Introduction au parallélisme

Mésocentre de calcul de Franche-Comté

Kamel Mazouzi

05/05/2010

UNIVERSITÉ DE FRANCHE-COMTÉ

mésocentre de calcul de franche-comté

Plan

- Introduction
- Calcul haute performance et parallélisme
- Les architectures parallèles
- Modèles de programmation parallèle
- Performance

Motivation pour la haute performance

- **Simulation numérique** : expériences !!
  - trop grandes : météorologie
  - trop petites : biologie, matériaux
  - trop coûteuses : crash-tests, conception aéronautique
  - impossibles : climat, astrophysique

Motivation pour la haute performance

- À réaliser en réalité :
  - serveurs traditionnels : Fichiers, Web, Vidéo, Base de données, …
  - serveurs de calcul : « on demand computing »
  - réalité virtuelle ou augmentée : médias, médical

Ces applications peuvent consommer une puissance de calcul et demander une capacité mémoire supérieure à celle d'un ordinateur personnel

Motivation pour la haute performance

- Besoins importants :
  - en puissance de calcul
  - en mémoire

- Ordres de grandeur : G : giga, T : téra, P : péta
  - 1 TeraFLOPs = 10^{12} = 1billion opérations flottantes par seconde
  - 1 PetaByte = vidéo de 2300 ans, 1 milliard de livres

Calcul haute performance ?

- **Calcul haute performance =**
  - algorithme efficace
  - calcul de solutions approchées
  - etc

+ Parallélisme ?

Le parallélisme ?

- **Programmation séquentielle :**
  - un seul flot d'exécution
  - une instruction exécutée à la fois
  - un processeur
- **Programmation parallèle :**
  - plusieurs flots d'exécution
  - plusieurs instructions exécutées simultanément
  - plusieurs processeurs

Parallélisme, pourquoi ?

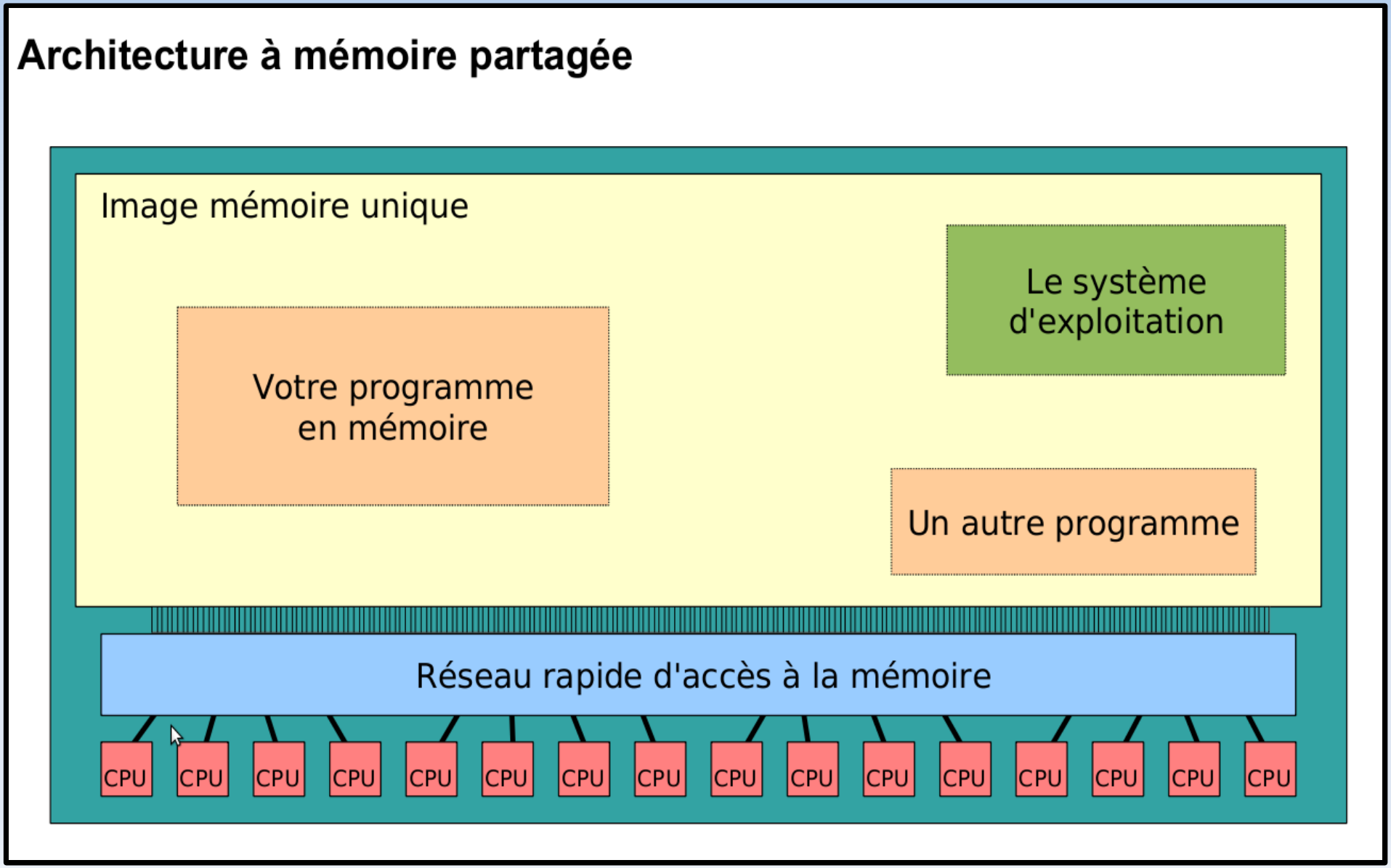
- **Résoudre de plus gros cas**
  - La parallélisation à mémoire distribuée permet de s'affranchir de la limite de mémoire physiquement présente sur un nœud de calcul, en distribuant les données
- **Résoudre les cas dans un temps plus court**
  - La parallélisation permet de diviser les calculs à effectuer entre plusieurs processeurs

Modèle pour le parallélisme

- **Programmation séquentielle**
  - modèle unique (von Neumann)
- **Programmation parallèle**
  - pas de modèle unique
  - différents aspects à considérer
    - **Architecture** (mémoire partagée/distribuée,...)
    - **Logiciel** (modèle de programmation parallèle,...)
    - **Algorithmes** (penser en parallèle,...)

Les architectures parallèles

Les architectures parallèles

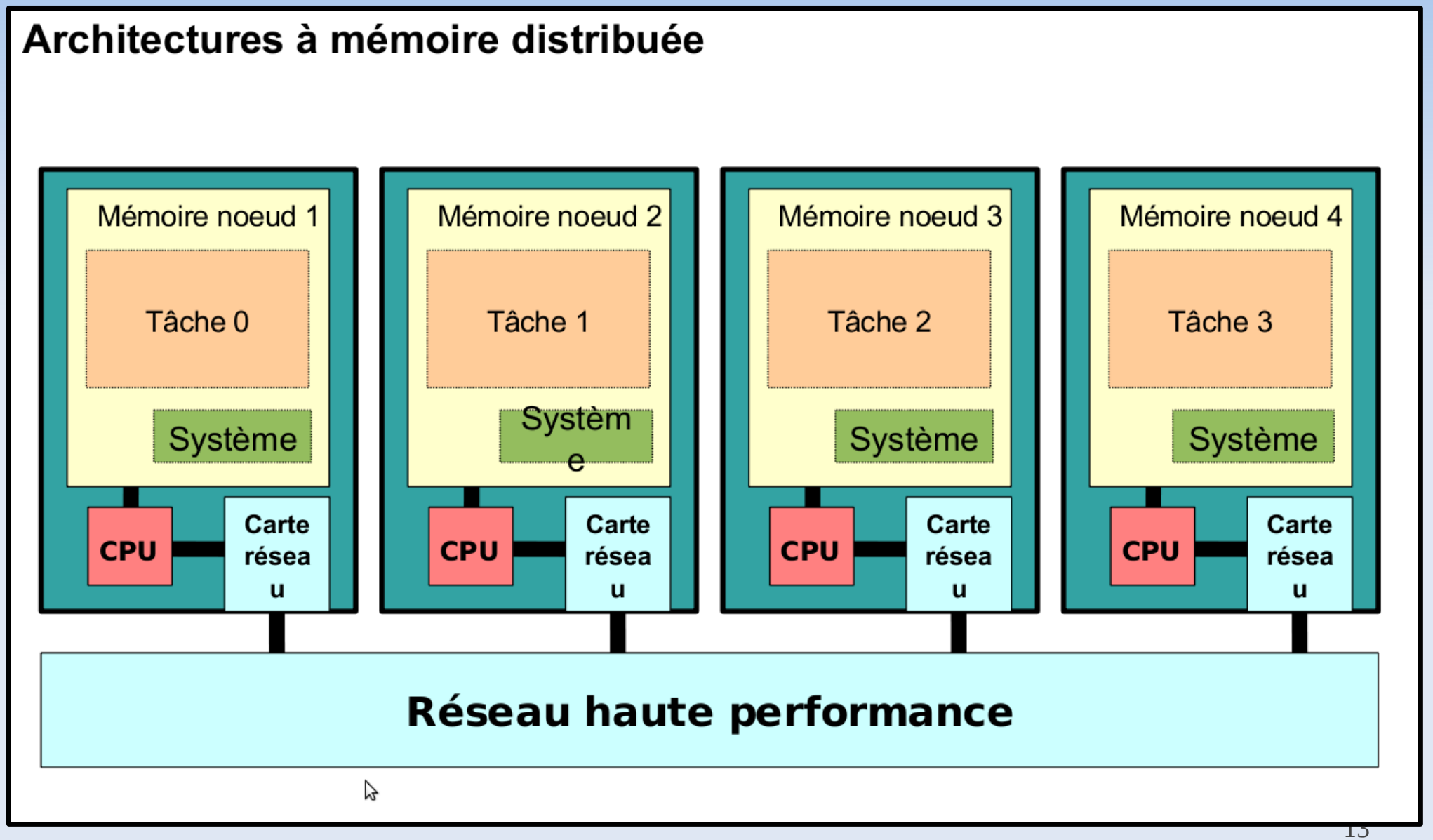


Les architectures parallèles

- **Avantages :**
  - espace d'adressage global fournit une perspective de programmation facile
  - le partage des données entre les tâches est à la fois rapide et uniforme

- **Inconvénients :**
  - faible extensibilité : ajout de CPU peut augmenter le trafic CPU-MEM
  - la synchronisation et la cohérence des données sont à la charge du programmeur

Les architectures parallèles

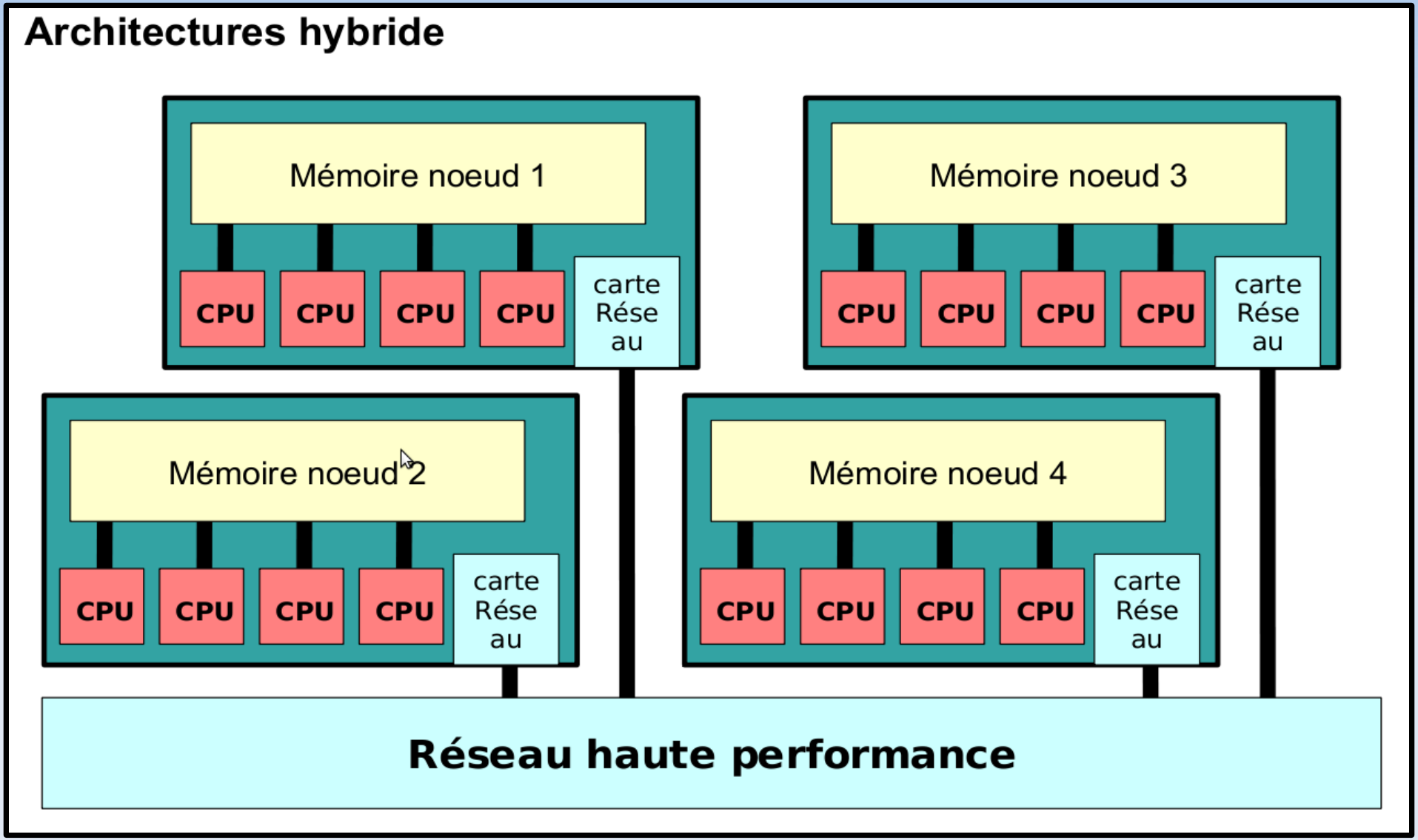


Les architectures parallèles

- **Avantages :**
  - la mémoire est extensible avec le nombre des processeurs
  - chaque processeur peut rapidement accéder à sa propre mémoire sans interférences

- **Inconvénients :**
  - la distribution des données et les communications entre les processeurs sont à la charge du programmeur
  - Il est parfois difficile de « mapper » des données sur une telle organisation de la mémoire

Les architectures parallèles



Modèles de programmation parallèle

Multiplicité des modèles de programmation

Deux grandes approches du parallélisme :

- **Parallélisme de tâches** (Control Parallelism)
  - décomposition d'une tâche en sous-tâches
    - variables partagées entre les tâches
    - communications par messages entre les tâches

Parallélisme = réalisation simultanée de différents traitement sur les données

  - Exemple : **MPI** sur architectures à mémoire distribuée

Multiplicité des modèles de programmation

- **Parallélisme de données** (Data Parallelism)
  - Structures homogènes « tableaux »
  - Affectation / distribution de données

Parallélisme = réalisation simultanée du même traitement sur des données différents

Exemple : **OpenMP** sur architectures à mémoire partagée

- **Combinaison des modèles !**

Exemple

- Calcul d'un vecteur de polynômes

```
pour i=0 à n-1 faire
vv[i]=a+b.v[i]^2 + c.v[i]^3 + d.v[i]^4
+ e.v[i]^5 + f.v[i]^6 + g.v[i]^7
fin pour
```

• Les instances du corps de la boucle (pour les différentes valeurs de **`i`**) peuvent s'exécuter indépendamment

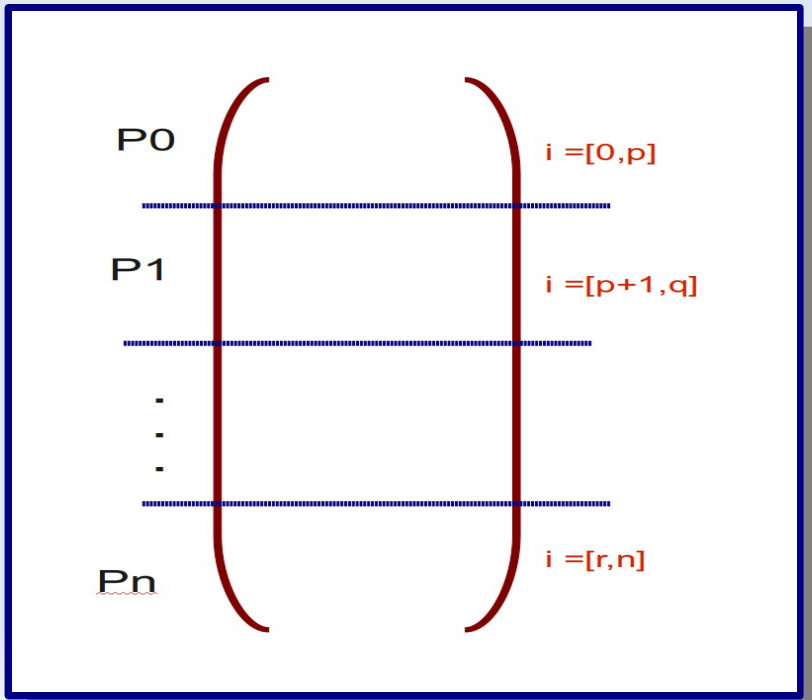
• Possibilité de les exécuter dans un ordre quelconque, ou en parallèle

Exemple (2)

```
pour i=0 à n-1 faire_en_parallèle
vv[i]=a+b.v[i]^2 + c.v[i]^3 + d.v[i]^4
+ e.v[i]^5 + f.v[i]^6 + g.v[i]^7
fin pour
```

- Le mot clé `faire_en_parallel` indique une exécution parallèle des différentes itérations de la boucle

- Si on dispose de `n` processeurs, chacun d'eux exécute une partie de la boucle pour une valeur différente de `i`



Exemple (3)

```
X = a+b.v[i]^2 + c.v[i]^3;
Y = d+e.v[i]+ f.v[i]^2 + g.v[i]^3
Z = v[i]^4

vv[i] = x+z.y
```

```
pour i=0 à n-1 faire
tâches_parallèle
X = a+b.v[i]^2 + c.v[i]^3;
||
Y = d+e.v[i]+ f.v[i]^2 + g.v[i]^3
||
Z = v[i]^4
fin tâches_parallèle
vv[i]=x+z.y
fin pour
```

Task 1

Task 2

Task 3

Chaque tâche s'exécute sur un processeur

Synchronisation pour calculer la valeur finale

Implications du parallélisme

- **Nouveaux concepts**
  - concurrence
  - communication
  - synchronisation
  - dépendances
- **Facteurs de performance**
  - distribution des données
  - granularité
  - équilibrage de charge

- **Problèmes Spécifiques**
  - interblocage
  - terminaison
  - convergence
- **Dimension algorithmique**
  - raisonnement parallèle
  - complexité

Granularité

Un échange d'information est requis pour permettre la coordination de plusieurs processeurs pour la résolution d'un problème

- La granularité est le **ratio** entre les **calculs**

  et les **communications** :

  - **Gros grain : plus** de calcul
  - **Faible grain : plus** de communication

Scalabilité (passage à l'échelle)

- Propriété d'une application à être exécutée efficacement sur un grand nombre de processeurs
- Facteurs favorisant une faible scalabilité :
  - architecture inadaptée
  - charge déséquilibrée
  - communications
  - algorithme

Dépendances

- Une dépendance sur les données existe lorsqu'il y a utilisation multiple du même espace mémoire
- **La dépendance réduit le potentiel parallèle d'une exécution**

```
do i=2,N
  A(i) = A(i-1) + B(i)
end do
```

Concurrence critique (Race Condition)

- Situation dans laquelle les résultats dépendent de l'ordre dans lequel les processus concurrents accèdent à une ressource commune

Processus 1

```
if (x>A)
 A=x
```

Processus 2

```
if (y>A)
  A=y
```

Interblocage (Deadlock)

- Un ensemble de processus est en interblocage si chacun des processus attend un événement que seul un autre processus dans l'ensemble peut déclencher
- Un événement peut être :
  - accès à une section critique
  - accès à une ressource

Processus 1

```
recv(Data,2)
send(Data,2)
```

Processus 2

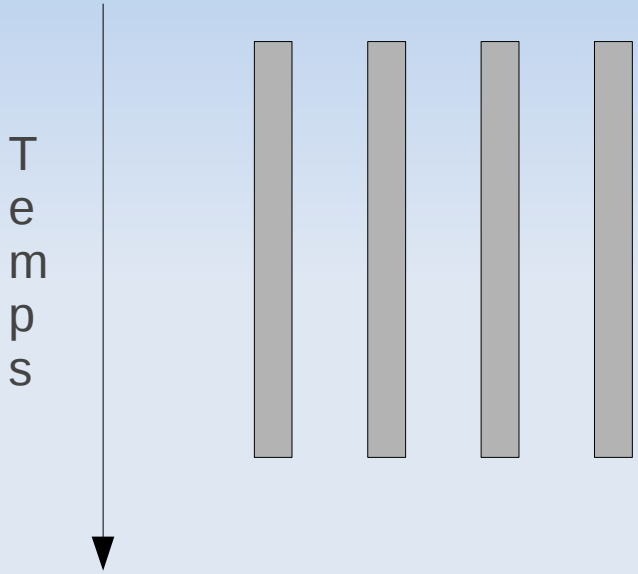
```
recv(Data,1)
send(Data,1)
```

Équilibrage de charge

- Statique
  - en général : trouver la distribution de données la plus adaptée
  - optimiser les communications
- Dynamique
  - dans le cas où le calcul dépend des données
  - dans le cas d'une machine hétérogène ou non-dédiée

Équilibrage de charge : exemple

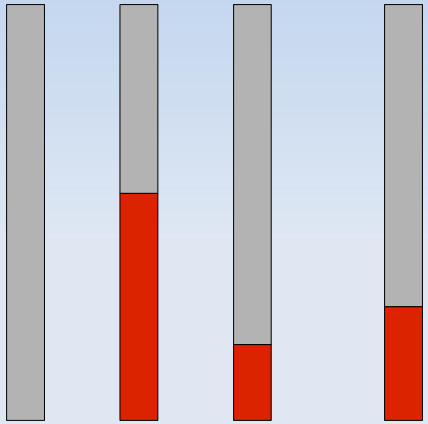
Répartition parfaite



Tous les processeurs terminent en même temps

Processeur inactif

Charge déséquilibrée



- Les tâches ont des temps de traitement différents
- Temps réel d'exécution augmente
- Les temps d'exécution ne sont plus proportionnels à la taille du problème

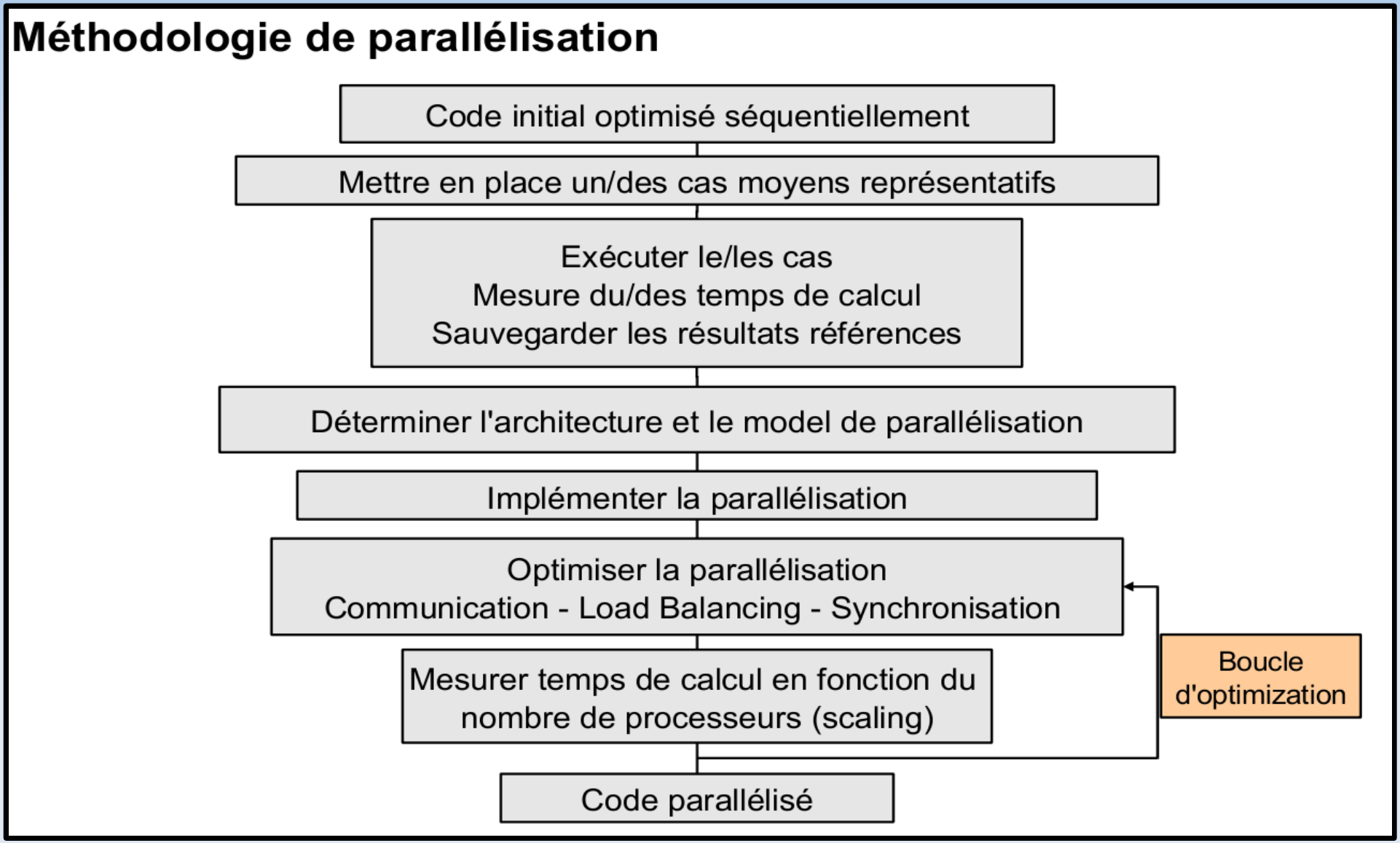
À la découverte du parallélisme

- **Où est le parallélisme dans l'application ?**
  - étude des **dépendances** entre les données
- Découpage en tâches élémentaires
  - parfois évident
  - peut nécessiter des transformations du programme
  - peut aller jusqu'à des transformations de l'algorithme
- Partitionnement des données
  - recherche des données « proches »
  - **but :** minimiser les communications

Choix du modèle de programmation

- Une affaire de compromis
  - efficacité
  - facilité de programmation
  - maintenance du code
  - expertise de l'utilisateur
  - compilateurs de la machine cible

Méthodologie



Performance

Performance

- Temps entre le début et la fin de l'application
  - utilisateur : minimiser ce temps
  - minimise le coût d'une exécution
    - heures CPU payantes

Métrique

- **Temps d'exécution**
  - en secondes : wallclock time ou elapsed time
  - en temps CPU
- **Nombre d'opérations (flottantes)**
  - nombre de cycles nécessaires à l'obtention du résultat
  - **`add, sub, mul`**= un cycle

    `div, sqrt` = 9 cycles

    `sin, exp` = 17 cycles
  - Unité : Flop, MFlop, GFlop
  - Vitesse : MFlop/s, GFlop/s

Facteur d'accélération

- Un algorithme
  - exécution sur **p** processeurs en un temps t_p
  - exécution sur **1** processeur en un temps t_1
- Accélération (speedup)

$$S_p = \frac{T_1}{T_p}$$

- Étude de l'accélération

$$1 \leq S_p \leq p$$

Loi d'Amdahl

- Trouver une borne à l'accélération
  - pour un problème donné
  - pour une taille de problème donnée
- Identifier la partie parallélisable du code
  - charge de travail fixe
  - distribution de cette charge entre P processeurs
  - décomposition de la charge : W

$$W = \alpha W + (1-\alpha) W$$

Partie séquentielle

Partie parallèle

Loi d'Amdahl

- Calcul de l'accélération :

$$S_p = \frac{t_1}{t_p} = \frac{W}{\alpha W + (1-\alpha)(W/P)} = \frac{P}{1+(p-1)\alpha} \rightarrow \frac{1}{\alpha} \; quand \; P \rightarrow \infty$$

- Accélération pour **P** processeur :

$$S_p = \frac{P}{1+(p-1)\alpha}$$

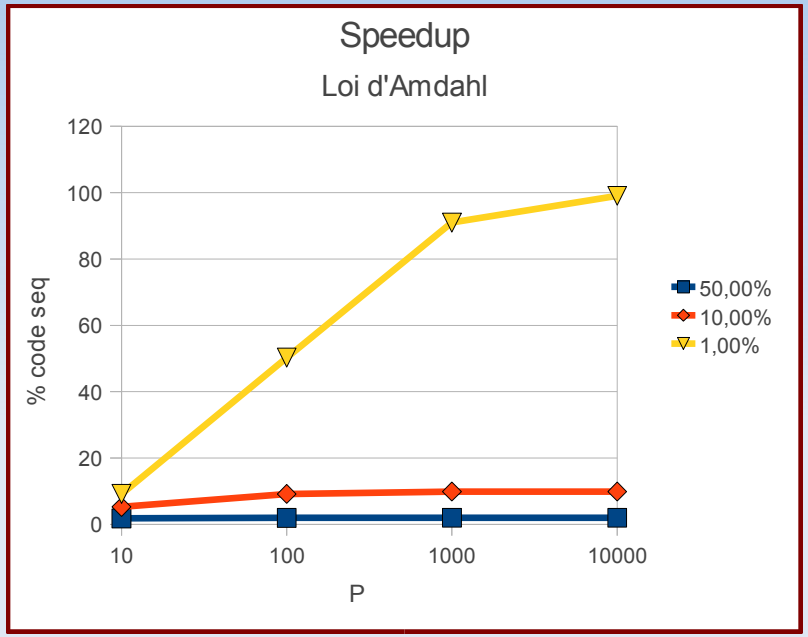
pourcentage de code séquentiel

  - **Code séquentiel :** $S_p = 1$
  - **Code «purement» parallèle :** $S_p = p$

Loi d'Amdahl

- Exemple

P	50 %	10%	1%
10	1,82	5,26	9,17
100	1,98	9,17	50,25
1000	1,99	9,91	90,99
10000	1,99	9,91	99,02



- Accélération limitée par la partie séquentielle
- Pour une meilleure accélération : diminuer la partie séquentielle

La partie non parallèle limite les performances et fixe une borne supérieure à la scalabilité

Questions ?