CENTRALE LYON

École Centrale Lyon

Rapport

BE Java Concurrent

Élèves :
Harish Prabakaran
Sarah Sebbane

Enseignant :
Alexander Saidi

2 novembre 2024

Table des matières

1 Introduction

Ce BE explore des exercices en Java qui appliquent les concepts de la programmation concurrentielle. L'implémentation se concentre principalement sur l'utilisation de **threads** pour paralléliser des calculs, un choix performant sur des processeurs multi-cœurs où chaque cœur peut gérer une tâche distincte simultanément.

2 Calcul de π

Plusieurs méthodes permettent d'approcher la valeur de π. Nous commencerons par la méthode d'approximation utilisant un cercle unitaire, puis nous explorerons des variantes similaires pour comparer leurs principes et leurs performances.

2.1 Méthode par cercle unitaire

L'approximation de π dans cette méthode repose sur la génération de points aléatoires dans un carré de côté 1, puis sur le calcul de la proportion de points tombant dans un quart de cercle de rayon 1. La parallélisation est appliquée ici en répartissant le calcul entre plusieurs threads.

La classe utilise **ThreadLocalRandom** pour générer des nombres aléatoires indépendants dans chaque thread, évitant ainsi les conflits d'accès au générateur de nombres aléatoires. La méthode **run()** exécutée par chaque thread gère sa part du calcul, et une synchronisation est mise en place pour additionner les résultats partiels.

Dans la méthode **main()**, le programme mesure le temps d'exécution total et utilise **join()** pour s'assurer que chaque thread termine son travail avant d'afficher le résultat final.

2.2 Méthodes Bonus

Les méthodes suivantes sont implémentées de manière similaire à la méthode par cercle unitaire.

2.2.1 Méthode par développement de Taylor

Cette méthode utilise la formule de Leibniz, soit le développement en série de Taylor de arctan(1), qui converge lentement vers π. Elle se traduit par la somme alternée des inverses des nombres impairs, mais reste relativement lente en comparaison avec les autres méthodes.

2.2.2 Méthode par série de Bâle

Cette méthode, en sommant les inverses des carrés des entiers, converge plus rapidement vers π que la méthode par développement de Taylor.

2.2.3 Méthode d'espérance

Dans cette méthode, nous générons N valeurs aléatoires pour l'abscisse X d'un point M dans $[0, 1]$ et calculons la moyenne des valeurs prises par $f(X) = \sqrt{1 - X^2}$, représentant l'aire d'un quart de cercle et donc, après normalisation, une approximation de π.

2.3 Résultats - Méthode Cercle unitaire

Pour cette méthode, deux expériences ont été faites : la première avec N = 10^7 points pris, la seconde avec N = 10^8 points pris pour calculer π. Nous avons fait varié le nombre de Threads de 1 à 10. Le programme sera exécuté plusieurs fois pour un Thread (ici 5 fois) dû à la composante aléatoire de l'expérience. Le temps calculé sera seulement une moyenne des résultats obtenus qui sont les temps d'exécution de notre programme en milliseconde.

Voici les résultats :

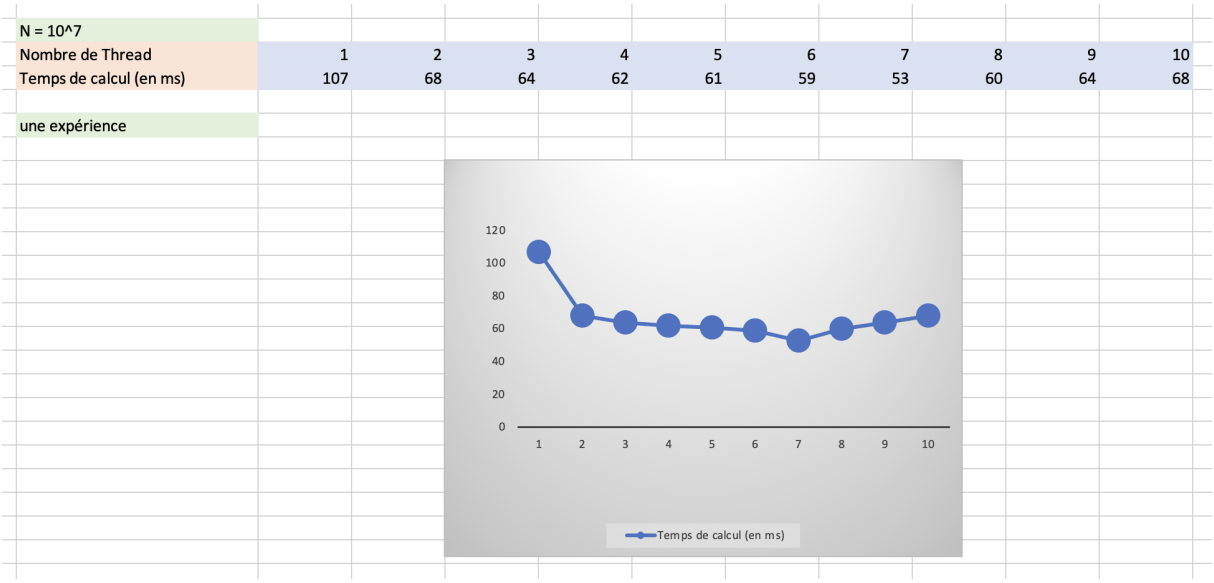

FIGURE 1 – N = 10^7

N = 10^8										
Nombre de Thread	1	2	3	4	5	6	7	8	9	10
Temps de calcul (en ms)	798,8	416,4	287,4	226,4	186,6	159,4	143,2	151,2	143,8	148,6
Moyenne sur 5 expériences										

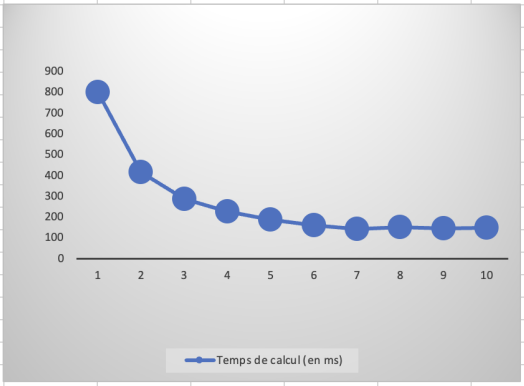

FIGURE 2 – N = 10^8

Pour la première expérience, malgré l'aspect aléatoire de l'expérience, les résultats ne changeaient pas pour le même Thread. Cela peut s'expliquer par un manque de précision dans les résultats, il faudrait mesurer les temps d'exécution en microseconde. Cela se confirme par la seconde expérience dans lesquelles les résultats sont beaucoup plus variés.

Dans les deux expériences, on remarque que le temps d'exécution diminue avec le nombre de Thread jusqu'à 4 Thread utilisés, puis atteint une valeur limite. Ainsi on observe bien qu'il y a un point optimal d'utilisation du nombre de Thread à partir duquel le nombre de Thread n'influence que peu le temps de calcul.

3 Méthode de tri

3.1 Par tri fusion

Le principe de l'algorithme de tri fusion consiste à diviser un tableau T de N éléments en deux sous-tableaux, T_1 et T_2, de tailles à peu près égales. Une fois divisés, ces sous-tableaux sont triés individuellement, puis fusionnés pour reconstruire le tableau original T dans un ordre trié. Comme chaque sous-tableau T_1 et T_2 est déjà trié après la première passe, leur fusion peut être réalisée efficacement en maintenant l'ordre croissant lors de l'assemblage final. Ce processus de division et de fusion se poursuit de manière récursive jusqu'à ce que les sous-sections du tableau soient réduites à des éléments individuels, assurant ainsi un tri efficace et rapide.

Dans l'implémentation proposée, le tableau **listeatrier** est initialisé avec 10 millions de nombres aléatoires. Chaque thread, représenté par une instance de la classe **Tri_Fusion**, est chargé de trier une section distincte du tableau principal. Au démarrage, chaque thread calcule ses indices de départ (**start**) et de fin (**end**) afin de copier ses données dans un tableau local nommé **local_list**. Ce tableau local est ensuite trié en appliquant la méthode **Tri**, qui utilise le tri fusion pour diviser et trier récursivement chaque sous-partie jusqu'à ce qu'elles soient réduites à des éléments individuels, pour ensuite les fusionner

progressivement.

La méthode **fusion** est utilisée pour combiner deux tableaux triés en un seul. Elle compare les éléments des deux tableaux d'entrée et les insère dans un tableau final **result** en maintenant l'ordre croissant. Si des éléments demeurent dans l'un des tableaux après le tri principal, ils sont ajoutés à la fin du tableau final pour compléter la fusion.

Afin de combiner les résultats partiels produits par chaque thread dans **liste_trie**, un bloc synchronisé est utilisé pour garantir qu'un seul thread à la fois accède et met à jour la liste globale. Cela permet d'éviter les conflits d'accès concurrentiel et assure une intégration ordonnée des portions triées dans **liste_trie** au fur et à mesure que les threads terminent leur tâche.

Dans la méthode **main**, une fois tous les threads créés et démarrés, le programme attend leur achèvement en appelant **join()** sur chaque thread. Cette étape est cruciale, car elle garantit que tous les threads ont terminé avant que le programme n'affiche le tableau final trié et le temps total d'exécution.

4 Game of life

4.1 Règles du jeu

Le **Jeu de la Vie** est un automate cellulaire à zéro joueur, fonctionnant selon des règles prédéfinies qui ne nécessitent aucune intervention humaine. La grille à deux dimensions contient des cellules pouvant être "vivantes" ou "mortes", et chaque cellule suit des règles d'évolution selon ses huit voisines :

- Une cellule morte avec trois voisines vivantes devient vivante.
- Une cellule vivante reste vivante si elle a deux ou trois voisines vivantes, sinon elle meurt.

4.2 Diagramme de classes

On choisit de ne représenter que les classes les plus importantes qui permet de réaliser l'interface graphique :

- **AppGameOfLife** : point d'entrée de l'application pour avoir un interface graphique
- **Fenetre** : interface graphique principale
- **Grille** : représentation et gestion de la grille
- **LogiqueDuJeu** : gestion des règles et de l'évolution des cellules du jeu

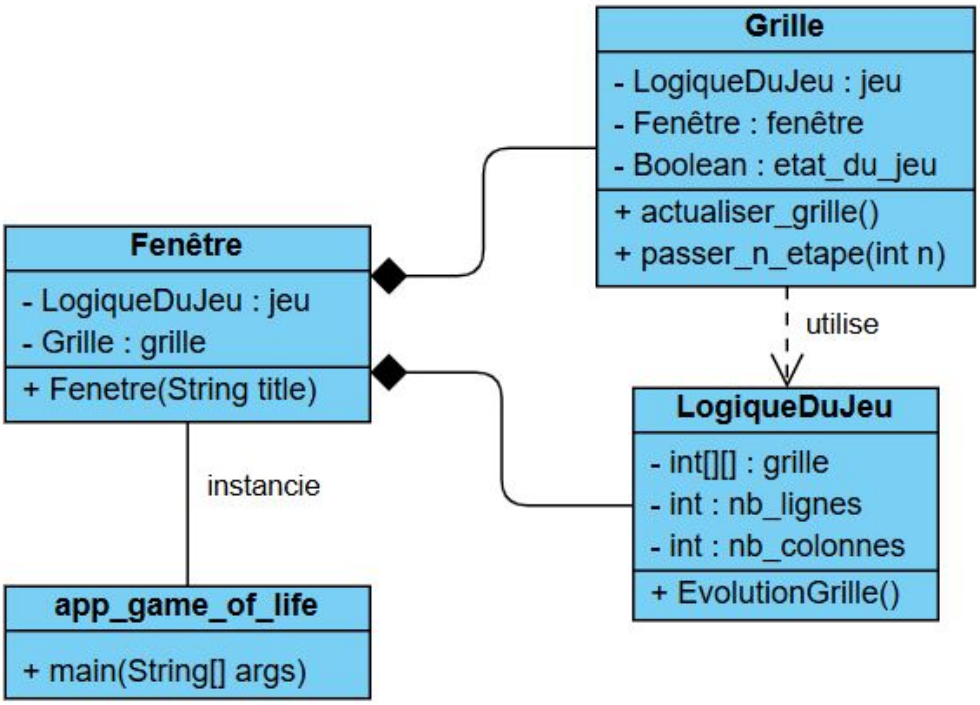

FIGURE 3 – Diagramme UML de l'application

4.3 Implémentation

Dans l'implémentation actuelle, la logique du jeu est dans le code dans la classe "LogiqueDuJeu". La classe "JeuDelaVie" permet de réaliser des tests avec l'utilisation de différents nombres de Thread pour accélérer le temps de calcul des opérations. Le reste des classes permet seulement à l'interface graphique d'être fonctionnelle. Ainsi pour ce BE, il suffirait seulement d'étudier les deux premières classes mais cela pose le problème de savoir si le comportement du jeu est celui attendu, d'où une interface graphique.

Pour avoir un environnement de calcul pas trop complexe, le jeu sera modélisé sur une grille finie. Le jeu évoluera de plus comme si les frontières de la grille sont constamment des cases en vie. C'est un choix qui n'est pas intéressant et important d'analyser sur le déroulement du BE car il est question de temps de calcul. Pour calculer le temps d'execution dans "JeuDeLaVie" avec des simulations variées dans des configurations complexes, la grille de jeu initiale est aléatoire.

4.4 Interface graphique

L'interface graphique permet de vérifier l'exactitude de nos lignes de codes en simulant et représentant graphiquement le jeu de la vie avec l'utilisation d'un seul Thread. Plus précisement, il permet de vérifier si notre programme "LogiqueDuJeu" est exacte, pour ensuite réaliser nos différents tests avec un nombre différent de thread avec "JeuDelaVie". Pour lancer l'interface graphique, il faut exécuter la classe " AppGameOfLife ".

Losque la fenêtre apparait il y a deux boutons : "Nouvelle partie" et "Commencer le jeu". De plus il y a une grille qui représente l'espace où notre jeu évolue.

- "Nouvelle partie" permet de réinitialiser la grille pour n'avoir que des cases blanches qui sont les cases mortes, et arrêter le jeu si il est lancé. (**ATTENTION** : le jeu s'arrête **et** se réinialise.)
- Sur la grille, chaque case est un bouton que l'on peut appuyer. Avant d'appuyer sur "Commencer le jeu", on peut choisir quelles cases seront en vie au début du jeu. En blanc elles sont mortes, en noir elles sont en vie.
- Une fois la configuration établie, il faut appuyer sur "Commencer le jeu", le jeu se lance sur 1000 étapes. La dernière étape est conservée sur la grille une fois la simulation terminée donc il est possible de relancer 1000 étapes encore et ainsi de suite pour voir évoluer le jeu sur plus d'étapes.

5 Conclusion

Ce BE démontre comment la programmation concurrentielle améliore les performances des calculs intensifs et permet la gestion de tâches parallèles comme le tri fusion ou l'approximation de π, qui sont beaucoup plus rapides sur des architectures multi-cœurs. Les threads permettent non seulement de diviser efficacement les tâches mais aussi d'assurer un traitement optimal des ressources du système.