Épreuve *Travaux pratiques de programmation* sujet 2022

Évaluation de performance : Automate cellulaire

On s'attendra à ce que les candidats traitent de manière au moins partielle les différentes parties : programmation en OCaml, en C, audit de code.

1 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 (`false` en OCaml, 0 pour l'affichage et en C) soit dans l'état actif (`true` en OCaml, 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.

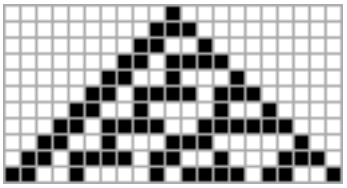
On considère le temps comme étant discret, et l'état d'une cellule au temps t dépend uniquement de son état et de celui de ses voisines au temps $t-1$. On appelle une génération l'état de l'ensemble des cellules à un tel instant t.

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.

Le but de ce TP est d'explorer plusieurs approches pour calculer l'évolution d'un automate cellulaire. On va alors réfléchir à l'impact de chaque approche sur la performance, et ceci de manière globale, i.e. non seulement algorithmique mais aussi lorsque cela fait sens, en tenant compte de l'architecture matérielle.

2 OCaml

Un automate cellulaire borné est la donnée d'un ruban délimité par ses extrémités et d'une fonction de transition `t : bool->bool->bool->bool`.

On pourra par exemple utiliser le type suivant :

```
type automate_cell= {
   ruban : bool list;
   evol : bool->bool->bool->bool}
```

Lorsqu'on veut appliquer la règle d'évolution sur une des deux extrémités du ruban (cases 0 et `longueur_ruban - 1`) alors on considère que les cases en dehors du ruban sont dans l'état `false`.

Voici deux exemples de fonctions d'évolution qui permettront d'illustrer votre travail :

1. On définit $evol\ a\ b\ c = (a + b + c) \mod 2$ (en assimilant ici un booléen à 0 ou 1). On rappelle que $n \mod 2 = 0$ ssi n est un nombre pair et $n \mod 2 = 1$ ssi celui-ci est impair. La fonction OCAML associée est `mod`.
2. On pose `evol 0 0 0 = 0`; et toutes les autres configurations ont pour image 1.

D'autres exemples sont les bienvenus.

Vous pourrez choisir l'état initial du ruban pour illustrer votre travail.

2.1 Une première implémentation

Écrire une fonction `transitions : automate_cell->int->automate_cell` qui prend en entrée un automate cellulaire et un entier `n` et qui renvoie l'automate dans la configuration obtenue après application de la règle d'évolution `n` fois.

On pourra ajouter un affichage de l'évolution de l'automate à chaque étape.

2.2 Implémentation alternative basée sur le pré-calcul d'un automate

On se propose de pré-calculer les transitions à l'aide d'un automate fini.

Pour cela, on va considérer un automate dont les états correspondent aux huit triplets possibles sur l'alphabet $\{0, 1\}$ et chaque transition correspondra à une étape de lecture du ruban (par exemple, si on est dans l'état 100 et qu'on lit un 1 on passera dans l'état 001). Ainsi en suivant le chemin correspondant à la lecture du ruban, l'état courant correspondra aux trois dernières lettres lues sur celui-ci.

Écrire une fonction qui construit cet automate.

Expliquer comment utiliser cet automate pour calculer une étape d'évolution de l'automate cellulaire.

Mettre en œuvre cette stratégie.

2.3 Approche inverse

Nous avons, dans les questions précédentes, implémenté une règle d'évolution donnée.

On va maintenant considérer l'approche inverse : on va chercher une règle qui permettrait de transformer un ruban donné en un autre ruban donné si elle existe. L'enjeu de cette question est de librement proposer une ou plusieurs stratégie(s) répondant à cette question et de discuter de leur(s) pertinence(s) et de leur(s) efficacité(s).

Écrire une fonction qui prend en entrée deux listes composées de `false` et de `true` qui correspondent à deux états successifs du ruban et renvoie `Some f` où f est une règle d'évolution permettant de passer de l'une à l'autre en une étape si cela est possible et `None` sinon. La fonction aura donc comme type de sortie un type option.

On rappelle que pour tout type `'a`, il existe un type `'a option` définit par :

```
type 'a option = None | Some 'a
```

qui permet de distinguer l'existence de la non existence d'un objet.

3 C

La partie précédente a montré une différence entre implémentation naïve et implémentation avancée utilisant une notion de pré-calcul. Le but de cette partie est d'évaluer de manière similaire le coût et le gain d'un pré-calcul.

On utilisera la même modélisation par un ruban limité en taille et dont les valeurs au-delà des deux extrémités sont considérées comme nulles.

On utilisera aussi de manière similaire une règle d'évolution définissant le comportement de l'automate cellulaire et prenant trois arguments. La fonction implémentant cette règle sera réutilisée pour les trois implémentations demandées dans les trois parties ci-dessous.

3.1 Implémentation naïve

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

```
unsigned short int *ruban
```

Écrire une fonction qui pourra être du type
`void naive_transitions(unsigned short int*, const unsigned int, const unsigned int)`
qui prend en entrée un ruban, sa taille, et un entier `n` et qui fait évoluer en place l'automate pendant `n` étapes. La fonction de transition de l'automate peut être rajoutée en tant qu'argument, ou être une fonction globale.

On pourra ajouter un affichage de manière similaire à la version OCaml.

3.2 Implémentation avec pré-calcul de sous-sections

L'utilisation de `unsigned short int` en remplacement de booléens n'est pas particulièrement efficace. Pour rendre le calcul plus efficace, nous allons dans la suite coder le ruban directement en binaire. On pourra par exemple utiliser la représentation suivante :

```
unsigned char *ruban
```

mais où chaque élement représentera 8 cases du ruban. Ainsi `{3, 4}` sous cette forme représentera le ruban `"0000001100000100"`

Ecrire une fonction `transitions` dont le comportement est similaire à la précédente mais en utilisant cette structure de données et un système de pré-calcul.

On pourra se limiter au cas où la longueur du ruban est un multiple de 8.

3.3 Implémentation multi-thread

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

Modifiez le code naïf pour effectuer le calcul en parallèle.

4 Audit de code

Dans cette partie de l'épreuve, il est demandé au candidat d'étudier un fichier source qui comporte des erreurs ou des maladresses, de qualité de code ou de fonctionnement, et il est demandé d'auditer ce fichier, c'est-à-dire :

— de comprendre et d'être capable d'expliquer le fonctionnement du code à l'oral ;
— de proposer des corrections en réécrivant certaines parties afin de corriger les erreurs ou maladresses éventuelles et de rendre le code plus clair, notamment dans une optique pédagogique ;
— de proposer des améliorations de la complexité en temps ou en mémoire, ou de la sûreté du code.

Le temps indicatif de préparation de cette partie est d'une heure.

Le code suivant est sensé être un code d'insertion d'une valeur entière dans un tableau d'entiers triés avec utilisation d'un tableau intermédiaire. On ne s'intéresse qu'à la fonction d'insertion, et on considère que le tableau est effectivement un tableau d'entiers trié.

Ce code est disponible dans votre compte à la racine sous le nom `~/audit.c`.

```
1  // v est la valeur à ajouter dans t (qui est un tableau trié)
2  // taille du tableau T
3  unsigned int inserer_trié(int *t, unsigned int T, int v) {
4    int *tt = malloc(T+1);
5    unsigned int p=0;
6    int d = 0;
7    for(p=0; p<T; p++) {
8      if(t[p] < v)
9        tt[p] = t[p];
10     if(t[p] >= v) {
11       if (d == 0) {
12         d = 1;
13         tt[p] = v;
14       } else tt[p] = t[p+1];
15     }
16   }
17   t = tt;
18   return T+1;
19 }
```

Réference C

pthread_create Créer un nouveau thread

pthread_join Attendre la fin d'un autre thread

pthread_mutex_init Opérations sur les mutex

pthread_mutex_lock Opérations sur les mutex

pthread_mutex_unlock Opérations sur les mutex

pthread_mutex_destroy Opérations sur les mutex