Projet de Compilation Avancée

Antoine Dujardin, Thibaut Milhaud, Haowen Wu

L'objectif de ce projet était de concevoir un plugin GCC détectant, à la compilation les éventuels problèmes de synchronisation (au niveau interne des fonctions) pouvant exister dans un code MPI. Et dans un second temps d'ajouter à ce plugin une gestion des pragma qui permet à l'utilisateur de sélectionner les fonction qu'il souhaite analyser.

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

Découpage des basic blocks

Dans un premier temps, il fallait découper les basic blocks de façons à ce que chacun d'en eux ne puisse contenir au maximum qu'une seule fonction MPI.

Solution algorithmique

Nous avons décidé de faire ce découpage en deux étapes :

1. Premièrement, compter le nombre de directives mpi dans chaque basic block et de stocker cette valeur;
2. Ensuite, pour chaque basic block, si il contient plusieurs fonctions mpi on le coupe juste après la première fonction MPI que l'on croise. Puis on continue sur le block ainsi obtenu. On itère ainsi tant que le block courant contient plus d'une fonction MPI.

Dans le code, cela se traduit de la manière suivante :

- `Pour chaque basic block bb :`
  - `Count = 0`
  - `Pour chaque statement stmt :`
    - `If (is_mpi(stmt))`
      - `Count ++`
  - `Bb -> aux -> count = count`
- `Pour chaque basic block bb :`
  - `Count = bb->aux->count`
  - `Stmt = premier statement de bb`
  - `Tant que count > 1 :`
    - `If (is_mpi(stmt)) :`
      - `Split(bb)`
      - `Count --`
      - `Bb = bb->next_bb`

        - `Bb->aux->count = count`
        - `Stmt = premier statement de bb`
    - `Else :`
        - `Stmt = next(stmt)`

On remarque qu'après exécution du pseudo code ci-dessus le champ `bb->aux->count` n'est pas à jour. Ce n'est pas grave car on n'en a plus besoin par la suite.

Cela correspond aux fonctions `mpi_count_bb` et `mpi_split_bb` de `plugin.cpp`.

Calcul du rang des directives

Afin de pouvoir partitionner les blocks par rang et directives communs, il faut commencer par calculer le range de chaque directives.

Solution algorithmique

L'algorithme que nous avons utilisé ici se rapproche fortement d'un parcours, en largeur. Cela dit, parcours en largeur oblige, il a fallu coder une file. Nous allons donc présenter les différents choix d'implantation que nous avons fait.

Implantation de file

Dans notre code, une file est codée de façon suivante :

- Un **tableau** `T` de basic blocks qui contient les élément de la file;
- Une **bitmap** `in_queue` qui indique quels sont les blocks à l'intérieur de la file;
- Deux entiers `top` et `bot` qui indiquent les indices du tableau entre lesquels il y a des éléments ([bot, top[ ).

Dans cette implantation de file on impose **l'unicité** des éléments à l'intérieur.

Pour ajouter un élément `bb` en queue de file (**push**) :

- `If(!bit_p(in_queue, bb->index))`
    - `T[top] = bb`
    - `top ++`
    - `set_bit(in_queue, bb->index)`

Vérifier si la file est vide (**is_empty**) :

- `bot == top`

Pour récupérer l'élément en tête de file dans `cur` (**pop**) :

- `cur = T[bot]`
- `bot ++`

- `unset_bit(in_queue, cur->index)`

Les avantages de cette implantation sont principalement la simplicité et l'efficacité : pas de structures compliquées et les opération de push/pop et is_empty se font en temps constant.

Cela entraîne néanmoins des inconvénients : il faut savoir combien de fois (au maximum) on va push dès la création de la file pour pouvoir créer un tableau de taille suffisante. Une file implantée de cette façon n'est pas destinée à durer dans le temps.

Dans le cas du parcours en largeur cette structure est convenable même si un peu coûteuse en mémoire. En effet, pour la taille de T, il suffit de prendre le nombre de basic blocks (noté n dans la suite). On consomme donc un tableau de pointeurs de longueur n, une bitmap capable de stocker n éléments (grossièrement (n/8)+1 bits) ainsi que 2 entiers.

Parcours en largeur et calcul des rangs

Maintenant que nous disposons d'une structure de file, nous pouvons implanter le parcours en largeur et calculer le rang des directives.

- edge_map (bitmap pour les arêtes)
- seen_map (bitmap pour les block déjà vus)
- n = nombre de blocks
- q = new_queue(n)
- push(q, ENTRY_BLOCK(fn))
- Tant que (!is_empty(q)):
    - cur = pop(q)
    - set_bit(seen_map, cur->index)
    - Pour chaque arête p sortant de cur :
        - next = p->dest
        - if(!bit_p(seen_map, next->index)) :
            - set_bit(edge_map, p->dest_idx)
        - push(q, next)
    - // CALCUL DU RANG
    - rank = 0
    - Pour chaque arête p entrant dans cur :
        - if(bit_p(edge_map, p->dest_idx)
            - prev = p->src
            - if(prev->aux->rank > rank)
                - rank = prev->aux->rank
    - if(contains_mpi(cur))
        - cur->aux->rank = rank + 1
    - Else
        - cur->aux->rank = rank

Remarque : On a condensé le calcul du sous graphe (défini par le sous ensemble d'arêtes dans edge_map) et le calcul du rang en un seul parcours en largeur eu lieu de deux au prix d'une boucle sur les arêtes entrantes le sommet courant.

En analysant grossièrement la complexité temporelle de cet algo, on obtient O(n+2m) avec n le nombre de basic blocks et m le nombre d'arêtes.

Frontières de post-dominance

Nous avons vu en cours les notion de (post-)domination et de frontière de (post-)dominance. Cependant comme il s'agit d'une partie centrale du travail demandé, nous avons choisi de rappeler ici la signification de ces notions dans le cas qui nous intéresse : post-domination.

Post-domination : On dit que X post-domine Y si tous les chemins de Y au puits passent par X.

Frontière de post-dominance : La frontière de post dominance d'un noeud Y est l'ensemble des noeuds que non post dominés par Y mais qui ont un successeur qui l'est. C'est-à-dire les sommets X tels que :

- Y post ne post domine pas X;
- Il existe x différent de Y successeur de X tel que Y post domine x.

On note PDF(Y) la frontière de post dominance de Y.

La frontière de post dominance d'un ensemble : La frontière de post dominance d'un ensemble U l'ensembles des X tels qu'il existe x successeur de X tel que pour tout u dans U,

- u post domine x;
- u ne post domine pas X.

Dans notre cas, il est intéressant de calculer la frontière de post dominance des ensembles C(r, d) qui contiennent toutes les directives d de rang r. En effet si cette dernière n'est pas vide, cela signifie qu'il peut y avoir un problème.

Remarque : Il fallait également travailler avec la frontière de post dominance itérée qui permet de donner le point du code qui pose problème. Malheureusement, dans la version actuelle du plugin, la fonction chargée de calculer cette frontière itérée renvoie toujours l'ensemble vide, ce qui est navrant.

Format des warning

On rappelle que l'objectif est de prévenir l'utilisateur des éventuels deadlocks de fonctions MPI qui peuvent exister dans son code. N'ayant malheureusement pas de frontière de post dominance itérée fonctionnelle les warnings que nous proposons à l'utilisateurs sont de la forme :

WARNING(projetCA): in function **{nom de la fonction}**, MPI conflict with **{Nom de la directive MPI}**. Potential forks are at lines [**{lignes des blocks de la frontière de post dominance}**].

Exemple de sortie de du plugin sur un test :

```
FUNCTION : mpi_call
[GRAPHVIZ] Generating CFG of function mpi_call in file <dot/mpi_call_test4.c_10_mpi.dot>
WARNING(projetCA): in function mpi_call, MPI conflict with MPI_Barrier. Potential forks are at lines [ 11 ].
WARNING(projetCA): in function mpi_call, MPI conflict with MPI_Barrier. Potential forks are at lines [ 11 ].
```

Algorithme récapitulatif

Pour chaque fonction, le plugin va effectuer les opérations suivantes :

- Setup
    - Split des block
    - Marquage des blocks (remplir bb->aux)
        - Code des directives
        - Rank
    - Calcul des informations de dominance (fonctions GCC)
- Noyau dur
    - Calcul de la PDF de chaque block
    - Pour r < rang max :
        - Pour c < code_max :
            - Set = {}
            - Pour chaque block bb :
                - If (bb->code == c && bb->rank == r)
                    - Ajout de bb dans set
            - pdf_set = PDF(set)
            - If (pdf_set non vide):
                - WARNING
- Cleanup
    - On vide bb->aux pour chaque block
    - On vide les informations de dominance

Voilà le squelette de l'algorithme effectué par le plugin.

Sources du projet

Vous trouverez les sources du projet à sur github à l'adresse : https://github.com/titiyeahti/CA.

Partie 2

Format et erreurs

Les formats des directives pour activer une directive MPI sont:

- `#pragma ProjetCA mpicoll_check f`, pour activer une fonction.
- `#pragma ProjetCA mpicoll_check (f1, f2)`, pour activer deux fonctions.

Les formats suivants ne sont pas valides, et provoquent l'affichage d'erreur:

- `#pragma ProjetCA mpicoll_check f)`
- `#pragma ProjetCA mpicoll_check f1, f2`
- `#pragma ProjetCA mpicoll_check (f1, f2`
- `#pragma ProjetCA mpicoll_check (f1, f2), f3`

Le message d'erreur inclut aussi le format correct.

Une erreur est aussi affichée si l'utilisateur essaye d'activer une directive MPI qui a déjà été activée, ou si la directive est dans une fonction.

Par exemple, la directive :

```
#pragma ProjetCA mpicoll_check (f1, f2
```

Donne ces erreurs :

```
Error: `#pragma ProjetCA mpicoll_check (string[, string]...)` is missing a closing parenthesis
```

Stockage des directives à analyser

La liste des fonctions à analyser est stockée dans un vecteur GCC, `MPICOLLECTIVES`. Ce vecteur est une variable globale qui peut être utilisée par les passes de compilation suivantes.

Les vecteurs stockent les éléments dans un tableau contigu. Les vecteurs GCC implémentent toutes les fonctions pour réserver l'espace mémoire nécessaire pour stocker le tableau, réserver de l'espace supplémentaire quand le tableau est plein, ou itérer sur les éléments du tableau.