Programmation concurrente

Yann **Régis-Gianas**
yrg@pps.univ-paris-diderot.fr

PPS - Université Denis Diderot – Paris 7

12 mars 2015

Plan

Plan

Le non-déterminisme

Non déterminisme

Le **déterminisme** est l'idée que, sous un certain angle, un événement est entièrement déterminé par les événements qui le précèdent. On dit qu'il existe une relation de **causalité** entre ces événements et l'événement considéré. Le **non déterminisme** est l'opposé du déterminisme : dans un système non déterministe, on ne peut qu'énoncer un **ensemble de possibles** suite à une séquence d'événements passés.

Exemples

- Dans un système chaotique, des situations initiales arbitrairement proches peuvent conduire à des conséquences arbitrairement distantes.
- Une suite d'événements régie par une loi de distribution aléatoire est une forme de non déterminisme.
- Le temps d'accès à un bloc de données d'un disque dur peut est aussi non-déterministe car il dépend de frottements mécaniques difficilement prédictibles.

Le non-déterminisme en informatique : un paradoxe ?

Un programme informatique est une machine logique, par essence déterministe. Est-ce que le non-déterminisme n'est pas exclus d'emblée par cette situation ?

Tout d'abord, ce serait dommage puisque **la plupart des programmes sont utilisés dans un environnement hautement non-déterministe**. Par exemple, les entrées fournies interactivement par un utilisateur arrivent dans un ordre non prédictible. Un autre exemple : en réalité, les composants internes de la machine, de part leur nature physique, ont un comportement non-déterministe.

Ensuite, un système déterministe peut très bien **simuler un système non déterministe**. Par exemple, vous savez transformer un automate fini non déterministe en automate déterministe.

La véritable question est donc de comprendre comment simuler le non déterminisme dans un programme et bien sûr, comment raisonner sur le comportement de ce programme. En effet, pour ce dernier point, la notion de flot de contrôle que nous avons disséquée dans les précédents cours semble *a priori* inadéquate puisque le **modèle naturel d'un système non déterministe est un ensemble de flots d'exécution possibles non déterminé à l'avance**.

Les différentes formes de non-déterminisme

Pour pouvoir simuler du non-déterminisme, il faut d'abord se rendre compte qu'il en existe plusieurs formes :

- **Non déterminisme avec connaissance nulle** (*Don't know*)
- **Non déterminisme avec connaissance partielle** (*I know a bit*)
- **Non déterminisme assumé** (*Don't care*)

1. ce qui est faux, bien sûr !
2. Ce qui est encore faux, bien sûr !

Les différentes formes de non-déterminisme

Pour pouvoir simuler du non-déterminisme, il faut d'abord se rendre compte qu'il en existe plusieurs formes :

- **Non déterminisme avec connaissance nulle** (*Don't know*)
  On représente un ensemble de possibles sans plus d'information. On doit supposer à chaque instant que tous les choix sont valides.
  Exemple : "Envoyez-moi tous un email avec vos projets."
  Si j'énonce cette directive, les étudiants peuvent m'envoyer leur projet et je ne m'occupe pas de l'ordre dans lequel ils arrivent. Je dois prendre en compte toutes les permutations possibles.
- **Non déterminisme avec connaissance partielle** (*I know a bit*)
- **Non déterminisme assumé** (*Don't care*)

1. ce qui est faux, bien sûr !
2. Ce qui est encore faux, bien sûr !

Les différentes formes de non-déterminisme

Pour pouvoir simuler du non-déterminisme, il faut d'abord se rendre compte qu'il en existe plusieurs formes :

- **Non déterminisme avec connaissance nulle** (*Don't know*)
- **Non déterminisme avec connaissance partielle** (*I know a bit*)
  L'ensemble des possibles est conditionné par des lois externes, comme par exemple une loi de distribution de probabilités.
  Exemple : "Je note les étudiants en lançant deux dés à dix faces." [1]
  Si je suis cette instruction, je ne sais pas exactement qu'elle est la note de chaque étudiant mais j'ai des informations globales sur la répartition des notes.
- **Non déterminisme assumé** (*Don't care*)

1. ce qui est faux, bien sûr !
2. Ce qui est encore faux, bien sûr !

Les différentes formes de non-déterminisme

Pour pouvoir simuler du non-déterminisme, il faut d'abord se rendre compte qu'il en existe plusieurs formes :

- **Non déterminisme avec connaissance nulle** (*Don't know*)
- **Non déterminisme avec connaissance partielle** (*I know a bit*)
- **Non déterminisme assumé** (*Don't care*)
  À un instant donné, il y a bien un ensemble de possibles, mais on fait un choix arbitraire d'un événement parmi ces possibles car cela n'a pas d'importance sur la suite du calcul.
  Exemple : "L'un de vous fera le cours à ma place la semaine prochaine." [2]

1. ce qui est faux, bien sûr !
2. Ce qui est encore faux, bien sûr !

Spécification incluant du non déterminisme

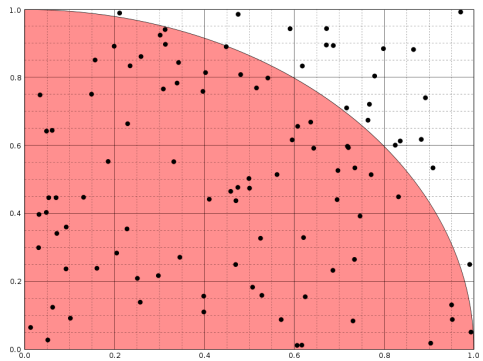
On utilise l'opérateur $+$ pour modéliser un choix non déterministe "Don't know".
Par exemple, pour un algorithme de recherche dans un domaine $\mathcal{D}$, on peut spécifier la recherche d'une solution dans deux sous-domaines disjoints $\mathcal{D}_1$ et $\mathcal{D}_2$, en écrivant $find_{answer}\ D_1\ +\ find_{answer}\ D_2$.

On utilise l'opérateur *choice* pour exprimer un choix "Don't care".
Par exemple, si on souhaite prendre la prochaine tâche à effectuer dans une liste de tâches indépendantes, on écrira `choice t into tasks in ...`.

Exemple d'algorithme probabiliste : calcul de π

Les méthodes de monte-carlo sont des méthodes numériques probabilistes pour évaluer une quantité numérique à l'aide d'un échantillon choisi aléatoirement dans le domaine de la fonction qui la caractérise.

Pour calculer π, on peut tirer des points (x, y) au hasard dans le domaine $[0, 1]^2$. Si $x^2 + y^2 < 1$ alors le point est à l'intérieur du quart de cercle. En faisant N tirages, si on obtient K points à l'intérieur, alors le rapport $\frac{K}{N}$ est une approximation de $\frac{\pi}{4}$.



Exercice

Spécifiez cet algorithme.

Comment programmer le non-déterminisme ?

Pour programmer le non-déterminisme *Don't know*, il faut être capable de représenter l'ensemble des flots d'exécution possibles, chacun prenant en compte une certaine possibilité. Pour cela, on doit se donner un moyen de représenter des **processus**, c'est-à-dire des fragments du programme qui s'exécutent de façon **concurrente**.

Pour programmer le non-déterminisme *Don't care*, il faut se donner un mécanisme pour choisir un représentant parmi les possibles. Cela signifie qu'il faut pouvoir observer les évaluations d'un ensemble de processus pour décider de ce que l'on fait ensuite. C'est donc un **procédé de communication** qui est nécessaire ici. Il sert essentiellement à restaurer un ordre entre les calculs.

Mécanismes calculatoires de la programmation concurrente

Dans la suite du cours, nous allons aborder les mécanismes calculatoires permettant d'écrire des programmes concurrents. Nous verrons que le raisonnement sur les programmes concurrents est complexe et pourquoi certains de ces mécanismes apportent plus de garanties que d'autres.

Introduction

Créer des processus

Restaurer un ordre global : la communication synchrone

Instaurer un ordre plus libertaire : la communication asynchrone

Plan

Les processus, une opportunité pour la parallélisation

Nous avons déjà parlé des co-routines, ces composants logiciels qui peuvent s'interrompre et transférer le contrôle à d'autres composants. Ce sont des **opérations suffisantes pour simuler du non déterminisme sur un unique processeur déterministe**. Dans ce cadre, la communication entre processus implémentés par des co-routines peut s'effectuer simplement à l'aide de variables globales (notez qu'il n'y a pas de problème d'accès concurrent aux ressources puisque les co-routines collaborent) ou *via* les arguments formels et les retours de fonctions.

Cependant, les machines modernes sont de plus en plus souvent formées de plusieurs processeurs, **capable d'exécuter des instructions différentes au même instant**. Les systèmes d'exploitations fournissent deux mécanismes pour qu'un programme puisse tirer parti de ces multiples processeurs : les fils d'exécution (*threads*) et les processus. L'avantage de ces mécanismes, c'est qu'ils augmentent les capacités de calcul. Leur inconvénient principal, c'est qu'ils introduisent des problèmes d'accès concurrents aux ressources de la machine. La mise en réseau des processeurs a augmenté de façon spectaculaire la capacité de calcul : un même programme peut désormais lancer des processus sur un très grand nombre de machines et les faire communiquer *via* une interface réseau. Dans ce cadre là, la probabilité d'une panne est non négligeable, ce qui rajoute un impératif de résistance aux pannes des programmes.

Les langages de programmation pour la concurrence

L'hétérogénéité des implémentations de processus rend extrêmement complexe l'implémentation de programmes concurrents, tant du point de vue de la portabilité que de la correction. C'est pourquoi les langages de programmation ont introduit des **abstractions** permettant d'ignorer les implémentations sous-jacentes des processus.

Pour commencer notre étude de ces abstractions, nous allons écrire un programme à l'aide de la bibliothèque d'**acteurs** de SCALA.

Exercice

Après avoir lu cette documentation :

http://www.scala-lang.org/docu/files/actors-api/actors_api_guide.html

Implémentez le calcul de π décrit plus haut sur un processeur puis en parallélisant votre calcul sur tous les cores de votre machine, et enfin sur plusieurs machines du réseau.

Décomposition d'un calcul par processus

C'est la notion de dépendances entre les calculs qu'il est essentiel d'expliciter pour déterminer une décomposition correcte et efficace d'un calcul à l'aide de processus. Il y a différents cas de figure :

- Des calculs indépendants
- Un calcul dépendant des résultats de plusieurs sous-calculs
- Plusieurs calculs dépendent de l'état d'un processus

Dans toutes ces situations, une **couche de communication** inter-processus plus ou moins riche doit être mis-en-œuvre. Comme pour toute communication, cela signifie qu'il faut se donner un media, c'est-à-dire un **protocole** pour lier emetteur et récepteur, ainsi qu'un langage de description des **messages** transportés par ce protocole. C'est très précisément sur cet aspect que l'on peut distinguer différents styles de programmation concurrente. Nous allons les décrire dans la suite du cours.

Décomposition d'un calcul par processus

C'est la notion de dépendances entre les calculs qu'il est essentiel d'expliciter pour déterminer une décomposition correcte et efficace d'un calcul à l'aide de processus. Il y a différents cas de figure :

- Des calculs indépendants : dans le cas d'un non déterminisme purement "*don't care*", les différents flots d'exécution possibles peuvent être englobés en évaluant les sous-calculs en parallèle sans avoir à mettre en place un mécanisme de communication intervenant durant l'exécution des différents processus. Ce cas se présente par exemple lorsqu'un même calcul doit être effectué sur des fragments indépendants de l'entrée (parallélisme de données) ou bien encore lorsque l'ordre dans lequel deux appels récursifs peuvent être effectués dans un ordre arbitraire (comme dans le cas du calcul de π précédent).
- Un calcul dépendant des résultats de plusieurs sous-calculs
- Plusieurs calculs dépendent de l'état d'un processus

Décomposition d'un calcul par processus

C'est la notion de dépendances entre les calculs qu'il est essentiel d'expliciter pour déterminer une décomposition correcte et efficace d'un calcul à l'aide de processus. Il y a différents cas de figure :

- Des calculs indépendants
- Un calcul dépendant des résultats de plusieurs sous-calculs : dans le cas où l'entrée d'un calcul dépend de la sortie de plusieurs autres, il est nécessaire de mettre en attente ce dernier. Il peut alors être utile de décomposer de nouveau ces différents calculs de façon à ce qu'ils puissent travailler sur des résultats partiels. Un cas d'architecture particulièrement efficace et simple à mettre en œuvre dans ce cas de figure est le *pipeline* : chaque processus travaille **incrémentalement** sur un **flot** de données. Dans le cas de figure idéal, tous les processus peuvent alors être actifs en même temps.
- Plusieurs calculs dépendent de l'état d'un processus

Décomposition d'un calcul par processus

C'est la notion de dépendances entre les calculs qu'il est essentiel d'expliciter pour déterminer une décomposition correcte et efficace d'un calcul à l'aide de processus. Il y a différents cas de figure :

- Des calculs indépendants
- Un calcul dépendant des résultats de plusieurs sous-calculs
- Plusieurs calculs dépendent de l'état d'un processus : c'est ici le cas de figure d'un accès concurrent à une ressource. C'est sans doute la situation la plus difficile à décomposer puisque des problèmes complexes d'interblocage ou de famine peuvent apparaître. Il est alors souvent nécessaire d'établir un protocole de négociation globale des ressources limitant le potentiel parallélisme du programme.

Problématique du raisonnement en concurrence

Le modèle de communication hérité immédiatement de l'architecture de Von Neumann consiste à augmenter de façon arbitraire le nombre des processeurs disponibles sur la machine. On peut imaginer alors qu'à la création d'un processus, un nouveau processeur est créé spécialement pour lui. Dans ce modèle de calcul, les processeurs peuvent communiquer *via* la mémoire globale. On dit que c'est une **communication par mémoire partagée**.

Le problème classique de ce modèle apparaît avec l'exemple suivant :

```
P1:
(*  y = x + 1  *)
incr y;
(*  y >= x + 2  *)
```

||

```
P2:
(* ... *)
x := 42
(* ... *)
```

En fonction de l'entrelacement de l'évaluation des processus, les suppositions du programmeur peuvent être compromises ! Il faut établir un protocole de **non interférence** entre ces deux processus explicitant le fait que le programmeur de P1 souhaite considérer la commande `incr y` comme **atomique**.

Spécification des attentes et garanties des processus

Le programmeur du processus P1 précédent aimerait spécifier **une fois pour toute** ce qu'il attend de l'environnement dans lequel il sera exécuté :

```
(*  y = x + 1  *)
(* Please, do what you want but do not increment "x" ! *)
incr y ;
(*  y >= x + 2  *)
```

Au contraire, le programmeur peut aussi
spécifier ce qu'il garantit aux autres processus :

```
incr y ;
if !y ≥ 0 then f () else g () ;
(* I will not modify "y" in the sequel. *)
```

Il existe une logique de Hoare adaptée au raisonnement dans ce cadre.

Mécanismes de réalisation des spécifications concurrentes

Les méthodes “classiques” d’implémentation des spécifications de la forme précédentes s’appuient sur une notion de **section critique** implémentée par des **verrous** ou des **sémaphores**. L’idée de ces mécanismes, que vous avez déjà manipulés, consiste à **publier de façon atomique la prise de possession exclusive** d’une ressource partagée, généralement un fragment de la mémoire faisant échouer ou en rendant bloquante toute tentative d’accès à cette ressource par un autre processus.

Nous allons rappeler rapidement les problèmes inhérents à cette approche et la suite du cours va présenter des méthodes alternatives d’implémentation de la concurrence.

À quoi s'expose-t-on lorsque l'on utilise des verrous ?

- Blocage
- Coût de l'accès aux ressources partagées
- Difficulté de la programmation avec verrous
- Difficulté du passage à l'échelle des verrous
- Inversion de priorité

À quoi s'expose-t-on lorsque l'on utilise des verrous ?

- Blocage :
  Lorsqu'un processus est bloqué, il ne fait rien d'autre qu'attendre. Cela induit **une certaine forme de fuite de mémoire** puisque le descripteur du processus est présent en mémoire sans avoir la garantie que l'on pourra le désallouer à un moment donné. Les pratiques modernes utilisent des verrous non bloquants ou bloquants pendant un temps borné : il faut alors rajouter dans la logique du programme, **le traitement de l'échec de la prise d'un verrou**.
- Coût de l'accès aux ressources partagées
- Difficulté de la programmation avec verrous
- Difficulté du passage à l'échelle des verrous
- Inversion de priorité

À quoi s'expose-t-on lorsque l'on utilise des verrous ?

- Blocage
- Coût de l'accès aux ressources partagées :
  Dans ce style de programmation, on place généralement un verrou sur toute donnée potentiellement partagée, même si la plupart du temps, seul un processus y accède à la fois. Cela augmente donc globalement le coût d'accès à cette ressource.
- Difficulté de la programmation avec verrous
- Difficulté du passage à l'échelle des verrous
- Inversion de priorité

À quoi s'expose-t-on lorsque l'on utilise des verrous ?

- Blocage
- Coût de l'accès aux ressources partagées
- Difficulté de la programmation avec verrous :
  Programmer avec des verrous est très difficile car il faut prendre en compte l'ensemble de tous les entrelacements possibles de toutes les évaluations possibles des processus. Il faut aussi prendre en compte la terminaison de tous les processus : en effet, si un processus pose un verrou et part dans une boucle infinie, tous les processus bloqués sur ce verrou seront interrompus. Il y a des cas d'**interblocages** encore plus subtils : même si deux processus pris indépendemment semblent ne pas poser de problème de terminaison, leur interaction peut les bloquer tous les deux. Un exemple simple : « P1 prend M1 puis M2, P2 prend M2 puis M1 ». Si les deux processus prennent en même temps leur premier verrou respectif, ils sont ensuite interbloqués. Par ailleurs, les erreurs de programmation des verrous sont extrêmement difficiles à reproduire.
- Difficulté du passage à l'échelle des verrous
- Inversion de priorité

À quoi s'expose-t-on lorsque l'on utilise des verrous ?

- Blocage
- Coût de l'accès aux ressources partagées
- Difficulté de la programmation avec verrous
- Difficulté du passage à l'échelle des verrous :
  Un verrou peut être vu comme une variable globale sur laquelle les opérations sont atomiques. À ce titre, **ils posent les mêmes problèmes de modularité**. En sus, les verrous ajoutent un nouveau type d'interdépendance : deux processus sont dépendants à travers un verrou si leurs évaluations respectives dépendent de ce verrou.
- Inversion de priorité

À quoi s'expose-t-on lorsque l'on utilise des verrous ?

- Blocage
- Coût de l'accès aux ressources partagées
- Difficulté de la programmation avec verrous
- Difficulté du passage à l'échelle des verrous
- Inversion de priorité :
  Il est naturel d'ordonné les processus à l'aide d'un mécanisme de priorités (comme c'est le cas par exemple dans les systèmes d'exploitation). Si un processus avec une faible priorité prend un verrou réclamé par des processus de priorité haute alors on assiste à une **inversion de la priorité** puisque le processus de priorité normalement faible pourra prendre possession de ressources auxquelles il n'aurait pas eu accès si les processus de priorités fortes avaient été actifs.

Communication sans verrous

Une première méthode pour éviter les problèmes cités dans le transparent précédent consiste à ne pas utiliser de verrous du tout. En effet, il existe des algorithmes et des structures de données concurrents sans verrous.

Algorithme de Peterson

L'algorithme de Peterson permet de réaliser un mécanisme d'exclusion mutuelle en utilisant uniquement la mémoire partagée :

```
type α critical = (unit → α) → α

let xpar (p1 : α critical → β) (p2 : γ critical → 'd) =
  let turn = ref 0 in
  let m = Array.create 2 0 in
  let critical i f =
    while true do
      m.(i) ← 1;
      turn := (i + 1) mod 2;
      while (m.((i + 1) mod 2) ≡ 1 ∧ !turn ≡ (i + 1) mod 2) do () done;
      f ()
    done
  in
  let pid1 = Thread.create p1 (critical 0) in
  let _ = Thread.create p2 (critical 1) in
  Thread.join pid1
```

Quel défaut a cet algorithme ?

Structure de données sans verrou

Il est souvent plus facile de concevoir des **structures de données concurrentes sans verrous** que des algorithmes concurrents sans verrous. En effet, on peut utiliser deux idées dans la conception :

- La structure de données propose **des opérations aux garanties tellement faibles** que quelque soit l'entrelacement de ces opérations, les hypothèses internes et externes sur la structure de données restent valides. C'est un cas idéal qui arrive en pratique ! (Voir quelques transparents plus loin.)
- On peut **réutiliser les mécanismes servant à implémenter les verrous** au sein de l'implémentation des opérations de la structure de données.
  Mais au fait, savez-vous comment sont implémentés les verrous ?

Implémentation des verrous

Si on considère que la fonction suivante est atomique :

```
int compare_and_swap (int* reg, int oldval, int newval) {
   int old_reg_val = *reg;
   if (old_reg_val == oldval)
       *reg = newval;
   return old_reg_val;
}
```

alors il suffit d'utiliser une variable globale `mutex` valant 1 si le verrou est posé et 0 dans le cas contraire. On peut alors écrire :

```
int lock () {
    if (mutex == 0)
       return (compare_and_swap (&mutex, 0, 1) == 0);
    return 0;
}
```

Structure de données concurrentes sans verrous

Voici une pile concurrente sans verrous :

```
#define empty (NULL)

typedef struct node { int x; struct node* next; } node_t;

node_t* new_node (int x, node_t* next) {
  node_t* node = (node_t*) (malloc (sizeof (node)));
  node→x = x;
  node→next = next;
  return (node);
}

node_t* stack;

void push (int x) {
  node_t* cell = new_node (x, empty);
  do {
    cell→next = stack;
  } while (compare_and_swap (&stack, cell→next, cell) != cell→next);
}
```

Comment implémenteriez-vous `pop` ?

Structure de données concurrentes sans verrous

```
int pop (int& x) {
  node_t* cell = stack;
  while (cell != empty) {
    if (compare_and_swap (&stack, cell, cell→next) == cell) {
      x = cell→x;
      return 1;
    }
    cell = stack;
  }
  return 0;
```

Cette implémentation est correcte mais sous certaines conditions . . .

Structure de données concurrentes sans verrous

Considérons les deux processus suivants :

```
P1:
/* t = 0 */ cell = stack;
/* t = 5 */
if (compare_and_swap (stack, cell, cell→next)
      == cell) {
    // Problem !
}
```

```
P2:
/* t = 0 */ cell = stack;
/* t = 1 */
if (compare_and_swap (stack, cell, cell→next)
      == cell) {
/* t = 2 */ z = cell→x;
/* t = 3 */ push (21);
/* t = 4 */ push (42);
```

Si le second appel à `push` dans P2 réutilise la même adresse que celle sur laquelle P1 s'appuie pour effectuer son test alors la cellule correspondant au `push (21)` sera perdue !

Structure de données concurrentes sans verrous

Si les structures de données sans verrous sont séduisantes, elles restent néanmoins subtiles à implémenter. Cependant, des solutions valides ont été proposées ces dernières années pour implémenter des files, des tables de hachages, des listes, etc. Pour plus d'information, google `lock-free data structures`.

Un cas plus facile

Dans certains cas, on peut se passer totalement de primitives atomiques ! Un exemple typique de communication inter-processus admettant une spécification suffisamment faible pour s'abstraire totalement des entrelacements survient lorsque la perte aléatoire [3] d'un message n'influe pas sur la correction finale de l'algorithme.

3. Il est important cependant que la probabilité d'une perte de message suive une loi uniforme vis-à-vis des processus.

Exemple : Les algorithmes de colonie de fourmis

Dans les années 90, des algorithmes d'optimisation ont été inventés en s'inspirant du comportement des colonies de fourmis. On a remarqué qu'une colonie de fourmis peut calculer relativement efficacement un plus court chemin de la fourmilière à une source de nourriture de la façon suivante :

1. Plusieurs fourmis partent de la fourmilière en suivant une marche aléatoire.
2. En marchant, une fourmi laisse des phéromones sur son chemin. Ses phéromones disparaissent avec le temps.
3. Quand une fourmi a trouvé une source de nourriture, elle suit le chemin inverse pour rentrer à la fourmilière.
4. Pour les itérations suivantes, les fourmis continuent à suivre une marche aléatoire pondérée par la concentration de phéromones : plus la concentration de phéromones est importante et plus une fourmi aura tendance à le suivre.

Plus un chemin est court et plus les fourmis repasseront dessus en déposant des phéromones. Au bout d'un moment, les chemins les plus courts seront donc ceux avec la concentration de phéromones la plus importante. Ce résultat n'est pas mis en danger si, de temps en temps, une fourmi n'arrive pas à déposer de phéromones en un point de son chemin.

La programmation concurrente structurée

Encore une fois, de façon à traiter la **complexité** de la programmation concurrente, on peut restreindre les mécanismes calculatoires accessibles au programmeur de façon à obtenir plus de garantie et à faciliter le raisonnement.

Une première méthode consiste à un introduire un **notion explicite de temps** dans les programmes concurrents. Une sorte d'horloge, plus ou moins globale, sert alors de point de repère aux processus pour établir un protocole de communication uniquement possible d'un instant à l'instant suivant. Les processus se synchronisent avec l'horloge : on dit qu'ils sont synchrones.

D'autre méthode abandonne la structuration à l'aide d'une horloge en faveur de protocoles plus souples, asynchrones tout en garantissant l'absence d'interblocage ou de famine.

Plan

Le modèle synchrone

Le modèle synchrone représente un programme concurrent comme un **système réactif**, c'est-à-dire un système qui maintent une relation continue entre les données et les sorties.

Dans ce modèle, à chaque instant est associé un ensemble d'événements. L'hypothèse de la programmation synchrone s'énonce ainsi :
"L'ensemble des calculs nécessaires pour déterminer les événements de l'instant $t+1$ à partir des événements de l'instant t prennent un temps plus petit que 1."

En d'autres termes, il est possible de déterminer un **temps de réaction** correspondant au temps maximal des calculs nécessaires au traitement d'une entrée avant qu'une nouvelle entrée ne survienne.

Exercice

Donner des exemples de programme vérifiant ces hypothèses.

Exemples de situations “synchrones”

▷ Dans une interface homme-machine, on suppose que l'on peut prendre en compte un événement de l'utilisateur avant que celui-ci n'en provoque un nouveau.

▷ Un système est dit “temps réel” quand il fournit des garanties temporelles sur son temps de réponse.

▷ Dans un protocole de communication, il est courant de borner, même de façon arbitraire, le temps alloué à l'aller-retour entre les deux participants.

Temps discret

La plupart des langages dit synchrones (Lustre, Esterel, Signal) font une hypothèse de temps discret : un programme peut alors être vu comme un transformateur de flot de données.

Dans ce cadre, une variable x est une suite $(x_n)_{n\in\mathbb{N}}$ de valeurs.

Un programme est défini alors par des suites de sorties exprimées en fonction de suites d'entrées ayant la contrainte essentielle que le terme d'une suite (x_n) au rang k ne peut dépendre que des valeurs **passées** des suites du programme à travers un calcul en mémoire **bornée**.

Le langage Lustre

Le langage Lustre est langage synchrone très simple mais très expressif dans lequel toute variable ou expression représente une suite de valeurs. Ainsi, si on écrit `x`, on parle de la suite des valeurs que prendra `x`. Si on écrit `42`, on parle de la suite constante valant uniquement `42`. On peut appliquer des opérations entre les suites en travaillant point à point.

Ainsi, `x + y` représente la suite $(x_0 + y_0, x_1 + y_1, \ldots)$.

De même, `if b then x else y` représente la suite valant x_i lorsque c_i vaut `true` et y_i dans le cas contraire.

Deux constructeurs de suite servent à introduire une dépendance temporelle bornée :

- `pre x` représente $(\emptyset, x_1, x_2, \ldots)$
- `x` $\rightarrow$ `y` représente $(x_0, y_1, y_2 \ldots)$

Les entiers naturels peuvent être définis ainsi :

```
x = 0 -> pre (x) + 1
```

Horloges

Lustre intègre une notion d'horloge permettant de définir des rythmes distincts pour différentes suites. Ainsi, si on écrit :

```
xt = x when b
```

alors les termes de la suite `xt` ne sont réalisés que lorsque `b` vaut `true`.

On ne peut alors pas écrire `x + xt` puisque ces deux suites ne sont pas définies au même rythme. Une sorte d'opérateur de "cast" d'horloge est alors applicable à `xt` pour construire une suite qui a une horloge sufisamment rapide pour être composé avec `x`. On l'écrit `current (xt)`, c'est la suite qui vaut la "dernière valeur" de `xt` tant que celle-ci n'a pas été renouvelée.

Raisonnement en temps discret

Dans ce cadre restreint – mais pourtant suffisamment expressif pour être utilisé pour implémenter certains contrôleurs dans l'avionique par exemple – il est possible de modéliser les suites à l'aide de logique temporelle, d'automates finis, etc. Cela signifie que l'on peut certifier le bon comportement de ces systèmes par des méthodes de *model-checking*.

Parallélisme compilé

Un autre avantage du modèle synchrone c'est qu'il se compile en code séquentiel. En effet, un programme Lustre peut être compilé en un automate fini dont la fonction de transition est efficacement représenté par du code C effectuant seulement des tests, des affectations et éventuellement des itérations bornées.

Du temps discret au temps continu

Si on sait borner le temps de réaction des calculs alors le modèle synchrone en temps discret est un choix à considérer car il simplifie drastiquement la complexité du système.

Dans le cas contraire, il est encore possible de rester dans un modèle synchrone, fournissant moins de garanties mais susceptible d'apporter une expressivité intéressante pour la modélisation d'un problème ayant une composante temporelle importante.

Dans l'approche synchrone en temps continu, on voit un programme non pas comme une suite indicée par des indices correspondant à des instants, mais directement comme une fonction dépendant du temps. Dans ce cadre, un calcul de type α est représenté par une fonction de type `time` $\rightarrow \alpha$, appelée un **signal**.

Programmation fonctionnelle réactive

La **programmation fonctionnelle réactive** s'appuie sur des combinateurs travaillant sur des signaux pour construire des programmes réactifs. Il existe des langages de programmation dédiés à ce paradigme, comme par exemple Reactive ML. Il existe aussi des bibliothèques en O'Caml ou en Haskell implémentant ces abstractions.

Un programme écrit avec la bibliothèque O'Caml React

```
let pr_time t =
  let tm = Unix.localtime t in
  Printf.printf "\x1B[8D%02d: %02d: %02d%!"
    tm.Unix.tm_hour tm.Unix.tm_min tm.Unix.tm_sec

open React;;

let seconds, run =
  (* Create a signal of values of type unit. *)
  let e, send = E.create () in
  (* Here is the generator of that signal: an infinite loop that sends the time *)
  (* at one time per second. *)
  let run () = while true do send (Unix.gettimeofday ()); Unix.sleep 1 done in
  e, run

(* Printer is also a signal of type unit t. *)
let printer = E.map pr_time seconds

let () = run ()
```

Application

Exercice

Modélisez puis implémentez un (mini)-tableur à l'aide de la bibliothèque React.

Avantages

La programmation réactive fournit un cadre conceptuel très clair et très expressif *via* l'abstraction du **signal**. En effet, utiliser le temps comme mécanisme de synchronisation entre les processus facilite la "prise de rendez-vous".

Inconvénients

Implémenter une horloge “globale” n’est pas forcément simple : par exemple, en temps discret, quand on fait l’union de deux signaux qui ont été définis à partir de deux horloges différentes, laquelle choisir ? En temps continu, le calcul automatique des dépendances entre les signaux peut échouer à cause d’un cycle quand on fait la liaison entre de deux signaux potentiellement différents.

Plan

Faire “sans le temps”

Introduire une notion d’horloge globale n’est souvent pas réaliste car, pour certains calculs, évaluer une borne du temps de réaction est trop difficile ou bien une sur-approximation inutilisable.

Si on abandonne la notion de temps, les processus s’exécutent moralement en parallèle à un rythme inconnu. Quels modèles de communication peut-on alors établir tout en minimisant l’utilisation concurrente de la mémoire partagée ?

Pas de communication : Parallélisme de données et de tâches

Parallélisme de tâches

Le parallélisme de tâches est obtenu en décomposant un programme en tâches indépendantes qui ne communiquent qu'à travers une transmission de leurs entrées et leurs sorties respectives uniquement (il n'y a pas d'interaction durant l'exécution de la tâche).
On a alors une architecture de type MISD (**M**ultiple **I**nstructions **S**imple **D**ata).

Parallélisme de données

Le parallélisme de données consiste à appliquer un même programme sur des données différentes.
On a alors une architecture de type SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata).

On peut bien sûr combiner ces deux types de parallélisme. (Architecture MIMD).
Encore une fois, le point important est la **décomposition** du problème dans l'espace (les données) et en temps (les tâches). Ces choix sont dépendants du problème mais aussi du matériel utilisé !

Parallélisme de tâches : solution séquentielle

```
def inside (rg: Random) : BigInt = {
  val x = rg.nextDouble
  val y = rg.nextDouble
  return (if (x * x + y * y ≤ 1) 1 else 0)
}

var chunk : BigInt = 10000

def compute (N: BigInt) : BigInt =
  if (N ≤ chunk) {
    var x : BigInt = 0;
    val rg = new Random
    var i = N
    while (i > 0) { x += inside (rg); i -= 1 }
    x
  }
  else
    compute (N / 2) + compute (N / 2 + N % 2))
```

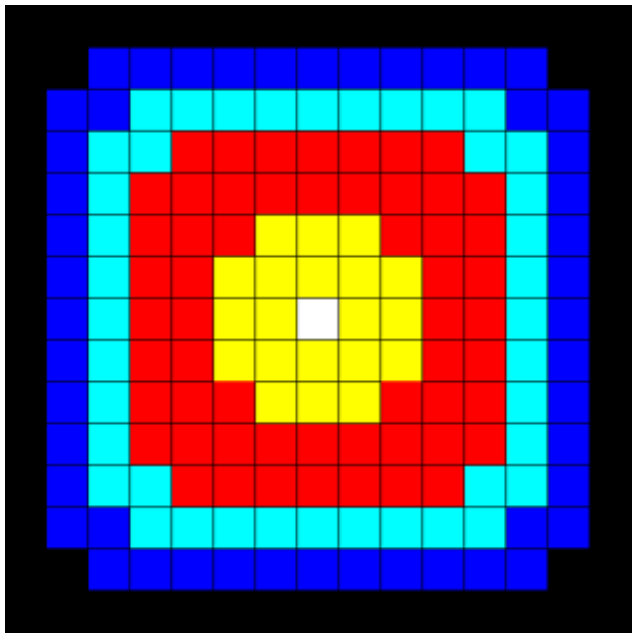
Parallélisme de tâches : Solution multi-cœurs

```
def inside (rg: Random) : BigInt = {
  val x = rg.nextDouble
  val y = rg.nextDouble
  return (if (x * x + y * y ≤ 1) 1 else 0)
}

def parplus (x : => BigInt, y: => BigInt) = {
  val xf = future { x }
  val yf = future { y }
  xf () + yf ()
}

var chunk : BigInt = 10000

def compute (N: BigInt) : BigInt =
  if (N ≤ chunk) {
    var x : BigInt = 0;
    val rg = new Random
    var i = N
    while (i > 0) { x += inside (rg); i -= 1 }
    x
  }
  else
    parplus (compute (N / 2), compute (N / 2 + N % 2))
```

Parallélisme de données

Le cas du calcul de π est un cas idéal où il n'y a aucune dépendance de données entre les tâches. Dans de nombreux cas, il est nécessaire de réfléchir à une bonne décomposition des données.

Exemple 1 : Équation de la chaleur



$$
\begin{aligned}
U_{t+1}(x,y) \quad = \quad & U_t(x,y) \\
& + C_x * (U_t(x+1,y) + U_t(x-1,y) - 2 * U_t(x,y)) \\
& + C_y * (U_t(x,y+1) + U_t(x,y-1) - 2 * U_t(x,y))
\end{aligned}
$$

Exercice

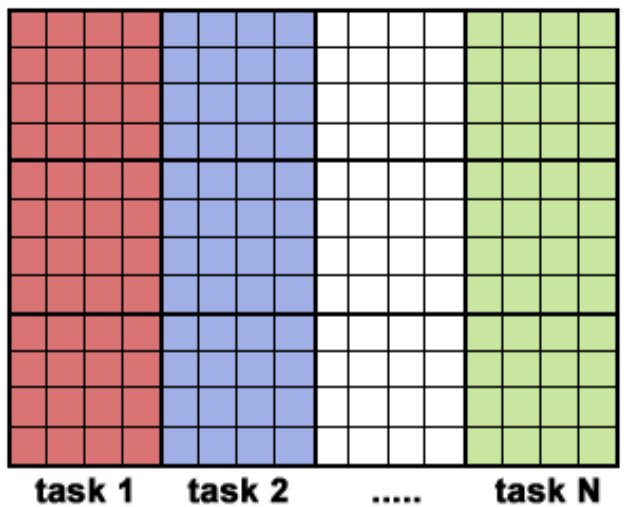
Implémenter la résolution de l'équation de la chaleur en SCALA à l'aide d'agents.

Exemple 2 : Une indépendance de surface

Soit une matrice `m`, on souhaite paralléliser le programme

```
Matrix.update (fun i j x → f i j x) m
```

Est-ce que la décomposition suivante est la plus efficace ?

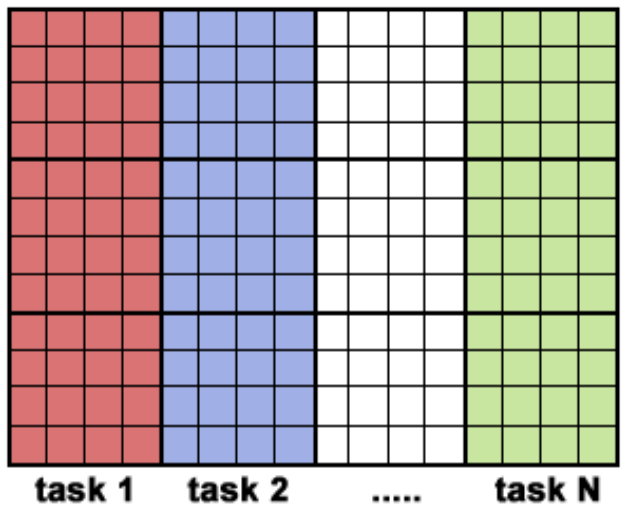


Exemple 2 : Une indépendance de surface

Soit une matrice `m`, on souhaite paralléliser le programme

```
Matrix.update (fun i j x → f i j x) m
```

Est-ce que la décomposition suivante est la plus efficace ?



Cela dépend de la représentation de la matrice en mémoire !

Mémoire transactionnelle

Une autre hypothèse de travail pour effectuer une communication à travers la mémoire partagée en évitant les problèmes des verrous consiste à implémenter des **primitives d'atomicité de plus haut niveau**.

Les mémoires transactionnelles sont des mémoires à accès concurrent qui autorisent **une séquence de plusieurs écritures et lectures** à être **atomique**.

Un exemple classique : le transfert entre deux comptes bancaires

Tiré de “Beautiful Code”, chapitre “Beautiful Concurrency”, Simon Peyton Jones, Ed. O’Reilly

Écrire une procédure permettant d’effectuer un transfert entre d’un compte bancaire et un autre. On suppose que ces comptes bancaires sont en mémoire. On suppose de plus que plusieurs processus peuvent appeler cette fonction simultanément. On souhaite qu’aucun processus ne puisse observer un état du système où l’argent est supprimé du compte de départ mais pas arrivé sur le compte d’arrivée.

Exercice

Comment implémenteriez-vous cette procédure en JAVA ?

En Java...

```
class Account {
  Int balance;
  synchronized void withdraw (int n) {
    balance = balance - n;
  }
  void deposit (int n) {
    withdraw (-n);
  }
}

void transfert (Account from, Account to, Int amount) {
  from.withdraw (amount);
  to.deposit (amount);
}
```

Exercice

Quelles sont les erreurs de ce programme ?

Une autre version en JAVA

```
void transfert (Account from, Account to, Int amount) {
   from.lock () ; to.lock () ;
   from.withdraw (amount) ;
   to.deposit (amount) ;
   from.unlock () ; to.unlock () ;
}
```

Exercice

Quelles sont les erreurs de ce programme ?

Une nouvelle version en JAVA

```
void transfert (Account from, Account to, Int amount) {
  if (from < to) {
     from.lock (); to.lock ();
  } else {
    to.lock (); from.lock ();
  }
  from.withdraw (amount);
  to.deposit (amount);
  from.unlock (); to.unlock ();
}
```

Exercice

Quelles sont les erreurs de ce programme ?

Mémoire transactionnelle

En Haskell, on écrit :

```
transfer :: Account → Account → Int → IO ()
transfer from to amount
= atomically (do {
        deposit to amount ;
        withdraw from amout
   })
```

La fonction `atomically` prend une action `a` de la monade `STM a` en argument et l'exécute atomiquement en garantissant :

- L'**atomicité** : Les effets de l'action `a` sont visibles des autres processus comme une action unique inséparable.
- L'**isolation** : Pendant son évaluation, l'action `a` n'est pas affectée par l'évaluation des autres processus, et réciproquement.

Concurrence optimiste

Comment peut-on implémentée une mémoire transactionnelle ?

Une façon relativement simple d'implémenter une mémoire transactionnelle s'appuie sur un *enregistrement des opérations d'écriture, de lecture et de leurs résultats* dans la mémoire. Ces opérations ne travaillent pas directement dans la mémoire partagée mais dans un fragment local de la mémoire : quand on lit une variable pendant la transaction, on vérifie d'abord qu'il n'en existe pas une version dans ce fragment local de mémoire avant d'aller lire dans la mémoire partagée.

Au moment où la transaction est terminée, deux opérations sont effectuées de façon atomique :

1. On vérifie que la valeur des variables qui ont été lues en mémoire partagée n'ont pas changées vis-à-vis de l'enregistrement.
2. Si l'étape 1 a réussi alors on effectue réellement les opérations d'écriture en mémoire partagée. Sinon, l'ensemble de la transaction échoue et on la reprend au début, ce qui ne pose pas de problème grâce à l'isolation.

Ce style de programmation concurrente suppose de façon optimiste que la fréquence des accès réellement concurrents à une ressource est faible.

Exercice

Exercice

Réimplémentez le calcul de l'équation de la chaleur à l'aide de la bibliothèque `CCSTM` de Scala.

Concurrence par passage de messages

Vous utilisez un protocole de communication asynchrone régulièrement : le courrier. L'idée du courrier est par essence asynchrone puisque l'action d'émission d'un message est indépendante dans le temps de sa réception. Par ailleurs, le transfert d'un message s'opère d'une boîte aux lettres à une autre sans qu'à aucun moment l'accès au message soit disputé par deux processus simultanément. Les problèmes liés aux verrous disparaissent ainsi.

Nous allons voir deux abstractions proposées par les langages de programmation pour implémenter la concurrence par passage de messages : les agents et le join-calcul.

Les Agents (de SCALA ou d'ERLANG)

Le principe des agents consiste à définir des composants logiciels possédant un **état**, **réagissant à la réception de messages** et **capable d'en envoyer**.

On peut implémenter les agents par des co-routines, des fils d'exécutions ou des processus contenant une fonction implémentant **l'action** associée à l'agent. Comme toute co-routine, un agent peut potentiellement ne jamais rendre la main à son créateur.

En SCALA, la définition d'un agent (appelé acteur dans le vocabulaire de SCALA), est en général de la forme suivante :

```
import scala.actors.Actor
class SomeActor extends Actor {
   def act () {
      ... // Some action
   }
}
```

On rend actif un acteur en exécutant sa méthode `start`.

Les acteurs de SCALA

La bibliothèque d'acteur fournit une fonction de création d'acteur à partir du code décrivant son action. Ainsi, le programme suivant est équivalent au transparent précédent :

```
val some_actor = actor {
  // Some action
}
```

Un acteur représentant une pile concurrente

```
sealed abstract class StackMessageIn[T]
case class Push[T] (x: T) extends StackMessageIn[T]
case class Pop[T] extends StackMessageIn[T]
sealed abstract class StackMessageOut[T]
case class Top[T] (x: T) extends StackMessageOut[T]

class StackActor[T] extends Actor {
  private var values : List[T] = Nil
  def act () {
    loop {
      react {
        case Push (x: T) => values = x :: values
        case Pop => {
          sender ! Top (values.head) // Respond the top to the sender.
          values = values.tail
        }
      }
    }
  }
}
```

Boîte aux lettres Tiré de http://programming-scala.labs.oreilly.com/ch09.html

```
val countActor = actor {
  loop {
    react {
      case "how many ?" => {
        println("I've got " + mailboxSize.toString + " messages in my mailbox.")
      }
    }
  }
}

countActor ! 1
countActor ! 2
countActor ! 3
countActor ! "how many ?"
countActor ! "how many ?"
countActor ! 4
countActor ! "how many ?"
```

Affiche :

```
I've got 3 messages in my mailbox.
I've got 3 messages in my mailbox.
I've got 4 messages in my mailbox.
```