Slide 1

Architectures parallèles: Un aperçu

1. Pourquoi des machines parallèles?
2. Classification de Flynn
3. Processeurs modernes et parallélisme
4. Évolution des architectures parallèles
5. Rôle et gestion des caches
6. Modèles de *consistence* de mémoire

Slide 2

1 Pourquoi des machines parallèles?

- Améliorations importantes au niveau des micro-processeurs
  - Augmentation de la vitesse d'horloge $\approx$ 30 % par année
  - Augmentation de la densité des circuits $\approx$ 40 % par année
- mais . . .
- Plusieurs applications ne peuvent pas être traitées en un temps raisonnable sur un ordinateur traditionnel
  - simulation et analyse de phénomènes physiques ou biologiques, e.g., prédiction météorologique, physique atomique, analyse de protéines, génome
  - traitement d'images
  - analyse du langage parlé
- Limite physique à l'amélioration de la vitesse des circuits

Slide 3

2 Classification de Flynn

Classe les machines selon le nombre de flux de données et d'instructions:

SISD = *Single Instruction stream, Single Data stream*: Uniprocesseur classique exécutant (du point de vue du programmeur) une seule instruction à la fois, e.g., machine RISC moderne.

SIMD = *Single Instruction stream, Multiple Data streams*: Machine où une seule instruction s'exécute sur plusieurs données indépendantes à la fois.

MISD = *Multiple Instruction streams, Single Data stream*: Aucune machine connue.

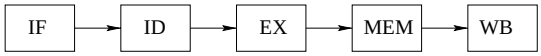
MIMD = *Multiple Instruction streams, Multiple Data streams*: Plusieurs instructions distinctes s'exécutant sur des données indépendantes.

Slide 4

3 Processeurs modernes et parallélisme

3.1 Pipeline d'analyse et d'exécution des instructions

- Plusieurs instructions consécutives peuvent être en cours d'exécution au même moment, mais à des étapes différentes



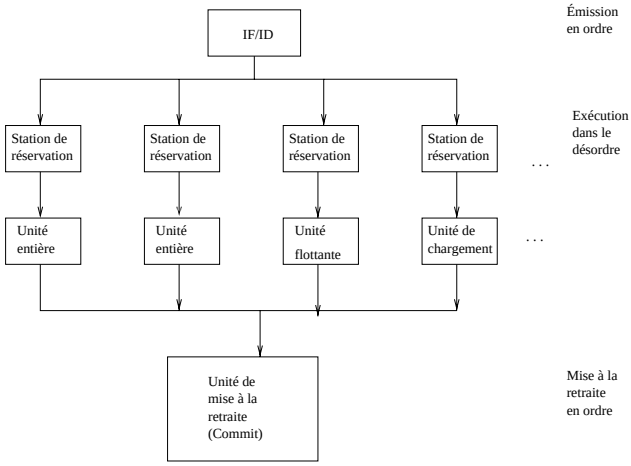
IF: Instruction Fetch
ID: Instruction Decode
EX: Execute
MEM: Memory Access
WB: Write Back

- Permet la réduction du temps de cycle du processeur (augmentation de la fréquence de l'horloge)
- Problèmes et solutions:
  - Aléas de données $\Rightarrow$ unité de *forwarding*
  - Aléas de contrôle $\Rightarrow$ *prédiction dynamique des branchements*

Slide 5

3.2 Machines superscalaires et ordonnancement dynamique des instructions

- Plusieurs instructions peuvent être en cours d'exécution dans des unités d'exécution différentes (en autant que les dépendances de données soient respectées)

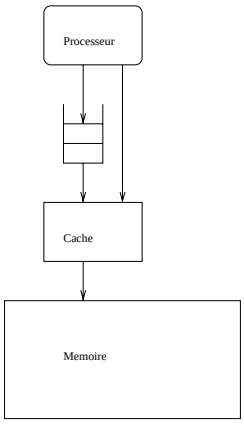


Slide 6

3.3 Tampon d'écriture et exécution pipelinée des accès mémoire

Exécution d'un `store`: Le processeur *amorce* l'exécution du `store` puis continue immédiatement avec l'instruction suivante

$\Rightarrow$ Plusieurs accès mémoire en cours d'exécution

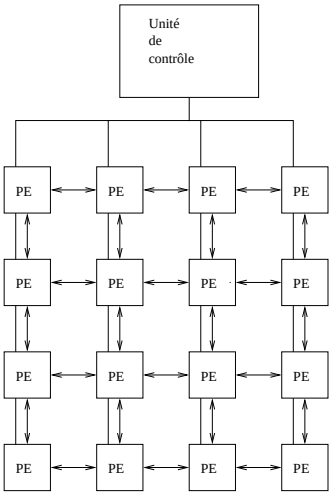



```
store $2, 100($6)      ; Complète immédiatement
store $4, 104($7)      ; Complète immédiatement
load $3, 104($7)       ; Attente pour la valeur
```

Slide 7

4 Évolution des architectures parallèles

4.1 Machine SIMD



- Très courantes parmi les premières machines parallèles mais de moins en moins usité de nos jours.
- Typiquement des machines qui manipulent des tableaux (matrices) de données, conduisant à du “parallélisme de données”.

```
VAR A, B, C: ARRAY[1..4, 1..4] OF REAL;
C := A + B;
```

Slide 8

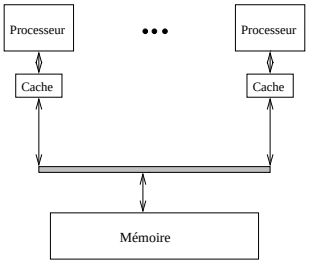
- Motivations initiales pour de telles machines:
  - Obtenir une machine parallèle de coût faible en dupliquant uniquement les unités d’exécution sans dupliquer l’unité de contrôle.
  - Minimiser l’espace mémoire requis en ayant une seule copie du code.
- Désavantage majeur: Excellent pour des programmes manipulant des tableaux uniformes de données, *mais* très difficile à programmer et à utiliser efficacement (les instructions conditionnelles peuvent conduire un grand nombre de processeurs à rester inactifs).

Dernier exemple important (années 80) d’une telle machine: *Connection Machine*, avec 64K processeurs . . . à 1 bit chacun

Slide 9

4.2 Machine MIMD avec bus et mémoire partagée

Multiprocesseurs



Avantages/désavantages:

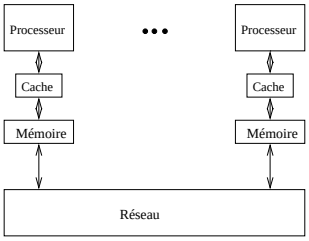
+ Problème de la cohérence des caches facilement résolu (*snoopy bus*)

− Ne permet pas le parallélisme massif (max. $\approx$ 30 processeurs) pcq. *1 seule* transaction à la fois sur le bus

Slide 10

4.3 Machine MIMD avec réseau et mémoire distribuée

Multi-ordinateurs



Avantages/désavantages:

+ Permet un plus grand parallélisme (transactions multiples sur le réseau)

= Problème de la cohérence des cache résoluble mais plus complexe

− Temps de latence *imprévisible* (*NUMA = Non-Uniform Memory Access*) $\Rightarrow$ Que faire en cas de faute de cache/accès réseau?

4.4 Architectures hybrides

4.4.1 Mémoire partagée vs. mémoire distribuée

Slide 11

Important de distinguer les niveaux *physique* et *virtuel*

- Physique: Est-ce que la mémoire est physiquement distribuée (répartie) entre les processeurs?
- Virtuel: Est-ce que, pour le programmeur, la mémoire semble partagée entre les processeurs?

Tendance récente sur de nombreuses machines = Mémoire virtuelle partagée mais physiquement distribuée

$=$ le système d'exécution (RTS) s'occupe de gérer les accès non-locaux

$\Rightarrow$ aucun contrôle sur les coûts de communication

4.4.2 Machines NOW

Slide 12

$=$ réseaux de stations de travail (*Network Of Workstations*)

- Les superordinateurs sont trop coûteux
  (mauvais rapport performance/prix, i.e., un peu plus performant mais *beaucoup* plus coûteux)
- Les processeurs *spécialisés* ne réussissent pas à suivre l'évolution rapide des processeurs

Exemple: Earthquake = Mise en oeuvre de l'architecture EARTH sur un *Beowulf cluster*

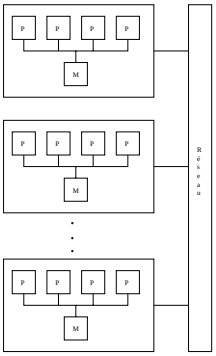
- 16 noeuds (boîtes Linux)
- 1 noeud = 1 processeur Pentium 500 MHz + 128 MB de mémoire
- Sauf 1 noeud contrôleur = 2 processeurs + 256 MB de mémoire
- Réseau 10/100 Mbps pouvant accepter jusqu'à 36 ports

Slide 13

4.4.3 SMP clusters

= grappes de multi-processeurs (*Symmetric Multi-Processors*)

- Chaque noeud est en fait un multi-processeur
  (groupe de processeurs reliés par un bus)
- La machine est composé d'un groupe de tels noeuds

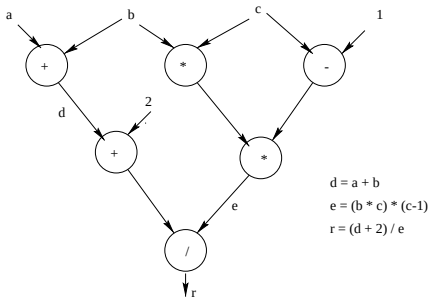


Slide 14

4.5 Architectures *dataflow* et multi-contextes

4.5.1 Architectures à flux de données (*Data Flow architectures*)

Idée de base: Programme = graphe des dépendances de données



$\Rightarrow$ Ordre partiel d'exécution $\Rightarrow$ beaucoup de parallélisme de fine granularité

Désavantage majeur:

- Granularité fine $\approx$ chaque expression est un *thread* indépendant
  $\Rightarrow$ coûts élevés de synchronisation

Slide 15

4.5.2 Architectures multi-contextes (*multithreaded architectures*)

- Contexte (*thread*) = `FP + IP + registres`
- Processeur multi-contextes ≈ hybride uniprocesseur/machine *dataflow*

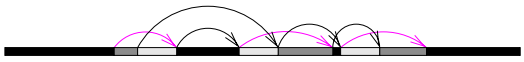
- Conserve plusieurs contextes indépendants ⇒ (presque) toujours plusieurs instructions prêtes à s'exécuter
- Changements de contexte rapides et peu coûteux

Slide 16

- Rapidité des changements de contexte
  ⇒ changement en cas d'accès réseau
  ⇒ Meilleure utilisation du processeur

Sans changement de contexte:

Avec changement de contexte:



Légende:

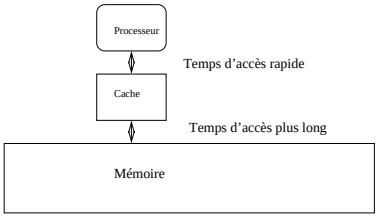
- Une flèche indique un accès non-local nécessitant un certain temps d'attente
- Un trait épais d'une certaine couleur indique un fil d'exécution
- Un espace vide indique que le processeur attend donc n'est pas utilisé

Slide 17

5 Rôle et gestion des caches

5.1 Rôle des caches

Objectif: diminuer le temps de latence des accès mémoire (localité spatiale et temporelle)



Cache: contient des copies des informations récemment accédées

Adresse désirée déjà dans le cache

$\Rightarrow$ temps accès $\approx$ temps cycle de machine

Adresses désirée pas dans le cache (faute de cache)

$\Rightarrow$ temps accès $\approx$ plusieurs cycles machine

$\Rightarrow$ **gel** du pipeline du processeur (*pipeline stall*) $\Rightarrow$ très coûteux

Slide 18

5.2 Caches multiple et protocoles de cohérence des caches

Multi-processeurs/ordinateurs $\Rightarrow$ plusieurs caches $\Rightarrow$ plusieurs copies

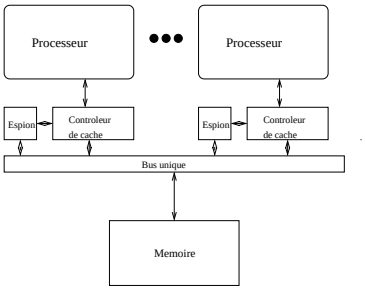
Cohérence des caches = assurer que les copies sont consistentes entre elles

Protocole de cohérence de caches = ensemble de règles pour assurer, en fonction de l'architecture sous-jacente, la cohérence du contenu des caches

Slide 19

5.3 Protocole pour bus: Espionnage

Espionnage = *snoopy bus*



- Contrôleur de cache écoute le bus pour déterminer s'il a ou non une copie d'un bloc transmis sur le bus:
  - Possède *une* copie en lecture d'un bloc demandé pour écriture $\Rightarrow$ le bloc est invalidé dans le cache
  - Possède *la* copie en écriture demandé pour lecture $\Rightarrow$ le bloc est écrit en mémoire (sera lu par demandeur) puis invalidé dans le cache

Slide 20

5.4 Protocole pour réseau: Utilisation d'un répertoire

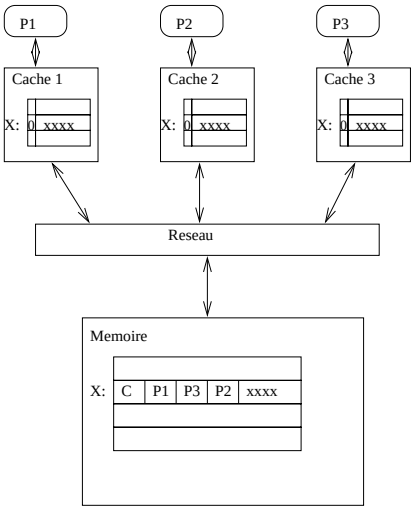
Une machine utilisant un réseau ne peut pas utiliser l'espionnage pcq. cela requiert la diffusion (*broadcasting*) des invalidations $\Rightarrow$ trop coûteux sur un réseau

Solution alternative: utilisation d'un répertoire (*directory scheme*)

- Un répertoire est associé à chaque module de mémoire
- Le répertoire contient une liste de *pointeurs vers les caches* qui ont une copie d'un bloc provenant du module mémoire associé

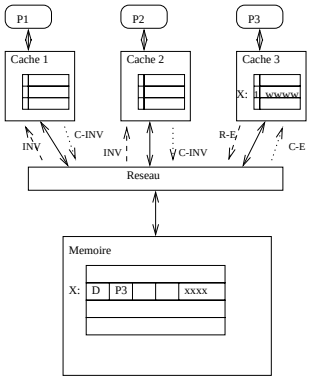
Slide 21

Exemple: P1, P2 et P3 ont une copie du contenu de l'adresse X dans leur cache.



Slide 22

Suites d'événements si le processeur P3 désire écrire à l'adresse X:



(1) R-E = Requete Ecriture (acces Exclusif) (=> P3 bloque (stalled))

(2) INV = INValidation

(3) C-INV = Confirmation INValidation

(4) C-E = Confirmation Ecrirure

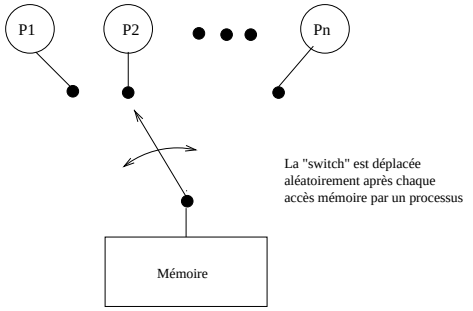
1. Le cache 3 détecte que le bloc est valide mais que l'accès en écriture exclusive n'est pas permis (lecteurs multiples).
2. Le cache 2 envoie une requête d'écriture à la mémoire et suspend (*stall*) le processeur 3.
3. Le module mémoire envoie des requêtes d'invalidation aux caches 1 et 2.
4. Les caches 1 et 2 recoivent les requêtes d'invalidation, invalident les blocs et retournent une confirmation.
5. Une fois toutes les confirmations reçues par le module mémoire, celui-ci envoie la permission d'écriture à la cache 3.
6. La cache 3 reçoit la permission d'écriture, met à jour le statut du bloc mémoire et réactive le processeur P3.

Slide 23

6 Modèles de consistence mémoire

Modèle de consistence mémoire (pour une mémoire partagée) = contraintes qui spécifient l'ordre dans lequel les opérations d'accès à la mémoire semblent se produire relativement les unes aux autres

6.1 Modèle de consistence séquentielle



= Les accès mémoires apparaissent comme un *entrelacement* des accès faits par les divers processus, en respectant l'ordre séquentiel de chacun des processus

Slide 24

Exemple

	P1		P2
I_1 :	b = 0	I_3 :	a = 0
I_2 :	a = 1	I_4 :	b = 1

Résultats possibles selon le modèle de consistence mémoire séquentielle:

$I_1 \rightarrow I_2 \rightarrow I_3 \rightarrow I_4 \quad \Rightarrow$ a = 0 b = 1

$I_1 \rightarrow I_3 \rightarrow I_2 \rightarrow I_4 \quad \Rightarrow$ a = 1 b = 1

$I_1 \rightarrow I_3 \rightarrow I_4 \rightarrow I_2 \quad \Rightarrow$ a = 1 b = 1

$I_3 \rightarrow I_1 \rightarrow I_2 \rightarrow I_4 \quad \Rightarrow$ a = 1 b = 1

$I_3 \rightarrow I_1 \rightarrow I_4 \rightarrow I_2 \quad \Rightarrow$ a = 1 b = 1

$I_3 \rightarrow I_4 \rightarrow I_1 \rightarrow I_2 \quad \Rightarrow$ a = 1 b = 0

Résultat impossible selon consistence séquentielle: a = 0 et b = 0

Mais . . .

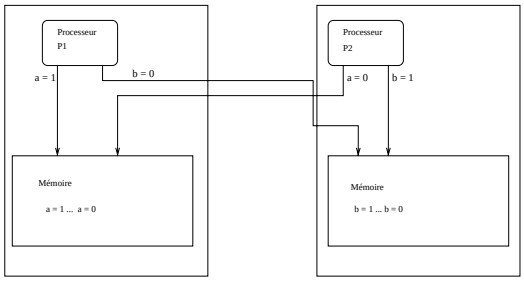
Supposons que

Slide 25

Supposons que a est dans la mémoire locale de P1 et b est dans la mémoire locale de P2

$\Rightarrow$ Temps pour que P1 accède à b est beaucoup plus long que le temps pour accéder à a

$\Rightarrow$ Temps pour que P2 accède à a est beaucoup plus long que le temps pour accéder à b



Donc, un résultat possible serait: a = 0 et b = 0!

Slide 26

Mise en oeuvre et coûts de consistence séquentielle

- Tous les accès mémoires d'un processeur doivent se compléter dans l'ordre dans lequel ils apparaissent dans le programme et cet ordre doit être le même *pour tous* les processeurs

  $\Rightarrow$ Nécessite de retarder l'émission d'un accès mémoire jusqu'à ce que les accès précédents aient été complétés

  $\Rightarrow$ Une écriture doit être *confirmée* avant qu'une lecture ou écriture subséquente puisse s'amorcer, donc la mémoire doit retourner une *confirmation*

Implications:

- Augmente le temps d'exécution des programmes (attente pour confirmations)
- Augmente le traffic sur le réseau (traffic de confirmations)

Slide 27

6.2 Modèles plus faibles de consistence mémoire

Objectif = Améliorer les performances en réduisant les temps d'attente et le traffic sur le réseau

Stratégie = Assurer une consistence séquentielle uniquement pour *certaines* instructions *spéciales* de synchronisation:

$=$ Consistence séquentielle uniquement pour les instructions spéciales

$\Rightarrow$ Instructions ordinaires d'accès peuvent s'exécuter rapidement
(sans tenir compte de la consistence)

Stratégie d'écriture des programmes:

1. Identifier les endroits où des résultats inconsistents peuvent être produits (accès concurrents à des variables partagées)
2. Insérer suffisamment d'instructions de synchronisation pour prévenir les effets indésirés

Slide 28

Exemple = *weak consistency*

P1	**P2**
`test&set(lock);`	`while (!mailbox_full)`
`mailbox = message1;`	`{}`
`mailbox_full = TRUE;`	`test&set(lock);`
`unset(lock);`	`m = mailbox;`
	`unset(lock);`

Traitement des instructions de synchronisation:

1. Une instruction de synchronisation ne peut s'amorcer que lorsque les instructions de synchronisation précédentes ont globalement complété.
2. Une instruction de synchronisation ne peut s'amorcer que lorsque *tous* les accès mémoire précédents sont *globalement* complétés.
3. Les accès qui suivent ne peuvent s'amorcer que lorsque l'instruction de synchronisation a elle-même *globalement* complété

Écritures $\Rightarrow$ confirmation que l'écriture a complété doit avoir été reçue (invalidation des caches!)

Slide 29

Exemple: *release consistency*

Encore moins contraignant que *weak consistency*:

P1	**P2**
`acquire(lock);`	`while (!mailbox_full)`
`mailbox = message1;`	`{}`
`mailbox_full = TRUE;`	`acquire(lock);`
`release(lock);`	`m = mailbox;`
	`release(lock);`

Traitement des instructions de synchronisation

- `acquire`: Retarde l'exécution des accès mémoire ordinaires jusqu'à ce que le *acquire* soit globalement complété
- `release`: Ne s'exécute que lorsque tous les accès précédents ont globalement complété