PCO Laboratoire 6

Producteur-Consommateur pour calcul différé (Hoare)

Benoît Delay, Eva Ray

January 27, 2024

1 Etape 1

Dans cette étape, nous avons mis en place la distribution des calculs grâce aux méthodes `requestComputation` et `getWork`.

1.1 Choix d'implémentation

Pour représenter le buffer contenant les calculs à réaliser, nous avons choisi d'utiliser une map ayant comme clé le type du calcul et comme valeur une `std::list` contenant les calculs sous forme de `Request` à réaliser de ce type. Notre technique consiste simplement à ajouter les calculs au début de la liste correspondante dans la map dès que `requestComputation` est appelée. Cependant, chaque liste a une taille maximale définie par la constante `MAX_TOLERATED_QUEUE_SIZE`. Pour modéliser cette contrainte, nous avons créé un tableau nommé `fullQueuePerType` contenant une `Condition` pour chaque type de calcul. Avant d'ajouter le calcul à la liste, on vérifie donc que la taille de la liste est inférieure à `MAX_TOLERATED_QUEUE_SIZE`. Si ce n'est pas les cas, on appelle la fonction `wait` sur la condition correspondante dans `fullQueuePerType` et on attend qu'un calcul du bon type arrive dans le buffer. La méthode `requestComputation` est aussi en charge de donner un indice unique à chaque calcul. Pour cela, nous avons utilisé un attribut statique `nextId` qui est incrémenté à chaque fois qu'un calcul est ajouté au buffer.

La méthode `getWork`, quant à elle, demande du travail d'un certain type. Pour cela, elle vérifie que la liste correspondante dans la map n'est pas vide. Si c'est le cas, elle appelle la méthode `pop_back` sur la liste afin de récupérer le dernier élément de la liste, qui sera donc le plus ancien. Si la liste est vide, elle appelle la méthode `wait` sur la condition correspondante dans `emptyQueuePerType` qui est un autre tableau contenant une `Condition` pour chaque type de calcul qui représente le fait qu'il n'y a pas de calcul de ce type dans le buffer. Dans ce cas, on attented donc qu'un calcul de ce type soit ajouté au buffer avant de pouvoir continuer.

Lorsqu'un calcul est ajouté au buffer dans la méthode `requestComputation`, on signale sur la condition correspondante dans `emptyQueuePerType` que le buffer contient maintenant un calcul de ce type. De même, lorsqu'un calcul est retiré du buffer dans la méthode `getWork`, on signale sur la condition correspondante dans `fullQueuePerType` que le buffer contient maintenant un calcul de ce type en moins.

1.2 Tests

2 Etape 2

Dans cette étape nous avons mis en place la gestion des résultats grâce aux méthodes `getNextResult` et `provideResult`.

2.1 Choix d'implémentation

Pour représenter la structure contenant les résultats, nous avons choisi d'utiliser une `std::list` contenant les résultats sous forme de `ResultWithId`. `ResultWithId` est une structure que nous avons rajoutée contenant un id et un résultat qui est optionel (grâce à `std::optional`). Ainis, nous pouvons ajouter le résultat dans la liste directement après avoir envoyé la requête au calculateur, même lorsque le résultat est encore en cours de calcul, ce qui nous permet de connaître l'ordre des calculs en cours ou terminé à tout moment. Dans les faits, un résultat est ajouté à la liste dès que `requestComputation` est appelée.

La méthode `getNextResult` est en charge de récupérer le résultat le plus ancien dans la liste. Pour cela, elle n'as donc qu'à accéder au dernier élément contenu dans la liste. Si la partie optionelle du résultat est vide, cela signifie que le résultat n'est pas encore disponible et donc que le calcul est toujours en cours. Dans ce cas, on appelle la méthode `wait` sur la condition `notExpectedResult`. Cette dernière est une condition qui représente le fait que le prochain résultat n'est pas encore disponible. On attend donc que le résultats attendu soit disponible dans la liste avant de pouvoir continuer. Lorsque c'est le cas, on retire le résultat de la liste et on le retourne.

La méthode `provideResult` permet au calculateur de retourner le résultat du calcul. Pour cela, elle cherche l'id du résultat donné dans la liste `results` et met à jour la partie optionelle du résultat avec le résultat donné. Ensuite, elle signale sur la condition `notExpectedResult` que le prochain résultat est disponible.

2.2 Tests

3 Etape 3

Dans cette étape, nous avons mis en place la possibilité d'annuler des calculs demandés par le client grâce aux méthodes `abortComputation` et `continueWork`.

3.1 Choix d'implémentation

La méthode `abortComputation` permet d'annuler un calcul en cours grâce à son identifiant. Il y a plusieurs cas à gérer. Si le calcul n'est pas encore en cours, il faut simplement le retirer de la map `buffer`. Dans ce cas, il faut signaler sur la condition `fullQueuePerType` que la queue pour ce type de calcul contient un calcul en moins. Si le calcul est en cours, il faut le retirer de la liste `results` et signaler sur la condition `notExpectedResult` pour potentiellement débloqué un thread qui attend sur ce résultat. Si le calcul est terminé, il faut simplement le retirer de la liste `results`. On notera qu'au vu de notre implémentation de `requestComputation`, qui ajoute directement un calcul demandé dans la liste `results`, il faut aussi retirer le calcul de cette liste s'il est demandé mais pas encore en cours.

La méthode `continueWork` et quant à elle assez simple, puisqu'elle va simplement chercher l'id passé en paramètre dans la liste `results`. Si l'id est présent, cela signifie que le calcul doit continuer, sinon, cela signifie que le calcul doit être annulé.

3.2 Tests

4 Etape 4

Dans cette étape, nous avons mis en place la gestion de la terminaison du buffer grâce aux méthodes `stop` et `throwStopException`.

4.1 Choix d'implémentation

La méthode `stop` permet de demander l'arrêt du buffer. Afin de modéliser le fait que le buffer est arrêté, nous avons créé un attribut booléen `stopped` qui est innitialisé à `false` au lancement du programme. Lorsque la méthode `stop` est appelée, on met cet attribut à `true`. De plus, on signale sur toutes les conditions existantes dans le buffer afin de débloquer tous les threads qui attendent sur ces dernières, afin qu'ils puissent s'arrêter.

Pour que l'arrête des threads se fasse comme attendu, nous avons dû apporter quelques modifications aux méthodes préalablement définies. Ainsi, dans toutes les fonctions qui contiennent une attente sur une condition, nous avons ajouté une vérification de l'attribut `stopped` avant et après l'attente sur la condition. Si on se trouve avant l'attente et que l'attribut `stopped` est à `true`, on sort du moniteur et on lance une exception. Si on se trouve après l'attente et que l'attribut `stopped` est à `true`, on signale sur la condition pour débloquer les threads qui attendent dessus, on sort du moniteur et on lance une exception. Cela permet en quelque sorte de réveiller les threads en cascade pour être sûr qu'ils s'arrêtent tous.

La méthode `continueWork` a aussi été modifiée afin de retourner `false` si l'attribut `stopped` est à `true`.

4.2 Tests