Langages de Programmation Concurrente, Parallèle, Distribuée.

Denis Caromel

Université de Nice Sophia Antipolis

INRIA Sophia Antipolis

Projet *OASIS*

DEA Réseaux et Systèmes Distribués

Filière Systèmes Distribués

DEA Informatique

Ecole Doctorale STIC,

Université de Nice-Sophia Antipolis

en collaboration avec

CMA France Telecom R&D EURECOM INRIA

---

www.inria.fr/oasis/Denis.Caromel/

caromel@unice.fr

Version 06

Préambule

Ce support de cours/transparents correspond au module intitulé Langages de Programmation Concurrente, Parallèle et Distribuée, enseigné au DEA Réseaux et Systèmes Distribués, Filière Systèmes Distribués, et au DEA Informatique.

L'objectif de ce cours est de présenter les langages parallèles asynchrones (par opposition aux langages dits «synchrones» tel Esterel par exemple), d'en dégager quelques concepts généraux (modèle asynchrone, communication synchrone ou asynchrone, points d'entrée ou procédure, etc.), et de mettre en pratique leurs utilisations sur des exemples concrets. On s'intéressera tout aussi bien à un parallélisme de situation (concurrence et distribution), qu'à un parallélisme de résolution.

Une partie importante du cours (au moins 60%) est consacrée aux langages et techniques à objets.

Plan sommaire du cours:

- Introduction - Cadre - Objectifs: (3 Heures)
  - Présentation du modèle asynchrone (par rapport a l'hypothèse synchrone).
  - Langages de haut niveau d'abstraction permettant à l'utilisateur de s'abstraire du placement, des communications, etc.
  - Objectifs: facilité et puissance d'expression, réutilisation, ...
  - Quelques concepts de bases.
- Rappels et approfondissement sur les langages classiques (3 Heures)
  - CSP, Ada
  - etc.

- Langages à objets parallèles: (6 Heures)
  - Principes, utilisation de l'héritage et du polymorphisme.
  - Réutilisation et transformation de code séquentiel.
  - Abstractions pour la programmation du contrôle des processus (expressions de chemins, compteurs de synchronisation, conditions d'activation, ensembles d'activation, ...)
  - Exemples de langages (langages d'Acteurs, extensions de C++, et surtout Java).
  - Migration d'objets actifs
  - Grid computing (calculs sur réseaux non-locaux) , déployement
- Séance sur machines (3 Heures)
- Outils et techniques complémentaires: (3 Heures)
  - Programmation parallèle de type fonctionnel.
  - Jini (Découverte dynamique de services)
  - Approches par bibliothèques (MPI, OpenMP, etc.)
- Autres approches de type asynchrone (3 Heures)
  - Programmation parallèle logique
  - Programmation parallèle par contraintes

Intervenants : F. Baude, D. Caromel, M. Rueher

Table des Matières

CHAPITRE 1 INTRODUCTION

1.1 Cadre et généralités

1.1.1 Définitions

Concurrence:

Accès, ou tentative d'accès, simultané à une ressource physique ou logique

Exemples:

zone ou mot mémoire, cache, secteur disque, écran, imprimante, etc.

Parallélisme:

Exécution, ou réalisation simultanée de plusieurs activités, ou processus

Exemples:

2 multiplications en même temps sur 2 processeurs différents, ou par 2 parties distinctes d'un processeur. Impression d'un fichier sur 2 imprimantes en même temps, stockage sur plusieurs disques (RAID: Redundant Arrays of Inexpensive Disks)

Distribution (en fait Répartition):

Plusieurs espaces d'adressage,

Exemples:

2 PCs LAN, cluster de PC, machines en WAN,

Cadre et généralités

Les 3 notions sont souvent mélangées,

à juste titre ou simplement pas confusion :

Parallélisme peut ==> Répartition

parallélisme de résolution,

scalability ... donc Message,

donc répartition

Répartition peut ==> Parallélisme

Application collaborative avec utilisateurs

répartis sur plusieurs sites

parallélisme de situation

Parallélisme peut ==> concurrence

2 fichiers en // sur une imprimante

Il y a aussi une dualité entre partage et duplication:

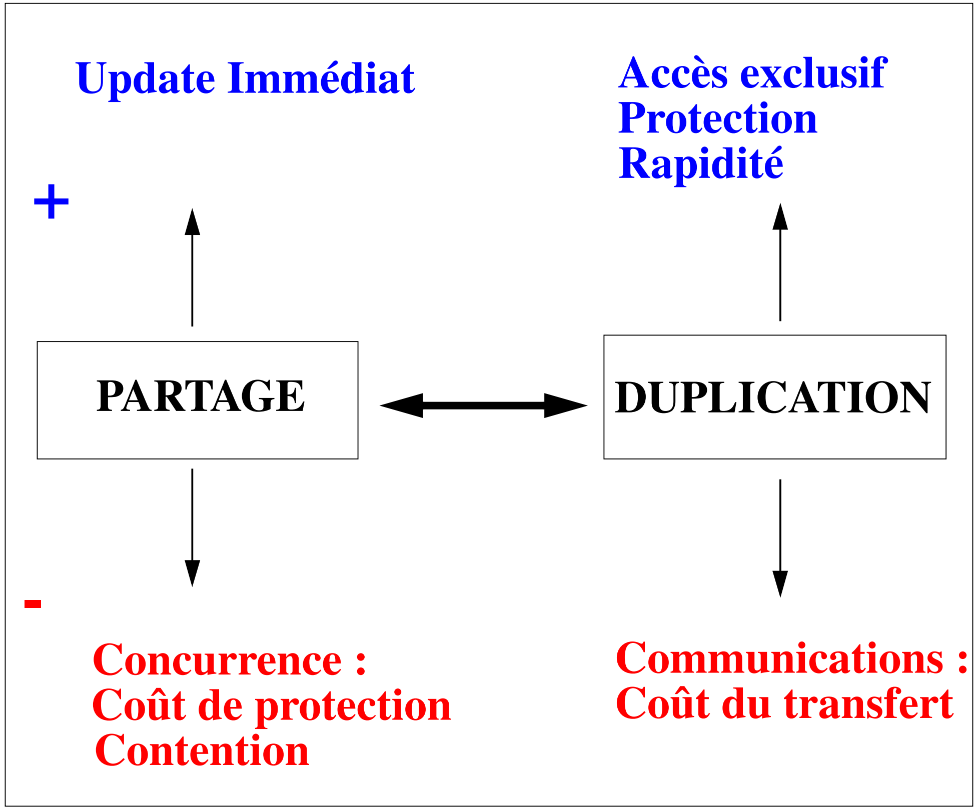


FIGURE 1 Machine séquentielle classique

1.1.2 Modèle Asynchrone

Cadre distribué:

Langages Asynchrone:

Pas d'horloge globale

"Loosely coupled":
processus faiblements couplés,
chacun progresse à son propre rythme

Communication: aucune hypothèse
temps (durée) de communication non défini, non borné, à priori non nul

Souvent des files de messages

--> • Pas d'ordre total sur les événements d'un système (ordre partiel uniquement)

Exemples:

CSP, Occam, Ada, Modula 3, PVM, MPI,

Electre, etc.

Acteurs: Actors, ABCL/x, ProActive, C++//, etc.

Types d'applications:

Applications distribuées, parallèles, etc.

Par opposition:

Hypothèse synchrone (langages synchrones):

- Un programme réagit "instantanément" à ses inputs, en un temps considéré nul.
- Communication: "Instantanée"
temps (durée) de communication nul (borné par un temps que l'on peut considéré nul).
"Instantaneous broadcasting": un processus reçoit un message (un événement) à la date où il est envoyé.
- Réaction “dans le même instant”, y compris à l’absence d’événement.
- "Tightly coupled":
processus fortement couplés.
- Calculs dirigés par les événements (input drivens).
- Horloge globale

--> • Ordre total sur les événements d'un système, simultanéité

Exemples:

Esterel, Lustre, Signal, StateCharts,

Types d'applications:

Applications Réactives, Déterministes

Problème de passage à l’échelle.

Une tentative de solution: les GALS

Globally Asynchronous Locally Synchronous

Ne pas confondre:

Modèle Synchrone / Asynchrone

et

Communication Synchrone / Asynchrone

Dans le modèle asynchrone, on trouve des langages à communication:

synchrone

ou

asynchrone

Un des challenges:

- Faire des modèles asynchrones déterministes

- Distribuer des modèles synchrones

1.1.3 Types d'applications

La programmation parallèle pourquoi faire?

Parallélisme de situation:

Le domaine à modéliser est intrinsèquement parallèle (Concurrence)

(Problem Domain parallelism)

Parallélisme de résolution:

Le parallélisme est un choix de conception (Parallélisme).

(Solution Domain parallelism)

Visant souvent à augmenter les performances

Eventuellement, la taille des problèmes (données) que l'on peut traiter

Parfois la fiabilité

Situation:

Systèmes d'Exploitation (OS): la 1ère appli!!

synchronisation de processus concurrents, mémoire partagée, communication, interblocage, correction, etc.

Applications distribuées

Systèmes Coopératifs, Collectiels
CSCW (Computer-Supported Cooperative Work),
Tableau blanc (whiteboard or blackboard)

Systèmes réactifs

Système de commandes de procédés industriels

Client/Serveur

Transactionnels (BD: Banques, Réservations, etc.)

Résolution

Applications numériques (surtout)

CFD (Computational Fluid Dynamics): Simulation of Any complex gas or liquid flow within engine, wings, etc.

N-body Methods (particle, molecules ...)

Image synthésis (Ray tracing, etc.)

Simulation (à événements discrets vs. numériques)

Data mining, etc.

Toutes les applications qui peuvent poser un problème de temps d'exécution, taille, ou fiabilité.

1.1.4 Contrôle et Données

Modèles de base:

Basés sur la structure des programmes

Contrôle:

Mécanisme d'exécution des instructions

Données:

Mécanisme d'acheminement des données

Parallélisme de Contrôle: control-driven

Le programme décrit explicitement le parallélisme (création, terminaison, communication, synchronisation)

Parallélisme de données: data-driven

Le parallélisme est structuré sur / par les données. (Implicite)

Parallélisme de réduction: demand-driven

Langages fonctionnels, à affectation unique.

Une instruction est sélectionnée et exécutée lorsque ses opérandes sont prêtes (calculées).

Data Flow.

1.1.5 Architectures de base

Basée sur la structure des machines, et les notions de flots de données, et de flots d'instruction (Flynn, 72)

Flot:

Séquence d'éléments traités par un processeur

SISD: Single Instruction, Single Data

Machine séquentielle classique,

Von Neumann, (schéma classique)

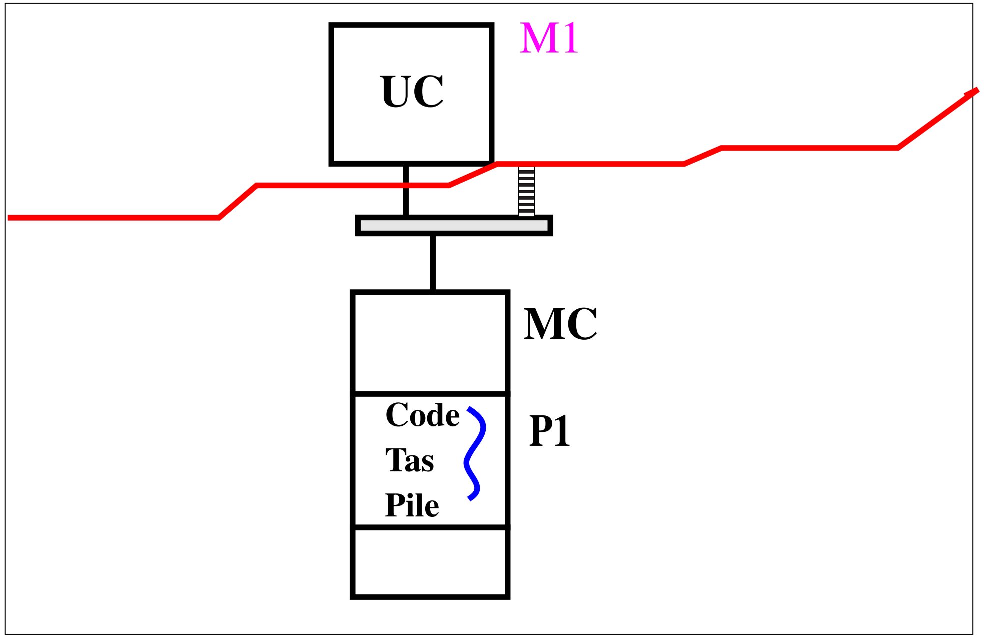


FIGURE 2 Machine séquentielle classique

SIMD: Single Instruction, Multiple Data

Type MMX (graphisme)

Vector-Array Processing:
Plusieurs éléments de calcul (processeur + mémoire) exécutant la même instruction sur des données différentes,

Supercalculateurs Vectoriels

MISD: Multiple Instructions, Single Data

Pipeline: chaque processeur exécute une instruction spécifique sur des données circulant (chaîne de montage).

MIMD: Multiple Instructions, Multiple Data

Plusieurs éléments de calcul, des données différentes.

Systèmes / Architectures distribués (répartis)

+ Multiprocesseur

Raffinement: Couplage

Couplage entre éléments de calcul / mémoire(s):

Couplage fort:

Vectoriel (SIMD)

Couplage moindre:

MIMD à mémoire partagée

Couplage faible:

MIMD sans mémoire partagée
(mémoire distribuée)

En général:

--> • Multiprocesseurs: mémoire partagée
SMP:
Symetric Multi-processing, ou
Shared Memory Parallelism

--> • Architecture répartie: mémoire distribuée
MPP:
Massively Paralel Processing, ou
Message Passing Parallelism

faiblement couplée, multimachines,“clusters”

Mais change un peu, maintenant:

souvent cluster de SMP

Architectures dites “à mémoire partagée”:

En réalité, plusieurs types existent:

UMA: Uniform Memory Access

Accès à la mémoire(s) via un bus avec temps d’accès uniforme

Cadre et généralités

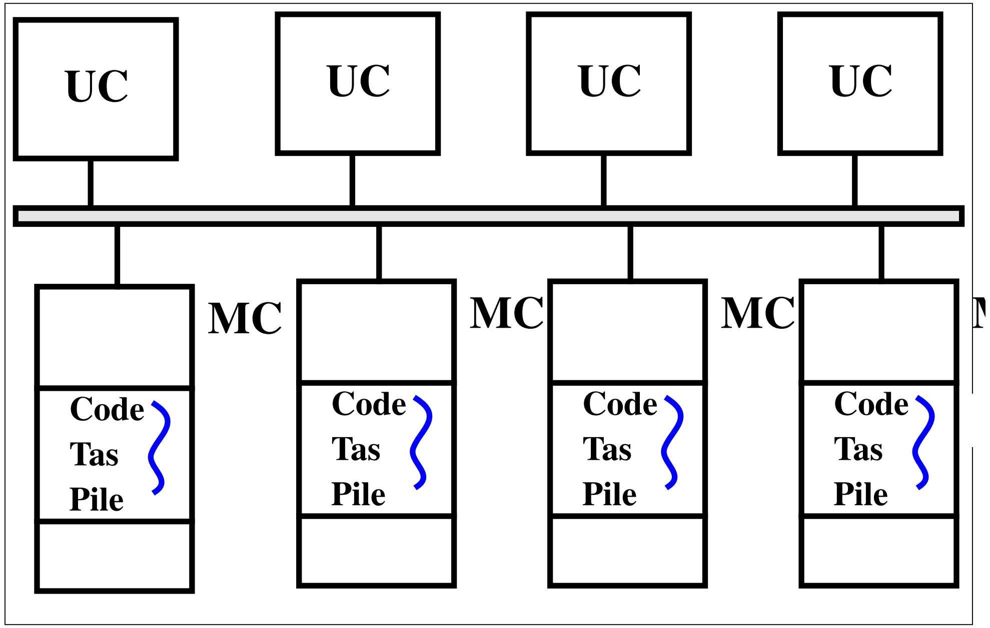


FIGURE 3 UMA: Uniform Memory Access

NUMA: Non Uniform Memory Access, ou

DSMA: Distributed Shared Memory

Mémoire souvent répartie par UC, avec temps d’accès différent selon la zone mémoire accédée et le processeur qui fait l’accès

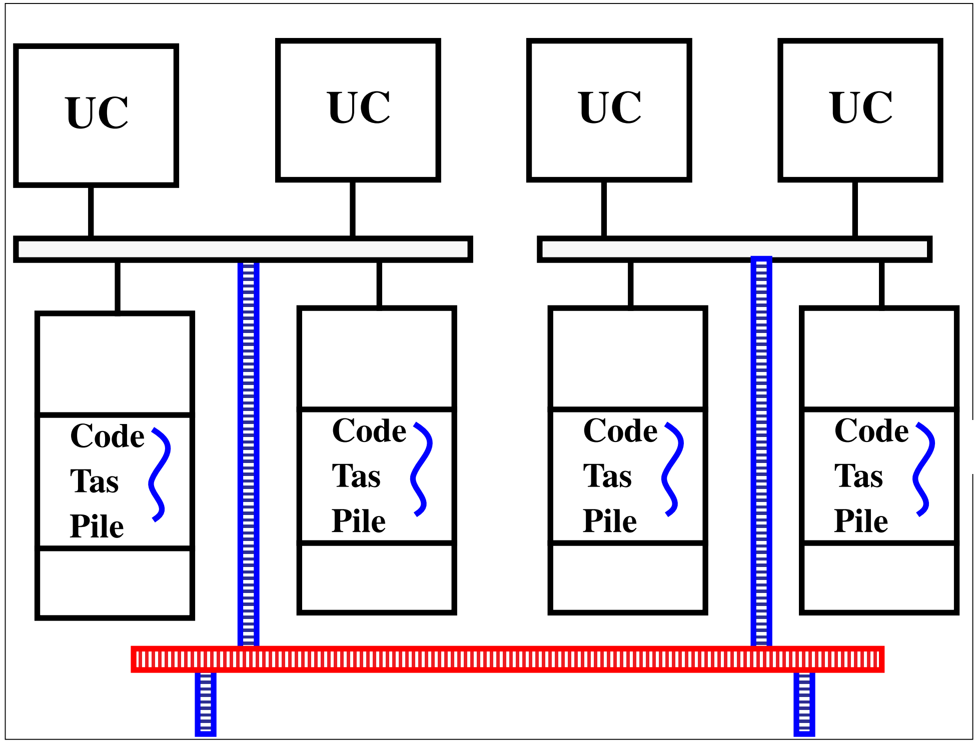


FIGURE 4 UMA: Uniform Memory Access

Le mapping devient alors très important.

COMA: Cache Only Memory Architecture

Architecture classique, mais chaque processeur n'utilise sa propre mémoire que comme un cache, + mémoire globale

Répartition dynamique des données sur le système (Hard + Soft, automatic data migration and replication capabilities of cache, page)

Proche de:

Distributed Virtual Shared Memory (DVSM) systems

Dans ce cas, on est encore “plus NUMA” que le NUMA d'origine rachitecture.

Extensions de la classification:

à la limite:

Modèle de programmation / Architecture:

qui se présente comme la classification architecture

SPMD: Single Program Multiple Data

Même programme qui s'exécute sur chaque processeur.

De type SIMD, data parallelism, mais synchronisation faible.

BSP: Bulk Synchronous Parallel Computer

De type SPMD

Barrier Synchronizations

Supersteps

two-level memory model

(+ ou - idem SPMD, mais "marketing"),

1.2 Propriétés souhaitables des langages/modèles

Une bonne abstraction des primitives de bases des architectures

- trop concret, bas niveau: obsolète rapidement
- trop haut niveau: performances difficiles à obtenir

Nécessité de compromis, en particulier par rapport à un certain nombre de propriétés.

1.2.1 Indépendant de l'architecture

Exécuter le programme sur des architectures différentes (SMP, MPP, ou cluster de SMP):

sans modifier le source

Le modèle, théoriquement, doit s'abstraire de toutes les caractéristiques spécifiques à certaines architectures.

1.2.2 Abstrait

Cacher certains aspects du parallélisme:

- Décomposition en activités parallèles (threads), surtout pour le Parallélisme (op. concurrence)

  e.g. 1000 processeurs, l'état du programme dépend de l'état des 1000 processeurs

  Impossible de l'appréhender

- Communication

  Actions d'envoi et de réception doivent être cachées

  Il existe différents niveaux:

  --> .ports, socket
  --> .RPC en C
  --> .CORBA, RMI, DCOM
  --> .ProActive (asynchrone)
  --> .

- Synchronisation

  Soit complètement cachée, soit gérée automatiquement d'une façon implicite

  Par exemple: data-driven

1.2.3 Méthode de développement

Développement structuré et guidé d'applications

Si possible en partant de "spécifications"

==> développements formels

Peut-être plus important que pour les programmes séquentiels

Post-vérifications ne marchent pas toujours, mieux: développement dans un formalisme:

WYPIWYE:

What You Prove Is What You Execute

1.2.4 Mesures de coût

Technique pour calculer le coût d'un programme, d'une conception donnée
sans que cela ne dépende des détails de l'architecture

Si composable, encore mieux

1.2.5 Granularité variable

Ne pas avoir de granularité prédéfinie de l'unité élémentaire de travail confié à une activité (processeur)

Une application:

- 2 à 20 : coarse-grain (gros grain)
- 20 à 200 : medium
- > 200 : fine-grain (grain fin)

La difficulté:

Faire du grain fin efficace !

1.3 Sémantique, spécification, et correction

Propriétés désirables des programmes parallèles:

1.3.1 Responsiveness: Vivacité

Programmes qui ne terminent jamais (e.g. OS, systèmes de réservations)

Un arrêt est une condition d'erreur

La vivacité assure que le programme pourra toujours s'exécuter.

1.3.2 Deadlock absence

Absence de situation qui ne peuvent pas évoluer, et où un ou plusieurs processus sont bloqués.

1.3.3 Livelock absence

Absence de situation qui ne peuvent pas évoluer, et où un ou plusieurs processus s'exécutent mais ne peuvent pas progresser.

1.3.4 Absence de Famine

Starvation absence, Lockout,

Un ou plusieurs processus, en général pour des raisons de priorité d'accès à une ressource, sont bloqués (indéfiniment).

Cas de Deadlock ou Livelock

1.3.5 Fairness, Equitabilité

Propriétés sur le modèle d'exécution

(et non pas un programme donné):

Lors de choix répétés entre plusieurs alternatives (e.g. accès à une ressource), une des alternatives n'est pas rejetée indéfiniment.

En théorie:

tout calcul fini est équitable.

1.3.6 Déterminisme et Non-Dét.

Aspect assez particulier:

- Quelques fois on veut du déterminisme:

  toujours le même résultat sur plusieures exécutions

- D'autres fois on souhaite du non-déterminisme

  e.g. expression abstraite de l'activité d'un buffer par exemple

Donc. l'important est de pouvoir choisir, programmer ce que l'on veux en fonction de la situation.

Voir plus loin

1.4 Différents types de langages

Langages ou paradigmes, ou API

Classification au niveau langage, et non pas architecture.

1.4.1 Shared-memory paradigm

Mémoire globale, à accès uniforme et transparent

Correspond au SMP. (e.g. OpenMP)

Le programme est responsable de la cohérence des accès multiples et simultanés à la même adresse mémoire.

Quelquefois, on utilise des techniques de type OS (sections critiques, etc.). Voir plus loin.

Modèle PRAM (Parallel Random Access Machine) pour les calculs de complexité des algos parallèles.

Approche particulière: Tuple Space

Coordination Languages (e.g. Linda)

Tuple space = shared associative memory

Les “threads” mettent et prennent dans cet espace (communication + synchronisation).

1.4.2 Mémoire distribuée

Distributed-memory

Processus qui coopèrent par échange de messages

Reflète l’architecture MPP, MIMD à mémoire distribuée. Processus communicants et coopérants (e.g. PVM, MPI, etc.)

1.4.3 Impératif

En général à mémoire distribuée, plutôt un cas particulier de ci-dessus.

Notion de processus, composé d’une liste d’instructions, en général séquentielles.

Points clés:

- Création des processus:
  - Statique (CSP)
  - Dynamique (ADA, objets, etc.)
- Coopération / Synchronisation:
  - Envoi de messages
  - Rendez-vous
  - Remote Procedure Call
  - Synchros pures, Terminaisons
  - Déterminisme/Non-Déterminisme

etc.

1.4.4 Parallel OO Programming

Notions de classes et d'objets pour la programmation parallèle.

Points clés:

- --> Utilisation de l'Héritage,
- --> Liaison dynamique,
- --> Envoi de messages

Voir plus loin.

1.4.5 Fonctionnel

Parallélisme en majorité implicite :

E.g. évaluation en parallèle des arguments d'une fonction

Points clés:

- --> Normal-order of fct. evaluation + control (Multilisp)
  or Eager
- --> Possibilité d'analyses statiques (e.g. strictness analysis)
- --> Fonctions de premier ou de second ordre (ou ordre N: e.g. Haskell)
- --> Dataflow

- --> Compromis entre parallélisme implicite et explicite

1.4.6 Logique

Evaluation en parallèle de clauses de Horn:

H <--- B1, B2, . . . Bn

Points clés:

- --> OR parallelism:
  plusieurs clauses qui toutes s'unifient avec un but donné
- --> AND parallelism:
  plusieurs sous-buts
- --> Compromis entre parallélisme implicite et explicite (annotations)

Voir plus loin.

1.4.7 Autres

Tout ce que l'on n'arrive pas à classer dans ces catégories, paradigmes mixtes, plus formel, etc.

CHAPITRE 2 Techniques de base

(Systèmes d'Exploitations)

Mémoire partagée

Origines:

Développement des Systèmes d'Exploitation multi-utilisateurs et temps-partagé

Techniques de base à connaître, mais également une perspective et un exemple d'évolution de la recherche.

2.1 Sémaphores

[Dijkstra 68]
The Structure of the THE Multiprogramming System.
In Commun. of the ACM 11, pages 341-346, May 1968.

2.1.1 Définitions

Un sémaphore est une variable (entière), et une liste de processus en attente.

Uniquement deux opérations sur un sémaphore S:

```
P(S)=if   S >= 1 then
        S := S -1;
     else
        The executing process is
        suspended and placed in
        S's waiting list.
     end;

V(S)= if   S's waiting list not empty then
        Resume one of S's waiting list
     else
        S := S+1;
     end;
```

Valeur Initiale = nombre de passages

2.1.2 Exemple: tampon partagé à N places

Sémaphore d'exclusion mutuelle.

Principes:

- *mutex* garantit l'accès exclusif à un tampon exclusion mutuel
- *libre* permet de suspendre le producteur si le tampon est plein (init = nombre de places)
- *vide* permet de suspendre le consommateur si le tampon est vide

Code:

```
semaphores:
    mutex (1) , vide (0) , libre (n)

Producteur= while ... do
                    ... produire ...
                    P(libre);
                    P(mutex);
                     ...mettre dans tampon ...
                    V(mutex);
                    V(vide);
                end;
```



```
Consommateur = while ... do
                    P(vide);
                    P(mutex);
                    ...prendre du tampon..
                    V(mutex);
                    V(libre);
                    ... consommer ...
                    end;
```

Problèmes des sémaphores:

- Très bas niveau,
- Programmes très difficiles à comprendre
- Sans même parler de vérifications formelles, il est très difficiles d'avoir l'intime conviction qu'un tel programme est correct.

2.2 Régions Critiques

Les remarques précédentes ont conduit à développer d'autres techniques: Régions Critiques proposées par Hansen 72 -73

Brinch-Hansen 73] P. Brinch-Hansen. Concurrent Programming Concepts. Computing Surveys 5(4): 224-245, 1973.

2.2.1 Définitions

Certaines variables seulement peuvent être partagées:

variable partagée Integer v;

Les variables partagées peuvent être utilisées dans des structures particulières: *régions critiques*

région v faire ... v ... fin

Les régions portant sur la même variable s'excluent dans le temps.

Toute variable partagée ne peut être accédée que depuis une région.

Toute variable non partagée (standard) ne peut être accédée que par un seul processus.

Exclusion mutuelle programmable par sémaphores

2.2.2 Régions Critiques Conditionnelles

Outre les aspects exclusion mutuelle fournis par les Régions Critiques, il est également nécessaire de disposer de synchronisations (faire attendre, faire reprendre, etc. comme cela était permi par les sémaphores) :

Régions Critiques Conditionnelles

Primitive pour attendre une condition:

```
région v faire
    ... code ...
    attendre     cond ( v ) ;
    ... code ...
fin
```

Si la condition *cond (v)* est fausse le processus en cours est interrompu

Le processus sera repris lorsque la condition sera vraie (implique la modification de la variable *v* par un autre processus et la sortie de la région critique par celui-ci (ou bloqué en attente).

2.2.3 Exemple:Producteur-Consommateur

Tampon à une seule place:

not v <==> case vide
v <==> case pleine

```
Processus producteur =
    while ... do
        .. produire ...
        region v faire
            attendre Not v;
            ...mettre dans tampon ...
            v := True;
        fin
    end;

Processus consommateur =
    while ... do
        region v faire
            attendre v;
            ... prendre du tampon ...
            v := False;
        fin
        ... consommer...
    end;
```

N places ==> v devient un entier

2.2.4 Exemple: Lecteurs - Rédacteurs

```
variable partagée
    (struct lecteur, redacteur: int ) v;
Processus lecteur =
    while ... do
        region v faire
            attendre ( v.redacteur = 0)
            v.lecteur := v.lecteur +1;
        fin
        ... lire ...
        region v faire
            v.lecteur := v.lecteur -1;
        fin;
    end;
Processus redacteur =
    while ... do
        region v faire
            v.redacteur := v.redacteur + 1;
            attendre ( v.lecteur = 0);
        fin
        region v faire ... écrire ... fin ;
        region v faire;
            v.redacteur := v.redacteur -1;
        fin;
    end;
```

Notes:

- priorité Rédacteurs

v.redacteur := v.redacteur + 1;

- 1 à la fois mais entrelacement

region v faire
v.redacteur := v.redacteur + 1;
attendre (v.lecteur = 0);
fin

- Pourquoi doit-on faire:

region v faire ... écrire ... fin ;
region v faire;

pour s'assurer qu'il n'y a qu'un seul rédacteur ...

-

2.3 Moniteurs

Evolution des Régions Critiques:

les moniteurs [Hoare 74]

Plus de variables partagées (même protégée).

Partage: objet plus abstrait

Type Abstrait de Données, (ADT)

Procédures pour manipuler l'objet partagé.

2.3.1 Définitions: procédures, conditions

Composantes d'un moniteur:

Ensemble de procédures publiques,

Procédures privées, données privées,
Code pour Initialisations,
Evt. . paramètres de création

Conditions:
variables de type particulier: permettent la synchronisation:
deux opérations prédéfinies: *Wait* et *Signal*

Sémantique:

- Toutes les procédures s'exécutent en exclusion mutuelle

- En fin d'opération (sortie du moniteur) on recherche et active un processus en attente (entrée, ou Signal)

cond.Signal;

- reprise du processus (thread) en attente (évt. le plus ancien) sur la condition cond.
- le processus en cours est suspendu

cond.Wait ;

- Suspend le processus en cours.

Exemple de moniteur:

```
moniteur m debut
    foo1 () ... c2.Signal; ...
    foo2 ( ... ) ... c1.Wait; ...
  private:
    bar1 () ...
    bar2 (...) ...
  conditions:
    c1, c2, ...
  Init
  ... initialisation des variables etc.
fin
```

Utilisation:

```
m.foo1();        m.foo2();        ....
```


2.3.2 Exemple: tampon à N places

```
moniteur tampon_borné debut

    Anytype t [0, N-1]; // Tampon
    conditions: vide, plein;

    ajouter ( x )
      if [plein] then plein.Wait end;
      ... mettre dans tampon ...
      vide.Signal;

    enlever ( x )
      if [vide] then vide.Wait end;
      ... prendre dans tampon ...
      plein.Signal;

    Init
      [Mettre à vide ]

fin
```

Conclusions sur les moniteurs:

Bas niveau :

- Sémantique de Signal assez difficile à saisir, de plus des variations existent.
- Signal est quelquefois utilisé à des fins d'optimisation.
- Les programmes réalisés avec des moniteurs deviennent très rapidement très complexes.

mais utiles:

- pas vraiment mieux quand on doit gérer explicitement de la mémoire partagée: Système d'Exploitation (multi-tâches, multi-utilisateurs, multi-processeurs, multi-threading)

Extensions quelquefois utilisées:

cond.*Queue* ;

- Retourne vrai s'il existe des processus en attente sur la condition

Nécessaire pour un lecteur-rédacteurs si l'on veut contrôler (un peu) les priorités

2.3.3 Moniteurs Java

Threads + Moniteur (objets)

Principes:

1 lock / objet

1 lock / classe

ou 1 lock par block

D'après:

Concurrent Programming in Java,
© 1996 Doug Lea, Addison-Wesley,
October 1996.

http://gee.cs.oswego.edu/dl/

Thread Control Methods:

suspend();

(itself or other, but deprecated)

temporarily halts a thread in a way that will continue normally after a (non-suspended) thread calls resume on that thread.

T.resume();

(other, but deprecated)

Resume a suspended thread.

sleep (ms);

(itself)

causes the thread to suspend for a given time (specified in milliseconds) and then automatically resume.

The thread might not continue immediately after the given time if there are other active threads.

thread.join()

suspends the caller until the target thread completes (that is, it returns when isAlive is false).

A version with a (millisecond) time argument returns control even if the thread has not completed within the specified time limit.

thread.interrupt()

causes a sleep, wait, or join to abort with an InterruptedException, which can be caught and dealt with in an application-specific way.

(deprecated)

Synchronization

Synchronization is implemented by exclusively accessing the underlying and otherwise inaccessible internal lock:

sometimes called a mutex, monitor

that is associated with each Java Object (including Class objects for statics).

Each lock acts as a counter.

If the count value is not zero on entry to a synchronized method or block because another thread holds the lock, the current thread is delayed (blocked) until the count is zero.

On entry, the count value is incremented. The count is decremented on exit from each :

synchronized method or

block,

even if it is terminated via an exception.

Waiting and Notification

The methods:

wait, notify, and notifyAll

may be invoked only when the synchronization lock is held on their targets.

This is normally ensured by using them only within methods or code blocks synchronized on their targets. Compliance cannot usually be verified at compile time.

Failure to comply results in an IllegalMonitorStateException at run time.

A wait invocation results in the following actions:

- The current thread is suspended.
- The Java run-time system places the thread in an internal and otherwise inaccessible wait queue associated with the target object:

  Dormant

- The synchronization lock for the target object is released, but all other locks held by the thread

  are retained. (In contrast, suspended threads retain all their locks.)

A notify invocation results in the following actions:

- If one exists, an arbitrarily chosen thread, say T, is removed by the Java run-time system from the internal wait queue associated with the target object (Dormant).
- T must re-obtain the synchronization lock for the target object, which will always cause it to block at least until the thread calling notify releases the lock.

  Dormant ---> blocked

  It will continue to block if some other thread obtains the lock first.
- T is then resumed at the point of its wait.

A notifyAll invocation:

works in the same way as notify except that the steps occur for all threads waiting in the wait queue for the target object.

Two alternative versions of the wait method take arguments specifying the maximum time to wait in the wait queue. If a timed wait has not resumed before its time bound, notify is invoked automatically.

If an interrupt occurs during a wait, the same notify mechanics apply except that control returns to the catch clause associated with the wait invocation.

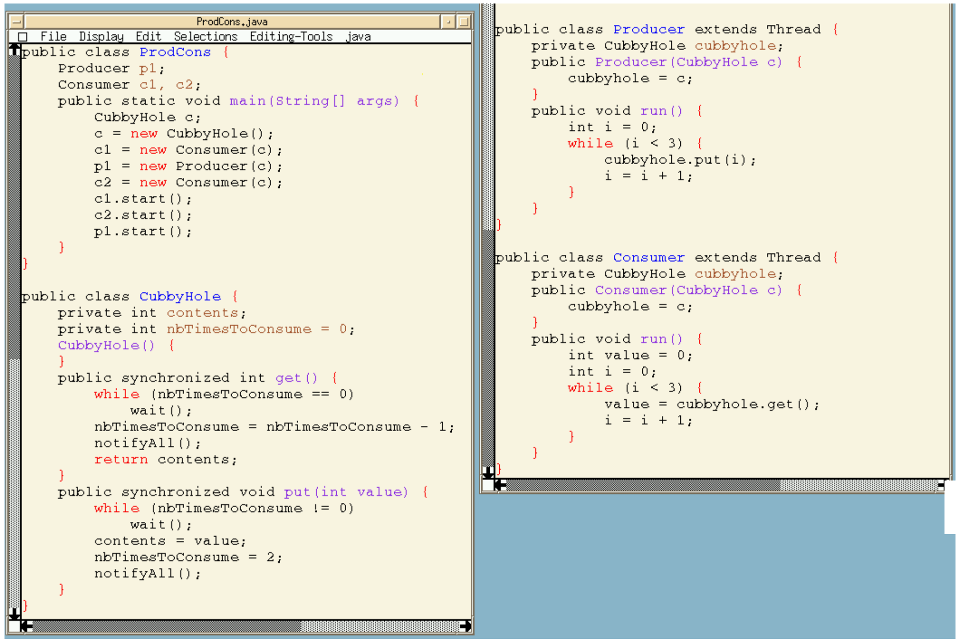
Récapitulatif:

Wait --> Dormant, perte du lock

notify --> choisi un Dormant -> blocked
le lock n'est pas perdu

Exemple:

Produser / Consumer in Java

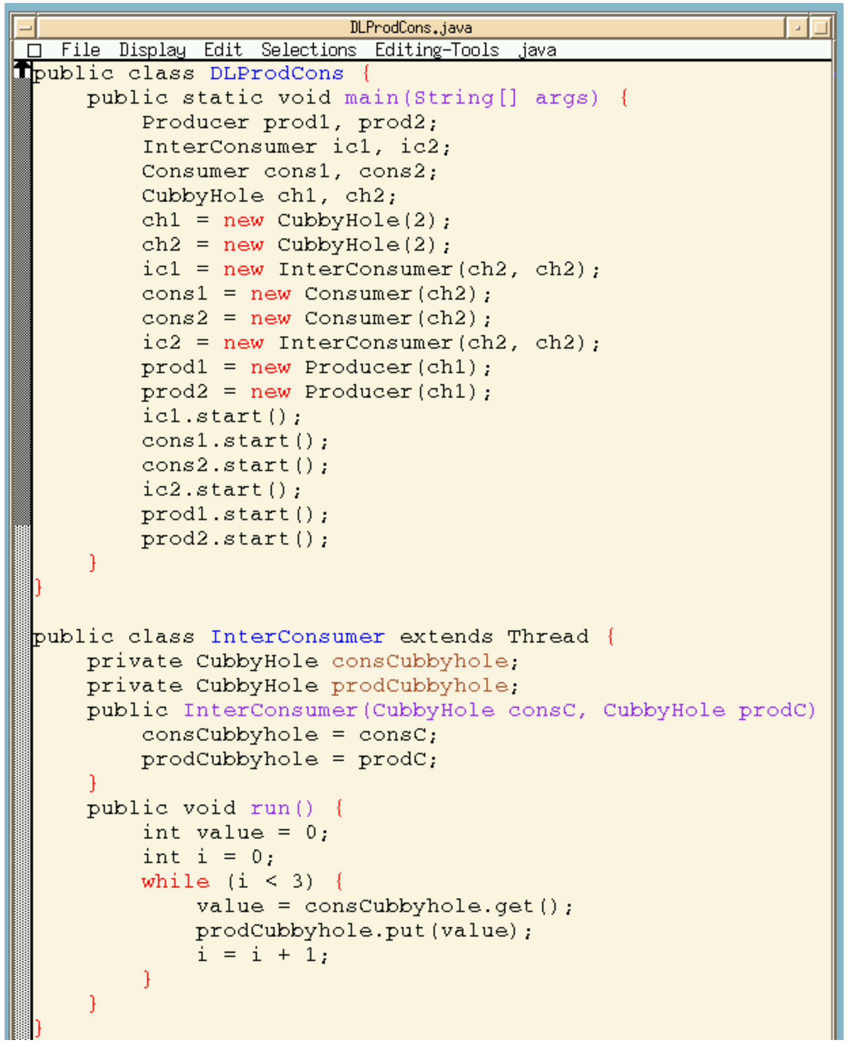


```
ProdCons.java
File Display Edit Selections Editing-Tools java
public class ProdCons {
    Producer p1;
    Consumer c1, c2;
    public static void main(String[] args) {
        CubbyHole c;
        c = new CubbyHole();
        c1 = new Consumer(c);
        p1 = new Producer(c);
        c2 = new Consumer(c);
        c1.start();
        c2.start();
        p1.start();
    }
}

public class CubbyHole {
    private int contents;
    private int nbTimesToConsume = 0;
    CubbyHole() {
    }
    public synchronized int get() {
        while (nbTimesToConsume == 0)
            wait();
        nbTimesToConsume = nbTimesToConsume - 1;
        notifyAll();
        return contents;
    }
    public synchronized void put(int value) {
        while (nbTimesToConsume != 0)
            wait();
        contents = value;
        nbTimesToConsume = 2;
        notifyAll();
    }
}
```

```
public class Producer extends Thread {
    private CubbyHole cubbyhole;
    public Producer(CubbyHole c) {
        cubbyhole = c;
    }
    public void run() {
        int i = 0;
        while (i < 3) {
            cubbyhole.put(i);
            i = i + 1;
        }
    }
}

public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    public Consumer(CubbyHole c) {
        cubbyhole = c;
    }
    public void run() {
        int value = 0;
        int i = 0;
        while (i < 3) {
            value = cubbyhole.get();
            i = i + 1;
        }
    }
}
```

FIGURE 5 Produser / Consumer in Java

Cubby Hole



```
DLProdCons.java
File Display Edit Selections Editing-Tools java
public class DLProdCons {
    public static void main(String[] args) {
        Producer prod1, prod2;
        InterConsumer ic1, ic2;
        Consumer cons1, cons2;
        CubbyHole ch1, ch2;
        ch1 = new CubbyHole(2);
        ch2 = new CubbyHole(2);
        ic1 = new InterConsumer(ch2, ch2);
        cons1 = new Consumer(ch2);
        cons2 = new Consumer(ch2);
        ic2 = new InterConsumer(ch2, ch2);
        prod1 = new Producer(ch1);
        prod2 = new Producer(ch1);
        ic1.start();
        cons1.start();
        cons2.start();
        ic2.start();
        prod1.start();
        prod2.start();
    }
}

public class InterConsumer extends Thread {
    private CubbyHole consCubbyhole;
    private CubbyHole prodCubbyhole;
    public InterConsumer(CubbyHole consC, CubbyHole prodC)
        consCubbyhole = consC;
        prodCubbyhole = prodC;
    }
    public void run() {
        int value = 0;
        int i = 0;
        while (i < 3) {
            value = consCubbyhole.get();
            prodCubbyhole.put(value);
            i = i + 1;
        }
    }
}
```

FIGURE 6 Cubby Hole program

Cubby Hole execution: deadlock

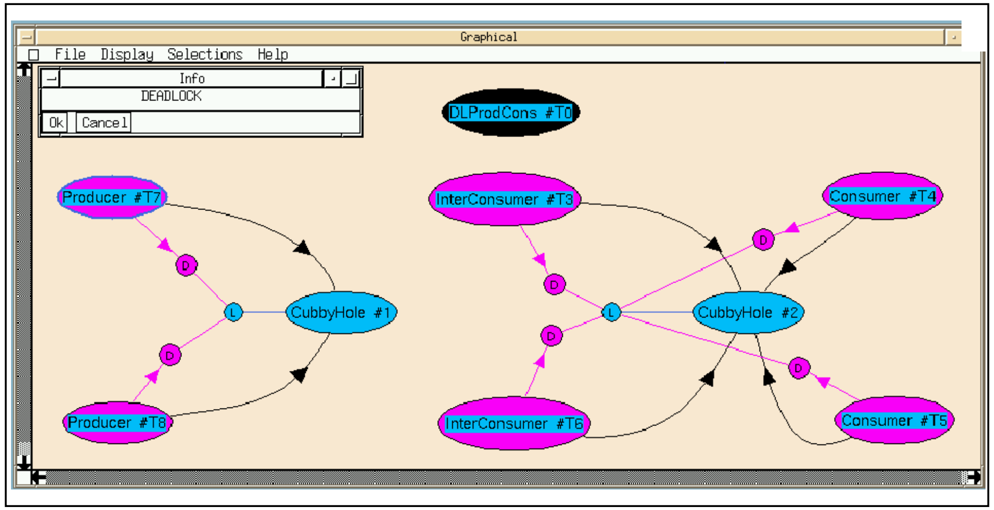


FIGURE 7 Cubby Hole execution: deadlock

Cubby Hole normal execution

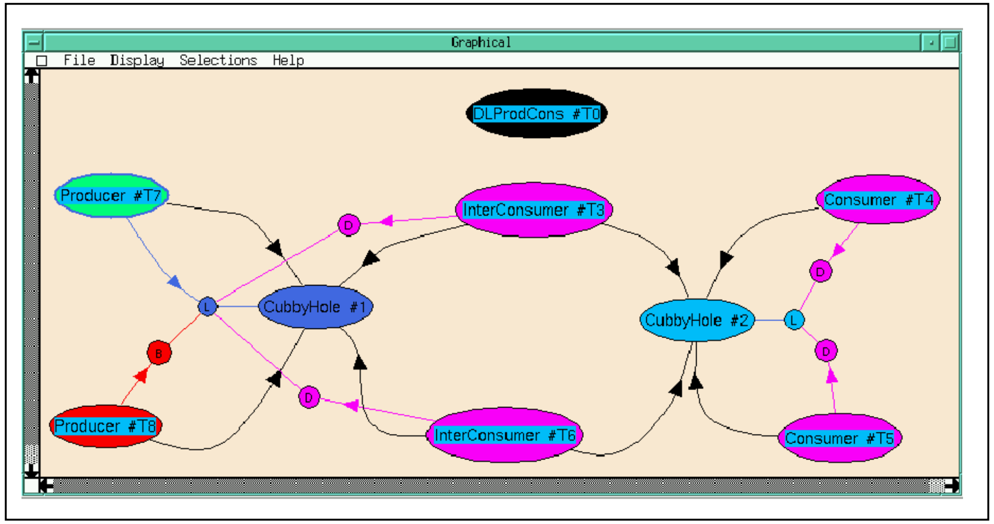


FIGURE 8 Cubby Hole normal execution