2024/25

Nom : DADA SIMEU CÉDRIC DAREL

Email : cedric-darel.dada@ensta-paris.fr

Titre : Compte rendu TP2

STIC

ENSTA Paris, Institut Polytechnique de Paris

Table des matières

Table des figures

1 Architecture de l'ordinateur

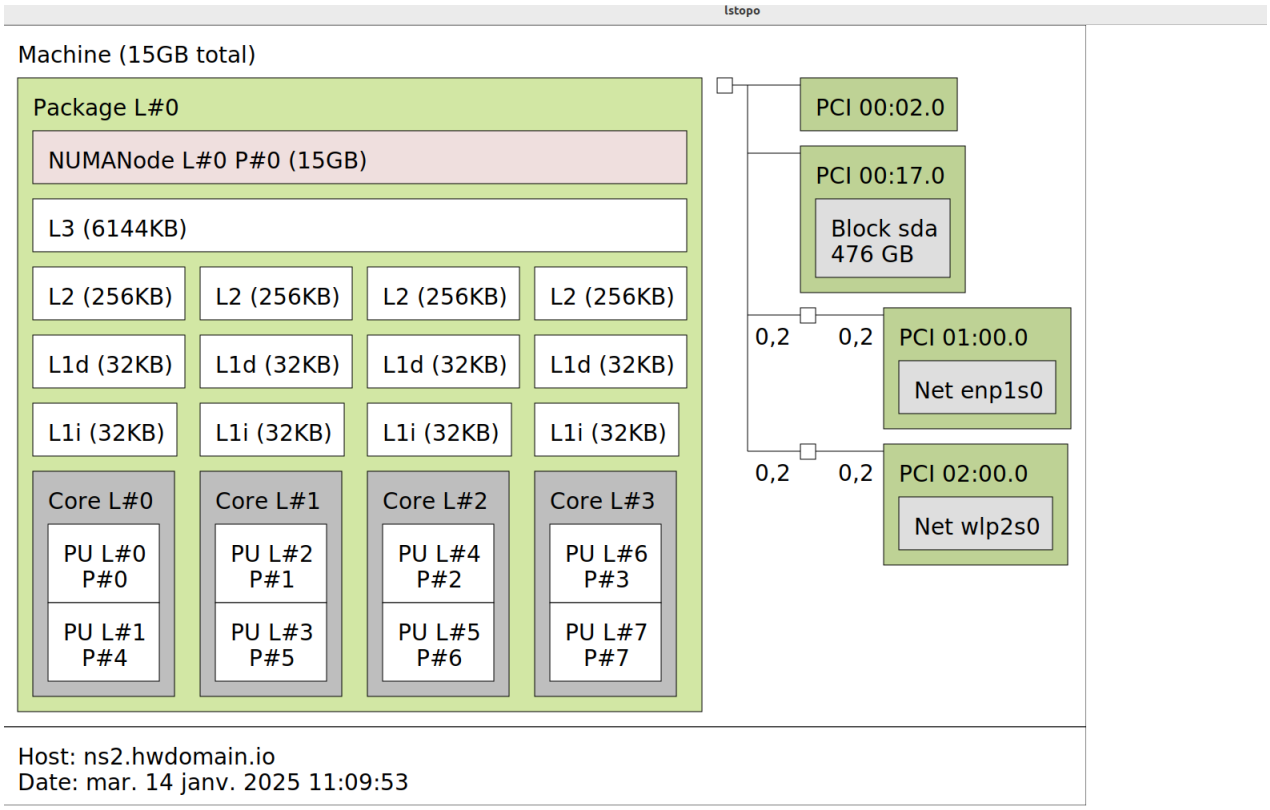

FIGURE 1 – Résultat de la commande lstopo : Nous pouvons visualiser les tailles des caches

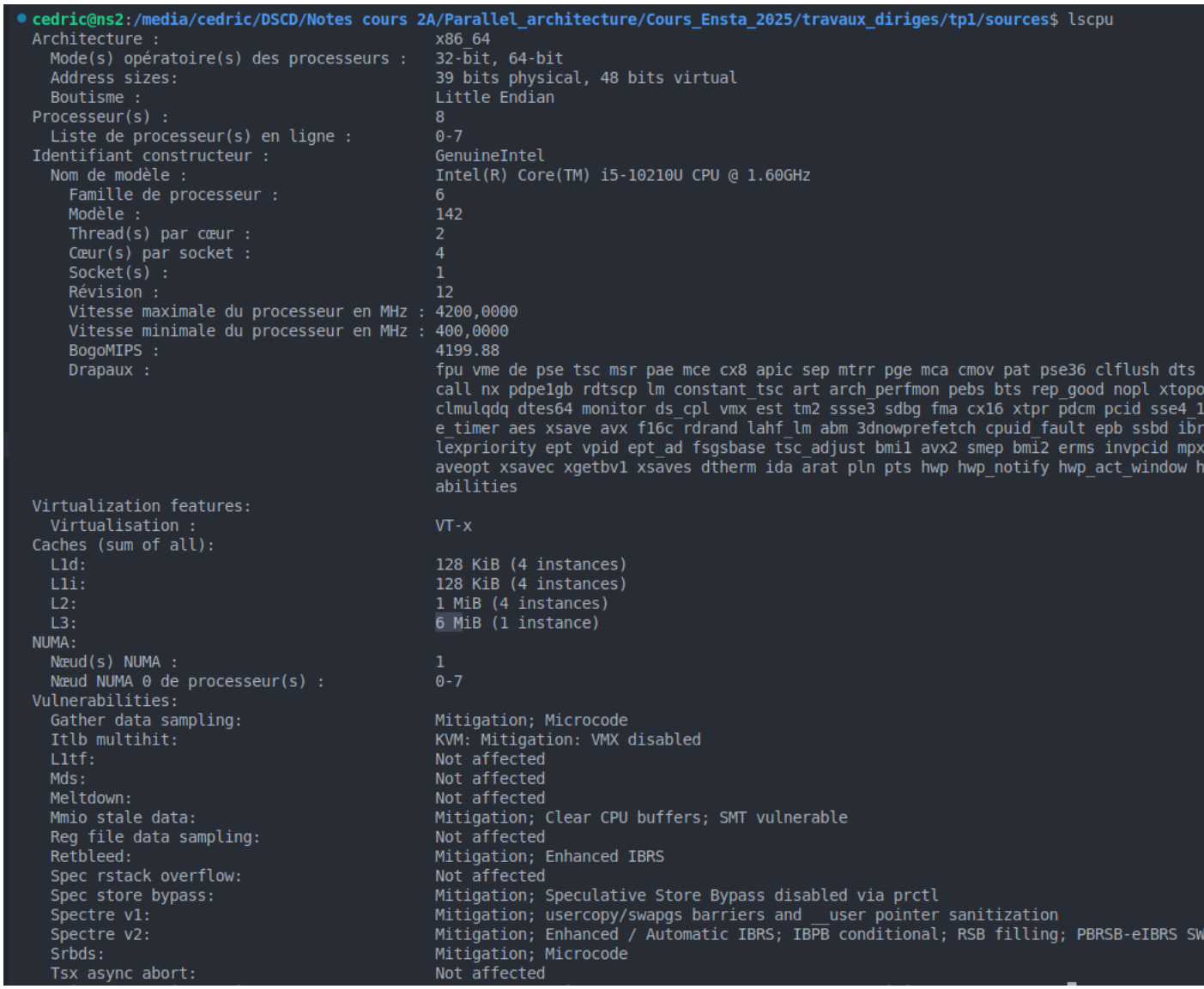

FIGURE 2 – Résultat de la commande lscpu

2 Parallélisation ensemble de Mandelbrot

2.1 Répartition par lignes de blocs

Notre objectif est de répartir équitablement les lignes d'une image de hauteur height entre nbp processus, en gérant le cas où height n'est pas divisible par nbp.

```
chunk = height // nbp       # Lignes de base par processus
rest = height % nbp         # Lignes restantes a distribuer

start = rank * chunk + min(rank, rest)  # Ajoute 1 ligne supplementaire pour les premiers rest processus
end = start + chunk + (1 if rank < rest else 0)  # +1 si le processus est dans les `rest` premiers

local_height = end - start # Nombre de lignes locales
```

Listing 1 – Bloc du fichier mandelbrot_question1.py montrant comment la taille des blocs de lignes répartis

Nous avons décider de définir les tailles des blocs de lignes ainsi pour les raisons suivantes :

— Equilibrage : Les premiers rest processus ont chunk + 1 lignes, les autres chunk.

— Garantie : Aucune ligne n'est perdue, et la charge est en première approximmation(nous ne tenons pas encore compte du fait que certains points peuvent etre plus complexes que d'autres) la plus équilibrée possible.

Le code complet se trouve dans le fichier `mandelbrot_question1.py`

Analyse des résultats (n ≤ 4)

1. Speedup : Le speedup augmente avec le nombre de processus, mais reste loin de l'idéal en raison :
   — Du déséquilibrage de charge (surtout pour n=3).
   — De la surcharge de communication (visible pour n=4).
2. Déséquilibrage :
   — Nous avons évaluer le déséquilibrage max/min (entre le processus le plus gourmand en temps et le processus le moins gourmand, pour une execution donnée) en terme de ratio ($\frac{\Delta t}{t_{max}} \times 100$)
   — Le ratio culmine à 13.5% pour n=3, ce qui s'explique par une répartition non optimale des lignes complexes.

TABLE 1 – Temps de calcul par processus et déséquilibrage.

Processus	Temps max (s)	Temps min (s)	Déséquilibre (%)	Temps total (s)
1	3.214	3.214	0.0	3.214
2	1.667	1.640	1.6	1.667
3	1.272	1.100	13.5	1.272
4	1.082	1.013	6.4	1.082
5	1.620	0.726	55.2	1.620
8	1.219	0.944	22.6	1.219

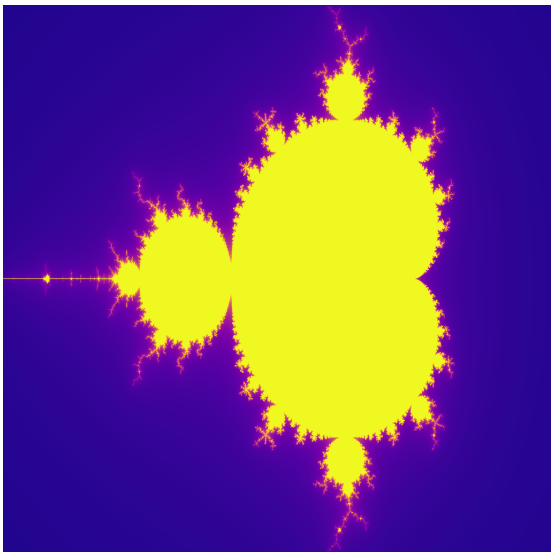

FIGURE 3 – Rendu de la fractale

TABLE 2 – Speedup par rapport au temps séquentiel (base = 1 processus).

Processus	**Temps total (s)**	**Speedup**
1	3.214	1.00
2	1.667	1.93
3	1.272	2.53
4	1.082	2.97
5	1.620	1.98
8	1.219	2.64

Analyse des résultats (n $\leq$ 8)

- Échec au préalable pour $n \geq 5$: La commande mpirun -n 5 python3 mandelbrot_question1.py provoque l'erreur "not enough slots" provient de la configuration matérielle : Le CPU possède 4 cœurs physiques (8 threads), mais Open MPI limite les slots par défaut aux cœurs physiques. **Nous avons utiliser l'option *–oversubscribe* pour contourner cette limite.**
- Speedup
  - Pour n = 5, le speedup chutte à 1.03 en raison d'un déséquilibrage extreme (55.2%)
  - Pour n = 8, le speedup remonte à 1.37, mais reste faible comparé au cas n = 4.
  - Conclusion : la scalabilité est limitée par la répartition statique et le surchage de la comunication
- Déséquilibrage
  - n = 5 montre un ratio de 55.2%, ce qui indique une mauvaise répartition des lignes complexes
  - n = 8 réduit le ratio à 22.6%, mais cela reste élevé pour une parallélisation efficace.
- Impact de –oversubscribe :
  - Permet d'exécuter plus de processus que de coeurs physiques, maiss entraine une contention de ressources (CPU, mémoire), par exemple pour n = 8, le CPU (4 coeurs / 8 threads) est sursouscrit, ce qui dégrade les performances.

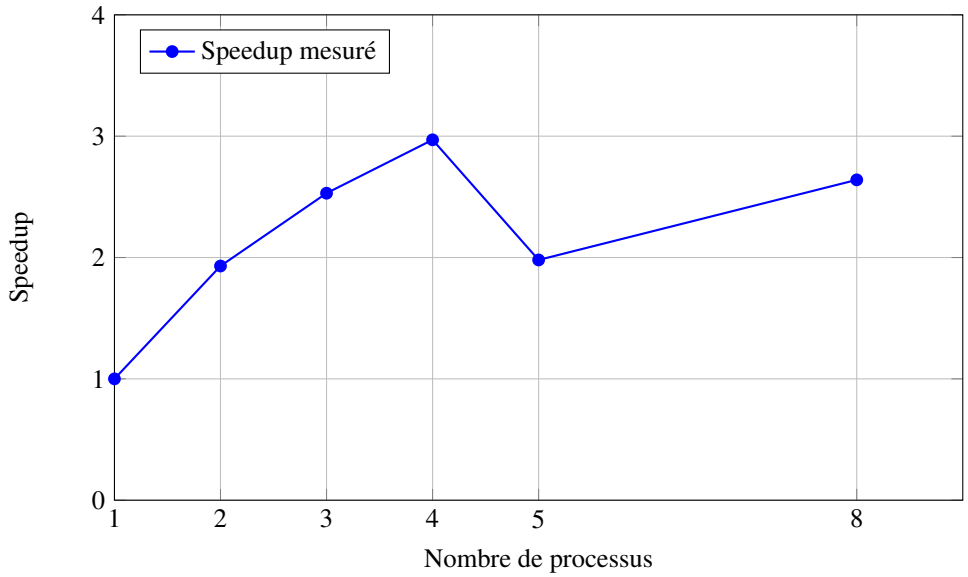

FIGURE 4 – Speedup mesuré (base = 1 processus).

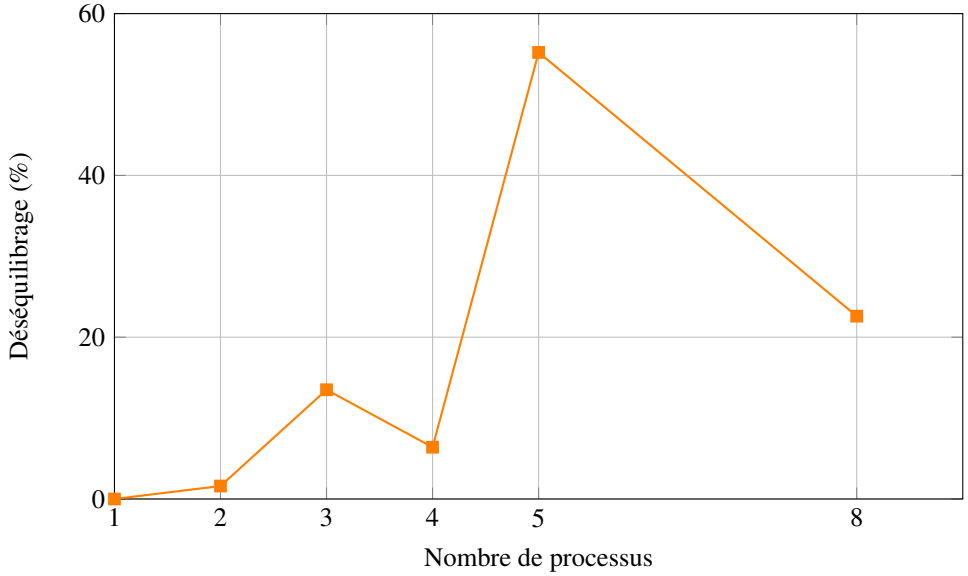

FIGURE 5 – Déséquilibrage en fonction du nombre de processus.

2.2 Meilleure répartition statique des lignes

Démarche : Équilibrer la charge en répartissant les lignes de manière cyclique (round-robin) pour atténuer le déséquilibrage causé par les zones complexes de l'image.

Répartition : Chaque processus reçoit des lignes éparpillées (exemple : pour 4 processus, les lignes 0,4,8,...vont au processus 0) Code modifié :

```
# Repartition cyclique des lignes
local_rows = [y for y in range(height) if y % nbp == rank]
local_height = len(local_rows)
convergence_local = np.empty((local_height, width), dtype=np.double)

# Calcul des lignes attribuees au processus
deb_local = time()
for y_local, y_global in enumerate(local_rows):
    for x in range(width):
        c = complex(-2. + scaleX * x, -1.125 + scaleY * y_global)
        convergence_local[y_local, x] = mandelbrot_set.convergence(c, smooth=True)
fin_local = time()
duration_local = fin_local - deb_local

# Rassemblement avec MPI_Gatherv
sendbuf = convergence_local.ravel()
local_heights = comm.gather(local_height, root=0)

if rank == 0:
    recv_counts = [h * width for h in local_heights]
    displacements = np.insert(np.cumsum(recv_counts[:-1]), 0, 0)
    convergence = np.empty((height, width), dtype=np.double)
    recvbuf = [convergence.ravel(), recv_counts, displacements, MPI.DOUBLE]
else:
    recvbuf = None

comm.Gatherv(sendbuf=sendbuf, recvbuf=recvbuf, root=0)
```

TABLE 3 – Temps de calcul et déséquilibre (code optimisé en round robin).

Nombre de processus	Temps max (s)	Temps min (s)	Déséquilibre (%)	Temps parallèle (s)
1	3.214	3.214	0.0	3.214
2	1.645	1.639	0.4	1.645
3	1.263	1.230	2.6	1.263
4	1.039	0.983	5.4	1.039
5	1.569	0.899	42.7	1.569
8	1.121	1.045	6.8	1.121

Analyse des résultats Nous voyons que le déséquilibrage de charge a été réduit significativement pour 2,3, 4, 5, 8 processus (voir Table 3). Ainsi notre approche semble optimiser la répartition des charges.

TABLE 4 – Speedup obtenu (base = exécution séquentielle à 3,214 s).

Nombre de processus	Temps parallèle (s)	Speedup
1	3.214	1.00
2	1.645	1.96
3	1.263	2.55
4	1.039	3.10
5	1.569	2.05
8	1.121	2.87

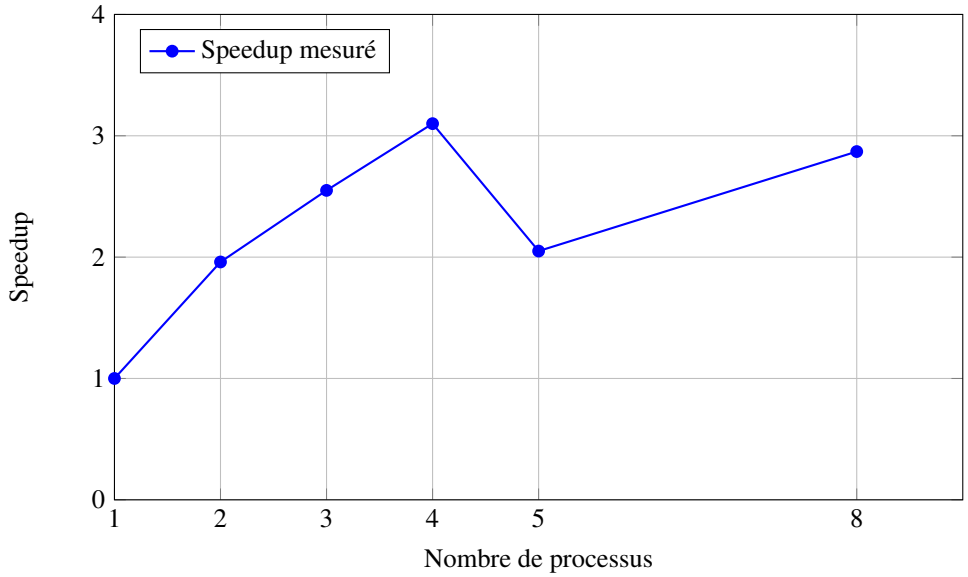

FIGURE 6 – Speedup mesuré (base = 1 processus, temps séquentiel = 3,214 s).

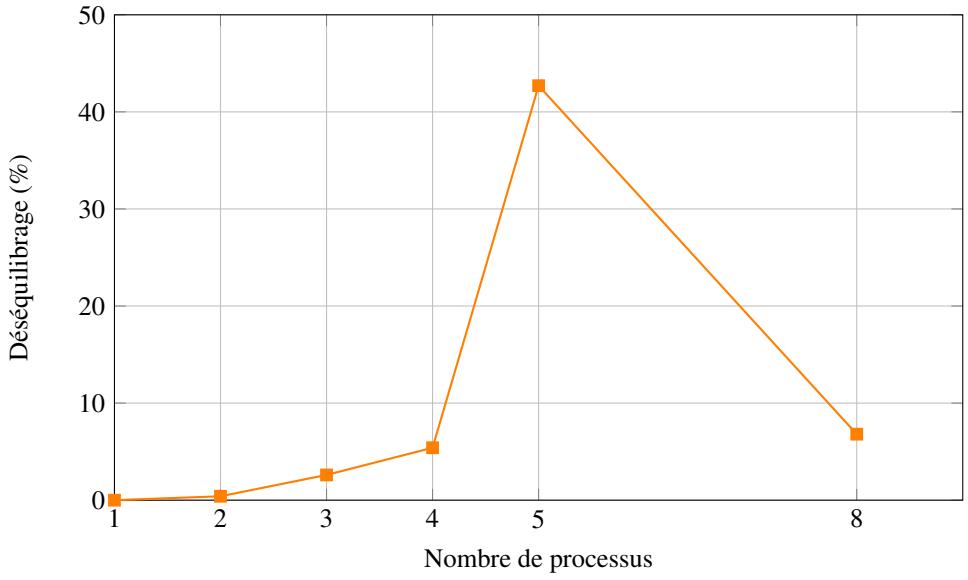

FIGURE 7 – Déséquilibrage en fonction du nombre de processus (code optimisé en round robin).

2.3 Maitre esclave

TABLE 5 – Temps de calcul et déséquilibre (code optimisé en round robin).

Nombre de processus	Temps max (s)	Temps min (s)	Déséquilibre (%)	Temps parallèle (s)
1	3.261	3.261	0.0	3.261
2	1.758	1.756	0.1	1.758
3	1.316	1.314	0.2	1.316
4	1.348	1.343	0.4	1.348
8	1.250	1.233	1.4	1.250

TABLE 6 – Speedup obtenu pour le maitre esclave (base = exécution séquentielle à 3,261 s).

Nombre de processus	Temps parallèle (s)	Speedup
1	3.261	1.00
2	1.758	1.86
3	1.316	2.48
4	1.348	2.42
8	1.250	2.61

Observation

— La stratégie maître-esclave minimise le déséquilibre de charge, quel que soit le nombre de processus.

— Même avec un nombre de processus inadapté (comme 5), le déséquilibre reste faible (1.5%).

— Le temps total est globalement réduit par rapport aux autres méthodes, car tous les processus sont pleinement utilisés.

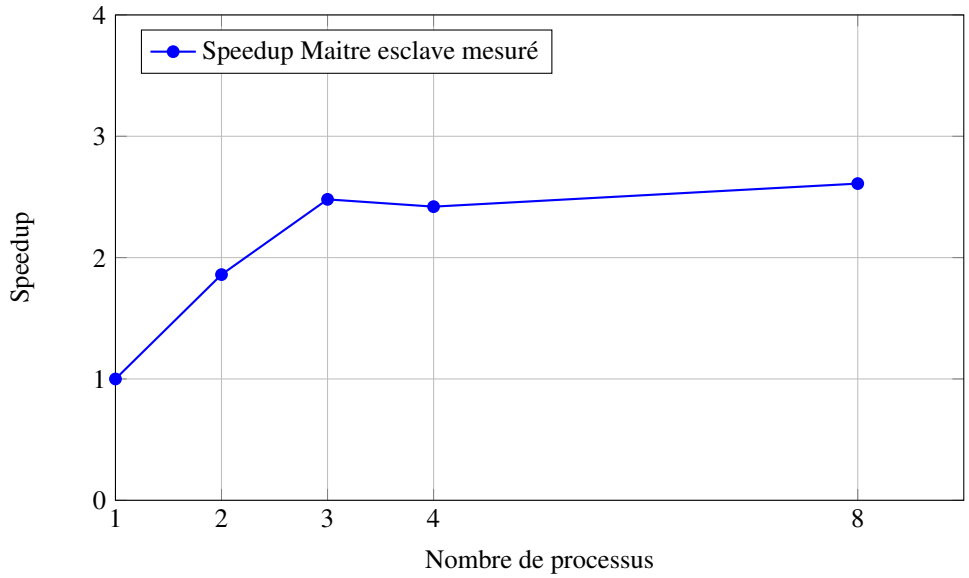

FIGURE 8 – Speedup mesuré (base = 1 processus, temps séquentiel = 3,261 s).

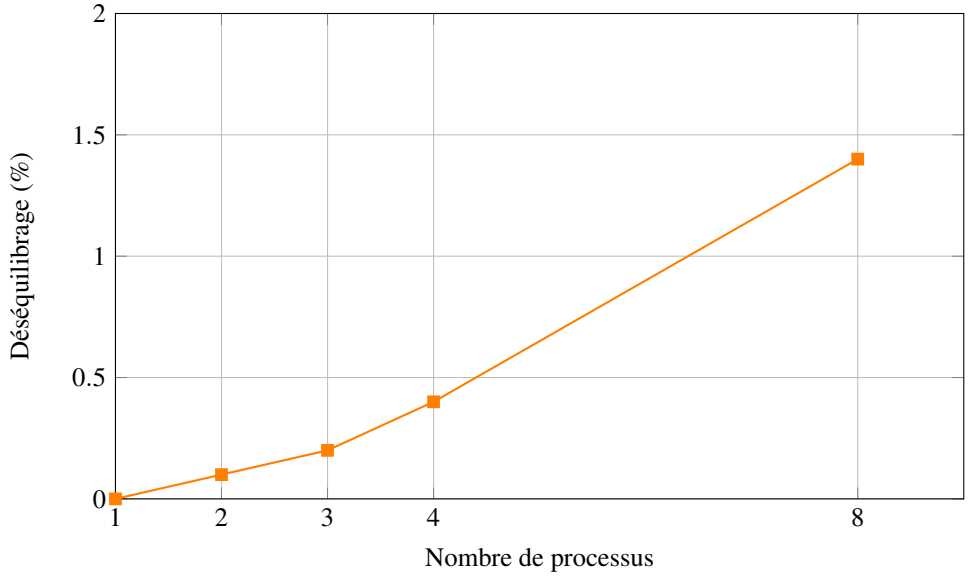

FIGURE 9 – Déséquilibrage en fonction du nombre de processus (code optimisé en round robin).

Comparaison des approches fig (11, 10)

1. Déséquilibrage des charges
   - Répartition Équitable (Q1.1) :
     - Bonne performance pour des nombres de processus bien adaptés à la taille de l'image.
     - Problèmes de déséquilibre importants avec des nombres de processus inadaptés (ex. 5 processus).
   - Round Robin Optimisé (Q1.2)
     - Améliore l'équilibrage de charge par rapport à la répartition équitable.
     - Toujours des problèmes de déséquilibre avec des nombres de processus inadaptés (ex. 5 processus).
   - Maître-Esclave (Q1.3) :
     - Meilleure répartition dynamique, minimisant le déséquilibre quel que soit le nombre de processus.
     - Le speedup reste légèrement inférieur à celui obtenu pour les autres approches. Nous justifions cela par le fait qu'un des processus est choisi comme maitre et ne réalise pas de calculs. Ainsi à nombres de processus égaux, cette approche présente un speedup moins important.

Ainsi, La stratégie maître-esclave (Q1.3) est globalement la meilleure option pour ce problème.

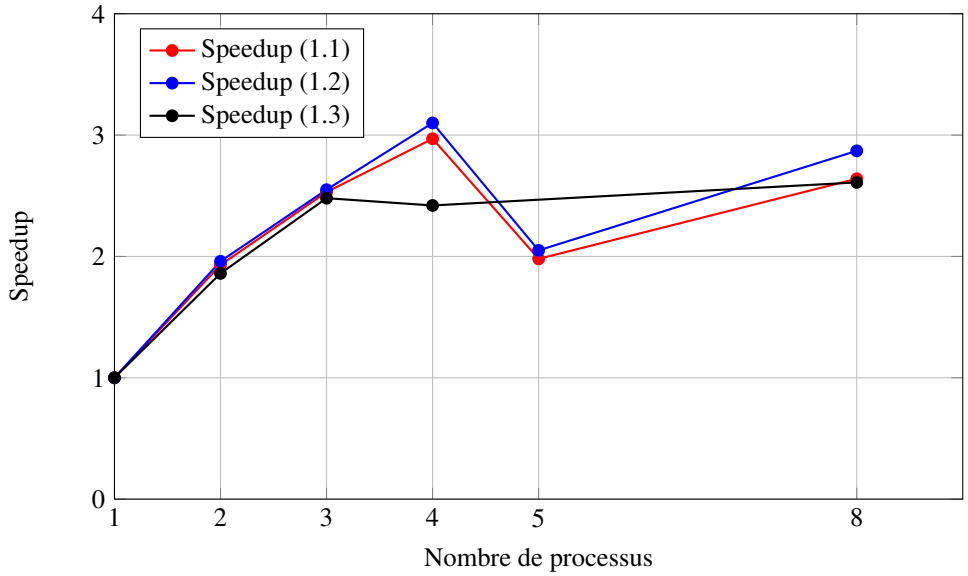

FIGURE 10 – Comparaison des différents speedup

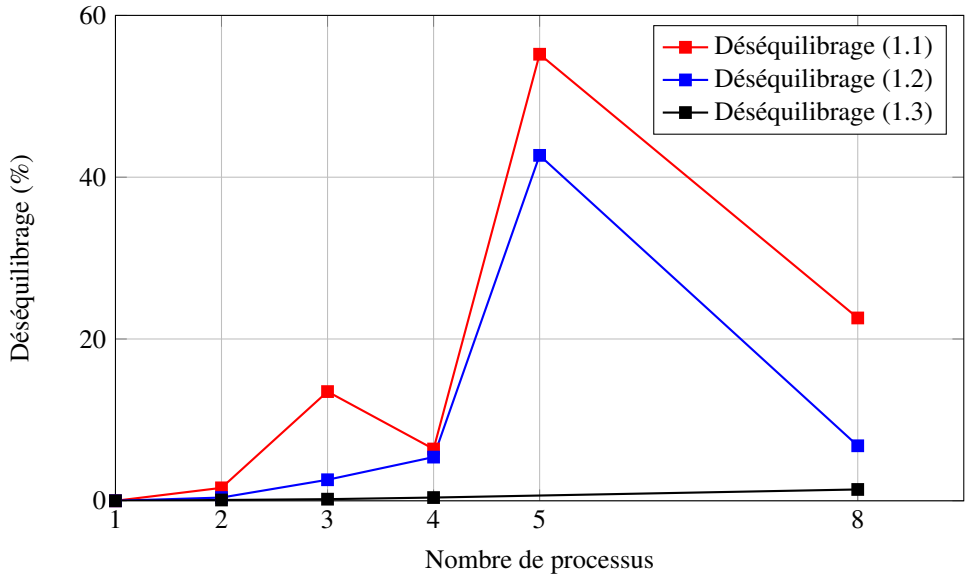

FIGURE 11 – Comparaison des différents Déséquilibrages en fonction du nombre de processus.

3 Produit matrice-vecteur

3.1 Produit matrice-vecteur colonnes

TABLE 7 – Temps de calcul et speedup pour le produit matrice–vecteur par colonnes

Nombre de processus	Temps (sec)	Speedup
1	0.005042	1.00
2	0.002711	1.86
3	0.009866	0.51
4	0.025791	0.20
5	0.049172	0.10
8	0.101329	0.05

TABLE 8 – Temps de calcul et speedup pour le produit matrice–vecteur par lignes

Nombre de processus	Temps (sec)	Speedup
1	0.000029	1.00
2	0.000017	1.71
3	0.000012	2.42
4	0.000015	1.93
5	0.000016	1.81
8	0.000015	1.93

3.2 Analyse des résultats

Produit matrice–vecteur par colonnes

Temps de calcul et répartition de la charge : Les mesures montrent que, pour le découpage par colonnes, les temps locaux varient fortement d'un processus à l'autre. Par exemple, avec 3 processus, le temps mesuré varie de 0.00987 s (processus 0) à 0.00196 s (processus 2), et avec 8 processus, le temps maximum atteint 0.10133 s alors que certains processus s'exécutent en moins de 0.001 s. Cette hétérogénéité indique un déséquilibre de charge, suggérant que certains processus effectuent plus de travail ou sont impactés par une surcharge de communication et d'accès mémoire.

Speedup obtenu : Le speedup est de 1.86 avec 2 processus, mais pour 3, 4, 5 et 8 processus, le speedup chute respectivement à 0.51, 0.20, 0.10 et 0.05. Un speedup inférieur à 1 pour un nombre de processus supérieur à 2 montre que la parallélisation, dans ce cas, dégrade les performances. Cela est très probablement dû à un coût de communication élevé (notamment lors de la réduction via `Allreduce`) et à un déséquilibre dans la répartition des colonnes entre les processus.

Produit matrice–vecteur par lignes

Temps de calcul et homogénéité du travail : Pour le découpage par lignes, chaque processus traite un bloc de lignes complet. Les temps de calcul locaux sont très faibles et quasiment homogènes (de l'ordre de 10^{-5} s) pour

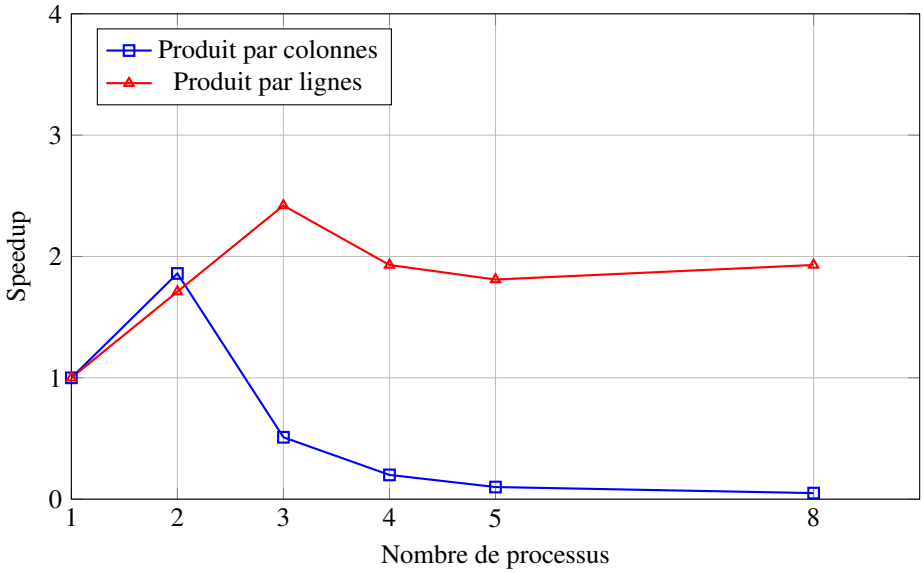

FIGURE 12 – Speedup en fonction du nombre de processus pour les deux approches.

tous les processus, indiquant une bonne répartition de la charge.

Speedup obtenu : Le speedup pour l'approche par lignes est nettement supérieur. On observe un speedup de 1.71 avec 2 processus et un pic à 2.42 avec 3 processus. Pour 4, 5 et 8 processus, le speedup se stabilise autour de 1.8 à 1.93. La saturation du speedup s'explique par la taille réduite du problème (dim = 120), où les coûts de communication et de synchronisation deviennent significatifs par rapport au temps de calcul effectif.

Comparaison des deux approches

Répartition de la charge : L'approche par lignes offre une répartition plus équilibrée du travail, avec des temps locaux très homogènes, tandis que l'approche par colonnes présente un déséquilibre important, affectant négativement la performance globale.

Scalabilité : Bien que l'approche par lignes montre un meilleur speedup, la taille de la matrice (dim = 120) ne permet pas d'exploiter pleinement le parallélisme pour un nombre élevé de processus. Pour des problèmes de plus grande envergure, les avantages de l'approche par lignes pourraient être encore plus prononcés.

Conclusion générale : Pour ce problème, l'approche par lignes se révèle préférable, grâce à une répartition de charge plus équilibrée et à un speedup global meilleur. En revanche, l'approche par colonnes, en raison de sa répartition inégale et des coûts de communication élevés, entraîne une dégradation des performances dès l'utilisation de plus de 2 processus.

4 Entraînement pour l'examen écrit

4.1 Application de la loi d'Amdahl

La loi d'Amdahl s'énonce par :

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}},$$

où p représente la fraction du temps pouvant être parallélisée et n le nombre de nœuds.

Ici, la partie parallèle représente 90% du temps d'exécution en séquentiel, donc :

$$p = 0.9 \quad \text{et} \quad 1 - p = 0.1.$$

Pour $n \gg 1$, le terme $\frac{p}{n}$ tend vers zéro et l'accélération maximale théorique est :

$$S_{\max} = \lim_{n \to \infty} S(n) = \frac{1}{1-p} = \frac{1}{0.1} = 10.$$

Ainsi, la vitesse d'accélération maximale théorique qu'Alice pourrait obtenir est 10.

Choix raisonnable du nombre de nœuds

En pratique, Alice observe une accélération maximale de 4 pour son jeu de données. Cela indique que, malgré une capacité théorique d'un speedup de 10, des facteurs tels que le surcoût de communication, la synchronisation ou d'autres inefficacités limitent l'amélioration réelle.

Il n'est donc pas pertinent d'augmenter indéfiniment le nombre de nœuds, car au-delà d'un certain seuil, le gain supplémentaire est négligeable. Dans ce cas, utiliser environ 4 à 8 nœuds semble raisonnable, car :

— Avec trop peu de nœuds, on n'exploite pas suffisamment la parallélisation.

— Avec trop de nœuds, l'overhead (communication, synchronisation) devient prépondérant, et on risque de gaspiller des ressources CPU.

4.2 Application de la loi de Gustafson

La loi de Gustafson prend en compte l'évolutivité du problème lorsque l'on augmente le nombre de nœuds. Elle s'énonce sous la forme :

$$S(n) = n - s\,(n-1),$$

où s est la fraction de temps séquentiel pour la charge de travail mise à l'échelle.

Dans le problème initial, on avait $s = 1 - p = 0.1$. Lorsque la quantité de données est doublée, et en supposant que la partie parallèle (qui était de 90% du temps) évolue linéairement, le temps de calcul parallèle double alors que le temps séquentiel reste inchangé. Pour le problème étendu :

$$T_{\text{séquentiel}} = 0.1\,T \quad \text{et} \quad T_{\text{parallèle}} = 2 \times 0.9\,T = 1.8\,T.$$

La fraction séquentielle pour le problème mis à l'échelle devient :

$$s' = \frac{T_{\text{séquentiel}}}{T_{\text{séquentiel}} + T_{\text{parallèle}}} = \frac{0.1}{0.1 + 1.8} = \frac{0.1}{1.9} \approx 0.0526.$$

Pour un grand nombre de nœuds, la vitesse d'accélération selon Gustafson s'exprime alors théoriquement (pour $n \gg 1$) de manière asymptotique comme :

$$S_{\max} \approx \frac{1}{s'} \approx \frac{1}{0.0526} \approx 19.$$

Ainsi, en doublant la quantité de données à traiter, Alice peut espérer une accélération maximale théorique d'environ 19.

4.3 Conclusion

— **Loi d'Amdahl :** Avec 90% du code parallélisable, l'accélération maximale théorique est de 10 pour $n \gg 1$. Toutefois, en pratique, Alice atteint une accélération de 4, ce qui suggère que l'augmentation du nombre de nœuds au-delà d'un certain seuil (environ 4 à 8 nœuds) ne serait pas rentable.

— **Loi de Gustafson :** En doublant la quantité de données, la fraction séquentielle se réduit à environ 5,26%, ce qui permet théoriquement d'atteindre un speedup maximal d'environ 19.

Ces résultats montrent que, pour le jeu de données initial, il vaut mieux ne pas utiliser un nombre excessif de nœuds pour éviter le gaspillage de ressources. En revanche, pour des problèmes de plus grande envergure (données doublées), la parallélisation peut être beaucoup plus efficace.