OS202 – Systèmes parallèles et distribués

TP4 : Jeu de la vie – MPI (synchrone vs asynchrone)

MENESES GAMBOA Carlos

École Nationale des techniques avancées
Février 2026

Table des matières

1 Résumé

Ce rapport étudie l'implémentation du *Jeu de la vie* de Conway sur une grille torique et la comparaison de plusieurs versions : (i) une version séquentielle "scalaire" (`game_of_life.py`), (ii) une version séquentielle vectorisée par convolution (`game_of_life_vect.py`), (iii) une version MPI à 2 processus avec échanges synchrones et envoi de la grille complète (`1_game_of_life.py`), (iv) une décomposition de domaine par lignes avec *halo exchange* et rassemblement (`2_game_of_life.py`), (v) une version MPI asynchrone où l'affichage ne bloque pas le calcul (`3_game_of_life_vect.py`).

Les performances sont analysées via des métriques de temps (calcul, communication, affichage, temps frame), ainsi que des indicateurs dérivés (FPS, accélération, efficacité). Une attention particulière est portée à l'équité des métriques entre la version séquentielle vectorisée et la version asynchrone, car elles ne mesurent pas exactement la même chose (découplage calcul/affichage).

2 Machine utilisée

TABLE 1 – Caractéristiques de la machine (à adapter si besoin).

Champ	Valeur
Architecture	x86_64
Model name	11th Gen Intel Core i5-1135G7 @ 2.40GHz
CPU(s)	8
Thread(s) per core	2
Core(s) per socket	4
L1d / L1i cache	192 KiB / 128 KiB
L2 cache	5 MiB
L3 cache	8 MiB
Environnement	Linux / WSL2 (mpirun `-oversubscribe`)

3 Jeu de la vie : rappel et métriques

3.1 Règles et tore

Le *Jeu de la vie* est un automate cellulaire binaire (cellule vivante/morte) sur une grille 2D. À chaque itération, l'état d'une cellule dépend du nombre n de voisins vivants (8-voisinage) :

— cellule vivante : meurt si $n < 2$ (sous-population) ou $n > 3$ (sur-population), survit si $n \in \{2, 3\}$;

— cellule morte : devient vivante si $n = 3$.

Dans ce TP, la grille est **torique** (*wrap-around*) : les bords se recollent (haut avec bas, gauche avec droite). Cette condition est implémentée en version vectorisée par `boundary='wrap'` dans la convolution.

3.2 Mesures et définitions

On utilise (après *warmup*) des moyennes sur un nombre d'échantillons (`samples`). Les métriques typiques affichées sont :

— $\overline{T}_{\text{compute}}$: temps moyen de calcul d'une génération (ms),
— $\overline{T}_{\text{draw}}$: temps moyen d'affichage (ms),
— $\overline{T}_{\text{frame}}$: temps total par itération de boucle principale (ms),
— FPS $\approx 1/\overline{T}_{\text{frame}}$: images/seconde (boucle UI),
— volume message : taille de la grille envoyée (ici $\approx 100 \times 90$ octets ≈ 8.8 KiB).

Point crucial (équité) : en version séquentielle, on calcule *exactement une génération par frame*. En version MPI asynchrone, le calcul et l'affichage sont *découplés* : le processus de calcul peut produire de nombreuses générations entre deux mises à jour réellement reçues par l'UI. Ainsi, $\overline{T}_{\text{frame}}$ côté UI ne mesure pas le débit de simulation, mais le débit d'animation (frames) et le coût d'affichage.

4 Résultats expérimentaux et analyse

Les exécutions ont été réalisées avec le pattern `glider`, une grille 100×90, et une résolution 800×800.

4.1 Versions séquentielles : scalaire vs vectorisée

4.1.1 Principe

— **Scalaire** (`game_of_life.py`) : calcul des voisins via boucles explicites (coût Python élevé).
— **Vectorisée** (`game_of_life_vect.py`) : calcul du nombre de voisins via convolution 3×3 (SciPy en C), puis application d'une fonction de décision.

4.1.2 Mesures

TABLE 2 – Séquentiel : comparaison des performances (glider, 100×90).

Version	$\overline{T}_{\text{compute}}$ (ms)	$\overline{T}_{\text{draw}}$ (ms)	$\overline{T}_{\text{frame}}$ (ms)	FPS
Scalaire (`game_of_life.py`)	86.587	19.350	106.103	9.42
Vectorisée (`game_of_life_vect.py`)	0.704	13.601	14.344	69.71

4.1.3 Analyse

— Le **gain sur le calcul** est massif : $\approx 86.587/0.704 \simeq 123\times$.
— Le **gain "bout-en-bout"** (frame/FPS) est plus modéré : $\approx 106.103/14.344 \simeq 7.4\times$, car l'affichage devient un goulot d'étranglement (Pygame remplit la grille cellule par cellule).
— Le compteur `changed/frame` n'est pas strictement comparable ici, car la liste `diff` n'est pas renseignée dans les versions vectorisées (retourne une liste vide).

4.2 Point 1 : MPI synchrone (2 processus, grille complète)

4.2.1 Principe

— **Rank 1** calcule la génération suivante puis envoie la **grille complète** à **rank 0**.
— **Rank 0** reçoit (communication bloquante), affiche, puis recommence.

Ce schéma crée un **handshake** naturel : l'affichage bloque le calcul car le rang de calcul attend implicitement la boucle du rang 0.

4.2.2 Mesures

TABLE 3 – Point 1 : métriques MPI synchrones (2 processus).

Rôle	$\overline{T}_{\text{comm}}$ (ms)	$\overline{T}_{\text{compute}}$ (ms)	$\overline{T}_{\text{draw}}$ (ms)	FPS (UI)
Rank 0 (UI)	recv = 97.879	–	22.379	8.31
Rank 1 (calcul)	send = 0.052	97.774	–	–

Observation : le temps de réception côté rank 0 ($\sim$ 98 ms) est du même ordre que le temps de calcul ($\sim$ 98 ms). Cela indique que le *pipeline* est fortement **synchronisé** : chaque itération attend l'autre, et l'UI ne peut pas "prendre de l'avance".

4.3 Point 2 : MPI décomposition par lignes + halo exchange

4.3.1 Principe

La grille est découpée en blocs de lignes. Chaque rang de calcul traite un sous-domaine et échange des **lignes fantômes** (*halo*) avec ses voisins (haut/bas, torus). Ensuite, rank 0 reconstruit la grille complète via un **gather** puis affiche.

4.3.2 Mesures

TABLE 4 – Point 2 : métriques selon le nombre de processus (glider, 100×90).

Proc. tot.	Rang(s) calcul	Rows/rang	$\overline{T}_{\text{halo}}$ (ms)	$\overline{T}_{\text{compute}}$ (ms)	$\overline{T}_{\text{gather}}$ (ms)	$\overline{T}_{\text{draw}}$ (ms)	FPS (UI)
2	1	100	0.019	56.428	56.587	23.750	12.42
3	2	50	0.051	37.613	40.383	29.122	14.37
4	3	33–34	0.056	29.078	32.532	33.692	15.08

4.3.3 Analyse

— **Scaling du calcul :** la moyenne $\overline{T}_{\text{compute}}$ diminue bien quand on ajoute des rangs de calcul (de 56.4 ms à 29.1 ms entre 1 et 3 rangs). L'accélération du calcul est $\approx 1.94\times$ avec 3 rangs de calcul (efficacité ≈ 0.65).
— **Halo exchange négligeable :** $\overline{T}_{\text{halo}} \ll \overline{T}_{\text{compute}}$ (ici $<$0.1 ms). Le coût de communication principal est plutôt le **rassemblement** vers rank 0.
— **Goulot UI :** le FPS n'augmente que de 12.4 à 15.1 malgré le gain de calcul, car **rank 0** reste séquentiel pour l'affichage (et subit aussi la contention CPU quand on augmente le nombre de processus).

— **Effet de contention :** $\overline{T}_{\text{draw}}$ augmente avec le nombre de processus (23.8 ms → 33.7 ms), ce qui est cohérent avec une machine à ressources limitées où l'UI se fait préempter par les processus MPI (surtout en `-oversubscribe`).

TABLE 5 – Accélération "bout-en-bout" (temps frame côté UI) par rapport au Point 1.

Cas	$\overline{T}_{\text{frame}}$ UI (ms)	FPS	Acc. vs P1
Point 1 (2 proc, sync)	120.288	8.31	1.00
Point 2 (2 proc, 1 calcul)	80.506	12.42	1.49
Point 2 (3 proc, 2 calcul)	69.581	14.37	1.73
Point 2 (4 proc, 3 calcul)	66.307	15.08	1.81

4.4 Point 3 : MPI asynchrone (probe + non-blocking send)

4.4.1 Principe

Le point 3 (`3_game_of_life_vect.py`) introduit un **découplage fort** :

— **Rank 0 (UI)** ne bloque jamais sur la réception : il utilise `Iprobe` et draine toutes les grilles disponibles, puis affiche le *dernier état* reçu.

— **Rank 1 (calcul)** utilise `Isend` et **n'attend pas** la fin d'envoi : si l'envoi précédent n'est pas terminé, l'itération courante est **droppée** (pas de blocage).

Ce design vise la réactivité UI (pas de `Recv` bloquant) et la maximisation du débit de calcul, au prix d'une **sous-échantillonnage** des états affichés.

4.4.2 Mesures

TABLE 6 – Point 3 : métriques mesurées (2 processus).

Rôle	$\overline{T}_{\text{poll+recv}}$ (ms)	$\overline{T}_{\text{draw}}$ (ms)	$\overline{T}_{\text{frame}}$ (ms)	FPS
Rank 0 (UI)	0.045	25.440	25.620	39.03

TABLE 7 – Point 3 : débit de calcul et envois (rank 1).

$\overline{T}_{\text{compute}}$ (ms)	sends_ok	drops	send_ok_rate	Interprétation
0.391	147	19187	0.01	envois très souvent saturés

4.4.3 Pourquoi "ça paraît plus rapide" alors que certaines métriques sont pires ?

C'est le point qui prêtait à confusion : la version asynchrone peut *paraître* beaucoup plus rapide visuellement, même si son "débit d'updates affichées" est inférieur à la version séquentielle vectorisée.

Inéquivalence des métriques :

— Dans `game_of_life_vect.py` (séquentiel), **1 frame = 1 génération**. Donc "FPS" ≈ "générations affichées/s".

— Dans `3_game_of_life_vect.py` (async), **1 frame UI** peut afficher **0 ou plusieurs** mises à jour reçues, et chaque mise à jour reçue peut correspondre à un état *très avancé* (car rank 1 a calculé de nombreuses générations entre-temps).

Donc comparer uniquement $\overline{T}_{\text{frame}}$ ou FPS entre les deux n'évalue pas le *débit de simulation*, mais le *débit d'animation*.

Pour rendre la comparaison plus pertinente, on distingue :

— **Débit affiché** : nombre d'**updates reçues** et effectivement affichées par seconde.

— **Débit de calcul (kernel)** : inverse du temps de calcul d'une génération (hors affichage).

TABLE 8 – Comparaison équitable : débit affiché vs débit de calcul.

Version	FPS (UI)	Updates/frame	Updates affichées/s	Débit calcul brut (gen/s)
Seq. vectorisée	69.71	≈ 1.00	≈ 69.71	$\approx 1/0.000704 \simeq 1420$
MPI async (P3)	39.03	0.50	$\approx 39.03 \times 0.50 \simeq 19.44$	$\approx 1/0.000391 \simeq 2558$

Interprétation :

— La version **séquentielle vectorisée** affiche davantage d'états (environ 70 updates/s) : animation plus fluide.

— La version **asynchrone** calcule beaucoup plus vite (kernel $\sim 1.8\times$ plus rapide ici), mais **n'arrive pas à transmettre** toutes les générations (taux d'envoi 1 %). L'UI n'affiche qu'$\sim$ 19 updates/s, mais chaque update correspond à un état "loin dans le futur", ce qui donne une impression de simulation très rapide (grands sauts temporels).

4.4.4 Conclusion sur le point 3

Le point 3 ne doit pas être lu comme "plus rapide en FPS" mais comme une **architecture plus robuste** :

— l'UI ne bloque pas (pas de `Recv` bloquant) ;

— le calcul tourne au maximum ;

— on accepte de **perdre** des frames de simulation pour conserver la réactivité.

5 Conclusion générale

— **Vectorisation = gain dominant :** passer du calcul scalaire Python à la convolution vectorisée réduit $\overline{T}_{\text{compute}}$ d'environ $123\times$. Le gain bout-en-bout est $\sim 7.4\times$ car l'affichage devient limitant.

— **MPI synchrone (Point 1) :** l'échange bloquant et l'envoi de grille complète imposent un verrouillage fort entre calcul et affichage (FPS ~ 8.3).

— **Décomposition (Point 2) :** le calcul scale correctement avec le nombre de rangs de calcul, mais le FPS plafonne car rank 0 (gather+draw) reste un goulot. L'oversubscription et la contention peuvent dégrader $\overline{T}_{\text{draw}}$.

— **Asynchrone (Point 3) :** les métriques "FPS" ne sont pas directement comparables à la version séquentielle car le calcul et l'affichage sont découplés. L'UI affiche moins d'updates par seconde, mais des états plus avancés (grands sauts), d'où l'impression d'une évolution plus rapide.