# Rentrée

- dernier cours et TP: petite initiation à Coq
  - mardi 8/1 de 8h à 10h: cours
  - mardi 8/1 de 10h15 à 12h15, et les autres créneaux habituels: TP
- examen: le sujet de l'an dernier est disponible sur la page

# Parallélimse

merci: JJ Lévy, L Marchal

# Programmes parallèles

- un programme est *parallèle* s'il y a *plusieurs flots d'exécution*

  *(par opposition à un programme séquentiel)*

- les programmes parallèles ressemblent en première approche à une composition *(mise en parallèle)* de programmes séquentiels (ici, impératifs)

$$\left(\begin{array}{l} \texttt{x := y+1;} \\ \texttt{z := x;} \end{array}\right) \quad || \quad \left(\begin{array}{l} \texttt{t := f(u);} \\ \texttt{t' := 52;} \end{array}\right)$$

  *(langage impératif imaginaire)*

- deux "directions"
  - composition séquentielle ;
  - composition parallèle ||
- exemple élémentaire: paralléliser pour accélérer

```
a := x+32; y := t*52; b := u*t; k := k+1;
```

  est remplacé par

```
( (a := x+32;|| j := t;) ; (y := j*52;|| b := u*j) ) ||  k := k+1;
```

# Modèle physique sous-jacent

on ne va pratiquement pas parler ici . . .

- de différentes architectures pour les parallélismes
  - du parallélisme matériel (au sein d'un processeur) aux supercalculateurs, et aux approches en réseau (peer to peer, la grille, etc.)
  - point faible de ce que je vais raconter: le modèle d'architecture sous-jacent est souvent *déterminant* *(un bout de la lorgnette)*
- de questions algorithmiques et/ou pragmatiques liées à l'exécution de calculs bestiaux (parallélisme de données, questions d'équilibrage de charge, etc.)
- cf. en particulier
  - systèmes, Eric Fleury, L3IF.2,
  - systèmes distribués, Eddy Caron, MIF
  - algorithmique parallèle, Anne Benoit, MIF

# Programmes parallèles: fonctions, équivalences

- monde impératif et fonctionnel: un programme peut être compris comme une fonction
  - dans le cas impératif, une fonction faisant passer d'un état à un autre état
- après `x := 1;` || `x := 2;`, on a $x \in \{1, 2\}$

  *(les deux composantes parallèles accèdent à la même référence)*

  BOUM:
  - le résultat n'est pas unique (perte de la confluence)
  - un programme n'est pas décrit par une fonction
- équivalence entre programmes?
  - `P` $\stackrel{def}{=}$ `x := 0; x := x+1;` et `Q` $\stackrel{def}{=}$ `x := 1;` sont équivalents en tant que programmes séquentiels
  - REBOUM: mais `P` || `Q` et `Q` || `Q` ne sont pas équivalents

# Programmation parallèle

- les programmes parallèles sortent “beaucoup” du cadre séquentiel
  - non déterminisme
  - non terminaison (p.ex. un serveur)
  - des irrégularités apparaissent
- comprendre (*définir*) “ce que fait” un programme parallèle est difficile
  - le parallélisme peut être une manière d’écrire un programme (calcul) séquentiel
  - on peut aussi vouloir développer des applications intrinsèquement parallèles *(sans référence séquentielle)*
- nous allons ici évoquer
  - des techniques qui ont été développées pour
    - écrire des programmes sensés
    - concevoir des langages sensés
  - des concepts qui ont été proposés pour exprimer et analyser les questions liées à l’exécution de programmes parallèles

*Mémoire partagée, exclusion mutuelle*

# Modèle, interaction

- mémoire partagée: les flots qui s'exécutent en même temps peuvent accéder à une partie commune de la mémoire
  - plusieurs processeurs sur la même machine
  - des threads s'exécutant sur un même processeur
  - généralement le compilateur traduit les variables en des adresses mémoire *globales*
  - . . . : on reste vague sur les considérations matérielles
- interaction:
  - au début, `x`$= 0$
  - puis on exécute `(x := x+1; x := x+1;)` || `x := 2*x;`
  - à la fin, `x`$\in \{2, 3, 4\}$ *les deux composantes peuvent interagir en* `x`

# Atomicité

- atomicité:
  - au début, `x`$= 0$
  - après l'exécution de `x := x+1;` || `x:= x+1;`, on a `x`$= 2$
  - mais si `x := x+1;` est compilé en `R := x; x:= R+1;`
    BOUM
- la description des exécutions d'un programme parallèle suppose la définition de ce qu'est l'*atome*, i.e. l'opération indivisible dans un flot: entre le début et la fin d'un événement atomique, rien ne peut se passer dans les autres flots
  - typiquement, le matériel définit l'atome: un accès mémoire est atomique, pas une multiplication sur un format avec grande précision

# Sections critiques, exclusion mutuelle

- deux portions de programmes accédant à des ressources communes

  $P_0 \;=\; \ldots C_0 \ldots \qquad P_1 \;=\; \ldots C_1 \ldots$

  - $C_0$ et $C_1$ sont des *sections critiques*
  - on veut "rendre ces sections *atomiques*", pour éviter les interférences
  - exclusion mutuelle: $C_0$ et $C_1$ ne s'exécutent jamais en même temps *(ne font jamais la course)*
  - on a intérêt à réduire la taille de $C_0$ et $C_1$, afin de ne pas trop réduire le parallélisme
  - il se peut que l'on doive passer *plusieurs fois* dans $C_0$ et/ou $C_1$

- première solution: au début, `turn = 0`

```
while turn <> 0 do (* rien *) ;
C0
turn := 1;
```

```
while turn <> 1 do (* rien *) ;
C1
turn := 0;
```

  - pas très satisfaisant: ordonnancement imposé

# D'autres tentatives

- au début, `a0 = a1 = false`

```
while a1 do (* rien *) ;
a0 := true ;
C0;
a0 := false;
```

```
while a0 do (* rien *) ;
a1 := true ;
C1;
a1 := false;
```

  - pas très correct

- au début, `a0 = a1 = false`

```
a0 := true;
while a1 do (* rien *) ;
C0;
a0 := false;
```

```
a1 := true;
while a0 do (* rien *) ;
C1;
a1 := false;
```

  - risques d'*interblocage*: chaque flot attend l'autre

# Algorithme de Dekker (1965)

- au début, `a0 = a1 = false`, `turn` $\in \{0, 1\}$

```
a0 := true;
while a1 do
  if turn <> 0 then begin
    a0 := false;
    while turn <> 0 do (* rien *) ;
    a0 := true;
  end;
```
$C_0$
```
turn := 1; a0 := false;
```

```
a1 := true;
while a0 do
  if turn <> 1 then begin
    a1 := false;
    while turn <> 1 do (* rien *) ;
    a1 := true;
  end;
```
$C_1$
```
turn := 0; a1 := false;
```

- "algorithme", ou *protocole*
- il y a de l'*attente active* : un flot interroge constamment la valeur de `turn` tandis que l'autre s'exécute
- si l'ordonnancement est *équitable*, l'algorithme l'est

# Exécutions, entrelacement, équité

- on peut décrire **les** exécution**s** d'un programme en considérant tous les *entrelacements* possibles d'atomes: l'ordre est fixé au sein d'un flot, pas entre les flots *(cf. threads)*
- diverses formes de *synchronisation* peuvent être mises en œuvre pour contrôler l'ordre d'exécution entre les flots
- l'*ordonnancement* détermine 'qui a droit à un tour de manège' parmi les flots susceptibles de s'exécuter à un moment donné
  - cela peut être un programme ou un oracle
- **une** version de l'équité:

  si à partir d'un certain moment une instruction est susceptible d'être exécutée *en continuation*, alors elle le sera un jour *(+ de nombreuses déclinaisons)*

# Algorithme de Peterson (1981)

- au début, `a0 = a1 = false`

```
a0 := true ;                                  a1 := true;
turn := 1;                                    turn := 0;
while a1 and turn <> 0 do (* rien *);         while a0 and turn <> 1 do (* rien *);
C0                                            C1
a0 := false;                                  a1 := false;
```

- plus compact et simple à justifier que l'algorithme de Dekker
- attente active encore

montons d'un niveau: *primitives pour la synchronisation*

*Primitives pour la synchronisation*

# Sémaphores

- un *sémaphore généralisé* est une variable entière `s` avec deux opérations *(sémaphore tout court: `s=1`)*
  - `acquire(s)`:
    - si `s`$>0$ alors `s := s-1` — de manière atomique
    - sinon rester bloqué sur `s`
  - `release(s)`:
    - si un flot est bloqué sur `s`, le réveiller
    - sinon `s := s+1` — de manière atomique
- l'exclusion mutuelle devient naturelle: on part de `s=1`

  `...acquire(s);` $C_0$ `; release(s);...`

  $\|$ `...acquire(s);` $C_1$ `; release(s);...`
- néanmoins, risques d'*interblocage*
  - si $A$ verrouille $s1$, $B$ verrouille $s2$, puis $A$ veut $s2$ et $B$ veut $s1$
  - discipline courante: un ordre partiel *(relation réflexive transitive antisymétrique)* sur les sémaphores, respectée au sein de chaque flot

# Primitives pour la programmation concurrente

- librairies (Modula, POSIX threads, . . . )
  - on pense à des *threads* (*tâches*) s'exécutant sur un ensemble (éventuellement singleton) de processeurs
- entités
  - un *mutex* est libre, ou verrouillé par un thread; naît libre
  - une *condition* est un ensemble de threads en attente; naît vide
- fonctions
  - `acquire(m)`: attend que le mutex `m` soit libre et le verrouille
  - `release(m)`: déverrouille le mutex `m`
  - `wait(m,c)`: déverrouille le mutex `m` et attend sur la condition `c` atomiquement; ensuite revérouille `m` et rend la main
  - `signal(c)`: un ou plusieurs threads attendant la condition `c` peuvent redémarrer
  - `broadcast(c)`: *tous* les threads attendant la condition `c` peuvent redémarrer

# Les cinq philosophes chinois (1)

- cinq philosophes $(\Phi_0, \ldots, \Phi_4)$ qui pensent et mangent, cinq baguettes à disposition sur une table ronde
- problème dû à Dijkstra pour tester les primitives concurrentes: verrous, conditions, sémaphores, sémaphores généralisés
- première solution: un tableau `s` de mutex

```
PROCEDURE Philosophe (i) =
BEGIN
  (* penser *)
acquire(s[i]);
acquire(s[(i+1) % 5]);
  (* manger *)
release(s[i]);
release(s[(i+1) % 5]);
END Philosophe;
```

- risques d'interblocage

# Philosophes chinois (2)

- deuxième solution

```
PROCEDURE Philosophe (i) =
BEGIN
while true do begin
  (* penser *)
  PrendreBaguettes(i);
  (* manger *)
  RelacherBaguettes(i);
end;
END Philosophe;
```

- on se donne les tableaux suivants
  - `f`, avec `f[i]` le nombre de fourchettes disponibles pour $\Phi_i$
  - `manger`, un tableau de conditions

# Philosophes chinois (3)

```
PROCEDURE PrendreBaguettes(i) =
BEGIN
acquire(m)
  while f[i] != 2 do
    wait (m, manger[i]);
  f[(i-1) % 5] := f[(i-1) % 5]-1;
  f[(i+1) % 5] := f[(i+1) % 5]-1;
release(m);
END PrendreBaguettes;
```

```
PROCEDURE RelacherBaguettes(i) =
BEGIN
VAR g := (i-1)%5, d := (i+1)%5;
acquire(m);
  f[g] := f[g]+1; f[d] := f[d]+1;
  if f[d] = 2 then
    signal(manger[d]);
  if f[g] = 2 then
    signal(manger[g]);
release(m);
END RelacherBaguettes;
```

- l'invariant $\Sigma_{i=0}^{4} f[i] = 10 - 2 * \mathit{mangeurs}$ est vérifié
- si interblocage, alors $\mathit{mangeurs} = 0$, et donc $\forall i.\, f[i] = 2$
  donc pas d'interblocage pour le dernier à demander à manger
- *famine* possible, si par exemple $\Phi_1$ et $\Phi_3$ complotent contre $\Phi_2$, qui mourra de faim

# Philosophes chinois (4)

- on reprend la première solution + sémaphore généralisé `salle` au début `salle = 4`
  - pour manger, les philosophes rentrent dans la salle
  - il y a au plus 4 philosophes dans la salle
  - ils sortent de la salle après le repas
  - et retournent penser dans leur cellule (ce sont en fait des moines philosophes)

```
PROCEDURE Philosophe (i) =
BEGIN
  (* penser *)
acquire(salle) ; (* ------ début zone critique ---- *)
acquire(s[i]);
acquire(s[(i+1) % 5]);
  (* manger *)
release(s[i]);
release(s[(i+1) % 5]);
release(salle) ; (* ------ fin zone critique ---- *)
END Philosophe;
```

# Philosophes chinois (5)

- 4 philosophes au plus dans la salle $\Rightarrow$ pas d'interblocage
- l'invariant suivant est vérifié:
  $$\text{salle} + \text{nombre de tâches dans la zone critique} = 4$$
- si $\Phi_i$ exécute `acquire(s[i])`, alors il finira cette instruction
- si $\Phi_i$ attend indéfiniment sur `acquire(s[(i+1)%5])`, alors $\Phi_{i+1}$ attend indéfiniment sur `acquire(s[(i+2)%5])`
- si $\Phi_i$ exécute `release(s[(i+1)%5])`, alors il finira cette instruction
- pas de famine

*Passage de messages*

# Un autre modèle

- montons encore d'un niveau: on laisse tomber le modèle de la mémoire partagée, les flots se *communiquent* des données à l'aide de primitives
- on travaille en fait volontiers avec des *architectures hybrides*: un réseau connectant des machines ayant chacune plusieurs processeurs
  - localement: mémoire partagée
  - entre les machines: communications

- est-il possible d'aller encore plus loin?
  *modèles de haut niveau, unifiant les approches?*

# Diverses formes d'interaction à distance

- communications point à point (entre 2 machines)
  - envois asynchrones (les canaux sont des files)
  - envois synchrones: l'envoi est bloquant
- le standard le plus utilisé est MPI (*Message Passing Interface*)
  - librairies (Fortran, C, C++) – on 'programme l'interaction'

`send( dest, type, address, length)`

· `dest` is an integer identifier representing the process to receive the message.

· `type` is a nonnegative integer that the destination can use to selectively screen messages.

· `(address, length)` describes a contiguous area in memory containing the message to be sent.

- des langages: Erlang, Occam
- invoquer un calcul à distance: RPC (Remote Procedure Call), la RMI (Remote Method Invocation) de Java
  - ex. de mise en œuvre: Corba – mettre en relation des serveurs de calcul qui proposent des méthodes
- NB: l'interaction n'a pas nécessairement lieu à distance
  - il existe des utilisations monoprocesseur des librairies pour le passage de messages *(si le code se conçoit naturellement comme ça)*

# La librairie `Event` de Caml

- adaptation de la librairie de J. Reppy, développée pour SML
  - Concurrent ML, qui a notamment été utilisé pour développer le système de fenêtres X eXene
- les événements sont l'entité élémentaire pour programmer la synchronisation *(cette vision se marie bien avec ML)*
- interface:

```
type 'a channel
val new_channel :  unit -> 'a channel
type 'a event
val send :  'a channel -> 'a -> unit event
val receive :  'a channel -> 'a event
val choose :  'a event list -> 'a event
val wrap :  'a event -> ('a -> 'b) -> 'b event
val sync :  'a event -> 'a
val select :  'a event list -> 'a

...
```

- exemples: DÉMO `events.ml, cell[1,2,3].ml`

# JoCaml – présentation

- une extension de Caml avec des idées issues du *join calcul*
  - le join calcul est issu de la recherche autour du $\pi$-calcul
  - on décante un tel calcul en quelques nouvelles primitives, bien comprises, et suffisamment expressives
- mécanisme de base: passage de messages (sur des canaux)
  - canaux asynchrones
    - `def affiche (x) = print_int x; print_string "!\n"; 0`
      - ♢ `affiche` est une *définition de processus*
      - ♢ 0 est ici "stop", le processus terminé
    - `spawn (affiche (27) & affiche (19))`
      - ♢ `affiche (i)` : émission sur le canal `affiche`
      - ♢ `&` : composition parallèle
  - canaux synchrones
    - l'idée est de programmer des fonctions à l'aide des émissions asynchrones
    - passage de continuations: `def f (x,r) = r (x+3)`
      - ♢ "canaux d'ordre supérieur" : $\pi$-calcul
    - "en dur" dans le langage:
      `def f(x) = reply (x+3) to f`

# JoCaml, suite

- les fonctions comme canaux synchrones ont ceci de particulier qu'elles peuvent être définies en utilisant des *join patterns*

```
def cellule (x0) =
   def cell(x) & get() = reply x to get & cell(x)
   or cell(x) & put(y) = reply to put & cell(y)
in
   cell(x0) & reply (get,put) to cellule
```

- la construction `def` a un double rôle
  - définir les noms de canaux
    - ici, `cell get put` d'une part, `cellule` d'autre part
  - définir les processus qui écoutent sur ces canaux
- les utilisateurs d'un `def` ne peuvent qu'émettre sur les noms qui ont été définis
  - et qu'on leur a transmis : ici `cell` n'est pas accessible depuis l'extérieur
  - ils peuvent sinon transmettre ces noms ailleurs

# Jocaml, exemples

- `iter_list.ml`
- `barriere.ml`
- `sched.ml`