Rapport examen machine OS202

LEGROS Gabriel

Mars 2025

Table des matières

1 Environnement de calcul

```
Get-WmiObject Win32_Processor | Format-List NumberOfLogicalProcessors, L2CacheSize, L3CacheSize
Get-WmiObject Win32_CacheMemory
```

Propriété	Valeur
NumberOfLogicalProcessors	8
L1 Cache Size	256 Ko
L2 Cache Size	1 Mo
L3 Cache Size	8 Mo

TABLE 1 – Informations sur le processeur et les caches

2 Parallélisation d'images issues d'une vidéo

2.1 Stratégie de parallélisation

Pour effectuer la parallélisation MPI, nous allons former des sous-groupes d'images à partir des images initiales.
En effet, le processus 0 (de rang zéro) va diviser le nombre d'images totale en nbp (correspondant au nombre de processus) groupes et les répartir aux processus suivants. Chacun va donc recevoir un lot d'images sur lesquelles il va calculer et donc appliquer le filtre Gaussien puis le filtre de netteté donc les noyaux sont :

$$F_G = \frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

$$F_S = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

Notons que si le nombre N d'images n'est pas divisible en nbp groupes, nous distribueront le reste le plus équitablement possible aux processus.

Une fois les opérations de chaque processus terminées, les résultats sont sauvegardés et le processus 0 s'occupe d'afficher le temps d'exécution

Cette approche est bien adaptée ici et optimale car premièrement les images sont indépendantes donc on peut tout à fait les répartir pour les calculs de convolution.
De plus, comme on a un nombre important d'image, il est plus convenable de les répartir sur des processus différents pour alléger la charge de calcul et minimiser le temps d'exécution.
Ensuite les images sont distribuées en seulement une communication avec le processus 0 puis sauvegardées par chaque processus ce qui permet de limiter la surcharge de communication, qui peut ralentir le calcul.

2.2 Parallélisation MPI

Voir fichier.

```
mpiexec -n 4 python movie_filter.py
```

A partir de $nbp = 5$ processus, le programme ne s'exécute plus car j'ai une erreur mémoire liée à sa capacité. Cela est sûrement lié à la taille importante des images.

2.3 Résultats et accélérations

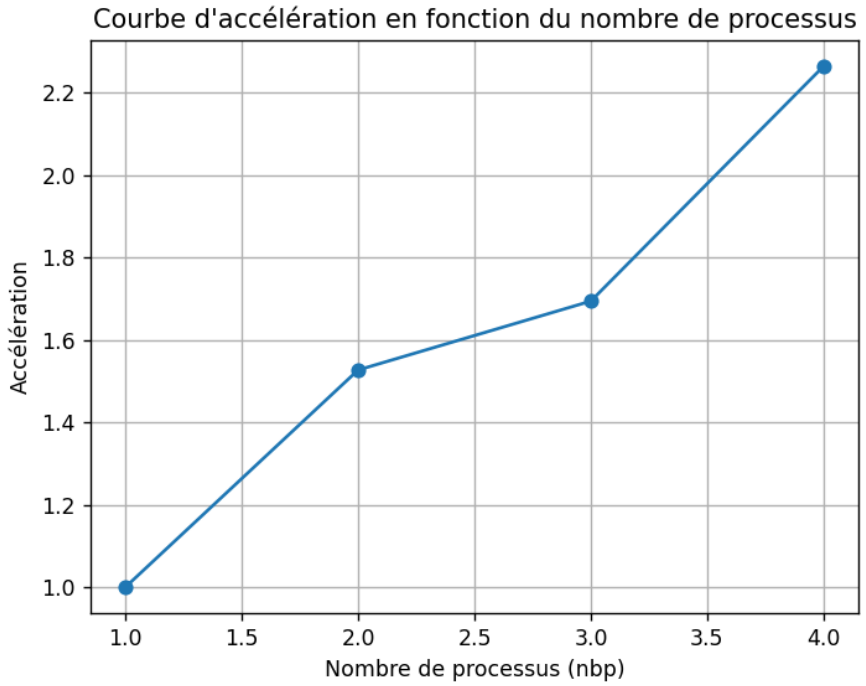
Nous constatons que le programmes fonctionne bien via la commande suivante :

```
md5sum -c perroquets.md5sum
Perroquet0001.jpg: OK
Perroquet0002.jpg: OK
Perroquet0003.jpg: OK
Perroquet0004.jpg: OK
....
```

Nous obtenons alors les résultats en temps d'exécution suivant :

Nombre de processus (nbp)	Temps d'exécution (secondes)
1	76.45
2	50.07
3	45.13
4	33.77

TABLE 2 – Temps d'exécution en fonction du nombre de processus



L'accélération augmente avec le nombre de processus, ce qui est une bonne indication que le programme bénéficie de la parallélisation. Par exemple, avec 4 processus le temps d'exécution est 2 fois plus rapide.

3 Parallélisation d'une photo en haute résolution (1)

3.1 Stratégie de parallélisation

Le processus 0 va découper l'image en lignes et va répartir ces lignes aux processus suivant. Chaque processus va effectuer les calculs de convolution sur sa section de lignes puis les envoyer au processus 0. Ce dernier va alors recevoir une par une et dans l'ordre des rangs les sections de lignes afin de les concaténer pour obtenir l'image finale. Il faudra cependant faire attention à ce que cette réception des résultats ne forme pas un goulot d'étranglement

Pour éviter de prendre trop de mémoire par processus, nous devons limiter les données manipulées : chaque processus travaille sur une partie bien définie de l'image, évitant ainsi de stocker l'image entière en mémoire.

Cette stratégie est adaptée et optimale car les tâches de calcul de convolution sont indépendantes donc on peut tout à fait diviser l'image pour lui appliquer des filtres. Cela permet une parallélisation efficace en diminuant la charge de travail.
Ensuite, en divisant l'image on limite la taille des données en mémoire de chaque processus ce qui permet d'alléger la tâche et de diminuer le temps d'exécution du programme surtout pour cette image de haute résolution.

Cette stratégie ne serait pas optimale pour le travail effectué en 2. car le problème majeur résidait dans la quantité d'images. Or, ici, on ne travaille que sur une seule image. Toutefois, pour 2., on pourrait, en plus de répartir les images aux processus, répartir en plus des sections des images pour avoir une double parallélisation en nombre et en taille d'image.

3.2 Parallélisation MPI

Voir fichier.

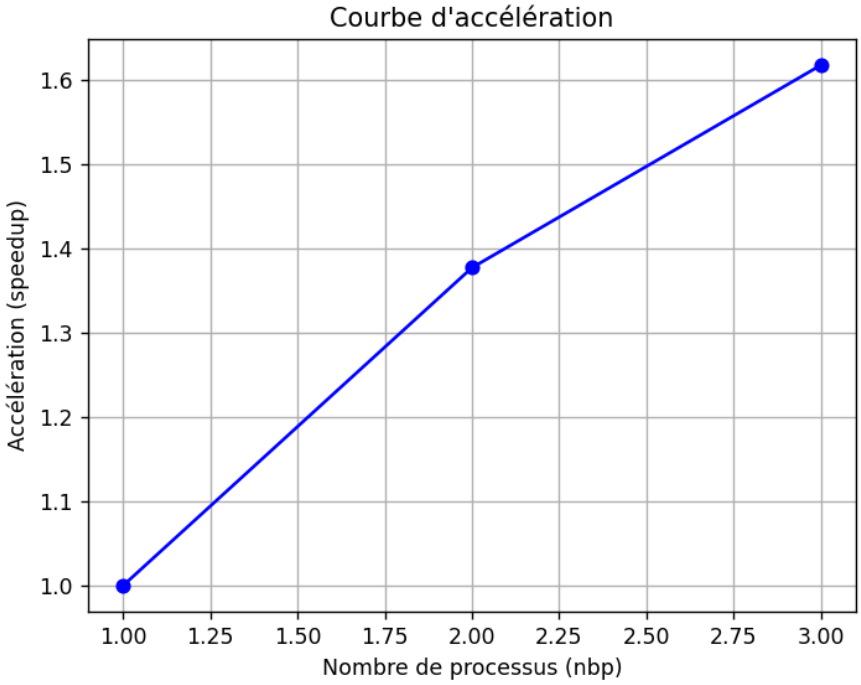
```
mpiexec -n 1 python double_size.py
```

Malheureusement, l'image est trop grosse et mon PC crash...
On réduit alors la taille de l'image sur python.

3.3 Résultats et accélérations

Nombre de processus (nbp)	Temps (secondes)
1	2.41
2	1.75
3	1.49
4	MemoryError

TABLE 3 – Temps d'exécution en fonction du nombre de processus



La parallélisation semble fonctionner car on est presque 2 fois plus rapide avec 3 processus.