

**ENSIIE – HPC/BigData**  
**Projet de Compilation Avancée 2023/2024**  
**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 cette 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 analyse 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 tel *deadlock*. Pour cela, il est nécessaire, pour chaque fonction, d'examiner tous les chemins du CFG allant de la source (point d'entrée de la fonction) aux différents puits (points de sorties de la fonction). 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, 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* apparaître.

Afin de limiter l'analyse dans le cas de programmes complexes, nous nous limiterons à l'analyse intra procédurale (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 binôme**
  - Date pour nous envoyer la liste des binômes : **02/10/2023**
- **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 : **06/11/2023**
- **Présentation et démonstration lors d'une soutenance**
  - Durée de la soutenance : **15 min (10 minutes de présentation et 5 minutes de démonstration de l'utilisation de votre projet)** suivies de questions
  - Date des soutenances : **06/11/2023**

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

Cette partie du projet représente le cœur 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 se décompose en quatre sous-parties :

- 1) Définition d'une passe permettant de parcourir le CFG et les nœuds *gimple*, et vérifier si les nœuds d'appel de fonction font référence à une fonction collective MPI (la liste des fonctions MPI à reconnaître vous sera fournie 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 nœud 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 d'identifier la nature et l'origine du problème potentiel.

### Passé 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 fourni 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 nœud 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'appels, 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 nœud. Servez-vous en pour construire la *PDF\** d'un ensemble de nœuds et retrouver ainsi le plus proche nœud ancêtre commun.

### Affichage d'un warning à l'utilisateur

Avec les informations recueillies (fonctions collectives MPI et nœuds Gimple en conflit, nœud 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 dès 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 (à l'exécution de l'application), vérifier si tous les processus MPI passent par une même fonction à la même position, et, si un conflit apparaît, avant de tomber dans le *deadlock*, arrêter proprement le programme en affichant un message d'erreur.

Cette partie se 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 nœud « join » de ces appels problématiques (les fonctions d'instrumentations à insérer vous seront fournies dans une bibliothèque dynamique).
- 2) Passer en paramètres 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 dû 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.

## Remarques générales

- Ce qui est pris en compte dans l'évaluation du projet :
  - La lisibilité du code
  - Ce qui a été fait
    - Rien ne marche ?
    - Une seule partie fonctionne ?
    - Tout fonctionne comme demandé ?
    - Vous avez fait toutes les parties, même facultatives ?
  - La lisibilité du rapport et sa rédaction
    - Explication des choix techniques
    - Stratégie d'émission d'avertissements et d'erreurs à l'utilisateur
    - Conception et implémentation de la bibliothèque dynamique
    - Discussion sur les performances
    - ...
  - La soutenance
    - Pertinence des transparents
    - Présentation de votre code et démo
    - Pistes pour modifier ou étendre le code et le projet
    - ...
- Si quelque chose n'est pas clair dans ce sujet, demandez. Comme tout projet, la question n'est pas de voir que vous pouvez faire cela sans aide ou en trichant mais de vous mener à bon port. Si vous êtes bloqués, il suffit de poser une question et, dans la mesure du possible, nous vous aiderons à trouver une solution.
  - Nous vous fournirons des exemples de code que vous pouvez tester au fur et à mesure du projet.