cours9

November 23, 2021

1 Programmation fonctionnelle

1.1 Semaine 9 : Programmation multicoeur

Etienne Lozes - Université Nice Sophia Antipolis - 2021

1.2 Plan de la séance

Pour cette dernière séance, nous allons tenter de donner un petit aperçu de la programmation multicoeur et des processus légers, aka "threads". `OCaml` a une histoire compliquée avec les threads, ils existent sous diverses formes, nous en verrons deux:

- la librairie `Thread`, qui permet une programmation "asynchrone"
- la librairie `Domain`, qui permet de paralléliser le code dans un soucis de performance

Nous n'aborderons pas les librairies `Lwt` et `Async`, qui forcent un style de programmation "monadique", moins naturel que les autres, et plus spécifique aux langages fonctionnels (surtout Haskell).

Note : la librairie `Domain` sera intégrée dans `OCaml 5.0`, on utilisera cette année la version bêta fournie par le compilateur `4.12.0+domains+effects` (cf multicore ocaml).

1.3 Hello, world avec `Thread`

Un thread est une unité d'exécution autonome au sein du processus. Un même processus peut contenir plusieurs threads en cours d'exécution, chacun effectuant une tâche spécifique. Pour créer un thread, j'utilise la fonction `Thread.create`, qui me renvoie un "capteur" (en anglais, un "handle") qui me permettra d'attendre la terminaison de ce nouveau thread avec `Thread.join`.

[ ]:
```
let my_thread msg = Printf.printf "%s\n" msg

let main () =
    let t = Thread.create my_thread "hello, world!" in
    Printf.printf "salut le monde!\n";
    Thread.join t;
    Printf.printf "bye\n"
```

```
let () = main ()
```

Pour exécuter ce programme, je peux - le sauver dans un fichier `hello.ml`, puis le compiler avec la commande `ocamlc -thread unix.cma threads.cma hello.ml`, puis l'exécuter avec `./a.out`. - l'exécuter depuis `utop`: au début de ma session `utop`, je dois juste taper `#require "threads";;` (ou `#require "threads.posix";;` si ça ne marche pas)

Mon programme affiche

```
salut le monde!
hello, world!
bye
```

Il aurait pu aussi afficher

```
hello, world!
salut le monde!
bye
```

Les deux salutations s'exécutent "en parallèle" (grosso modo, on y reviendra). Par contre `bye` s'affiche forcément en dernier.

Il est important de bloquer le thread principal français en attente du thread secondaire anglais. Sinon, le thread principal peut terminer et arrêter brutalement tous les threads secondaires. Retirez le `Thread.join` du programme précédent et observez…

2 Lecture de l'entrée avec relance automatique

Je veux faire le programme suivant: l'ordinateur demande à l'utilisateur de saisir son nom. Si au bout de 5 secondes l'utilisateur n'a rien saisi, l'ordinateur affiche à nouveau sa demande, et ainsi de suite jusqu'à ce que l'utilisateur ait enfin saisi quelque chose.

Le problème, c'est que la fonction `input_line` est bloquante: si je l'appelle pour lire le nom de l'utilisateur, je ne récupèrerai la main qu'une fois que l'utilisateur aura saisi quelque chose.

Il y a plusieurs solutions pour résoudre ce problème (notamment à base de signaux ou d'entrée/sorties non bloquantes, pour ceux qui ont suivi le cours de système), mais il y a surtout une solution assez simple qui utilise un thread secondaire pour gérer l'affichage du rappel.

[ ]:
```
let rec my_thread () =
    print_string "Quel est ton nom? J'attends...";
    print_newline ();
    Thread.delay 5.; (* s'endort 5 secondes *)
    my_thread ()

let main () =
    Thread.create my_thread ();
    let nom = input_line stdin in
    print_string "Bonjour, ";
    print_string nom;
```

```
    print_newline()

let () = main ()
```

Notez que l'on n'a pas mis de `Thread.join`: c'est la terminaison du thread principal qui entraine la terminaison du thread secondaire. Ce n'est pas très propre, mais comme il n'existe pas (ou presque pas) de fonction `Thread.kill`, on ne voit pas comment faire autrement pour le moment…

Mais supposons maintenant que l'on veuille inverser les rôles, et que ce soit le thread secondaire qui fasse le `input_line`.

[ ]:
```
let my_thread () =
    let nom = input_line stdin in
    print_string "Bonjour, ";
    print_string nom;
    print_newline()


let main () =
    Thread.create my_thread ();
    while true do (* <- ???? *)
        print_string "Quel est ton nom? J'attends...";
        print_newline ();
        Thread.delay 5.;
    done;

let () = main ()
```

C'est pire! Le thread principal ne termine pas, je dois le forcer à quitter avec Crtl+C.

Ne peut-on pas faire mieux?

3 La communication entre threads

C'est l'un des grands thèmes de la programmation concurrente: comment les threads se synchronisent-ils? Il existe de très nombreuses **primitives de synchronisation** (on en verra quelques unes), mais pour le problème qui nous concerne, il existe une solution très simple, même si "un peu sale" (on verra plus tard pourquoi) de résoudre le problème.

[ ]:
```
let finished = ref false (* <- variable partagée! *)

let my_thread () =
    let nom = input_line stdin in
    print_string "Bonjour, ";
    print_string nom;
    print_newline();
    finished := true  (* "signal" *)
```

```
let main () =
    let t = Thread.create my_thread () in
    while not !finished do (* <- mieux! *)
        print_string "Quel est ton nom? J'attends...";
        print_newline ();
        Thread.delay 5.;
    done;
    Thread.join t (* <- quittons proprement *)

let () = main ()
```

Pour les L3 qui ont suivi le cours de système 2: la grande nouveauté des threads par rapport aux processus, c'est qu'ils partagent leur mémoire. Par comparaison, qu'aurait donné ce programme avec un `Unix.fork` à la place du `Thread.create`?

Les threads sont plus rapides à créer que les processus (même si la création d'un thread reste une opération "coûteuse) parce qu'il n'y a pas besoin de dupliquer toutes les ressources du processus (la mémoire, mais aussi les descripteurs de fichiers, les handlers de signaux, etc). La communication entre threads est aussi beaucoup plus simple et rapide qu'entre processus… il y a donc beaucoup de situations où on privilégie les threads aux processus.

Un exemple ultra classique est un "serveur multi-thread" qui crée un nouveau thread à chaque nouvelle connection et lui délègue l'interaction avec le nouveau client. On va le simplifier un peu, en particulier pour les L2 qui n'ont pas encore vu tout ça.

4 Une commande `cat` parallèle

On rappelle que `cat fichier1 fichier2 fichier3 ...` lit les fichiers passés en arguments et les écrit sur la sortie standard. On va créer un programme `parallel_cat` qui fait la même chose, mais qui traite chaque fichier dans un thread différent.

Allons y doucement, et commençons par gérer le cas où il y a 2 fichiers, on généralisera ensuite.

Précisons d'abord ceci: lorsque j'exécute `Thread.create f e`, le nouveau thread appelle la fonction `f` avec l'argument `e` (on parle d'**appel de fonction asynchrone**). Je peux donc "passer un argument" à un thread au moment de sa création.

[ ]:
```
let cat filename =
    let ic = open_in filename in
    let rec aux () =
        try
            print_string (input_line ic);
            print_newline ();
            aux ()
        with
            End_of_file -> ()
    in aux (); close_in ic
```

```
let parallel_cat filename1 filename2 =
    let t1 = Thread.create cat filename1 in (* <- passage d'argument *)
    let t2 = Thread.create cat filename2 in
    Thread.join t1; Thread.join t2

let () = parallel_cat Sys.argv.(1) Sys.argv.(2)
```

Pour tester mon code, je compile et j'exécute `./a.out fichier1 fichier2` avec des fichiers un peu gros, par exemple un milier de lignes contenant "a" dans le premier et "b" dans le second. J'observe des choses comme

```
a
a
a
b
a
ba

a
...
```

Je peux aussi tester mon code avec des fichiers FIFO:

```
mkfifo fifo1 fifo2
./a.out fifo1 fifo2
```

Puis j'ouvre deux autres terminaux dans lesquels je tape `cat > fifo1` et `cat > fifo2`. J'ai alors deux "sessions" ouvertes mon `parallel_cat` qui sont indépendantes l'une de l'autre. C'est, de façon très simplifiée, le même patron de parallélisation que celui qu'on utiliserait pour écrire un serveur multi-thread.

Je vais maintenant généraliser mon code pour pouvoir traiter un nombre arbitraire de fichiers. Il va donc falloir que je crée un nombre arbitraire de threads. Je dois d'abord tous les créer, puis ensuite faire `join` pour chacun d'eux.

Pour conserver leurs "handles" (pour pouvoir faire `join` à la fin), il faut que je les stocke quelque part. Une première possibilité est de créer une liste de "handles" `[t1; t2;...;tn]` qui s'allonge à chaque `Thread.create` (c'est donc une liste mutable).

[ ]:
```
let parallel_cat filenames_array =
    let thread_handles = ref [] in
    for i = 0 to Array.length filenames_array - 1 do
        let t = Thread.create cat filenames_array.(i) in
        thread_handles := t :: !thread_handles
    done;
    List.iter Thread.join !thread_handles
    (* i.e. for t in thread_handles do Thread.join t done *)

let () = parallel_cat (Array.sub Sys.argv 1 (Array.length Sys.argv - 1))
```

Notez que j'utilise `List.iter` au lieu d'une boucle pour parcourir la liste des handles. Vous devriez commencer à avoir l'habitude, c'est le style fonctionnel, et dans ce cours c'est ce qu'on privilégie.

On va d'ailleurs remplacer aussi la première boucle `for` par une fonction d'ordre supérieur. Avez vous une idée de quelle fonction d'ordre supérieur on va utiliser?

[ ]:
```
let parallel_cat filenames_array =
    Array.map (Thread.create cat) filenames_array |> Array.iter Thread.join
```

Pour un code plus conceptuel, je peux passer par une fonction plus générale qui implémente une version parallèle de `Array.iter`.

[ ]:
```
let parallel_array_iter f array =
    Array.map (Thread.create f) array |> Array.iter Thread.join

let parallel_cat = parallel_array_iter cat
```

5 Évaluation des performances

Est-ce que mon programme `parallel_cat` est plus rapide que `cat`?

Pour répondre à cette question, je vais faire une expérience: je vais mesurer le temps que met `cat fichier1 fichier2` et `parallel_cat fichier1 fichier2` avec des fichiers `fichier1`, `fichier2` de tailles similaires. Je sais déjà que `parallel_cat` n'ira pas deux fois plus vite que `cat`, parce que la création de threads prend du temps. Je vais donc regarder ce qui se passe lorsque la taille de ces fichiers devient grande. Mon espoir est que pour des fichiers de tailles importantes, le coût de la création de thread soit négligeable devant le gain de temps permis par la parallélisation.

Pour comparer ce qui est comparable, je vais réécrire `cat` en `OCaml`. Pour mesurer le temps d'exécution, je vais utiliser la fonction `Unix.gettimeofday` (qui a l'avantage d'être disponible quel que soit l'OS). Le code est disponible ici (bon exercice: le faire vous-même).

Résultat: pour de petites entrées, `parallel_cat` est plus lent, comme prévu… mais pour de grandes entrées, il reste plus lent!

Pire, l'échelle logarithmique de la première courbe masque le fait (visible sur la deuxième courbe) que le ratio temps séquentiel sur temps parallèle tend vers 0.9.

Plus les données sont grandes, plus on perd de temps dans la version parallèle!

En fait, les threads créés par `Thread` ne s'exécutent pas vraiment "en parallèle", mais plutôt "en entrelacé". Il n'y a qu'un seul coeur de mon CPU qui exécute mon programme. Ceci est lié au ramasse-miettes ("garbage collector") historique de Caml (ce n'est d'ailleurs pas spécifique à Caml, Python a un problème similaire).

Mais alors à quoi bon?

La programmation multithread avec `Thread` est intéressante pour **clarifier le code**.

Tous les exemples qu'on a vus peuvent se faire sans `Thread` (et en très légèrement plus performants) en utilisant des fonctions pour faire de la scrutation ("poll") comme `Thread.select`, en gérant des timers, etc. Mais le code est moins lisible, et plus difficile à maintenir.

Les applications dans lesquelles les threads ont leur intérêt sont donc surtout celles où il y a à gérer des évènements de sources différentes, et où une notion “d’agent” est assez naturelle, comme un serveur, ou un jeu vidéo.

Quelques précisions sur la façon dont les threads créés par `Thread` s’entrelacent.

Ces threads sont parfois décris comme “non préemptifs”.

Un thread qui a la main la garde tant qu’il n’est pas bloqué par une instruction (un appel système en général). Lorsque c’est le cas, l’ordonnanceur peut passer la main à un autre thread.

Un thread peut indiquer par un appel à `Thread.yield` qu’il est d’accord pour passer la main: il reprendra la main plus tard à cet endroit quand l’ordonnanceur l’aura de nouveau choisi. Ceci permet d’accroitre l’entrelacement des threads.

En TP on s’en servira pour simuler des threads “préemptifs” avec notre Caml 4.xx, en attendant Caml 5.0.

6 La programmation multicoeur

Nous allons maintenant voir comment tirer pleinement parti de l’architecture multicoeur de notre ordinateur pour paralléliser du code et améliorer les performances en utilisant la version bêta `4.12.0+domains+effects`.

Avant de voir les détails, un peu de contexte. Au siècle dernier, la croissance de la puissance de calcul reposait sur une augmentation exponentielle de la puissance de calcul: la loi de Moore, qui s’est vérifiée tout au long du siècle dernier, prédisait un doublement de la puissance de calcul tous les 18 mois.

L’industrie des microprocesseurs a capitulé dans cette course folle au tournant des années 2000. Désormais, même si la puissance de calcul d’une unité de calcul s’améliore régulièrement, c’est aussi la multiplication des coeurs au sein du CPU qui permet d’accroitre sa puissance de calcul.

D’où la nécessité d’écrire des programmes “multicoeurs” pour pouvoir bénéficier des avancées technologiques à venir.

Un CPU à 2 coeurs ne calculera jamais aussi vite qu’un CPU à un seul coeur 2 fois plus rapide: c’est ce que dit, de manière plus formelle, la loi d’Amdhal.

En substance, la loi d’Amdhal nous dit quelque chose d’assez trivial: pour que notre programme sur CPU bicoeur soit aussi rapide que sur un CPU monocoeur 2 fois plus rapide, il faut qu’on puisse l’écrire avec deux processus légers (des threads “multicoeurs”, pas des threads de la librairie `Thread`) qui travaillent en permanence et ne s’attendent jamais.

Comme il y a toujours une phase d’initialisation où on crée ces threads et une phase de conclusion où on rassemble les résultats, ce n’est pas possible, mais il faut éviter autant que possible toutes les autres attentes inutiles.

Une autre leçon de la loi d’Amdhal est qu’il vaut mieux ne pas sur-paralléliser notre programme: si on lance n threads préemptifs avec n plus grand que le nombre de coeurs, on va perdre du temps en ordonnancement, et il serait plus efficace d’avoir autant de threads que de coeurs (ce qu’on vient d’observer avec cat pour 1 coeur).

7 La librairie `Domain`

Pour la suite du cours, on suppose qu'on a installé la version `4.12.0+domains+effects` du compilateur:

```
opam switch create 4.12.0+domains+effects
opam install domainslib utop dune ocaml-lsp-server ...
```

On peut ensuite passer d'un compilateur à l'autre avec `opam switch default` ou `opam switch 4.12.0+domains+effects`.

Les **domaines** sont des flots d'instructions autonomes, au sein d'un même processus, qui peuvent s'exécuter en parallèle les uns des autres sur des coeurs différents, ou être interrompus de manière préemptive par l'ordonnanceur.

Dans d'autres langages on appelerait cela des threads, mais pour éviter la confusion avec les threads Caml historiques, on les appele des domaines.

La fonction `Domain.spawn thunk` crée un nouveau domaine qui exécute le glaçon `thunk`. Elle renvoie un **futur** (aussi appelé *promesse*) qui permettra de récupérer le résultat du thunk avec la fonction `Domain.join`.

[ ]:
```
let my_domain greetings =
    let name = read_line () in
    let res = greetings ^ " " ^ name in
    res

let main () =
    let fut = Domain.spawn (fun () -> my_domain "hello") in
    let str = Domain.join fut in
    print_string s;
    print_newline ()

let () = main ()
```

Examinons le type de `Domain.spawn` et `Domain.join`.

```
Domain.spawn : (unit -> 'a) -> 'a Domain.t
Domain.join : 'a Domain.t -> 'a
```

Ça ne vous rappelle rien?

La semaine dernière on avait vu

```
paresse : (unit -> 'a) -> 'a paresse
force : 'a paresse -> 'a
```

Il y a une petite ressemblance!

8 Parallélisation de boucle

On considère la fonction suivante, qui calcule le nombre d'occurrences de x dans le tableau arr

[ ]:
```
let nb_occurs x arr =
  let res = ref 0 in
  for i=0 to Array.length arr - 1 do
    if arr.(i) = x then res := !res + 1
  done;
  !res
```

On voudrait paralléliser cette fonction en utilisant deux domaines (pour commencer).

L'idée est la suivante: un domaine va calculer le nombre d'occurrences dans la première moitié du tableau, pendant qu'un autre domaine fait ce calcul pour la deuxième moitié. Quand ils ont terminé, on récupère leurs résultats et on les ajoute.

[ ]:
```
let nb_occurs_slice x from upto arr =
  let res = ref 0 in
  for i = from to upto - 1 do
    if arr.(i) = x then res := !res + 1
  done;
  !res

let parallel_nb_occurs x arr =
  let n = Array.length arr in
  let fut1 = Domain.spawn (fun () -> nb_occurs_slice x 0 (n/2) arr) in
  let fut2 = Domain.spawn (fun () -> nb_occurs_slice x (n/2) n arr) in
  let res1 = Domain.join fut1 in
  let res2 = Domain.join fut2 in
  res1 + res2
```

A nouveau, comparons les performances du code séquentiel et du code parallèle. Cette fois-ci, comme on pourrait s'y attendre, pour de petites entrées, la version séquentielle est plus rapide, mais pour de grandes entrées c'est la version parallèle qui est la plus rapide, et qui tend à être deux fois plus rapide (attention à l'échelle logarithmique sur la première courbe).

9 Map-reduce

Map-reduce est un "patron de calcul parallèle" qui est très populaire dans le traitement de données volumineuses.

Nous allons nous inspirer de ce patron de calcul parallèle pour revisiter la fonction `parallel_nb_occurs` que nous venons d'écrire, et donner une idée (un peu inexacte, mais plus simple à expliquer) de ce qu'il y a derrière ce mot.

Le nom même du patron map-reduce devrait déjà vous donner une petite idée de ce qu'il fait. Map, vous vous souvenez de ce que c'est, non? Hum… Quelques exemples ne seront peut-être pas inutiles pour tout le monde..

[ ]:
```
List.map (fun x -> x = 0) [0; 1; 2; 3; 0] (* [true; false; false; false; true]␣
↪*)
Array.map int_of_string [|"1"|] (* [|1|] *)
```

```
List.map (Option.map (fun x -> x + 1)) [None; Some 0] (* [None; Some 1] *)
```

Reduce, vous le connaissez déjà sous un autre nom, c'est `fold_left` (mais aussi `fold_right`, ou juste `fold`).

On rappele que `List.fold_left (++) x0 [x1;...;xn]` renvoie `(..((x0 ++ x1) ++ x2) .. ++ xn)`.

Reduce fait la même chose, mais ne s'utilise que si `(++, x0)` est un monoïde commutatif. On peut alors enlever les parenthèses dans la somme, et l'ordre des xi n'est pas important.

Définissons maintenant un map-reduce "non parallèle" (si on peut encore parler de "map-reduce" dans ce cas) sur les tableaux.

[ ]:
```
let sequential_map_reduce f (++) _0 array =
  array |> Array.map f |> Array.fold_left (++) _0

(* ou avec une boucle for, pour ceux qui préfèrent *)

let sequential_map_reduce f (++) _0 array =
  let acc = ref _0 in
  for i = 0 to Array.length array - 1 do
     acc := !acc ++ f array.(i)
  done;
  !acc
```

Autrement dit, je commence par appliquer `f` à chaque élément du tableau, puis je fais la somme, au sens du monoïde `(++, _0)`, de toutes les images par `f`.

Il devient alors aisé d'écrire la fonction `nb_occurs` :

[ ]:
```
let sequential_nb_occurs x arr =
   sequential_map_reduce (fun y -> if x=y then 1 else 0) (+) 0 arr
```

L'intérêt de ce patron, c'est qu'il est très général, et qu'on peut l'utiliser pour exprimer beaucoup de calculs utiles (en particulier des calculs statistiques en science des données).

À titre d'exemple, je peux calculer le plus grand élément, en valeur absolue, d'un tableau d'entiers, ou encore la moyenne d'un tableau de flottants.

[ ]:
```
let sequential_max_of_int_array = sequential_map_reduce abs max 0

let mean_of_float_array arr =
  let (++) (x1,n1) (x2,n2) = (x1 +. x2, n1 + n2) in
  let _0 = (0. ,0) in
  let (sum, len) =  sequential_map_reduce (fun x -> (x,1)) (++) _0 in
  sum /. (float len)
```

Je vais donc maintenant m'intéresser à la façon dont je pourrais implémenter ce patron de calcul parallèle… de manière parallèle, toujours avec deux domaines pour commencer.

Simplement, je vais appliquer map-reduce avec chaque domaine sur une moitié du tableau, puis je combinerai les résultats avec `++`.

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

  let n = Array.length array in

  let map_reduce_on_slice from upto =
    let acc = ref _0 in
    for i = from to upto - 1 do acc := !acc ++ f array.(i) done;
    !acc 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
```

9.1 Recyclage de domaines: “pool”, “workers”, et tâches

En suivant l’approche qu’on vient de voir, il est facile de généraliser la fonction à n domaines (cf exercice en TD). L’idéal est de prendre n égal au nombre de coeurs - 1 (le domaine principal, qui fait les `spawn`s et les `join`s, peut occuper un coeur): si on prend plus de domaines que de coeurs, on va perdre du temps en ordonnancement (cf loi d’Amdahl).

Il faut aussi faire attention au coût de la création de domaines: si on doit paralléliser une seule boucle dans le programme, on peut suivre l’approche qu’on vient de voir, mais si on doit en paralléliser plusieurs, on gâche du temps en `spawn` et `join`, et il vaudrait mieux “recycler” des domaines qu’on a créés plutôt que de les supprimer (avec `join`) pour en créer d’autres (avec `spawn`) à la boucle à paralléliser suivante.

La librairie `Domainslib` fournit un certain nombre de fonctions qui permettent de mettre en oeuvre cette idée de “recyclage de domaine”.

L’idée est de créer un **pool** de domaines (de nombre égal au nombre de coeurs - 1), appelés “workers”, puis d’assigner à un pool un certain nombre de **tâches** (potentiellement plus de n tâches).

Les tâches deviennent alors l’unité d’exécution parallèle qu’on manipule dans tout le programme.

`Task.async` et `Task.await` sont les équivalents, pour les tâches, de `Domain.spawn` et `Domain.join`.

[ ]:
```
open Domainslib
let nb_cores_on_my_machine = 8
let pool = Task.setup_pool ~num_additional_domains:(nb_cores_on_my_machine - 1)

let my_task greetings =
    let name = read_line () in
    let res = greetings ^ " " ^ name in
    res

let main () =
```

```
    let promise = Task.async pool (fun () -> my_task "hello") in
    let str = Task.await pool promise in
    print_string s;
    print_newline ()

let () = main ()
```

Notons aussi que les problèmes de parallélisation de boucle qu'on a regardé induisent des tâches qui prennent toutes le même temps.

Ce n'est pas toujours le cas.

Supposons par exemple qu'on veuille paralléliser le programme suivant (ce n'est certainement pas le meilleur algorithme pour résoudre le problème considéré, mais peu importe, c'est un exemple).

[ ]:
```
let is_prime n =
  let rec iter i = i * i > n || (n mod i <> 0 && iter (i+1)) in
  n >= 2 && iter 2

let count_primes upto =
  let res = ref 0 in
  for k = 0 to upto - 1 do
     if is_prime k then res := !res + 1
  done; !res
```

On aimerait distribuer le travail de manière équitable.

Cependant, on ne sait pas à l'avance, pour un `k` donné, si `is_prime k` va prendre peu de temps ou beaucoup de temps.

Ce n'est donc pas évident de découper à l'avance l'ensemble des entiers entre 0 et `upto - 1` en n sous-ensembles nécessitant des temps de calculs comparables.

Par contre, on peut créer une tâche par entier, ou par tronçon (en anglais "chunk") de quelques entiers consécutifs .

Cela fait donc beaucoup de tâches, beaucoup plus que le nombre de domaines dans le pool. Cependant c'est la bonne approche: chaque domaine du pool est toujours actif: quand il a fini une tâche il peut en "voler" une autre ("task stealing") qui n'a pas encore été assignée à un domaine.

Pour que cela soit bien efficace, il faut que l'ordonnanceur de tâche de `Domainslib` soit non préemptif: un domaine (donc idéalement un coeur) qui commence une tâche ne sera pas préempté tant qu'il n'a pas fini cette tâche (pour éviter les commutations de tâches inutiles, et couteuses).

Voyons maintenant comment coder cette idée.

[ ]:
```
let parallel_count_primes pool upto =

  (* je lance une tâche par entier,
     et je stocke les promesses dans un tableau *)
  Array.init upto
```

```
    (fun i -> Task.asynch pool
        (fun () -> Bool.to_int (is_prime i))) |>

  (* je récupère les valeurs de retour des tâches
     et je les stocke dans un tableau *)
  Array.map (Task.await pool) |>

  (* je somme ce dernier tableau *)
  Array.fold_left (+) 0
```

9.2 Accès mémoire concurrents : les “data races”

Un accès mémoire concurrent, en anglais “data race”, ou juste “race”, correspond à la situation où deux threads (ou domaines) tentent d’accéder “simultanément” (sans synchronisation particulière) à une même case mémoire, et l’un de ces accès au moins est un accès en écriture.

Nous avons déjà écrit du code qui contenait un accès mémoire concurrent, au tout début du cours. Vous vous souvenez quand? Revoyons ce code…

[ ]:
```
let finished = ref false (* <- variable partagée *)

let my_thread () =
    let nom = input_line stdin in
    print_string "Bonjour, ";
    print_string nom;
    print_newline();
    finished := true  (* <- accès en écriture *)


let main () =
    let t = Thread.create my_thread () in
    while not !finished do (* <- accès en lecture *)
        print_string "Quel est ton nom? J'attends...";
        print_newline ();
        Thread.delay 5.;
    done;
    Thread.join t

let () = main ()
```

Faisons un autre exemple d’accès concurrent: un compteur partagé qu’on incrémente depuis deux domaines.

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

let f () =
  let i = ref 0 in
```

```
  while !i < n do
    c := !c + 1; (* <- race! *)
    i := !i + 1; (* <- pas race *)
  done

let () =
  let pool = Task.setup_pool ~num_additional_domains:2 () in
  let t1 = Task.async pool f in
  let t2 = Task.async pool f in
  Task.await pool t1;
  Task.await pool t2;
  Format.printf "!c = %d\n" !c
```

Lorsque j'exécute ce programme, il n'affiche pas 2n, et si je le lance plusieurs fois, j'obtiens des résultats différents à chaque fois. Quel est le problème?

On peut interpréter le problème de la façon suivante: l'instruction `c := !c +1` correspond à deux instructions machines: une lecture et une écriture, alors qu'on voudrait avoir un incrément **atomique**. En fait c'est plus subtil que ça: même la lecture ou l'écriture d'une référence "de base", si elle n'est pas synchronisée, n'est pas une opération "sûre" (rendez-vous en master pour en savoir davantage).

Retenez simplement: les data races sont des erreurs de conception du programme, elles sont toujours à éviter.

9.3 Éviter les races: notion de droit d'accès

À tout moment, un thread (ou un domaine) "possède" une partie de la mémoire partagée: il a le droit d'accès sur ce qu'il possède, et uniquement sur ce qu'il possède.

Pour éviter les data races, le programmeur doit donc se poser la question suivante à chaque accès à une donnée: *est-ce que cette donnée appartient bien au thread qui y accède, à cet instant*?

Faisons un petit exemple.

[ ]:
```
let x = ref 0
let y = ref 0

let my_domain () =
   (* quand il démarrera, ce domaine "prendra" le droit d'accès à x *)
   x := 1

let main () =
  (* au départ main "possède" x et u *)

  let fut = Domain.spawn my_domain in
  (* en lançant le domaine secondaire, main lui "transfère" x *)
  (* x := 2 ne serait donc pas légal, ce serait une race *)
  y := 1;  (* <- j'ai le droit, main "possède" toujours y *)
```

```
  Domain.join fut;
  x := !x + 1 (* <- j'ai le droit, le join a tranféré x à main *)
```

Cette notion de droit d'accès est malheureusement souvent juste une histoire que le programmeur se raconte pour se convaincre qu'il n'y a pas de race. Seuls quelques langages de programmation (le plus célèbre est sans doute Rust) refusent de compiler si une discipline d'ownership n'est pas respectée.

Le droit d'accès est fractionnable: si j'ai un droit d'accès exclusif (de valeur 1), je peux transférer 1/2 de ce droit d'accès à un domaine, et garder 1/2 pour moi. Avec ce 1/2 droit d'accès, je n'ai que le droit de lire. Pour pouvoir écrire, je dois attendre de récupèrer le 1/2 droit d'accès manquant: je ne peux écrire qu'avec un droit d'accès de valeur 1.

9.4 Références atomiques

Revenons sur l'exemple où deux domaines faisaient des incréments concurrents sur un compteur. Il est possible de corriger le code en utilisant une "référence atomique" de la librairie `Atomic`.

[ ]:
```
let c = Atomic.make 0  (* ref 0 "atomique" *)
let n = 1_000_000

let f () =
  let i = ref 0 in
  while !i < n do
    Atomic.incr c; (* incrément "atomique" *)
    i := !i + 1;
  done

let () =
  (* ... *)
  Format.printf "!c = %d\n" (Atomic.get c) (* lecture "atomique" *)
```

Note: Java a aussi ses "références atomiques". En Java le mot-clé `volatile` permet de déclarer des variables qui sont des "références atomiques".

Du point de vue de l'ownership, les références atomiques peuvent être décrites comme des ressources qui n'appartiennent à aucun thread.

Quand un thread fait un accès à une référence atomique, il s'agit en réalité d'une opération de synchronisation, qui occasionne potentiellement des transferts d'ownership, de même que l'opération de synchronisation `join` occasionne potentiellement un transfert (acquisition) d'ownership.

L'ownership de la référence atomique est acquis par le thread le temps de l'opération atomique, et aussitôt relâché.

La librairie `Atomic` contient de nombreuses opérations atomiques intéressantes, notamment "la reine des opérations atomiques":

```
compare_and_set : 'a t -> 'a -> 'a -> bool
(* compare_and_set r seen v sets the new value of r to v only
   if its current value is physically equal to seen --
```

```
   the comparison and the set occur atomically.
   Returns true if the comparison succeeded
   (so the set happened) and false otherwise. *)
```

Pour en savoir plus sur la hiérarchie d'expressivité des opérations atomiques, je vous recommande le cours de Rachid Guerraoui au collège de France (niveau très accessible).

9.5 Exclusion mutuelle

Dans certaines situations, les opérations atomiques de base ne suffisent pas.

Supposons par exemple qu'on veuille partager un tableau d'entiers entre plusieurs threads, qui doivent pouvoir lire et écrire de manière atomique une case du tableau. Mais supposons aussi que les threads doivent pouvoir trier le tableau en place de manière atomique: il faut alors acquérir le droit d'accès exclusif (écriture) sur tout le tableau pour empêcher les autres threads de le lire ou le modifier pendant toute la durée du tri.

Il n'existe malheureusement pas d'opération atomique qui permette de faire un tri de tableau, il faut la programmer.

Il y a plusieurs approches pour ce type de problème:

- l'approche "non-bloquante", qui est souvent privilégiée, mais plus compliquées à mettre en oeuvre (rendez-vous en cours de master pour en savoir plus),
- l'approche par **exclusion mutuelle**, à base de verrou ou d'autre primitives de **synchronisation bloquante**.

Nous allons maintenant discuter de cette deuxième approche.

Un verrou (ou **mutex**), est un objet sur lequel on peut faire deux opérations: `lock`, et `unlock`.

L'opération `lock` permet d'acquérir le verrou. C'est une opération bloquante: si autre thread a acquis le verrou, mon appel à `lock` va suspendre mon exécution le temps que la situation évolue.

Un verrou s'utilise généralement pour faire une **section critique**, qui commence par `lock` et termine par `unlock`.

Revisitons l'exemple du compteur partagé en utilisant une section critique.

```
let c = ref 0  (* <- référence non atomique, attention aux races *)

let mut = Mutex.create ()

let f () =
  let i = ref 0 in
  while !i < n do
    Mutex.lock m; (* entrée en section critique *)
    c := !c + 1 ; (* j'ai le droit, j'ai acquis c *)
    Mutex.unlock m; (* sortie de section critique, je rend c *)
    i := !i + 1;
  done
```

9.6 Interblocage

Un **interblocage** (en anglais "**deadlock**") se produit lorsque des threads s'attendent mutuellement (on parle aussi d'*attente circulaire*). L'exemple le plus simple dans lequel on peut rencontrer un interblocage est celui de 2 threads qui essaient d'acquérir deux verrous dans des ordres différents

```
lock(mut1);lock(mut2);unlock(mut1,mut2) || lock(mut2);lock(mut1);unlock(mut1,mut2)
```

Un outil conceptuel intéressant pour comprendre le problème est le graphe des prises de mutex (GPM).

Dans ce graphe on représente chaque mutex et chaque thread par un noeud. Il y a une arête `m->t` si le thread `t` a déjà acquis le mutex `m` mais ne l'a pas encore relâché. Il y a une arête `t->m` si le thread `t` est en attente sur le mutex `m`.

Une configuration d'interblocage se caractérise par un GPM qui contient un cycle.

Dans l'exemple précédent, l'interblocage était caractérisé par le GPM

`mut1 -> thread gauche -> mut2 -> thread droit -> mut1`.

S'il n'est pas possible (quel que soit l'ordonnancement) d'atteindre un GPM contenant un cycle, c'est qu'il n'y a pas d'interblocage possible.

Une condition suffisante pour éviter tout risque d'interblocage est la discipline de **prise de verrou ascendante**: on suppose que les threads se sont mis d'accord sur une relation d'ordre entre tous les verrous existants, et que chaque thread, lorsqu'il fait un `lock(m)`, n'a déjà acquis que des verrous `m' < m`.

Si tous les threads respectent cette discipline, on peut voir que les GPM atteignables vérifient tous la propriété suivante: le long de tout chemin, les verrous sont en ordre croissant. Il est alors impossible de former un cycle.

9.7 Variables de condition et sémaphores

Il existe de nombreuses primitives de synchronisation entre threads. Nous allons pour finir en évoquer deux d'entre elles qui ont de nombreuses applications: les variables de condition, et les sémaphores.

Un sémaphore est un entier *positif ou nul* muni d'opérations d'incrément et de décrément atomiques. À la différence d'un simple entier atomique, le décrément d'un sémaphore est bloquant (si la valeur du sémaphore est 0). Les opérations d'incrément et de décrement ont parfois des noms exotiques, en Caml elles s'appellent `acquire` (décrément) et `release` (incrément).

On peut facilement tester les sémaphores au toplevel.

[ ]:
```
module S = Semaphore.Counting;;
let s = S.make 0;;
(* val s : S.t = <abstr> *)
let f () = S.acquire s;print_int (S.get_value s);;
let d = Domain.spawn f;;
(* rien ne s'affiche *)
S.release s;;
(* affiche 0 *)
```

Je vais maintenant essayer de reprogrammer les sémaphores.

Pour commencer, je vais me baser sur l'instruction atomique `compare_and_set` que nous avons vue tout à l'heure.

[ ]:
```
module MySemaphore = struct

  type t = int Atomic.t

  let make n = assert(n>=0); Atomic.make n

  let release sem = Atomic.incr s

  let rec acquire sem =
    let n = Atomic.get sem in
    if n = 0 || not Atomic.compare_and_set sem n (n-1)
    then acquire sem
end
```

Cette implémentation est basée sur de l'**attente active**: le thread n'est pas suspendu et continue de s'exécuter. C'est une approche intéressante si l'attente reste courte, mais pour une attente plus longue, il est préférable d'informer l'ordonnanceur qu'il peut passer la main à un autre thread.

Voyons maintenant une autre implémentation basée sur un verrou et une variable de condition.

[ ]:
```
module MySemaphore = struct

  type t = {
     mutable value: int;
     mutex: Mutex.t;
     not_null: Condition.t;
  }

  let make n =
     assert(n>=0);
     {value = n; mutex = Mutex.create (); not_null = Condition.create ()}

  let release sem =
     Mutex.lock sem.mutex;
     sem.value <- sem.value + 1;
     Condition.signal sem.not_null;
     (* sort un waiter (s'il y en a) de la file d'attente de not_null
        et le place dans la file d'attente de son verrou *)
     Mutex.unlock sem.mutex

  let rec acquire sem =
    Mutex.lock sem.mutex;
    while sem.value = 0 do
       Condition.wait sem.not_null sem.mutex;
```

```
      (* relâche le verrou et s'endort dans la file d'attente de not_null;
         peut reprendre quand il a été signalé puis sorti de la file
         d'attente de mutex *)
    done;
    sem.value <- sem.value - 1;

end
```

Note: il existe aussi la fonction `Condition.broadcast` qui permet de remettre en file d'attente de leur verrou tous les threads qui étaient en dans la file d'attente de la variable de condition.

9.8 Conclusion

Nous avons vu comment programmer avec des threads en OCaml 4.xx, et avec des domaines et des tâches en Multicore OCaml. Les choses auront peut-être évolué légèrement dans quelques mois avec la sortie d'Ocaml 5.0… mais peu importe, les concepts de threads, verrous, et parallélisation de boucles sont universels.

Nous n'avons fait bien sûr qu'effleurer le sujet, en espérant vous donner envie d'en savoir plus, par exemple à l'aide du best-seller sur la programmation multicoeur de Herlihy et Shavit.

J'espère aussi vous avoir montré en quoi certaines idées de programmation fonctionnelle influencent d'autres façon de programmer, comme la programmation parallèle, et que ce que vous avez vu tout au long de ce semestre, et qui vous a peut-être paru, parfois, un cours d'écriture de haïku, est source d'inspiration pour de nombreux nouveaux langages qui ne se définissent pas toujours comme fonctionnel.