Université Versailles-Saint-Quentin

Paris-Saclay

Spécialité :
Calcul Haute Performance, Simulation

Module :
Architecture et Optimisation de Code

February 22, 2025

Rapport :
Rapport TP1.

Présenté par:
- Sofiane ARHAB.

Encadré par:
- Emmanuel Oseret.

Contents

List of Figures

List of Tables

1 Description de l'architecture cible :

La suite de ce TP sera réaliser sur l'architecture cible suivante, on décris également les différentes versions des programmes utilisé :

Modèle	Coeurs	Fréquence(GHz)	L1(KiB)	L2(MiB)	L3(MiB)	Cache line (o)
AMD Ryzen 7 6800H	8	1.6 - 4.7	256	4	16	64

Table 1: Architecture cible (x86_64)

Compilateur	Version
GCC	13.1.0
OpenMP	201511
MAQAO	2.21.1
OS	Linux Mint 21.2 x86_64

Table 2: Versions programmes utilisé

Comme une partie de ce TP reposent sur des I/O, les caractéristiques du disque sont également a prendre en considération. Afin de déterminer le disque de la machine cible on utilise la commande `lshw -class disk -class storage`, pour la suite de ce travail toutes les mesures seront effectué sur le matériel suivant :

Modèle	Type	Interface	Protocole	Capacité (Gb)	Vitesse Lecture séquentielle Max (MB/S)	Vitesse Ecriture séquentielle Max (MB/S)	Cache (Mb)
Intel SSD 670p SSDPEKNU 512GZ[?]	SSD M.2	PCIe 3.0 x4	Nvme 1.3	512	3000	1600	256

Table 3: Caractéristique du disque

2 Protocole de mesure

Afin de garantir une stabilité et une reproductibilité des résultats, on s'assure de mettre en oeuvre les points suivants

- Brancher le laptop à la prise d'alimentation.
- Fermer toutes application tierces, les autre processus indésirable et couper le réseau.
- Tuer l'interface graphique `Ctrl + ALt + F3` (tty3).
- Définir le Gouverneur du CPU en mode "performance" avec la commande `sudo cpupower frequency-set -g performance`.
- Les optimisations seront cumulatives , chaque nouvelle optimisation se base sur la précédente.

- Durant les optimisation de nature séquentielle on pineras l'exécutions de notre kernel sur un cœur unique avec la commande `taskset -c 3 ./mat_grid [50] [2000] [3000]`. afin d'éviter les effets de migration du processus entre les cœurs fait par l'OS.
- De même durant les optimisation parallèle utilisant OPENMP, on utiliseras `OMP_PLACES=cores OMP_PROC_BIN`

3 Mat-Grid

Le code à optimiser est un code jouet vissant a reproduire un environnement de code industriel HPC dans lequel on serait amener a faire des IO dans des fichiers afin d'écrire ou de lire des données,, typiquement les résultats de simulation par exemple, bien que le format du fichier est un format basique `.txt`, qui ne correspond pas aux environnement HPC qui utilisent en générale le format `HDF5`, cet exercice représente une bonne mise en situation. La version de base sera compilé avec la commande suivante `gcc -g3 -O2 baseline.c -o exe`

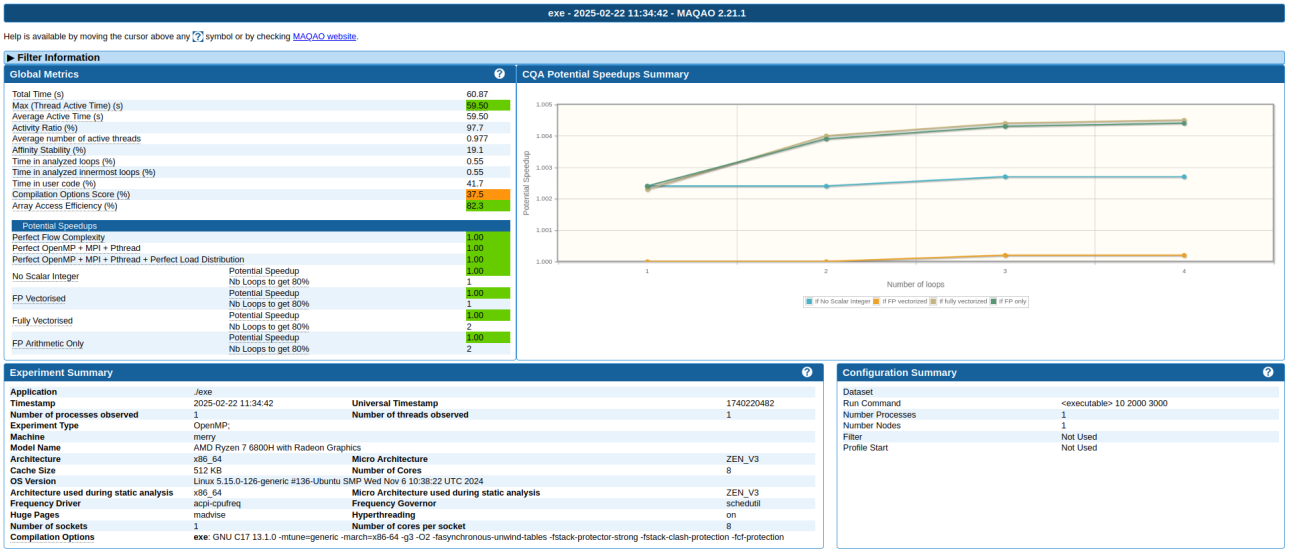

Figure 1: MAQAO - Sortie One View Version de Base

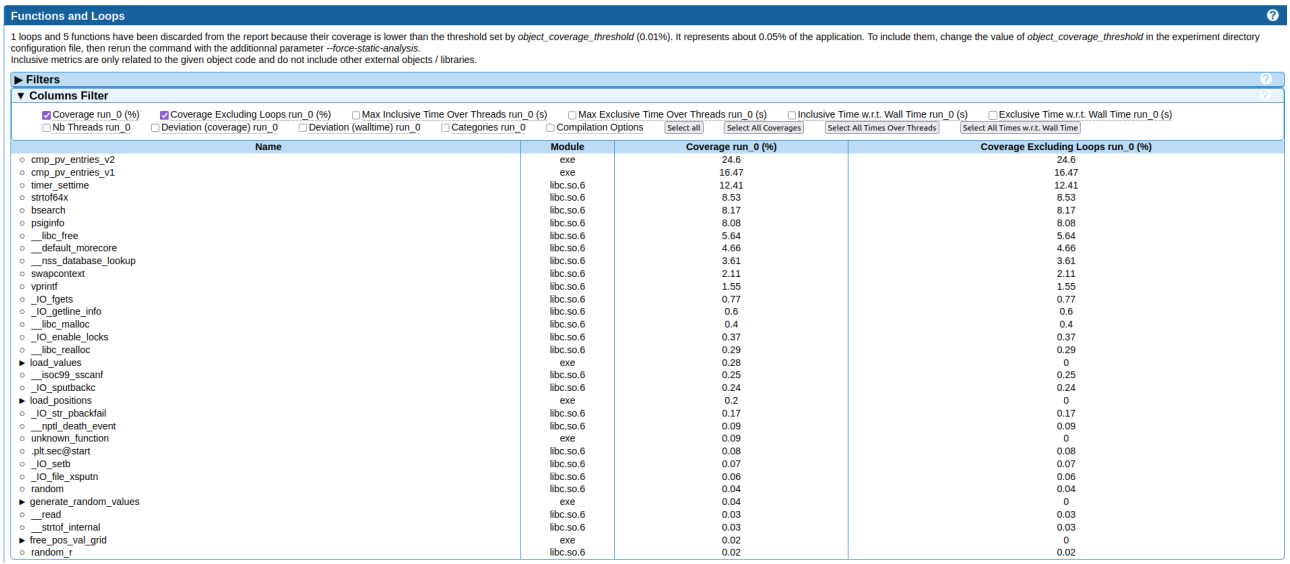

Figure 2: MAQAO - Répartition du temps par fonction

Version	Temps d'exécution (s)
Baseline	291.28

Table 4: Temps d'exécution par version

On constate que la majeur partie du temps est passé dans les opération de comparaison nécessaire à l'algorithme du quicksort, on retrouve également beaucoup de temps passé dans les I/O ce qui n'est

pas surprenant vu la nature du code , mais également beaucoup de temps passer dans les fonctions de la `GlibC` surtout `malloc/free`, ce qui s'explique par le faite qu'on alloue individuellement chaque paire de données et leurs positions

3.1 Optimisation 0 : Compilation

Le premier rapport MAQAO nous indique un score de compilation faible, on ajoute donc les flags d compilations suivant `-ftreevectorize` afin d'inciter le compilateur a vectoriser automatiquement si il le peux, `-funroll-loops` afin d'inciter le compilateur a dérouler les boucles si il le peux, `-fno-omit-frame-pointer` ce flag a été suggérer par MAQAO afin avoir plus de détails sur les callchains, `-march=native` pour cibler l'architecture de notre CPU (ZEN3), ainsi que `-Ofast` plus grand niveau d'optimisation de `GCC` étant donné qu'il n'y a pas de calcul numérique, on ne risque rien en terme de précision.

Version	Temps d'exécution (s)
Baseline	291.28
OPT0 (Compilation)	59.22

Table 5: Temps d'exécution par version

3.2 Optimisation 1 : Scanf

Comme on le disait en introduction, sur l'ensemble du temps d'exécution du programme une grande partie se perd dans les IO, ceux ci sont fait en utilisant un combo `fgets+fprintf` ou `fscanf`, nous allons simplement remplacer par des scanf pour les lectures sans passez par le buffer de `fgets` qui est surtout utilisé pour des raisons de sécurité.

Version	Temps d'exécution (s)
Baseline	291.28
OPT0 (Compilation)	59.22
OPT1 (Scanf)	54.78

Table 6: Temps d'exécution par version

3.3 Optimisation 2 : Remplacement qsort

L'approche consistant a trier le tableau afin d'en trouver le plus grand élément est correcte mais inutilement chère, le trie `quicksort` est un algorithme de trie efficace mais il reste $O(logN)$, à la place pour trouver le maximum on itère tout simplement une fois dans le tableau en $O(N)$. C'est de loin l'optimisation la plus rentable hormis la compilation avec les bons flags.

Version	Temps d'exécution (s)
Baseline	291.28
OPT0 (Compilation)	59.22
OPT1 (Scanf)	54.78
OPT2 (Remplcaement qsort)	16.67

Table 7: Temps d'execution par version

3.4 Optimisation 3 : Passage pointeur 1D

La structure actuelle du code fait autant d'allocation qu'il n'y a de paire a générer, tout ces appels à `malloc/free` sont couteux , nous allons donc à la place changer de structure en passant sur un pointeur 1D sur `entries`, afin de pouvoir allouer en une seule fois un gros bloc mémoire , et le libérer en une seule fois également.

!

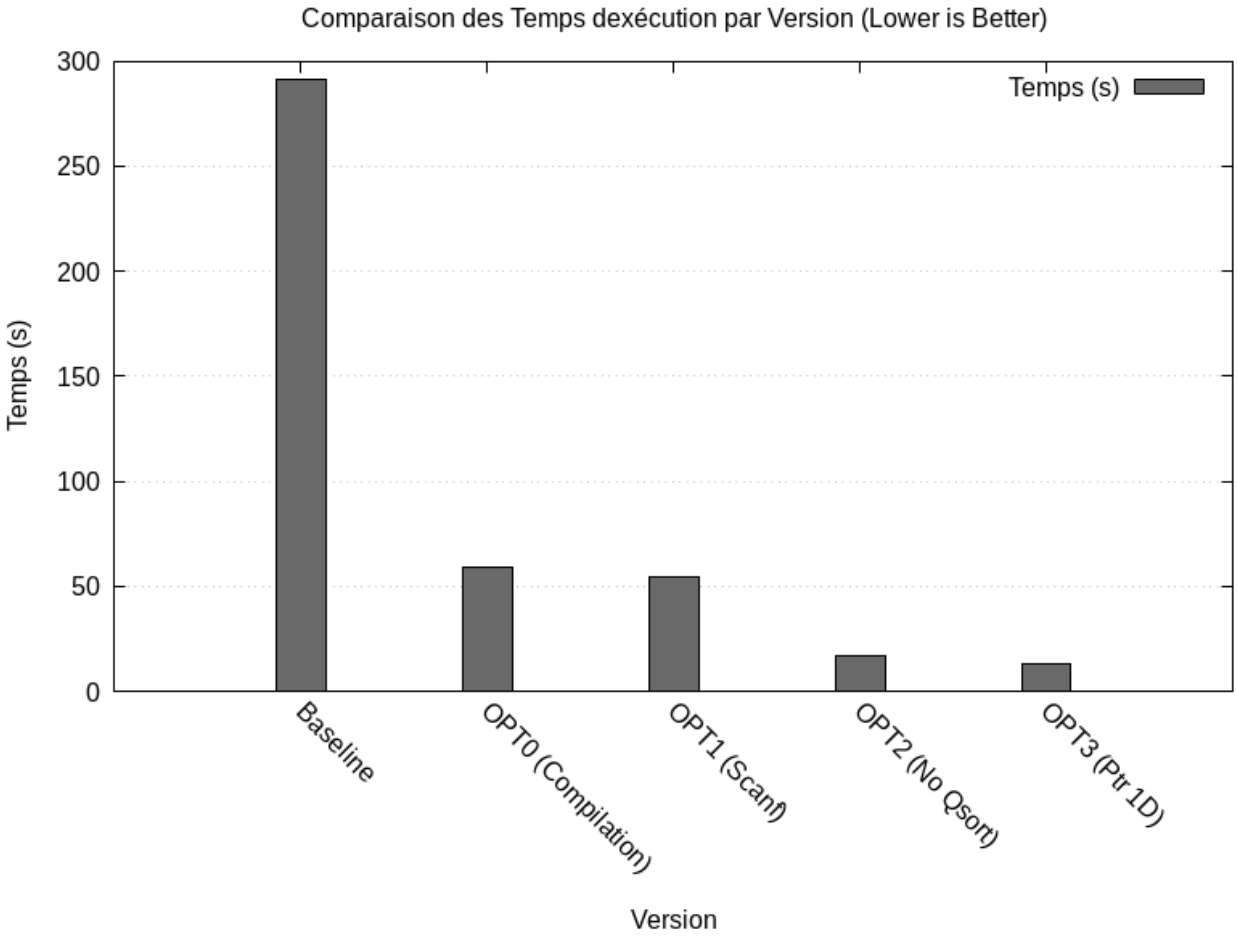

Figure 3: Temps d'exécution des différente version repm=10 X=2000 Y=3000

L'intégralité de ce travail est disponible dans ce dépot Github.