Laboratoire 5 : Quicksort multithreadé

Etudiants

- Calum Quinn
- Urs Behrmann

Table des matières

Introduction

Ce laboratoire a pour objectif d'explorer les principes de la programmation concurrente à travers l'implémentation de l'algorithme de tri rapide (Quicksort) en version multithreadée. Ce projet met en pratique la gestion des threads et la synchronisation via un moniteur de Mesa, permettant une répartition efficace des tâches de tri.

Les étapes du travail incluent la conception d'une solution capable d'exploiter un nombre défini de threads, l'intégration de mécanismes de synchronisation, et la vérification de l'efficacité à travers des tests et des benchmarks. Une attention particulière sera portée à la documentation du processus et des choix d'implémentation.

Conception

Pour notre algorithme, nous nous sommes inspirés du modèle producteur-consommateur, où le producteur joue également le rôle de consommateur.

Les étapes utilisées par le processus principal sont les suivantes:

1. Le thread principal crée les threads de travail qui effectueront le tri.
   Ces threads attendent à chaque fois leur tour grâce à un moniteur de Mesa.
2. Le thread principal ajoute la première tâche dans un buffer FIFO et réveil un thread pour qu'il commence à travailler.
3. Le thread principal attend ensuite la fin du travail global à l'aide d'un moniteur dont le seul but est de permettre aux threads de signaler la fin du tri.
4. Lorsque le signal de fin de tri est reçu, le signal d'arrêt est envoyé à tous les threads pour qu'ils se terminent. Le thread principal attend ensuite la fin de chaque thread et se termine lui aussi.

En ce qui concerne les threads qui effectuent réellement le tri, ils ont tous le même fonctionnement:

1. Ils commencent par contrôler si le signal d'arrêt à été lancé et arrête leur exécution si c'est le cas.
2. Sinon, ils attendent un signal du moniteur de Mesa pour indiquer qu'il y a au moins une tâche à faire pour commencer leur travail. Une fois réveillés, ils prennent la première tâche disponible dans le buffer FIFO et démarre la fonction de tri.

3. Dans la fonction de tri, le thread effectue la partition standard du quicksort qui permet de calculer le pivot. Ensuite il ajoute deux tâches au buffer, chacune d'elle contenant une des deux parties du tableau qui sont passées récursivement à la fonction quicksort habituelle. Tous les threads en attente sont ensuite réveillés. (Nous n'en réveillons pas qu'un dû au fait que plusieurs tâches sont ajoutées)
4. Une fois que le thread a ajouté les deux tâches au buffer, il sort de la fonction de tri avant de contrôler si le buffer contient des tâches en attente.
5. Le thread contrôle s'il y a encore des autres threads actifs. Si c'est le cas il recommence sa fonction et attend le signal du moniteur pour continuer.
6. Si le buffer est vide et aucun autre thread n'est encore actif, le thread envoi le signal de terminaison au processus principal.

Choix d'implémentation

Pour l'implémentation des moniteurs nous avons décidé d'utiliser le système producteurs/consommateurs car il se porte bien à l'idée de création et traitement des tâches de tri. Le choix des tâches est tiré directement du pseudo-code du quicksort qui appel récursivement la même fonction avec des sous-parties du tableau original.

Nous avons décidé d'utiliser une `std::queue` pour stocker les tâches car il implémente déjà l'ordre FIFO. Ceci facilite donc l'ajout et la reprise des tâches avec les méthodes de la structure de données.

Tests

On a fait 2 types de tests pour vérifier le bon fonctionnement de notre programme. Le premier groupe représente les tests en rapport avec le bon fonctionnement de l'algorithme de tri. Le deuxième groupe de tests est en rapport avec la gestion des threads, la synchronisation et la concurrence.

Tests de l'algorithme de tri

Pour vérifier que notre algorithme de tri fonctionne correctement, on a fait les tests suivants :

- Tests "simple" de tri avec des tableaux de tailles variées allant jusqu'à `std::numeric_limits<int>::max() / 100` ce qui représente environ 21'474'836 éléments.
    - Taille 1-10
    - Taille 21'474'836-21'474'840
    - Taille 0

    But: *contrôler que la taille du tableau n'a aucun effet sur le bon fonctionnement de l'algorithme de tri*.

    Spécificités: *Il a fallu ajouter le code du test `testSize0` pour implémenter la nécessité d'un array vide à la fin du "tri"*.
- Tests de tri avec des tableaux particuliers.
    - Déjà trié
    - Déjà trié mais en sens inverse
    - Avec des doublons
    - Avec que le même nombre (uniforme)

    But: *contrôler que la forme du tableau, e.g. ordre ou choix des éléments, n'a aucun effet négatif*.

    Spécificités: *Il a fallu ajouter le code du test `testPreGenerated` pour pouvoir contrôler manuellement le contenu du tableau à trier*.

Tests de concurrence

Pour vérifier que notre programme fonctionne correctement en concurrence, on a fait les tests suivants :

- Tests "simple" de tri avec différents nombres de threads (basé sur les quantités utilisées dans les benchmarks).

    - 1 thread
    - 2 threads
    - 4 threads
    - 8 threads
    - 16 threads

  But: *contrôler que le nombre de threads n'a aucun effet sur le bon fonctionnement de l'algorithme de tri*.
- Tests de nombres particuliers de threads.
    - Plus de threads que d'éléments à triés
    - 0 threads
    - 1000 threads (nombre maximum choisi aléatoirement)
    - `std::numeric_limits<int>::max()` threads

  But: *Vérifier que le nombre de threads n'impacte pas le tri correct du tableau*.

  Spécificités: *Le code du test* `testInvalidThreads` *a dû être ajouté afin de gérer les exceptions lancées lors de la construction du* `Quicksort`.

Résultats de nos tests

```
[----------] Global test environment set-up.
[----------] 16 tests from SortingTest
[ RUN      ] SortingTest.Test1Thread
[       OK ] SortingTest.Test1Thread (2 ms)
[ RUN      ] SortingTest.Test2Threads
[       OK ] SortingTest.Test2Threads (0 ms)
[ RUN      ] SortingTest.Test4Threads
[       OK ] SortingTest.Test4Threads (1 ms)
[ RUN      ] SortingTest.Test8Threads
[       OK ] SortingTest.Test8Threads (19 ms)
[ RUN      ] SortingTest.Test16Threads
[       OK ] SortingTest.Test16Threads (23 ms)
[ RUN      ] SortingTest.Test16Threads2
[       OK ] SortingTest.Test16Threads2 (13 ms)
[ RUN      ] SortingTest.TestSize0
[       OK ] SortingTest.TestSize0 (0 ms)
[ RUN      ] SortingTest.TestSmallArrays
[       OK ] SortingTest.TestSmallArrays (18 ms)
[ RUN      ] SortingTest.TestLargeArrays
[       OK ] SortingTest.TestLargeArrays (136611 ms)
[ RUN      ] SortingTest.TestDuplicates
[       OK ] SortingTest.TestDuplicates (2 ms)
[ RUN      ] SortingTest.TestUniformArray
[       OK ] SortingTest.TestUniformArray (5 ms)
[ RUN      ] SortingTest.TestOrderedArray
[       OK ] SortingTest.TestOrderedArray (1 ms)
[ RUN      ] SortingTest.TestInvertedArray
[       OK ] SortingTest.TestInvertedArray (3 ms)
[ RUN      ] SortingTest.Test0Threads
[       OK ] SortingTest.Test0Threads (8 ms)
[ RUN      ] SortingTest.TestMaxThreads
[       OK ] SortingTest.TestMaxThreads (0 ms)
[ RUN      ] SortingTest.TestMaxAllowedThreads
[       OK ] SortingTest.TestMaxAllowedThreads (596 ms)
[----------] 16 tests from SortingTest (137312 ms total)

[----------] Global test environment tear-down
[==========] 16 tests from 1 test suite ran. (137312 ms total)
[  PASSED  ] 16 tests.
```

Benchmark

Résultats des benchmarks

Pour les benchmark, on a fait nos mesures avec 5'000'000 éléments par défaut.

Avec 6 cœurs, on a obtenu les résultats suivants:

```
2024-12-03T21:34:50+01:00
Running ./PCO_LAB05_benchmarks
Run on (6 X 3193.91 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x6)
  L1 Instruction 32 KiB (x6)
  L2 Unified 512 KiB (x6)
  L3 Unified 16384 KiB (x1)
Load Average: 0.71, 0.39, 0.15
-------------------------------------------------------------------------
Benchmark                               Time             CPU   Iterations
-------------------------------------------------------------------------
BM_QS_MANYTHREADS/1/real_time  5035947695 ns    190409419 ns            1
BM_QS_MANYTHREADS/2/real_time  6087301537 ns    240792633 ns            1
BM_QS_MANYTHREADS/4/real_time  8581520662 ns    219756537 ns            1
BM_QS_MANYTHREADS/8/real_time  11169041640 ns    198524880 ns            1
BM_QS_MANYTHREADS/16/real_time 4528929533 ns    199421505 ns            1
```

Avec 8 cœurs, on a obtenu les résultats suivants:

```
2024-12-08T17:07:14+01:00
Running ./PCO_LAB05_benchmarks
Run on (8 X 3600 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x8)
  L1 Instruction 32 KiB (x8)
  L2 Unified 256 KiB (x8)
  L3 Unified 16384 KiB (x8)
Load Average: 0.88, 0.89, 0.50
-------------------------------------------------------------------------
Benchmark                               Time             CPU   Iterations
-------------------------------------------------------------------------
BM_QS_MANYTHREADS/1/real_time  4506367308 ns    179581665 ns            1
BM_QS_MANYTHREADS/2/real_time  5543833629 ns    177624502 ns            1
BM_QS_MANYTHREADS/4/real_time  8743797248 ns    182073802 ns            1
BM_QS_MANYTHREADS/8/real_time  11248056772 ns    175554711 ns            1
BM_QS_MANYTHREADS/16/real_time 6280193388 ns    184946058 ns            1
```

Ensuite on a fait des benchmarks avec 5'000 éléments avec 8 cœurs:

```
2024-12-08T17:11:00+01:00
Running ./PCO_LAB05_benchmarks
Run on (8 X 3600 MHz CPU s)
CPU Caches:
  L1 Data 32 KiB (x8)
  L1 Instruction 32 KiB (x8)
  L2 Unified 256 KiB (x8)
  L3 Unified 16384 KiB (x8)
Load Average: 0.83, 0.84, 0.57
-------------------------------------------------------------------------
Benchmark                               Time             CPU   Iterations
-------------------------------------------------------------------------
BM_QS_MANYTHREADS/1/real_time     4746760 ns       513040 ns          147
BM_QS_MANYTHREADS/2/real_time     9185617 ns       858406 ns           87
BM_QS_MANYTHREADS/4/real_time    11338751 ns       880581 ns           62
```

```
BM_QS_MANYTHREADS/8/real_time    15132855 ns      2417156 ns           42
BM_QS_MANYTHREADS/16/real_time   24591277 ns      3618174 ns           28
```

Analyse des résultats

Il est compliqué de savoir si le code est le plus efficace possible car nous n'avons que pu comparer avec nous même.

L'avantage par contre est que les benchmarks nous permettent de savoir si une version de notre algorithme est plus efficace qu'une autre.

Ce que l'on peut voir c'est que le temps de tri augmente avec le nombre de threads. Cela est probablement dû au fait qu'on a un seul mutex pour la synchronisation des threads. Donc au final, un seul thread peut travailler à la fois.

On a aussi fait des tests avec une version qui crée les tâches et libère ensuite les threads pour qu'ils puissent traiter les tâches. Cette version a été beaucoup plus rapide, car chaque thread a reçu une tâche de taille similaire. Mais cette version n'était pas conforme à l'énoncé du laboratoire d'après notre compréhension, donc nous avons décidé de ne pas la garder.

Conclusion

L'algorithme a été séparée en deux parties principales qui sont le processus principal qui gère les threads de "travail" ainsi que les threads qui effectuent réellement le tri séparé en tâches.

Les moniteurs de type Mesa sont utilisés pour implémenter un système de producteurs/consommateurs pour la gestion des tâches bufferisées.

Les programmes benchmarks et tests nous ont permis de vérifier le bon fonction et l'efficacité de l'algorithme au fil du développement.