Nom : Francisco **Prénom :** Gaspar da Rosa **Étudiant n° :** 2502991

Rapport de rendu en programmation parallèle

Je vous parle ici des trois implémentations différentes d'algorithmes de backtracking que j'ai réalisés pour résoudre le problème de labyrinthe multi-niveaux. Ces implémentations m'ont permis ainsi de renforcer ma compréhension liée aux avantages et aux défis de la programmation concurrente.

Première Approche : SequentialSolver

L'algorithme explore le labyrinthe **en profondeur (DFS)** depuis le point de départ (D), de manière linéaire.

Fonctionnement :

1. Commence au point de départ
2. Explore récursivement une direction à la fois (nord, sud, est, ouest)
3. Marque les cellules visitées → pour éviter les cycles
4. Collecte les clés et traverse les portes/téléporteurs entre niveaux
5. En cas d'impasse, fait marche arrière (backtracking) et essaie d'autres chemins

L'appel récursif construit **un seul chemin (path)** en explorant chaque possibilité.

Deuxième Approche : ParallelDirectionSolver

La deuxième implémentation parallélise la recherche par direction initiale. L'algorithme lance **un thread par direction** (haut, bas, gauche, droite) à partir du point de départ. Chaque thread explore en profondeur uniquement cette branche initiale.

Fonctionnement :

1. Crée quatre threads, un pour chaque direction cardinale à partir du point de départ
2. Chaque thread possède sa propre copie des niveaux (levels) pour éviter les conflits, et sa propre structure de chemin afin de construire son propre chemin (threadPath).

3. Les threads explorent indépendamment leur branche du labyrinthe
4. Le premier thread qui trouve une solution valide, l'enregistre via un mutex
5. Si plusieurs solutions sont trouvées, la plus courte est conservée

J'ai utilisé '**`mutex pathMutex`**' uniquement pour **protéger l'accès concurrent** à la variable globale **`path`**, dans le cas où plusieurs solutions sont trouvées. Je n'ai pas eu besoin de protéger '**`visited`**' ou '**`maze`**', car **copie locale** de **`levels`**.

L'utilisation de **mutex** et **lock_guard** permet de protéger l'accès au chemin final.

Troisième Approche : ParallelLevelSolver

La troisième implémentation divise le problème par niveau du labyrinthe. Chaque niveau du labyrinthe est **exploré indépendamment dans un thread séparé**.

Fonctionnement :

1. Crée un thread dédié pour chaque niveau du labyrinthe
2. Chaque thread résout son propre niveau indépendamment
3. Les points d'entrée/sortie varient selon le niveau
4. Les threads partagent certaines structures avec protection par mutex
5. Chaque thread remplit un '**localPath**' qui est fusionné dans **path** final si tous les niveaux sont résolus.

J'ai utilisé '**std::mutex mtx**' pour protéger les accès concurrents à **levels[pos.level].visited[...]**, les appels à **resetVisits()** et marquages.

L'utilisation de '**std::lock_guard<std::mutex>**' garantit une libération automatique du verrou.

Ce que j'ai compris

Les trois approches présentent des compromis différents :

1. **Performance :**
    - Sur un système mono-cœur, l'approche séquentielle peut être la plus rapide en évitant le surcoût de synchronisation
    - Sur un système multi-cœurs, les approches parallèles peuvent offrir des accélérations significatives
    - L'approche par direction est généralement plus efficace pour les labyrinthes à longues branches

    - L'approche par niveau excelle quand les niveaux sont relativement indépendants
2. **Consommation mémoire :**
    - L'approche séquentielle a la plus faible empreinte mémoire
    - L'approche par direction duplique toutes les données pour chaque thread
    - L'approche par niveau partage certaines structures avec synchronisation
3. **Évolutivité :**
    - L'approche par direction est limitée à 4 threads (nombre de directions)
    - L'approche par niveau est limitée au nombre de niveaux
    - Une combinaison des deux approches pourrait offrir une meilleure évolutivité

Conclusion

Ces trois implémentations m'ont permis d'apprendre les compromis fondamentaux en programmation concurrente entre :

- La simplicité de conception
- L'utilisation efficace des ressources matérielles
- Les stratégies de partage et synchronisation des données

Le choix de l'approche optimale dépend des caractéristiques du labyrinthe et de l'architecture matérielle cible. Je pense que dans certains cas, une approche hybride combinant différentes stratégies de parallélisation pourrait offrir les meilleures performances.

Je vous remercie de votre attention.