Slide 1

INF8541 Paradigmes de programmation:

Motivation et introduction

Paradigme = “Modèle théorique de pensée qui oriente la recherche et la réflexion scientifique”
(Le Petit Larousse 1997)

Paradigme de programmation $\approx$ Façon d’aborder un problème de programmation, généralement à l’aide d’un type de langage qui supporte bien certains mécanismes d’abstractions

- Paradigme procédurale $\Rightarrow$ procédures et sous-routines
- Paradigme orienté objets $\Rightarrow$ classe d’objets
- Paradigme fonctionnel $\Rightarrow$ valeurs et fonctions (mathématiques)

Paradigme comme façon de voir le monde

“*Quand le seul outil que tu connaîs est le marteau, tu vois des clous partout!*”

Slide 2

Soit l’algorithme suivant (maximum parmi une série de nombres):

```
FONCTION max( s: sequence{integer}; i, j: nat ): integer
PRE-CONDITION: i <= j & i IN domain(s) & j IN domain(s)
DEBUT
   max <- s[i]
   POUR k <- i+1 A j FAIRE
      SI a[k] > max ALORS
        max <- a[k]
      FIN
   FIN
   RETOURNER( max )
FIN
```

Quel sera le temps d’exécution pour une séquence S de longueur n?
Plus précisément, quelle est la complexité asymptotique de cet algorithme?

Question: Est-il possible de faire mieux, c’est-à-dire d’obtenir un temps d’exécution qui soit inférieur à $O(n)$ pour une séquence de longueur $n = 2^k$? Si oui, comment?

Slide 3

Solution alternative:

```
FONCTION max( s: sequence{integer}; i, j: nat ): integer
PRE-CONDITION: i <= j & i IN domain(s) & j IN domain(s)
DEBUT
  SI i = j ALORS
    RETOURNER( s[i] )
  SINON
    m <- (i+j) / 2
    max1 <- max( s[i..m] )
    max2 <- max( s[m+1..j] )
    SI max1 > max2 ALORS
      RETOURNER( max1 )
    SINON
      RETOURNER( max2 )
    FIN
  FIN
FIN
```

Question: Quel sera le temps d'exécution de cette algorithme pour une séquence `S` de longueur n ($n = 2^k$)?

Slide 4

Sortons maintenant du cadre habituel où tout s'exécute de façon *séquentielle*.

Supposons que les appels de fonction se fassent *en parallèle*.

```
FONCTION max( s: sequence{integer}; i, j: nat ): integer
PRE-CONDITION: i <= j & i IN domain(s) & j IN domain(s)
DEBUT
  SI i = j ALORS
    RETOURNER( s[i] )
  SINON
     m <- (i+j) / 2
     EN PARALLELE
       max1 <- max( s[i..m] )  // max2 <- max( s[m+1..j]
     FIN
     SI max1 > max2 ALORS
       RETOURNER( max1 )
     SINON
       RETOURNER( max2 )
     FIN
  FIN
FIN
```

Question: Quel sera alors le temps d'exécution en fonction de $n = 2^k$?

Slide 5

Autre exemple: trouver la fin d'une liste chaînee

Soit une liste chainée séquentielle (voir figure au tableau).

Supposons que chaque noeud de la liste soit sur un processeur différent.

Supposons que chaque noeud puisse aussi avoir un pointeur auxiliaire `voisin` en plus du pointeur vers le noeud `suivant`.

Question: Quelle sera la complexité asymptotique du temps d'exécution pour trouver le dernier élément d'une liste de longueur n et faire en sorte que le champ `voisin` du premier noeud pointe vers le dernier?

Slide 6

Soit l'algorithme suivant:

```
SUR CHACUN des processeurs FAIRE
  voisin <- suivant
  TANTQUE (voisin != null) & (voisin->voisin != null) FAIRE
    voisin <- voisin->voisin
  FIN
FIN
```

Question: Quelle est alors la complexité asymptotique du temps d'exécution, pour une liste de longueur $n = 2^k$?

Slide 7

Introduction à la programmation concurrente et parallèle

Ce qu'est la programmation concurrente et parallèle

Programme séquentiel

= programme défini par une *séquence* d'actions

$\Rightarrow$ une instruction après l'autre

= Processus, tâche, *thread*

thread = fil d'exécution

Slide 8

Programme concurrent

= programme qui contient deux ou plusieurs processus qui *communiquent* entre eux

Deux façons possibles de communiquer:

- Par l'intermédiaire de variables *partagées*
- Par l'échange de messages et de signaux

Niveau matériel (*hardware*)

Différentes classes de matériel

- Machine uniprocesseur: boîte contenant un seul processeur
- Multi-processeurs: boîte contenant plusieurs processeurs
  - Communication via un *bus*
- Multi-ordinateurs: plusieurs boîtes inter-connectées
  - Communication via un réseau

Slide 9

Différents types d'applications concurrentes

Application multi-contextes (*multi-threaded***)** = contient plusieurs *thread*s

Note importante: On considère généralement un *thread* comme étant un processus léger (*lightweight thread*)

Utilisations: Pour mieux organiser/structurer une application (plus grande modularité)

- Système d'exploitation multi-tâches
- Fureteurs multi-tâches
- Interface personnes-machines vs. application
- ...

Slide 10

Application parallèle = chaque processus s'exécute sur son propre processeur

Utilisations: Pour résoudre plus rapidement un problème ou pour résoudre un problème plus gros

- Prévisions météorologiques
- Prospection minière
- Physique moderne
- Bio-informatique (génomique)
- ...

Slide 11

Application distribuée = les processus communiquent entre eux par l'intermédiaire d'un réseau ($\Rightarrow$ délais plus longs)

Utilisations:

- Serveurs de fichiers
- Accès à distance à des banques de données

Slide 12

Emphase du cours:

Paradigmes de programmation [concurrente et parallèle]

Différentes façons de catégoriser les langages parallèles

Parallélisme implicite	Parallélisme explicite	
Variables partagées Échanges de messages		Impératif [Procédural] [Objets]
		Déclaratif [Fonctionnel] [Logique]

Slide 13

Parallélisme explicite vs. implicite

- Explicite
  - Instructions spéciales de création de *thread*s, de synchronisation et communication, etc.
  - Exemples: `fork/join`, `cobegin/coend`, `forall`
- Implicite
  - Programme *ordinaire*: c'est le compilateur qui se charge d'identifier et exploiter le parallélisme
  - Exemples: langages fonctionnels, *dusty decks*

Slide 14

Impératif vs. déclaratif

- Impératif
  - Basé sur l'utilisation de structures de contrôle et d'affectations (notion d'*état modifiable*)
  - L'ordre d'exécution est spécifiée par le programmeur
- Déclaratif
  - Basé principalement sur l'utilisation d'expressions, de valeurs et de fonctions
  - L'ordre d'exécution est souvent non-spécifié: dépend des dépendances de données

Slide 15

Variables partagées vs. Échanges de messages

- Variables partagées
  - Les processus partagent un état global modifiable
  - L'information est communiquée entre processus en modifiant les variables partagées
- Échanges de messages
  - Chaque processus n'a accès qu'à ses propres variables locales
  - Les processus communiquent en s'envoyant des messages
    - Synchrone: Les deux processus s'attendent l'un l'autre pour communiquer ($\approx$ téléphone sans boîte vocale)
    - Asynchrone: Un processus peut envoyer un message sans attendre que l'autre soit prêt ($\approx$ courriel)

Slide 16

Quelques langages que nous verrons dans le cours

- ⋆ Threaded-C: explicite, impératif (procédural, mais inspiré du modèle *dataflow*), hybride (variables partagées et échanges de messages)
- Java: explicite, impératif (objets), variables partagées
- MPI: explicite, impératif (procédural), échanges de messages
- pH: implicite, déclaratif (fonctionnel), variables partagées
- `Pthreads` (librairie POSIX)
- ...