Rapport du Laboratoire 05 : Quicksort Multithreadé

Auteurs : Nicolet Victor, Léon Surbeck

Introduction

Le laboratoire 05 vise à implémenter un algorithme de tri multithreadé basé sur le Quicksort. L'objectif est d'utiliser des threads et un moniteur de type Mesa pour gérer les tâches, en minimisant les coûts de création et de gestion des threads tout en assurant une synchronisation correcte. Ce laboratoire met l'accent sur la gestion de ressources partagées et l'optimisation des performances à l'aide de parallélisme.

Objectifs

- Implémenter l'algorithme Quicksort avec un modèle multithreadé.
- Utiliser des primitives de synchronisation comme `PcoConditionVariable` pour assurer la gestion des tâches.
- Évaluer les performances de la solution en fonction du nombre de threads utilisés.

Choix d'implémentation

Structure Générale

- La classe `Quicksort` hérite de la classe abstraite `MultithreadedSort<T>` et fournit une fonction publique `sort()` pour trier un vecteur d'entiers.
- Le constructeur initialise un pool de threads dont la taille est définie par l'utilisateur.
- Les threads fonctionnent selon un modèle de "workers", où ils attendent qu'une tâche soit disponible, l'exécutent, puis se remettent en attente.

Algorithme Quicksort

Fonctionnement Général

1. **Partitionnement** :
    - L'élément pivot est choisi comme dernier élément du sous-tableau.
    - Les éléments inférieurs au pivot sont déplacés à gauche, et les éléments supérieurs à droite.
    - La fonction retourne l'indice final du pivot.
2. **Division des Tâches** :
    - Les sous-tableaux gauche et droit (par rapport au pivot) sont transformés en tâches distinctes.
    - Ces tâches sont insérées dans une file partagée.

3. **Seuil pour Sous-Tableaux** :
    - Pour des sous-tableaux de petite taille (inférieure à un certain seuil), un tri séquentiel avec `std::sort` est utilisé pour réduire le coût du multithreading.

Synchronisation

- **Gestion des Tâches** :
    - Une file protégée par un `PcoMutex` et une `PcoConditionVariable` est utilisée pour partager les tâches entre les threads.
    - Chaque thread attend qu'une tâche soit disponible ou qu'un signal d'arrêt soit reçu.
- **Comptage des Tâches Actives** :
    - Un compteur atomique `activeTasks` est utilisé pour suivre les tâches en cours.
    - Lorsque ce compteur atteint zéro, tous les threads peuvent être arrêtés.

Tests effectués

1. **Tests de Rapidité / Efficacité** :
    - Vérification que l'algorithme trie correctement des vecteurs de différentes tailles et contenu aléatoire.
    - Validation du fonctionnement pour 1, 2, 4, 8 et 16 threads et que les résultats sont bien meilleurs lorsque le nombre de threads augmente.
2. **Tests de Robustesse** :
    - Manipulation du seuil pour ajuster le basculement entre multithreading et tri séquentiel.
    - Tests avec des données d'entrée déjà triées ou complètement désordonnées.
3. **Analyse des Performances** :
    - Utilisation des benchmarks fournis pour mesurer les temps d'exécution en fonction du nombre de threads.
    - Identification des goulots d'étranglement (par ex. contention sur la file de tâches).

Résultats et Analyse

Performances

Nombre de Threads	Temps (ms)	Accélération
1	2170.35	1×
2	1401.14	1.55×
4	812.36	2.67×
8	675.02	3.21×
16	681.44	3.18×

Observations :

- L'algorithme montre une amélioration des performances jusqu'à 8 threads.
- À partir de 16 threads, les performances stagnent, indiquant des goulots d'étranglement (probablement dus à la contention ou à la surcharge de gestion des threads).
- La meilleure accélération est obtenue avec 8 threads, montrant une efficacité optimale avant que le parallélisme ne devienne limité par d'autres facteurs.

Limites

- La synchronisation par mutex limite l'évolutivité. Une approche lock-free pourrait améliorer les performances.
- La surcharge due au démarrage des threads pourrait être réduite en maintenant un pool de threads persistants.

Conclusion

L'algorithme Quicksort multithreadé implémenté répond aux objectifs pédagogiques en démontrant une gestion efficace des ressources partagées et une parallélisation. Bien que performant jusqu'à un certain nombre de threads, il présente des limitations en termes d'évolutivité dues à la contention et à la gestion des tâches. Ces résultats ouvrent la voie à des améliorations futures, notamment l'utilisation de structures lock-free pour une meilleure utilisation des ressources.