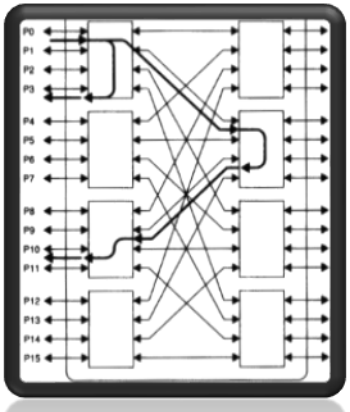
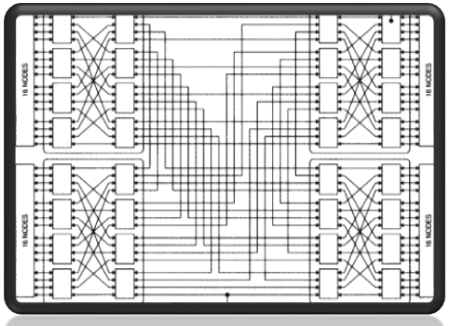


Traitement parallèle de l'information

Dr Maziz Asma

Cours

Master I : Vision Artificielle



Chapitre 3 :
Programmation parallèle

Plan

Généralement, la programmation parallèle est une programmation dans un langage permettant d'indiquer que certaines instructions peuvent être exécutées de façon simultanée sur différents processeurs.

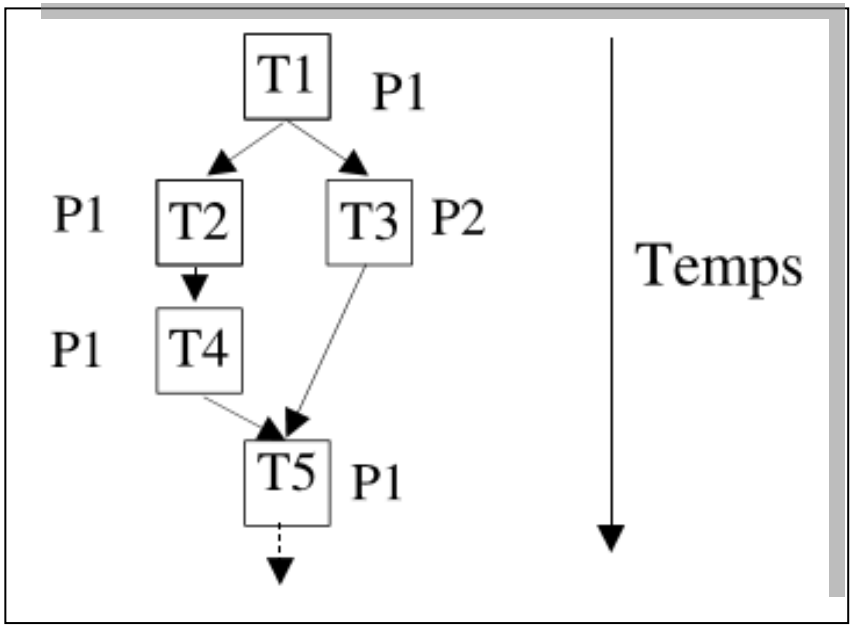
3.1. Sources de parallélisme

On peut avoir différentes sources de parallélisme :

1) Parallélisme de contrôle
2) Parallélisme de données
3) Pipelining

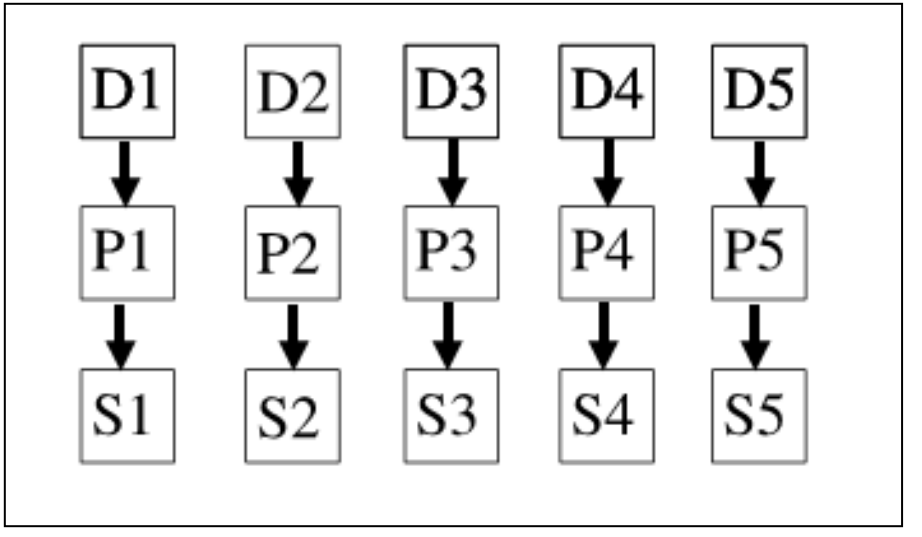
3.1.1. Parallélisme de contrôle

Des opérations sont réalisées simultanément sur plusieurs processeurs. Le programme présente des séquences d'opérations indépendantes qui peuvent être exécutées en parallèle.



3.1.2. Parallélisme de données

- La même opération est effectuée en parallèle par chaque processeur sur des données différentes.



3.1.3. Pipelining

- Une tâche est divisée en plusieurs paliers
- Chaque palier est un processus obtenant son entrée du palier précédent

3.2. Notion de base :

3.2.1. Dépendance

Pour implémenter les algorithmes parallèles, il faut comprendre bien le concept des dépendances des données. Aucun programme ne peut fonctionner plus rapidement que la plus longue chaîne de calculs dépendants (connu sous le nom **chemin critique**), puisque les calculs qui dépendent de calculs antérieurs de la chaîne doivent être exécutées dans l'ordre.

Supposons $\mathbf{P_i}$ et $\mathbf{P_j}$ deux programmes. Les conditions de Bernstein (voire Chaptre1) décrivent quand les deux sont indépendants et peuvent être exécutées en parallèle. Pour $\mathbf{P_i}$: $\mathbf{E_i}$ toutes les variables d'entrée et $\mathbf{S_i}$ les variables de sortie, et de même pour $\mathbf{P_j}$. $\mathbf{P_i}$ et $\mathbf{P_j}$ sont indépendantes si elles satisfont :

1) $S_i \cap E_j = \varphi$,

2) $E_i \cap Sj = \varphi$,

3) $S_i \cap S_j = \varphi$

1) Lorsque la première condition n'est pas vérifiée, il est introduit une **dépendance** d'écoulement, correspondant au premier programme produisant un résultat utilisé par le deuxième programme.
2) La seconde condition représente une **anti-dépendance**, lorsque le deuxième programme produit une variable requise par le premier programme.
3) La troisième et dernière condition représente une **dépendance de sortie**: lorsque deux programme écrite au même emplacement, le résultat provient du dernier programme logiquement exécuté.

Considérez les instructions suivantes, qui démontrent plusieurs types de dépendances:

I1: C = A * B

I2: D = 5 * C

Dans cet exemple, l'instruction I2 ne peut pas être exécutée avant (ou même en parallèle avec) l'instruction 1, car l'instruction 2 utilise un résultat d'instruction 1. Il viole **la condition 1**, et introduit ainsi une **dépendance de flux**.

I1: C = A* B

I2: D = 3 * A

I3: E = A + B

Dans cet exemple, il n'y a pas de dépendances entre les instructions, afin qu'ils puissent tous être exécutés en parallèle.

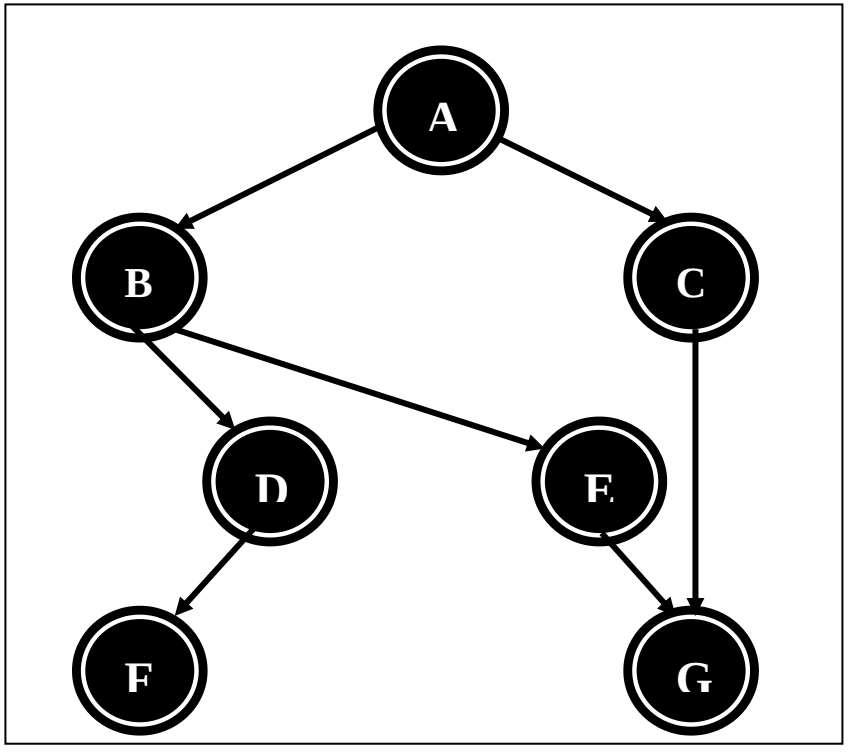
Les conditions de **Bernstein** ne permettent pas de **mémoire** à partager entre les différents processus. Pour cela, des moyens d'exécution d'un ordre entre les accès sont nécessaires, tels que des **sémaphores** ou une autre **méthode de synchronisation.**

3.2.2. Graphe de dépendance

Permet de représenter visuellement le parallélisme d'un algorithme

- Sommet = tâche
- Arc = Relation de dépendance

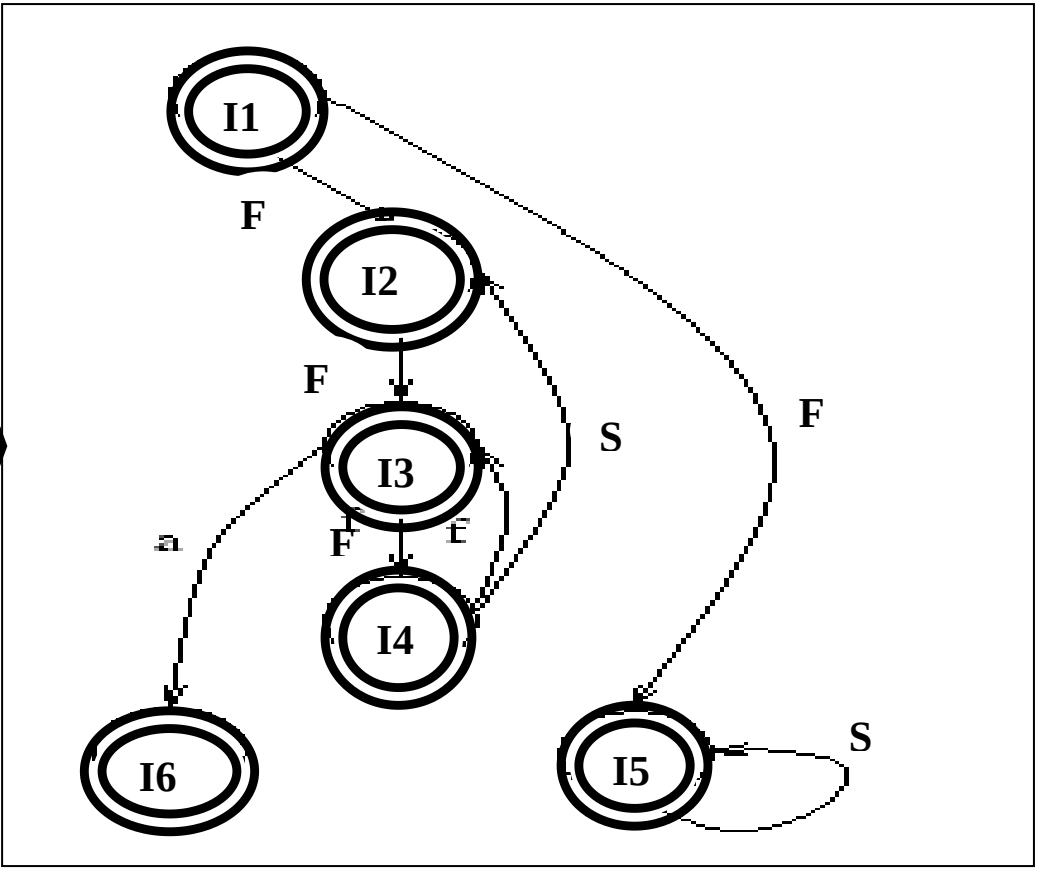
La notion de graphe de dépendance permet de décrire les types de dépendances entre données et d'extraire le meilleur parallélisme possible d'un programme.



Supposons le sous-programme suivant :

```
I1: x=y+1;
    for (i=2;i<=30;i++)
    {
I2: C[i]=x+B[i];
I3: A[i]=C[i-1]+z;
I4: C[i+1]=B[i]*A[i];
    for (j=2;j<=50;j++)
   {
I5:  F[i,j]=F[i,j-1]+x;
    }
     }
I6: z=y+3;
```

Graphe de dependence associé:



3.2.3. Synchronisation

La coordination des tâches en temps réel est souvent associée aux communications, elle est souvent implémentée en introduisant un point de synchronisation au-delà duquel la tâche ne peut continuer tant qu'une ou plusieurs autres tâches ne l'ont pas atteint.

3.2.4. Communication

Les tâches parallèles échangent des données par différents moyens physiques : via un bus à mémoire partagée, via un réseau. Quel que soit la méthode employée, on parle de «communications».

3.3. Programmer un système parallèle

Il existe quatre stratégies:

1) Compilateur étendu

2) Langage étendu : Ajouter des opérations parallèles

3) Ajouter une couche au-dessus d'un langage séquentiel

4) Définir un nouveau langage parallèle

3.3.1. Compilateur étendu

- Traduire un programme séquentiel en programme parallèle

Avantages :

- ✓ Gain du temps
- ✓ Ne requiert aucune formation pour les programmeurs
- ✓ Il est plus facile de programmer de façon séquentielle
- ✓ ……..

3.3.1. Compilateur étendu

Désavantages :

- L'aide du programmeur est nécessaire pour extraire un maximum de parallélisme d'un problème,
- La technologie actuelle ne fonctionne que dans des situations simples: boucle, etc.
- Le code résultant peut devenir complexe.

3.3.2. Langage étendu

- On ajoute des fonctionnalités à un langage existant
  - Créer et terminer des processus parallèles
  - Synchronisation nécessaire
  - On doit permettre aux processus de communiquer
  - Ex. MPI, PVM, Processus/thread, OpenMP

3.3.2. Langage étendu

Avantages :

- ✓ Les compilateurs sont faciles à implémenter
- ✓ Grande flexibilité dû à l'accès à des fonctions de bas niveau
- ✓ Méthode très populaire

3.3.2. Langage étendu

Désavantages :

- Aucune aide de la part du compilateur
- Programmes très difficiles à déboguer

3.3.3. Ajouter une couche

➢ Couche inférieure

- On peut utiliser un langage standard
- Manipule sa part des données et sa partie des sorties

➢ Couche supérieure

- Création et synchronisation des processus
- Répartition des données entre les processus
- Un nouveau langage parallèle est nécessaire
- Le compilateur est responsable de traduire ces deux niveaux en code exécutable.

3.3.3. Ajouter une couche

Quelques prototypes existent :

Computationally Oriented Display Environment (CODE).

3.3.4. Nouveau langage

- Développer un langage entièrement nouveau
  - Exemple : Occam (1983)
- Ajouter de nouvelles instructions à un langage existant:
  - Fortran 90
  - High Performance Fortran
  - C*
  - Cilk

3.3.4. Nouveau langage

Avantages :

- Permet au programmeur d'indiquer au compilateur où se trouve le parallélisme
- Permet d'améliorer les performances

3.3.4. Nouveau langage

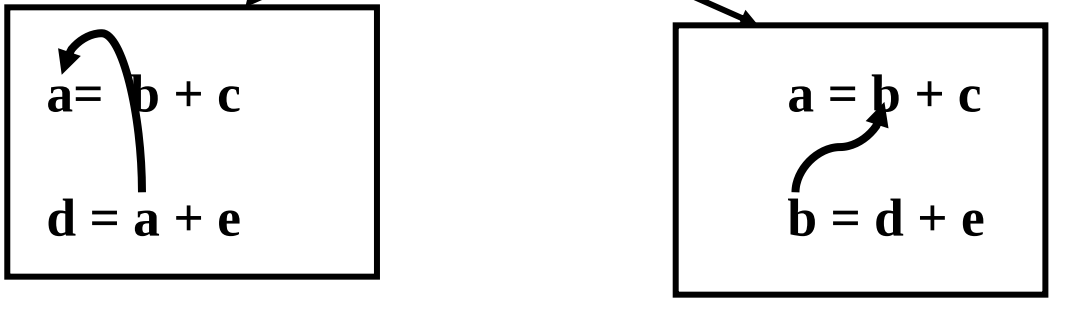
Désavantages :

- Requiert un nouveau compilateur
- Pas de standard
- Réticence des programmeurs.

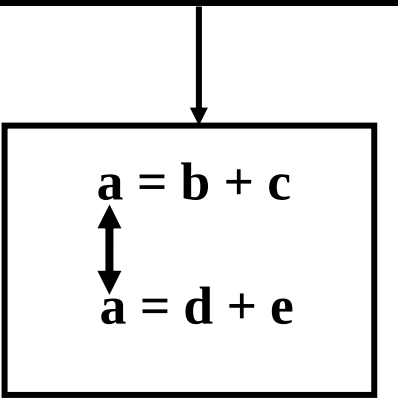
3.4. Limitation du parallélisme

a) Les dépendances de données :

Dépendance de flot



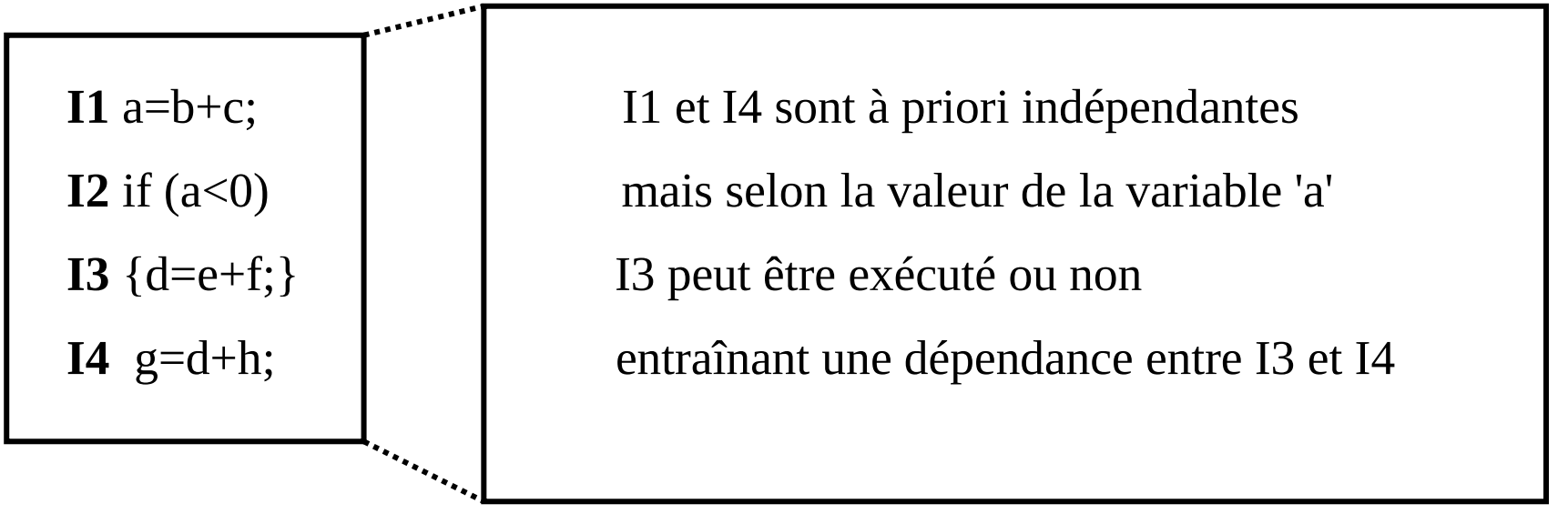
Dépendance de sortie



- Les instructions doivent être indépendantes :
  - exécuter dans un ordre quelconque
  - simultanément

3.4. Limitation du parallélisme

b) Les dépendances de contrôle :



3.4. Limitation du parallélisme

c) Les dépendances de ressources :

- Nombre insuffisant de processeurs pour effectuer les instructions en parallèle alors qu'elles sont indépendantes les unes des autres.

3.4. Limitation du parallélisme

d) Rapport temps de communication / temps de calcul :

- Il n'est pas toujours avantageux de paralléliser une application. Les communications peuvent dans certain cas augmenter le temps d'exécution.