Rapport TP n°2

Systèmes Parallèles & Distribués

Rowan HOUPA

Janvier 2025

Table des matières

Partie 1 : Parallélisation ensemble de Mandelbrot

Configuration des tests :

- Image : 1024×1024 pixels
- Itérations maximales : $N_{\max} = 1000$
- Rayon d'échappement : $R = 2$
- Machine : 4 cœurs disponibles

1 Question 1 : Partition équitable par blocs de lignes

1.1 Principe

La première approche consiste à diviser l'image en blocs contigus de lignes. Chaque processus p (sur P processus) calcule les lignes de l'indice $\lfloor p \cdot H/P \rfloor$ à $\lfloor (p+1) \cdot H/P \rfloor - 1$. Le processus 0 rassemble ensuite l'image complète via `MPI_Gatherv`.

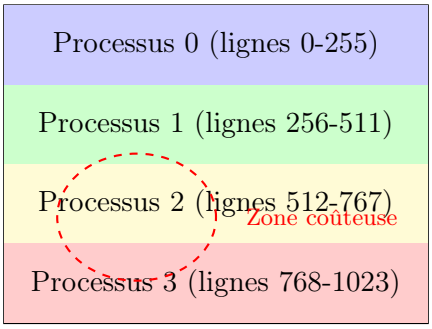

FIGURE 1 – Partition par blocs – Les processus 1 et 2 héritent de la zone coûteuse

1.2 Résultats expérimentaux

TABLE 1 – Performances de la partition par blocs de lignes

Processus	Temps total (s)	Temps calcul (s)	Speedup	Efficacité (%)
1	15.1558	15.0694	1.00	100.0
2	7.6496	7.6453	1.98	99.0
4	4.8157	4.8113	3.15	78.75
8	4.7873	4.7820	3.17	39.63

1.2.1 Temps de calcul par processus (4 processus)

1.3 Courbes de scaling

1.4 Interprétation des résultats

L'efficacité chute rapidement (39.63% à 8 processus) pour les raisons suivantes :

1. **Déséquilibre de charge** : L'écart-type des temps de calcul ($\sigma = 0.4228$ s) est élevé. Les processus 1 et 2 traitent la zone centrale de l'ensemble de Mandelbrot, plus coûteuse en calcul.

TABLE 2 – Temps de calcul individuel par processus (partition par blocs, 4 proc.)

Processus	**Lignes**	**Temps calcul (s)**
0	0–255	4.8113
1	256–511	3.8921
2	512–767	3.9306
3	768–1023	4.6938
Moyenne	–	4.3319
Écart-type	–	**0.4228**

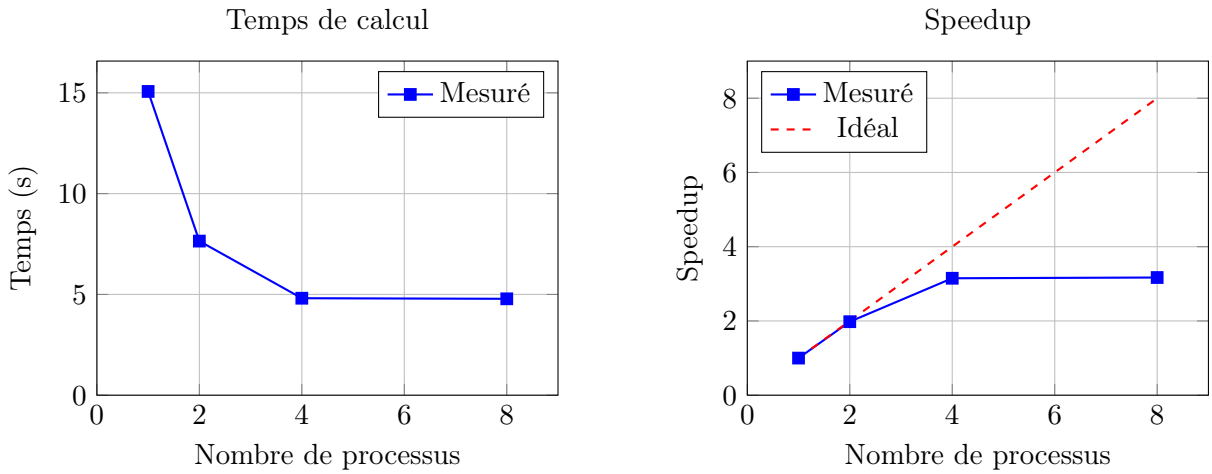

FIGURE 2 – Scaling de la partition par blocs

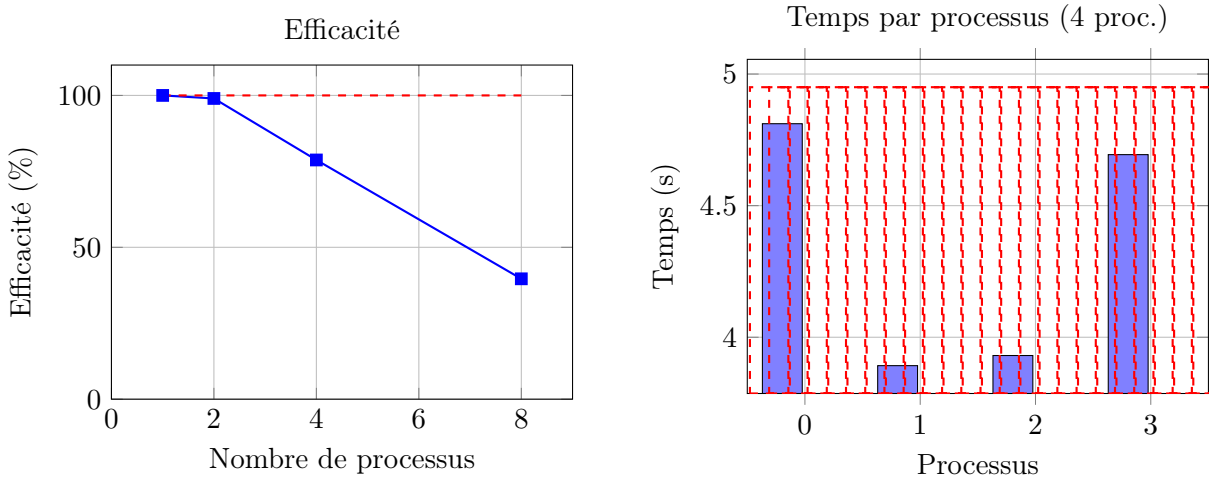

FIGURE 3 – Efficacité et distribution des temps de calcul

2. **Synchronisation imposée** : Le `MPI_Gatherv` bloque jusqu'à ce que *tous* les processus aient terminé. Le temps total est donc dicté par le processus le plus lent.
3. **Processus inactifs** : Les processus 0 et 3, qui terminent plus tôt, attendent inutilement.

Conclusion : Cette stratégie simple est inefficace pour des problèmes à charge hétérogène comme Mandelbrot.

2 Question 2 : Répartition entrelacée (cyclique)

2.1 Principe et motivation

Pour mieux équilibrer la charge, on distribue les lignes de manière **cyclique** :
— Processus 0 : lignes 0, 4, 8, 12, ...
— Processus 1 : lignes 1, 5, 9, 13, ...
— Processus 2 : lignes 2, 6, 10, 14, ...
— Processus 3 : lignes 3, 7, 11, 15, ...
Cette distribution mélange statistiquement les lignes "faciles" (loin de la frontière) et "difficiles" (proches de la frontière) sur chaque processus.

FIGURE 4 – Répartition entrelacée des lignes (cyclique)

2.2 Résultats expérimentaux

TABLE 3 – Performances de la répartition entrelacée

Processus	Temps total (s)	Temps calcul (s)	Speedup	Efficacité (%)
1	6.3274	6.2547	1.00	100.0
2	3.3391	3.3355	1.89	94.5
4	1.9141	1.9105	3.31	82.75
8	1.2008	1.1932	5.27	65.88

2.2.1 Temps de calcul par processus (4 processus)

2.3 Courbes de scaling et comparaison

2.4 Analyse comparative

2.5 Problèmes potentiels de cette stratégie

Malgré ses bonnes performances, la répartition entrelacée présente des limitations :

1. **Surcharge de communication** : Comme chaque processus doit envoyer et recevoir des données à des positions non-contiguës en mémoire, le `MPI_Gatherv` peut être moins efficace. Cela peut entraîner une surcharge de communication, surtout si le nombre de processus est élevé.

TABLE 4 – Temps de calcul individuel (répartition entrelacée, 4 proc.)

Processus	**Lignes**	**Temps calcul (s)**
0	0, 4, 8, ...	1.7951
1	1, 5, 9, ...	1.7998
2	2, 6, 10, ...	1.8257
3	3, 7, 11, ...	1.9104
Moyenne	–	1.8328
Écart-type	–	**0.0463**

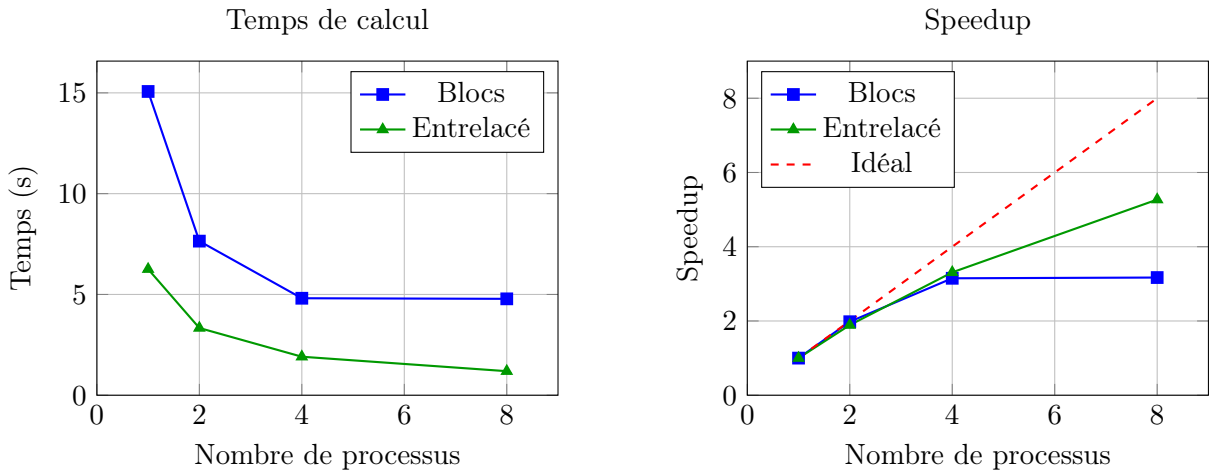

FIGURE 5 – Comparaison blocs vs entrelacé

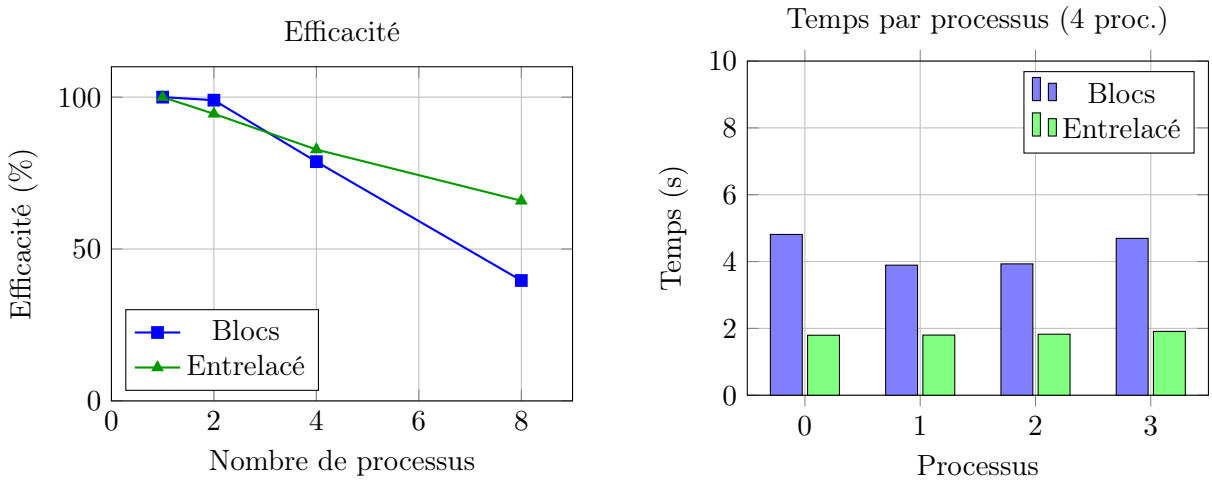

FIGURE 6 – Efficacité et équilibrage de charge

TABLE 5 – Comparaison de l'équilibrage de charge (4 processus)

Métrique	**Blocs**	**Entrelacé**
Temps moyen par processus (s)	4.3319	1.8328
Écart-type (s)	0.4228	0.0463
Speedup (4 proc.)	3.15	3.31
Efficacité (4 proc.)	78.75%	82.75%

2. **Latence accrue** : La communication non-contiguë entre les processus peut introduire une latence supplémentaire, ce qui peut être problématique pour les applications sensibles au temps.
3. **Scalabilité limitée** : Avec un très grand nombre de processus, le grain de calcul (une ligne) peut devenir trop fin, augmentant le ratio communication/calcul.
4. **Pas d'adaptation dynamique** : La répartition est fixée au départ. Si un processus est ralenti (autres tâches sur le nœud), il n'y a pas de mécanisme de compensation.

3 Question 3 : Stratégie maître-exécutant

3.1 Principe

La stratégie maître-exécutant implémente un **équilibrage dynamique** de la charge :

1. Le **maître** (processus 0) maintient une file de tâches (lignes à calculer)
2. Les **exécutants** (processus 1 à $P-1$) demandent du travail, calculent, renvoient le résultat
3. Le maître distribue les tâches au fur et à mesure des demandes
4. Quand la file est vide, le maître envoie un signal de terminaison

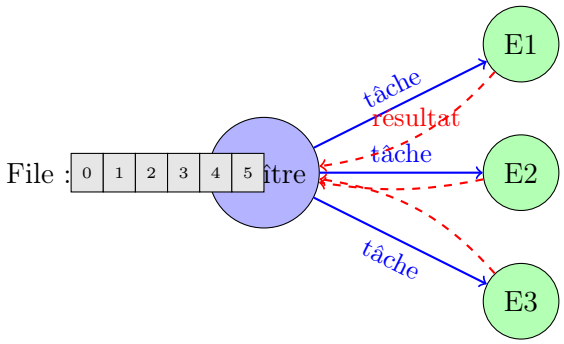

FIGURE 7 – Stratégie maître-esclave avec distribution dynamique

3.2 Résultats expérimentaux

TABLE 6 – Performances de la stratégie maître-esclave

Processus	Temps total (s)	Temps calcul (s)	Speedup	Efficacité (%)
2	6.4322	6.2970	1.00	50.0
3	3.4478	3.4060	1.95	65.0
4	2.4567	2.4256	2.92	73.0
8	0.9254	0.8907	6.60	82.5

3.2.1 Temps de calcul par esclave (4 processus = 3 esclaves)

3.3 Courbes de scaling et comparaison finale

3.4 Analyse et conclusions

3.4.1 Observations principales

1. **Meilleur équilibrage** : La stratégie maître-esclave offre le meilleur équilibrage (σ = 0.0021 s), suivie de l'entrelacé ($\sigma = 0.0463$ s).

TABLE 7 – Temps de calcul individuel (maître-esclave, 4 proc.)

Esclave	Lignes traitées	Temps calcul (s)
1	341	2.4209
2	337	2.4256
3	346	2.4251
Moyenne	341.3	2.4239
Écart-type	–	**0.0021**

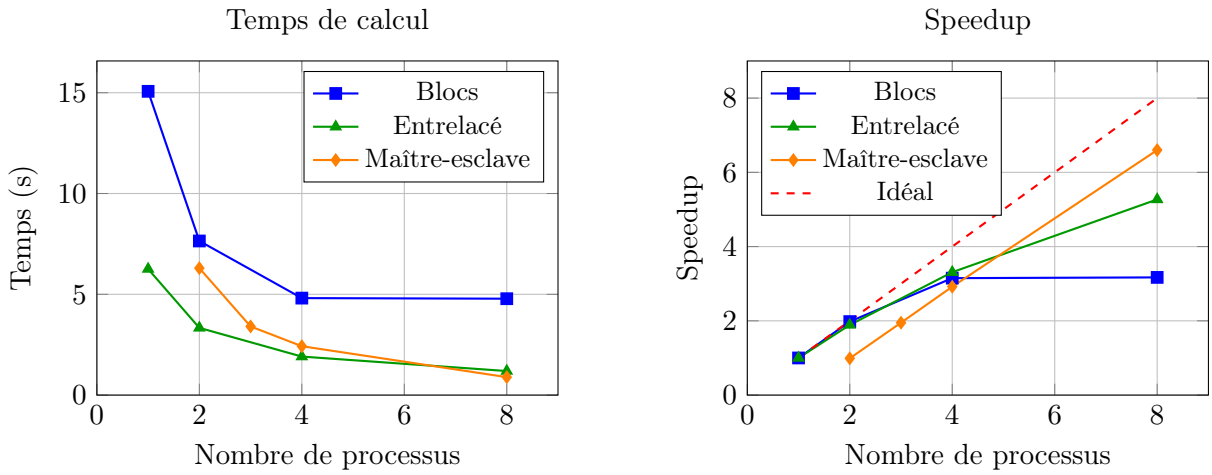

FIGURE 8 – Comparaison des trois stratégies

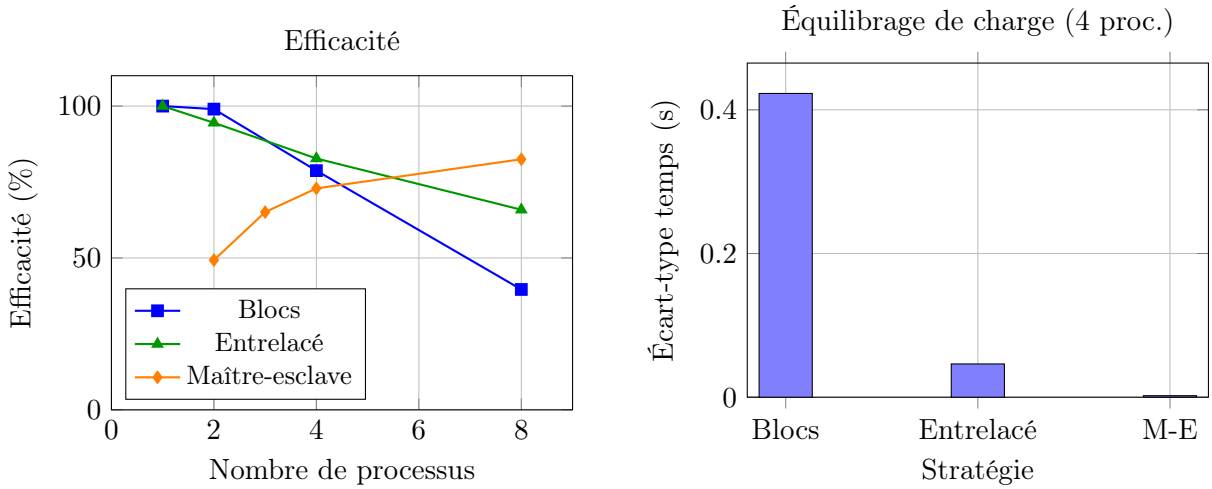

FIGURE 9 – Efficacité et qualité de l'équilibrage

2. **Meilleure performance globale** : La répartition entrelacée est la plus performante pour ce problème car :
   — Pas de processus dédié à la coordination (tous calculent)
   — Overhead de communication minimal (un seul `Gatherv`)
   — Équilibrage suffisant pour ce problème
3. **Maître-esclave : overhead significatif** :
   — Le processus 0 ne calcule pas (perte d'un cœur)
   — Communications fréquentes (2 messages par ligne)
   — Avantage croissant avec le nombre de processus
4. **Recommandations** :
   — **Peu de processus** ($P \leq 8$) : Répartition entrelacée
   — **Beaucoup de processus** ou charge très hétérogène : Maître-esclave
   — **À éviter** : Partition par blocs pour les problèmes à charge variable

Partie 2 : Produit matrice-vecteur

4 Présentation du problème

On considère le produit d'une matrice carrée A de dimension N par un vecteur u de même dimension :

$$v = A \cdot u \quad \text{où} \quad v_i = \sum_{j=0}^{N-1} A_{ij} \cdot u_j$$

La matrice est définie par : $A_{ij} = (i + j) \mod N + 1$
Par simplification, on suppose N divisible par le nombre de processus P.

5 Question 2a : Partitionnement par colonnes

5.1 Principe

Chaque processus possède $N_{loc} = N/P$ colonnes de la matrice A.

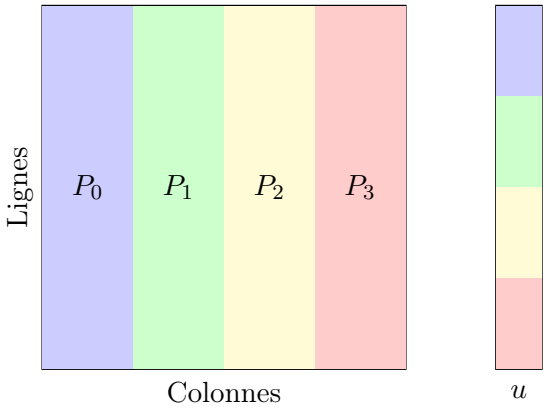

FIGURE 10 – Partitionnement par colonnes – Chaque processus a N_{loc} colonnes et la partie correspondante de u

5.2 Algorithme

1. Chaque processus p stocke les colonnes $[p \cdot N_{loc}, (p+1) \cdot N_{loc})$ de A
2. Chaque processus p possède la partie correspondante de u : $u[p \cdot N_{loc} : (p+1) \cdot N_{loc}]$
3. Chaque processus calcule une **somme partielle** : $v_i^{(p)} = \sum_{j=p \cdot N_{loc}}^{(p+1) \cdot N_{loc}-1} A_{ij} \cdot u_j$
4. `MPI_Allreduce` avec `MPI_SUM` pour sommer toutes les contributions

5.3 Résultats expérimentaux

5.4 Analyse

Avantages :
— Chaque processus ne stocke que $N \times N_{loc}$ éléments de A
— Chaque processus ne possède que N_{loc} éléments de u
Inconvénients :
— Communication `Allreduce` de taille N (le vecteur résultat complet)
— Coût de communication croissant avec N

TABLE 8 – Performances du produit matrice-vecteur par colonnes ($N = 1200$)

Processus	Temps (s)	Speedup	Efficacité (%)
1	0.675	1.00	100.0
2	0.370	1.82	91
4	0.212	3.18	79.5
8	0.179	3.77	47.13

6 Question 2b : Partitionnement par lignes

6.1 Principe

Chaque processus possède $N_{loc} = N/P$ lignes de la matrice A.

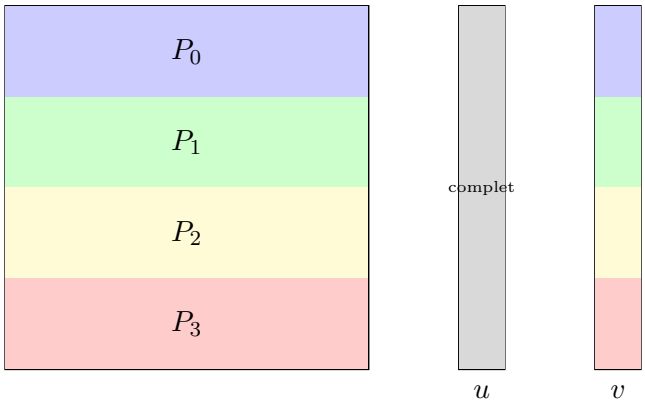

FIGURE 11 – Partitionnement par lignes – Chaque processus a N_{loc} lignes et calcule N_{loc} éléments de v

6.2 Algorithme

1. Chaque processus p stocke les lignes $[p \cdot N_{loc}, (p+1) \cdot N_{loc})$ de A
2. Chaque processus possède le vecteur u **complet**
3. Chaque processus calcule N_{loc} éléments **complets** de v : $v_i = \sum_{j=0}^{N-1} A_{ij} \cdot u_j$ pour $i \in [p \cdot N_{loc}, (p+1) \cdot N_{loc})$
4. `MPI_Allgather` pour collecter tous les morceaux de v

6.3 Résultats expérimentaux

TABLE 9 – Performances du produit matrice-vecteur par lignes ($N = 1200$)

Processus	Temps (s)	Speedup	Efficacité (%)
1	0.685	1.00	100.0
2	0.383	1.79	89.5
4	0.213	3.22	80.5
8	0.191	3.59	44.88

6.4 Analyse

Avantages :
— Chaque processus calcule des éléments **indépendants** de v (pas de réduction)
— Communication `Allgather` de taille N_{loc} par processus
Inconvénients :
— Chaque processus doit posséder le vecteur u complet
— Si u change à chaque itération (méthodes itératives), il faut un `Allgather` de u à chaque étape

7 Comparaison des deux approches

TABLE 10 – Comparaison colonnes vs lignes

Critère	**Par colonnes**	**Par lignes**
Stockage de A par proc.	$N \times N_{loc}$	$N_{loc} \times N$
Stockage de u par proc.	N_{loc}	N (complet)
Type de calcul	Somme partielle	Éléments complets
Communication	`Allreduce`(N)	`Allgather`(N_{loc})
Efficacité (8 proc.)	47.13%	44.88%

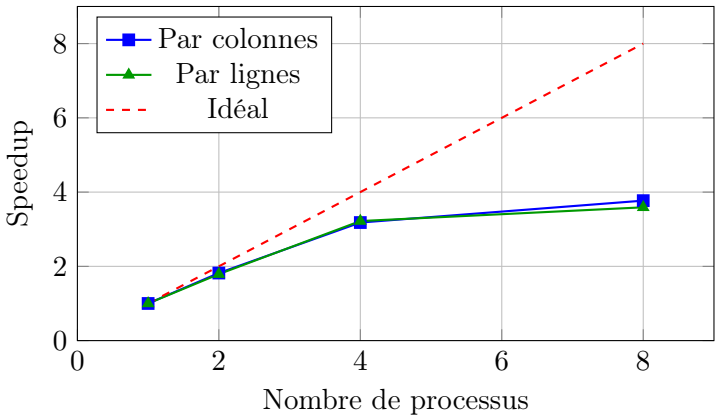

FIGURE 12 – Comparaison des speedups : colonnes vs lignes

Conclusion : Le partitionnement par lignes est légèrement plus efficace car :
— L'`Allgather` est moins coûteux que l'`Allreduce` pour ce problème
— Chaque processus calcule des résultats indépendants (pas de réduction arithmétique)

Partie 3 : Entraînement pour l'examen écrit

8 Contexte du problème

Alice a parallélisé un code sur machine à mémoire distribuée. Pour un jeu de données spécifique :

— La partie parallélisable représente **90%** du temps d'exécution séquentiel
— La partie séquentielle représente donc **10%** du temps

Notations :

— $f = 0.90$: fraction parallélisable
— $(1 - f) = 0.10$: fraction séquentielle
— n : nombre de processeurs

9 Question 1 : Accélération maximale (loi d'Amdahl)

9.1 Rappel de la loi d'Amdahl

La loi d'Amdahl prédit l'accélération maximale théorique pour un problème de taille fixe :

$$\boxed{S(n) = \frac{1}{(1-f) + \frac{f}{n}}} \tag{1}$$

où :

— $S(n)$ est le speedup avec n processeurs
— f est la fraction du code parallélisable
— $(1 - f)$ est la fraction séquentielle incompressible

9.2 Calcul de l'accélération maximale

Pour $n \gg 1$ (nombre de processeurs très grand), le terme $\frac{f}{n}$ devient négligeable :

$$S_{max} = \lim_{n\to\infty} S(n) = \lim_{n\to\infty} \frac{1}{(1-f) + \frac{f}{n}} = \frac{1}{1-f} \tag{2}$$

Application numérique avec $f = 0.90$:

$$\boxed{S_{max} = \frac{1}{1-0.90} = \frac{1}{0.10} = 10} \tag{3}$$

9.3 Interprétation

Résultat : L'accélération maximale théorique est de **10**, quel que soit le nombre de processeurs utilisés.

Explication physique : Même avec une infinité de processeurs, les 10% de code séquentiel ne peuvent pas être accélérés. Le temps d'exécution minimal est donc $0.10 \times T_{seq}$, ce qui correspond à une accélération de $1/0.10 = 10$.

Limite fondamentale : Cette limite montre l'importance cruciale de la fraction séquentielle. Même une petite partie non parallélisable peut limiter drastiquement les gains.

10 Question 2 : Nombre raisonnable de nœuds

10.1 Analyse de l'efficacité

L'efficacité parallèle est définie par :

$$E(n) = \frac{S(n)}{n} = \frac{1}{n \cdot \left[(1-f) + \frac{f}{n}\right]} = \frac{1}{n(1-f)+f} \tag{4}$$

10.2 Calcul pour différentes valeurs de n

TABLE 11 – Speedup et efficacité en fonction du nombre de processeurs

n	$(1-f)+f/n$	$S(n)$	$E(n)$	Gain marginal
1	1.000	1.00	100.0%	–
2	0.550	1.82	90.9%	+0.82
4	0.325	3.08	76.9%	+1.26
8	0.213	4.71	58.8%	+1.63
16	0.156	6.40	40.0%	+1.69
32	0.128	7.80	24.4%	+1.40
64	0.114	8.77	13.7%	+0.97
∞	0.100	10.00	0%	–

10.3 Critère de choix

On considère généralement qu'une efficacité $E(n) \geq 50\%$ est acceptable. Au-delà, les ressources sont "gaspillées".

Résolution : Trouver n tel que $E(n) \geq 0.50$

$$\frac{1}{n(1-f)+f} \geq 0.50 \tag{5}$$

$$n(1-f)+f \leq 2 \tag{6}$$

$$n \times 0.10 + 0.90 \leq 2 \tag{7}$$

$$n \leq \frac{2-0.90}{0.10} = 11 \tag{8}$$

Recommandation :

$$\boxed{n \leq 8 \text{ à } 10 \text{ nœuds}} \tag{9}$$

10.4 Justification économique

— Avec **8 nœuds** : $S = 4.71$, $E = 58.8\% \Rightarrow$ bon compromis
— Avec **16 nœuds** : $S = 6.40$, $E = 40.0\% \Rightarrow$ gaspillage significatif
— Avec **32 nœuds** : $S = 7.80$, $E = 24.4\% \Rightarrow 3/4$ des ressources gaspillées

Conclusion : Au-delà de 8-10 nœuds, chaque nœud supplémentaire apporte un gain marginal décroissant pour un coût constant. Il est préférable de limiter à 8-10 nœuds pour ce problème.

11 Question 3 : Loi de Gustafson (problème mis à l'échelle)

11.1 Observation d'Alice

Alice observe une accélération maximale de **4** (et non 10 comme prédit par Amdahl). Cela signifie que la fraction séquentielle *effective* est plus grande que prévu :

$$S_{obs} = 4 = \frac{1}{1 - f_{eff}} \implies 1 - f_{eff} = 0.25 \implies f_{eff} = 0.75 \tag{10}$$

En pratique, 25% du temps est séquentiel (communication, synchronisation, etc.).

11.2 Rappel de la loi de Gustafson

La loi de Gustafson s'applique au **scaling faible** (weak scaling) : on augmente la taille du problème proportionnellement au nombre de processeurs.

$$\boxed{S_{Gustafson}(n) = n - (1 - f)(n - 1) = (1 - f) + f \cdot n} \tag{11}$$

Hypothèse clé : La partie parallèle croît avec la taille du problème, mais la partie séquentielle reste constante.

11.3 Analyse du problème doublé

Soit les temps pour le problème original :
— Temps séquentiel : T_s
— Temps parallèle (sur 1 proc.) : $T_p = 3T_s$ (car $f_{eff} = 75\%$ signifie $T_p = 3T_s$)
Quand on **double** les données avec une complexité linéaire :
— Nouveau temps séquentiel : $T'_s = T_s$ (inchangé – initialisation, I/O, etc.)
— Nouveau temps parallèle : $T'_p = 2T_p = 6T_s$ (double – traitement des données)

11.4 Calcul du speedup maximal

Pour le problème doublé, le temps séquentiel total est :

$$T_{seq,new} = T_s + 6T_s = 7T_s \tag{12}$$

Sur n processeurs :

$$T_{par}(n) = T_s + \frac{6T_s}{n} \tag{13}$$

Le speedup est donc :

$$S(n) = \frac{T_{seq,new}}{T_{par}(n)} = \frac{7T_s}{T_s + \frac{6T_s}{n}} = \frac{7}{1 + \frac{6}{n}} = \frac{7n}{n + 6} \tag{14}$$

Pour $n \gg 1$:

$$\boxed{S_{max,doubled} = \lim_{n \to \infty} \frac{7n}{n + 6} = 7} \tag{15}$$

11.5 Vérification avec la formule de Gustafson

La nouvelle fraction séquentielle est :

$$(1 - f') = \frac{T_s}{7T_s} = \frac{1}{7} \approx 14.3\% \tag{16}$$

En appliquant Gustafson :

$$S_{max} = \frac{1}{1 - f'} = \frac{1}{1/7} = 7 \quad \checkmark \tag{17}$$

11.6 Interprétation et généralisation

TABLE 12 – Évolution du speedup maximal avec la taille du problème

Taille	T_s	T_p	S_{max}
$\times 1$ (original)	T_s	$3T_s$	4
$\times 2$	T_s	$6T_s$	7
$\times 4$	T_s	$12T_s$	13
$\times k$	T_s	$3kT_s$	$3k+1$

Formule générale : Pour un facteur d'échelle k :

$$S_{max}(k) = 1 + 3k \tag{18}$$

11.7 Conclusion sur Gustafson

1. **Résultat** : En doublant les données, l'accélération maximale passe de **4 à 7**.
2. **Insight de Gustafson** : Pour les grands problèmes, la partie séquentielle devient proportionnellement négligeable. Plus le problème est grand, plus on peut utiliser efficacement un grand nombre de processeurs.
3. **Différence avec Amdahl** :
   — **Amdahl** (strong scaling) : taille fixe ⇒ limite stricte
   — **Gustafson** (weak scaling) : taille croissante ⇒ pas de limite théorique

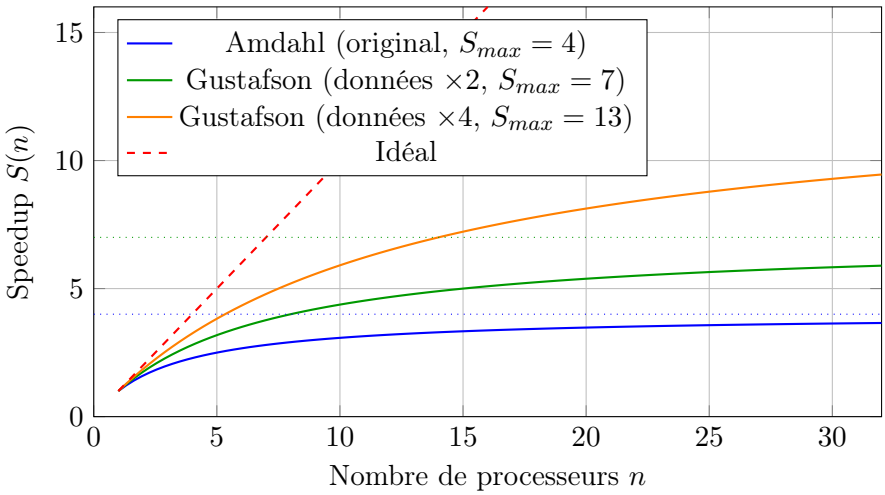

FIGURE 13 – Comparaison Amdahl vs Gustafson : l'augmentation de la taille du problème repousse la limite du speedup

12 Commandes d'exécution

N'ayant pas assez de coeurs sur ma machine, les commandes suivantes ont été utilisées pour d'exécuter les différentes implémentations :

```
# Version par blocs
mpirun --oversubscribe -np 4 python mandelbrot_mpi_block.py
```

```
# Version entrelacée
mpirun --oversubscribe -np 4 python mandelbrot_mpi_interleaved.py

# Version maître-exécutant
mpirun --oversubscribe -np 4 python mandelbrot_mpi_maitre_exécutant.py

# Par colonnes
 mpirun --oversubscribe -np $np python matvec_col.py

# Par lignes
mpirun --oversubscribe -np $np python matvec_row.py
```