ENSIIE HPC - Projet de Compilation Avancée 2019/2020

Vérification statique des séquences d'appels aux fonctions collectives MPI

Ce document présente le projet du module CA - Compilation Avancée.

La norme MPI impose que les appels aux fonctions collectives soient réalisés dans le même ordre pour tous les processus MPI. Lorsque la séquence d'appel est différente entre plusieurs processus concernés par ces appels MPI, cela crée un deadlock

Le but de ce projet est de réaliser une première vérification à la compilation des fichiers pour vérifier si le positionnement des appels aux fonctions collectives MPI dans le code ne risque pas de générer un deadlock. Pour cela, il est nécessaire d'examiner tous les chemins du CFG allant de la source (point d'entrée du programme) aux différents puits (points de sorties du programme). Pour chaque chemin existant entre la source et les puits, si la séquence d'appel est la même, alors tous les processus invoqueront les fonctions collectives MPI dans le même ordre. Au contraire, s'il existe des chemins avec des séquences d'appels différentes, alors il y a une possibilité pour que plusieurs processus prennent des chemins différents, n'invoquant pas les fonctions collectives dans le même ordre, et créant ainsi un deadlock. Dans ce cas, nous nous proposons d'avertir l'utilisateur avec un warning, et d'instrumenter le code pour vérifier la présence du problème et le cas échéant, arrêter proprement le programme sans laisser le deadlock apparaitre.

Afin de limiter l'analyse dans le cas de programmes complexes, nous nous limiterons à l'analyse intraprocédurale (fonction par fonction, sans considérer les interactions entre les fonctions). De plus pour réduire la quantité de travail, nous allons définir des directives (#pragma en C/C++) pour spécifier, à la compilation, les fonctions à considérer pour notre analyse.

Modalités du projet:

- Projet par binome
  - Date pour nous envoyer la liste des binômes : 17/09/2019
- Rédaction d'un rapport
  - Le rapport doit contenir une description du projet incluant les choix techniques, les algorithmes, quelques détails d'implémentation et des résultats expérimentaux
  - Date pour rendre le rapport et le code source: 22/10/2019
- Présentation et démonstration lors d'une soutenance
  - Durée de la soutenance : 5 min (10 minutes de présentation et 5 minutes de démonstration de l'utilisation de votre projet) suivies de questions
  - Date des soutenances : 22/10/2019

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

Cette partie du projet représente le coeur de notre analyse : à la compilation du code source, déterminer si chaque chemin du CFG contient la même séquence d'appels (mêmes appels dans le même ordre) aux fonctions collectives MPI, et, si ce n'est pas le cas, émettre un warning compréhensible à l'utilisateur

Cette partie de décompose en quatre sous parties :

1. Définition d'une passe permettant de parcourir le CFG et les noeuds gimple, et vérifier si les noeuds d'appel de fonction font référence à une fonction collective MPI (la liste des fonctions MPI à reconnaitre vous sera fourni sous la forme d'un fichier def)
2. Utilisation de la notion de frontière de post dominance itérée (iterated post dominance frontier ou PDF") pour retrouver, en cas de conflit, le noeud contenant le fork à l'origine de ce conflit.
3. Regroupement de toutes les données collectées pour fournir à l'utilisateur un warning compréhensible, contenant suffisamment d'information pour lui permettre la nature et l'origine du problème potentiel.

Passe de compilation pour trouver les appels aux fonctions collectives MPI

Nous allons considérer uniquement le CFG de chaque fonction courante, sans tenir compte de l'interaction possible entre les appels aux fonctions collectives MPI dans les différentes fonctions du programme test. Pour chacune de ces fonctions sélectionnées pour l'instrumentation, il va falloir identifier tous les appels à d'autres fonctions, et récupérer la cible de cet appel pour la comparer au fichier .def fournit contenant la liste des fonctions MPI à vérifier.

Définition d'une numérotation permettant de suivre la position d'une fonction collective MPI dans la séquence d'appel

La représentation que vous allez choisir pour suivre la position d'une fonction collective MPI dans la séquence d'appel d'un chemin est très importante. Elle doit vous permettre de vérifier rapidement si chaque chemin possède la même séquence d'appel, et donc que les fonctions MPI avec le même numéro doivent être les mêmes.

Utilisation de la notion de frontière de post dominance itérée pour retrouver le noeud d'origine du deadlock potentiel

Pour retrouver facilement le fork ayant conduit à la présence de deux chemins n'ayant pas la même séquence d'appel, vous devez vous servir des notions de dominance, post-dominance et frontière de post-dominance itérée dans un graphe. GCC fournit déjà une représentation permettant de retrouver les post-dominateurs d'un noeud. Servez-vous en pour construire la PDF d'un ensemble de nouds et retrouver ainsi le plus proche noeud ancêtre commun

Affichage d'un warning à l'utilisateur

Avec les informations recueillies (fonctions collectives MPI et noeuds Gimple en conflit, naud à l'origine du conflit), vous devez, en cas de problèmes, émettre un warning permettant à l'utilisateur de retrouver facilement l'origine du deadlock potentiel dans son code.

Partie 2: Gestion des directives (obligatoire)

Définition et format

Nous allons définir une directive (pragma) permettant de sélectionner les fonctions à analyser. Cette directive se compose de deux formes:

- `#pragma ProjetCA mpicoll check f` : Cette forme permet d'activer l'analyse pour la fonction f uniquement
- `#pragma ProjetCA mpicoll check (f1, f2)` : Cette seconde forme permet d'activer l'analyse pour un ensemble de fonctions : cet ensemble est donné sous la forme d'une liste d'éléments séparés par des virgules. Le nombre d'éléments n'est pas borné.

Gestion des directives

La gestion de cette directive est active des que le plugin GCC de ce projet est utilisé. Les fonctions présentes dans ces directives sont alors analysées. Si une fonction est présente dans le code source à compiler, mais non renseignée par une telle directive, l'analyse n'est pas effectuée. Un fichier source peut utiliser plusieurs fois cette directive en mélangeant les deux formes. Cette directive ne peut être utilisée qu'en dehors de toute fonction. Un

warning devra être émis à l'utilisateur s'il spécifie une fonction dans cette directive qui n'existe pas dans le code source. De plus, un warning sera émis si une fonction est spécifiée plusieurs fois dans l'ensemble des directives

Partie 3: Instrumentation dynamique du code (facultatif

Cette partie du projet permet d'affiner notre analyse et de gérer proprement les conflits: dynamiquement (a l'exécution de l'application), vérifier si tous les processus MPI passent par une même fonction à la même position, et, si un conflit apparait, avant de tomber dans le deadlock, arrêter proprement le programme en affichant un message d'erreur.

Cette partie de décompose en deux sous-parties :

1. Définition d'une passe permettant d'ajouter des sondes avant les appels aux fonctions collectives MPI annotées comme potentiellement problématique, ainsi qu'au noeud "join" de ces appels problématiques (les fonctions d'instrumentations à insérer vous seront fournies dans une bibliothèque dynamique)
2. Passer en parametres de ces fonctions les informations recueillies lors de la phase d'analyse statique pour permettre l'affichage d'un message d'erreur détaillé

Partie 4 : Analyse inter procédurale (facultatif)

Afin d'affiner notre analyse statique et de réduire le surcoût du à une instrumentation excessive, nous proposons d'améliorer notre première passe d'analyse pour considérer les interactions entre les différentes fonctions du programme cible. Chaque fonction peut être annotée avec sa séquence d'appels aux fonctions MPI. Le parcours du graphe d'appel de fonction peut alors permettre de détecter les conflits s'effaçant lorsque l'on considère la séquence d'appel globale du programme, et non plus seulement locale à une fonction.