TP Concurrence

16 et 23 mars

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 demandé 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 `OCamL` un programme multi-threadé qui calcule la somme des éléments d'un tableau en le partitionnant en parts égales. Vous compléterez le fichier `sum_thread.ml`. On devra être en mesure de modifier la variable `nb_thread`.

3 Automates cellulaires

Un automate cellulaire est un système comprenant des cellules réparties régulièrement. Chaque cellule contient un état, qui représentera l'état actif ou inactif d'une cellule. Ainsi une cellule peut être soit dans l´état inactif (0 pour l'affichage et en C) soit dans l'état actif (1 pour l'affichage et en C). L'évolution dans le temps de l'état de chaque cellule ne dépend que de l'état de ses cellules voisines. Une règle, identique pour toutes les cellules qui constituent l'automate, définit cette évolution.

Le cas le plus simple, et qui sera étudié ici, est le cas appelé 1D. Il s'agit d'une ligne de cellules, chaque cellule ayant deux voisines.

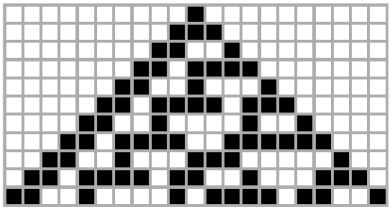
On considère le temps comme étant discret, et l'état d'une cellule au temps t dépend uniquement de son état et de celui de ses voisines au temps $t-1$. On appelle une génération l'état de l'ensemble des cellules à un tel instant t.

On peut ainsi définir de manière univoque l'évolution d'un tel système en utilisant uniquement l'état initial du système et la règle permettant en fonction du voisinage d'obtenir le nouvel état. On appellera cette règle, règle d'évolution.

Par exemple, la table suivante donne un exemple de règle :

Motif initial (t)	111	110	101	100	011	010	001	000
Valeur suivante de la cellule centrale (t+1)	0	0	0	1	1	1	1	0

Dans le cas où une cellule contenant un 1 est environnée de deux 0, elle contiendra toujours un 1 à la génération suivante. Dans le cas où ce 1 est lui même environné de deux 1, elle contiendra un 0 à la place. L'utilisation de cette règle sur une ligne de cellules ne contenant qu'une seule cellule à 1 donnera l'évolution suivante (lecture de haut en bas) :



Dans notre cas, nous allons considérer avoir une ligne de départ de taille fixée, une règle, et un nombre de générations. On considère que les cases à côté des cellules placées aux extrémitées sont évaluées à 0 par défaut.

On modélisera l'état de notre automate par un ruban limité en taille et dont les valeurs au-delà des deux extrémités sont considérées comme nulles.

On utilisera aussi une règle d'évolution définissant le comportement de l'automate cellulaire et prenant trois arguments.

3.1 Implémentation naïve

La version naïve de l'implémentation utilisera le type suivant pour le ruban :

```
short int *ruban
```

Écrire une fonction qui pourra être du type

```
void naive_transitions(short int* ruban,int taille,int nb_tour)
```

qui prend en entrée un ruban, sa taille, et un entier `n` et qui fait évoluer en place l'automate pendant `n` étapes.

On pourra ajouter un affichage.

Vous travaillerez à partir du fichier `aut_cell_naif.h`. Vous devez définir une règle d'évolution, un état initial du ruban, une fonction d'affichage et la fonction `naive_transitions`.

3.2 Implémentation multi-thread

Dans le cas d'un système multi-cœur il est possible d'accélérer les calculs en les répartissant pour les effectuer en parallèle.

Modifiez le code naïf pour effectuer le calcul en parallèle. On pourra commencer par créer 2 threads puis écrire un code qui permet de faire un nombre quelconque de threads, nombre passé en argument du `main`. On pourra alors faire passer la longueur du ruban en argument du `main` aussi afin de procéder à des tests.

Une fois encore, appuyez vous sur le fichier `aut_cell_threade.h` pour la structure de votre code.