UNIVERSITÉ CÔTE D'AZUR

MASTER INFORMATIQUE

Rapport de TP3

Placement des threads sur les cœurs: Threads affinity

Réalisé par
Mehdi Mansour

Encadré par
Professeur Sid Touati

Master Informatique
à Université Côte d'Azur
Premier semestre de l'année 2024-2025

Abstract

Ce rapport a pour objectif de détailler les différents résultats obtenus lors de ce TP3 disponible dans l'archive, nommé ≪Affinity.pdf≫.

Architecture utilisée pour ce TP

CPU name: Intel(R) Core(TM) i7-10700F CPU @ 2.90GHz
CPU type: Intel Cometlake processor
CPU stepping: 5

Hardware Thread Topology

Sockets: 1
Cores per socket: 8
Threads per core: 2
Socket 0: (0 8 1 9 2 10 3 11 4 12 5 13 6 14 7 15)

NUMA domains: 1
Domain: 0
Processors: (0 8 1 9 2 10 3 11 4 12 5 13 6 14 7 15)
Distances: 10
Free memory: 21746.9 MB
Total memory: 31991.4 MB

Topologie du PC

Topologie graphique								
Core	0 8	1 9	2 10	3 11	4 12	5 13	6 14	7 15
Cache L1	32 kB	32 kB	32 kB	32 kB	32 kB	32 kB	32 kB	32 kB
Cache L2	256 kB	256 kB	256 kB	256 kB	256 kB	256 kB	256 kB	256 kB
Cache L3	16 MB							

Contents

Part I
Introduction

L'objectif de ce TP est d'étudier le placement des threads sur divers cœurs ainsi que leur impact sur les performances. Dans le cadre de ce TP, nous utiliserons le code C d'un fichier nommé «matrixmatrixmultiply.c», constitué de plusieurs multiplication de matrices, est un code massivement parallèle. Ce code est une implémentation d'une version parallèle d'OpenMP, son exécution créera plusieurs threads.
Lors de ce TP, nous utiliserons la commande *time* pour connaître le temps réal de notre programme ainsi que *RStudio*, un IDE pour le langage R, permettant de visualiser graphiquement des datas que nous récolterons au fur-et-à-mesure du TP.

Part II
Mesures des performances

Dans cette partie nous réaliserons des benchmarks pour tester les performances du fichier C «matrixmatrixmultiply.c». Nous allons pour cela, utiliser des scripts pour générer des données, avec la commande *time*, à partir de notre fichier compiler avec gcc et icx avec du placement de thread. On aura trois types de données :

- données générées avec des threads sans affinité (placé par l'OS)
- données générées avec des threads compact (assignés à des cœurs proches pour favoriser le partage de cache)
- données générées avec des threads scatter (placé sur des cœurs éloignés pour favoriser l'utilisation des caches individuels)

Ces expérimentations nous permettront de comparer l'impact des différentes stratégies de placement des threads sur les performances de notre programme.

1 Exécution des scripts

Après la commande *make*, qui nous permet de compiler notre programme avec les différents compilateur, on exécute les trois scripts du dossier *bench/* pour obtenir, pour chaque compilateur, trois fichiers de données. les fichiers générés sont les suivant :

- gcc.data et icx.data pour les données de threads sans affinité
- gcc_compact.data et icx_compact.data pour les données de threads compact
- gcc_scatter.data et icx_scatter.data pour les données de threads scatter

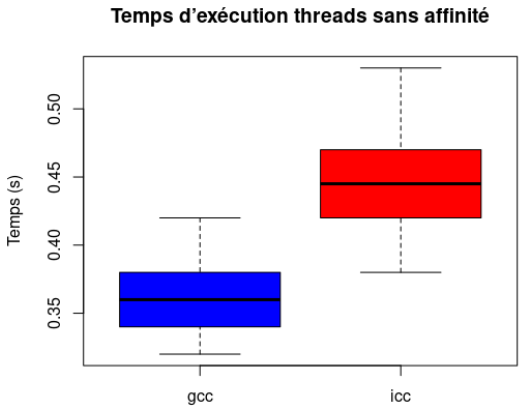
Chaque fichier est composé de 50 temps réals qui nous servirons pour visualiser l'efficacité de nos stratégies.

2 Résultats obtenus

A l'aide de *RStudio* et de son outil *boxplot*, nous pouvons visualiser les données que nous avons généré plus haut.

2.1 Threads sans affinité

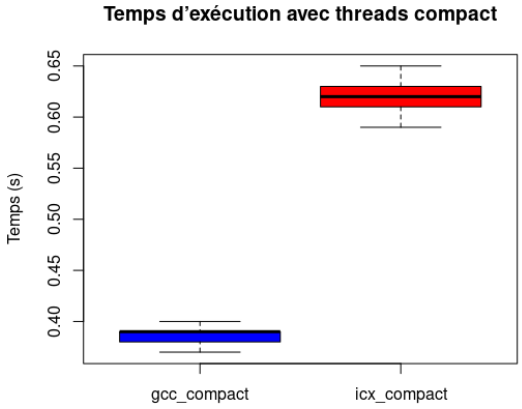
Le placement de threads sans affinité est fait par l'OS qui les place automatiquement. C'est lui qui décide quels cœurs physique ou logiques seront utilisés pour exécuter les threads. Voici les résultats obtenus pour les exécutions avec threads sans affinité :



On remarque que pour des threads sans affinité, *gcc* est plus performant que *icx*. On remarque également qu'il y a un intervalle non régulier de données, allant de 0,32 à 0,42 pour *gcc* et de 0,38 à 0,57 pour *icx*.

2.2 Threads compact

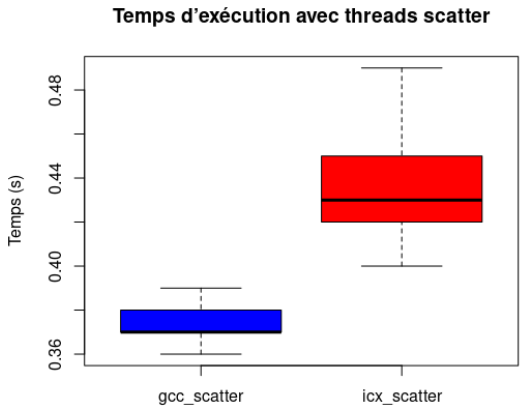
Ici le placement des threads se fait manuellement grâce au script.
Voici les résultats obtenus pour les exécutions avec placement de threads rapproché :



On remarque que pour des threads placés de manière rapproché, *gcc* est une nouvelle fois plus performant que *icx*. On a un intervalle de données restreint, allant de 0,37 à 0,41 pour *gcc* et de 0,59 à 0,65 pour *icx*.

2.3 Threads scatter

Ici le placement des threads se fait manuellement grâce au script.
Voici les résultats obtenus pour les exécutions avec threads éloignés :



On remarque que pour des threads placés de manière éloigné, *gcc* est une nouvelle fois plus performant que *icx*. On a un intervalle de données allant de 0, 35 à 0,40 pour *gcc* et de 0,40 à 0,52 pour *icx*.

Part III
Conclusion

Durant ce TP, nous avons étudié l'importance des placements de threads pour améliorer les performances de notre programme. Nous avons, pour cela, exploré trois types de placements de threads :

Le placement de threads dit sans affinité : Ici c'est l'OS qui place les threads automatiquement. Les résultats obtenus ont montré une grande diversité de temps de part le grand intervalle de valeurs générés. L'avantage de ce placement est que l'OS s'occupe de tout mais cette méthode peut être inefficace en termes d'utilisation du cache car les threads peuvent être déplacés d'un cœur à un autre au cours de l'exécution, ce qui explique le grand intervalle de données.

Le placement de threads compact : On assigne volontairement des threads à des cœurs proches les uns des autres pour un partage du cache. Cette méthode permet un partage du cache plus efficace et donc d'améliorer la communication entre les threads, ce qui réduit le temps d'exécution. On le confirme avec les résultats obtenus, l'intervalle de données étant petit. Les inconvénients ici sont la saturation du cache et la sous-utilisation des autres cœurs.

Le placement de threads scatter : On placement les threads sur des cœurs éloignés les uns des autres. Cela permet au thread d'avoir un accès dédié au cache, réduisant les interférences entre les threads. Utile pour les applications qui ne partagent pas les même données. Cela peut également augmenter la latence de communication entre les threads s'il y a des dépendances de données entre les eux.

En résumé, il est nécessaire d'étudier notre application en profondeur pour connaître les éventuelles dépendances de données pour placer nos threads en fonction des besoins.