TP 6 : Concurrence

08 et 16 décembre

1 Boites noires

Pour cette partie, on vous fournit :

- Un fichier template.c en langage C qui montre comment lancer trois threads et initialiser, prendre et libérer un mutex ;
- un fichier `boite_noire1.h` qui est une interface pour une bibliothèque.
- un fichier `boite_noire1.o` qui est le fichier compilé correspondant à l'interface décrite ci-haut et dont le code source n'est pas fourni.

Dans cette partie, il est possible d'accéder à des données (par exemples des variables globales ou tableaux) depuis plusieurs threads. Il n'y a pas besoin de faire attention aux données auxquelles on n'accède qu'en lecture (quand leur contenu n'est jamais modifié une fois que les threads sont lancés) mais, pour toute mémoire qui est modifiée après le lancement des threads, tout accès à cette variable (ou case de tableau) que ce soit en lecture ou en écriture, ne peut se faire que depuis un seul thread à la fois. Il est demander de garantir cette unicité d'accès à l'aide de mutex.

Pour compiler vos programmes, il est conseillé d'utiliser la ligne de commande suivante (`main1.c` est le nom de votre fichier C ici et `boite_noire1.o` est le nom de la bibliothèque que l'on veut utiliser dans le premier exercice) :

```
gcc main1.c -Wall boite_noire1.o -lpthread
```

L'objectif de ce premier exercice est de faire un programme C qui fait un appel à la fonction `demarre_boite` puis fait `OBJECTIF` (qui ici vaut 100) appels à la fonction `boite_noire` puis un appel à la fonction `eteint_boite`. Toutes ces fonctions sont définies dans le fichier `boite_noire1.h`.

L'objectif de l'exercice est de faire le code qui fait le plus rapidement possible les 100 appels à la fonction `boite_noire` sachant que chaque appel à `boite_noire()` prend un certain temps. Votre programme peut utiliser plusieurs threads pour accélérer le temps d'exécution mais il ne doit pas utiliser plus de 10 threads (en comptant le thread principal). Pour cet exercice uniquement, on ne connaît pas la durée que met chaque tâche mais on peut supposer qu'une tâche met autant de temps à s'exécuter quel que soit le thread qui l'exécute et quel que soit ce que les autres threads font (peu importe qu'ils travaillent sur une tâche ou non). Les différentes tâches ne prennent pas forcément toutes le même temps à s'exécuter (certaines peuvent durer 5s et d'autres 0.5s).

2 Somme dans un tableau

Ecrire en `C` un programme multi-threadé qui calcule la somme des éléments d'un tableau contenant les premiers entiers consécutifs. Le `main` prendra en argument la taille du tableau et le nombre de threads.

3 Boruvka

Dans cette partie nous allons voir comment paralléliser l'algorithme de Boruvka. Nous utiliserons ici une implémentation qui repose sur la structure union find.

On va paralléliser la recherche des arêtes sures en créant un thread pour chaque composante connexe. Ainsi, contrairement aux implémentations déjà rencontrées, nous allons chercher les arêtes composante par composante et non pas globalement (les solutions que nous avions vues généraient la liste de toutes les arêtes sures alors qu'ici on va écrire une fonction qui trouve l'unique arête sure d'une composante connexe passée en argument.

3.1 Adaptation de la structure union-find

Pour pouvoir faire cela, on a besoin de parcourir une composante connexe donnée ce qui n'est pas vraiment possible efficacement avec la structure `union-find` que nous avons créée en cours. Nous allons donc enrichir cette structure en y ajoutant un tableau des fils qui contiendra dans la case `i` la liste des éléments qui pointent sur le sommet `i` privée de l'élément lui-même lorsqu'il est le représentant de sa classe.

On va donc considérer le type suivant :

```
type uf = {link : int array;
           sons : int list array;
           rank : int array;
           mutable nb : int};;
```

Ecrire les fonctions `create`, `find` et `union` adaptées à cette nouvelle structure. On se référera au fichier `union_find_with_sons.ml` à compléter.

3.2 Recherche des arêtes sures

On va maintenant chercher à écrire une fonction qui cherche l'arête sure dans le graphe `g` d'une composante connexe de la forêt `f`.

On utilisera la type suivant pour représenter les graphes pondérés : à chaque sommet `s`, on associe la liste de ses voisins, chacun associé au poids de l'arête de `s` à ce voisin.

```
type graphe = (int*int) list array
```

Ecrire une fonction `arete_sure_sommet : uf->graphe->int->int*int*int`
telle que `arete_sure_sommet uf g i` renvoie l'arête de poids minimal dans le graphe `g` qui relie le sommet `i` à un sommet qui n'est pas dans la même composante connexe que lui en utilisant `uf` comme représentation de l'ensemble des composantes connexes (la forêt n'est pas passée en entrée explicitement mais uniquement la structure `uf` qui représente ses composantes connexes).

Ecrire une fonction récursive `aretes_sures_ccc: uf->graphe->int*int*int->int->int->int*int*int` telle que `aretes_sures_ccc uf g ar poidsmin i` renvoie l'arête sure (sous forme de triplet : extrémités,poids) dans `g` de la classe de `uf` dont `i` est le représentant. Les variables `ar` et `poidsmin` ont un role d'accumulateur : `ar` sera l'arête ultimement renvoyée et `poidsmin` est le poids de celle-ci que l'on ajoute en argument pour simplifier la lisibilité du code en évitant de devoir déconstruire le triplet `ar`.

En déduire une fonction `aretes_sures_cc : uf->graphe->ref (int*int*int) list->graphe->int->unit` telle que `aretes_sures_cc uf g aretes f i` calcule l'arête sure dans `g` de la composante connexe de `i` dans la forêt `f` dont les composantes connexes sont représentées par `uf` et où on sait que `i` est le représentant de sa classe. La fonction prend de plus en entrée un référence de liste : `aretes` et la met à jour afin de lui ajouter la nouvelle arête sure trouvée, la fonction met aussi à jour la forêt `f` en y ajoutant la nouvelle arête trouvée. Cette fonction fait donc appel à la fonction précédente et procède à une mise à jour de la liste des arêtes sures et de `f`.

3.3 Algorithme de Boruvka

Ecrire une fonction récursive `boruv : graphe -> uf -> graphe -> graphe` telle que `boruv f uf g` renvoie l'arbre couvrant de poids minimal de `g` à partir de la donnée de `f` qui est une forêt sous-graphe de celui-ci et dont les composantes connexes sont représentées par `uf`.

En déduire une fonction `boruvka : graphe->graphe` telle que `boruvka g` renvoie l'arbre couvrant de poids minimal de `g`.

3.4 Parallélisation

Réécrire la fonction `boruv` pour qu'elle lance un thread pour chaque composante connexe plutôt que de calculer les arêtes sures successivement. Il conviendra dans un premier temps de construire un tableau de taille `uf.nb` qui contienne le représentant de chaque classe afin d'associer un thread à chacun d'entres eux.

3.5 Concurrence

A chaque tour, on met à jour la forêt `f` ainsi que la structure associée `uf` pour qu'elle continue de représenter ses composantes connexes. Afin d'éviter des problèmes de concurrence, il convient de sécuriser la mise à jour

de `uf` qui ne doit pas interférer sur les calculs des arêtes sures en cours. L'organisation des étapes de gestion de la structure ainsi que l'utilisation d'un mutex seront nécessaires.

4 Boite noire le retour

Comme dans la partie 1, on dispose des fichiers `boite_noire2.o` et `boite_noire2.h`.

Dans cet exercice on a toujours `OBJECTIF` tâches à accomplir (ici `OBJECTIF` vaut 1000) mais cette fois les tâches ne sont plus indépendantes et l'on a de l'information sur les tâches à accomplir. Plus précisément, on a, pour chaque tâche, une liste de ses prérequis et le temps (en microsecondes) qu'elle prend. Quand une tâche A a une tâche B pour prérequis, il faut que B soit finie avant de pouvoir lancer A. De nouveau, on veut une solution qui finisse l'ensemble des tâches le plus rapidement possible et l'on se donne comme limite de ne pas utiliser plus de deux threads (en comptant le thread principal).

Chaque fonction prend en paramètre un entier entre 0 (inclus) et `OBJECTIF` (exclu) qui correspond à l'identifiant d'une tâche. L'objectif de cet exercice est de proposer des heuristiques qui permettent de résoudre toutes les tâches le plus rapidement possible avec 2 threads. Consulter le fichier `boite_noire2.h` pour une documentation sommaire de la biblothèque.

Prouver que déterminer la solution optimale pour cet exercice est un problème difficile (au sens de la NP-complétude). Pour cela, on pourra utiliser la NP-complétude du problème PARTITION. Le problème PARTITION est le suivant : étant donnés n entiers $v_1, \ldots, v_n$, déterminer s'il existe une partition de $\{1, \ldots, n\}$ en deux ensembles S_1 et S_2 tels que $\sum_{i \in S_1} v_i = \sum_{i \in S_2} v_i$.