Labo 05

Groupe

- Valentin Ricard
- Alexandre Philibert

Introduction au problème

Dans ce laboratoire, il nous est demandé de réaliser une implémentation de l'algorithme Quicksort multithreadé. Pour réaliser cette implémentation, nous devons nous aider d'un moniteur de Mesa.

Ce laboratoire nous permet de mettre en pratique le pattern producteur-consomateur que nous avons pu le voir en cours.

Choix d'implémentation

Lancement des threads

Le constructeur de la classe `Quicksort` prends en paramètres le nombre de threads `nbThreads` qui devront être lancés pour effectuer le tri. Dans notre implémentation, nous commencons tout d'abord par créer ces threads que nous avons nommés `workers`.

Ce pool de threads à pour but de prendre une tâche d'une liste de tâches à effectuer, d'effectuer la tâche, et enfin d'ajouter de nouvelles tâches si cela est nécessaire. Nous pouvons donc facilement observer qu'il s'agit d'une implémentation faisant appel à la notion de producteur-consommateur.

Algorithme Quicksort

En analysant le pseudo-code de l'algorithme quicksort, nous pouvons observer qu'il est composé de plusieurs parties:

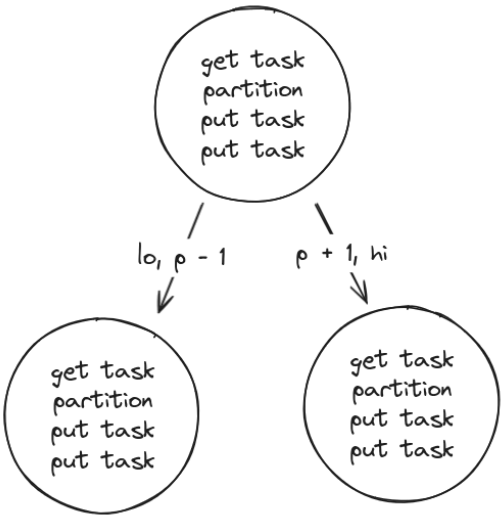
1. Condition d'arrêt
2. Partition et calcul du pivot
3. Appels récursifs

Les opérations de tri se font dans la partition, c'est donc cette partie de l'algorithme que nous souhaitons répartir sur nos différents threads.

Implémentation producteur-consommateur

Un aspect intéressant de notre implémentation est qu'un thread est à la fois un producteur et un consomateur. En effet, un worker thread et un consomateur car il prend une tâche de la liste de tâches dans le but de l'exécuter. Le thread est également un producteur car il peut ajouter de nouvelles tâches à la liste. Dans ce laboratoire, ces tâches sont les appels récursifs à `quicksort` dans le pseudo-code proposé dans la donnée du laboratoire.

Voici une illustration de notre implémentation pour l'exécution d'une tâche :



Cette approche de thread qui porte le rôle de producteur et de consommateur ce manifeste car il nous est nécessaire d'appeler la fonction de partition pour obtenir l'index du pivot. Il y a donc une dépendance entre chaque appel récursif dans l'algorithme de quicksort.

En ce qui concerne l'échange de valeur dans le tableau, il n'est pas nécessaire d'effectuer d'exclusion mutelle. Il n'y a aucun moment ou 2 sections du tableau en cours de tri se chevauchent. Chaque worker thread est assuré d'avoir un accès en lecture et écriture exclusif de par la nature de quicksort.

Moniteur de Mesa

Le moniteur de Mesa nous permet de notifier un thread en attente d'exécution lorsqu'une tâche est disponible dans la liste des tâches. Voici un pseudo-code permettant d'illustrer notre approche:

```
tri() {
  debut_section_critique()
  // Regarde si le worker doit être mis en attente ou non
  tant que (taches.est_vide()) {
    attendre_tache_queue()
  }

  // Récupération de la tâche
  tache = liste_taches.get()
  liste_taches.enlever_premiere()
  fin_section_critique()

  // Algorithme Quicksort
  p = partition()

  debut_section_critique()
  ajouter_tache(...)
  ajouter_tache(...)
  fin_section_critique()

  // On notifie 2x car nous venons d'ajouter 2 tâches à la liste
  notifier_liste_tache_non_vide()
  notifier_liste_tache_non_vide()
}
```

Test effectués

Tests unitaires

En ce qui concerne les tests unitaires, nous avons ajouter plusieurs tests avec différentes tailles, nombre de threads et seeds. Nous avons également ajouter un test qui permet de vérifier qu'une exception est levée si le nombre de threads est 0.

Nous avons aussi un test executant 1000 fois un sort sur un petit array avec plusieurs threads en concurrence afin de s'assurer qu'il n'y a pas de race condition.

Un autre test que nous avons ajouté est le contrôle que les valeurs présentes dans le tableau au début du tri soient également présentes après que le tri soit effectué. Cela permet de vérifier qu'il n'y a pas eu d'écriture concurrentes qui corrompent les valeurs présentes dans le tableau.

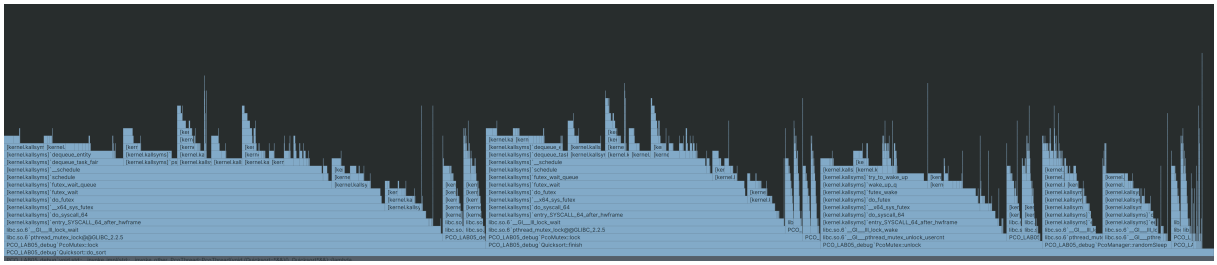
Analyse des performances

Pour analyser les performances de notre implémentation, nous nous sommes aidés de flamegraphs qui nous permettent de relever le temps d'exécution total des différentes méthodes présentes.

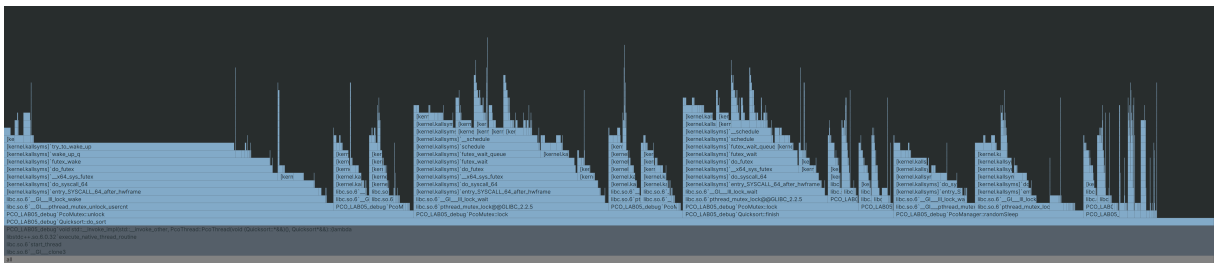
Les flamegraphs ont été réalisés à l'aide du Profiler[1] de CLion, sur Linux 6.6.63.

Les temps d'exécutions moyens ont été calculé sur un total de 30 exécutions sur le même ordinateur.

Flamegraph pour 16 threads et 10 millions de valeurs avec un temps d'exécution moyen de 19 secondes:



Flamegraph pour 2 threads et 10 millions de valeurs avec un temps d'exécution moyen de 11 secondes:

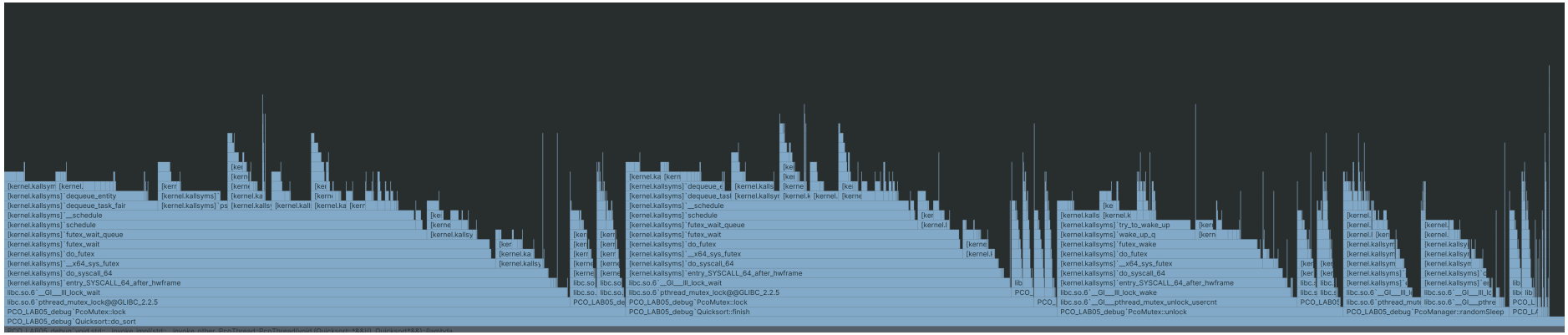


Nous pouvons observer que `PCOMutex::lock()` et `PCOMutex::unlock()` ont un temps d'exécution total plus important sur le flamegraphe de 16 threads. Nous pouvons donc dire qu'il y a probablement une contention sur les mutex présents dans notre implémentation.

Nous pouvons également constater qu'il ne suffit pas d'ajouter des threads à un problèmes pour accelérer le calcul. Il faut prendre en compte les coûts intrinsèques à une implémentation multi-thread ou encore multi-processus.

[1] https://www.jetbrains.com/help/clion/cpu-profiler.html

Flamegraph pour 16 threads et 10 millions de valeurs



Flamegraph pour 2 threads et 10 millions de valeurs