Slide 1

Résumé de quelques éléments du chapitre 1 du livre de Andrews

1.3 Les cinq patrons de base de la programmation concurrente

1. Parallélisme itératif: Programme contenant plusieurs processus avec des boucles
2. Parallélisme récursif: Programme avec procédures récursives exécutées en parallèle
3. Producteurs/consommateurs (pipelines, parallélisme fonctionnel): Processus qui communiquent entre eux de façon uni-directionnelle, généralement organisés sous forme de pipeline
4. Clients/serveurs: Processus clients font des requêtes et attendent les réponses; processus serveurs attendent les requêtes et y répondent
5. *Interacting peers*: Processus exécutant le même code et s'échangeant des messages pour accomplir une certaine tâche

Slide 2

1.4 Parallélisme itératif

- Programme itératif = utilise des boucles pour examiner des données et calculer des résultats
- Programme parallèle itératif = contient 2 ou plusieurs processus itératifs
- Chaque processus calcule les résultats pour un sous-ensemble des données, puis les résultats sont combinés
- Souvent utilisé pour des calculs *embarassingly parallel* = problèmes pouvant être décomposés en un grand nombre de parties indépendantes, générant ainsi une grande quantité de parallélisme
- Exemple: Multiplication de matrices, pp. 14–16

Slide 3

1.5 Parallélisme récursif

- Utile si le problème peut être résolu à l'aide d'un programme contenant de nombreux appels récursifs *indépendants* les uns des autres $\Rightarrow$ on fait les appels en parallèle
- Si trop de parallélisme (trop nombreux appels récursifs), alors l'arbre de récursion peut être tronqué (*pruned*), e.g., solution non-récursive/non-parallèle lorsque le sous-problème devient *suffisamment simple*
- Exemple: Intégration numérique, pp. 18–19

Slide 4

1.6 Producteurs/consommateurs (pipeline)

- Producteur = processus qui produit un flot (*stream*) de résultats
- Consommateur = processus qui reçoit un flot de données et le traite
- Filtre = consomme un flot en entrée et produit un nouveau flot en sortie
- Pipeline = séquence de filtres, par ex.:

```
cat $1.tex \
 | sed '/\\begin{figure}/,/\\end{figure}/d' \
 | sed '/\\begin{table}/,/\\end{table}/d' \
 | sed '/\\begin{picture}/,/\\end{picture}/d' \
 | grep -v "^%" \
 | tr "[~]" "[ ]" \
 | tr "[\t]" "[\n]"\
 | tr "[ ]" "[\n]"\
 | grep -v '\\'  \
 | wc -w
```

Slide 5

1.7 Clients/serveurs

Producteur/consommateur ⇒ flot uni-directionnel d'information
(du producteur vers le consommateur)

Client/serveur ⇒ flot *bi-directionnel*

- Client fait une requête au serveur puis attend la réponse
- Serveur reçoit la requête du client, la traite, puis retourne une réponse

Analogie:

- Client = procédure appelante
- Serveur = procédure appelée (par plusieurs autres)

Exemple: systèmes d'exploitation, serveurs de fichiers, bases de données

Slide 6

1.8 Pairs interagissant

peers = pairs

Interacting peers

⇒ processus égaux, interagissant de façon symétrique

⇒ chaque processus exécute le même algorithme et communique avec les autres processus de façon à calculer sa partie des résultats

⇒ communication par échange de messages

- Exemple: multiplication de matrices, p. 25

Slide 7

1.9 Sommaire du pseudo-code concurrent d'Andrews

Pseudo-code basé essentiellement sur le langage C, mais avec des ajouts pour la programmation concurrente

1.9.1 Déclarations

```
int a[n];                          # int a[0:n-1];
int c[2:2*m+2];
double c[n, n] = ([n] ([n] 1.0));
```

1.9.2 Instructions séquentielles de contrôle

```
if (condition)
  statement;
else {
   statement1;
   ...
   statementN;
}
```

```
while (condition) {
  statement1
  ...
  statementN
}
```

Slide 8

```
for [quantifier1, ..., quantifierM] {
   statement1;
   ...
   statementN;
}
```

Quantificateurs possibles:

```
for [i = 0 to n-1] ...

for [i = 0 to n-1, j = 0 to n-1] ...

for [i = 0 to n-1]
  for[j = 0 to n-1] ...

for [i = 0 to n by 2] ...

for [i = 0 to n-1 st i != x] ...
```

Slide 9

1.9.3 Instructions de contrôle `co`, processus et procédures

```
co  statement1;
//  ...
//  statementN;
oc
```

```
co [i = 0 to n-1] {
  statement1;
  ...
  statementN;
}
```

Notes:

- Une instruction `co` se termine lorsque chacune des instructions qui la compose se termine
- La forme de droite crée `n` processus, un pour chaque valeur de `i`, où chaque processus utilise une valeur distincte de `i`

Slide 10

Processus: déclaré comme une procédure, mais amorcé implicitement et s'exécutant en arrière-plan (pas d'attente, comme dans un `co`, qu'il se termine)

```
process foo {
  statement1;
  ...
  statementN;
}
```

```
process bar [i = 1 to n] {
 statement1;
 ...
 statementN;
}
```

Procédure: comme en C

```
int foo( int v ) {
  statement1;
  ...
  statementN;
  return( ... );
}
```

```
void bar() {
  statement1;
  ...
  statementN;
}
```