



# Examen

**Nicolas RINCON VIJA**

nicolas.rincon@ensta.fr

École Nationale Supérieure de Techniques Avancées - ENSTA  
Systèmes parallèles et distribués - OS02  
Palaiseau, France

ANNÉE ACADEMIQUE 2024-2025

---

## Table des matières

<b>1 Environnement de calcul</b>	<b>3</b>
<b>2 Parallélisation d'images issues d'une vidéo</b>	<b>3</b>
2.1 Stratégie de parallélisation et justification . . . . .	3
2.2 Parallélisation du programme . . . . .	4
2.3 Courbe d'accélération du programme . . . . .	4
<b>3 Parallélisation d'une photo en haute résolution (1)</b>	<b>5</b>
3.1 Stratégie de parallélisation et justification . . . . .	5
3.2 Comparaison avec la parallélisation de <code>movie_filter.py</code> . . . . .	6
3.3 Parallélisation de <code>double_size.py</code> . . . . .	6
<b>4 Parallélisation d'une photo en haute résolution (2)</b>	<b>7</b>
4.1 Stratégie de parallélisation et justification . . . . .	7
4.2 Comparaison avec la parallélisation précédente . . . . .	8
4.3 Parallélisation de <code>double_size2.py</code> . . . . .	8
4.4 Courbe d'accélération du programme . . . . .	9

## Remarque préalable

Ce travail a été réalisé sur la base des résultats obtenus à partir des exécutions disponibles sur le dépôt GitHub suivant : [https://github.com/Nrinconv/OS02\\_ENSTA](https://github.com/Nrinconv/OS02_ENSTA).

Para convertir el texto a LaTeX, debemos considerar que LaTeX es un lenguaje de marcado para la creación de documentos. A continuación, te proporciono el texto convertido a LaTeX :

## 1 Environnement de calcul

`lscpu`

### Contexte d'exécution

- **Système d'exploitation** : Ubuntu (natif).
- **Runtime** : Exécution native sur le matériel, sans virtualisation intermédiaire.
- **Avantages du natif** : Meilleure performance grâce à l'absence de surcharge liée à une couche de virtualisation.

### Architecture et processeur

- **Architecture** : x86\_64 (64 bits).
- **Modèle du processeur** : Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz.
- **Cœurs et threads** : 4 cœurs physiques, 8 threads (Hyper-Threading).

### Cache

- **L1d** : 128 KiB (4 instances).
- **L1i** : 128 KiB (4 instances).
- **L2** : 1 MiB (4 instances).
- **L3** : 8 MiB (1 instance).
- **Importance** : Réduit la latence d'accès à la mémoire, crucial pour les simulations intensives en calcul.

## 2 Parallélisation d'images issues d'une vidéo

### 2.1 Stratégie de parallélisation et justification

Alors, pour la parallélisation, j'ai choisi de distribuer les images entre les processus disponibles. Comme ça, chaque processus peut traiter un petit groupe d'images tout seul, sans déranger les autres.

Je pense que c'est une bonne façon de faire pour plusieurs raisons :

- Chaque image est traitée séparément, donc les processus n'ont pas besoin de se parler trop entre eux.
- La charge de travail est répartie de manière égale entre les processus, ce qui signifie que personne n'est surchargé.
- On utilise moins de mémoire, car chaque processus ne charge que les images dont il a besoin.

En fin de compte, je pense que cette façon de paralléliser est plutôt efficace pour ce problème, parce qu'elle réduit les échanges entre les processus et qu'elle utilise bien les ressources disponibles.

## 2.2 Parallélisation du programme

La parallélisation du programme `movie_filter.py` repose sur la distribution des images entre les processus de MPI. Voici les étapes principales :

1. Le processus maître (`rank = 0`) récupère la liste de toutes les images et les divise en sous-ensembles :

```
1  if rank == 0:  
2      image_list = [f"Perroquet{i+1:04d}.jpg" for i in range(37)]  
3      chunks = [image_list[i::size] for i in range(size)]  
4  else:  
5      chunks = None
```

2. Ensuite, chaque processus reçoit un sous-ensemble d'images grâce à l'opération `scatter` :

```
1  local_images = comm.scatter(chunks, root=0)
```

3. Chaque processus applique le filtre à ses images :

```
1  for image in local_images:  
2      sharpen_image = apply_filter(path + image)  
3      print(f"image {image} processed")  
4      processed_images.append((image, sharpen_image))
```

4. Une synchronisation est effectuée avant de mesurer le temps total d'exécution :

```
1  comm.Barrier()  
2  max_time = comm.reduce(total_time, op=MPI.MAX, root=0)  
3  if rank == 0:  
4      print(f"total execution time: {max_time:.2f} seconds")
```

5. Enfin, les images modifiées sont envoyées au processus maître et enregistrées :

```
1  gathered_images = comm.gather(processed_images, root=0)  
2  
3  if rank == 0:  
4      all_images = [img for sublist in gathered_images for img in sublist]  
5      all_images.sort()  
6      for img_name, img in all_images:  
7          img.save(out_path + img_name)  
8      print("images saved")
```

Grâce à cette approche, chaque processus effectue ses calculs indépendamment et la charge de travail est équilibrée. Cela permet d'accélérer le traitement des images et d'exploiter efficacement les ressources disponibles.

## 2.3 Courbe d'accélération du programme

L'accélération du programme est calculée comme suit :

$$S = \frac{T_1}{T_p}$$

où  $T_1$  est le temps d'exécution en mode séquentiel et  $T_p$  est le temps d'exécution avec  $p$  processus.

L'accélération obtenue montre une amélioration significative des performances, atteignant un facteur de 2.24 avec 4 processus. Cependant, l'efficacité diminue avec l'augmentation du nombre de processus, ce qui peut être attribué à plusieurs facteurs. Tout d'abord, le processeur utilisé possède 4 cœurs physiques et 8 threads, mais l'Hyper-Threading n'apporte pas de gain significatif. Ensuite, la gestion du cache peut limiter la performance en raison des accès fréquents à la mémoire. Enfin, la surcharge de communication MPI devient un facteur limitant lorsque le nombre de processus augmente, réduisant ainsi l'efficacité globale. (J'ai essayé de faire la simulation avec plus de processus, mais mon ordi n'a pas tenu le coup)

Nombre de processus	Temps d'exécution (s)	Speedup
1 (Séquentiel)	84.80	1.00
2	53.68	1.58
3	40.56	2.09
4	37.81	2.24

TABLE 1 – Temps d'exécution et Speedup en fonction du nombre de processus

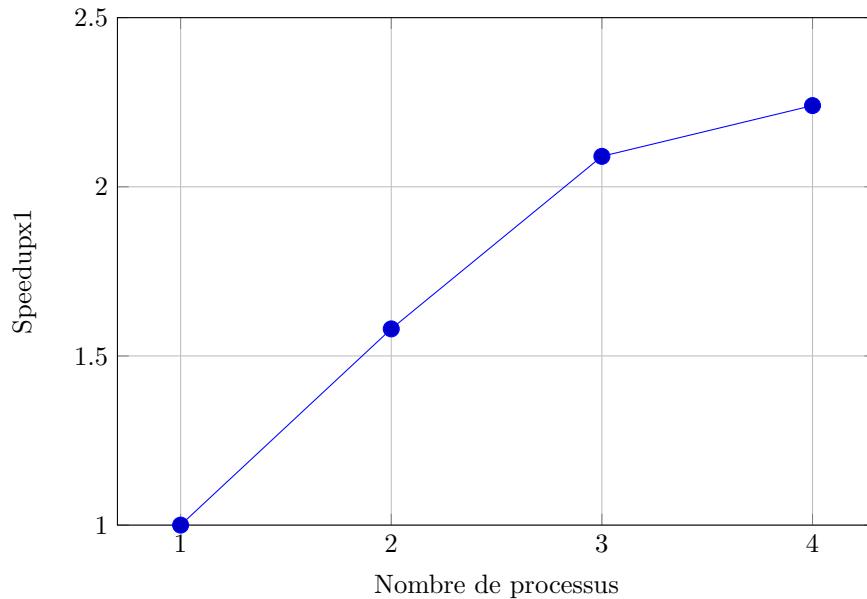


FIGURE 1 – Courbe d'accélération en fonction du nombre de processus

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

#### 3.1 Stratégie de parallélisation et justification

J'ai opté pour une stratégie de parallélisation qui consiste à diviser l'image en plusieurs sections horizontales, puis à les distribuer aux processus disponibles à l'aide de MPI. Chaque processus applique les transformations nécessaires à sa section de l'image, puis renvoie les résultats au processus principal, qui reconstruit l'image finale.

Cette approche est bien adaptée pour plusieurs raisons :

- Chaque section peut être traitée de manière indépendante, ce qui réduit considérablement la communication entre les processus.
- La charge de travail est répartie de manière égale, car chaque processus reçoit une section de taille similaire, ce qui évite les déséquilibres de charge.
- La reconstruction de l'image finale est simple et efficace, car il suffit de concaténer verticalement les sections traitées pour obtenir l'image complète.

En utilisant cette stratégie de parallélisation, nous pouvons profiter des avantages de la parallélisation, tels que la réduction du temps de traitement et l'amélioration de la scalabilité, tout en minimisant les complexités liées à la communication entre les processus.

## 3.2 Comparaison avec la parallélisation de movie\_filter.py

Dans `movie_filter.py`, j'ai appliqué une parallélisation basée sur la distribution des images. Chaque processus traitait une image complète de manière indépendante. Dans `double_size.py`, la situation est différente car je travaille sur une seule image. J'ai donc dû diviser l'image en lignes horizontales pour que plusieurs processus puissent travailler simultanément sur différentes parties.

Une difficulté spécifique est la reconstruction de l'image, qui doit être effectuée avec soin pour garantir l'ordre correct des sections. Pour cela, j'ai ajouté un mécanisme permettant de rassembler et de trier les sections avant la reconstruction.

## 3.3 Parallélisation de double\_size.py

La parallélisation de `double_size.py` suit les étapes suivantes :

```
1 # le processus ma tre charge l'image et la divise en plusieurs sections horizontales
2 sections = np.array_split(img_array, size, axis=0)
3
4 # les sections sont distribu es aux processus avec scatter
5 local_section = comm.scatter(sections, root=0)
6
7 # chaque processus applique le traitement sa section
8 processed_section = double_size(local_section)
9
10 # les sections trait es sont rassembl es et ordonn es
11 gathered_sections = comm.gather((rank, processed_section), root=0)
12 if rank == 0:
13     gathered_sections.sort()
14     final_image = np.vstack([section for _, section in gathered_sections])
```

Grâce à cette méthode, chaque processus traite une partie de l'image indépendamment, et la synchronisation est uniquement nécessaire à la fin du programme pour reconstruire l'image finale.

## Courbe d'accélération du programme

L'accélération du programme est calculée comme suit :

$$S = \frac{T_1}{T_p}$$

où  $T_1$  est le temps d'exécution en mode séquentiel et  $T_p$  est le temps d'exécution avec  $p$  processus.

Nombre de processus	Temps d'exécution (s)	Accélération
1 (Séquentiel)	10.87	1.00
2	4.90	2.22
3	3.88	2.80
4	3.61	3.01

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

L'accélération obtenue montre une nette amélioration des performances, atteignant un facteur de 3.01 avec 4 processus. Cependant, le gain diminue à mesure que le nombre de processus augmente, probablement en raison de la saturation des ressources du processeur (4 coeurs physiques, 8 threads), de l'overhead introduit par MPI et des limites du cache mémoire. Malgré cela, la parallélisation reste efficace, surtout avec 2 ou 3 processus, avant d'atteindre un plateau à 4 processus.

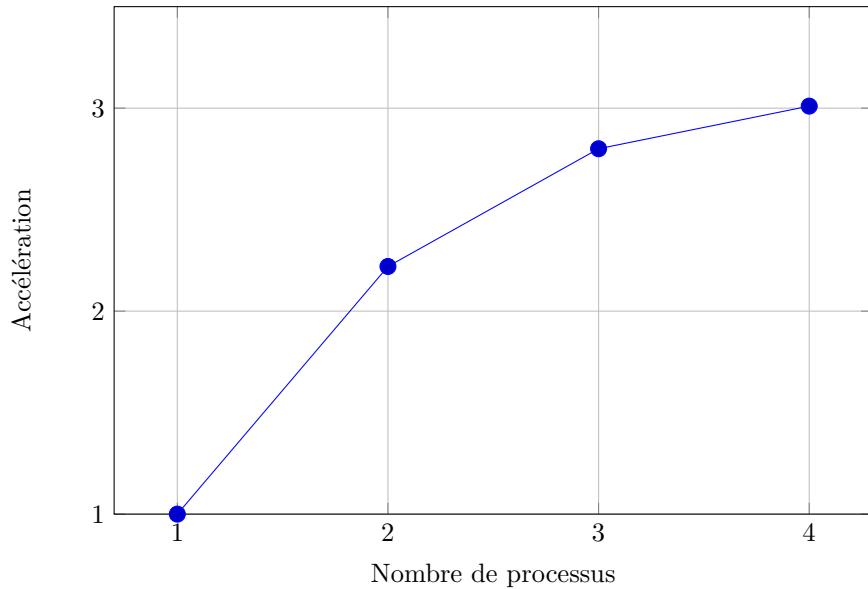


FIGURE 2 – Courbe d'accélération en fonction du nombre de processus



FIGURE 3 – Image originale



FIGURE 4 – Image agrandie (taille doublée)

## 4 Parallélisation d'une photo en haute résolution (2)

### 4.1 Stratégie de parallélisation et justification

Pour paralléliser le programme `double_size2.py`, nous avons choisi une division en blocs avec gestion des cellules fantômes. L'image est divisée en plusieurs sections horizontales, et chaque processus traite une partie de l'image avec les filtres nécessaires. Cette méthode est bien adaptée car :

- Chaque processus travaille sur une partie indépendante de l'image, minimisant ainsi la dépendance entre les processus.
- L'utilisation de cellules fantômes permet d'assurer la continuité du traitement aux bords des sections.
- La reconstruction finale est simple et efficace grâce à un `gather` et un tri des sections reçues.

Le processus maître divise l'image et l'envoie aux processus via `send/recv`. Chaque processus applique ensuite les filtres Gaussien et de netteté avant d'envoyer les résultats pour l'assemblage final.

## 4.2 Comparaison avec la parallélisation précédente

Dans la question précédente, nous avons utilisé une parallélisation en **lignes horizontales** avec MPI. Dans ce problème, nous avons amélioré l'approche en intégrant des **cellules fantômes** et une distribution plus équilibrée des données.

**Différences principales :**

- **Gestion des bords** : Ici, les cellules fantômes permettent d'éviter les artefacts aux frontières des sections, contrairement à la première approche.
- **Répartition plus équilibrée** : La distribution des blocs est plus efficace, surtout si l'image est grande et nécessite plusieurs coeurs.
- **Complexité** : Cette approche est plus complexe car elle nécessite la gestion des échanges entre processus pour assurer une continuité entre les blocs.

Par exemple, dans l'approche précédente, nous divisions directement l'image en sections :

```
1 sections = np.array_split(img_array, size, axis=0)
2 local_section = comm.scatter(sections, root=0)
```

Dans cette nouvelle approche, nous devons gérer les indices de début et de fin pour prendre en compte les cellules fantômes :

```
1 valid_start, valid_end = compute_local_indices(total_rows, size, rank)
2 ghost_top = 1 if valid_start > 0 else 0
3 ghost_bottom = 1 if valid_end < total_rows else 0
4 slice_data = img[valid_start - ghost_top : valid_end + ghost_bottom, :, :].copy()
```

Cette gestion assure que chaque section contient des informations des blocs voisins, ce qui améliore la qualité du traitement.

---

## 4.3 Parallélisation de double\_size2.py

Les étapes de la parallélisation sont les suivantes :

1. Le processus maître charge l'image et la divise en sections avec cellules fantômes :

```
1 slices.append((slice_data, ghost_top, ghost_bottom, valid_rows, valid_start))
```

2. Les sections sont envoyées aux processus via send/recv :

```
1 if rank == 0:
2     for r in range(1, size):
3         comm.send(slices[r], dest=r, tag=77)
4     else:
5         local_data = comm.recv(source=0, tag=77)
```

3. Chaque processus applique les filtres à sa section :

```
1 local_gauss = apply_gaussian_filter_local(local_slice)
2 local_valid = local_gauss[ghost_top : ghost_top + valid_rows, :, :]
3 local_valid[:, :, 2] = apply_sharpen_filter_local(local_valid[:, :, 2])
```

4. Les sections traitées sont rassemblées et triées pour reconstruire l'image finale :

```
1 gathered.sort(key=lambda x: x[0])
2 processed_slices = [slice_valid for (start, slice_valid) in gathered]
3 full_img = np.vstack(processed_slices)
```

Cette approche permet une parallélisation plus efficace et une meilleure gestion des transitions entre les blocs.

#### 4.4 Courbe d'accélération du programme

L'accélération du programme est calculée comme suit :

$$S = \frac{T_1}{T_p}$$

où  $T_1$  est le temps d'exécution en mode séquentiel et  $T_p$  est le temps d'exécution avec  $p$  processus.

Nombre de processus	Temps d'exécution (s)	Accélération
1 (Séquentiel)	7.52	1.00
2	4.95	1.52
3	3.84	1.96
4	4.81	1.56

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

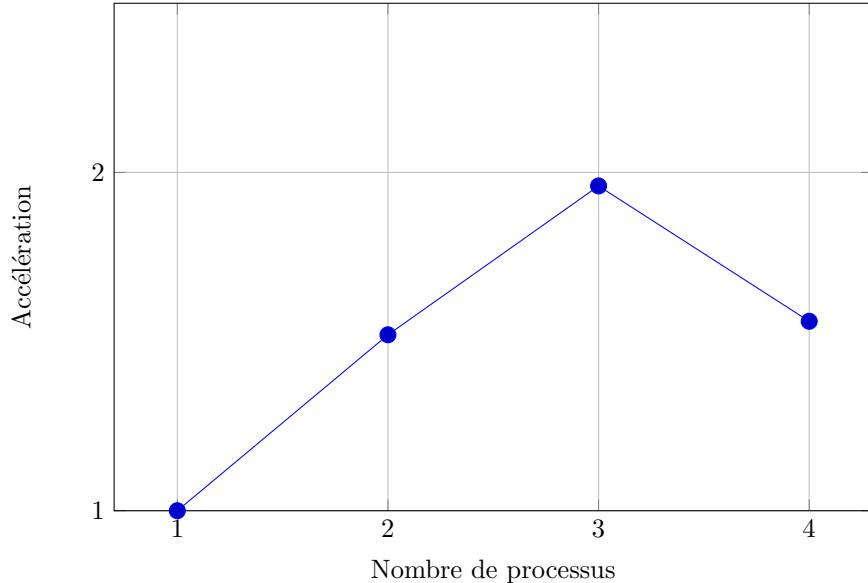


FIGURE 5 – Courbe d'accélération en fonction du nombre de processus

L'accélération augmente jusqu'à 3 processus ( $S = 1.96$ ), mais diminue légèrement avec 4 processus ( $S = 1.56$ ). Cette réduction peut s'expliquer par plusieurs facteurs. D'abord, la saturation des ressources, car le processeur possède 4 coeurs physiques et l'ajout d'un quatrième processus entraîne une concurrence accrue. Ensuite, l'overhead MPI introduit une surcharge dans la gestion des communications, ce qui réduit le gain de performance. Enfin, l'accès mémoire devient un facteur limitant lorsque plusieurs processus sollicitent simultanément le cache processeur (L1 : 128 KiB, L2 : 1 MiB, L3 : 8 MiB). Ainsi, bien que la parallélisation améliore le temps d'exécution, elle atteint une limite où les gains deviennent négligeables, voire contre-productifs.