Fonctionnalité et Modularité

Cours 10

Jean-Jacques Lévy

jean-jacques.levy@inria.fr

http://jeanjacqueslevy.net/prog-fm

Plan

- parallélisme
- calcul parallèle
- domaines Ocaml 5
- parallélisme imbriqué
- concurrence
- fils concurrents

télécharger Ocaml en `http://www.ocaml.org`

Programmation parallèle

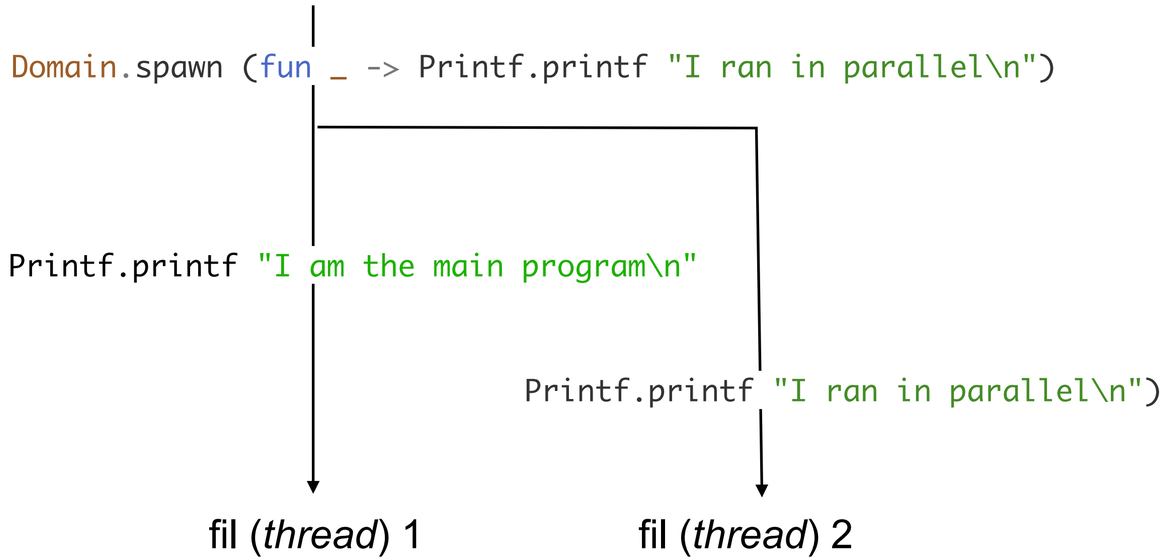
- en Ocaml, un domaine définit une unité de parallélisme

```
Domain.spawn (fun _ -> Printf.printf "I ran in parallel\n") ;;
Printf.printf "I am the main program\n" ;;
```

- 2 fils (système) de calcul sont créés

- ils s'exécutent en parallèle

- l'ordre des impressions est indéfini



(les domaines n'existent qu'à partir de Ocaml 5)

Programmation parallèle

- on considère la fonction de Fibonacci avec un argument sur la ligne de commande

```
let n = try int_of_string Sys.argv.(1) with _ -> 1 ;;

let rec fib n = if n < 2 then 1 else fib (n - 1) + fib (n - 2) ;;

let _ =
  let r = fib n in
  Printf.printf "fib(%d) = %d\n%!" n r ;;
```

- on compile et exécute

```
% ocamlc -o fib fib.ml

% ./fib 39
fib(39) = 102334155
```

```
% hyperfine './fib 39'
Benchmark 1: ./fib 39
  Time (mean ± σ):      1.659 s ±  0.061 s    [User: 1.554 s, System: 0.001 s]
  Range (min … max):    1.550 s …  1.704 s    10 runs
```

Programmation parallèle

- on exécute 2 fois la fonction de Fibonacci en parallèle

```
let n = int_of_string Sys.argv.(1)

let rec fib n = if n < 2 then 1 else fib (n - 1) + fib (n - 2)

let _ =
  let d1 = Domain.spawn (fun _ -> fib n) in
  let d2 = Domain.spawn (fun _ -> fib n) in
  let r1 = Domain.join d1 in
  Printf.printf "fib(%d) = %d\n%!" n r1;
  let r2 = Domain.join d2 in
  Printf.printf "fib(%d) = %d\n%!" n r2
```

`join` attend la fin du fil et renvoie son résultat

- on compile et exécute

```
% ocamlc -o fib2 fib2.ml

% ./fib2 39
fib(39) = 102334155
fib(39) = 102334155
```

```
% hyperfine './fib2 39'
Benchmark 1: ./fib2 39
  Time (mean ± σ):      1.699 s ±  0.058 s    [User: 3.200 s, System: 0.002 s]
  Range (min … max):    1.601 s …  1.748 s    10 runs
```

- 1 fil même temps d'exécution que 2 fils parallèles !!

Programmation parallèle imbriquée

- on exécute 2 fois la fonction de Fibonacci en parallèle

```
let n = try int_of_string Sys.argv.(1) with _ -> 1

let rec fib n =
  if n < 2 then 1 else begin
    let d1 = Domain.spawn (fun _ -> fib (n - 1)) in
    let d2 = Domain.spawn (fun _ -> fib (n - 2)) in
    Domain.join d1 + Domain.join d2
  end

let _ =
  let r = fib n in
  Printf.printf "fib(%d) = %d\n%!" n r
```

- on compile et exécute

```
% ocamlc -o fib_par fib_par.ml

% ./fib2 39
Fatal error: .. failed to allocate domain
```

- création de trop de domaines

Programmation parallèle imbriquée

- on exécute la fonction de Fibonacci en parallèle

```
let num_domains = int_of_string Sys.argv.(1)
let n = int_of_string Sys.argv.(2)

let rec fib n = if n < 2 then 1 else fib (n - 1) + fib (n - 2)

module T = Domainslib.Task

let rec fib_par pool n =
  if n > 20 then begin
    let a = T.async pool (fun _ -> fib_par pool (n-1)) in
    let b = T.async pool (fun _ -> fib_par pool (n-2)) in
    T.await pool a + T.await pool b
  end else fib n

let _ =
  let pool = T.setup_pool ~num_domains:(num_domains - 1) () in
  let res = T.run pool (fun _ -> fib_par pool n) in
  T.teardown_pool pool;
  Printf.printf "fib(%d) = %d\n" n res
```

`async` lance en parallèle asynchrone
`await` attend la fin de l'exécution

`pool` est l'ensemble de domaines
`run` exécute dans ces domaines
`teardown` supprime ces domaines

- on limite le nombre de domaines

Programmation parallèle imbriquée

- on compile et exécute

```
% ocamlfind ocamlc -package domainslib -linkpkg -o fib_par2 fib_par2.ml

% hyperfine './fib 42' './fib_par2 2 42' './fib_par2 4 42' './fib_par2 8 42'

Benchmark 1: ./fib 42
  Time (mean ± σ):      6.644 s ±  0.022 s    [User: 6.505 s, System: 0.002 s]
  Range (min … max):    6.591 s …  6.661 s    10 runs

Benchmark 2: ./fib_par2 2 42
  Time (mean ± σ):      3.544 s ±  0.026 s    [User: 6.800 s, System: 0.004 s]
  Range (min … max):    3.519 s …  3.594 s    10 runs

Benchmark 3: ./fib_par2 4 42
  Time (mean ± σ):      1.892 s ±  0.045 s    [User: 7.054 s, System: 0.007 s]
  Range (min … max):    1.767 s …  1.921 s    10 runs

Benchmark 4: ./fib_par2 8 42
  Time (mean ± σ):      1.367 s ±  0.063 s    [User: 9.886 s, System: 0.022 s]
  Range (min … max):    1.278 s …  1.438 s    10 runs

Summary
  ./fib_par2 8 42 ran
    1.38 ± 0.07 times faster than ./fib_par2 4 42
    2.59 ± 0.12 times faster than ./fib_par2 2 42
    4.86 ± 0.23 times faster than ./fib 42
```

Itérateur parallèle

- calcul d'une combinaison linéaire $v' \leftarrow A\,v$

```
let n = try int_of_string Sys.argv.(1) with _ -> 32

let multiply a v v'=
  let n = Array.length v - 1 in
  for i = 0 to n do
    let vi = ref 0. in
    for j = 0 to n do vi := !vi +. a.(i).(j) *. v.(j) done;
    v'.(i) <- !vi) ;;

let _ =
  let a = Array.make_matrix n n 1.0 in
    for i=0 to n-1 do for j=0 to n-1 do
        a.(i).(j) <- float (n*i + j)
      done done;
  let v = Array.init n (fun i -> 1. /. (float (i+1)))
   and v' = Array.create_float n in
  multiply a v v' ;
  let r = Array.fold_left (+.) 0. v' in
  Printf.printf "%f\n" r ;;
```

- opération courante dans un réseau de neurones

Itérateur parallèle

- souvent on peut exécuter une itération en parallèle

```
let num_domains = try int_of_string Sys.argv.(1) with _ -> 1
let n = try int_of_string Sys.argv.(2) with _ -> 32

module T = Domainslib.Task

let multiply pool a v v'=
  let n = Array.length v - 1 in
  T.parallel_for pool ~start:0 ~finish:n ~body:(fun i ->
    let vi = ref 0. in
    for j = 0 to n do vi := !vi +. a.(i).(j) *. v.(j) done;
    v'.(i) <- !vi) ;;

let _ =
  let pool = T.setup_pool ~num_domains:(num_domains - 1) () in
  let a = Array.make_matrix n n 1.0 in
    for i=0 to n-1 do for j=0 to n-1 do
        a.(i).(j) <- float (n*i + j)
      done done;
  let v = Array.init n (fun i -> 1. /. (float (i+1)))
   and v' = Array.create_float n in
  T.run pool (fun _ ->
     multiply pool a v v') ;
  T.teardown_pool pool;
  let r = Array.fold_left (+.) 0. v' in
  Printf.printf "%f\n" r ;;
```

`parallel_for` itérateur parallèle

`pool` est l'ensemble de domaines
`run` exécute dans ces domaines
`teardown` supprime ces domaines

- on limite le nombre de domaines

Itérateur parallèle

- souvent on peut exécuter une itération en parallèle

```
Benchmark 1: ./mult 16384
  Time (mean ± σ):      2.047 s ±  0.190 s    [User: 0.815 s, System: 0.747 s]
  Range (min … max):    1.951 s …  2.575 s    10 runs

Benchmark 2: ./mult_par 2 16384
  Time (mean ± σ):      1.508 s ±  0.127 s    [User: 0.834 s, System: 0.624 s]
  Range (min … max):    1.349 s …  1.782 s    10 runs

Benchmark 3: ./mult_par 4 16384
  Time (mean ± σ):      1.424 s ±  0.161 s    [User: 0.867 s, System: 0.785 s]
  Range (min … max):    1.210 s …  1.713 s    10 runs

Benchmark 4: ./mult_par 8 16384
  Time (mean ± σ):      1.355 s ±  0.081 s    [User: 1.029 s, System: 0.911 s]
  Range (min … max):    1.258 s …  1.497 s    10 runs

Summary
  ./mult_par 8 16384 ran
    1.05 ± 0.13 times faster than ./mult_par 4 16384
    1.11 ± 0.11 times faster than ./mult_par 2 16384
    1.51 ± 0.17 times faster than ./mult 16384
```

Processeurs parallèles

- le parallélisme permet d'effectuer plusieurs tâches simultanément
- les processeurs actuels ont plusieurs processeurs qui peuvent donc s'exécuter en parallèle
- avec N processeurs, on ne peut faire plus qu'un facteur N d'accélération
- mon PC a 4 processeurs (*cores*) peuvent exécuter 8 tâches parallèles
- les GPU (*graphic processor units*) ont des centaines de processeurs
- les *neural engines* ont plusieurs milliers de processeurs

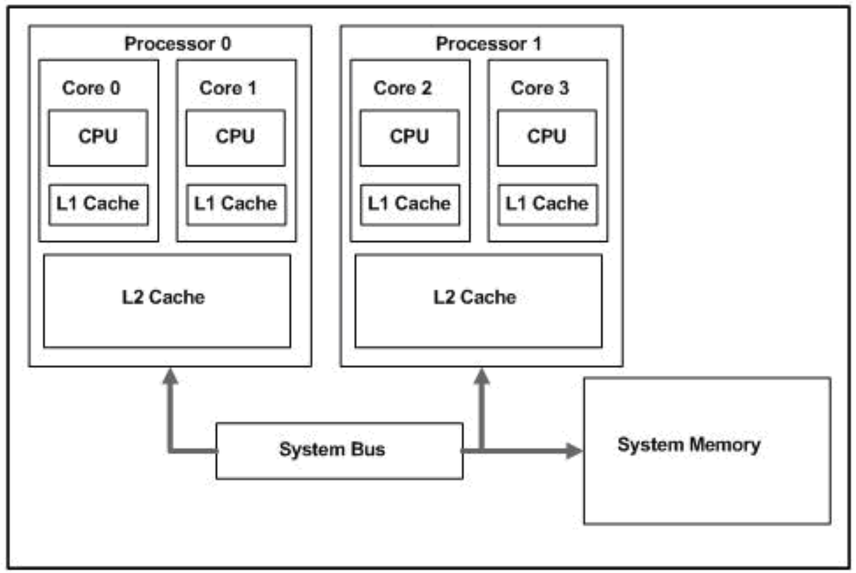
Architectures parallèles

• SIMD (*single instruction multiple data*) : tous les processeurs font la même opération

• MIMD (*multiple instruction multiple data*) : chaque processeur peut faire une opération différente

• la place de la mémoire par rapport aux processeurs est importante

- mémoire *on chip* avec des caches
- cohérence de la mémoire partagée



Opérateurs atomiques

- opérations indivisibles

```
let twice_in_parallel f =
  let d1 = Domain.spawn f in
  let d2 = Domain.spawn f in
  Domain.join d1;
  Domain.join d2 ;;

let plain_ref n =
  let r = ref 0 in
  let f () = for _i=1 to n do incr r done in
  twice_in_parallel f;
  Printf.printf "Non-atomic ref count: %d\n" !r ;;

let atomic_ref n =
  let r = Atomic.make 0 in
  let f () = for _i=1 to n do Atomic.incr r done in
  twice_in_parallel f;
  Printf.printf "Atomic ref count: %d\n" (Atomic.get r) ;;

let _ =
  let n = try int_of_string Sys.argv.(1) with _ -> 1 in
  plain_ref n;
  atomic_ref n ;;
```

```
Atomic.make
Atomic.incr
Atomic.get
```

Opérateurs atomiques

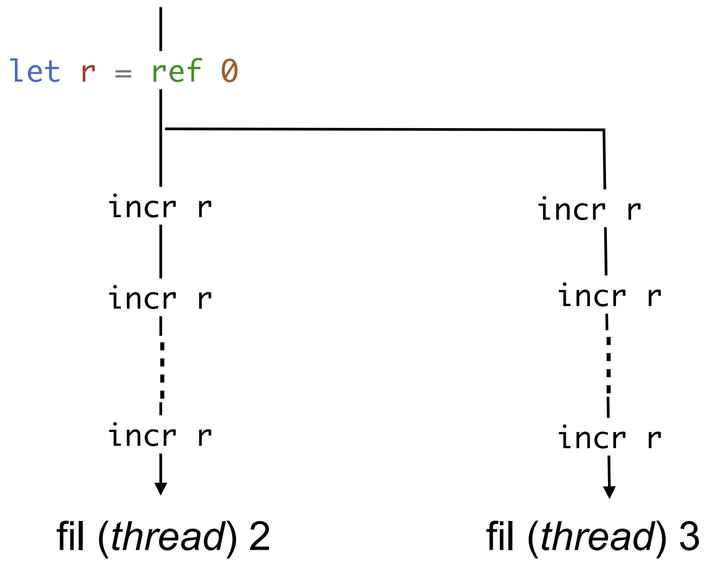
- on compile et exécute

```
% ocamlopt -o incr_atom incr_atom.ml

% ./incr_atom 1_000_000
Non-atomic ref count: 842372
Atomic ref count: 2000000
```

- `incr` n'est pas atomique

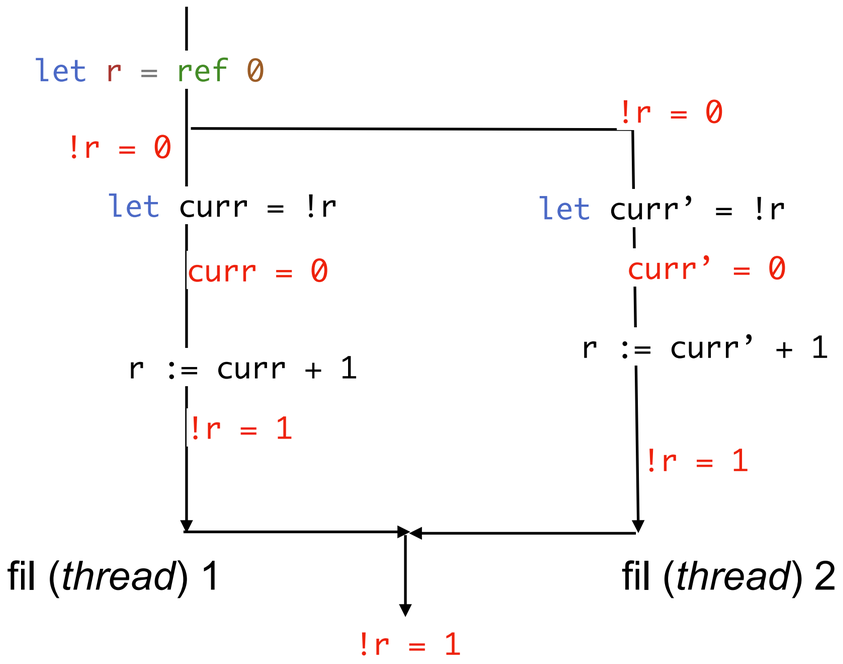
```
let incr r =
  let curr = !r in
        r := curr + 1
```

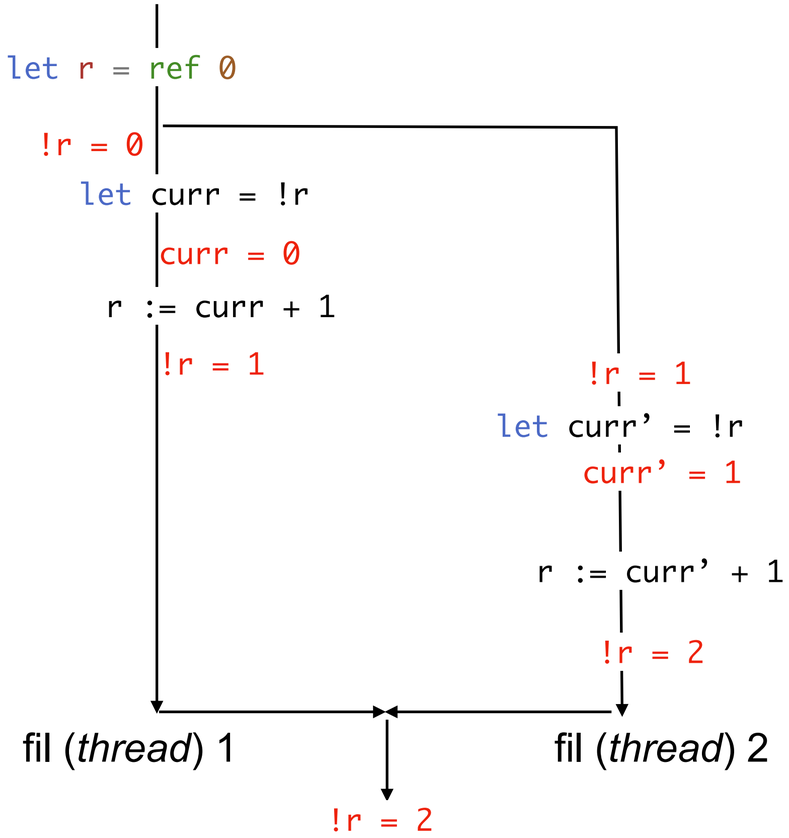


Mémoire partagée

- `incr` n'est pas atomique

```
let incr r =
  let curr = !r in
        r := curr + 1
```





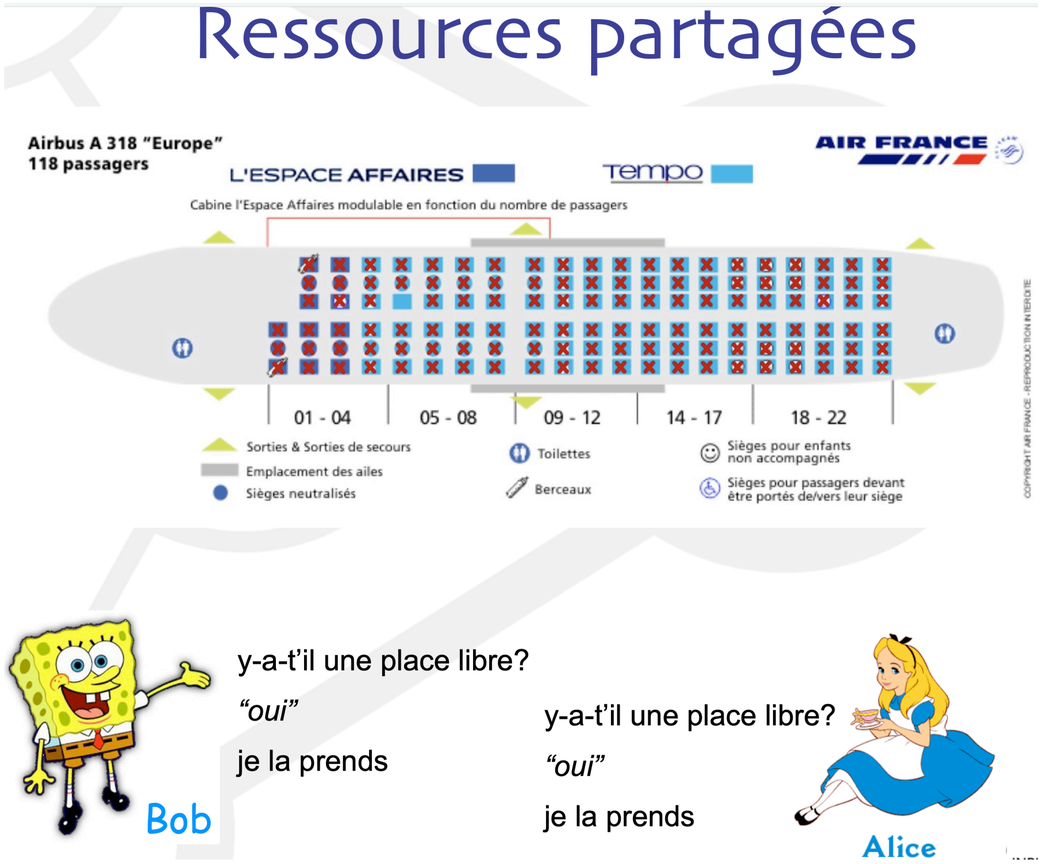
Concurrence

- sections critiques (sémaphores, mutuelle exclusion)
- moniteurs (sémaphores, conditions)
- *dead locks (*interblocage)
- famine
- (non) déterminisme
- vérification de programmes concurrents
- . . .

Mémoire partagée

- la mémoire partagée non modifiable peut être traitée en parallèle
- la mémoire partagée modifiable pose des problèmes de cohérence
- aucun problème si chaque processeur modifie des mémoires séparées
- les processus concurrents ont été longuement étudiés
  - avec mémoire partagée
  - avec mémoire distribuée (sur un réseau)

Ressources partagées



Langages fonctionnels

- OCAML et autres langages de programmation fonctionnels

  - LISP [1960, McCarthy], MIT
  - ML (SML, SMLNJ, polyML) [1980, Milner] U. of Edinburgh
  - Haskell (LML, GHC) [1990, Peyton Jones] Microsoft ➔ Excel
  - F# [2002, Syme] Microsoft
  - Erlang [1986, Amstrong] Ericsson
  - Clojure [2007, Hickey] Java
  - Scala [2003, Odersky] EPFL
  - Wolfram language, Clean, Oz, CDuce, PureScript, Agda, …

concepts adaptés dans d’autres langages non fonctionnels

Conclusion

VU:

- langage de programmation fonctionnel ==> parallélisme
- typage fort ==> sécurité
- écriture légère (avec synthèse des types)
- modularité ==> sécurité
- compilation séparée ==> gros logiciels

vive l'informatique

et

la programmation !