxe de la directive, s'assurant que le format est correct et que les parenthèses sont équilibrées. Si la directive est mal formée, un message d'erreur est affiché.

Validation et Enregistrement des Noms de Fonctions

Si la directive `pragma` est bien formée, les noms des fonctions spécifiées sont ajoutés à **`pragma_instrumented_functions`**.

S'il y a des doublons, des avertissements sont émis, mais les noms sont quand même conservés pour garantir que toutes les instances de directives seront considérées.

Enregistrement du Callback Pragma

Enfin, la fonction **`my_callback_mpicoll_register`** enregistre la directive `pragma` auprès de GCC. Cela garantit que lorsque GCC trouve une directive `pragma` correspondante dans le code source, notre fonction **`pragma_mpicoll_check`** sera exécutée.

Voici un exemple d'exécution avec ce fichier :

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#pragma ProjetCA mpicoll_check main
```

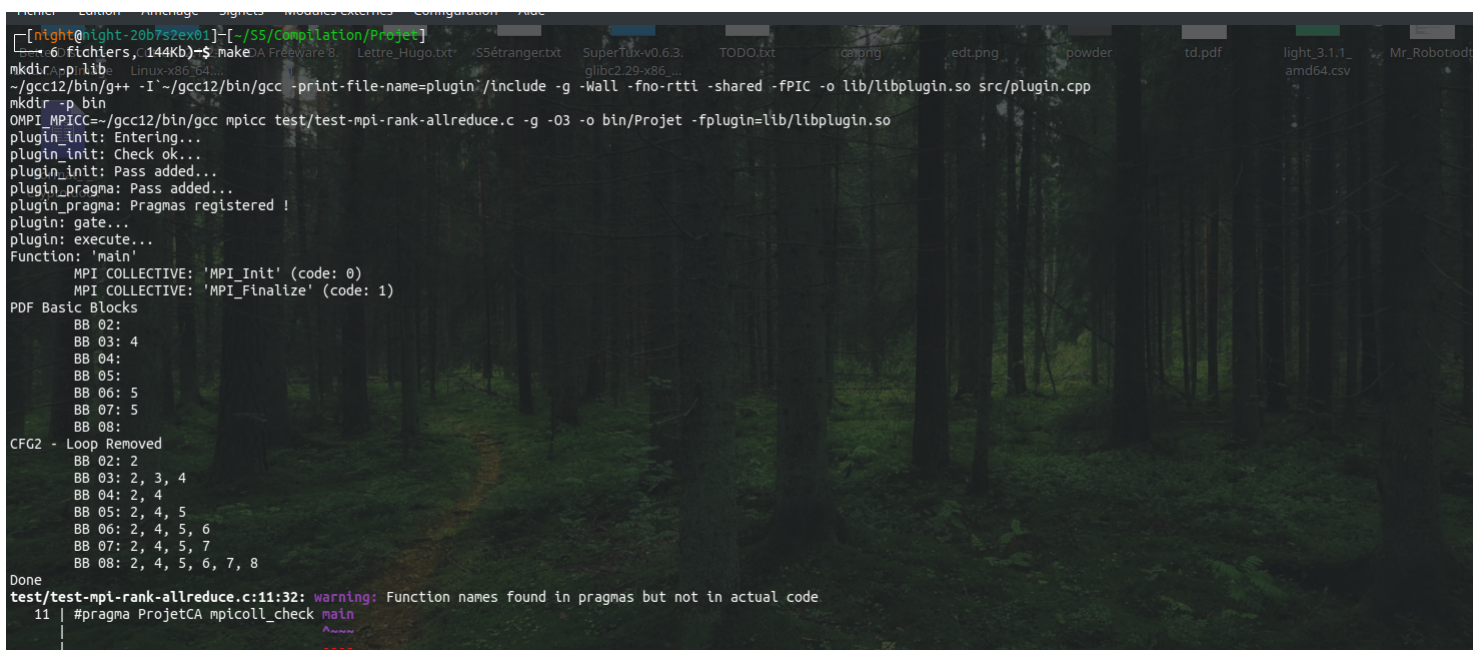
```

int main(int argc, char * argv[])
{
    MPI_Init(&argc, &argv);

    int P, N = 100;
    int me, i, sum, sum_loc = 0;
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    MPI_Comm_size(MPI_COMM_WORLD, &P);
    for( i = 1 + me*N/P ; i <= (me+1)*N/P ; i++ ){
        sum_loc += 1;
    }
    if (me != 3){
        MPI_Allreduce(&sum_loc, &sum, 1, MPI_INT,
                     MPI_SUM, MPI_COMM_WORLD);
    }else{
        printf("I 'm number 3, i 'm not working today !\n");
    }
    printf("1+    +%d = %d\n", N, sum);

    MPI_Finalize();
    return 1;
}

```



```

[night@night-20b752ex01]~/SS/Compilation/Projet
mkldir -p lib: Linux-x86_64...
~/gcc12/bin/g++ -I~/gcc12/bin/gcc -print-file-name=plugin/include -g -Wall -fno-rtti -shared -fPIC -o lib/libplugin.so src/plugin.cpp
mkldir -p bin
OMPI_MPICC=~/gcc12/bin/gcc mpicc test/test-mpi-rank-allreduce.c -g -O3 -o bin/Projet -fplugin=lib/libplugin.so
plugin_init: Entering...
plugin_init: Check ok...
plugin_init: Pass added...
plugin_pragma: Pass added...
plugin_pragma: Pragas registered !
plugin: gate...
plugin: execute...
Function: 'main'
MPI COLLECTIVE: 'MPI_Init' (code: 0)
MPI COLLECTIVE: 'MPI_Finalize' (code: 1)
PDF Basic Blocks
BB 02:
BB 03: 4
BB 04:
BB 05:
BB 06: 5
BB 07: 5
BB 08:
CFG2 - Loop Removed
BB 02: 2
BB 03: 2, 3, 4
BB 04: 2, 4
BB 05: 2, 4, 5
BB 06: 2, 4, 5, 6
BB 07: 2, 4, 5, 7
BB 08: 2, 4, 5, 6, 7, 8
Done
test/test-mpi-rank-allreduce.c:11:32: warning: Function names found in pragmas but not in actual code
11 | #pragma ProjetCA mpicoll_check main
    |                               ^~~~~

```

0.4 Conclusion

Au terme de ce projet, nous avons entrepris une analyse approfondie des séquences d’appels aux fonctions collectives MPI, avec pour objectif de prévenir les deadlocks dans les applications de calcul haute performance. Nous avons développé un plugin pour GCC qui permet de visualiser le graphe de flux de contrôle (CFG) et d’analyser les appels MPI à travers les chemins d’exécution. Bien que notre travail ait été marqué par des avancées significatives, notamment dans **l’analyse intra-procédurale et la gestion des directives pragma**, nous avons également rencontré des obstacles techniques.

Malheureusement, la génération de CFG est actuellement entravée par une erreur de segmentation que nous n’avons pas réussi à résoudre. Cela limite la capacité de l’outil à fournir une visualisation complète et à effectuer une vérification exhaustive des appels MPI. De même, l’émission complète des avertissements, cruciale pour guider les utilisateurs vers les points précis de divergence potentielle, n’a pas encore été entièrement implémentée.

Cependant, il est important de souligner que le reste du code fonctionne comme prévu. La capacité de marquer spécifiquement des fonctions pour l’analyse via des directives pragma est opérationnelle, et elle constitue un outil puissant pour les développeurs qui cherchent à optimiser leurs applications MPI.

Alors que ce projet a certainement ses défis, les fondations établies sont solides et prometteuses. La conception du plugin, les stratégies d’analyse adoptées, et l’interface utilisateur via pragma forment un cadre qui, avec un débogage et une amélioration continue, peut potentiellement transformer la manière dont les développeurs interagissent avec le MPI.

Dans les étapes futures, nous envisageons l’intégration de fonctionnalités supplémentaires, telles que l’analyse inter-procédurale et le support pour d’autres architectures de calcul parallèle. Avec ces améliorations, nous anticipons que notre outil sera non seulement un mécanisme de prévention des erreurs mais aussi un facilitateur pour l’optimisation des performances des applications HPC.