TD9

December 5, 2023

1 Programmation fonctionnelle

1.1 TD 9 : Programmation multicoeur

1.1.1 Etienne Lozes - 2021

1.2 Exercice 1 : Map-reduce

1. En utilisant le patron map-reduce, définissez les fonctions suivantes

- `norme: float array -> float`
- `plus_proche_voisin: ~distance:('a -> 'a -> float) -> 'a -> 'a array -> 'a`
- `filter: ('a -> bool) -> 'a array -> 'a array`

- `ecart_type: float array -> float`

2. En utilisant la librairie `Domainslib`, définissez une fonction `ultra_parallel_map_reduce: Task.pool -> ('a -> 'b) -> ('b -> 'b -> 'b) -> 'b -> 'a array -> 'b` qui crée une tâche par case du tableau. Vous pouvez vous inspirer du comptage de nombres premiers vu en cours.
3. Définissez une fonction `chunk_parallel_map_reduce: Task.pool -> ~nb_tasks:int -> ('a -> 'b) -> ('b -> 'b -> 'b) -> 'b -> 'a array -> 'b` qui crée `nb_tasks` tâches d'efforts équivalents.

1.3 Exercice 2 : Data races

On considère l'implémentation suivante de map-reduce, légèrement différente de celle vue en cours.

[ ]:
```
let parallel_map_reduce f (++) _0 array =

  let n = Array.length array in

  let acc = ref _0 in

  let map_reduce_on_slice from upto =
    for i = from to upto - 1 do acc := !acc ++ f array.(i) done in

  let d1 = Domain.spawn (fun () -> map_reduce_on_slice 0 (n/2)) in
```

```
  let d2 = Domain.spawn (fun () -> map_reduce_on_slice (n/2) n) in
  Domain.join d1 ; Domain.join d2 ; !acc
```

Quel est le problème de cette implémentation? Comment le corriger?

On reprend l'exemple vu en cours:

[ ]:
```
let c = ref 0
let n = 1_000_000

let f () =
  let i = ref 0 in
  while !i < n do
    c := !c + 1;
    i := !i + 1;
  done

let () = (execute_en_parallele f f ); Format.printf "!c = %d\n" !c
```

On a vu que la valeur affichée pouvait être plus petite que 2n. Mais quelle est la plus petite valeur qui peut être affichée? (On suppose que chaque incrément est une lecture atomique suivi d'une écriture atomique).

1.4 Exercice 3 : Deadlocks

1. On écrit `use(mut,mut')` pour `lock(mut);lock(mut');unlock(mut);unlock(mut')` Quels programmes peuvent conduire à des deadlocks?

- `use(m1,m2) || use(m2,m3) || use(m3,m1)`
- `use(m1,m2) || use(m2,m3) || use(m1,m3)`
- `lock(m1);use(m2,m3);unlock(m1) || lock(m1);use(m3,m2);unlock(m1)`

2. On a n philosophes assis autour d'une table ronde qui doivent manger des nouilles avec des baguettes. De part et d'autre de chacun, une baguette. Pour manger, il leur faut deux baguettes, mais ils ne peuvent pas prendre les deux simultanément. Si la baguette est prise, ils attendent. Comment éviter qu'ils meurent de faim? Wikipedia propose la solution suivante lorsque n est pair: les philosophes aux positions paires prennent la baguette de gauche, puis celle de droite. Les philosophes aux positions impaires prennent leurs baguettes dans l'autre ordre. Selon wikipedia, cette solution est correcte. Qu'en pensez-vous? Que dire si n est impair?

1.5 Exercice 4 : Algorithme d'exclusion mutuelle de Peterson

L'algorithme de Peterson est un algorithme permettant d'implémenter un verrou partagé entre deux threads.

[ ]:
```
(* à compiler avec ocamlopt *)

let last_interested = Atomic.make 0
let interested = [|Atomic.make false; Atomic.make false|]
```

```
let lock me =
  Atomic.set interested.(me) true;
  Atomic.set last_interested me;
  let other = 1 - me in
  while (Atomic.get last_interested = me) && Atomic.get interested.(other)
  do () done

let unlock me =
  Atomic.set interested.(me) false

let n = 10_000_000

let test me c =
  for i=1 to n do
    lock me;
    c := !c + 1;
    unlock me
  done

let () =
  let c = ref 0 in
  let d0 = Domain.spawn (fun () -> test 0 c) in
  let d1 = Domain.spawn (fun () -> test 1 c) in
  Domain.join d0; Domain.join d1;
  Printf.printf "!c = %d\n" !c
```

1. Démontrer que l'algorithme implémente bien l'exclusion mutuelle: il n'y a pas de race.
2. Remplacer les atomiques par des références classiques, compilez, et observez les races.
3. Montrer que si l'on permute l'ordre des deux `Atomic.set` dans `lock` l'algorithme n'implémente plus l'exclusion mutuelle.

Fable : on peut donner une version "30 millions d'amis" de l'algorithme de Peterson. Alice et Bob sont voisins et partagent leur jardin, l'un a un chien, l'autre un chat. Pour sortir leur ami sans risque, ils mettent en place un protocole. Chacun a un drapeau qu'il peut hisser ou abaisser depuis sa maison. Il y a une longue corde qui relie les deux maisons avec un trait au milieu. Pour sortir, il faut hisser son drapeau *puis* tirer sur la corde. On sort lorsque le drapeau de l'autre n'est pas levé ou le trait sur la corde est chez le voisin. La corde est "atomique" (elle ne casse pas, le trait ne peut pas se retrouver au milieu, etc). Le drapeau est lui aussi atomique (pas de brouillard, la vitesse de la lumière est infinie, etc). Si on veut faire un parallèle avec les modèles mémoires faibles, on peut rajouter que Alice et Bob font les choses exactement dans l'ordre, ils n'ont pas de serviteur à qui ils délèguent et qui va optimiser et changer l'ordre des deux actions, parce que, "de son point de vue", c'est la même chose