Projet de Compilation Avancée

Gavoille Clément - Skutnik Jean-Baptiste

Introduction

Le framework `MPI` met en place des fonctions permettant aux programmes de communiquer lors de l'exécution pour faciliter la parallélisation de calculs. Cependant, le fait d'introduire des communications dans un code peut entraîner des ralentissements, voire des arrêts indéfinis lors d'une erreur ou d'un mauvais agencement des fonctions de communication par l'utilisateur.

Ce projet propose de créer un plugin `GCC` qui, lors de la compilation, va vérifier que les fonctions collectives `MPI` sont traversées par tous les cas possibles d'exécution du programme, assurant ainsi que le programme ne sera jamais dans une impasse. En effet, tant que tous les processus MPI n'appelle pas la même fonction collective, ils se bloquent.

Pour se faire, l'utilisation de l'`API GCC` est indispensable car elle permet d'interagir avec les étapes de compilation. L'ensemble du projet consiste à l'implémentation d'une "passe" de compilation.

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

La passe de compilation travaille sur une représentation du code interne à `GCC` qui prend la forme d'un graphe orienté nommé `Control Flow Graph` ou `CFG` dans la suite de ce rapport. Il représente le code et les différentes suites d'instructions pouvant être exécutées lors de l'exécution d'un programme.

Les liaisons entre les différents noeuds correspondent aux tests logiques effectués lors de l'exécution du programme. Les noeuds qui le composent sont eux appelés des `Basic Blocks`.

La lecture de ce graphe est **indépendante du langage du code source** dont il résulte. Les portions de code sont transformées en déclarations de forme normée `GENERIC`, mais l'analyse est faite sur des simplifications de cette norme: le format `GIMPLE`.

Détection des appels aux fonctions collectives MPI

La détection des appels de fonction à l'intérieur du code se fait en parcourant la liste des instructions contenues dans les blocs de codes du `CFG`.
Ces blocs, sous format `GIMPLE` sont parcourus à la recherche d'appels de fonctions. On compare ensuite celles-ci à une liste pré-déterminée des fonctions d'intérêt sur lesquelles on base notre analyse.

Le fichier `MPI_collectives.def` définit les appels `MPI` supportés. Il contient ici les déclarations suivantes:

```
DEFMPICOLLECTIVES( MPI_INIT, "MPI_Init" )
DEFMPICOLLECTIVES( MPI_FINALIZE, "MPI_Finalize" )
DEFMPICOLLECTIVES( MPI_REDUCE, "MPI_Reduce" )
DEFMPICOLLECTIVES( MPI_ALL_REDUCE, "MPI_AllReduce" )
DEFMPICOLLECTIVES( MPI_BARRIER, "MPI_Barrier" )
```

Ce format nous permet de créer dynamiquement un `enum` à l'aide de `macros C`:

```
#define DEFMPICOLLECTIVES( CODE, NAME ) if(!strcmp(func_name, NAME)){return i;}else{i++;};
int is_mpi(const char* func_name) {
    int i = 0;
#include "MPI_collectives.def"
    return -1;
};
#undef DEFMPICOLLECTIVES
```

Rajouter une collective cible à notre analyse revient donc à ajouter une simple ligne dans le fichier `MPI_collectives.def`.

Préparation du contexte

Les `basic blocks`, composant les noeuds du `CFG`, peuvent contenir de multiples appels aux fonctions cibles. Pour faciliter l'analyse du code, la première étape de notre passe modifie le graphe en scindant les `basic blocks` afin qu'ils contiennent **au plus** un appel de collective `MPI`.

La mise en place de cet axiome facilite grandement l'analyse du graphe, et ne changera en rien la sémantique du programme. En effet, les basic blocks ainsi créés seront ensuite recollés par une autre passe d'optimisation. Par la suite, il sera supposé que chaque appel à une collective `MPI` est unique dans le `basic block` où elle se trouve.

La séparation des blocs se fait grâce à la fonction `isolate_mpi()` définie dans `mpi_detection.cpp`. Elle appelle la méthode `split_block()` de l'`API GCC` pour scinder les noeuds, en donnant l'assertion `GIMPLE` correspondante.

Étude du graphe de flot de contrôle pour déterminer les divergences

Les noeuds contenant des collectives `MPI` à analyser sont regroupés en ensembles, représentés en mémoire par des `bitmaps` définies dans l'`API GCC`.

Dans le code, c'est la fonction `mpi_calls()` qui effectue ce traitement. Elle renvoie un tableau de `bitmaps` de taille égale au nombre de collectives `MPI` définies dans le programme. Ainsi, dans la `bitmap` d'index `k`, les bits d'index `i` valent `1` si le `basic block` d'index `i` contient la collective d'index `k` correspondante.

Ces ensembles sont alors utilisés dans le calcul de leur frontières de post-dominance. On calcule la frontière de post-dominance d'un noeud de la manière suivante:

$$On\ note\ n\ post\ domine\ z\ par\ n \triangleright z$$

$$PDF(n) = \{z \mid \forall (m,z) \in \Omega,\ m \to z \mid n \triangleright m,\ n \not\triangleright z\}$$

À l'aide des frontières de post-dominance des noeuds, la frontière de post-dominance de l'ensemble est calculée suivant la formule:

$$PDF(N) = \{z \mid z \in \cup_{n \in N} PDF(n),\ z \in \cup_{k \in \bar{N}} PDF(k)\}$$

La frontière de post-dominance d'un ensemble nous permet de déterminer les noeuds à partir desquels il existe un chemin ne passant pas par un noeud de l'ensemble. La fonction `compute_pdf_sets()` renvoie un objet similaire à `mpi_calls` contenant cette information. Ainsi, si la frontière de post-dominance d'un ensemble est non-vide, on ne va pas appeler les fonctions collectives de cet ensemble et il y a donc possiblité de créer un deadlock.

Détermination des noeuds à risque

Une fois l'ensemble des noeuds à tester déterminé, il s'agit de vérifier que les divergences potentielles existent. Pour ce faire, un parcours en profondeur du graphe depuis les noeuds sélectionnés va être effectué.

L'algorithme effectuant le parcours est le suivant, en pseudo-code:

```
std::vector<unsigned int> suite = {}
typedef stack_el std::pair<basic_block, unsigned int>
std::vector<stack_el> pile = {}

suite = parcours_simple(bb)

tant que pile.size() > 0:
    current_bb, index = pile.pop()
    si collective dans current_bb:
        si collective != suite[index]:
            retourner false
        sinon
            index = index + 1

    pour tout successeur de current_bb:
        pile.push((successor, index))

retourner true
```

Tout d'abord, pour simplifier le code, une suite est déterminée avec un parcours simple du graphe: seul le premier successeur de chaque noeud est pris en compte, et tous les appels aux collectives sont relevés et stockés, dans l'ordre, dans le tableau 'suite'.

Cette suite n'a pas pour vocation d'être exacte, mais sert de référence pour le reste de l'algorithme: si toutes les suites d'appels sont identiques, alors cette suite est égale à tous les autres suites calculées et chaque chemin va donc appeler le sfonctions cibles dans le même ordre.

L'algorithme va alors effectuer un parcours en profondeur du graphe, mais en stockant dans la pile, en plus de l'index du basic block, l'index de la collective recherchée dans la suite d'appel.
À chaque appel `MPI` rencontré, le programme vérifie qu'il correspond à celui de la suite de référence à l'index indiqué.

Si il y a correspondance, le parcours continue, en incrémentant de 1 l'index. Sinon, le programme s'arrête en renvoyant `false`, car cela implique que cette suite d'appel ne suit pas la référence.

Cet algorithme, et les parcours en profondeur en général, peuvent rester bloqués lors de l'analyse de boucles. Or, dans des cas réels d'utilisation, des boucles sont garanties.

Nous avons donc pris la décision de ne pas analyser les boucles.

Le framework `GCC` délimitant et catégorisant les boucles à été utilisé pour déterminer l'appartenance d'un `basic_block` à une boucle, mais le système de prédictions de `GCC` n'a pas été utilisé pour faire des suppositions. À la place, le programme évite l'analyse si un noeud à risque est dans une boucle.

Affichage d'un warning à l'utilisateur

L'implémentation des avertissement est basée sur la structure fournie par `GCC` :

```
warning_at (location_t location, int opt, const char *gmsgid, ...)
```

Pour s'intégrer au mieux aux messages du compilateur et profiter de l'archtecture existante autour de ces derniers. Le code traduit le `basic block` et le code de la collective générant l'avertissement en accédant à la dernière assertion du `basic block`, en traduisant le nom de la collective à partir de son code, et en fournissant les deux à `gcc` :

```
warning_at(gimple_location(stmt),   // Assertion posant problème
           0,                       // Options, aucune ici
           "Calls to %s may be avoided from this location",
           mpi_collective_name[collective]);
```

Cette gymnastique produit des avertissements conventionnels pour `GCC` :

```
test2.c: In function ‘main’:
test2.c:15:4: warning: Calls to MPI_Barrier may be avoided from this location
   15 |  if(c<10)
      |    ^
```

Partie 2: Gestion des directives

Définition et format

L'utilisation des directives dans le programme permet à l'utilisateur de contrôler facilement le comportement du plugin, en utilisant un format de messages standardisé dans `GCC` .

Dans ce plugin, les directives définies servent à spécifier les fonctions à analyser. Elles prennent les formes suivantes:

```
#pragma mpicoll check f1
#pragma mpicoll check (f2, f3)
```

Les directives introduites dans le plugin suivent les règles suivantes:

- Elles ne peuvent apparaître dans un corps de fonction ;
- Plusieurs directives peuvent apparaître, mais chaque fonction doit être spécifiée une seule fois ;
- Une fonction peut être spécifiée uniquement si elle est présente dans le code source.

Enfreindre une de ces règles enverra un avertissement pour l'utilisateur.

Gestion des directives

Les directives sont interprêtées par deux fonctions définies par le plugin :

- `handle_pragma_function` : lit les directives et enregistre les fonctions dans une zone mémoire interne. Les directives erronées ou redondantes génèrent un avertissement à l'utilisateur.
- `wrap_mpicoll` : génère un avertissement pour toutes les fonctions dans la zone mémoire à la fin de l'exécution de `GCC` , indiquant que leur définition n'a pas été trouvée dans le code.

Lors de l'exécution de la passe, la fonction `gate` du plugin prend en compte les fonctions enregistrées pendant `handle_pragma_function` pour ne lancer l'exécution que sur les fonctions spécifiées.

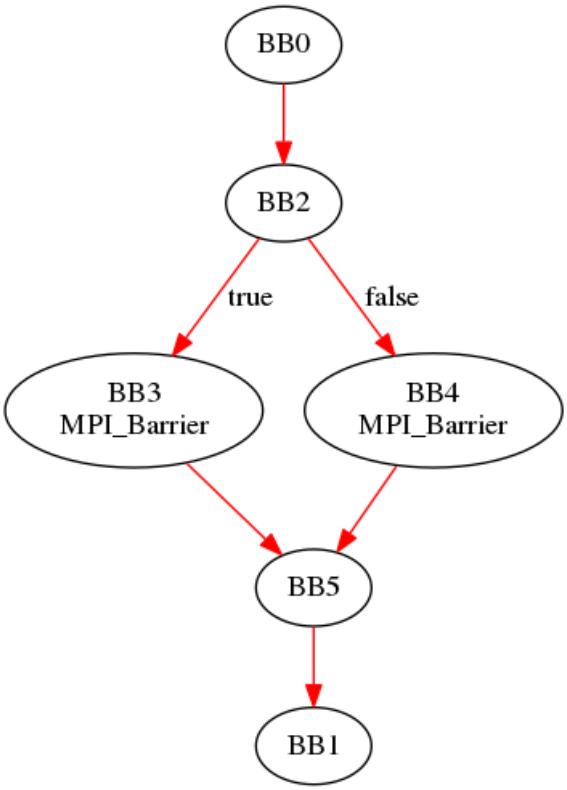
Tests

Test 1 :

Code source:

```
int easy1(int a) {
        if (a = 0){
                MPI_Barrier(MPI_COMM_WORLD);
        }
        else
                MPI_Barrier(MPI_COMM_WORLD);
        return 0;
}
```

CFG:



Résultat:

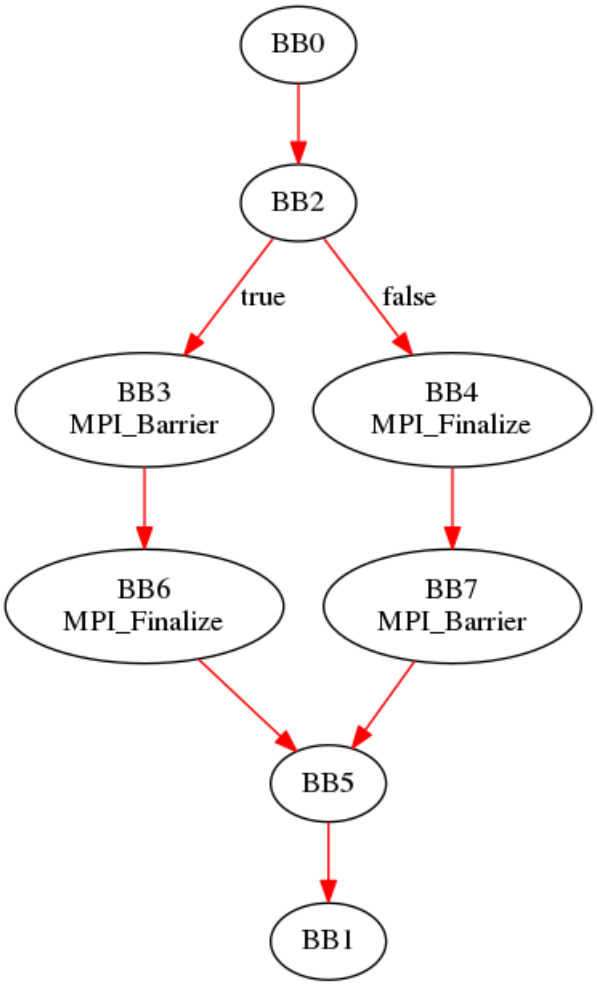
Aucun problème, la PDF de l'ensemble {3,4} est nulle, il n'y a donc aucun noeuds à risque.

Test 2 :

Code source:

```
int easy2(int a) {
        if (a = 0){
                MPI_Barrier(MPI_COMM_WORLD);
        MPI_Finalize();
        }
        else {
        MPI_Finalize();
                MPI_Barrier(MPI_COMM_WORLD);
    }
        return 0;
}
```

CFG:



Résultat:

```
test2.c: In function ‘easy2’:
test2.c:18:5: warning: Calls to MPI_Finalize may be avoided from this location
   18 |  if (a = 0){
      |     ^
test2.c:18:5: warning: Calls to MPI_Barrier may be avoided from this location`
   18 |  if (a = 0){
      |     ^
```

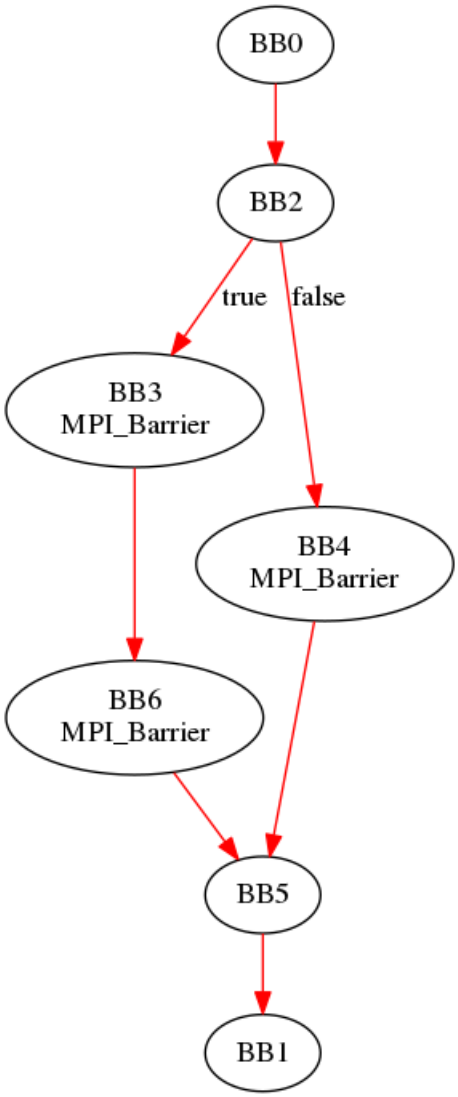
Ici, la PDF des deux ensembles contient le noeud 2. Lors de l'analyse des chemins à partir du noeud 2, les séquences ne sont pas les mêmes. La passe renvoie donc un warning vers le basic block 2.

Test 3

Code Source:

```
int easy3(int a) {
        if (a = 0){
                MPI_Barrier(MPI_COMM_WORLD);
                MPI_Barrier(MPI_COMM_WORLD);
        }
        else
                MPI_Barrier(MPI_COMM_WORLD);
        return 0;
}
```

CFG:



Résultat:

La passe ne renvoie rien alors qu'il y a un problème. En effet, elle travaille sur l'ensemble de noeuds {3,4,6} et la PDF de cet ensemble est vide. Elle n'effectue donc pas l'analyse des chemins et ne détecte donc pas la différence de séquence d'appels entre les deux chemins. Pour qu'elle traite ce problème, il faudrait séparer l'ensemble de travail en {3,4} et {6}.

Conclusion

Le projet dans son ensemble répond à la problématique donnée. La passe de compilation analyse l'algorithme, et renvoie un avertissement à l'utilisateur lorsqu'une divergence est détectée.

La gestion des directives est elle assez claire et flexible pour permettre le contrôle efficace de son comportement.

Cependant, ne pas prendre en compte les boucles s'est révélé être un pari risqué, le programme pouvant ne pas détecter un problème, voire tomber dans une impasse lorsque plusieurs boucles sont adjacentes: le parcours de graphe n'arrive tout simplement pas à suivre.

En outre, l'implémentation d'une analyse inter-procédurale permettrait à la passe de détecter plus de cas posant des problèmes et même de corriger des cas ne posant finalement pas de problème. Cela lui permettrait d'effectuer donc une analyse enore plus juste du programme.

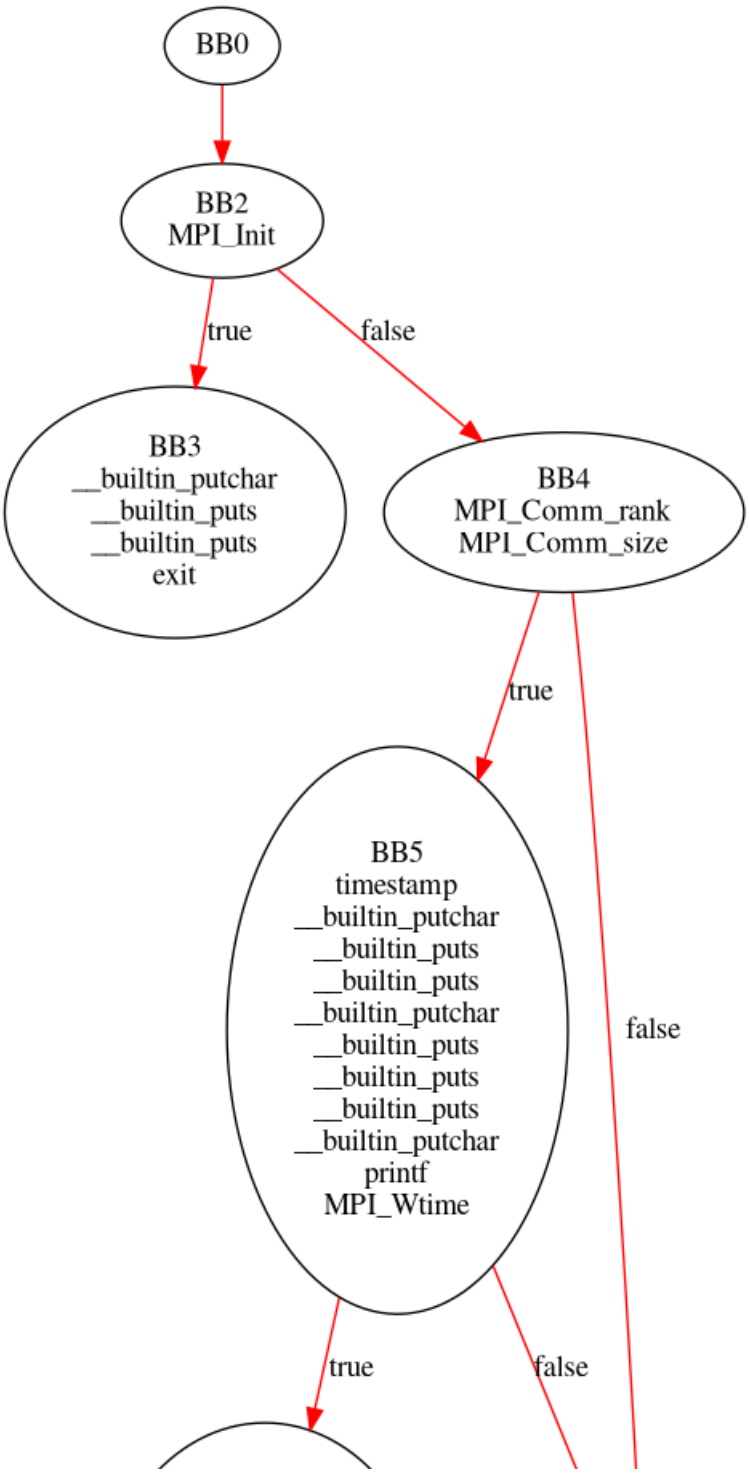
Sources

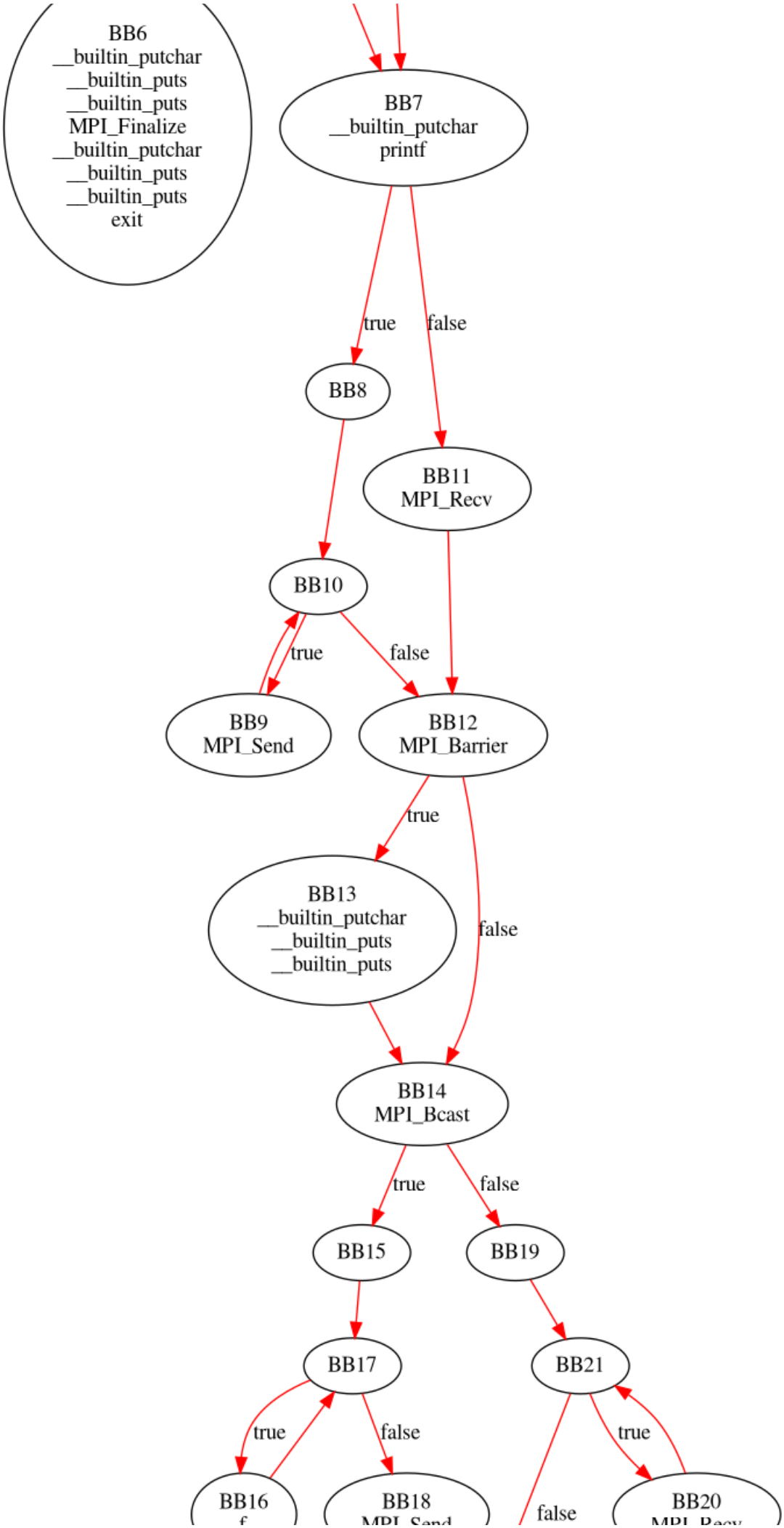
Nom	Source
`Control Flow Graph`	GCC Documentation
`Basic Blocks`	GCC Documentation
`GENERIC`	GCC Documentation
`GIMPLE`	GCC Documentation

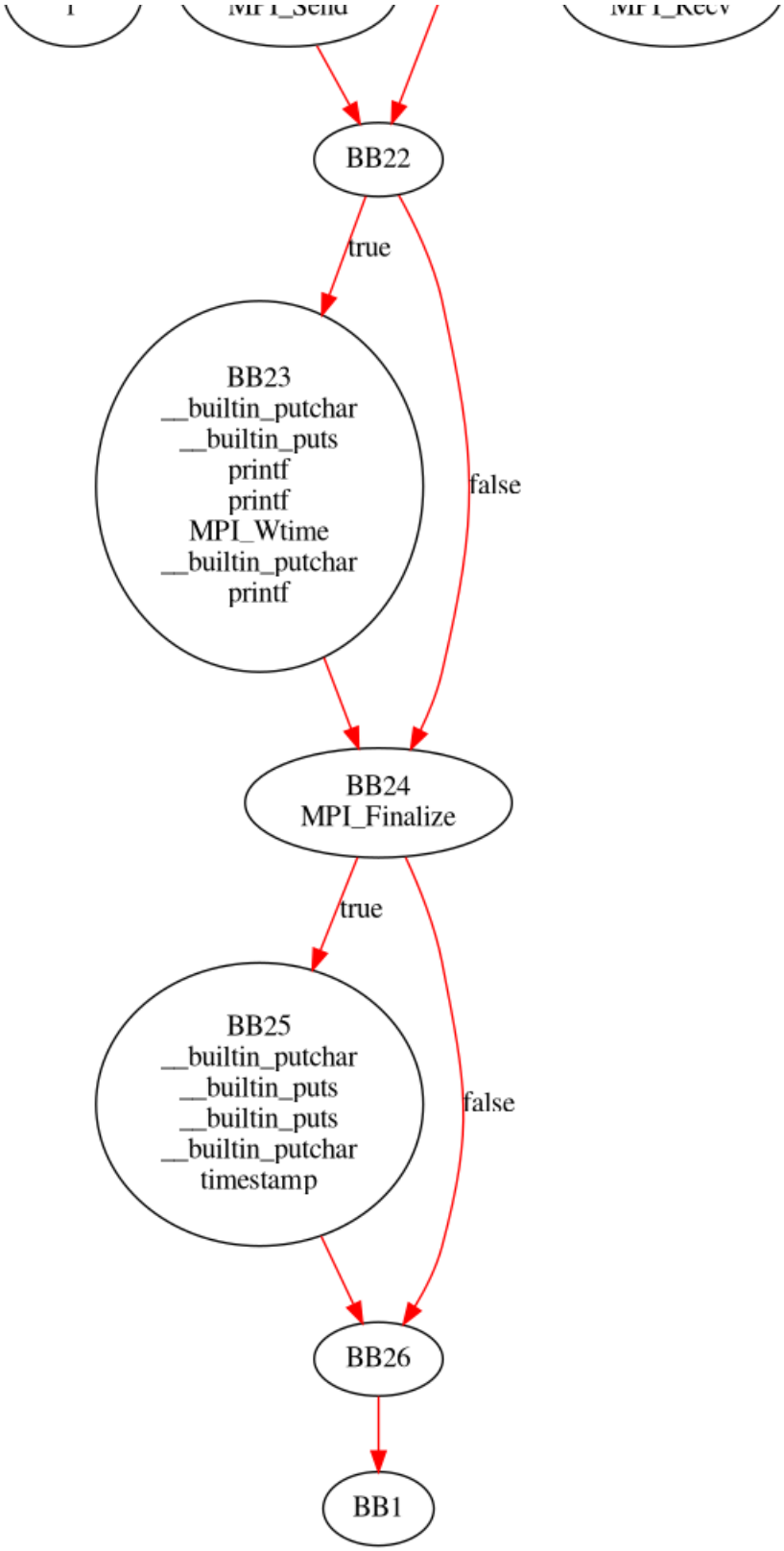
Annexe 1: Code source réel

Le fichier intervals contient un code source `MPI` existant sur lequel notre passe détecte les divergences de séquence de colectives.

CFG:




BB6
__builtin_putchar
__builtin_puts
__builtin_puts
MPI_Finalize
__builtin_putchar
__builtin_puts
__builtin_puts
exit
BB7
__builtin_putchar
printf
true
false
BB8
BB11
MPI_Recv
BB10
true
false
BB9
MPI_Send
BB12
MPI_Barrier
true
BB13
__builtin_putchar
__builtin_puts
__builtin_puts
false
BB14
MPI_Bcast
true
false
BB15
BB19
BB17
BB21
true
false
true
BB16
BB18
false
BB20


MPI_Send
MPI_Recv
BB22
true
false
BB23
__builtin_putchar
__builtin_puts
printf
printf
MPI_Wtime
__builtin_putchar
printf
BB24
MPI_Finalize
true
false
BB25
__builtin_putchar
__builtin_puts
__builtin_puts
__builtin_putchar
timestamp
BB26
BB1

Résultat:

```
intervals.c: In function ‘main’:
intervals.c:113:6: warning: Calls to MPI_Finalize may be avoided from this location
  113 |   if ( ierr != 0 )
      |      ^
intervals.c:132:6: warning: Calls to MPI_Finalize may be avoided from this location
  132 |   if ( process_id == master )
      |      ^
intervals.c:147:8: warning: Calls to MPI_Finalize may be avoided from this location
  147 |     if ( process_num <= 1 )
      |        ^
intervals.c:113:6: warning: Calls to MPI_Barrier may be avoided from this location
  113 |   if ( ierr != 0 )
      |      ^
intervals.c:132:6: warning: Calls to MPI_Barrier may be avoided from this location
  132 |   if ( process_id == master )
      |      ^
intervals.c:147:8: warning: Calls to MPI_Barrier may be avoided from this location
  147 |     if ( process_num <= 1 )
      |        ^
```

On le retrouve ici.