Description du travail réalisé: Parallélisation avec OpenMP AMS I-03

Jincheng KE

4 janvier 2024

1 Introduction

Ce rapport présente le travail réalisé dans le cadre du projet de parallélisation d'un code de calcul scientifique en utilisant OpenMP. L'objectif était d'optimiser les performances sur un processeur AMD R9 3900 sous Windows 11, en comparant différentes approches de parallélisation.

2 Environnement de Développement

Le projet a été développé sur un système d'exploitation Windows 11. Pour la compilation du code, le compilateur Intel icpx de l'Intel oneAPI a été choisi pour sa compatibilité et son efficacité, bien qu'il ne soit pas entièrement optimisé pour les processeurs AMD comme celui utilisé (AMD R9 3900, 12 cœurs, 24 threads).

Au début, les tests ont été effectués sur le sous-système Ubuntu de WSL2 (Windows Subsystem for Linux), mais en raison de la variabilité des performances et de l'instabilité de la virtualisation, les tests ont été déplacés directement sur le système Windows. Cela a assuré des résultats de test plus cohérents et fiables. Les outils de parallélisation tels qu'OpenMP ont été intégrés via le compilateur Intel, favorisant une exploration approfondie de différentes stratégies de parallélisation, tant fine-grain que coarse-grain.

3 Parallélisation Fine-Grain

La stratégie de parallélisation fine-grain a été appliquée dans la méthode `iteration_domaine` de la classe `Scheme`. Cette méthode utilise un triple boucle for, ce qui représente un candidat idéal pour la parallélisation. On a employé `#pragma omp parallel for` pour paralléliser ces boucles.

Les variables `i, j, k, du, du1, du2, x, y,` et `z` ont été définies comme privées pour chaque thread pour éviter les conflits d'accès en écriture. La variable `du_sum` a été traitée différemment : puisqu'elle accumule les résultats de tous les threads, une simple variable partagée aurait conduit à des conflits et des erreurs de calcul. Pour résoudre ce problème, on a utilisé `reduction(+:du_sum)`. Cette directive OpenMP permet à chaque thread de travailler sur une copie locale de `du_sum` et de combiner ensuite tous ces résultats en une valeur unique de manière thread-safe, assurant ainsi l'intégrité des données et l'efficacité du calcul parallèle.

Nous avons décidé de ne pas utiliser `collapse(3)` malgré la présence de boucles imbriquées, car nos tests ont montré que cela n'améliorait pas les performances sur notre système spécifique, mettant en évidence l'importance de tester et d'ajuster les stratégies de parallélisation en fonction de l'architecture cible.

4 Parallélisation Coarse-Grain

L'approche coarse-grain a été mise en œuvre dans la méthode `iteration` de la classe Scheme. Contrairement à la parallélisation fine-grain, cette méthode divise le domaine de calcul global entre les threads. Utilisant `#pragma omp parallel`, nous avons créé un bloc parallèle où chaque thread exécute un segment de calcul distinct, défini par `startIndex` et `endIndex`.

Dans ce contexte, les variables liées à chaque segment de calcul sont déclarées comme privées à chaque thread pour éviter les conflits d'accès. En ce qui concerne la variable `m_duv`, qui accumule des valeurs de tous les threads, nous avons employé `reduction(+:m_duv)`. Cette directive assure que chaque thread contribue de manière sûre à la variable partagée, en évitant les problèmes d'accès concurrents et en préservant l'intégrité des résultats.

5 Équilibrage de Charge

L'équilibrage de charge est une technique avancée mise en œuvre dans notre projet pour optimiser la performance de la parallélisation coarse-grain. Cette méthode, intégrée dans la classe `Parameters` via la fonction `balance`, rééquilibre la charge de travail entre les threads en fonction de leur performance historique.

Le processus commence par mesurer et enregistrer le temps de calcul pour chaque ligne du domaine par chaque thread, stocké dans `m_times_lines`. Cette mesure permet de déterminer les sections du domaine qui prennent plus de temps à calculer, fournissant une base pour la répartition des charges de travail lors de l'équilibrage.

La fréquence de l'équilibrage est contrôlée par la variable `m_nBalances`, qui détermine à quelle fréquence la répartition des tâches doit être réévaluée et ajustée. L'équilibrage est effectué en recalculant les indices de début (`startIndex`) et de fin (`endIndex`) pour chaque thread, s'assurant ainsi qu'ils traitent un nombre équivalent de lignes nécessitant une charge de calcul similaire.

Cette approche dynamique permet d'optimiser l'utilisation des ressources en fonction des besoins réels de calcul, menant à des gains significatifs en termes de vitesse d'exécution et d'efficacité. Nos tests ont montré que l'équilibrage de charge, surtout lorsqu'il est réalisé fréquemment, améliore notablement les performances par rapport à une distribution statique du travail, en particulier dans des configurations avec un nombre élevé de threads.

6 Résultats Expérimentaux

Les résultats ont montré une amélioration linéaire de la performance avec la parallélisation fine-grain, atteignant un speedup de 2.612 avec 8 threads. En revanche, la parallélisation coarse-grain sans équilibrage de charge a donné un speedup de 2.419 avec 8 threads. L'introduction de l'équilibrage de charge a significativement amélioré les performances, atteignant un speedup de 4.689 avec 8 threads, démontrant l'efficacité de cette approche.

7 Conclusion

En conclusion, ce projet a démontré l'efficacité des techniques de parallélisation dans l'optimisation des performances de calculs intensifs. L'implémentation de la parallélisation fine-grain et coarse-grain, ainsi que l'ajout d'un système d'équilibrage de charge dynamique, ont permis d'obtenir des améliorations significatives en termes de speedup, dans le contexte de l'utilisation de plusieurs cœurs de processeur.