Rapport de projet de compilation avancée

Hugo Lallemant, Mickaël Saes-Vincensini

5 Novembre 2024

Contents

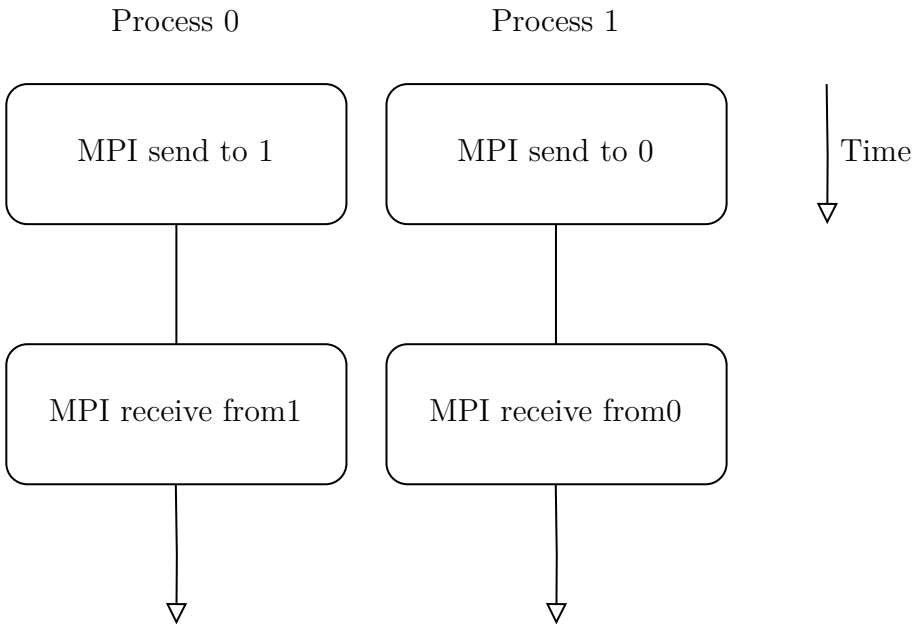


Figure 1: Schéma d'un deadlock dans une exécution de code `MPI`

1 Contexte et deadlocks MPI

L'objectif du projet est de réaliser un plugin pour le compilateur `GCC` (La version 12.2.0). Ce plugin analysera les positions des collectives `MPI` dans un code C, pour permettre d'identifier des potentiels *deadlocks* dans le code.

`MPI` (Message Passing Interface) est une norme standardisée qui permet de la communication de messages dans le contexte de calcul parallèle. `MPI` propose un ensemble de fonctions utilisables, parmi elles, certaines sont appelées des collectives. Ces collectives permettent de faire communiquer l'ensemble des processus entre eux. Si ces collectives ne sont pas appelées dans de certaines conditions, elles peuvent créer des *deadlocks*.

Un deadlock, ou verrou mortel en français, est une situation qui peut se produire dans un code parallèle, dans le cas où un ou plusieurs processus attendent indéfiniment, comme dans la figure 1 où les deux processus 0 et 1 sont bloqués car ils attendent un "receive" avant de continuer.

2 Solution et implémentation

La solution proposée est donc la réalisation d'un plugin `GCC`.

Le plugin réalisé va marcher via l'ajout d'une passe de compilation *gimple middle-end*, et va manipuler le *control flow graph* (CFG) généré par `GCC` des fonctions à analyser, et fait remonter, ou non à l'utilisateur la présence d'un potentiel *deadlock* dans son code, avec un warning `GCC`.

Pour ce faire, le plugin va tout d'abord s'intéresser à la présence ou non de collectives `MPI` dans un basic block du CFG, et séparer, si nécessaire, les basic blocks contenant plus d'une collective `MPI`. Ensuite, le plugin va calculer et stocker les rangs de toutes les collectives pour chaque basic block. Ces rangs permettent ensuite de calculer la frontière de post-dominance itérée par groupes de collectives de rangs identiques. On affichera

finalement un warning pour tous les ensembles dont la frontière de post-dominance itérée n'est pas vide.

2.1 Préparation du CFG

Pour travailler sur le CFG dans la suite, nous devons au préalable, séparer les basic blocks contenant plusieurs appels à des collectives `MPI`, puis stocker dans le champ auxiliaire de la structure des basic blocks des informations nous permettant par la suite de connaître le code de la collective présente ou non dans le basic block et de stocker le rang de chaque collective dans celui-ci..

2.1.1 Séparation des basic blocks contenant plus d'une collective `MPI`

Pour pouvoir étudier correctement le CFG, il est nécessaire qu'un basic block ne soit associé qu'à une seule et unique collective `MPI`. Or, par défaut, ce n'est pas le cas. Pour régler ce problème, nous allons séparer les basic blocks contenant plus d'une collective `MPI` en plusieurs basic blocks.

Pour cela, nous parcourrons chaque basic block, on compte le nombre de collectives `MPI` présentes en parcourant les gimple statements et en augmentant un compteur du nombre de collectives présentes dans le basic block. Pour chacun de ses statement, on appelera une fonction qui va récupérer, si le statement le permet, le nom de la fonction appelée, et comparer la chaîne de caractères au tableau des collectives `MPI`. Si c'est une collective `MPI`, on renvoie son code, sinon, on renvoie la taille du tableau des collectives. S'il y a plus d'une collective `MPI`, alors on va re parcourir les gimple statements du basic block, et dès que l'on détecte une collective `MPI`, on fait appel à la fonction `split_block`.

La figure 3a montre le résultat après séparation des basic block. On remarque que l'ancien basic block 4 est devenu les basic blocks 4 et 13.

2.1.2 Remplissage des champs aux

Chaque basic block contient un champ auxiliaire "aux" qui peut être rempli par un utilisateur.

Ce champ permet de stocker des informations dans un basic block et de permettre de les récupérer quand on parcourt les basic blocks.

Dans notre cas, voici la structure que nous stockons dans chaque champ auxiliaire :

```
typedef struct mpi_ranks
{
        int code;
        int * ranks;
} mpi_ranks;
```

Nous parcourrons tous les basic blocks (même 0 et 1) initialiser ce struct et le placer dans le champ auxiliaire.

`int code` est le rang de la collective `MPI` contenue dans le basic block, ou la valeur LAST_AND_UNUSED_MPI_COLLECTIVE_CODE si aucune collective n'est présente dans le block.

`int *ranks` est un tableau qui contiendra les rangs par collective au basic block courant lorsqu'ils seront calculés par la suite.

Cette implémentation nous permet, lorsque nous travaillons sur un block de savoir facilement s'il contient une collective et laquelle, puis lorsque nous allons créer les ensembles de blocks par collective et par rang le tableau ranks nous permettra d'initialiser et de parcourir ces ensembles.

2.2 Génération d'une image du CFG

Avant tout, pour modifier le CFG, il serait intéressant de pouvoir l'afficher pour s'aider et débugger. Pour cela, on se donne 3 fonctions :

- `cfgviz_dump(function * fun, const char * suffix, bitmap valid_edges=NULL)`
  Une fonction qui prend la fonction à analyser en argument, un suffixe que l'on rajoute à la fin du nom du fichier, et une liste de edges invalides. Par défaut, aucun edge n'est invalide. Cette fonction appelle les deux prochaines.

- `cfgviz_generate_filename(function * fun, const char * suffix)`
  Pour générer le nom du fichier de sorti en fonction du nom d'une fonction

- `cfgviz_internal_dump(function * fun, FILE* out, bitmap valid_edges)`
  Fonction principale de génération des graphes, on parcourt tous les basic blocks, et tous ses gimple statement. On va ensuite vérifier si le gimple est une collective `MPI`, et on rajoute dans le fichier de sortie le nom de la fonction si oui. On va ensuite parcourir tous les edges et on vérifie si le edge est invalide ou non : s'il l'est, on l'affiche en rouge, en bleu sinon.

Pour générer ces graphiques, on va générer des fichiers `.dot`, donc on tient bien évidemment compte de la syntaxe dans le code de `cfgviz_internal_dump`.

On peut voir sur la figure 2 le graphe initial.

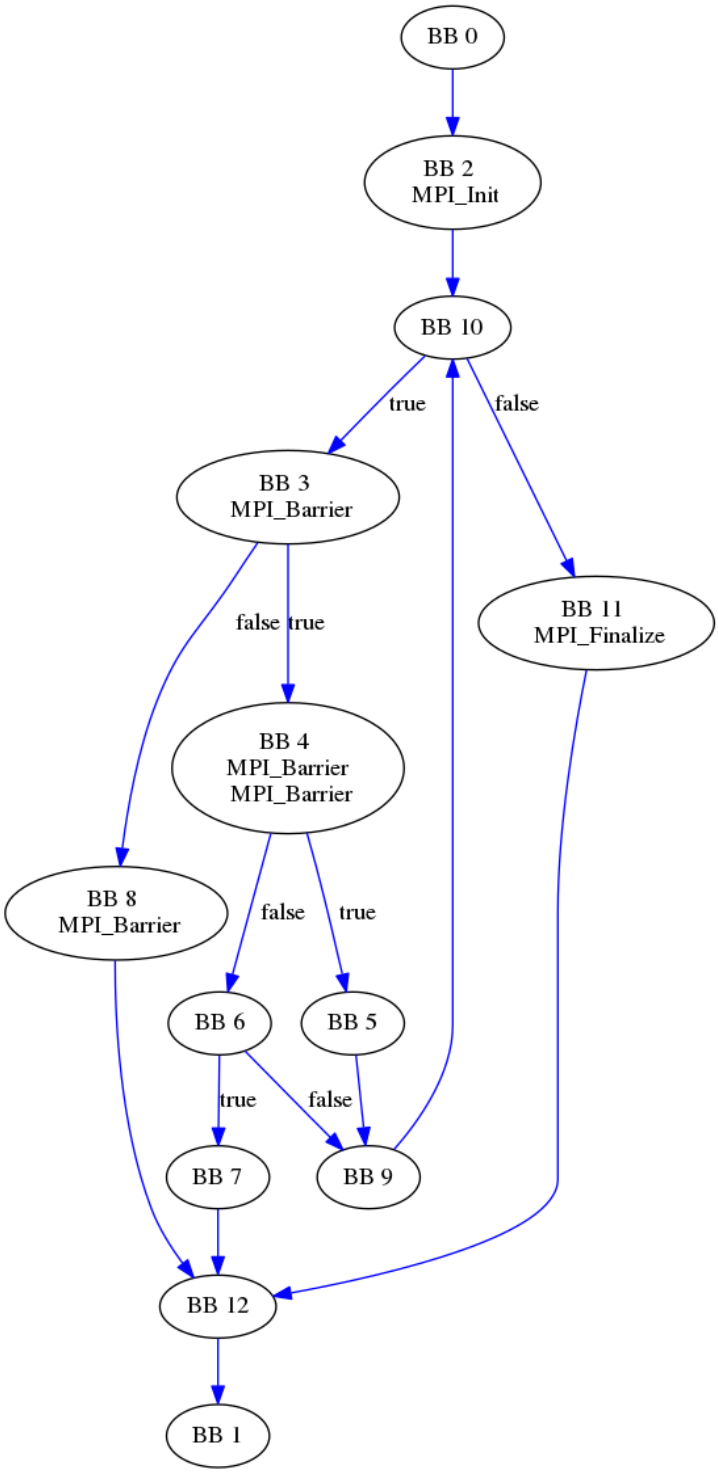


Figure 2: Premier affichage du CFG

2.3 Rang des collectives

2.3.1 Edges invalides

On va modifier le CFG pour le calcul des rangs des collectives, car s'il y a un cycle dans le graphe, le rang d'une collective sera incrémenté à l'infini lorsqu'on calcule le rang, et donc qu'on itère sur les edges du CFG.

Pour cela, on va générer un `bitmap_head* invalid_edges` (un tableau de bitmaps) qui va avoir les propriétés suivantes :

- Chaque élément du tableau (= un bitmap) correspond à un basic block
- Pour chaque bitmap, un de ses index est set si il correspond à un indice d'arc invalide.

Par exemple, si le edge 2 du basic block 4 est invalide, alors le 2e élément de invalid_edges[4] est set.

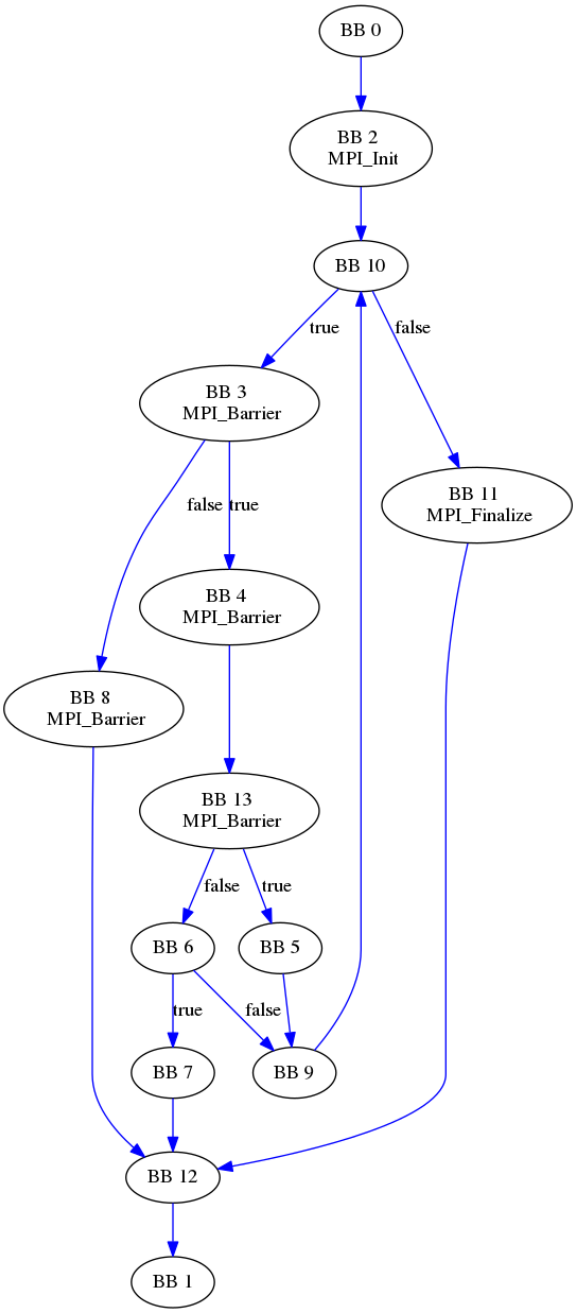
Pour ce faire, on crée un tableau de bitmap pour connaître dans chaque basic block les nœuds traversés pour arriver jusqu'ici, et on parcours en profondeur le CFG depuis le nœud 0 en transmettant à chaque fois au nœud suivant les nœuds visités précédemment.

Pour chaque basic block, on regarde ses edges et les child liés aux edges. Si un des child a déjà été visité pour arriver ici, on ajoute l'edge aux invalides, sinon on rajoute le child dans la liste à visiter. Avec le code suivant :

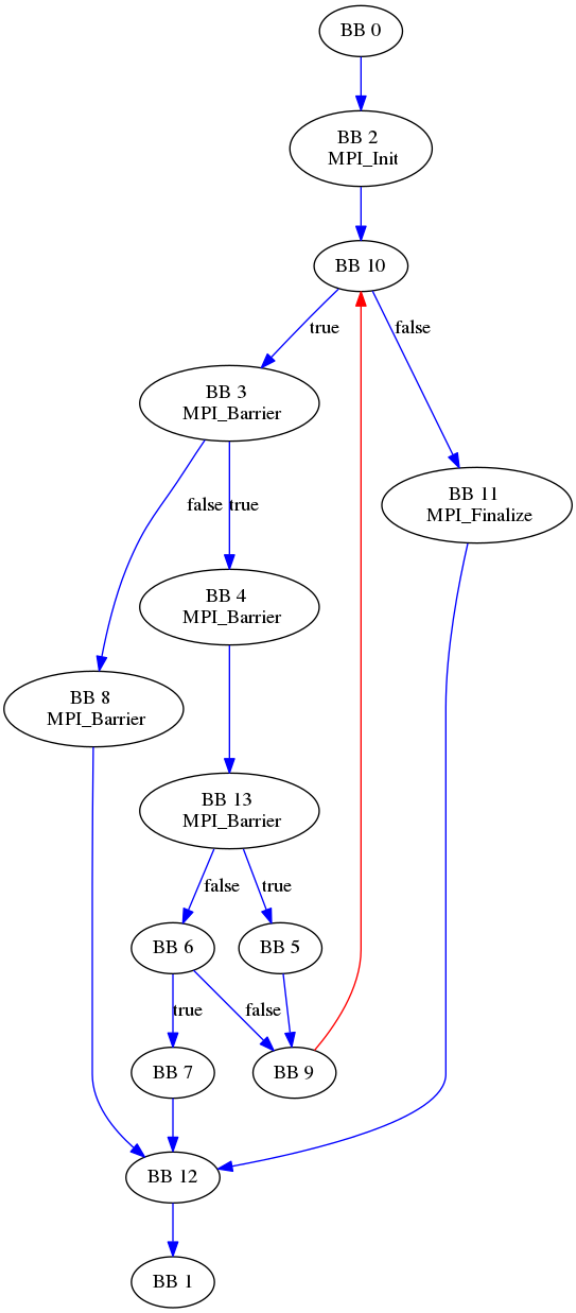
```
std::vector <basic_block> to_visit;
to_visit.push_back(ENTRY_BLOCK_PTR_FOR_FN(fun));
while (to_visit.size() != 0) {
    bb = to_visit.back();
    to_visit.pop_back();
    int index = bb -> index;
    bitmap_set_bit(&visited[index], index);

    edge_iterator it;
    edge e;

    int edge_index = 0;

    FOR_EACH_EDGE(e, it, bb -> succs) {
        basic_block child = e -> dest;
        if (bitmap_bit_p(&visited[index], child -> index)) {
            bitmap_set_bit(&invalid_edges[index], edge_index);
        }
        else {
            to_visit.push_back(child);
            /*transmit the visited blocks to the next */
            bitmap_ior_into(&visited[child -> index], &visited[index]);
        }
        edge_index++;
    }
}
```

Code de calcul du nouveau CFG qui tient compte des edges invalides

Le tableau de bitmap contenant les edges invalides est ensuite retourné pour être utilisé dans le calcul des rangs.
La figure 3b montre le CFG au quel on a retiré les edges non valides, dans notre cas celui reliant le basic block 9 au basic block 10, car il génère un cycle.



(a) CFG avec les basic blocks séparés par collectives `MPI` uniques



(b) CFG où l'on retire les edges non valides (colorés en rouges)

2.3.2 Calcul des rangs

Pour calculer le rang des collectives dans le CFG, on procède de la manière suivante :

On part du premier basic block, et on le met dans un `std::vector`, on itère sur les éléments du vecteur tant qu'il n'est pas vide. On récupère dans la boucle while le premier

basic block du vecteur, et on itère sur ses edges. On a ensuite l'algorithme suivant pour chaque edge :

```
basic_block child = e -> dest;
mpi_ranks *child_aux_ranks = (mpi_ranks *) child -> aux;
int* father_ranks = father_aux_ranks -> ranks;
int* child_ranks = child_aux_ranks -> ranks;
if (!bitmap_bit_p(&invalid_edges[index], edge_index)) {
    for (int i=0; i < LAST_AND_UNUSED_MPI_COLLECTIVE_CODE; i++) {
        if (father_ranks[i] >= child_ranks[i]) {
            child_ranks[i] = father_ranks[i];
            if (i == child_aux_ranks -> code) child_ranks[i] += 1;
        }
    }
    to_visit.push_back(child);
}
else {
    //since we do not transmit the ranks through looping edges we check
    //if the ranks in this block are superior to the ranks in the last
    //this way we ensure that the last block contains the max rank for
    //each collective, this will be usefule to create the sets
    for (int i=0; i < LAST_AND_UNUSED_MPI_COLLECTIVE_CODE; i++) {
        if (father_ranks[i] > last_ranks[i]) last_ranks[i] = father_ranks[i];
    }
}
edge_index++;
```

Code du calcul des ranks pour chaque basic block et collective

On regarde donc si le edge que l'on étudie est invalide ou non selon le calcul de la partie précédente, et on récupère le maximum de rang entre le père et le fils pour chaque collective, si le père est supérieur on le transmet au fils, si le fils contient cette collective on augmente le rang de 1. Si le edge est invalide, on effectue cette comparaison avec le dernier nœud. Cela permet de s'assurer d'avoir le rang maximum des collectives dans le graphe dans le dernier nœud.

Cela revient à utiliser l'équation suivante :

$$\text{Rang}(n) = \max_{x \in \text{preds}(n)} \text{Rang}(x) \tag{1}$$

2.3.3 Création des ensembles

Maintenant que nous avons le rang de chaque collective dans chaque nœud, nous pouvons créer des ensembles contenant tous les nœuds faisant appel à la même collective au même rang. Pour créer ces ensembles, nous allons créer bitmap_head **sets. Ce tableau est initialisé de manière à ce que sets[i][j] donne un bitmap qui servira à contenir les nœuds faisant appel à la collective i au rang j. Ce bitmap sets[i][j] sera set à un indice k si le basic block d'indice k fait partie de l'ensemble. C'est pour cela que nous avons besoin du rang max de chaque collective (stocké dans le champ aux du dernier nœud) pour allouer les tableaux sets[i], car le nombre de bitmaps à créer dépend du rang max de la collective. Ces ensembles sont créés dans le code suivant :

```
basic_block last = EXIT_BLOCK_PTR_FOR_FN(fun);
mpi_ranks *aux_ranks = (mpi_ranks *) last -> aux;
/* ranks in the last block containing the max ranks */
int *ranks = aux_ranks -> ranks;

bitmap_head **sets = XNEWVEC(bitmap_head*, LAST_AND_UNUSED_MPI_COLLECTIVE_CODE);
/* coordinates i, j are the collective code and the rank */
for (int i=0; i < LAST_AND_UNUSED_MPI_COLLECTIVE_CODE; i++) {
    /* getting the max rank for this collective to create enough bitmaps */
    int max_rank = ranks[i];
    if (max_rank != 0) {
        sets[i] = XNEWVEC(bitmap_head, max_rank);
        for(int j=0; j < max_rank; j++) {
            bitmap_initialize(&sets[i][j], &bitmap_default_obstack);
            }
    }
}

basic_block bb;
FOR_EACH_BB_FN(bb, fun) {
    mpi_ranks *aux_ranks = (mpi_ranks *) bb -> aux;

    int code = aux_ranks -> code;
    int *ranks = aux_ranks -> ranks;

    int index = bb -> index;
    /* if the block contains a collective we set its index
    in right bitmap for the code and rank */
    if (code < LAST_AND_UNUSED_MPI_COLLECTIVE_CODE) {
        bitmap set = &sets[code][ranks[code]-1];
        bitmap_set_bit(set, index);
    }
}
return sets;
```

Code de création des ensembles de nœud de collective et de même rang

Dans la suite, toutes les propriétés se rapportant à cet ensemble de nœuds seront stockées dans un tableau de la même forme permettant de récupérer, pour une collective i de rang j, un bitmap représentant les nœuds concernés par la propriété.

2.4 post-dominance

Nous avons besoin, pour la suite, de connaître l'ensemble de post-dominance d'un ensemble de basic blocks.

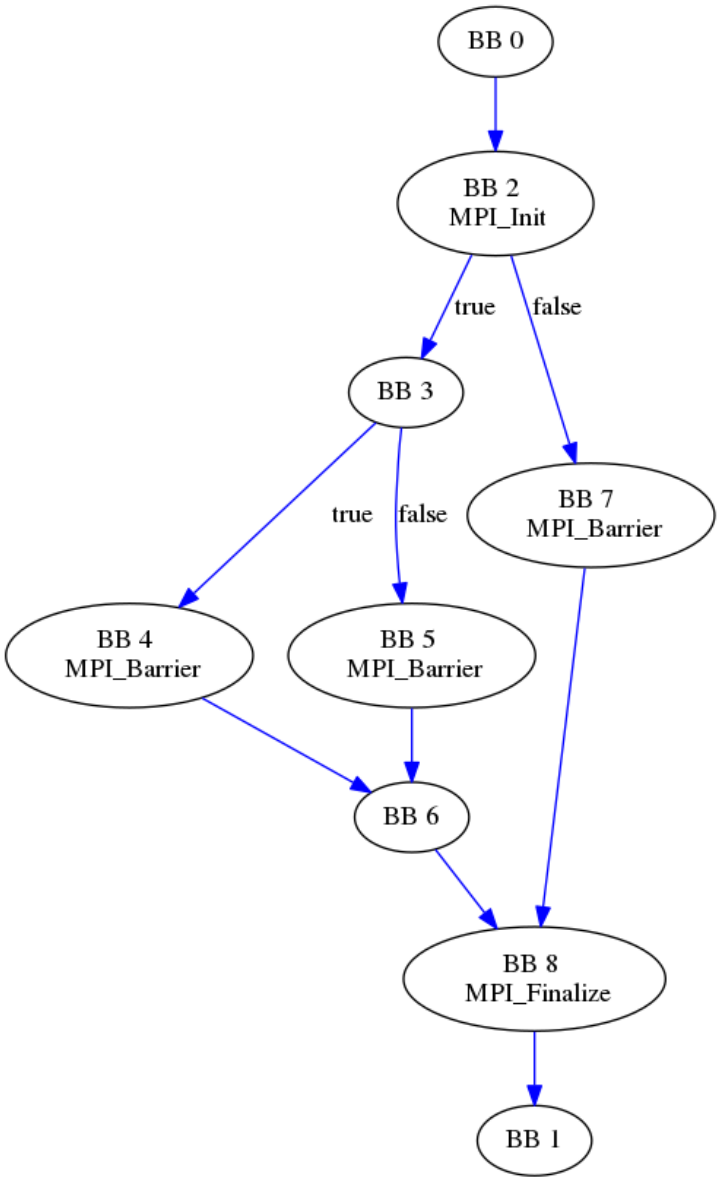
Pour récupérer l'ensemble de post-dominance, nous nous basons sur cette définition :

- Intuitivement, un ensemble de nœuds S post-dominent un nœud X si tous les chemins allant de X au puits passent par au moins un nœud appartenant à S.

Et nous n'utilisons **pas** cette définition :

$$E \gg_p X \iff \forall S \in \text{succ}(X), \exists Y \in E, \text{ tel que } Y \gg_p S \tag{2}$$

En effet, celle-ci est pour nous incorrecte. Prenons par exemple ce graphe :



Si on s'intéresse à l'ensemble {4,5,7}, il est assez intuitif de dire que cet ensemble post-domine 2. En effet, pour aller de 2 au puits, il faut nécessairement passer par soit 4, soit 5, soit 7. Si on prend l'équation 2:

Si on pose $X = 2$, soit $succs(X) = 3$, $\nexists Y \in E = \{4, 5, 7\}$, tel que $Y \gg_p 3$, car ni 4, ni 5, ni 7 ne post-dominent 3.

On a donc un problème.

On se donne l'algorithme suivant pour trouver la frontière de post-dominance d'ensembles.

```
bitmap_head** set_post_dominance_frontiers(function * fun, bitmap_head **sets
                                           bitmap_head **post_dominated,
                                           bitmap_head *frontiers)
{
  basic_block last = EXIT_BLOCK_PTR_FOR_FN(fun);
  mpi_ranks *aux_ranks = (mpi_ranks *) last -> aux;
  int *ranks = aux_ranks -> ranks;
  bitmap_head **set_frontiers = XNEWVEC(bitmap_head*,
                                        LAST_AND_UNUSED_MPI_COLLECTIVE_CODE);
  bitmap_head valid_frontiers;
  bitmap_initialize(&valid_frontiers, &bitmap_default_obstack);
  for (int i=0; i < LAST_AND_UNUSED_MPI_COLLECTIVE_CODE; i++) {
    int max_rank = ranks[i];
    if (max_rank != 0) {
      set_frontiers[i] = XNEWVEC(bitmap_head, max_rank);
      for(int j=0; j < max_rank; j++) {
        bitmap_initialize(&set_frontiers[i][j], &bitmap_default_obstack);
        for (int k=0; k < last_basic_block_for_fn(fun); k++) {
          if (bitmap_bit_p(&post_dominated[i][j], k)
              || bitmap_bit_p(&sets[i][j], k)) {
            bitmap_and_compl(&valid_frontiers, &frontiers[k],
                             &post_dominated[i][j]);
            bitmap_ior_into(&set_frontiers[i][j], &valid_frontiers);
          }
        }
      }
    }
  }
  return post_dominated;
}
```

Code de calcul de l'ensemble de post-dominance des ensembles

et algorithme parcourt le graphe depuis la fin. On traverse un nœud parent s'il ne fait pas partie de l'ensemble et s'il n'est pas marqué comme non post dominé. Chaque nœud traversé est marqué comme non post dominé, car si on atteint un nœud, c'est qu'il existe un chemin jusqu'à la fin du graphe ne passant pas par un élément de l'ensemble. Ainsi, en prenant l'inverse des nœuds marqués, on obtient les éléments post dominés par un ensemble. On applique cet algorithme pour tous les ensembles et on retourne post_dominated contenant tous les bitmaps représentant les éléments post dominés par chaque ensemble.

2.5 Frontières de post-dominance et frontières itérées

2.5.1 Frontière de post-dominance de chaque nœud

Dans la suite, nous aurons besoin de la frontière de post-dominance de certains nœuds. Nous allons donc utiliser un algorithme pour calculer les frontières et les stocker dans un

tableau de bitmaps. Pour cela, nous utilisons cet algorithme[1], modifié pour calculer la frontière de post-dominance plutôt que la frontière de dominance :

```
for all nodes, b
    if the number of predecessors of b ≥ 2
        for all predecessors, p, of b
            runner ← p
            while runner ≠ doms[b]
                add b to runner's dominance frontier set
                runner = doms[runner]
```

Figure 4: Algorithme du calcul de la frontière de post-dominance d'un nœud

2.5.2 Frontière de post-dominance d'un ensemble

Pour le calcul de la frontière de post-dominance d'un ensemble, nous allons tout d'abord commencer par prouver que :

$$\text{Soit E un ensemble de nœuds}$$
$$\text{Soit PDF(E)} = \{\ \text{X} \mid \exists S \in \text{Succs}(X) \text{ tq } E \gg_{sp} S,\ E \not\gg_p X\}$$
$$\text{Soit F} = \{\ \text{X} \mid \exists n \in E \text{ tq } E \gg_{sp} n,\ X \in \text{PDF}(n),\ E \not\gg_p X\}$$
$$\text{On va montrer que PDF(E)} = \text{F}$$

$$\text{Soit } \text{x}_1 \in F$$
$$\Leftrightarrow \exists n \in E,\ x_1 \in \text{PDF}(n),\ E \gg_p n,\ E \not\gg_p x_1$$
$$\Leftrightarrow \exists n \in E,\ \exists S \in \text{Succ}(x_1),\ n \gg_{sp} S,\ n \not\gg_p x,\ E \gg_p n,\ E \not\gg_p x_1$$
$$\Rightarrow \exists S \in \text{Succ}(x_1),\ E \gg_p n \gg_p S,\ E \not\gg_p x_1$$
$$\Rightarrow x_1 \in \text{PDF}(E) \Rightarrow F \subseteq \text{PDF}(E)$$

$$\text{Soit } \text{x}_2 \in \text{PDF}(E)$$
$$\Leftrightarrow \exists S \in \text{Succ}(x_2),\ E \gg_{sp} S,\ E \not\gg_p x_2$$
$$\Rightarrow S \not\gg_p x_2, \text{ car } E \gg_{sp} S \text{ et } E \not\gg_p x_2$$
$$\text{donc } \text{x} \in \text{PDF}(S), \text{ car } S \in \text{Succ}(x_2), S \gg_{sp} S, \text{ et } S \not\gg_p x_2$$
$$\text{donc } \text{x}_2 \in F \Longrightarrow \text{PDF}(E) \subseteq F$$

[1] https://www.cs.tufts.edu/comp/150FP/archive/keith-cooper/dom14.pdf

Ainsi, on en tire l'algorithme suivant :

```
bitmap_head**
set_post_dominance_frontiers(function * fun, bitmap_head **post_dominated, bitmap_hea
{
    basic_block last = EXIT_BLOCK_PTR_FOR_FN(fun);
    mpi_ranks *aux_ranks = (mpi_ranks *) last -> aux;
    int *ranks = aux_ranks -> ranks;
    bitmap_head **set_frontiers = XNEWVEC(bitmap_head*,
                                          LAST_AND_UNUSED_MPI_COLLECTIVE_CODE);
    bitmap_head valid_frontiers;
    bitmap_initialize(&valid_frontiers, &bitmap_default_obstack);
    for (int i=0; i < LAST_AND_UNUSED_MPI_COLLECTIVE_CODE; i++) {
        int max_rank = ranks[i];
        if (max_rank != 0) {
            set_frontiers[i] = XNEWVEC(bitmap_head, max_rank);
            for(int j=0; j < max_rank; j++) {
                bitmap_initialize(&set_frontiers[i][j], &bitmap_default_obstack);
                for (int k=0; k < last_basic_block_for_fn(fun); k++) {
                    if (bitmap_bit_p(&post_dominated[i][j], k)) {
                        bitmap_and_compl(&valid_frontiers, &frontiers[k],
                                         &post_dominated[i][j]);
                        bitmap_ior_into(&set_frontiers[i][j], &valid_frontiers);
                    }
                }
            }
        }
    }
    return set_frontiers;
}
```

L'algorithme parcours les éléments post dominés pour chaque ensemble et ajoute leur frontière à la frontière de l'ensemble si celle-ci n'est pas post dominée par l'ensemble.

2.5.3 Frontière itérée

Une fois la frontière de post-dominance récupérée, on calcule la frontière de post-dominance itérée. Pour cela, on ajoute dans chaque frontière les frontières des éléments qu'elle contient. Puis celle des éléments que l'on vient d'ajouter, et ainsi de suite jusqu'à ce qu'aucun nouvel élément ne soit ajouté.

Concrètement, si la frontière de post-dominance était par exemple l'ensemble 4,5, on calcule la frontière de post-dominance de 4, supposons qu'elle renvoie 6, puis la frontière de post-dominance de 5, supposons qu'elle renvoie 6, 2. Puis on recommence avec 2, 4, 5, 6, et ainsi de suite.

On a le code suivant qui est exécuté pour chaque frontière de post-dominance (une par collective/rang) :

```
bitmap_initialize(&set_iterated_frontiers[i][j], &bitmap_default_obstack);
bitmap_ior_into(&set_iterated_frontiers[i][j], &set_frontiers[i][j]);
/* we add the frontiers of the blocks in the frontier and we keep adding */
/* the frontiers of those frontiers until no new blocks are added */
unsigned long old_count = bitmap_count_bits(&set_iterated_frontiers[i][j]);
unsigned long new_count = 0;
while (old_count != new_count) {
    old_count = bitmap_count_bits(&set_iterated_frontiers[i][j]);
    for (int k=0; k < last_basic_block_for_fn(fun); k++) {
        if (bitmap_bit_p(&set_iterated_frontiers[i][j], k)) {
            bitmap_ior_into(&set_iterated_frontiers[i][j], &frontiers[k]);
        }
    }
    new_count = bitmap_count_bits(&set_iterated_frontiers[i][j]);
}
```

Code du calcul de la frontière de post-dominance itérée.

2.6 Affichage d'un warning

Finalement, une fois la frontière de post-dominance itérée récupérée, on peut afficher un warning à l'utilisateur. Si la frontière itérée d'un ensemble n'est pas vide, il faut avertir l'utilisateur qu'il y a un potentiel problème.

L'algorithme est le suivant :

On fait une boucle sur le nombre de collectives, puis une boucle sur le nombre d'états de rangs différents (donné par max_rank). On regarde si la frontière de post-dominance itérée pour l'ensemble de collective i et de rang j est vide. Si non, on cherche pour chaque basic block de l'ensemble le statement contenant l'appel à la collective `MPI` et on émet un warning pour que l'utilisateur voit le groupe de collectives de même rang concerné par le problème.

```
warning_at(gimple_location(stmt), 0, "Potential issue: MPI
collective %s in block %d", mpi_collective_name[i], k);
```

Ensuite, on parcourt l'ensemble des basic blocks contenus dans la frontière et on récupère leur dernier statement, afin d'émettre un warning pour notifier à l'utilisateur les lignes qui provoquent le potentiel deadlock pour les collectives notifiées précédemment.

```
warning_at(gimple_location(stmt), 0, "Potential
issue caused by the following fork in block %d", k);
```

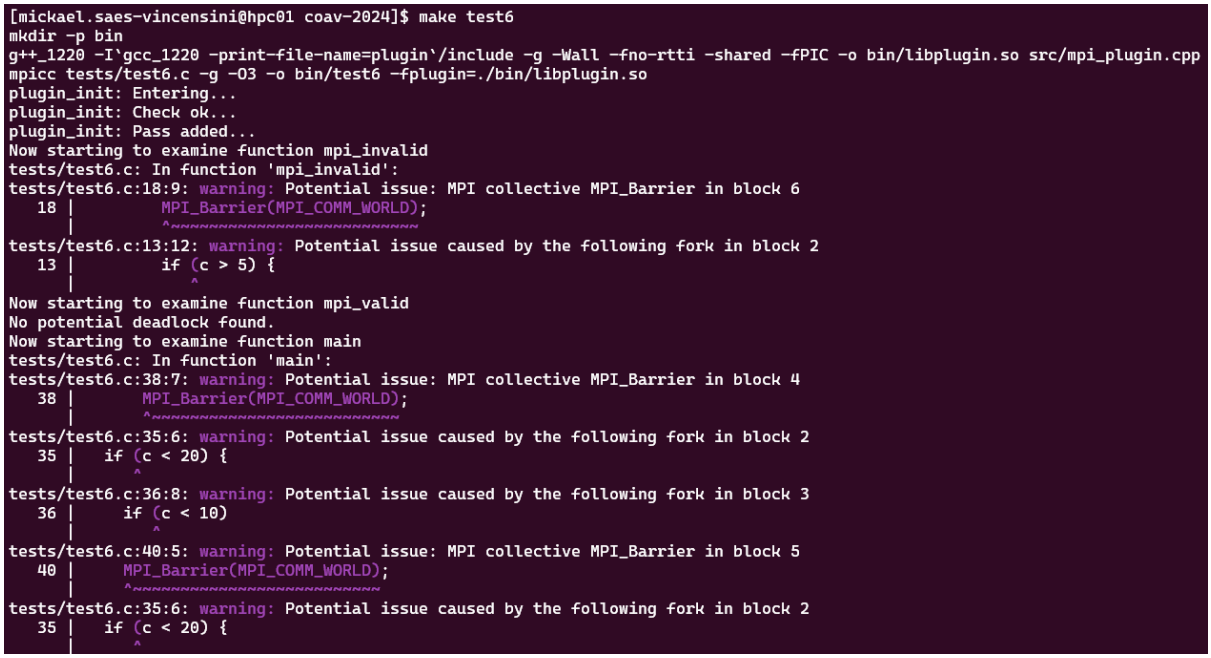


Figure 5: Affichage généré par le warning

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#pragma Projet_CA mpicoll_check (main, mpi_invalid, mpi_valid)
void mpi_not_checked() {
        MPI_Barrier(MPI_COMM_WORLD);
}
void mpi_invalid(int c) {
    if (c > 5) {
        MPI_Barrier(MPI_COMM_WORLD);
        return;
    } else MPI_Barrier(MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    return;
}
void mpi_valid(int c) {
    if (c > 5) {
        MPI_Barrier(MPI_COMM_WORLD);
    } else MPI_Barrier(MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    return;
}
int main(int argc, char * argv[])
{
    MPI_Init(&argc, &argv);
    int c = 5;
    if (c < 20) {
        if (c < 10)
    {
        MPI_Barrier(MPI_COMM_WORLD);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    }
    MPI_Finalize();
    return 1;
}
```

Code utilisé pour générer les warning de la figure 5

3 Passe de compilation et Directives #pragma

3.1 Passe de compilation

La passe est insérée après la passe cfg, et applique le execute suivant:

```
unsigned int execute (function *fun)
{
    cfgviz_dump(fun, "initial");
    prepare_cfg(fun);
    cfgviz_dump(fun, "split");
    calculate_dominance_info(CDI_POST_DOMINATORS);
    bitmap_head *frontiers = post_dominance_frontiers(fun);
    bitmap_head *invalid_edges = cfg_prime(fun);
    cfgviz_dump(fun, "invalid_edges", invalid_edges);
    calculate_rank(fun, invalid_edges);
    bitmap_head **sets = collective_rank_set(fun);
    bitmap_head **set_postdominated = set_post_dominance(fun, sets);
    bitmap_head **set_frontiers = set_post_dominance_frontiers(fun, sets,
                                                    set_postdominated, frontiers);
    bitmap_head **it_frontier = iterated_post_dominance_frontiers(fun,
                                                    set_frontiers, frontiers);
    bool warnings = print_warnings(fun, it_frontier, sets);
    if (!warnings) printf("No potential deadlock found.\n");
    clean_aux_field(fun, 0);
    free_dominance_info(CDI_POST_DOMINATORS);
    return 0;
}
```

Code exécuté par la passe du plugin

Rien de compliqué ici, on applique juste les fonctions les unes après les autres et on affiche un message à la place des warnings si aucun deadlock potentiel n'a été détecté.

3.2 Directives #pragma

Le but de ces directives est de choisir les fonctions sur lesquelles la passe est appliquée. Elles peuvent prendre les formes suivantes:

```
#pragma ProjetCA mpicoll_check f
#pragma ProjetCA mpicoll_check (f1, f2, ..., fn)
```

Le compilateur émet une erreur si ces formes ne sont pas respectées, et un warning si une fonction est référencée plusieurs fois ou si une fonction inexistante est référencée. On utilise la fonction suivante pour parser les pragmas et vérifier leur forme:

```
static void handle_pragma_fx(cpp_reader *dummy ATTRIBUTE_UNUSED) {
    enum cpp_ttype token;
    bool close_paren_needed = false;
    tree pragma_arg;
```

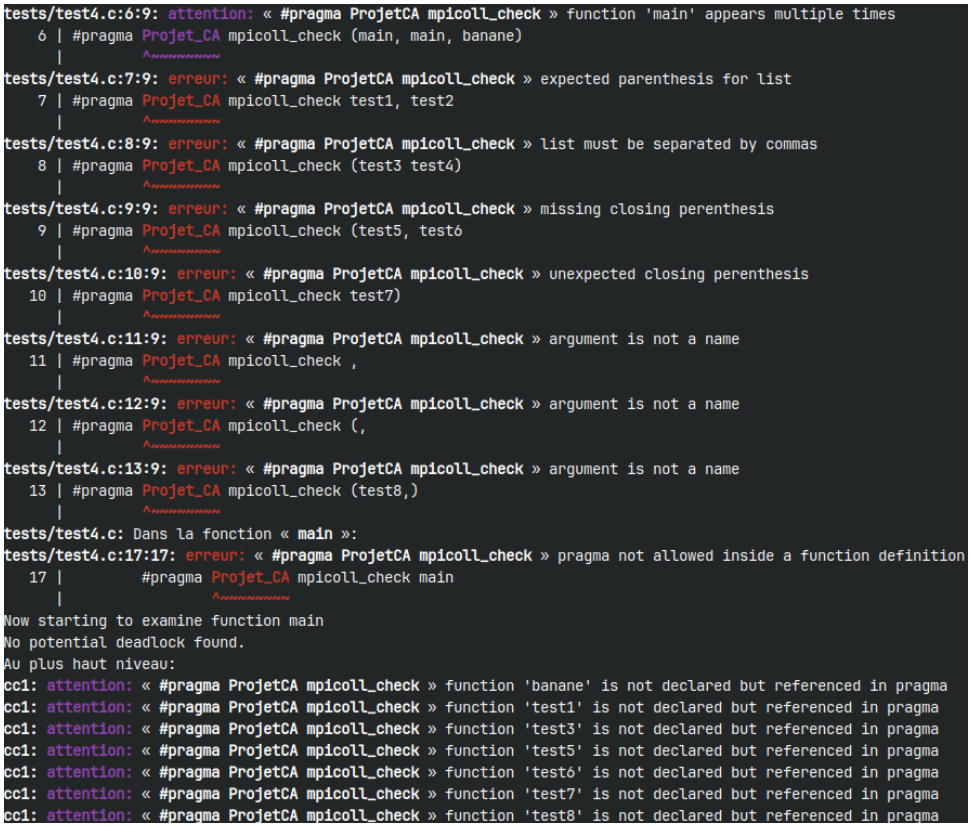
```
if (cfun) {
    error("%<#pragma ProjetCA mpicoll_check%> \
    pragma not allowed inside a function definition");
    return;
}

token = pragma_lex(&pragma_arg);
if (CPP_OPEN_PAREN == token) {
    close_paren_needed = true;
    token = pragma_lex(&pragma_arg);
    if (CPP_NAME != token) {
        error("%<#pragma ProjetCA mpicoll_check%> \
        argument is not a name");
        return;
    }

    handle_arg(pragma_arg);

    token = pragma_lex(&pragma_arg);
    if (token == CPP_NAME) {
        error("%<#pragma ProjetCA mpicoll_check%> \
        list must be separated by commas");
        return;
    }
    while (CPP_COMMA == token) {
        token = pragma_lex(&pragma_arg);
        if (CPP_NAME != token) {
            error("%<#pragma ProjetCA mpicoll_check%> \
            argument is not a name");
            return;
        }
        handle_arg(pragma_arg);
        token = pragma_lex(&pragma_arg);
    }
} else if (CPP_NAME == token) {
    handle_arg(pragma_arg);
    token = pragma_lex(&pragma_arg);
}else {
     error("%<#pragma ProjetCA mpicoll_check%> \
     argument is not a name");
    return;
}
if (CPP_CLOSE_PAREN == token) {
    if (!close_paren_needed) {
        error("%<#pragma ProjetCA mpicoll_check%> \
        unexpected closing perenthesis");
        return;
    }
```

```
        close_paren_needed = false;
        token = pragma_lex(&pragma_arg);
    }
    if (CPP_EOF == token) {
        if (close_paren_needed) {
        error("%<#pragma ProjetCA mpicoll_check%> \
        missing closing perenthesis");
        }
    }
    else {
        error("%<#pragma ProjetCA mpicoll_check%> \
        expected parenthesis for list");
        return;
    }

}
```

Cette fonction place les arguments valides des pragmas dans un vecteur. Lorsque la passe est appelée sur une fonction la gate recherche et retire le nom de la fonction du vecteur et retourne true pour exécuter la passe. Si la fonction n'est pas dans le vecteur, la gate retourne false et la passe n'est pas exécutée.

```
    bool gate (function *fun)
        {
            const char* fname = fndecl_name(cfun->decl);
            if (remove_function_from_pragma_list(fname)) {
                printf("Now starting to examine function %s\n", fname);
                return true;
            }
            return false;
        }
```

On ajoute à la fin du plugin un callback (avec PLUGIN_FINISH) permettant de vérifier si le vecteur est vide. Si le vecteur n'est pas vide, il contient alors tous les arguments de pragmas qui ne correspondent pas à une fonction déclarée. On parcourt donc ce vecteur pour émettre un warning pour chaque élément du vecteur. Voici un example des warnings que les pragmas peuvent déclancher:



4 Conclusion

Ce projet a donc permis de réaliser un plugin pour `GCC` 12.2.0, permettant la détection statique de deadlock dans des codes C contenant des collectives `MPI`. Nous avons également intégré des directives pragma afin de choisir quelles fonctions analyser dans le code.

En l'état, notre projet est fonctionnel et correspond aux parties 1 et 2 du sujet. Pour aller plus loin, la suite du sujet est un axe de réflexion sur l'amélioration du plugin. Notamment, nous ne regardons les collectives `MPI` seulement à l'intérieur d'une fonction, c'est-à-dire que nous n'analysons notamment pas les échanges entre processus. Cet axe permettra aussi d'améliorer notre plugin.

Ce projet a tout de même permis de prendre en main une première fois ce qui touche à `GCC` et l'implémentation de plugins pour celui-ci.