Examen – Cours 2.37.1 –
Programmation des machines multicœur à mémoire partagée

3 mars 2021

Important: *Utiliser des feuilles séparées pour la partie I et la partie II.*

Partie I

Cette partie de l'examen comprend deux exercices indépendants. Tous deux se focalisent sur la programmation parallèle déterministe dans un langage fonctionnel à la ML. On suppose que ce langage a été étendu avec les fonctions du module `Seq` dont l'interface est fournie à la figure 1.

Tri par base parallèle

L'objectif est d'implémenter un tri par base parallèle sur les entiers. Une telle procédure trie une séquence de séquences par ordre lexicographique. Comme on va le voir, elle peut être implémentée plus efficacement qu'un tri par comparaison général. De plus, dans le cas d'une séquence d'entiers machines, l'ordre lexicographique coïncide avec l'ordre usuel, et un tri par base remplace donc avantageusement les algorithmes de tri généraux (tri fusion, tri rapide, etc.).

On suppose dans cet exercice que tous les entiers manipulés sont positifs et codés sur W bits, pour W fixé. On désigne par a_i le $i$ème bit de a, avec $i \in [0, W[$. De plus, on suppose que a_0 est le bit de poids le plus faible de a et a_{W-1} son bit de poids le plus fort. Étant donné deux entiers x et y, on dit que x est *lexicographiquement plus petit* que y, dénoté $x \leq_{lex} y$, ssi il existe $m \in [0, W[$ tel que $x_m \leq y_m$ et que pour tout $i \in [m+1, W[$ on a $x_i = y_i$.

Pour chaque question ci-dessous, vous fournirez du code ML utilisant les fonctions du module `Seq`. Vous spécifierez la complexité en travail et en profondeur de votre code, qui dépendra des séquences parallèles. Tous vos programmes doivent satisfaire le déterminisme interne. Vous êtes fortement encouragés à réutiliser votre code d'une question sur l'autre.

Question 1. Écrire une fonction `extract_bit : int seq -> int -> bool seq` qui, appliquée à une séquence de nombres s de taille n et à un entier k, renvoie une séquence s' de booléens de taille n tel que s'_i correspond au $k$ème bit de s_i. Par exemple, `extract_bit [5; 7; 3; 1; 4; 2; 7; 2] 0` doit s'évaluer vers `[true; true; true; true; false; false; true; false]`.

Question 2. Écrire une fonction `init : int -> (int -> 'a) -> 'a seq` telle que `init n f` renvoie la séquence de longueur `n` dont l'élément d'indice `i` est `f i`. Par exemple, `init 4 (fun x -> 4 * x)` doit s'évaluer en `[0; 4; 8; 12]`.

Question 3. Écrire une fonction `permute : 'a seq -> int seq -> 'a seq` qui, appliqué à une séquence s de taille n et à une séquence d'entiers p représentant une permutation de $[0, n[$, renvoie la séquence s' dont le $i$ème élément est s_j où j est l'unique indice tel que $p_j = i$. Par exemple, `permute [5; 7; 3; 1; 4; 2; 7; 2] [3; 4; 5; 6; 0; 1; 7; 2]` doit s'évaluer en `[4; 2; 2; 5; 7; 3; 1; 7]`.

Question 4. Écrire une fonction `rev : 'a seq -> 'a seq` qui renverse une séquence. Par exemple, `rev [0; 2; 1; 3]` doit s'évaluer en `[3; 1; 2; 0]`.

Question 5. Écrire une fonction `splitp : bool seq -> int seq` qui, appliquée à une séquences booléenne f de longueur n, renvoie une permutation (au sens de `permute`) de $[0, \mathtt{n}[$ qui place les indices i tels que f_i est faux à gauche de tous ceux tels que f_i est vrai. L'ordre à l'intérieur des deux classes d'éléments doit être préservé. Par exemple, l'application `splitp [true; true; true; true; false; false; true; false]` s'évalue en `[3; 4; 5; 6; 0; 1; 7; 2]`.

```
module Seq : sig
  (** [length s] returns the length of [s]. Runs in constant time. *)
  val length : 'a seq -> int

  (** [make x n] is the sequence of length [n] whose elements are all [x]. *)
  val make : 'a -> int -> 'a seq

  (** [read s i] returns the [i]th element of [x]. Runs in constant time. *)
  val read : 'a seq -> int -> 'a

  (** [write s i x] mutates [s] to set its [i]th element to [x]. Runs in
      constant time. *)
  val write : 'a seq -> int -> 'a -> unit

  (** [map f s] applies in parallel [f] to all the elements of [s]. Runs in
      constant span and linear work. *)
  val map : ('a -> 'b) -> 'a seq -> 'b seq

  (** [psum s] returns the sequence [s'] whose element at index [i] is the sum
      of the first [i] elements of [s], e.g., its first element is always [0].
      Runs in constant span and linear work. *)
  val psum : int seq -> int seq

  (** Similar to [map]. All sequences should be of the same length. *)
  val map2 : ('a -> 'b -> 'c) -> 'a seq -> 'b seq -> 'c seq

  (** Similar to [map2]. All sequences should be of the same length. *)
  val map3 : ('a -> 'b -> 'c -> 'd) -> 'a seq -> 'b seq -> 'c seq -> 'd seq
end
```

Figure 1: Un module de séquences parallèles pour ML

Question 6. Écrire une fonction `radix_sort : int seq -> int seq` qui trie une séquence d'entiers. Par exemple, `radix_sort [5; 7; 3; 1; 4; 2; 7; 2]` s'évalue en la séquence `[1; 2; 2; 3; 4; 5; 7; 7]`.

Sémantique de coût

Dans cet exercice, on étudie la sémantique d'un langage doté de primitives constituant une version idéalisée du module `Seq`. Le but est d'établir une sémantique de coût, suivant les principes vus en cours.

Syntaxe. La grammaire du langage est la suivante.

$M, N, P ::=$		Termes
	$\mid\ c$	constante
	$\mid\ \underline{op}(M_1, \ldots, M_n)$	opérateur
	$\mid\ x$	variable
	$\mid\ \mathsf{fun}\, x.M$	fonction anonyme
	$\mid\ M\,N$	application (séquentielle !)
	$\mid\ \mathsf{length}(M)$	`Seq.length M`
	$\mid\ M[N]$	`Seq.read M N`
	$\mid\ M[N] \leftarrow P$	`Seq.write M N P`
	$\mid\ M\,\hat{}\,N$	`Seq.make M N`
	$\mid\ [M\ \mathsf{for}\ x\ \mathsf{in}\ N]$	`Seq.map (fun x -> M) N`
$c ::=$		Constantes
	$\mid\ ()$	unité
	$\mid\ \mathsf{true} \mid \mathsf{false}$	booléen
	$\mid\ \underline{i}$	entier ($i \in \mathbb{Z}/\mathbb{Z}_W$)

Opérateurs. On ne spécifie volontairement pas la liste exacte des opérateurs arithmétiques et logiques. Celle-ci peut comprendre n'importe quel opérateur implémentable en temps constant sur les booléens ou des entiers de taille fixe : addition, soustraction, décalage de bits, comparaison, négation booléenne, etc.

Question 7. Définissez un jugement d'évaluation inductif $M \Downarrow V; w; s$ spécifiant que le terme clos M s'évalue en la valeur V avec le travail w et la profondeur s, où w et s sont des entiers. Vous définirez en particulier la catégorie syntaxique des valeurs. La profondeur et le travail doivent correspondre à ceux spécifiés à la figure 1. *Ne traitez pas la construction $M[N] \leftarrow P$.*

Question 8. Quel problème intervient lorsque vous essayez d'ajouter la construction $M[N] \leftarrow P$ à votre sémantique de coût ? Pouvez-vous ébaucher une solution ? Votre réponse ne devrait pas faire plus d'un paragraphe.

Partie II

Note : Cette partie comprend deux exercices indépendants.

Séquentiellement consistant ou pas ?

Les programmes suivants sont écrits en pseudo-code. Ils suivent nos conventions habituelles. À savoir : `x` et `y` sont des variables partagées, tandis que `r0` et `r1` sont des registres. En outre, `x <- 1` représente une écriture en mémoire et `r0 <- x` une lecture de la mémoire. Toutes les variables et registres contiennent zéro initialement. Un *comportement* est un choix de valeurs finales pour des variables ou registres observés

Figure 2: Quatre petits programmes

```
      Test 1                  Test 2
---------+---------     ---------+---------
   T0    |   T1            T0    |   T1
---------+---------     ---------+---------
 x <- 1  | y <- 1        x <- 1  | x <- 2
 y <- 2  | x <- 2        y <- 2  | y <- 1
---------+---------     ---------+---------
  Observe x,y            Observe: x,y

      Test 3                  Test 4
---------+---------     ---------+---------
   T0    |   T1            T0    |   T1
---------+---------     ---------+---------
 x <- 1  | r0 <- y        x <- 1 | r0 <- x
 y <- 1  | r1 <- x        y <- 1 | r1 <- y
---------+---------     ---------+---------
 Observe r0,r1           Observe r0,r1
```

et indiqués pour chaque test. Plus précisément, `Test 1` et `Test 2` observent les variables partagées `x` et `y`, tandis que `Test 3` et `Test 4` observent les registres `r0` et `r1`.

Un comportement *valide* résulte (1) d'un choix d'appariement entre les évéments de lecture et d'écriture et (2) du choix d'un ordonnancement des écritures de chaque variable. Cela revient à choisir les relations read-from `rf` et cohérence `co` — voir les transparents 12 et 13 de la leçon 03. L'initialisation d'une variable donnée x est représentée par un événement d'écriture de la valeur zéro dans x, qui précède dans `co` toutes les écritures de x effectuées par le programme.

Question 9. Donner tous les comportements valides des tests de la figure 2. Aucune justification n'est demandée.

Question 10. Donner tous les comportements non-séquentiellement consistants des programmes. Autrement dit, donner les comportements *valides* qui ne peuvent *pas* survenir en exécutant les programmes sur une machine qui suit le modèle SC. Aucune justification n'est demandée.

Ticket Locks

Note : Les descriptions des fonctions de librairie dont vous pourrez avoir besoin sont regroupées en fin d'énoncé. La question 14 est jugée difficile, elle peut être gardée pour la fin.

Les *ticket locks* sont des dispositifs d'exclusion mutuelle inspirés des systèmes de gestion de file d'attente que l'on trouve dans les administrations les hôpitaux, les gares, etc[1]. Le système comprend

[1] Et aussi au rayon fromage à la coupe de certains supermarchés et au rayon bois à la coupe de certaines grandes surfaces de bricolage.

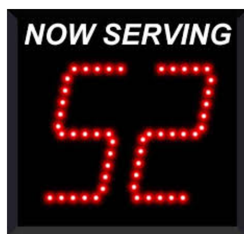

Figure 3: Système de gestion de file d'attente

un distributeur de tickets et un panneau d'affichage, représentés respectivement à droite et à gauche de la figure 3. Les usagers prennent un ticket numéroté lors de leur arrivée et attendent que le panneau affiche le numéro de leur ticket pour se présenter au guichet. On notera que les tickets sont numérotés en séquence et que, quand il est prêt à s'occuper de l'usager suivant, le guichetier appuie simplement sur un bouton dont l'effet est d'incrémenter le numéro affiché. Ce système presente une propriété d'équité : les usagers sont servis dans l'ordre leur arrivée.

Du point de vue de la programmation, les *ticket locks* sont sensés offrier la même fonctionnalité et interface que les *locks* POSIX avec une meilleure équité.

Voici une tentative d'implementation des *ticket locks* en C. Le type `ticket_lock_t` décrit des enregistrements à deux champs. Le champ `next_ticket` correspond au numéro du prochain ticket servi par le distributeur, et `now_serving` au numéro affiché sur le panneau.

```
typedef struct {
  volatile unsigned next_ticket;
  volatile unsigned now_serving;
} ticket_lock_t;
```

```
ticket_lock_t *alloc_ticket_lock(void) {
  ticket_lock_t *p = malloc_check(sizeof(*p));
  p->now_serving = p->next_ticket = 0;
  return p;
}
```

On notera que la fonction de création `alloc_ticket_lock` initialise les deux champs à zéro.

Les sections critiques sont délimités par des appels aux fonctions `lock_ticket` et `unlock_ticket`. Pour entrer en section critique un thread appelle la fonction `lock_ticket` suivante :

```
void lock_ticket(ticket_lock_t *p) {
  unsigned my_ticket = p->next_ticket; // Get ticket
  p->next_ticket = my_ticket+1;        // Increment ticket
  // Wait as long my ticket is not the one being served.
  while (my_ticket != p->now_serving);
}
```

Lors de son arrivée, le thread lit puis incrémente le compteur de tickets (champ `p->next_ticket`). Il compare ensuite son ticket (variable locale `my_ticket`) et celui en cours de service (champ `p->now_serving`). En cas d'égalité, le thread entre entre en section critique en retournant de l'appel à `lock_ticket(...)`. Sinon, un autre thread est en section critique et le thread entre en attente active.

Un thread quitte la section critique en appelant la fonction `unlock_ticket` dont le code est simple :

```
void unlock_ticket(ticket_lock_t *p) {  p->now_serving++; }
```

La section critique est libérée en incrémentant le numéro du ticket en cours de service. L'opération autorise l'accès à la section critique du thread qui détient ou détiendra le numéro nouvellement « affiché ». Par la suite, vous pouvez ignorer les effets résultant de la taille limitée de la représentation des entiers en machine.

```c
volatile int x = 0;
ticket_lock_t *tck;

void *T(void *p) {
  ticket_lock(tck);  x++;  ticket_unlock(tck);
  return NULL;
}

int main(int argc,char **argv) {
  tck = alloc_ticket_lock();
  pthread_t th1,th2;
  create_thread(&th1,T,NULL); create_thread(&th2,T,NULL);
  join_thread(&th1); join_thread(&th2);
  printf("x=%i\n",x);
}
```

Figure 4: Un programme simple pour tester les *ticket locks*.

On considère mainteant le programme de test de la figure 4. Il y a deux variables partagées, `tck` pointeur vers un *ticket lock* et `x` de type **int**.

Question 11. On exécute le programme de la figure 4. Certaines tentatives ne terminent pas. Donner un scénario explicatif.

Question 12. Corriger la fonction `lock_ticket`, afin que le programme affiche toujours 2 *sur une machine SC*.

Question 13. Expliquer qu'une correction similaire de la fonction `unlock_ticket` n'est pas nécessaire.

Question 14. *Question difficile.* On exécute le programme corrigé sur une machine ARM, dont le modèle mémoire est plus faible que SC. On observe que le programme affiche parfois `1`. Donner un scénario explicatif.

Question 15. Corriger les fonctions `lock_ticket` et `unlock_ticket` afin que le programme affiche toujours 2. On suppose que la machine fournit une instruction barrière forte accessible comme l'appel `__sync()`. La correction prime sur l'efficacité.

Question 16. Afin d'ignorer le modèle mémoire de la machine et d'éviter l'attente active, écrire une implémentation des *ticket locks* conforme au modèle DRF à l'aide des mutex et des variables de condition POSIX. Il est rappelé que la correction prime sur l'efficacite.

Annexe : fonctions utiles

Les fonctions présentées ici sont un sous-ensemble de celles utilisées dans le cours. Elles se contentent d'appeler les primitives POSIX correspondantes en se chargeant du contrôle d'erreur qui se borne à arrêter le programme à la première erreur.

"Atomic" builtins

Les « fonctions » ci-dessous donnent accès à des primitives de la machine.

```
/* Strong fence (lesson 02, slide 39)*/
void __sync(void);

/* Read-modify-write operations */
typ __sync_atomic_fetch_and_op(typ *ptr,typ value);
typ __sync_atomic_op_and_fetch(typ *ptr,typ value);
```

Les operations *read-modify-write* sont données comme des schémas ci-dessus, où *typ* est un type scalaire (**char**, **int**, etc.) et *op* est le nom d'une opération (`add`, `or`, etc.). Ces primitives effectuent l'opération `*ptr = *ptr` *op* `value` de façon atomique. Les *builtins* « `fetch_and_`*op* » renvoient la valeur de `*ptr` avant l'opération, tandis que les *builtins* « *op*`_and_fetch` » renvoient le résultat de l'opération. Voir la leçon 2, transparent 5. On suppposera que ces *builtins* commencent et finissent par l'exécution d'une barrière forte (*strong fence*).

Threads (leçon 01, transparent 13)

```
/* Asynchronously excute f(x) on a fresh thread.
   Thread idedentity is stored in th. */
void create_thread(pthread_t *th,void *(*f)(void *),void *x) ;

/* Wait for the thread whose identity is stored in th to terminate */
void *join_thread(pthread_t *th) ;
```

Mutex (leçon 01, transparent 33)

```
/* Allocate  and perform any POSIX-level initialisation  */
pthread_mutex_t *alloc_mutex(void) ;

/* Lock and unlock */
void lock_mutex(pthread_mutex_t *p) ;
void unlock_mutex(pthread_mutex_t *p) ;
```

Variables de condition (leçon 01, transparent 54)

```
/* Allocate  and perform any POSIX-level initialisation  */
pthread_cond_t *alloc_cond(void) ;

/* Suspend (wait) on c, unlocking mutex m */
void wait_cond(pthread_cond_t *c, pthread_mutex_t *m) ;

/* Awake threads suspended on c */
void signal_cond(pthread_cond_t *c) ;      // One thread
void broadcast_cond(pthread_cond_t *c) ;   // All threads
```