Travaux dirigés n°4

Nicolas RINCON VIJA

nicolas.rincon@ensta.fr

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

Année académique 2024-2025

Table des matières

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.

1 Introduction

Le but de ce travaux dirigé est d'implémenter une version parallèle du *Jeu de la Vie* de Conway en utilisant **MPI** pour gérer la répartition des calculs entre plusieurs processus. Deux stratégies de parallélisation sont considérées :

— **Première approche (2 processus)** : un processus (rang 0) est chargé de dessiner la grille, tandis qu'un second processus (rang 1) effectue les calculs nécessaires à chaque itération.

— **Deuxième approche (N processus)** : le processus 0 s'occupe de l'affichage, mais cette fois, plusieurs processus (rang 1 à N-1) effectuent les calculs en parallèle. Dans cette méthode, la grille est divisée en sous-grilles assignées à chaque processus de calcul.

Ces deux approches permettent d'observer l'impact de la parallélisation sur la performance de la simulation.

2 Le Jeu de la Vie

Le *Jeu de la Vie* est un automate cellulaire qui évolue sur une grille en suivant des règles simples :

— Une cellule vivante avec moins de deux voisins meurt (sous-population).

— Une cellule vivante avec deux ou trois voisins reste vivante.

— Une cellule vivante avec plus de trois voisins meurt (surpopulation).

— Une cellule morte avec exactement trois voisins devient vivante (reproduction).

Dans cette implémentation, nous avons utilisé la bibliothèque **Pygame** pour l'affichage de la grille, et **MPI** pour la gestion du calcul parallèle.

Pour tester et comparer les performances, nous avons utilisé différents **modèles de configuration initiale**. Ces modèles varient en termes de **dimensions de la grille**, de **densité initiale de cellules vivantes** et de **comportement évolutif** au fil des générations. Voici quelques-uns des modèles disponibles :

— *blinker*, *toad*, *acorn*, *beacon*, *boat*, *glider*

— *glider gun*, *space ship*, *die hard*, *pulsar*, *floraison*

— *block switch engine*, *u*, *flat*

Parmi ces modèles, nous avons choisi de tester en détail **acorn** et **u**, car ils présentent des caractéristiques intéressantes qui permettent d'évaluer notre implémentation sous différentes conditions :

— **Acorn** : ce modèle est de dimension **100x100**. Avec seulement 7 cellules initialement vivantes, il génère rapidement une structure complexe qui continue d'évoluer sur un grand nombre d'itérations. Il est intéressant car il met en évidence la dynamique émergente du système et la nécessité d'un calcul efficace.

— **U** : ce modèle est beaucoup plus grand, avec une grille de **200x200**. Il a une faible densité initiale de cellules vivantes, ce qui permet d'observer l'expansion des structures et l'évolution de la simulation sur une échelle plus large. Son évolution est plus lente, nécessitant une gestion efficace du temps de calcul.

Le choix de ces deux modèles nous permet donc d'observer comment notre implémentation réagit à des situations contrastées : une configuration à croissance rapide et complexe (*acorn*), et une configuration plus vaste nécessitant une gestion efficace des ressources de calcul (*u*).

3 Première Approche : 2 processus (Maître-Esclave)

Cette première approche a été implémentée dans le fichier `game_of_life_parallel.py`. Il s'agit d'un modèle de parallélisation simple où :

— **Rang 0 (Maître)** : gère l'affichage de la grille et envoie des requêtes pour récupérer les mises à jour du système.

— **Rang 1 (Esclave)** : effectue les calculs des générations successives et envoie les nouvelles configurations au processus maître.

La communication entre les processus est réalisée à l'aide des fonctions `MPI_Send` et `MPI_Recv`, qui permettent d'échanger efficacement les données entre le processus d'affichage et le processus de calcul. Voici un extrait du code utilisé :

```
if rank == 0:
    while loop:
        comm.send("request_data", dest=1)
        grid.cells = comm.recv(source=1)
        appli.draw()

elif rank == 1:
    while True:
        msg = comm.recv(source=0)
        if msg == "exit":
            break
        grid.compute_next_iteration()
        comm.send(grid.cells, dest=0)
```

3.1 Évaluation

Afin d'évaluer les performances de cette implémentation, nous avons mesuré plusieurs paramètres clés :

— **Nombre total d'itérations** : nombre de générations simulées avant l'arrêt.

— **Temps total de simulation** : durée totale d'exécution.

— **Temps moyen par itération** : temps moyen nécessaire pour calculer et afficher une nouvelle génération.

— **Écart-type du temps par itération** : mesure de la variabilité des temps d'exécution entre les itérations.

— **Nombre moyen de cellules vivantes par itération** : permet d'observer la dynamique du système.

3.2 Résultats obtenus

Nous avons testé cette approche sur les modèles **acorn** et **u**.

3.2.1 Patron : acorn

```
--- Simulation Summary ---
Grid size: 100 rows x 100 columns
Initial number of living cells: 7
Total iterations: 2932
Total simulation time: 14.68s
Average time per iteration: 0.004131s
Iteration time standard deviation: 0.001612s
Average living cells per iteration: 250.95
```

3.2.2 Patron : u

```
--- Simulation Summary ---
Grid size: 200 rows x 200 columns
Initial number of living cells: 13
Total iterations: 4031
Total simulation time: 14.73s
Average time per iteration: 0.002810s
Iteration time standard deviation: 0.000968s
Average living cells per iteration: 258.51
```

3.3 Analyse des résultats

Les résultats obtenus montrent des différences significatives entre les deux configurations testées :

— Le modèle **acorn** génère une dynamique de croissance rapide, augmentant rapidement le nombre de cellules vivantes en raison de son motif initial explosif. Cela se traduit par une moyenne élevée de cellules vivantes par itération (250.95).

— Le modèle **u**, en revanche, présente une évolution plus lente et structurée. Le nombre de cellules vivantes évolue de manière plus stable, mais le grand format de la grille (200x200) rend le calcul plus coûteux à chaque itération.

— En comparant les temps moyens par itération, on observe que **acorn** (0.004131s) est plus rapide que **u** (0.002810s) malgré un plus grand nombre de cellules vivantes en moyenne. Cela est probablement dû à la structure plus dispersée de **u**, qui nécessite plus de calculs pour déterminer l'évolution de la simulation.

Ces résultats montrent que la complexité des interactions cellulaires joue un rôle clé dans les performances de la simulation. Tandis que **acorn** profite de la simplicité de ses règles d'évolution pour maximiser son expansion, **u** est un bon exemple de la difficulté croissante à gérer des grilles de grandes tailles avec une faible densité initiale de cellules vivantes.

4 Seconde Approche : N processus (Maître-Multi-Esclaves)

Dans cette seconde approche, implémentée dans le fichier `game_of_life_parallel_plus.py`, nous avons étendu le modèle maître-esclave pour permettre l'utilisation de plusieurs processus de calcul (**N processus**). Cette fois-ci :

— **Rang 0 (Maître)** : gère l'affichage de la grille et reçoit les mises à jour des processus de calcul.

— **Rangs 1 à N-1 (Esclaves)** : chaque processus effectue une partie des calculs de la grille, en travaillant sur une sous-partie spécifique.

4.1 Gestion des Bordures et Grille Torique

Une des complexités de cette approche réside dans la gestion des **bordures des sous-grilles**. La grille étant **torique** (*Grille torique*), les bords de la grille sont connectés, ce qui signifie que :

— La première ligne de la grille est voisine de la dernière ligne.

— La première colonne est voisine de la dernière colonne.

— Chaque sous-grille calculée par un processus doit échanger ses bordures avec les sous-grilles adjacentes.

L'échange des bordures est réalisé avec des communications `MPI_Sendrecv` entre processus voisins. Voici un extrait du code utilisé :

```
def update_ghost_cells(self):
    req1 = newCom.Irecv(self.cells[-1, :], source=(newCom.rank + 1) % newCom.size, tag=101)
    req2 = newCom.Irecv(self.cells[0, :], source=(newCom.rank - 1) % newCom.size, tag=102)
    newCom.Send(self.cells[-2, :], dest=(newCom.rank + 1) % newCom.size, tag=102)
    newCom.Send(self.cells[1, :], dest=(newCom.rank - 1) % newCom.size, tag=101)
    req1.Wait()
    req2.Wait()
```

Cette méthode garantit que chaque processus dispose des informations correctes sur les cellules voisines, assurant ainsi une simulation correcte.

4.2 Résultats obtenus

Nous avons testé cette approche avec différents nombres de processus (3, 4, 8, 16) sur les modèles **acorn** et **u**.

makecell

4.2.1 Patron : acorn

Nombre de processus	Total d'itérations	Temps total (s)
3	1028	14.66
4	972	14.75
8	719	14.26
16	346	13.37

TABLE 1 – Résultats pour le patron *acorn*

Nombre de processus	Temps moyen par itération (s)	Écart-type (s)	Cellules vivantes moyennes
3	0.012504	0.002788	228.33
4	0.013205	0.002972	230.94
8	0.014904	0.004551	249.41
16	0.024192	0.007419	281.73

4.2.2 Patron : u

Nombre de processus	Total d'itérations	Temps total (s)
3	1277	14.61
4	1194	14.65
8	832	14.36
16	449	13.74

TABLE 2 – Résultats pour le patron *u*

4.3 Analyse des résultats

Nous observons plusieurs tendances intéressantes :

Nombre de processus	Temps moyen par itération (s)	Écart-type (s)	Cellules vivantes moyennes
3	0.009770	0.003363	299.82
4	0.010346	0.003102	298.75
8	0.012231	0.004203	276.73
16	0.016428	0.007462	176.63

— L'augmentation du nombre de processus entraîne une réduction du nombre total d'itérations en 14 secondes de simulation.

— Le temps moyen par itération augmente avec le nombre de processus, indiquant un surcoût de communication entre les processus.

— L'écart-type du temps par itération augmente avec le nombre de processus, ce qui signifie une variabilité accrue dans la durée de chaque mise à jour de la grille.

— Le nombre moyen de cellules vivantes évolue différemment en fonction du modèle choisi, montrant que certaines configurations bénéficient davantage du calcul parallèle.

Une différence notable apparaît entre les résultats du modèle **acorn** et ceux du modèle **u**.

Dans le cas du **modèle acorn** (*100x100*), nous constatons une amélioration des performances en augmentant le nombre de processus jusqu'à un certain point. Cela s'explique par le fait que la grille est suffisamment petite pour que les échanges entre processus restent efficaces, tout en bénéficiant d'une répartition du calcul.

En revanche, dans le cas du **modèle u** (*200x200*), nous remarquons que l'augmentation du nombre de processus au-delà de 8 n'apporte pas d'amélioration, voire ralentit la simulation. Cela est dû au fait qu'une grille plus grande entraîne une fragmentation excessive lorsque le nombre de partitions augmente, augmentant ainsi le coût des communications MPI. La gestion des bordures devient également plus complexe, ce qui peut ralentir le calcul global.

Ainsi, la parallélisation apporte une amélioration notable dans le cas de grilles de taille moyenne, mais sur des grilles plus grandes, l'optimisation du nombre de processus est essentielle pour éviter un surcoût en communication.

5 Conclusion

Ces observations montrent que la parallélisation du *Jeu de la Vie* est efficace lorsque la taille de la grille et le nombre de processus sont équilibrés. Il est essentiel de prendre en compte non seulement la distribution du calcul, mais aussi le coût des communications entre les processus. Une **grille de taille moyenne** bénéficie généralement davantage du parallélisme que les grilles plus grandes, où la fragmentation excessive peut engendrer des pertes de performance.

Ainsi, une stratégie efficace consisterait à adapter dynamiquement la répartition des sous-grilles en fonction du nombre de processus disponibles et des caractéristiques du modèle simulé. Des optimisations supplémentaires, telles que la réduction des communications MPI ou l'utilisation d'un partitionnement adaptatif, pourraient permettre d'améliorer encore les performances de la simulation.