HEIG-VD — PCO

Laboratoire 6 – Rapport

Loïc Herman

27 janvier 2024

1 Étape 1 – distribution des calculs

1.1 Conception

Pour stocker les demandes de calcul, nous utilisons une `std::map` qui stockera pour chaque type de calcul une liste doublement chaînée qui permettra le stockage en mode FIFO des demandes tout en offrant les mécanismes d'itération pour les étapes suivantes.

À l'entrée de la réception, on vérifie si la file d'attente peut encore accommoder une demande de plus, sinon on place le thread en attente au moyen des conditions `queueFullCondition` (le nom étant donné par la condition déclenchée). Les conditions `queueFullCondition` et `queueEmptyCondition` ci-après sont définies une fois par type de calcul au moyen d'une `std::map`, ceci pour éviter de signaler des threads qui ne sont pas attribués au type de calcul en questions.

Chaque demande reçoit un id et est ensuite placée dans la file d'attente correspondant au type de calcul. On signale ensuite un potentiel thread qui s'est mis en attente car la file était vide avec la condition `queueEmptyCondition` correspondante.

Quand un calculateur demande un calcul, on vérifie en premier temps s'il y a un calcul à disposition dans la file d'attente correspondante, sinon un wait sur la condition `queueEmptyCondition` est effectué. Une demande démarrée est retirée immédiatement de la file d'attente lorsqu'un calculateur commence à travailler dessus.

1.2 Tests effectués

En plus de la vérification du bon déroulement des tests automatiques, des tests manuels ont été faits dans l'interface pour vérifier que les demandes sont prises dans l'ordre d'arrivée et que les calculateurs commencent bien à travailler dessus.

La fonction d'attente a aussi été vérifiée quand la file d'attente est pleine pour la création des demandes de calculs et quand la file d'attente est vide pour la demande des calculateurs.

La bonne attribution des IDs et le remplissage des deux structures de données a aussi été vérifiée au moyen de l'interface et le debugger.

2 Étape 2 – gestion des résultats

2.1 Conception

Lors de l'enregistrement d'une demande de calcul, le moniteur va automatiquement créer une entrée dans une liste doublement chaînée de paires de résultats et de l'ID correspondant. Les résultats sont wrappés dans un `std::optional` ce qui va nous permettre de déterminer si le calculateur a rendu le résultat ou non. Comme les entrées de la liste sont ajoutées au moment où la demande de calcul est faite, la fonction pour récupérer les résultats peut simplement prendre la première entrée, vérifier si le optional est défini et, s'il l'est alors le résultat est retiré de la liste et rendu au thread appelant, sinon le thread est bloqué au moyen de la condition unique `pendingResultCondition`.

Quand un thread veut fournir un résultat, une itération dans la liste des résultats est effectuée pour retrouver celui qui correspond à l'ID du calcul et l'optional est rempli avec la valeur donnée par le calculateur. La condition `pendingResultCondition` est signalée à ce point au cas où un thread attend un résultat.

La méthode `provideResult` ne tient pas compte de l'ordre des résultats lors du signal, car la méthode `getNextResult` le gère déjà au moyen d'une boucle `while` qui refera la vérification de la présence du résultat à chaque itération.

2.2 Tests effectués

Avec le debugger, nous avons pu vérifier que les optional étaient bien vide et qu'il n'y avait pas d'appel illégal dessus depuis la fonction qui récupère les résultats. Nous avons aussi en plus des tests automatiques vérifiés que les résultats étaient bien livrés dans l'ordre d'arrivée en utilisant l'interface.

Nous avons aussi vérifié avec le debugger que les structures de données étaient correctement nettoyées une fois un résultat obtenu.

3 Étape 3 – annulation de calculs

3.1 Conception

L'annulation d'une demande a été facile à implémenter au moyen des structures de données présentées ci-dessus. Une itération est effectuée sur toutes les files d'attentes des types de calculs et si une demande avec l'ID donné est trouvée, alors elle est retirée.

Une itération et suppression similaire est faite sur la liste des résultats, ce qui permettra à la méthode `continueWork` de retourner `false` si la paire (id, optional) n'est plus trouvée dans la liste.

La méthode d'annulation va ensuite en fonction de ce qui a été retiré des listes faire des signal sur les conditions nécessaires.

3.2 Tests effectués

Après le bon déroulement des tests automatiques, nous avons testé que la méthode d'annulation fonctionne bien dans le cas où le calcul demandé n'est pas trouvé. Nous avons également vérifié le bon fonctionnement que le résultat était bien retiré de la liste si le calcul avait déjà été fini, ainsi que quand le résultat est toujours en cours. Nous avons en outre vérifié qu'une demande en attente ne soit pas démarrée si une demande d'annulation est parvenue avant, en contrôlant également que celui-ci était retiré de la file d'attente.

4 Étape 4 – terminaison

4.1 Conception

Pour la terminaison, chaque appel de `wait` a été précédé et succédé par une vérification sur l'état du moniteur. Si le moniteur est arrêté, alors un signal sur la même condition est fait pour réveiller les threads suivants et l'exception est lancée immédiatement après.

La méthode `stop` va donc mettre le booléen correspondant à `false` et signaler une fois toutes les conditions du moniteur afin de démarrer la chaîne de signal.

4.2 Tests effectués

Nous avons testé avec *valgrind* et le debugger *gdb* que lors de l'arrêt du moniteur toutes les files soient proprement clôturées et que les threads soient tous arrêtés, donc qu'il n'y ait plus de threads en attente. Nous avons vérifié ce cas en ayant des threads en attente sur la file pleine, d'autres sur la file vide (donc deux types de calculs différents), ainsi que plusieurs résultats en attente. Cela nous a permis de vérifier que la chaîne de `signal` soit fonctionnelle et que le moniteur s'arrête dans un état plus ou moins propre pour qu'il soit capable de reprendre plus tard.