École Centrale Lyon

UE Java
LaTeXApprofondi
Rapport

Rapport Programmation Concurrente

Élèves :
Paul Hoareau

Enseignant :
Alexandre Saidi

9 novembre 2023

Table des matières

1 Introduction

La programmation concurrente en Java est une discipline qui permet aux développeurs d'écrire des programmes qui exécutent plusieurs tâches en même temps. Cela peut être réalisé en utilisant des threads, qui sont des unités d'exécution légères qui peuvent fonctionner indépendamment.

La programmation concurrente peut être utilisée pour améliorer les performances des applications en divisant une tâche complexe en plusieurs tâches plus petites qui peuvent être exécutées en parallèle. Cela peut se faire pour une variété de tâches.

Ce rapport présente une exploration de cette notion à travers trois exercices pratiques : le calcul de pi, le tri fusion, et le projet GameOfLife.

2 Calcul de *pi*

On effectue une estimation de la valeur de en utilisant la méthode de Monte Carlo. La simulation consiste à générer aléatoirement des points dans un carré unité et à compter combien de ces points se trouvent à l'intérieur du cercle inscrit. La probabilité qu'un point aléatoire soit à l'intérieur du cercle est utilisée pour estimer la valeur de . Le code utilise la programmation multithread pour paralléliser la génération de points.

2.1 Structure du Code

Variables et Initialisation : Le tableau resultats stocke les probabilités locales calculées par chaque thread. Chaque thread, de la classe interne MyThread, génère des points aléatoires et met à jour sa probabilité locale.

2.2 Programmation Concurrente

Threads pour la Génération de Points : Quatre threads sont créés, chacun instancié à partir de la classe interne MyThread, pour générer aléatoirement des points dans le carré unité et mettre à jour la probabilité locale. Chaque thread a un numéro attribué de 1 à 4.

Synchronisation des Threads : La section de code qui met à jour le tableau resultats est protégée par un bloc synchronized (this) pour éviter les conflits d'accès concurrents. Chaque thread met à jour la case correspondant à son numéro dans le tableau.

Calcul de la Probabilité Globale : Une fois que tous les threads ont terminé leur exécution, le programme calcule la probabilité globale en sommant les probabilités locales de chaque thread. Cette probabilité est ensuite utilisée pour estimer la valeur de *pi*.

3 Tri fusion

Le code présenté implémente l'algorithme de tri fusion parallèle, une version optimisée du tri fusion classique qui utilise la programmation multithread pour améliorer les

performances.

3.1 Structure du Code

Méthode Principale (triFusion) : La méthode triFusion effectue le tri fusion parallèle sur une liste d'entiers. Si la taille de la liste est égale à 1 ou moins, la liste est considérée comme triée. Sinon, la liste est divisée en deux parties (gauche et droite) et le tri fusion parallèle est appliqué à chaque partie en utilisant deux threads distincts.

Threads pour Tri Fusion (threadGauche et threadDroite) : Deux threads sont créés, l'un pour trier la partie gauche de la liste et l'autre pour trier la partie droite. Les threads sont démarrés simultanément et attendus pour assurer la synchronisation.

Méthode de Fusion (fusionner) : Une fois que les parties gauche et droite sont triées, la méthode fusionner les fusionne en une seule liste triée. Elle utilise trois indices (indexGauche, indexDroite, et indexListe) pour comparer et fusionner les éléments des deux parties.

3.2 Programmation Concurrente

Threads pour Tri Fusion : L'utilisation de deux threads (threadGauche et threadDroite) permet de paralléliser le tri des parties gauche et droite de la liste.

Synchronisation des Threads : Les threads sont synchronisés à l'aide de la méthode join, qui attend que chaque thread respectif ait terminé son exécution avant de passer à l'étape suivante. Cette synchronisation garantit que la fusion n'est effectuée qu'après que les deux parties ont été triées.

4 GameOfLife

Le code met en œuvre le "Game Of Life" (Jeu de la vie) en utilisant la programmation concurrente en Java. Cette simulation célèbre modélise l'évolution d'une grille bidimensionnelle de cellules, chaque cellule pouvant être soit "vivante" (ALIVE) soit "morte" (DEAD). L'état futur de chaque cellule dépend du nombre de voisines vivantes. Le code utilise des threads pour calculer la prochaine génération de manière concurrente.

4.1 Structure du Code

Variables et Initialisation : Les variables grid, nextGenGrid, et random sont utilisées pour représenter la grille actuelle, la grille de la génération suivante, et l'objet Random respectivement. L'interface graphique est gérée par la classe CellPanel et le bouton "Suivant" est associé à un objet JButton. Un objet CountDownLatch appelé latch est utilisé pour synchroniser les threads.

Interface Graphique : La classe CellPanel étend JPanel pour afficher graphiquement l'état de la grille. Chaque cellule est représentée par un rectangle coloré en noir (DEAD) ou blanc (ALIVE).

4.2 Programmation Concurrente

Threads pour le Calcul : Lorsque le bouton "Suivant" est pressé, quatre threads sont créés pour paralléliser le calcul de la prochaine génération. Chaque thread est responsable d'une section de la grille déterminée par ses coordonnées de départ, sa largeur et sa hauteur. Ceci est réalisé par la classe interne CalculateNextGenTask implémentant l'interface Runnable.

Synchronisation des Threads : La synchronisation entre les threads est assurée par la section de code où la grille de la génération suivante (nextGenGrid) est modifiée. Cette section est protégée par un bloc synchronized (nextGenGrid) pour éviter les conflits d'accès concurrents.

CountDownLatch : L'objet latch est initialisé avec un compte de 4 (le nombre de threads), et chaque thread appelle latch.countDown() pour signaler sa fin d'exécution. La méthode latch.await() est utilisée pour attendre que tous les threads aient terminé avant de mettre à jour la grille principale.

4.3 Calcul de la Génération Suivante

Règles du Game of Life : La méthode calculateNextGenState applique les règles du Jeu de la Vie pour déterminer l'état de chaque cellule à la prochaine génération en fonction de son état actuel et du nombre de voisines vivantes.

Initialisation de la Grille : La méthode initializeGrid initialise la grille principale avec des cellules vivantes et mortes de manière aléatoire.

Comptage des Voisines Vivantes : La méthode countLiveNeighbors calcule le nombre de voisines vivantes d'une cellule en explorant les huit voisins possibles.

4.4 Affichage et Mise à Jour de l'Interface Graphique

Redessin du Panneau : L'interface graphique est redessinée à chaque génération pour refléter les changements dans la grille. Les cellules vivantes sont représentées en blanc et les cellules mortes en noir.

Mise à Jour de la Grille Principale : Une fois que tous les threads ont terminé leur calcul, la grille principale (grid) est mise à jour avec les résultats de la génération suivante, ce qui assure la cohérence des données.

5 Conclusion

En conclusion, ce rapport a exploré plusieurs aspects de la programmation concurrente en Java à travers trois applications pratiques : le calcul de pi par la méthode de Monte Carlo, le tri fusion parallèle, et le "Jeu de la Vie". Chacune de ces applications a permis de mettre en lumière les avantages et les défis de la programmation concurrente dans des contextes différents.

Dans le cas du calcul de , la méthode de Monte Carlo a été mise en œuvre de manière parallèle à l'aide de threads. Cette approche a permis de répartir efficacement le calcul sur plusieurs threads, améliorant ainsi les performances et exploitant le potentiel des processeurs multicœurs.

Le tri fusion parallèle a illustré comment la programmation concurrente peut être appliquée à des algorithmes de tri pour accélérer le traitement des grandes quantités de données. L'utilisation de threads a permis de diviser le travail en tâches indépendantes, augmentant ainsi l'efficacité du tri.

Enfin, le "Jeu de la Vie" a démontré la programmation concurrente dans un contexte de simulation complexe. L'utilisation de threads a permis de paralléliser le calcul de la génération suivante du jeu, offrant un exemple concret de l'application de la concurrence dans des applications interactives.

En analysant ces exemples, nous avons également souligné les défis associés à la programmation concurrente, tels que la synchronisation des threads, la gestion des accès concurrents aux données partagées, et les problèmes potentiels de performance liés à une mauvaise gestion de la concurrence.

En fin de compte, la programmation concurrente en Java offre des solutions puissantes pour améliorer les performances des applications en exploitant le parallélisme offert par les architectures modernes. Cependant, elle nécessite une compréhension approfondie des mécanismes de synchronisation et de gestion des threads pour garantir une exécution correcte et efficace des programmes concurrents.