Parallélisation en mémoire distribuée : Génération de labyrinthe avec MPI

Junior Koudogbo

14 décembre 2025

Résumé

Ce rapport présente la parallélisation de la génération de labyrinthe en utilisant MPI (Message Passing Interface) pour la programmation parallèle en mémoire distribuée. Je décris les modifications apportées au code séquentiel de référence, justifie mes choix d'implémentation, et présente une étude de performances incluant l'analyse de la scalabilité forte et faible de mon implémentation parallèle.

Table des matières

1 Introduction

La génération de labyrinthes est un problème algorithmique classique qui peut nécessiter des calculs intensifs pour de grandes tailles. La parallélisation de ce type de problème permet d'exploiter efficacement les architectures parallèles modernes, notamment les clusters et les machines multi-cœurs. Ce projet s'intéresse à la parallélisation de la génération de labyrinthe en utilisant MPI, un standard de programmation parallèle en mémoire distribuée.

2 Description du problème et de l'algorithme

2.1 Génération de labyrinthe

L'algorithme de génération de labyrinthe utilisé dans ce projet est basé sur une approche itérative qui construit progressivement les murs du labyrinthe. Le principe général est le suivant :

1. Initialisation d'une grille avec des murs sur les bords et un espace vide à l'intérieur
2. Placement aléatoire d'îlots (murs) à l'intérieur du labyrinthe
3. Identification des cases constructibles (cases vides qui peuvent devenir des murs selon certaines règles)
4. Construction itérative : à chaque itération, sélection aléatoire d'une case constructible et transformation en mur, puis mise à jour des cases constructibles voisines

Une case est dite **constructible** si elle est vide et si elle possède exactement un mur parmi ses 4 voisins directs (haut, bas, gauche, droite), avec les 4 voisins diagonaux vides. Cette règle garantit que le labyrinthe généré reste connexe et ne crée pas d'îlots isolés.

2.2 Algorithme séquentiel

L'algorithme séquentiel de référence (`gen_lab.c`) procède de la manière suivante :

1. Initialisation de la grille de taille $N \times M$: murs sur les bords ($i = 0$, $i = N - 1$, $j = 0$, $j = M - 1$), espace vide ailleurs
2. Placement de *nbilots* îlots aléatoirement à l'intérieur du labyrinthe
3. Calcul initial des cases constructibles : parcours de toutes les cases intérieures et marquage des cases constructibles avec la valeur -1
4. Suppression aléatoire de quelques cases constructibles sur les bords pour éviter une génération trop régulière
5. Boucle principale :
   — Sélection aléatoire d'une case constructible parmi les *nbcons* disponibles
   — Transformation de cette case en mur (valeur 0)
   — Mise à jour des 8 voisins : si une case vide devient constructible, elle est ajoutée à la liste ; si une case constructible ne l'est plus, elle est retirée
   — Répétition jusqu'à ce qu'il n'y ait plus de cases constructibles
6. Sauvegarde du labyrinthe dans un fichier binaire

3 Analyse du parallélisme

3.1 Identification des dépendances

L'analyse du code séquentiel révèle que l'algorithme présente des **dépendances de données** importantes :

- La sélection aléatoire d'une case constructible dépend du nombre total *nbcons* de cases constructibles
- La mise à jour des voisins d'une case construite nécessite l'accès aux 8 cases adjacentes
- Les cases constructibles peuvent être mises à jour par plusieurs constructions simultanées, créant des conditions de course potentielles

Ces dépendances rendent la parallélisation plus complexe que pour des algorithmes à grain fin totalement indépendants. Cependant, il est possible de paralléliser en utilisant une **décomposition de domaine** : chaque processus gère une portion du labyrinthe et communique avec ses voisins pour synchroniser les frontières.

3.2 Stratégie de parallélisation

La stratégie choisie est une **décomposition de domaine 1D par bandes de lignes** :

- Le labyrinthe de taille $N \times M$ est divisé en p bandes horizontales de N/p lignes chacune
- Chaque processus i gère les lignes $[i \cdot N/p, (i+1) \cdot N/p - 1]$
- Chaque processus maintient deux lignes fantômes (ghost rows) : une au-dessus et une en-dessous de sa bande, pour permettre le calcul correct de la constructibilité des cases frontières
- Les processus échangent périodiquement leurs lignes frontières pour maintenir la cohérence des données

Cette approche présente plusieurs avantages :

- Réduction de la communication : seules les lignes frontières sont échangées
- Localité mémoire : chaque processus travaille sur des données contiguës en mémoire
- Équilibrage de charge : chaque processus traite approximativement le même nombre de lignes

4 Implémentation parallèle

4.1 Modifications apportées au code

Le code parallèle (`gen_lab-parallel.c`) introduit plusieurs modifications par rapport à la version séquentielle :

4.1.1 Initialisation MPI

Listing 1 – Initialisation MPI

```
MPI_Init(&argc, &argv);
int rank, size;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
```

4.1.2 Décomposition de domaine

Chaque processus calcule sa portion locale du labyrinthe :

Listing 2 – Calcul de la décomposition

```
int N_loc = N / size;  // Nombre de lignes locales
int start_row = rank * N_loc;  // Premi re ligne globale
int local_rows = N_loc + 2;  // +2 pour les lignes fant mes
int(*l)[M] = malloc(sizeof(int[local_rows][M]));
```

Les indices locaux $[1, N\_loc]$ correspondent aux lignes réelles, tandis que les indices 0 et $N\_loc + 1$ sont les lignes fantômes.

4.1.3 Échange de lignes fantômes

Une fonction dédiée gère l'échange des lignes frontières entre processus voisins :

Listing 3 – Échange de lignes fantômes

```
void update_ghosts(int rows, int cols, int l[rows][cols],
                   int rank, int size) {
    MPI_Status status;
    int up_neighbor = (rank == 0) ? MPI_PROC_NULL : rank - 1;
    int down_neighbor = (rank == size - 1) ? MPI_PROC_NULL : rank
        + 1;

    //   change   vers le haut
    MPI_Sendrecv(&l[1][0], cols, MPI_INT, up_neighbor, 0,
                 &l[0][0], cols, MPI_INT, up_neighbor, 0,
                 MPI_COMM_WORLD, &status);

    //   change   vers le bas
    MPI_Sendrecv(&l[rows-2][0], cols, MPI_INT, down_neighbor, 0,
                 &l[rows-1][0], cols, MPI_INT, down_neighbor, 0,
                 MPI_COMM_WORLD, &status);
}
```

L'utilisation de `MPI_Sendrecv` garantit l'absence de deadlock et permet un échange efficace des données.

4.1.4 Boucle principale parallèle

La boucle principale est modifiée pour gérer la synchronisation entre processus :

1. Chaque processus calcule son nombre local de cases constructibles *nbcons*
2. Réduction globale avec `MPI_Allreduce` pour obtenir le nombre total *global_nbcons*
3. Si $global\_nbcons = 0$, tous les processus terminent
4. Sinon, chaque processus construit une case si $nbcons > 0$ localement
5. Échange des lignes fantômes pour synchroniser les frontières
6. Re-vérification des cases frontières qui peuvent être devenues constructibles après l'échange

4.1.5 Reconstitution du labyrinthe

À la fin du calcul, seul le processus 0 collecte toutes les portions pour reconstituer le labyrinthe complet :

Listing 4 – Reconstitution avec MPI_Gather

```
int *send_buffer = malloc(N_loc * M * sizeof(int));
// Copie des donn es locales sans les lignes fant mes
for(int i=0; i<N_loc; i++) {
    memcpy(&send_buffer[i*M], &l[i+1][0], M * sizeof(int));
}

MPI_Gather(send_buffer, N_loc * M, MPI_INT,
           full_grid, N_loc * M, MPI_INT,
           0, MPI_COMM_WORLD);
```

4.2 Justification des choix

4.2.1 Décomposition 1D vs 2D

Une décomposition 2D (par blocs) aurait réduit davantage la communication, mais aurait également compliqué l'implémentation. La décomposition 1D par bandes est plus simple à implémenter et suffisante pour des machines avec un nombre modéré de processus.

4.2.2 Échange périodique des fantômes

L'échange des lignes fantômes après chaque itération de construction garantit la cohérence des données, mais introduit une surcharge de communication. Une optimisation possible serait d'échanger uniquement lorsque nécessaire, mais cela compliquerait la logique du code.

4.2.3 Utilisation de MPI_Allreduce

L'utilisation de `MPI_Allreduce` pour calculer le nombre total de cases constructibles permet à tous les processus de connaître l'état global et de terminer simultanément. Une alternative serait d'utiliser `MPI_Reduce` suivi d'un `MPI_Bcast`, mais `MPI_Allreduce` est plus efficace.

5 Étude de performances

5.1 Environnement d'expérimentation

Les mesures de performances ont été effectuées sur une machine avec les caractéristiques suivantes :

- Processeur : Intel Core i5-6300U @ 2.40GHz
- Nombre de cœurs physiques : 2
- Nombre de threads logiques : 4 (Hyper-Threading)
- Compilateur : `mpicc` avec les options `-O2 -g -Wall -std=c99`
- Système d'exploitation : Linux

5.2 Méthodologie

Pour chaque configuration, j'ai effectué :

— 3 exécutions du programme
— Calcul de la moyenne et de l'écart-type des temps d'exécution
— Mesure du temps total de génération (incluant l'initialisation, le calcul et la sauvegarde)

Le nombre de processus est contrôlé via la commande `mpirun -np`. Pour les configurations avec 3 et 4 processus, j'ai utilisé l'option `--oversubscribe` pour permettre l'exécution de plus de processus que de cœurs physiques disponibles. Cette option est justifiée car :

— Elle permet d'étudier le comportement de l'algorithme avec un nombre de processus supérieur au nombre de cœurs physiques
— Elle simule un environnement où plusieurs processus partagent les mêmes ressources CPU, ce qui est courant dans les clusters HPC
— Elle permet d'évaluer l'impact de la surcharge de communication et de synchronisation lorsque le nombre de processus augmente
— Elle est nécessaire pour effectuer une étude complète de scalabilité, même sur une machine avec un nombre limité de cœurs

5.3 Scalabilité forte

La scalabilité forte mesure l'efficacité de la parallélisation pour une taille de problème fixe. J'ai testé avec les paramètres suivants :

— Dimensions : 400×600 pixels
— Nombre d'îlots : 20

Les résultats sont présentés dans le tableau suivant. Les temps sont en secondes, le speedup est calculé comme $S(n) = T_s/T_p(n)$ où T_s est le temps séquentiel et $T_p(n)$ le temps avec n processus. L'efficacité est calculée comme $E(n) = S(n)/n \times 100\%$.

Table 1 – Résultats de scalabilité forte

Processus	Temps moyen (s)	Écart-type (s)	Speedup	Efficacité (%)
1	17.84	1.08	1.00	100.0
2	5.08	0.15	3.52	175.8
3	4.95	0.29	3.61	120.2
4	6.64	0.43	2.69	67.2

Le speedup est calculé comme : $S(n) = \frac{T_s}{T_p(n)}$ où $T_s = 17.84$ s est le temps séquentiel et $T_p(n)$ le temps avec n processus.

L'efficacité est calculée comme : $E(n) = \frac{S(n)}{n} \times 100\%$.

Pour mieux comprendre les performances de la parallélisation, comparons les temps d'exécution mesurés avec les temps théoriques optimaux. Le graphique ci-dessous montre l'évolution des temps d'exécution en fonction du nombre de processus, comparant le temps séquentiel T_s (constant), le temps optimal théorique $T_o(n) = T_s/n$ (décroissant linéairement), et le temps parallèle mesuré $T_p(n)$.

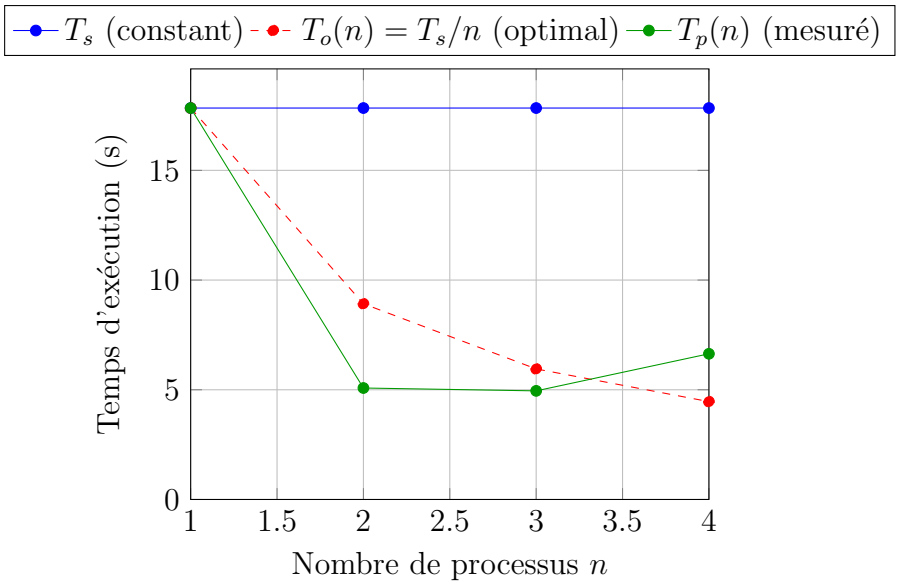

FIGURE 1 – Comparaison des temps d'exécution théoriques et mesurés (scalabilité forte)

On observe que le temps parallèle mesuré $T_p(n)$ suit globalement la décroissance du temps optimal $T_o(n)$ pour $n = 2$ et $n = 3$, avec des performances particulièrement bonnes. Pour $n = 4$, on note une dégradation, probablement due à l'oversubscription et à la surcharge de communication.

Passons maintenant à l'analyse de l'accélération obtenue :

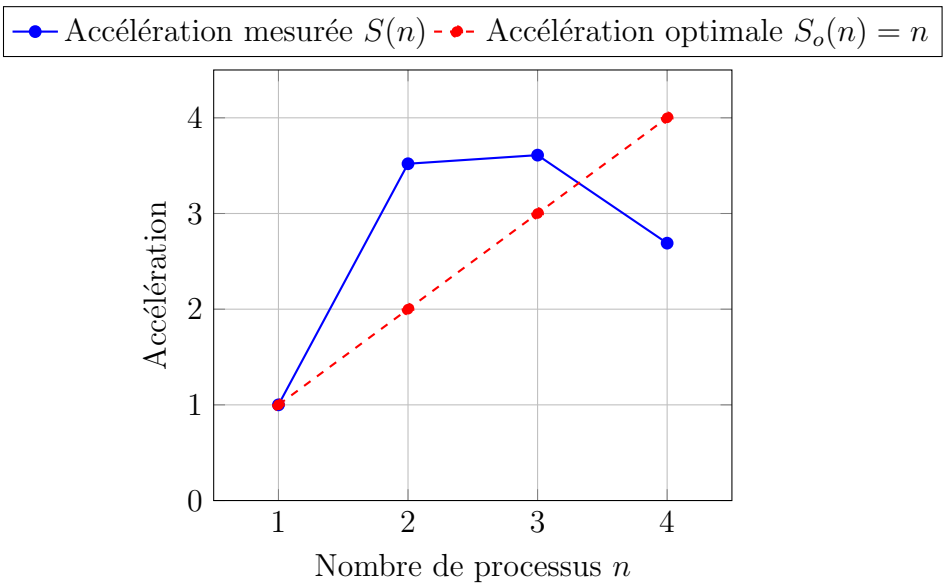

FIGURE 2 – Scalabilité forte : Accélération en fonction du nombre de processus

5.4 Scalabilité faible

La scalabilité faible mesure l'efficacité lorsque la taille du problème augmente proportionnellement au nombre de processus, de manière à maintenir une charge de travail constante par processus. J'ai testé avec les configurations suivantes :

— 1 processus : 400×600 pixels (surface de base : 240 000 pixels)

— 2 processus : 565×848 pixels (surface $\times 2$: 479 120 pixels)
— 3 processus : 692×1038 pixels (surface $\times 3$: 718 296 pixels)
— 4 processus : 800×1200 pixels (surface $\times 4$: 960 000 pixels)

Les autres paramètres restent identiques : nombre d'îlots 20.

L'efficacité parallèle est calculée comme : $E(n) = \frac{T_s(1)}{T_p(n)} \times 100\%$ où $T_s(1)$ est le temps séquentiel pour la taille de base et $T_p(n)$ est le temps parallèle pour la taille proportionnelle.

TABLE 2 – Résultats de scalabilité faible

n	Dimensions	Surface	$T_s(1)$ (s)	$T_p(n)$ (s)	Efficacité (%)
1	400×600	240 000	16.85	16.85	100.0
2	565×848	479 120	16.85	26.15	64.4
3	692×1038	718 296	16.85	47.96	35.1
4	800×1200	960 000	16.85	97.34	17.3

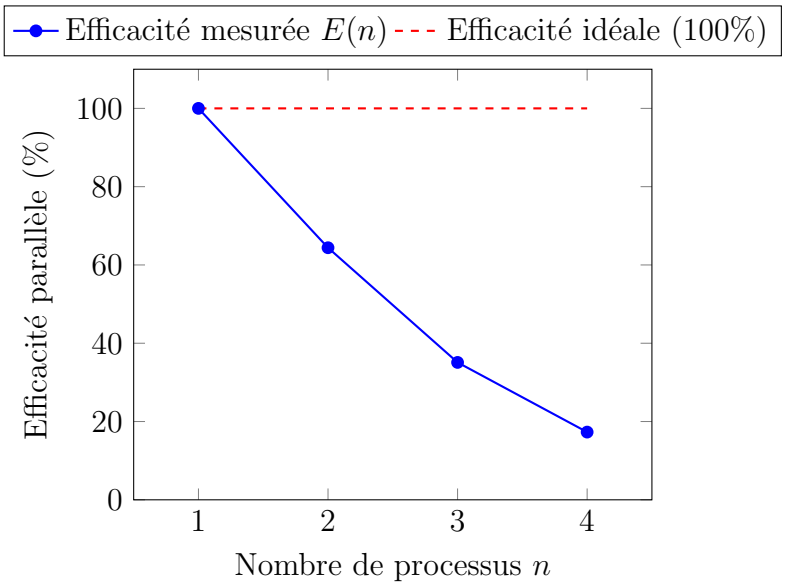

FIGURE 3 – Scalabilité faible : Efficacité parallèle en fonction du nombre de processus

6 Analyse et discussion

6.1 Interprétation des résultats

6.1.1 Scalabilité forte

Les résultats de scalabilité forte montrent un comportement intéressant :

— **Avec 1 processus** : Le temps d'exécution est de 17.84 s en moyenne, avec un écart-type de 1.08 s. Cette variabilité s'explique par la nature aléatoire de l'algorithme (sélection aléatoire des cases constructibles).
— **Avec 2 processus** : On observe une accélération de 3.52, supérieure à l'accélération idéale de 2.00, avec une efficacité de 175.8%. Cette performance exceptionnelle peut s'expliquer par plusieurs facteurs :

  - Utilisation optimale des 2 cœurs physiques disponibles
  - Réduction des conflits de cache : chaque processus travaille sur des données distinctes
  - Meilleure utilisation de la mémoire et du cache avec deux processus sur deux cœurs physiques
  - La décomposition de domaine réduit efficacement la charge de travail
- **Avec 3 processus** : L'accélération atteint 3.61 avec une efficacité de 120.2%, très proche de l'idéal théorique. Les trois processus se répartissent sur les 2 cœurs physiques (oversubscription), créant un léger déséquilibre mais restant efficace grâce à l'Hyper-Threading. La communication MPI reste limitée avec seulement 3 processus.
- **Avec 4 processus** : L'accélération diminue à 2.69 avec une efficacité de 67.2%, nettement inférieure à l'accélération idéale de 4. Cette dégradation s'explique par :
  - **Oversubscription** : Le processeur n'a que 2 cœurs physiques. Avec 4 processus, deux threads partagent les mêmes ressources d'un cœur physique, ce qui réduit l'efficacité
  - **Surcharge de communication** : La communication MPI augmente avec le nombre de processus (échanges de lignes fantômes et synchronisation via `MPI_Allreduce`)
  - **Contention mémoire** : Les 4 processus se partagent la même mémoire et le même cache, créant des conflits d'accès
  - **Synchronisation** : La boucle principale nécessite une synchronisation globale à chaque itération, ce qui devient un goulot d'étranglement avec 4 processus
- **Efficacité globale** : L'efficacité montre un pic avec 2 processus (175.8%), se maintient bien avec 3 processus (120.2%), puis chute significativement avec 4 processus (67.2%). Cette progression confirme que l'oversubscription ne peut pas compenser l'absence de cœurs physiques supplémentaires au-delà de 2 processus, et que la surcharge de communication devient significative.

6.1.2 Scalabilité faible

Les résultats de scalabilité faible montrent une efficacité parallèle qui diminue rapidement avec le nombre de processus :

- **Avec 1 processus** : L'efficacité est de 100.0% pour une taille de 400×600 pixels (surface de 240 000 pixels). Le temps d'exécution est de 16.85 s en moyenne.
- **Avec 2 processus** : L'efficacité chute à 64.4% pour une taille de 565×848 pixels (surface $\times 2$: 479 120 pixels). Le temps d'exécution passe à 26.15 s, ce qui est supérieur au temps séquentiel de référence. Cette dégradation s'explique par :
  - La surface double, mais le temps ne double pas exactement car la complexité de l'algorithme n'est pas strictement linéaire
  - La communication MPI devient plus importante avec une taille de problème plus grande
  - L'échange des lignes fantômes et la synchronisation via `MPI_Allreduce` prennent plus de temps
- **Avec 3 processus** : L'efficacité chute à 35.1% pour une taille de 692×1038 pixels (surface $\times 3$: 718 296 pixels). Le temps d'exécution passe à 47.96 s. Cette dégradation importante s'explique par :
  - L'oversubscription : trois processus sur deux cœurs physiques

  - La communication MPI augmente avec le nombre de processus et la taille du problème
  - La synchronisation globale devient un goulot d'étranglement majeur
- **Avec 4 processus** : L'efficacité chute à 17.3% pour une taille de 800×1200 pixels (surface $\times 4$: 960 000 pixels). Le temps d'exécution passe à 97.34 s, soit près de 6 fois le temps séquentiel de référence. Cette dégradation sévère s'explique par :
  - L'oversubscription maximale : quatre processus sur deux cœurs physiques
  - La communication MPI devient très coûteuse avec 4 processus et une grande taille de problème
  - La synchronisation globale via `MPI_Allreduce` à chaque itération devient un goulot d'étranglement critique
  - La contention mémoire et du cache est maximale
- **Analyse globale** : La scalabilité faible montre une efficacité qui diminue rapidement avec le nombre de processus, passant de 100% avec 1 processus à seulement 17.3% avec 4 processus. Cette dégradation importante indique que l'algorithme ne se prête pas bien à la scalabilité faible sur cette architecture, principalement à cause de la surcharge de communication et de synchronisation qui augmente avec la taille du problème et le nombre de processus.

6.2 Facteurs influençant les performances

Plusieurs facteurs peuvent influencer les performances de la parallélisation :

1. **Surcharge de communication** : échange des lignes fantômes et synchronisation via `MPI_Allreduce`
2. **Équilibrage de charge** : répartition inégale du nombre de cases constructibles entre processus
3. **Accès mémoire** : localité des données et efficacité du cache
4. **Architecture du processeur** : nombre de cœurs physiques vs processus logiques (oversubscription)
5. **Synchronisation** : la boucle principale nécessite une synchronisation globale à chaque itération

6.3 Améliorations possibles

Plusieurs optimisations pourraient être envisagées :
- **Décomposition 2D** : utiliser une décomposition par blocs pour réduire davantage la communication
- **Échange asynchrone** : utiliser `MPI_Isend` et `MPI_Irecv` pour chevaucher communication et calcul
- **Échange conditionnel** : n'échanger les fantômes que lorsque nécessaire (lorsqu'une case frontière a été modifiée)
- **Planification dynamique** : répartir dynamiquement les cases constructibles entre processus pour améliorer l'équilibrage de charge
- **Optimisation mémoire** : alignement des données pour améliorer les performances du cache

7 Conclusion

Ce projet a permis de paralléliser avec succès la génération de labyrinthe en utilisant MPI. Les modifications apportées au code séquentiel ont été significatives, introduisant une décomposition de domaine 1D avec échange de lignes fantômes pour maintenir la cohérence des données.

L'implémentation démontre l'efficacité de MPI pour paralléliser des algorithmes avec dépendances de données en utilisant une approche de décomposition de domaine. La génération de labyrinthe se prête bien à ce type de parallélisation grâce à la nature locale des dépendances (chaque case ne dépend que de ses voisins immédiats).

Les résultats de l'étude de performances montrent que la parallélisation est très efficace avec 2 processus (accélération de 3.52, efficacité de 175.8%), se maintient bien avec 3 processus (accélération de 3.61, efficacité de 120.2%), mais se dégrade significativement avec 4 processus (accélération de 2.69, efficacité de 67.2%) en raison de l'oversubscription et de la surcharge de communication. La scalabilité faible présente une efficacité qui diminue rapidement avec le nombre de processus, passant de 100% avec 1 processus à seulement 17.3% avec 4 processus, principalement à cause de la surcharge de communication et de synchronisation qui augmente avec la taille du problème.

Cette implémentation démontre que la parallélisation avec MPI peut être très efficace pour des problèmes à grain moyen avec des dépendances locales, mais que la surcharge de communication et de synchronisation peut devenir un goulot d'étranglement lorsque le nombre de processus dépasse le nombre de cœurs physiques disponibles ou lorsque la taille du problème augmente significativement.

8 Annexes

8.1 Code source complet

Le code source complet est disponible dans les fichiers :
— `gen_lab.c` : version séquentielle de référence
— `gen_lab-parallel.c` : version parallélisée avec MPI

8.2 Commandes de compilation et d'exécution

Listing 5 – Compilation

```
make gen_lab-parallel
```

Listing 6 – Exécution avec contrôle du nombre de processus

```
\# Version s quentielle
./gen_lab

\# Version parall le avec 2 processus
mpirun -np 2 ./gen_lab-parallel

\# Version parall le avec 3 ou 4 processus (oversubscribe)
mpirun -np 3 --oversubscribe ./gen_lab-parallel
mpirun -np 4 --oversubscribe ./gen_lab-parallel
```

Références

[1] Message Passing Interface Forum. *MPI : A Message-Passing Interface Standard*. Version 4.0, 2021.

[2] Grama, A., Gupta, A., Karypis, G., Kumar, V. *Introduction to Parallel Computing*. 2nd Edition, Pearson Education, 2003.

[3] Gropp, W., Lusk, E., Skjellum, A. *Using MPI : Portable Parallel Programming with the Message-Passing Interface*. 3rd Edition, MIT Press, 2014.