UNIVERSITÉ CÔTE D'AZUR

MASTER INFORMATIQUE

Rapport de TP4

Mesure et analyse de la consommation d'énergie électrique des programmes

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 TP4 disponible dans l'archive fournisse.

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 la consommation d'énergie électrique des programmes. Pour cela, nous utiliserons l'outil EcoFloc, qui capture la consommation d'énergie d'un programme dans un temps donné grâce à son pid. Notre objectif est donc de calculer la consommation de trois programmes (disponible dans l'archive fournisse) :

- ≪cpu.c≫
- ≪memory.c≫
- ≪rw.c≫

Pour récupérer des données, il faut suivre une méthode fastidieuse. Il faut deux terminaux, l'un pour exécuter le programme et récupérer son pid, l'autre pour effectuer les commandes EcoFloc. Pour faciliter notre travail, nous allons utiliser un script nommé ≪exec.sh≫ qui réalisera cette méthode à notre place. Pour visualiser nos résultats nous utiliserons enfin *RStudio*, un IDE pour le langage R, permettant de visualiser graphiquement des données que nous récolterons au fur-et-à-mesure du TP.

Part II
Installation et prise en main

Cette partie est sans doute la plus difficile de ce TP, il faut suivre en détail les indications d'installations puis tester l'outil. Dans notre cas, pour effectuer nos calculs il faut obligatoirement lancer la commande :

```
./ecofloc-cpu.out -i 1000 -t 10 -p 55334

exemple de resultat obtenu  :
****************************
/ECOFLOC_CPU_PID_55334
****************************
Average Power : 0.38 Watts
Total Energy : 3.41 Joules
****************************
```

du répertoire */opt/ecofloc* ce qui n'est pas spécifié lors de l'installation, peut être car l'installation a été mal faite de notre coté.

Pour que EcoFloc récupère bien des données il faut que notre programme ne se finisse pas trop vite, ce qui était le cas au début.

Nous avons donc opté, dans un premier temps, pour une boucle infini, mais cela posait problème car nous ne pouvions exécuter un script avec une boucle infini (impossible de passer d'une exécution à une autre).

Nous avons , dans un second temps, au lancement du programme, bloquer l'exécution avec un *getchar()* pour récupérer le pid du programme sans entrer dans la boucle et ainsi lancer la commande d'EcoFloc pour récupérer nos données. Cette méthode n'était pas possible à généraliser avec un script car la boucle infini ne peut se terminer que si on l'interrompt manuellement, ce qui interrompt également le script.

Nous avons donc finalement opté pour l'ajout d'une horloge, grâce à la librairie *time.h*, pour que la boucle se termine au bout d'un certain temps. Nous n'avions plus besoin du *getchar()* car grâce à l'horloge et au script ≪exec.sh≫, nous pouvions exécuter le fichier binaire, récupérer le pid, lancer la commande d'EcoFloc et écrire les résultats dans un fichier csv pour être interpréter par RStudio. On initialise l'horloge en passant en argument à l'exécution notre temps.
Exemple :

```
./fichier_binaire 10, la boucle durera 10sec
```

Pour généraliser cela, notre script pouvant prendre 2 arguments : le temps (obligatoire) et le nombre de répétition (par défaut à 1), va exécuter cette méthode avec tous les fichier binaires.
Exemple :

```
./exec.sh 10 2
```

EcoFloc va récupérer pour chaque fichier binaire, la consommation d'énergie sur les 10 prochaines secondes et le tout 2 fois. Pour éviter d'être trop rapide, on rajoute 2 secondes au temps passé en argument. Ce qui veut dire que pour chaque exécution, la boucle du programme durera 2 secondes de plus que le temps de calcul d'EcoFloc pour lui permettre de récupérer les données à temps.

Part III
Micro-benchmarks CPU-bound

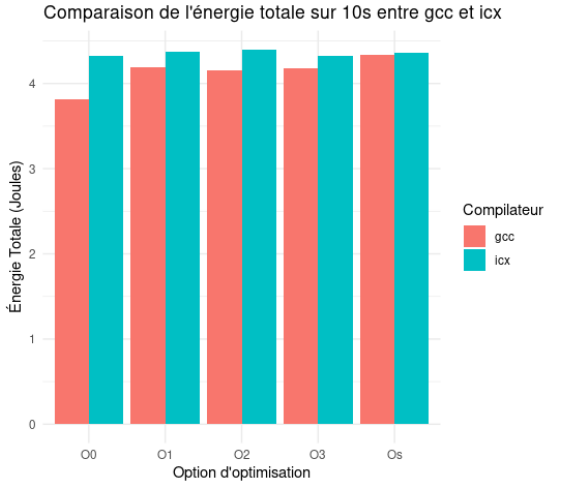
Dans cette partie nous allons appliqué la méthode décrite plus haut au programme ≪cpu.c≫. Nous allons calculer la consommation de ce programme pour différentes option d'optimisation et différents compilateur (ici *gcc* et *icx*). Pour cela, nous compilons notre programme grâce au script ≪compiler.sh≫.

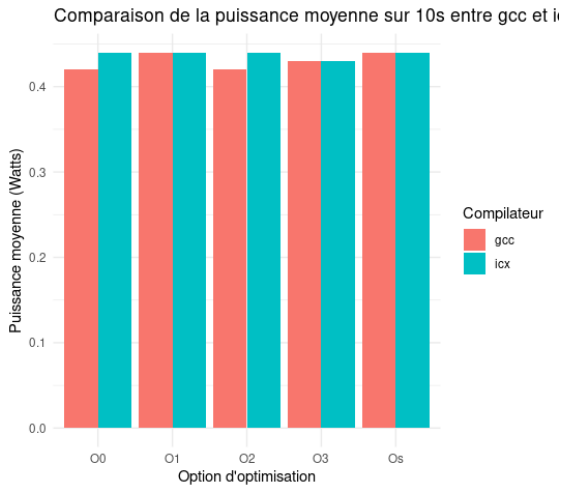
1 Capture des données sur 10 secondes

Nous allons dans un premier temps exécuter notre script ≪exec.sh≫ 2 fois pour 10 secondes avec la commande :

```
./exec.sh 10 2
```

Nos résultats sont stockés dans ≪res10s.csv≫ puis traités dans *RStudio*. Nous obtenons les graphiques suivant :





On remarque que *gcc* consomme moins d'énergie que *icx*. On remarque également que la consommation électrique augmente en fonction de l'option d'optimisation pour *gcc*. Pour

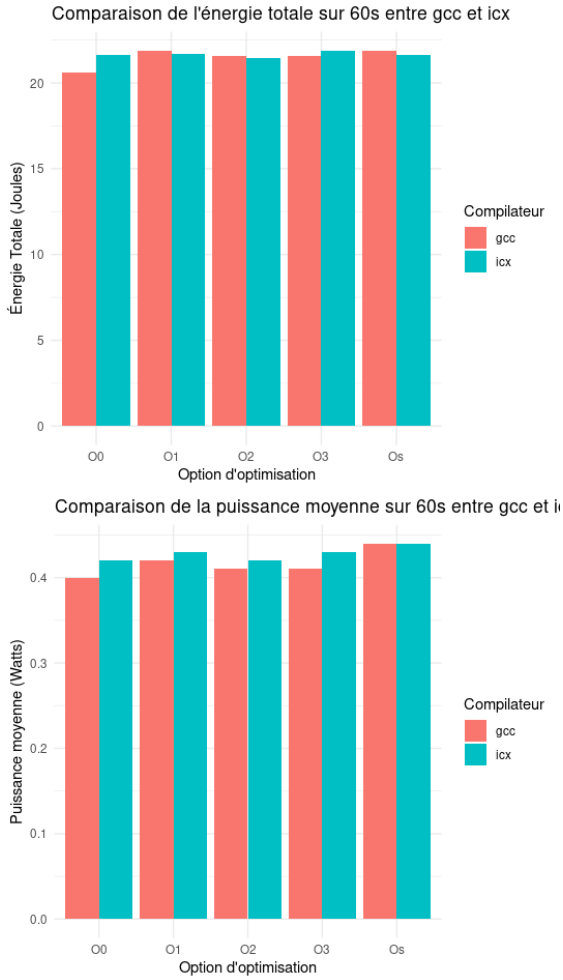
icx cette consommation est stable Pour la puissance moyenne, cela varie mais globalement *gcc* utilie moins de puissance que *icx*.

2 Capture des données sur 60 secondes

Nous allons dans un premier temps exécuter notre script ≪exec.sh≫ 1 fois pour 60 secondes avec la commande :

```
./exec.sh 60
```

Nos résultats sont stockés dans ≪res60s.csv≫ puis traités dans *RStudio*. Nous obtenons les graphiques suivant :



On remarque que *gcc* consomme plus d'énergie de manière général que *icx* (consomme plus que *icx* en O1, O2, Os). Cette consommation augmente légèrement en fonction des options. Pour la puissance, comme pour 10 secondes, *gcc* consomme moins de puissance.

Part IV
Micro-benchmarks Memory-bound

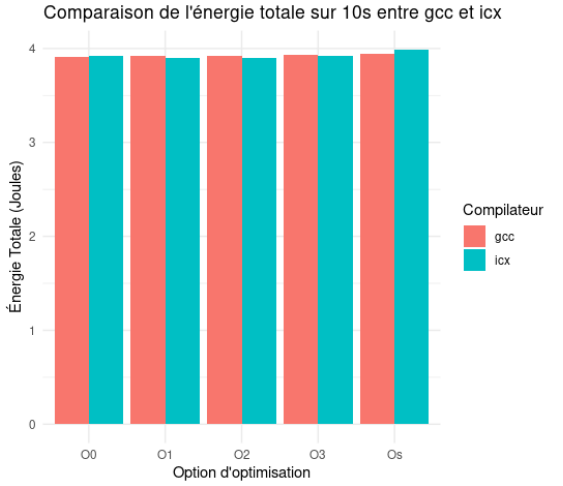
Dans cette partie nous allons appliqué la méthode décrite plus haut au programme ≪cpu.c≫. Nous allons calculer la consommation de ce programme pour différentes option d'optimisation et différents compilateur (ici *gcc* et *icx*). Pour cela, nous compilons notre programme grâce au script ≪compiler.sh≫.

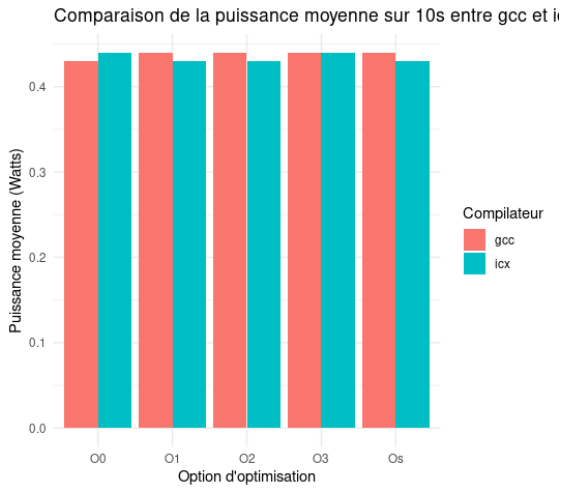
1 Capture des données sur 10 secondes

Nous allons dans un premier temps exécuter notre script ≪exec.sh≫ 2 fois pour 10 secondes avec la commande :

```
./exec.sh 10 2
```

Nos résultats sont stockés dans ≪res10s.csv≫ puis traités dans *RStudio*. Nous obtenons les graphiques suivant :





On remarque que *icx* consomme moins d'énergie que *gcc* pour les options O1, O2 et O3. On a une consommation d'énergie stable quel que soit l'option pour les deux compilateurs.

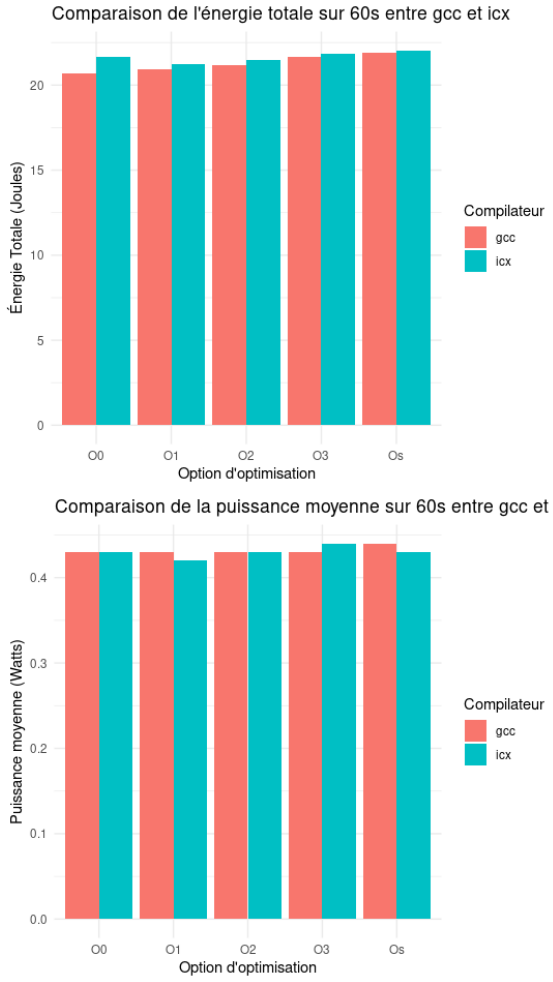
Pour la puissance, *gcc* en consomme plus qu'*icx* pour les options O1, O2, Os.

2 Capture des données sur 60 secondes

Nous allons dans un premier temps exécuter notre script ≪exec.sh≫ 1 fois pour 60 secondes avec la commande :

```
./exec.sh 60
```

Nos résultats sont stockés dans ≪res60s.csv≫ puis traités dans *RStudio*. Nous obtenons les graphiques suivant :



On remarque que pour 60 secondes *gcc* consomme d'énergie moins que *icx* pour chaque option. On constate que la consommation d'énergie augmente en fonction des options pour les deux compilateurs. En ce qui concerne la puissance, les deux se valent (*icx* légèrement mieux).

Part V
Micro-benchmarks avec des appels système

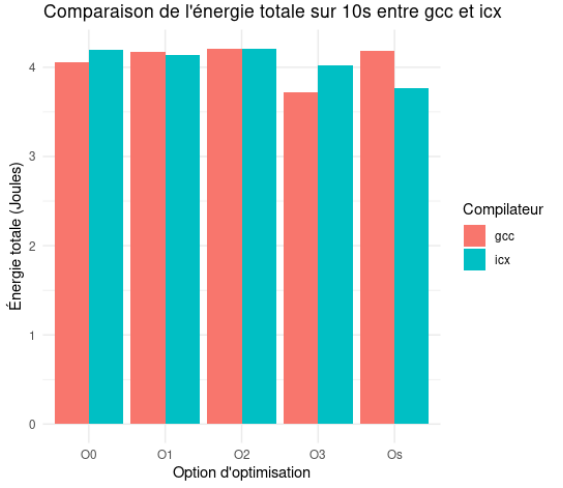
Dans cette partie nous allons appliqué la méthode décrite plus haut au programme «cpu.c». Nous allons calculer la consommation de ce programme pour différentes option d'optimisation et différents compilateur (ici *gcc* et *icx*). Pour cela, nous compilons notre programme grâce au script «compiler.sh».

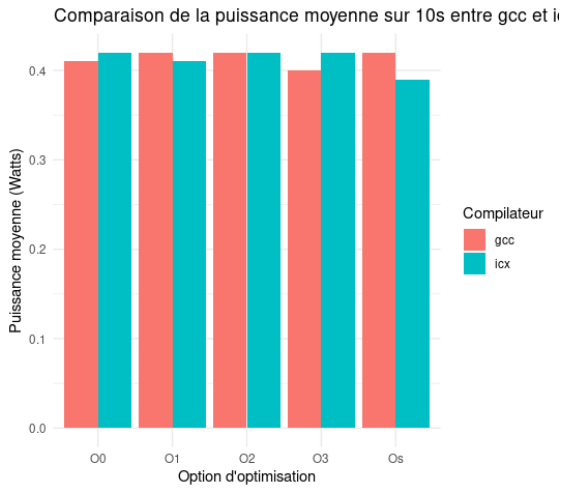
1 Capture des données sur 10 secondes

Nous allons dans un premier temps exécuter notre script «exec.sh» 2 fois pour 10 secondes avec la commande :

```
./exec.sh 10 2
```

Nos résultats sont stockés dans «res10s.csv» puis traités dans *RStudio*. Nous obtenons les graphiques suivant :





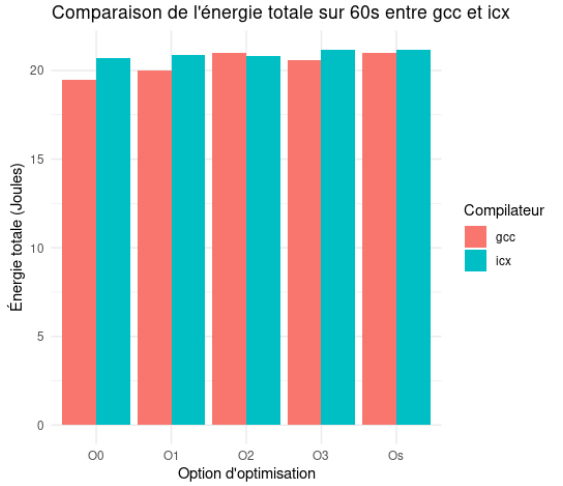
On remarque que les compilateurs se valent pour la consommation d'énergie. La consommation d'énergie est rélativement stable quel que soit l'option. En ce qui concerne la puissance, les deux compilateurs se valent également.

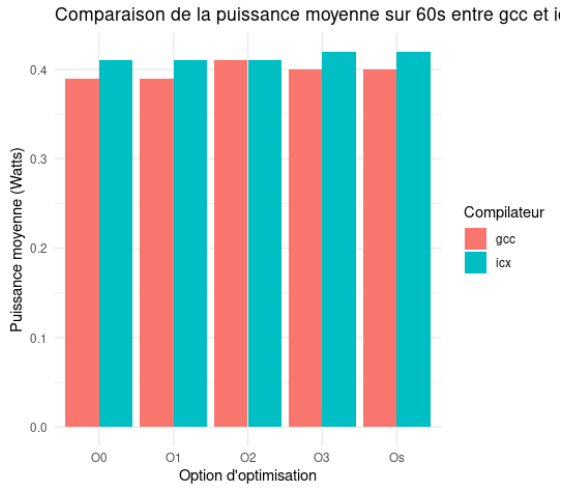
2 Capture des données sur 60 secondes

Nous allons dans un premier temps exécuter notre script ≪exec.sh≫ 1 fois pour 60 secondes avec la commande :

```
./exec.sh 60
```

Nos résultats sont stockés dans ≪res60s.csv≫ puis traités dans *RStudio*. Nous obtenons les graphiques suivant :





On remarque que *gcc* consomme moins d'énergie que *icx*. Il consomme également moins de puissance que *icx*. La consommation d'énergie est rélativement stable quel que soit l'option pour les deux compilateurs.

Part VI
Conclusion

Dans le cadre de ce TP, nous avons étudié la consommation d'énergie électrique des programmes. Nous avons remarqué que plus l'exécution est long est plus la consommation augmente. Les résultats obtenus lors de ce TP doivent être interprétés avec précaution, car la consommation peut varier d'une exécution de script à l'autre en raison de la présence d'autres processus en cours. Il ne faut pas négliger cela. Ces resultats donnent malgré tout un ordre d'idée de la consommation d'énergie électrique d'un programme.
En ce qui concerne EcoFloc, c'est un outil utile mais qui devient fastidieux s'il est utilisé manuellement pour chaque programme. Dans notre cas, l'utilisation d'un script était nécessaire.