

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
7. Conclusion : Problèmes des superordinateurs modernes

1 Pourquoi des machines parallèles?

- Améliorations importantes au niveau des micro-processeurs (Loi de Moore)
 - Augmentation de la vitesse d'horloge $\approx 30\%$ par année
 - Augmentation de la densité des circuits $\approx 40\%$ par année

Loi de Moore : Nombre de transistors sur les puces de 1971 à 2011
(Notez l'échelle verticale **logarithmique**!)

Voir figure page suivante — Source : [serc.carleton.edu/csinparallel/modules/
intro_parallel.html](http://serc.carleton.edu/csinparallel/modules/intro_parallel.html)

- mais ...
- Plusieurs applications ne peuvent pas être traitées en un temps raisonnable sur un ordinateur traditionnel
 - simulation de phénomènes physiques ou biologiques : prévisions météorologiques «précises», modélisation du climat, physique haute énergie, analyse de protéines, analyse du génome
 - traitement d'images
 - analyse du langage parlé
- Limites physiques à l'amélioration de la vitesse des circuits
 - Vitesse de la lumière
 - Consommation d'énergie et dissipation de la chaleur

SIX CHIX

BY RINA PICCOLO



2 Classification de Flynn

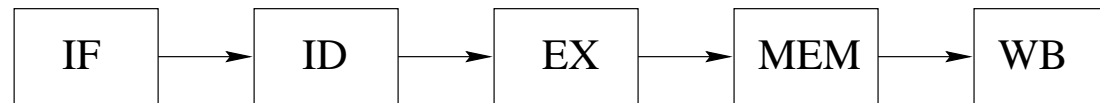
Classe les machines selon le nombre de flux de données et d'instructions :

		Nombre de flux de données	
		Un seul	Plusieurs
Nombre de flux d'instructions	Un seul	SISD	SIMD
	Plusieurs	MISD	MIMD

3 Processeurs (SISD) 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 d'envoi* (forwarding)
 - Aléas de contrôle \Rightarrow *prédiction (dynamique) des branchements*

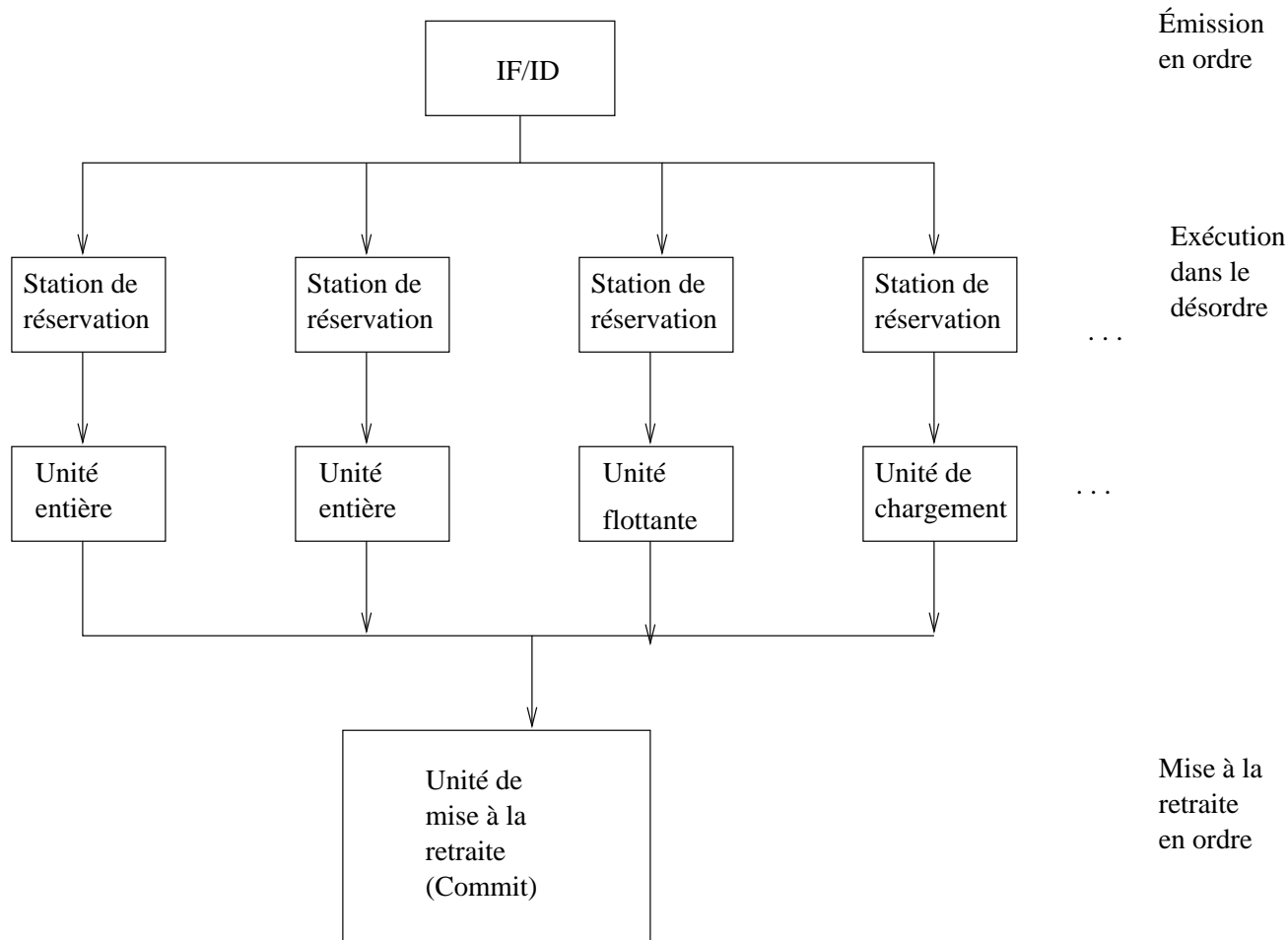
Analogie pour les temps d'accès aux caches vs. mémoire vs. disque

- *L1 Cache - There is a sandwich in front of you.*
- *L2 Cache - Walk to the kitchen and make a sandwich*
- *RAM - Drive to the store, purchase sandwich fixings, drive home and make sandwich*
- *Hard Drive - Drive to the store. purchase seeds. grow seeds, harvest lettuce, wheat, etc. Make sandwich.*

Source : <https://keon.io/cs/computer-scientists-trivia>

3.2 Machines superscalaires et ordonnancement dynamique des instructions

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

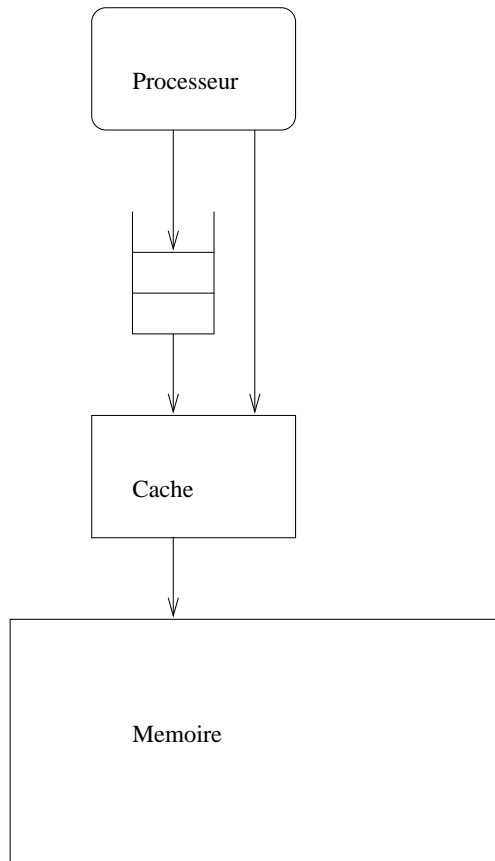


Mais, en pratique : au plus quatre (4) instructions par cycle 😞

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

⇒ Plusieurs accès mémoire en cours d'exécution



<code>store \$2, 100(\$6)</code>	; Complète immédiatement
<code>store \$4, 104(\$7)</code>	; Complète immédiatement
<code>load \$3, 104(\$7)</code>	; Attente pour la valeur

4 Évolution des architectures parallèles

1960–70 Machines SIMD

1980– Machines multi-processeurs à mémoire partagée

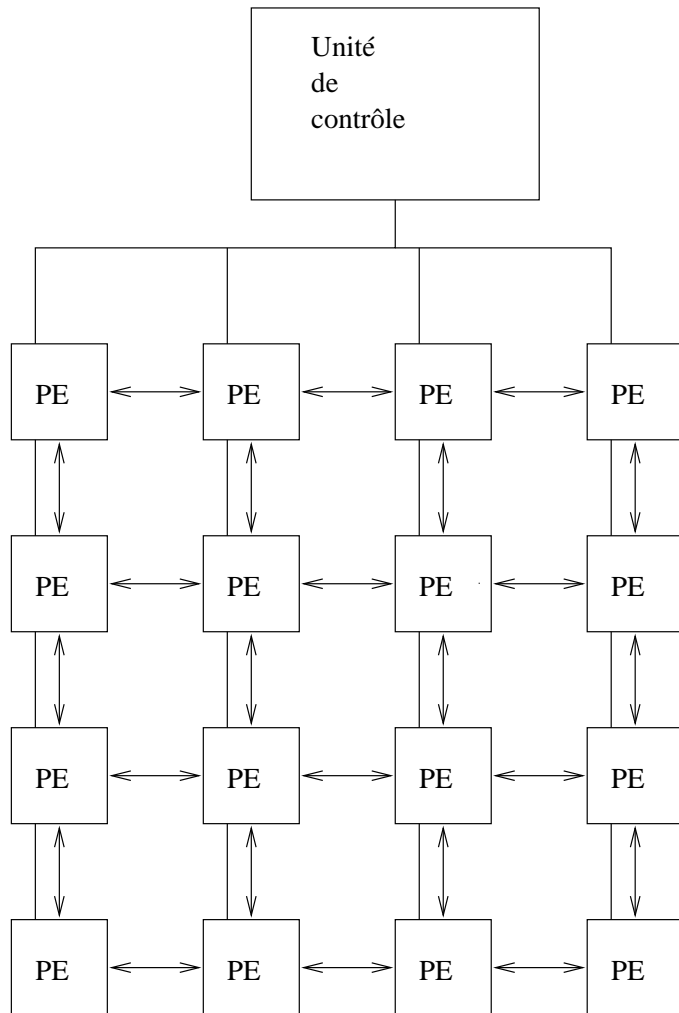
1990– Machines multi-ordinateurs à mémoire distribuée

1990– Machines hybrides

2000– Machines à coeurs multiples

2000– Processeurs graphiques (GPU)

4.1 Machine SIMD



- Courant parmi les premières machines parallèles, a disparu pendant approx. 20–30 ans, pour ensuite réapparaître (GPU).
- Typiquement : machines qui manipulent des tableaux (matrices) de données, conduisant à du **parallélisme de données**.

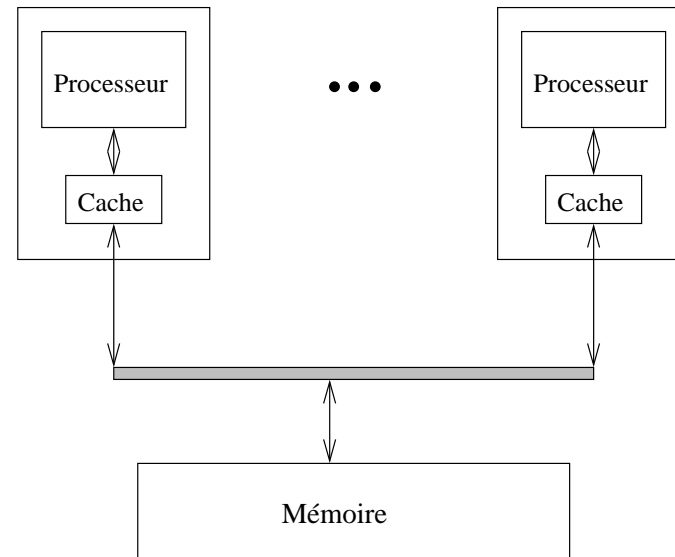
```
VAR A, B, C : ARRAY[1..4, 1..4] OF REAL;  
C := A + B;
```

- 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

4.2 Machine MIMD avec bus et mémoire partagée

Multiprocesseurs

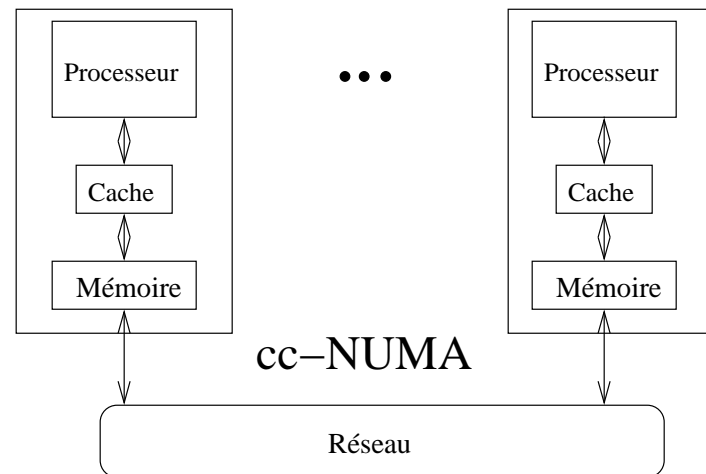


Avantages/désavantages :

- + Problème de la cohérence des caches facilement résolu (*snoopy bus*)
- Ne permet pas le parallélisme massif (max. ≈ 30 processeurs) pcq. *une seule* transaction à la fois sur le bus

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

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 certaines machines = Mémoire virtuelle partagée mais physiquement distribuée

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

⇒ aucun contrôle sur les coûts de communication

4.4.2 Machines NOW (*clusters*)

NOW = *Network Of Workstations* = Réseaux de stations de travail

- Les superordinateurs sont trop coûteux
(mauvais rapport performance/prix, i.e., plus performant mais *très* coûteux)
- L'utilisation des processeurs *spécialisés* ne permet pas de suivre l'évolution rapide des processeurs

Un *cluster* style Beowulf (première machine style NOW)



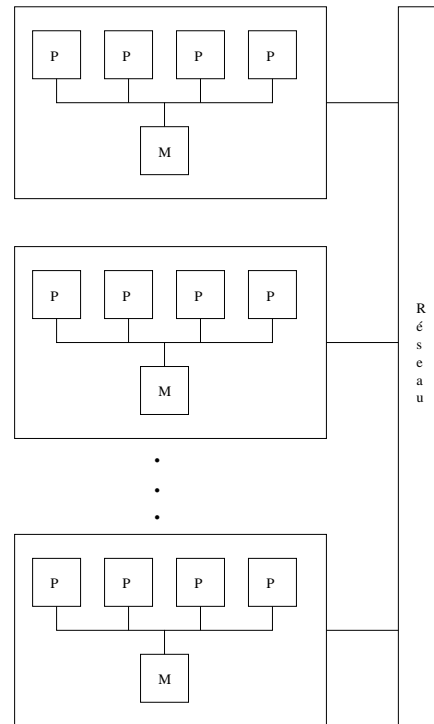
Le superordinateur Cray 1 (circa 1976)



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 composée d'un groupe de tels noeuds



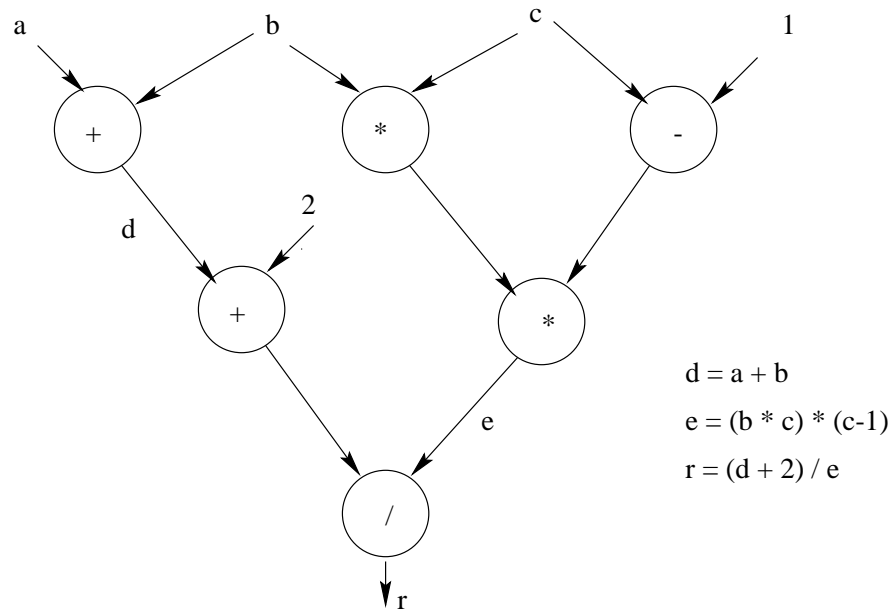
Exemple (2014) : *cluster* turing.hpc.uqam.ca du LAMISS

- 1 *head node* — turing
- 30 noeuds — quark20, ..., quark49
 - 1 noeud = 4 processeurs x86_64
- Système d'exploitation = Linux Cent OS

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



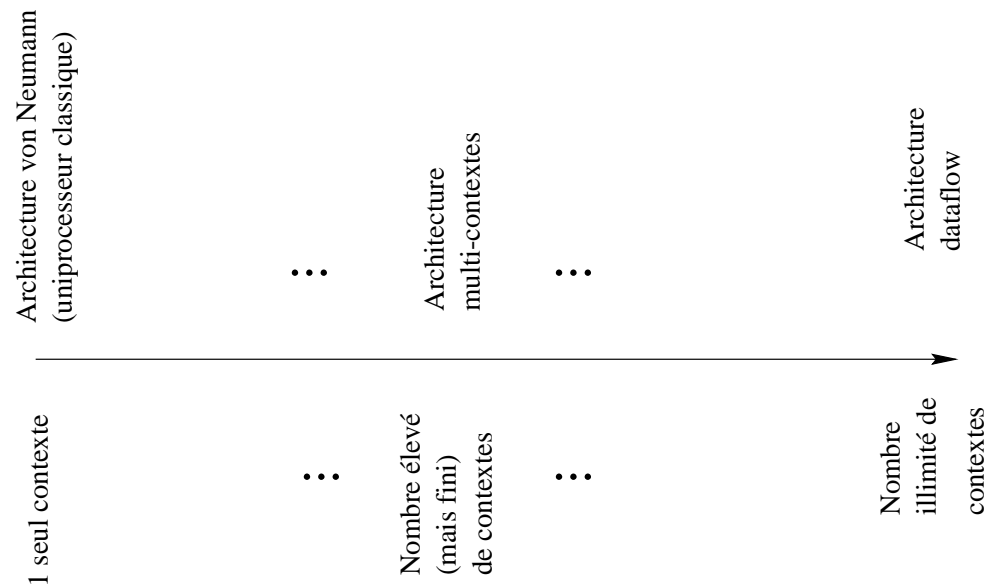
⇒ Ordre partiel d'exécution ⇒ beaucoup de parallélisme de fine granularité

Désavantage majeur :

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

4.5.2 Architectures multi-contextes (*multithreaded architectures*)

- Contexte (*thread*) = FP + IP + registres
- Processeur multi-contextes \approx hybride uniprocasseur/machine *dataflow*



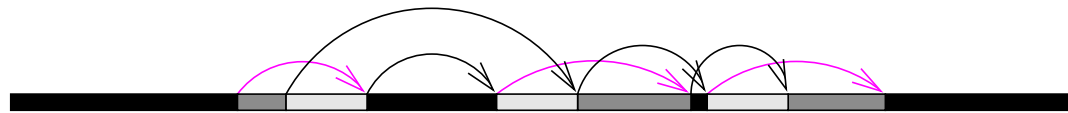
- Conserve plusieurs contextes indépendants \Rightarrow (presque) toujours plusieurs instructions prêtes à s'exécuter
- Changements de contexte rapides et peu coûteux

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

Sans changement de contexte:



Avec changement de contexte:



Légende :

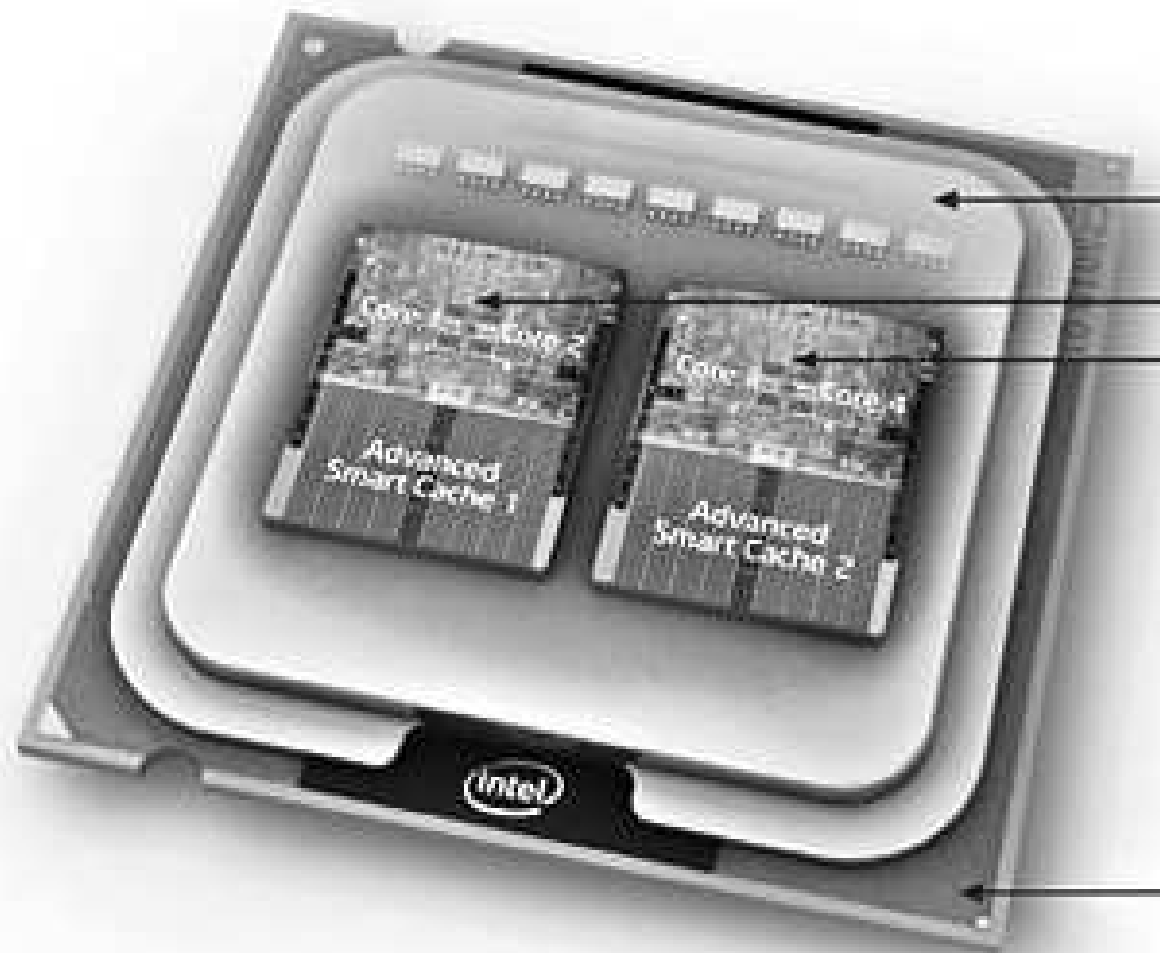
- Une flèche indique un accès non-local nécessitant un certain temps d'attente
- Un trait épais d'une certaine couleur indique un fil d'exécution
- Un espace vide indique que le processeur attend donc n'est pas utilisé

4.6 Machines *multicore*

- Constat depuis quelques années :
 - On ne réussit plus à augmenter la vitesse d'horloge ☹️
 - + On réussit encore à réduire la taille des circuits 😊

- Solutions pour utiliser l'espace des puces :
 - Pipeline d'instruction
 - Plusieurs unités de traitement (machines superscalaires)
 - Plusieurs contextes (*multithreading*)
 - Augmentation de la taille des caches (L1 et L2) *sur la puce*

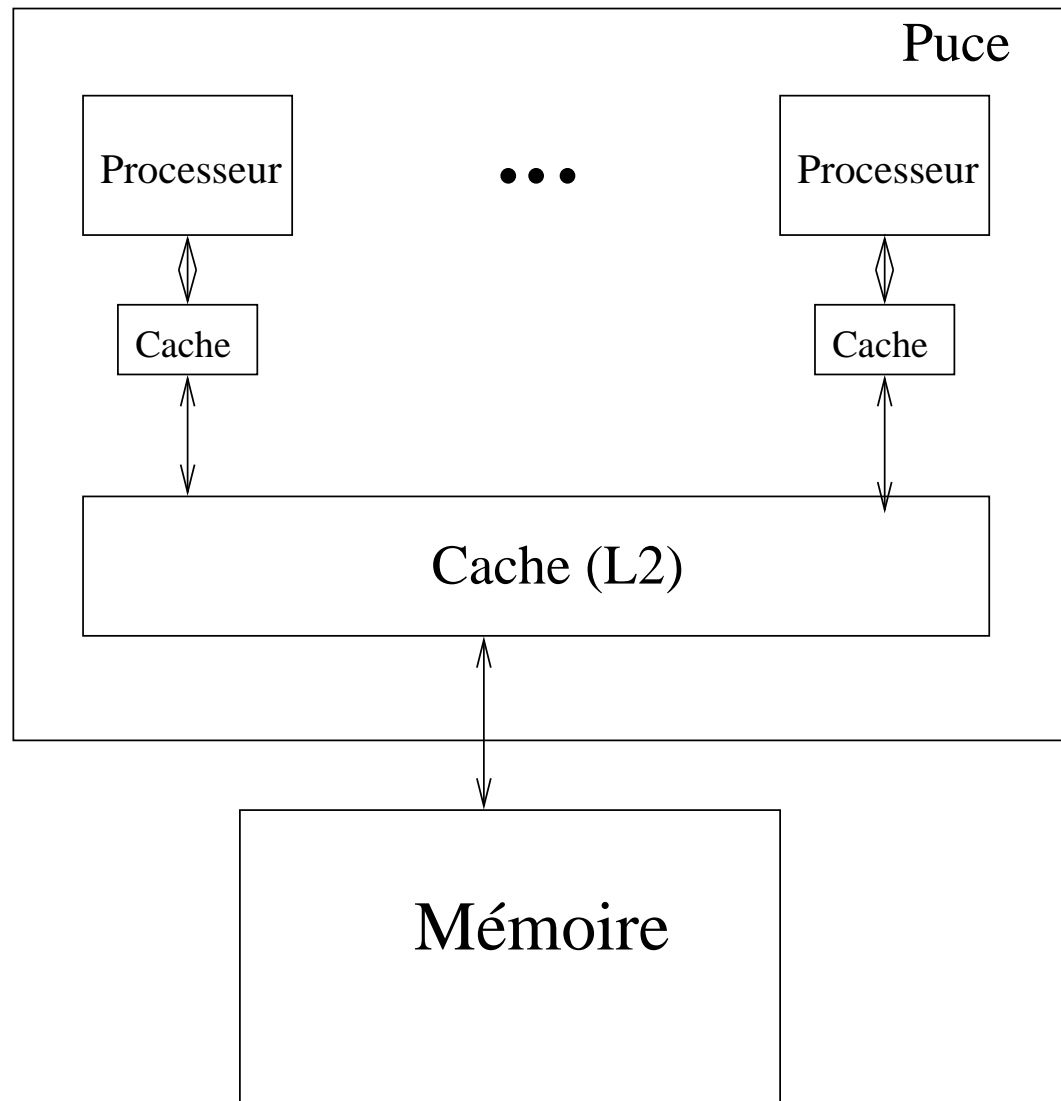
- + **Plusieurs processeurs sur une même puce**



Source : www.techspot.com/articles-info/23/images/img2.jpg

Processeurs *multicore* = *Chip multiprocessors* (CMP)

- Plusieurs processeurs sur une même puce, chacun avec son cache L1
- Les processeurs peuvent, ou non, partager le cache L2

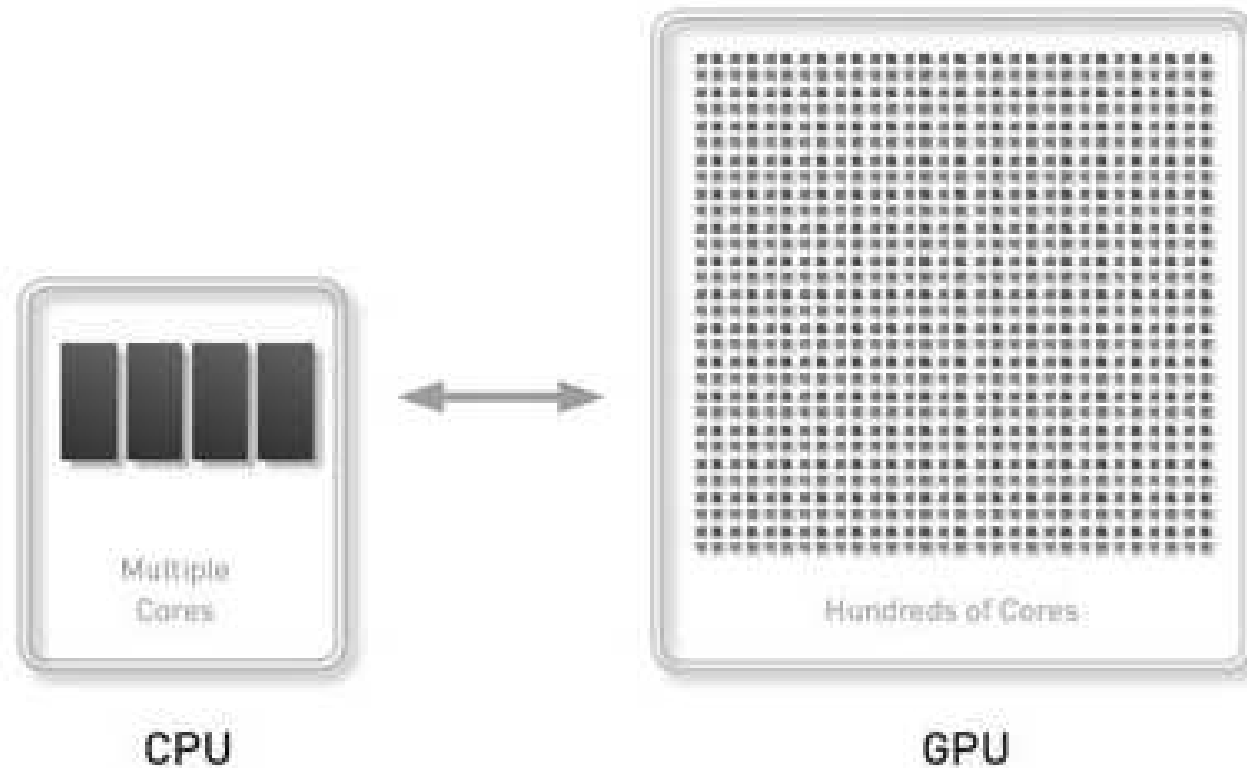


Avantages/désavantages :

- + Réduction relative de la vitesse d'horloge et, donc, de la consommation d'énergie
(énergie $\propto V^2$)
- + Support direct du parallélisme style “serveurs multi-tâches”
(requêtes indépendantes)
 - \Rightarrow Amélioration facile du *throughput*
(nombre moyen de tâches exécutées par unité de temps)
- Difficile d'accélérer une application donnée, à moins qu'elle n'ait été conçue spécifiquement pour cela
 - \Rightarrow Le programme doit être conçu et écrit à l'aide d'un modèle de programmation parallèle

4.7 Processeurs graphiques

- CPU *multicore* \Rightarrow **quelques** coeurs
- GPU (*Graphics Processing Unit*) = *manycore* \Rightarrow **beaucoup de coeurs**

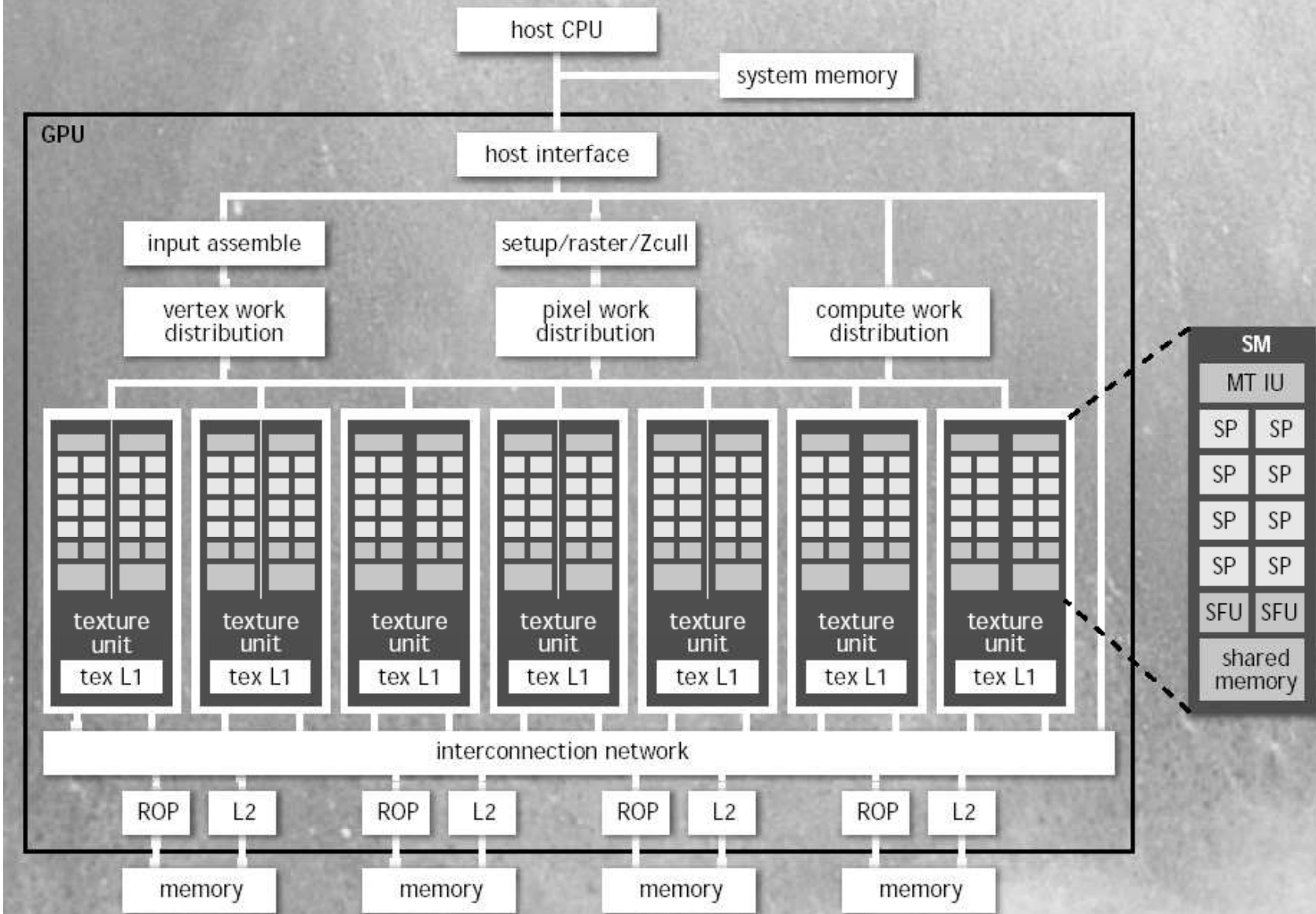


Source : www.nvidia.com/object/GPU_Computing.html

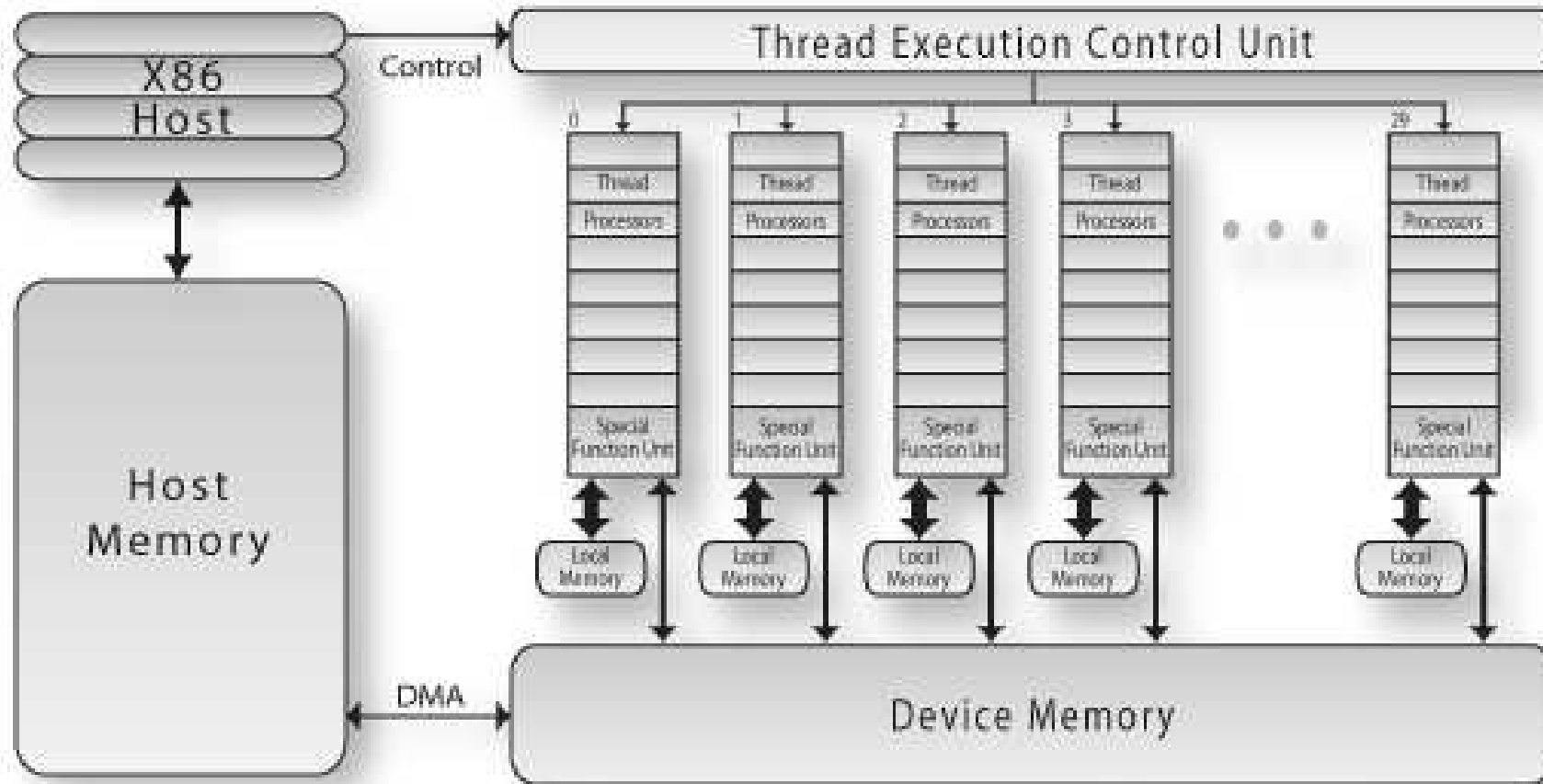
G80 \approx 12 000 threads / chip ; GT200 \approx 30 000 threads / chip

NVIDIA Tesla GPU (queue.acm.org/detail.cfm?id=1365500)

NVIDIA Tesla GPU with 112 Streaming Processor Cores



Particularité de certains GPUs = Retour du style SIMD :
SIMT = Single Instruction Multiple Threads



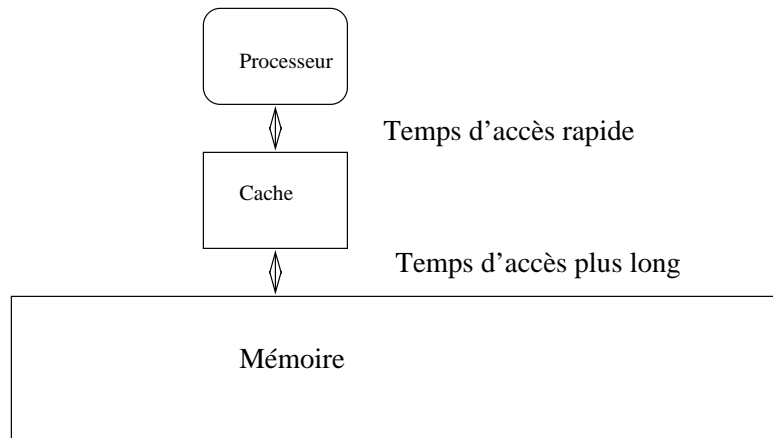
NVIDIA Tesla Block Diagram

Source : www.pgroup.com/lit/articles/insider/v2n1a5.htm

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

⇒ temps accès \approx temps cycle de machine

Adresse désirée pas dans le cache (faute de cache)

⇒ temps accès \approx plusieurs cycles machine

⇒ **gel** du pipeline du processeur (*pipeline stall*) ⇒ très coûteux

5.2 Caches multiple et protocoles de cohérence des caches

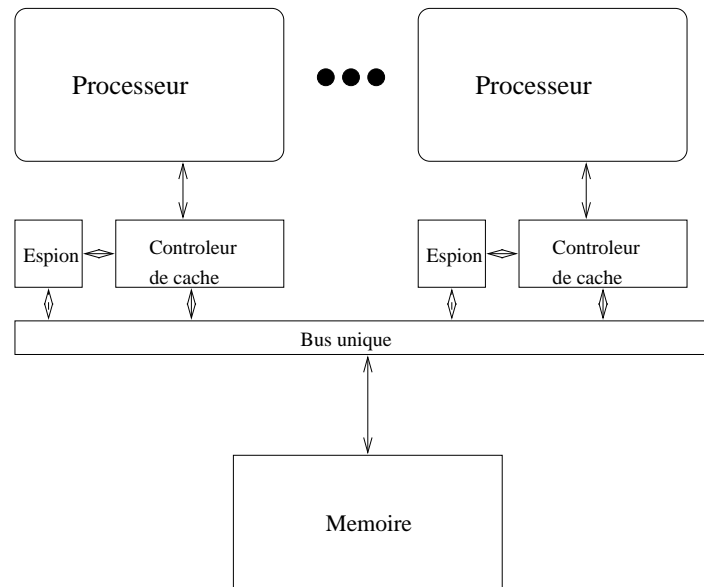
Multi-processeurs/ordinateurs \Rightarrow plusieurs caches \Rightarrow plusieurs copies

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

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

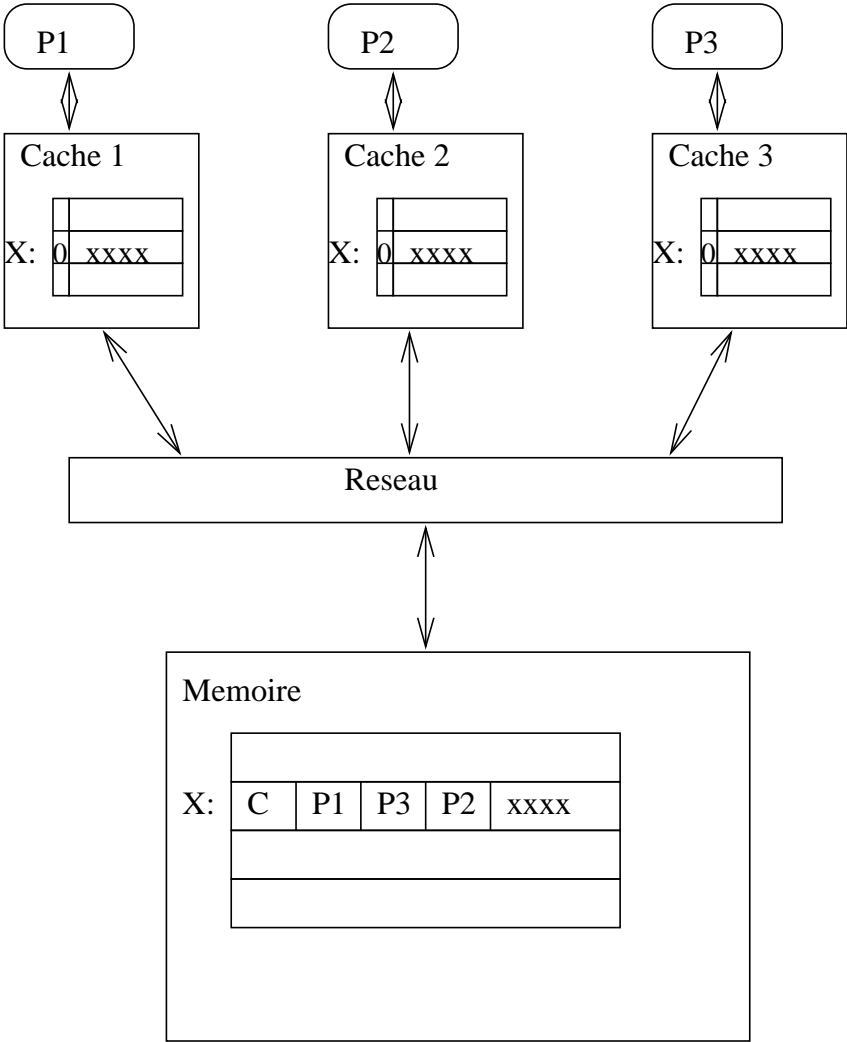
5.4 Protocole pour réseau : Utilisation d'un répertoire

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

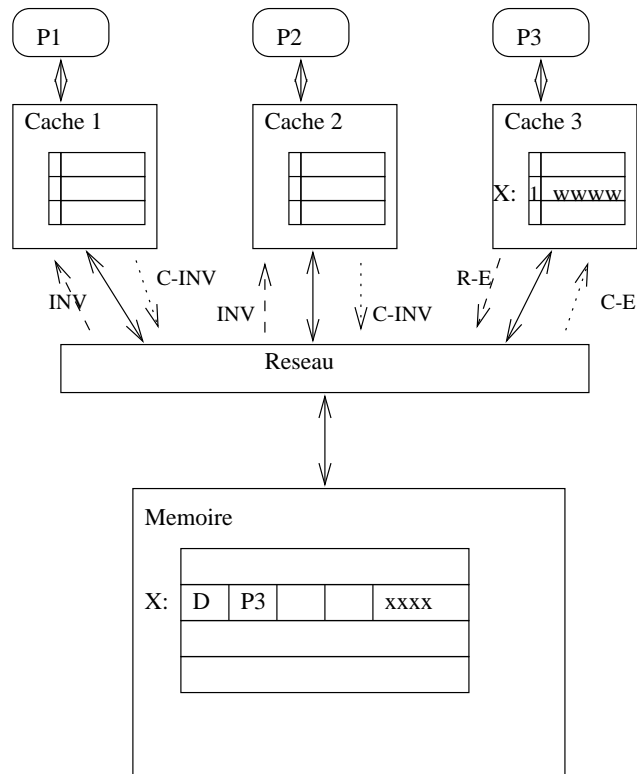
Solution alternative : utilisation d'un répertoire (*directory scheme*)

- 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

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



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



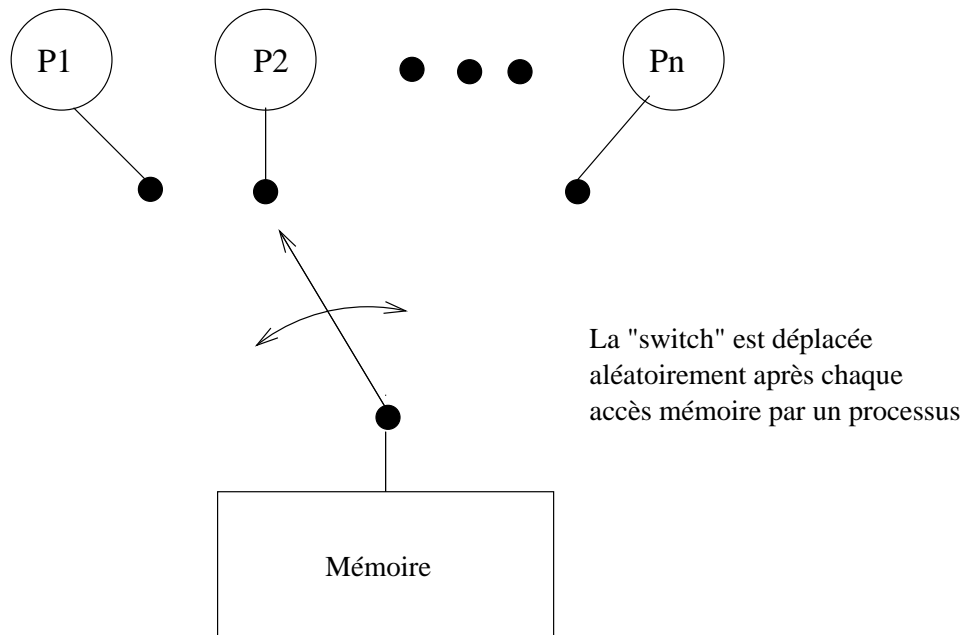
- (1) R-E = Requete Ecriture (acces Exclusif) (=> P3 bloque (stalled))
- (2) INV = INValidation
- (3) C-INV = Confirmation INValidation
- (4) C-E = Confirmation Ecrire

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 reçoivent 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.

6 Modèles de consistance mémoire

Modèle de consistance 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 consistance séquentielle



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

Exemple

P1

$$I_1 : b = 0$$

$$I_2 : a = 1$$

P2

$$I_3 : a = 0$$

$$I_4 : b = 1$$

Résultats possibles selon le modèle de consistance mémoire séquentielle :

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

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

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

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

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

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

Résultat impossible selon consistance séquentielle : $a = 0$ et $b = 0$

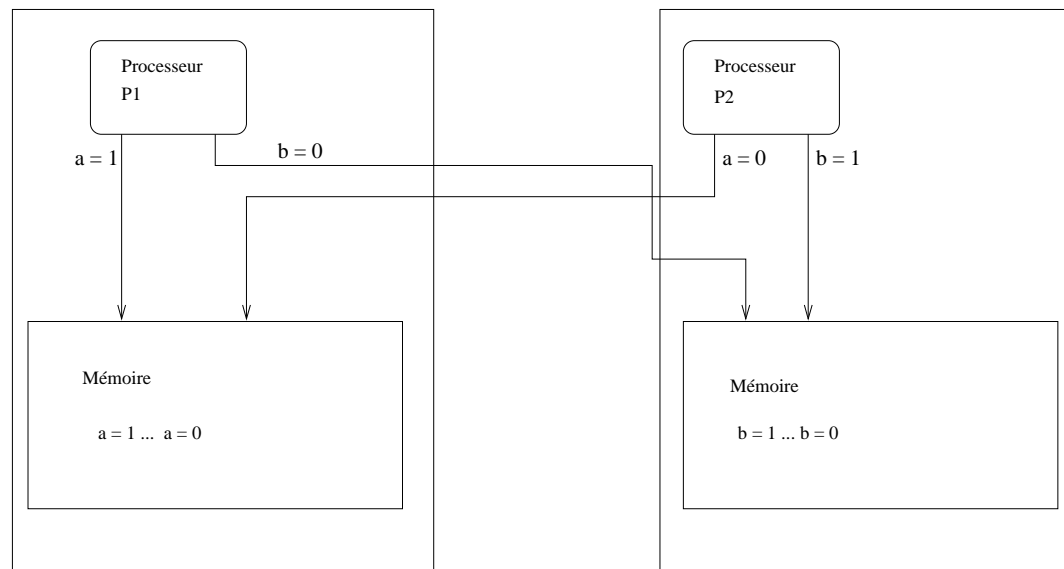
Mais ...

Supposons que

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

⇒ Temps pour que P1 accède à b est beaucoup plus long que le temps pour accéder à a

⇒ 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 $b = 0$!

Mise en oeuvre et coûts de consistance 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

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

⇒ 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 trafic sur le réseau (trafic de confirmations)

6.2 Modèles plus faibles de consistance mémoire

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

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

= Consistance séquentielle uniquement pour les instructions spéciales

⇒ Instructions ordinaires d'accès peuvent s'exécuter rapidement
(sans tenir compte de la consistance)

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

Exemple = *weak consistency*

P1

```
test&set(lock);  
mailbox = message1;  
mailbox_full = TRUE;  
unset(lock);
```

P2

```
while (!mailbox_full)  
    {}  
test&set(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!)

Exemple : *release consistency*

Encore moins contraignant que *weak consistency* :

P1

```
acquire(lock);  
mailbox = message1;  
mailbox_full = TRUE;  
release(lock);
```

P2

```
while (!mailbox_full)  
    {}  
acquire(lock);  
m = mailbox;  
release(lock);
```

Traitement des instructions de synchronisation

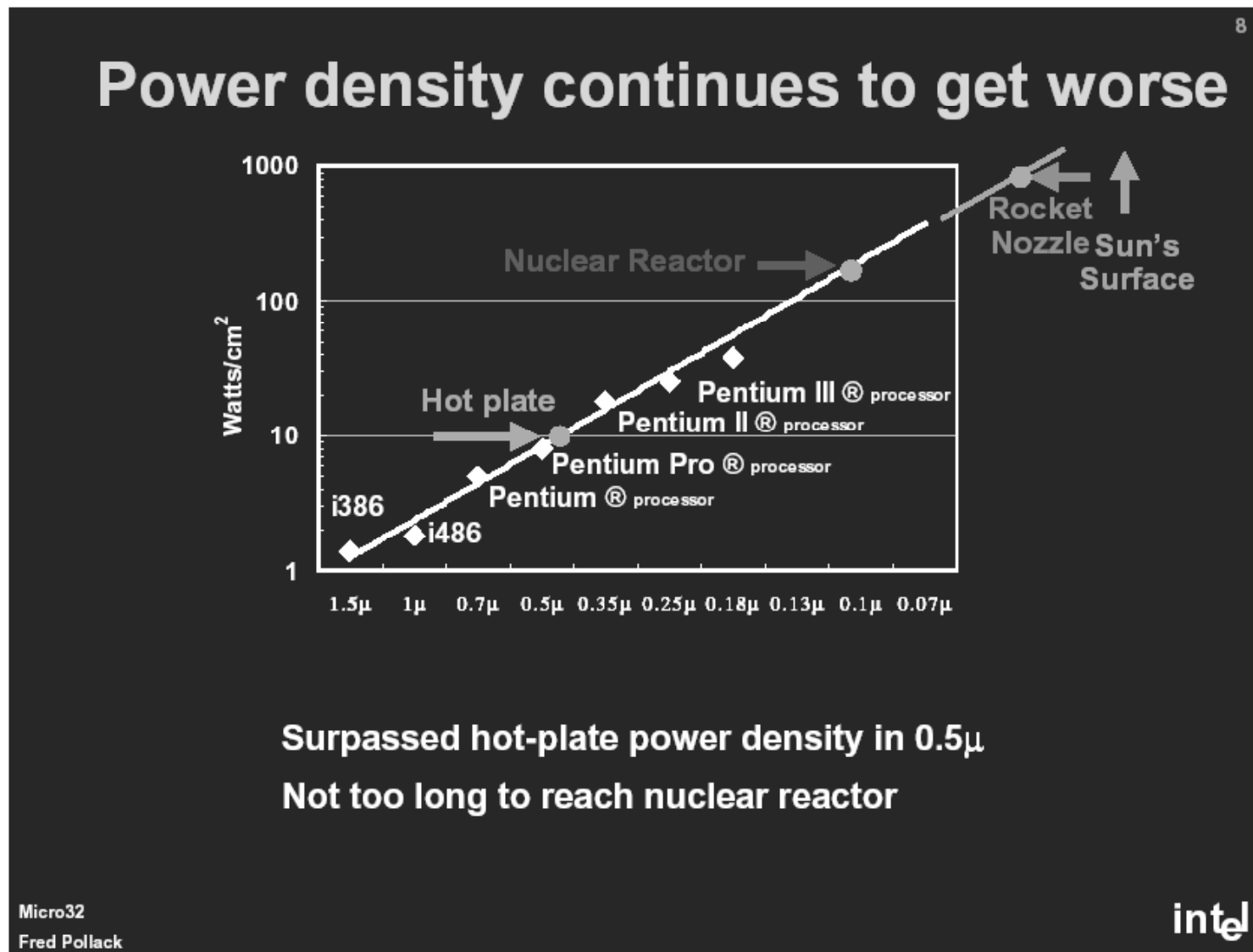
- **acquire** : Retarde l'exécution des accès mémoire ordinaires jusqu'à ce que le *acquire* soit globalement complété
- **release** : Ne s'exécute que lorsque tous les accès précédents ont globalement complété

7 Conclusion : Les problèmes des superordinateurs modernes — ou pourquoi la programmation parallèle est difficile ☹

7.1 Niveau matériel

- Depuis 1992, les performances des ordinateurs haute performance ont augmenté de 10 000 fois, mais...
 - les performances par watt ont augmenté de 300 fois
 - les performances par pied carré ont augmenté de 65 fois
- Faits :
 - Les circuits deviennent si petits que les particules α peuvent générer des problèmes (comme dans l'espace)
 - Une augmentation *linéaire* de la vitesse d'horloge requiert une augmentation cubique de la consommation d'énergie

Tendance prédite par Pollack (Intel) en 1999 :



Source : hpc.ac.upc.edu/Talks/dir07/T000065/slides.pdf

- Les superordinateurs les plus puissants (www.top500.org) utilisent jusqu'à 10 MW \approx ville de 40 000 habitants!

Exemple : IBM Blue Gene/L, la machine la plus rapide en 2007, conçue pour minimiser l'énergie (sic), avec 106 496 processeurs

⇒ Systèmes complexes de refroidissement

⇒ Coûts d'utilisation plus grands que les coûts d'acquisition

- Bref, selon D. Patterson, la “vision traditionnelle” doit changer :
 - **Avant** : *Power free, Transistors expensive*
 - **Maintenant** : *Power expensive, Transistors free*
«Can put more on chip than can afford to turn on»

Première position de www.top500.org — décembre 2010 :

[Home](#) ▶ [Sites](#) ▶ [National Supercomputing Center in Tianjin](#)

Tianhe-1A

Details

Performance/Linpack Data

Ranking History

System Name	Tianhe-1A
Site	National Supercomputing Center in Tianjin
System Family	NUDT MPP
System Model	NUDT YH MPP
Computer	NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C
Vendor	NUDT
Application area	Research
Main Memory	229376 GB
Installation Year	2010
Operating System	Linux
Memory	229376 GB
Interconnect	Proprietary
Processor	Intel EM64T Xeon X56xx (Westmere-EP) 2930 MHz (11.72 GFlops)

Nombre total de coeurs = 186 368

Première position de www.top500.org — juin 2015 :

TIANHE-2 (MILKYWAY-2) - TH-IVB-FEP CLUSTER, INTEL XEON E5-2692
12C 2.200GHZ, TH EXPRESS-2, INTEL XEON PHI 31S1P

Site:	National Super Computer Center in Guangzhou
Manufacturer:	NUDT
Cores:	3,120,000
Linpack Performance (Rmax)	33,862.7 TFlop/s
Theoretical Peak (Rpeak)	54,902.4 TFlop/s
Nmax	9,960,000
Power:	17,808.00 kW
Memory:	1,024,000 GB
Processor:	Intel Xeon E5-2692v2 12C 2.2GHz
Interconnect:	TH Express-2
Operating System:	Kylin Linux
Compiler:	icc
Math Library:	Intel MKL-11.0.0
MPI:	MPICH2 with a customized GLEX channel

Nombre total de coeurs = 3 120 000

Trois premières positions de www.top500.org — novembre 2015 :

RANK	SITE	SYSTEM	CORES	RMAX (TFLOP/S)	RPEAK (TFLOP/S)	POWER (KW)
1	National Super Computer Center in Guangzhou (/site/50365) China	Tianhe-2 (MilkyWay-2) - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P (/system/177999) NUDT	3,120,000	33,862.7	54,902.4	17,808
2	DOE/SC/Oak Ridge National Laboratory (/site/48553) United States	Titan - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x (/system/177975) Cray Inc.	560,640	17,590.0	27,112.5	8,209
3	DOE/NNSA/LLNL (/site/49763) United States	Sequoia - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom (/system/177556)	1,572,864	17,173.2	20,132.7	7,890

Première position de www.top500.org — novembre 2016 et juin 2017 :

Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway

Site:	National Supercomputing Center in Wuxi
Manufacturer:	NRCPC
Cores:	10,649,600
Linpack Performance (Rmax)	93,014.6 TFlop/s
Theoretical Peak (Rpeak)	125,436 TFlop/s
Nmax	12,288,000
Power:	15,371.00 kW (Submitted)
Memory:	1,310,720 GB
Processor:	Sunway SW26010 260C 1.45GHz
Interconnect:	Sunway
Operating System:	Sunway RaiseOS 2.0.5

Quelques centrales d'Hydro-Québec (wikipedia) :

Centrale hydroélectrique ²	Type	Puissance (MW)	Groupes	Hauteur de chute	Mise en service	Réservoir
Beauharnois	Fil de l'eau	1 903	38	24,39	1932-1961	Aucun
Beaumont	Fil de l'eau	270	6	37,8	1958	Aucun
Bersimis-1	Réservoir	1 178	8	266,7	1956	Réservoir Pipmuacan
Bersimis-2	Fil de l'eau	869	5	115,83	1959	Aucun
Brisay	Réservoir	469	2	37,5	1993	Réservoir de Caniapiscau
Bryson	Fil de l'eau	56	3	18,29	1925	Aucun
Carillon	Fil de l'eau	753	14	17,99	1962	Aucun
Chelsea	Fil de l'eau	152	5	28,35	1927	Aucun
Chute-Allard	Fil de l'eau	62	6	17,83	2008	Aucun
Chute-Bell	Fil de l'eau	10	2	17,8	1915-1999	Aucun
Chute-des-Chats ^{note 1}	Fil de l'eau	92	4	16,16	1931	Aucun
Chute-Hemmings	Fil de l'eau	29	6	14,64	1925	Aucun
Drummondville	Fil de l'eau	16	4	9,1	1910	Aucun

7.2 Niveau logiciel

- Toutes les applications ne sont pas parallélisables
- Développer un programme parallèle... est souvent *très (!) difficile*

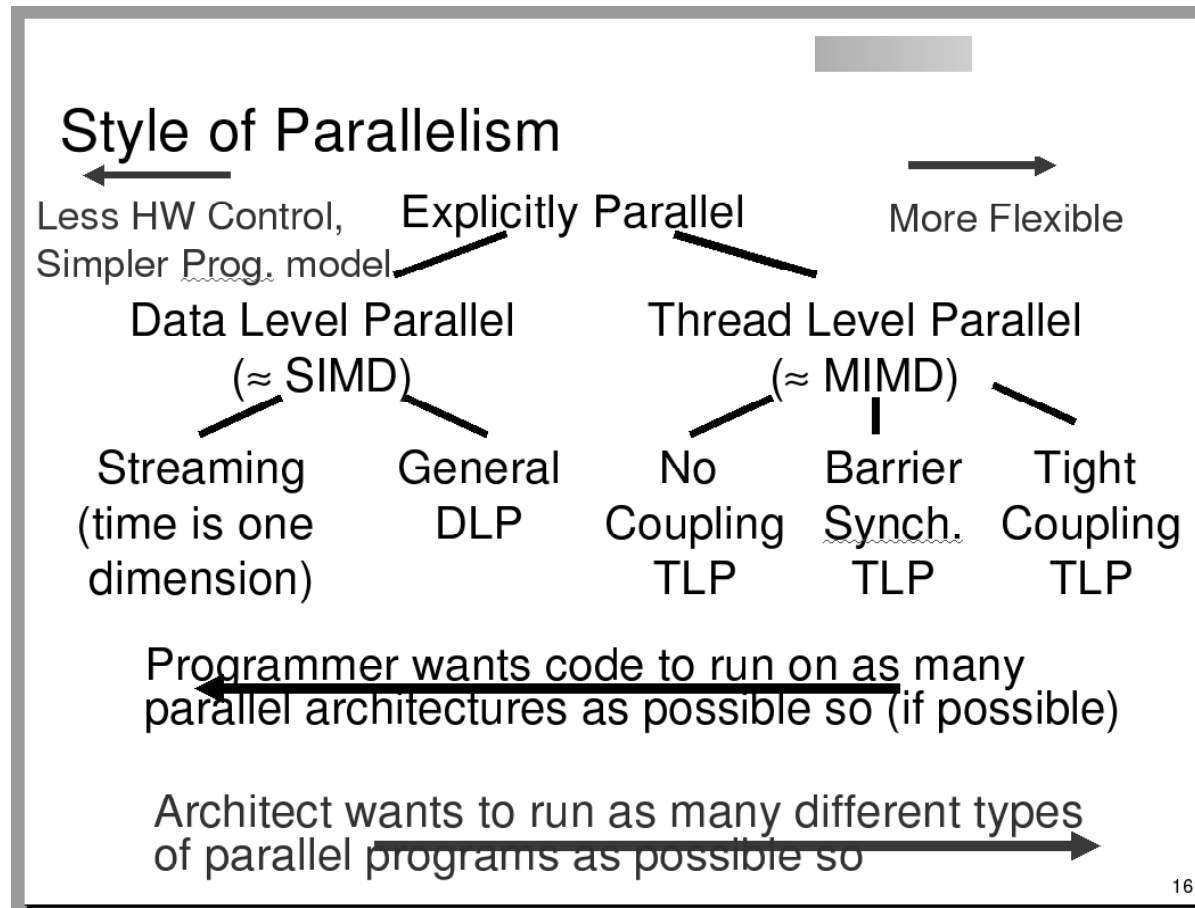
- Toutes les applications ne sont pas parallélisables + Loi d'Amdahl ☹️



Source : pragprog.com/magazines/2009-09/images/iStock_000005736241Small.jpg

- Développer un programme parallèle... est souvent *très (!) difficile*
 - Plusieurs facteurs à prendre en compte, plusieurs dimensions de conception indépendantes
 - Les types de données disponibles dans certains langages sont parfois limités
 - Certains langages sont intimement liés à un type de machine
 - Peu d'outils (débogage, profilage)
 - Pour de nombreux problèmes, on ne sait pas encore quelles sont les bonnes approches

- Plusieurs modèles de programmation
 - Modèles souvent liés à des architectures particulières, générant des programmes qui dépendent du nombre de processeurs
 - ⇒ Peut bien fonctionner sur une machine, mais pas sur une autre



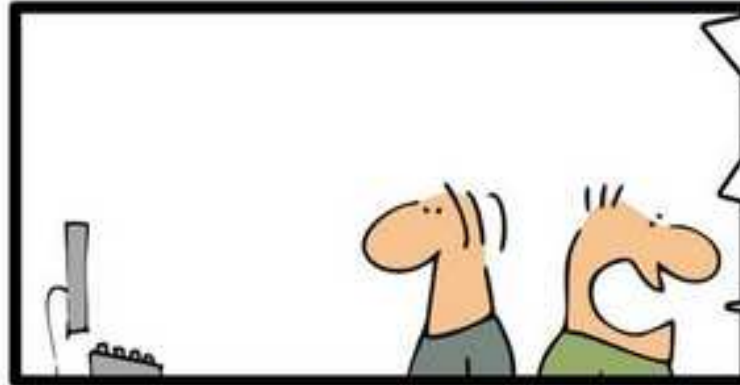
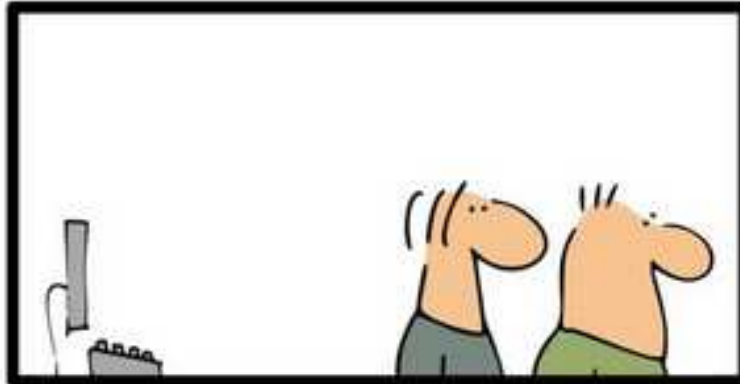
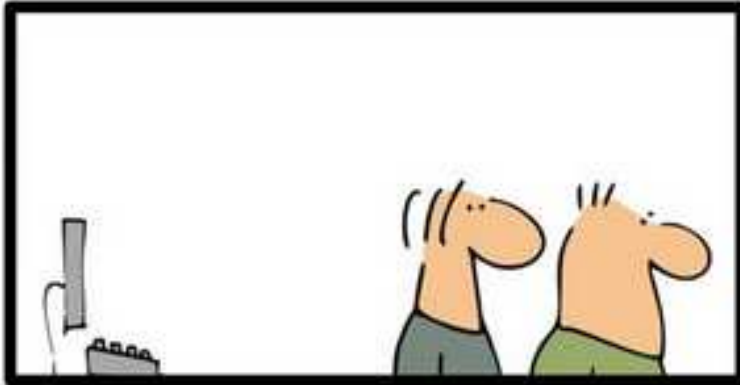
- Certains considèrent que le passage à la programmation parallèle sera encore plus difficile que le passage de la programmation structurée à la programmation orientée objets 😞
-

Question : Combien d'années se sont écoulées entre le premier langage avec objets et le moment où la programmation orientée objets est devenue «*mainstream*»?

- Premier langage avec objets : SIMULA-67 — 1967
- Premier langage orienté objets «pur» : Smalltalk-80 — 1980
- Premier langage objet *mainstream* : Java 1.0 — 1991

THE ART OF BUGFIXING

geek & poke



DON'T LOOK
AT THE
SCREEN!!!!

HOW TO DEBUG HEISENBUGS