# Rapports-TD-OS202

Laura Montagnier

Février 2024

# 1 TD numéro 1

## 1.1 Produit matrice-matrice

### 1.1.1 Question 1

Les calculs sont toujours bien plus longs, et même parfois deux fois plus longs, pour la dimension 1024 que pour les dimensions 1025 et 1023.

**Pourquoi donc cela ?**

Les calculs nécessitent un octet de plus pour stocker les données.

```
laura@Ordi2Lau:~/Promotion2024/TravauxDiriges/TD_numero_1/sources$ ./TestProductMatrix.exe 1023
Test passed
Temps CPU produit matrice-matrice naif : 1.80956 secondes
MFlops -> 1183.27
laura@Ordi2Lau:~/Promotion2024/TravauxDiriges/TD_numero_1/sources$ ./TestProductMatrix.exe 1024
Test passed
Temps CPU produit matrice-matrice naif : 12.6695 secondes
MFlops -> 169.501
laura@Ordi2Lau:~/Promotion2024/TravauxDiriges/TD_numero_1/sources$ ./TestProductMatrix.exe 1025
Test passed
Temps CPU produit matrice-matrice naif : 2.04724 secondes
MFlops -> 1052.04
```

### 1.1.2 Question 2

Définition : Les MFLOPS (Millions de Floating Point Operations Per Second) est une unité de mesure de la performance d'un système informatique en termes de capacité à effectuer des opérations à virgule flottante par seconde.

Le tableau suivant présente les résultats en MFlops.

| Imbrication des boucles | 1023 | 1024 | 1025 |
|---|---|---|---|
| ijk | 161 | 102 | 164 |
| kij | 81 | 41 | 79 |
| ikj | 79 | 51 | 83 |
| jki | 985 | 978 | 991 |
| jik | 162 | 106 | 158 |
| kji | 825 | 415 | 816 |

Au plus le nombre de MFlops est élevé, au plus le temps d'exécution est faible. La boucle optimale est donc jki.

**Pourquoi changer l'ordre des boucles change le temps d'exécution alors que la complexité reste la même ?**

C'est à cause de la complexité d'accès mémoire : de la hiérarchie des caches. C'est un problème qui n'existait pas quand les processeurs n'avaient qu'un seul coeur.

L'information est stockée en colonnes, c'est donc plus rapide de parcourir les colonnes plutôt que les lignes : parcourir les lignes implique un chemin NON CONTIGU, alors que parcourir les colonnes implique un chemin CONTIGU. Il faut donc maximiser le parcours par colonnes :

- pour j in colonne
- pour k in colonne
- pour i in colonne

### 1.1.3 Question 3

**Utilisation de OpenMP :** Je n'ai pas compris grand chose à l'utilisation de OpenMP. Cela me bloque pour les questions 3 et 4.

### 1.1.4 Question 4

### 1.1.5 Question 5

## 1.2 Parallélisation MPI

### 1.2.1 L'algorithme de la pièce

home > laura > piece.py > ...

```python
from mpi4py import MPI

# On initialise le MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# On initialise le jeton
jeton = None

# On code pour le procédé zéro, qui doit en quelque sorte "initialiser"
if rank == 0:
    jeton = 1

    dest_rank = (rank + 1) % size #On travaille modulo le nombre de procédés.
    comm.send(jeton, dest=dest_rank)

    jeton = comm.recv(source=size - 1)

    print(f"Le jeton est égal à {jeton}")


# Il faut coder l'action symétrique à l'envoi, c'est-à-dire la réception,
# pour tous les procédés non égaux à zéro.

else:
    # On reçoit le jeton du procédé précédent
    source_rank = (rank - 1) % size
    jeton = comm.recv(source=source_rank)


    jeton += 1


    dest_rank = (rank + 1) % size
    comm.send(jeton, dest=dest_rank)

# On "ferme" MPI
MPI.Finalize()

```

### 1.2.2 L'algorithme de calcul de pi

Le code est présenté dans le dépôt Github. Son résultat est :

```
laura@Ordi2Lau:~/OS201$ /bin/python3 /home/laura/OS201/pi.py
Temps mis pour calculer Pi: 7.876174211502075 seconds
La valeur de Pi est approximativement: 3.1418881
```

Remarque : Le temps d'exécution semblait varier entre 5 et 30 secondes. Cela reste très bas pour obtenir une approximation au centième près !