Rapport de Laboratoire de Programmation Concurrente

Informations Générales

- **Auteurs**: Berberat Alex, Surbeck Léon
- **Cours**: Laboratoire de Programmation Concurrente
- **Sujet**: Initiation Thread Management / Bogosort
- **Temps à disposition**: 4 périodes

Objectifs pédagogiques

- Se familiariser avec la gestion des threads
- Stopper élégamment des threads

Étapes appliquées pour le thread manager:

1. Initialisation d'un drapeau booléen `finished` à false, qui sera utilisé pour contrôler l'exécution des threads.
2. Création d'un vecteur `thread_list` pour stocker les pointeurs uniques vers les objets `PcoThread`.
3. Création d'un vecteur `result` pour stocker la séquence triée.
4. Calcul du nombre total de permutations (`total_order`) de la séquence d'entrée `seq`.
5. Division du nombre total de permutations par le nombre de threads (`nbThreads`) pour déterminer la plage de permutations que chaque thread traitera.
6. Création de `nbThreads` threads, chacun responsable du tri d'une partie des permutations en utilisant l'algorithme Bogosort. Chaque thread est ajouté à la `thread_list`.
7. Entrée dans une boucle qui continue jusqu'à ce que le drapeau `finished` soit mis à true par l'un des threads, indiquant que le tri est terminé.
8. Une fois le tri terminé, demande à chaque thread de s'arrêter et attend qu'ils se rejoignent (fin d'exécution).
9. Retourne la séquence triée stockée dans le vecteur `result`.

Étapes appliquées pour les threads de tri :

1. Initialisation de la variable `increment` pour mettre à jour la barre de progression du tri.
2. Initialisation d'un vecteur temporaire `temp` pour stocker les permutations.
3. Boucle sur la plage de permutations assignée au thread (`range_begin` à `range_end`).
4. Vérification si le thread a reçu une demande d'arrêt via `PcoThread::thisThread()->stopRequested()`. Si oui, le thread se termine. Cette vérification se fait dans la boucle pour que si une solution est trouvée pendant le tri, le tri est arrêté immédiatement.
5. Génération de la permutation courante de la séquence en utilisant la fonction `get_permutation`.

6. Vérification si la permutation générée est triée en utilisant `is_sorted`.
7. Si la permutation est triée, mise à jour du vecteur `soluce` avec cette permutation et définition du drapeau `finished` à true dans le `ThreadManager`, puis sortie du thread.
8. Mise à jour du pourcentage de progression du tri en appelant `pManager->incrementPercentComputed(increment)`.

Note : L'algorithme Bogosort est très inefficace et est utilisé ici à des fins de démonstration. Il permute aléatoirement la séquence jusqu'à ce qu'elle soit triée.

Vérifications

Pour vérifier le programme, nous avons testé avec le même nombre de valeurs et la même graine, mais avec un nombre variable de threads.
Le programme fonctionne correctement. Les résultats sont assez imprévisibles en raison de la nature aléatoire de l'algorithme de tri.
Par exemple, pour une même séquence, si avec 3 threads le thread n°2 trouve la solution très rapidement, il est possible qu'en ajoutant plus de threads, la permutation correcte soit trouvée plus tardivement.

Conclusion

Ce laboratoire a permis de se familiariser avec la gestion des threads et l'arrêt élégant de ceux-ci. Bien que l'algorithme Bogosort soit inefficace, il a servi de bon exemple pour illustrer la coordination entre plusieurs threads.
Les résultats obtenus montrent l'impact de la concurrence sur le temps de calcul, soulignant l'importance de la gestion des threads dans les applications concurrentes.
Ce travail a également mis en évidence la nécessité de mécanismes robustes pour arrêter les threads de manière contrôlée et sécurisée.