Rapport de Projet C++ : jeu de la vie

Meghna BHAUGEERUTTY, Matthieu MONNOT, Baptiste ZLOCH
Master 272 : Finance Quantitative, Université Paris Dauphine - PSL

Février 2024

1 Introduction

Le "Jeu de la vie", imaginé par John Horton Conway, est un automate cellulaire qui simule l'évolution de cellules sur une grille à deux dimensions. Ce projet implémente le Jeu de la vie en C++ et permet d'observer l'évolution de configurations cellulaires complexes à partir de règles initiales simples au fil des itérations.

2 Fondements mathématiques

Les principes mathématiques sous-jacents dans le Jeu de la Vie sont des règles de transition d'état pour les cellules basées sur le nombre de voisins actifs. Ces règles incarnent des concepts de combinatoire et de théorie des automates cellulaires, qui est une collection discrète de cellules sur une grille de forme carrée. L'évolution du système est déterministe, suivant une progression séquentielle où l'état futur dépend uniquement de la configuration actuelle. Ainsi, chaque cellule a un état (vivante ou morte) et l'état de toutes les cellules est mis à jour simultanément à chaque étape en fonction de règles définies. La règle de base stipule que l'état futur d'une cellule est déterminé par son état actuel et le nombre de voisins vivants qu'elle a.
Les transitions d'état sont dictées par des règles précises :

- Une cellule vivante avec deux ou trois voisins vivants reste vivante à la génération suivante.
- Une cellule morte avec exactement trois voisins vivants devient une cellule vivante.
- Dans tous les autres cas, une cellule vivante meurt ou reste morte.

Ainsi, dans les jeux de ce type, le calcul des voisins d'une cellule est crucial. Mathématiquement, pour une cellule à la position (i, j), ses huit voisins sont aux positions (i-1, j-1), (i-1, j), (i-1, j+1), (i, j-1), (i, j+1), (i+1, j-1), (i+1, j), (i+1, j+1). Le code gère les bords de la grille et évite de compter la cellule elle-même comme son propre voisin. Les mathématiques de la logique conditionnelle sont utilisées pour appliquer les règles de transition. Les conditions *if-else* sont utilisées pour vérifier les règles et mettre à jour l'état de la grille. Les grilles sont représentées par des structures de données bidimensionnelles (`std::vector<std::vector<Cell>>`). L'accès aux éléments et leur mise à jour sont effectués en utilisant des indexation matricielles, qui sont des opérations de base en algèbre linéaire discrète.

3 Analyse détaillée du code

Le code source de notre projet "Jeu de la Vie" est réparti en plusieurs fichiers, distinguant les définitions de classes (*.hpp*) des implémentations (*.cpp*), favorisant ainsi une conception claire et une maintenance aisée. Le code s'articule autour de trois composants majeurs :

- **La classe 'Cell'** *(cell.hpp/cpp)* : La classe 'Cell' incarne l'unité fondamentale de notre automate cellulaire. Elle est dotée de l'attribut 'state' qui reflète l'état actuel de la cellule (vivante ou morte). La méthode `getState()` permet de récupérer cet état, tandis que la méthode `setState(bool)` sert à le modifier, illustrant le principe d'encapsulation en programmation orientée objet. La méthode `toString()` convertit l'état de la cellule en une chaîne de caractères qui représente graphiquement la cellule dans la console, facilitant ainsi la visualisation de la grille durant la simulation. Cette représentation est particulièrement utile pour le débogage et la présentation des résultats intermédiaires de la simulation.
- **La classe 'Game of Life'** *(game_of_life.hpp/cpp)* : La classe 'GameOfLife' est le cœur du mécanisme de simulation. Elle intègre un vecteur bidimensionnel de 'Cell', représentant la grille sur laquelle le Jeu de la Vie se déploie. Le constructeur de cette classe initialise la grille avec une taille spécifiée par l'utilisateur et génère une configuration initiale aléatoire des cellules. La méthode `updateGrid()` met en œuvre les règles de Conway pour transiter d'une génération à la suivante, en se basant sur l'état des voisins calculé par `countNeighbors(int, int)`. Cette dernière est une fonction clé qui détermine le nombre de voisins vivants autour d'une cellule spécifique, impactant directement l'évolution de la grille. Les directives OpenMP intégrées (`#pragma omp`) illustrent l'application de calculs parallèles pour améliorer les performances lors de l'évaluation simultanée de l'état des cellules, ce qui est particulièrement avantageux pour des grilles de grande taille.
- **Le Main** *(main.cpp)* : Le fichier *main.cpp* constitue le point d'entrée de notre application. Il est responsable de l'interprétation des arguments de la ligne de commande, offrant à l'utilisateur la possibilité de spécifier la taille de la grille et le nombre de générations à simuler. Ce fichier illustre également l'importance de la gestion des erreurs et de la validation des entrées, des concepts fondamentaux pour assurer la robustesse de l'application. Après vérification des arguments, il instancie la classe *GameOfLife* et déclenche le processus de simulation.

Le résultat de notre jeu de la vie, avec 100 générations se trouve ci-dessous. Le jeu prend 10.3 secondes à s'exécuter avec 4 coeurs en parallèle.

Figure 1: Représentation du jeu de la vie

De plus, afin d'avoir une représentation visuelle de l'exécution du code, nous avons réalisé un algorigramme de celui-ci :

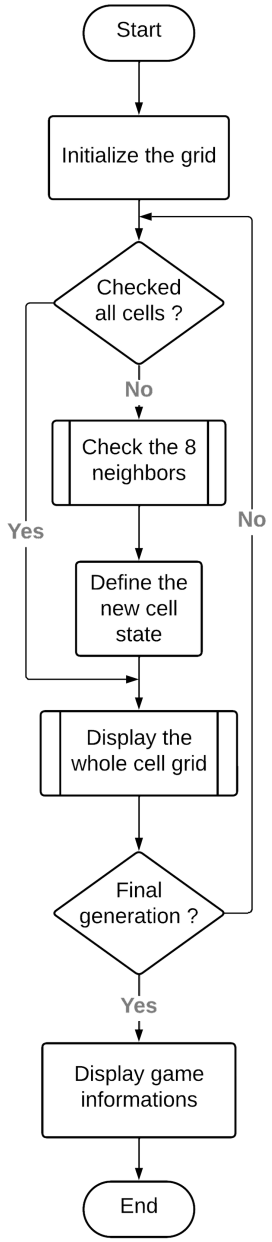

Figure 2: Algorigramme de l'exécution du code

4 Difficultés rencontrées et solutions apportées

4.1 Optimisation de la mémoire

L'un des défis majeurs du projet a été d'optimiser la gestion de la mémoire, une considération critique lors de l'exécution de simulations complexes telles que le Jeu de la Vie. En C++, contrairement à des langages de haut niveau comme Python où la gestion de la mémoire est abstraite, il est nécessaire de gérer explicitement la création et la destruction des objets. Nous avons choisi d'utiliser des pointeurs intelligents pour gérer nos instances de cellules. Cela nous a permis d'éviter les fuites de mémoire, qui auraient pu se produire en allouant dynamiquement de la mémoire sans la libérer correctement. L'utilisation de pointeurs pour modifier directement les références mémoire, au lieu de créer des copies inutiles d'objets, a également amélioré l'efficacité de notre programme en réduisant l'empreinte mémoire et en accélérant l'accès aux données.

4.2 Calcul parallèle

La parallélisation est une technique informatique cruciale utilisée pour accélérer le traitement en exécutant simultanément plusieurs opérations. Dans notre implémentation du "Jeu de la Vie", cette technique est essentielle étant donné la nature calculatoire intensive du processus de mise à jour de la grille, particulièrement pour les grandes dimensions. Pour cela, nous avons utilisé OpenMP (Open Multi-Processing) pour la parallélisation qui a permis de diviser le travail de calcul des voisins vivants entre les différents cœurs du processeur. La directive `#pragma omp` est utilisée pour instruire le compilateur à paralléliser la portion de code qui suit. Dans le contexte de notre classe 'GameOfLife', cette directive est appliquée à la méthode `updateGrid()` :

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < rows; ++i) {
for (int j = 0; j < cols; ++j) {
// Logique pour mettre à jour la grille
}
}
```

La clause `collapse(2)` permet de fusionner les deux boucles for imbriquées en une seule grande itération, rendant possible la parallélisation sur deux dimensions de la grille. Cela signifie que le calcul du nombre de voisins pour différentes cellules peut être réalisé en parallèle, réduisant ainsi le temps nécessaire pour une mise à jour complète de la grille. Voici un algorigramme de l'exécution de notre code en parallèle :

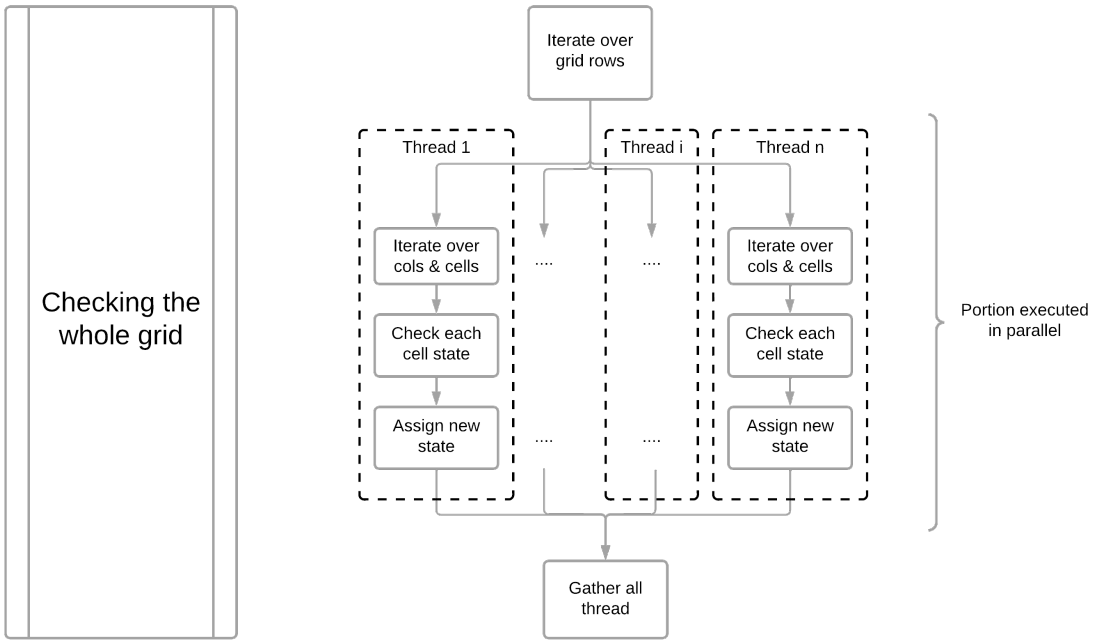

Figure 3: Algorigramme de l'exécution du code en parallèle

La gestion de la concurrence est une considération importante lors de l'utilisation de la parallélisation. Il faut s'assurer que plusieurs threads n'essaient pas de lire ou d'écrire dans la même zone de mémoire simultanément. Pour notre projet, chaque thread travaille sur une portion distincte de la grille, ce qui minimise le risque de conditions de concurrence.
L'effet de la parallélisation sur les performances est significatif, particulièrement pour les grilles de grande taille où le nombre de cellules à évaluer peut être considérable. En distribuant le travail sur plusieurs cœurs de processeur, nous avons constaté une réduction notable du temps d'exécution, rendant l'expérience de simulation plus fluide et plus rapide. Cependant, cela a exigé une compréhension approfondie de la synchronisation des threads et de la gestion des dépendances de données pour éviter les conditions de concurrence et garantir l'intégrité des résultats.

4.3 Rigueur des types et accessibilité des attributs

Contrairement à Python, où le typage est dynamique et tous les attributs sont publics par défaut, C++ impose une discipline stricte en termes de typage et de contrôle d'accès. Nous avons été confrontés à la nécessité de déclarer explicitement les types de toutes nos variables, ce qui a renforcé la sécurité de notre code mais a aussi exigé une vérification minutieuse des types pour éviter les erreurs de compilation. De plus, la gestion correcte de l'accessibilité des attributs (public, protected, private) a nécessité une conception attentive pour s'assurer que seul le code approprié avait accès aux membres de nos classes. Cette rigueur a permis de maintenir l'encapsulation et de promouvoir une architecture logicielle solide et maintenable.

5 Conclusion et perspectives futures

Cette implémentation du Jeu de la Vie en C++ a servi d'application pratique de la programmation orientée objet, des mathématiques discrètes et des concepts de calcul parallèle. Pour les travaux futurs, l'intégration d'une interface utilisateur graphique pourrait rendre la simulation plus accessible.