Calcul différé

Auteurs : Edwin Haeffner, Arthur Junod Date : 28.01.2024

Introduction

Dans ce laboratoire, nous allons implémenter toutes les fonctions qui utilisent un moniteur de Hoare et permettent de réaliser des calculs répartis parmi plusieurs calculateurs différents suivant le type de l'opération à faire.

Ces opérations doivent être ensuite retournées dans l'ordre dans lequel elles ont été demandé.

Choix d'implémentation

Structure de données

Nous avons utilisé ces trois structures de données :

- `std::map` : Pour associer les éléments à leur type de calcul. Cela nous permet de répartir les requêtes vers le bon buffer en fonction du type. Et nous faisons de même pour les conditions d'attente, car chaque buffer peut être soit pleins, soit vides.
- `std::list` : Utilisée pour stocker les requêtes en attente de calcul de chaque type. Au début, nous avions utilisé de simples vecteurs avec des pointeurs d'écriture et de lecture, mais nous nous sommes heurté à des problèmes lié à la synchronisation lorsque nous enlevions des éléments du vecteur. En effet, lorsque nous enlevions un élément du vecteur avec le `abortComputation()`, il était compliqué de faire fonctionner ce paradigme. Donc, nous nous sommes tourné vers les listes, car elles nous permettent d'avoir un ordre simple avec le fait de pouvoir facilement enlever un élément au début de la liste avec la méthode `pop_front()`.
- `std::set` : Finalement, un set pour le buffer de résultats. En effet, nous devions renvoyer les réponses dans l'ordre des IDs, or, elles n'arrivent pas forcément dans cet ordre. Il était plus simple d'utiliser un set qui met automatiquement les éléments dans le bon ordre selon l'ID.

Result

Pour pouvoir utiliser la structure de donnée `std::set` (comme expliqué au-dessus), nous avons dû surcharger les opérateurs `<` et `>` dans la classe `Result`. C'est grâce à ces surcharges qui garantissent l'ordre par l'id dans le set.

waitingId

Nous avons créé une liste d'id, en plus des deux buffers, qui permet de savoir quels sont les ids sur lesquelles le retour des réponses (du buffer de réponses) doit attendre. Cela permet de savoir si on doit encore attendre le résultat d'un calcul, même si les réponses sont déjà triées dans le bon ordre grâce au set.

Test

Tous les tests fournis passent sans problème. Ils nous ont été très utiles pour détecter les interlocks et quelques erreurs d'implémentation.

Nous avons ensuite testé, grâce au gui, de manière visuel l'application. C'est de cette manière que nous nous sommes rendus compte que la première version de notre fonction `abortComputation()` ne marchait pas. On pouvait voir que tous les calculateurs étaient bloqué après l'arrêt d'une opération et que seulement les opérations avec un id plus petit que celle arrêtée pouvaient retourner leur réponse. Cela a été évidemment réglé par la suite comme expliqué plus haut, avec l'utilisation des listes.

Explication par étape

1. Mise en place de la distribution des calculs

La méthode `requestComputation()` permet à un client de soumettre un nouveau calcul. Elle l'ajoute dans le buffer correspondant au type de calcul (A, B ou C). Si le buffer est plein (taille maximale `MAX_TOLERATED_QUEUE_SIZE` atteinte), le thread appelant est bloqué sur la condition `wait(waitFullComputation[type])` jusqu'à ce que de la place se libère dans le buffer.

La méthode `getWork()` est appelée par les calculateurs pour récupérer un travail à effectuer. Elle récupère la tâche la plus ancienne du buffer correspondant au type demandé `bufferComputation[type].front()` et la supprime du buffer. Si le buffer est vide, le thread appelant est bloqué sur la condition `wait(waitEmptyComputation[type])` jusqu'à ce qu'un calcul soit disponible.

2. Mise en place de la gestion des résultats

C'est à partir de cette étape que nous avons rajouté la liste `waitingId`, car nous nous sommes rendus compte que nous devions vérifier l'ordre de retours des résultats en prenant également en compte les opérations non finies. C'est aussi à ce moment que nous avons décidé d'utiliser un `setd::set` pour le buffer des réponses, après avoir testé la `std::priority_queue` et `std::list`.

La méthode `getNextResult()` vérifie si le résultat du calcul le plus ancien (celui avec le plus petit id dans waitingIds) est disponible dans le set `bufferAnswers` : S'il ne l'est pas, le thread appelant

sera bloqué sur la condition `wait(waitAnswer)` jusqu'à ce qu'un nouveau résultat soit disponible.

Lorsqu'un calcul se termine, on signale la condition `waitAnswer` pour réveiller un client en attente.

S'il est disponible, le résultat est retourné et retiré de bufferAnswers. L'identifiant correspondant est également retiré de `waitingId`.

Cela permet aux clients d'attendre de façon synchronisée le résultat du calcul le plus anciens, tout en laissant les autres calculs et leurs résultats s'accumuler dans le buffer en attendant d'être récupérés.

3. Annulation des opérations

Nous avons commencé par retirer les ids de la liste `waitingId` à l'appel de `abortComputation()`. Nous nous sommes ensuite rendu compte qu'il fallait également enlever l'opération dans celles en attentes et celles dont la réponse avait déjà été calculée.

Il faut également signaler les threads qui attendent sur les conditions des buffers afin qu'ils revérifient leur condition notamment pour celle du buffer des réponses. Car si l'opération sur laquelle on attendait a été arrêtée, on doit sauter son attente.

4. Mise en place de la gestion de la terminaison

Afin de pouvoir stopper tous les threads lors de l'arrêt de la logique du programme, nous devons débloquer tous les threads en attente. C'est ce que nous faisons dans la méthode `stop()`.

Ensuite, dans chaques sections de notre programme, il fallait devoir quitter les fonctions si le programme était en état de terminaison. Donc faire un check de la variable `stopped` et quitter la fonction si elle est à true. Certaines fonctions doivent retourner une valeur, ainsi à la place, nous utilisons la fonction `throwStopException()` qui envoie une exception qui est gérée par l'appelant des fonctions ce qui nous permet de sortir de ces fonctions.

Pour certaines fonctions l'appel en cascade de signaux pour débloquer les threads était utile afin de libérer tous les threads en attente.

Conclusion

Grace au moniteur de Hoare, nous avons pu nous rendre compte de la simplicité qu'il apporte dans le code lié à la synchronisation des threads. Le fait de pouvoir simplement utiliser les méthodes `wait` et `signal` sans devoir se soucier de mutex ou bien de niveau de sémaphore est agréable lors de la conception.