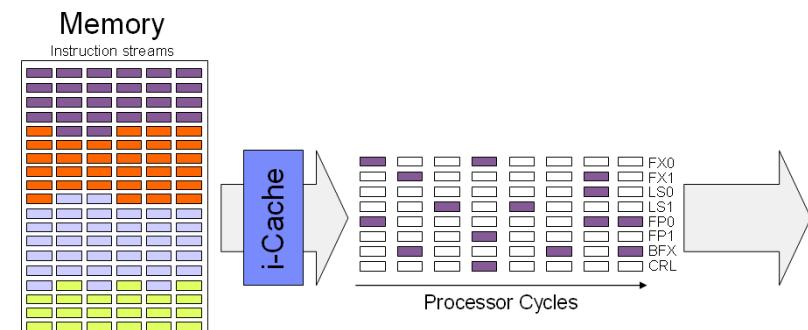


Thread Level Parallelism

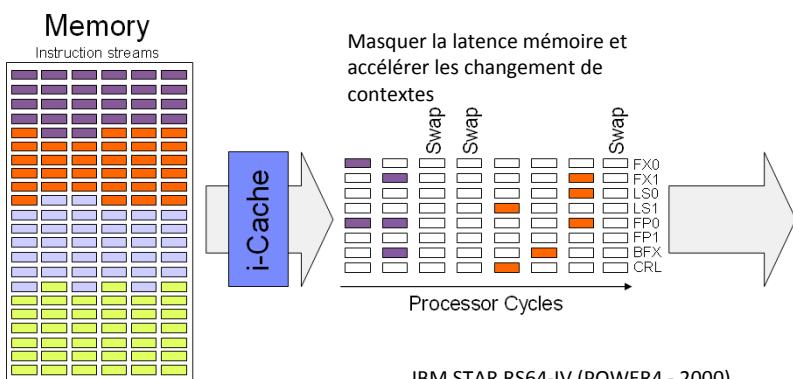
- Mur de la chaleur
 - Inutile d'augmenter la profondeur du pipeline
- Comment exploiter les progrès en finesse de gravure ?
 - Augmenter le nombre d'unités de calcul
 - Augmenter la taille du cache
 - Améliorer les prédicteurs
- OoO ne peut rien contre les défauts de cache !
 - Prefetching logiciel
 - Prefetching matériel
- Exploiter plusieurs flux d'instructions
 - Multithreading au niveau du pipeline
 - Permet de bénéficier des unités fonctionnelle
 - Processeurs Multicœurs

niveau	temps	cycles 2,5GHz	Exemple (cycles)
L1	<1ns	2-4	Nehalem 4
L2	2-5ns	8-20	Nehalem 10, Opteron 7, PIV 22, Core-2 14
L3	10ns	40	Nehalem 40
Mem	60ns	240	

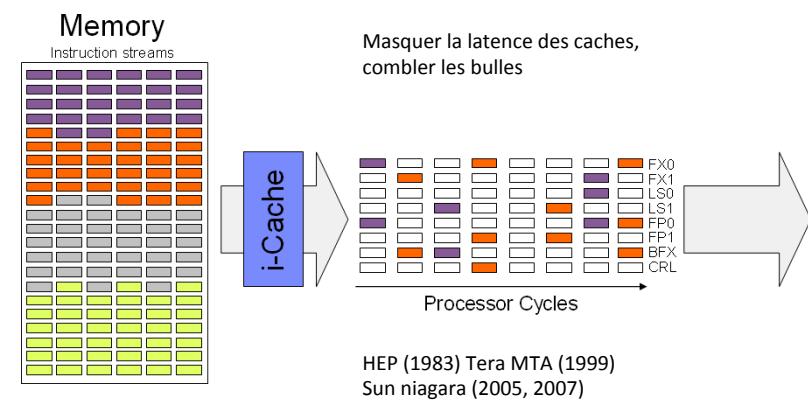
Multithreading



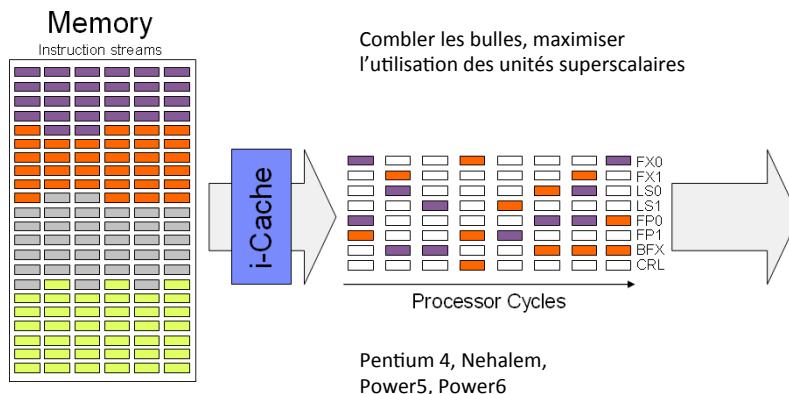
Multithreading Coarse Grained MT



Multithreading Fine Grained MT



Multithreading Simultaneous MT

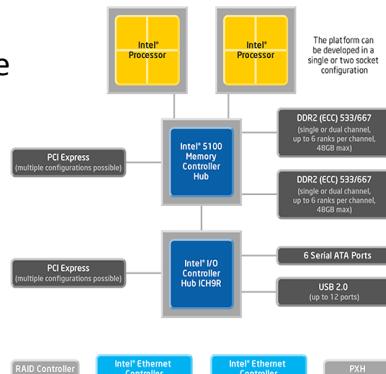


Multithreading CMT – FMT - SMP

- Avantages
 - Meilleure utilisation du pipeline
 - Partage de données entre threads
 - Thread helper (prefetching, évaluation spéculative)
 - Hétérogénéité des coeurs
 - Un paramètre en plus pour répartir la charge
- Inconvénients
 - Partage de cache source de dégradation et d'imprédictibilité des performances
 - Coloration de la mémoire
 - Gestion des priorités / de l'équité
 - Cache aware scheduling
 - Solutions matérielles
 - Consommation du processus idle
- Solutions
 - Appariement de threads compatibles
 - Au pire Désactivation (Pentium IV)

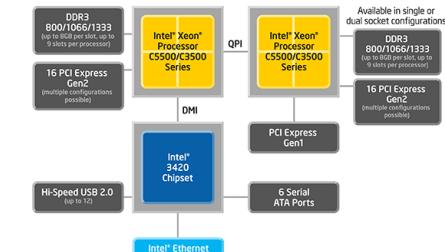
Machines multiprocesseurs à mémoire partagée

- SMP
 - Mémoire centralisée



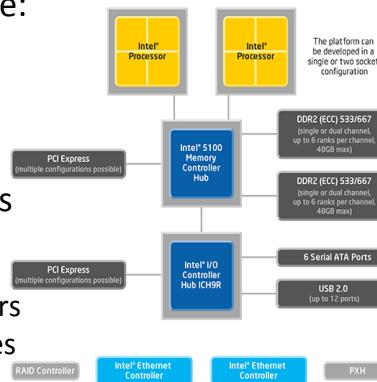
Machines multiprocesseurs à mémoire partagée

- SMP
 - Mémoire centralisée
- NUMA
 - Mémoire répartie



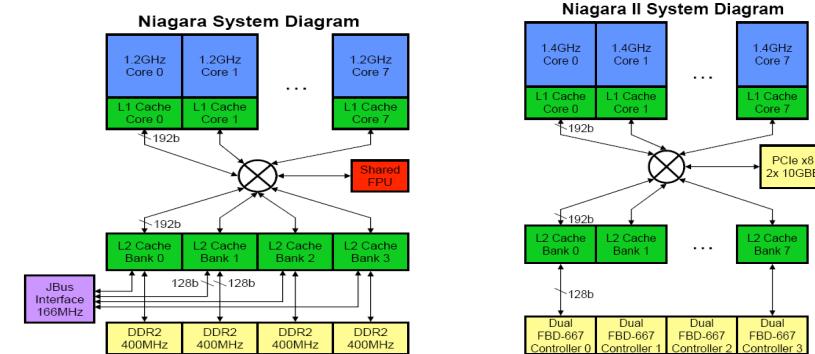
SMP

- Organisation classique:
 - Mémoire
 - Bus
 - Cache
 - Cœur
- Le bus synchronise les différents cœurs.
 - Bus unique pas plus d'une dizaine de cœurs
 - Bus en étoile (avec des filtres)



SMP

- Niagara (graup au cremi) : le cache L2 est devant la mémoire



SMP

synchronisation de la mémoire

- Une même donnée peut être à la fois :
 - En mémoire,
 - Dans un cache, dans des caches,
 - Dans un registre, dans des registres,
- Quelle est la bonne version ?
 - Faire en sorte qu'il n'y ait qu'une version
 - Seulement aux yeux du programmeur
 - Donner au programmeur la même vision de la mémoire que sur une machine monoprocesseur programmée à l'aide de threads (exécutés en séquence).

SMP

synchronisation de la mémoire

→ Consistance séquentielle (Lamport 1979), Sémantique de l'entrelacement :

« Un multiprocesseur est séquentiellement consistant si toute exécution résulte d'un entrelacement des séquences d'instructions exécutées par les processeurs et qui préserve chacune des séquences. En particulier tous les processeurs voient les écritures dans le même ordre. »

- Exemple : au départ { $x = 0, y = 0$ } on lance 2 threads :


```
T1 { x = 1; print y }
T2 { y = 1; print x }
```

On ne devrait pas voir 0 0.

SMP

synchronisation de la mémoire

→ Consistance séquentielle (Lamport 1979), Sémantique de l'entrelacement :

« Un multiprocesseur est séquentiellement consistant si toute exécution résulte d'un entrelacement des séquences d'instructions exécutées par les processeurs et qui préserve chacune des séquences. En particulier tous les processeurs voient les écritures dans le même ordre. »

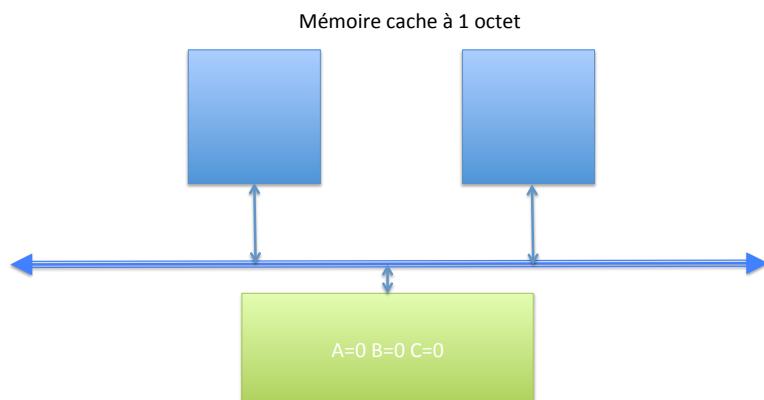
- Exemple : au départ $\{x = 0, y = 0\}$ on lance 2 threads :

```
T1 {x = 1; print y}  
T2 {y = 1; print x}
```

On ne devrait pas voir 0 0.



Implémentation de la cohérence séquentielle



SMP

synchronisation de la mémoire

→ Consistance séquentielle (Lamport 1979), Sémantique de l'entrelacement :

« Un multiprocesseur est séquentiellement consistant si toute exécution résulte d'un entrelacement des séquences d'instructions exécutées par les processeurs et qui préserve chacune des séquences. En particulier tous les processeurs voient les écritures dans le même ordre. »

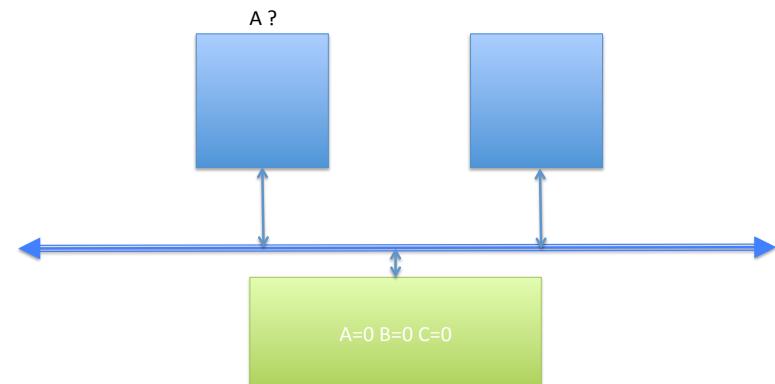
- Exemple : au départ $\{x = 0, y = 0\}$ on lance 2 threads :

```
T1 {x = 1; print y}  
T2 {y = 1; print x}
```

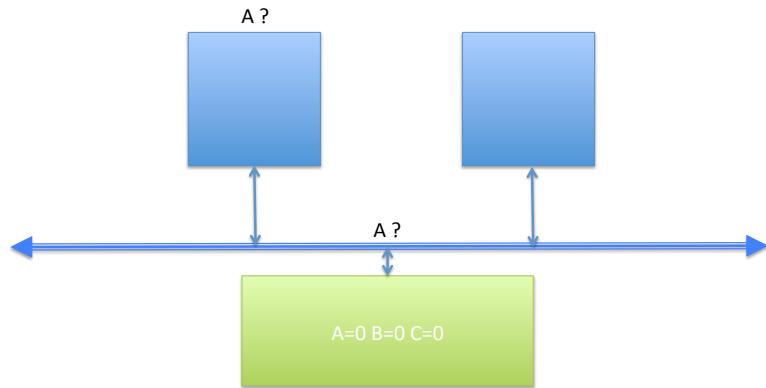
On ne devrait pas voir 0 0.



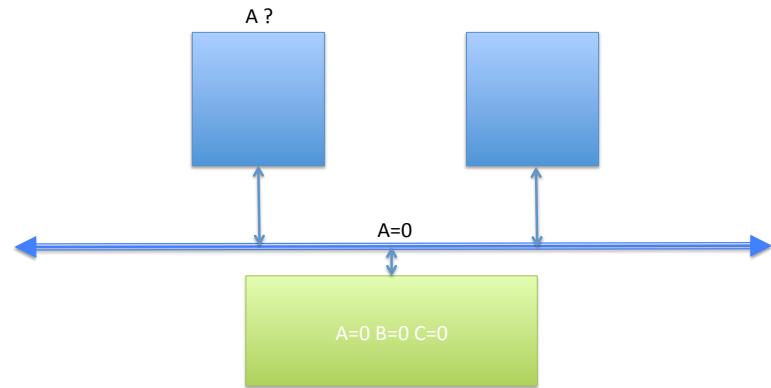
Implémentation de la cohérence séquentielle



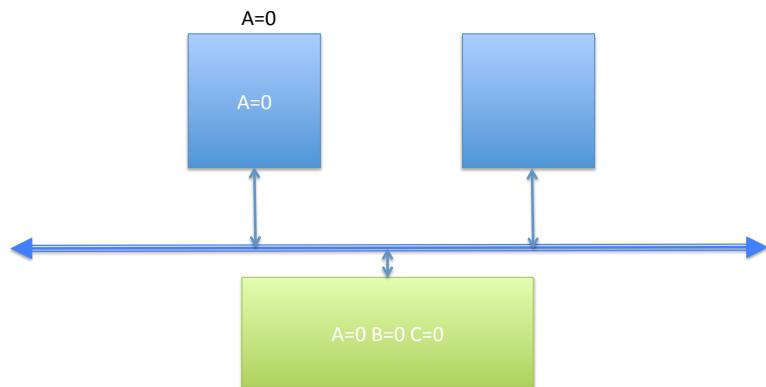
Implémentation de la cohérence séquentielle



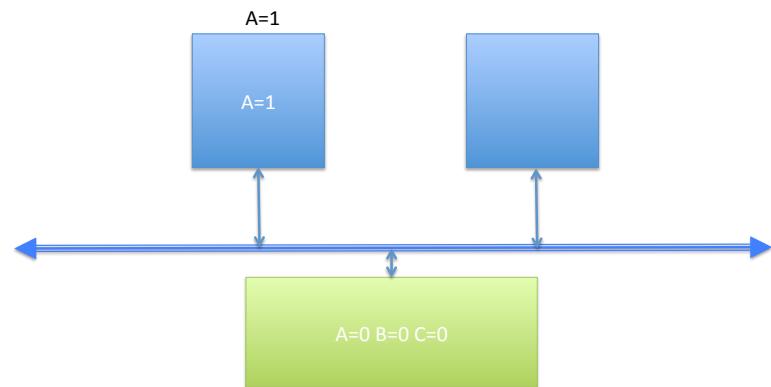
Implémentation de la cohérence séquentielle



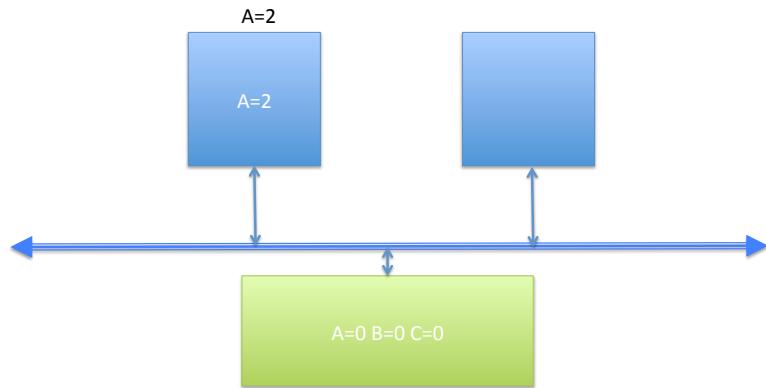
Implémentation de la cohérence séquentielle



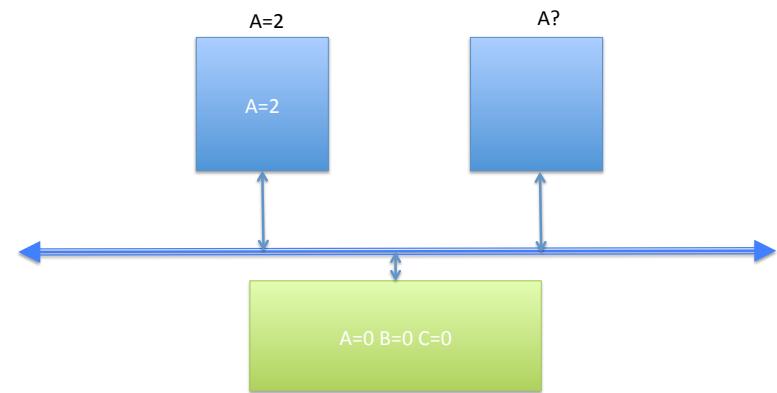
Implémentation de la cohérence séquentielle



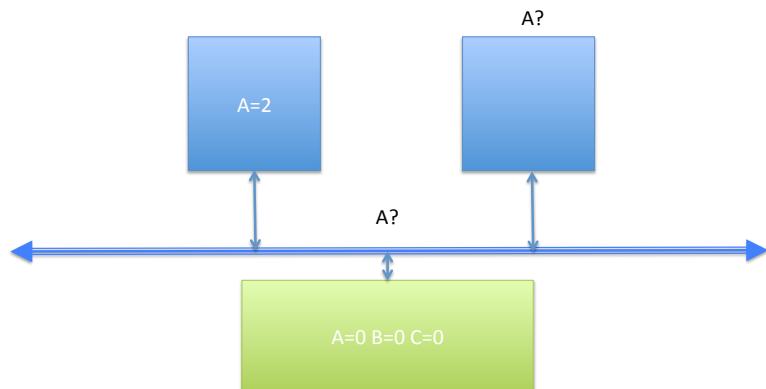
Implémentation de la cohérence séquentielle



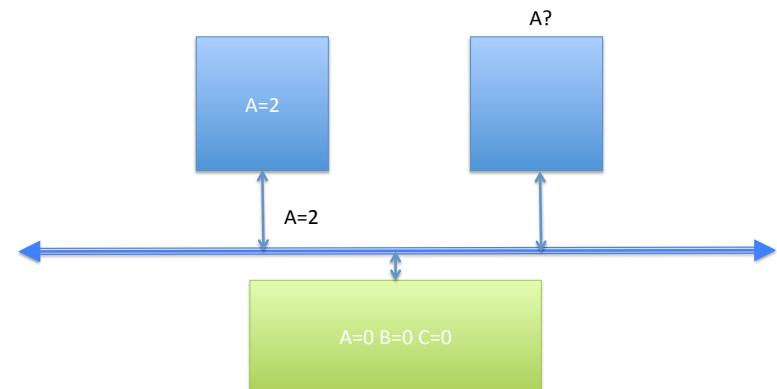
Implémentation de la cohérence séquentielle



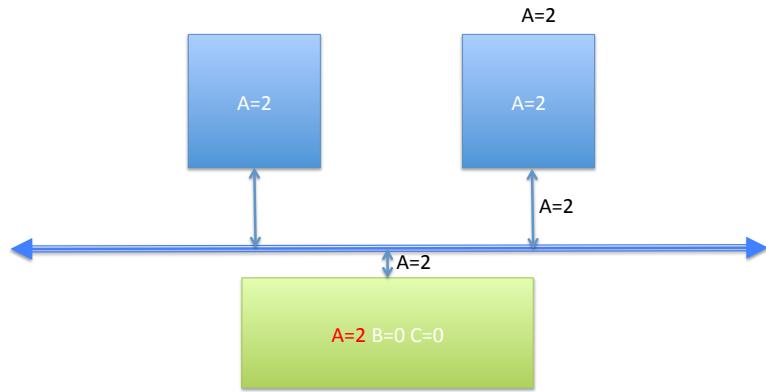
Implémentation de la cohérence séquentielle



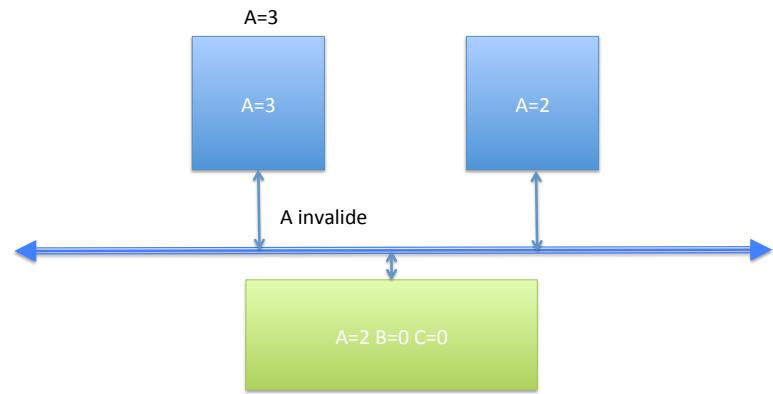
Implémentation de la cohérence séquentielle



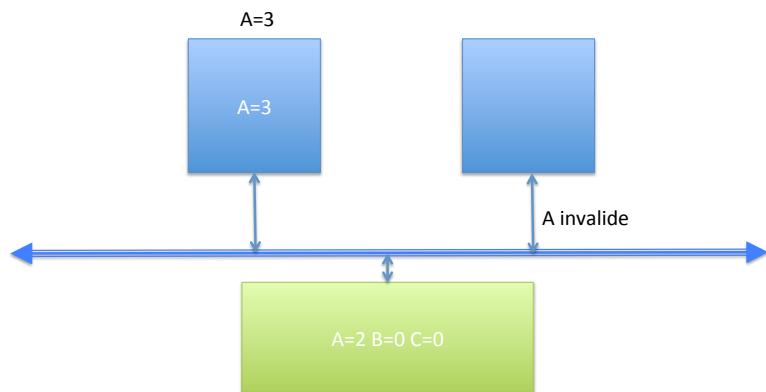
Implémentation de la cohérence séquentielle



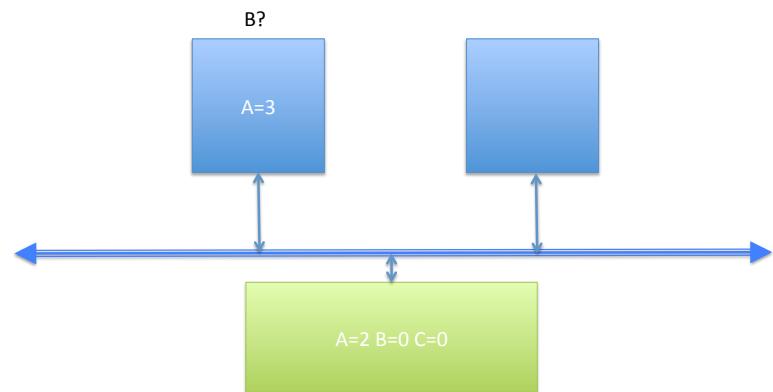
Implémentation de la cohérence séquentielle



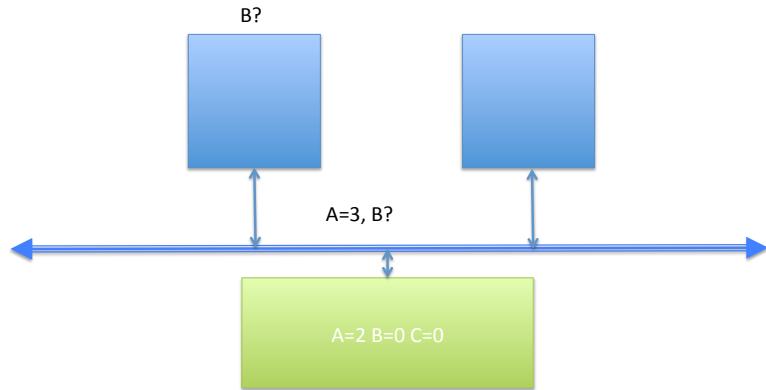
Implémentation de la cohérence séquentielle



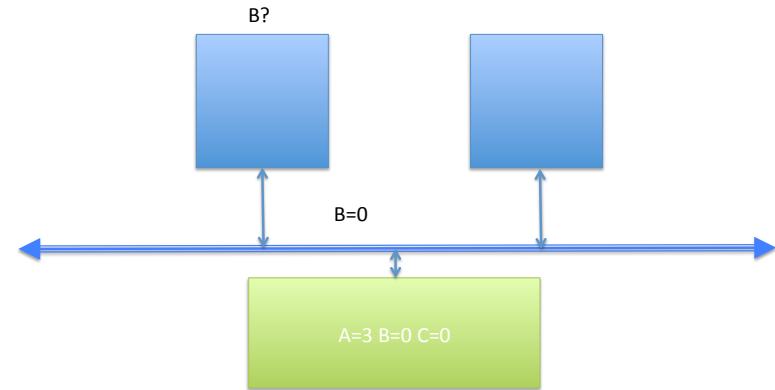
Implémentation de la cohérence séquentielle



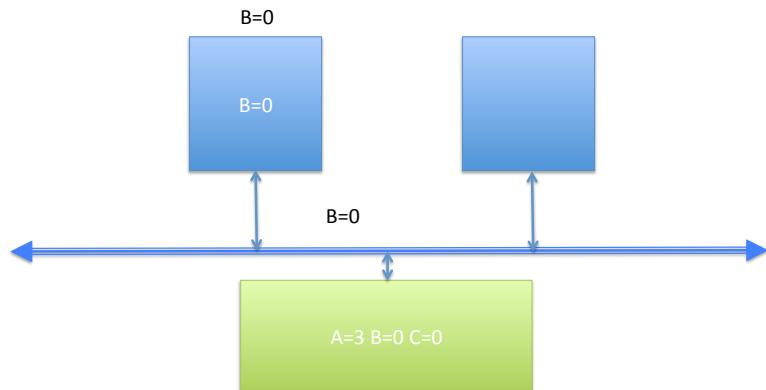
Implémentation de la cohérence séquentielle



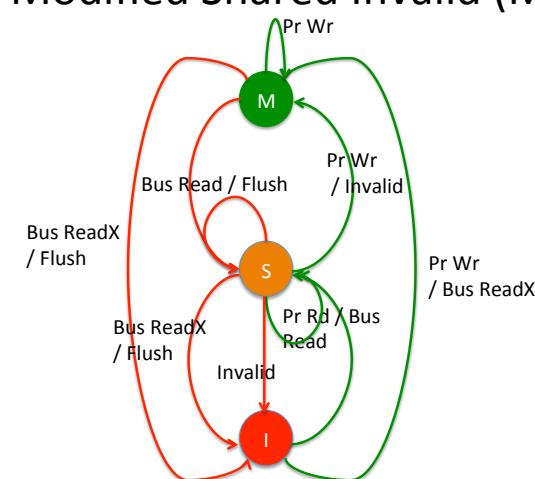
Implémentation de la cohérence séquentielle



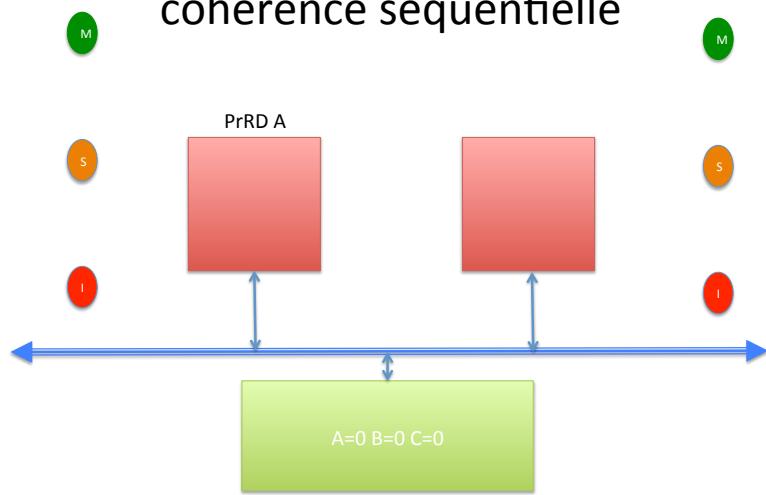
Implémentation de la cohérence séquentielle



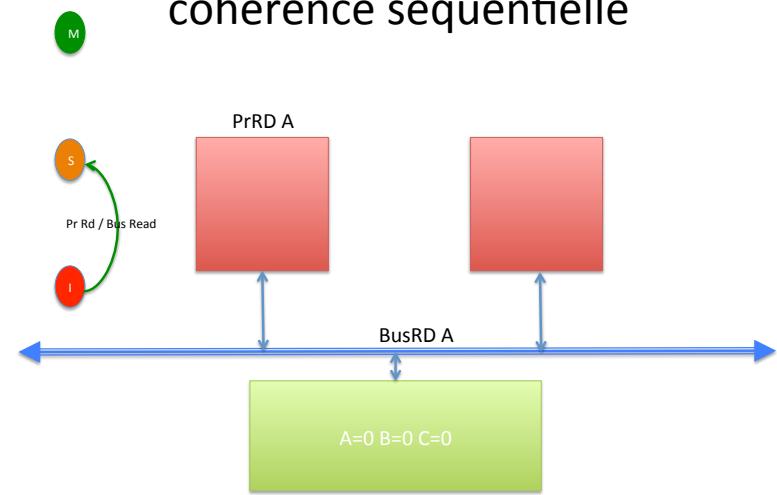
Protocole Modified Shared Invalid (MSI)



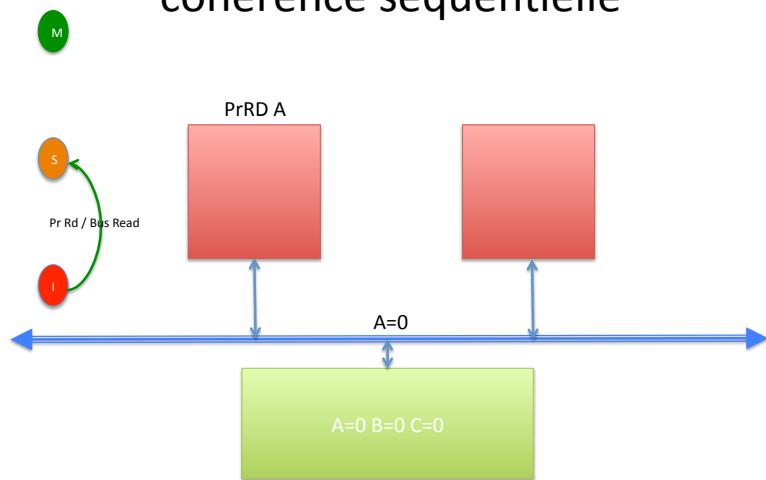
Implémentation de la cohérence séquentielle



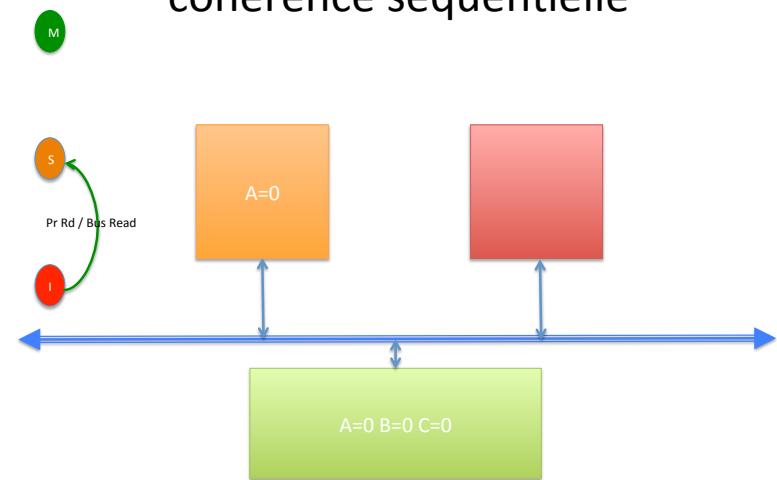
Implémentation de la cohérence séquentielle



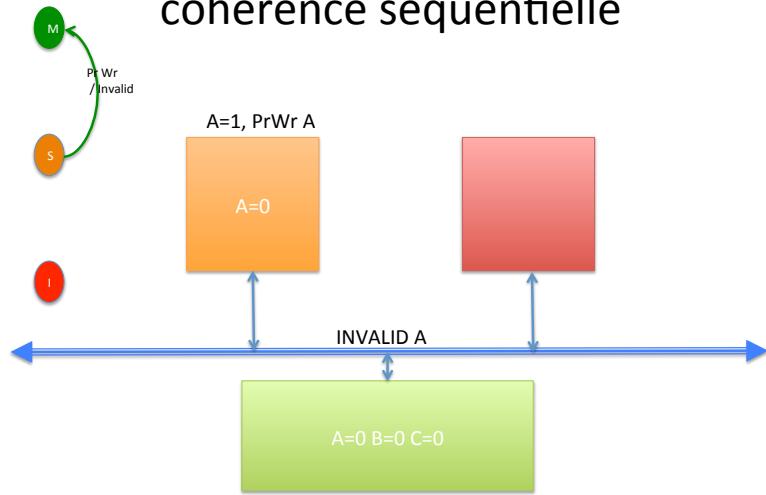
Implémentation de la cohérence séquentielle



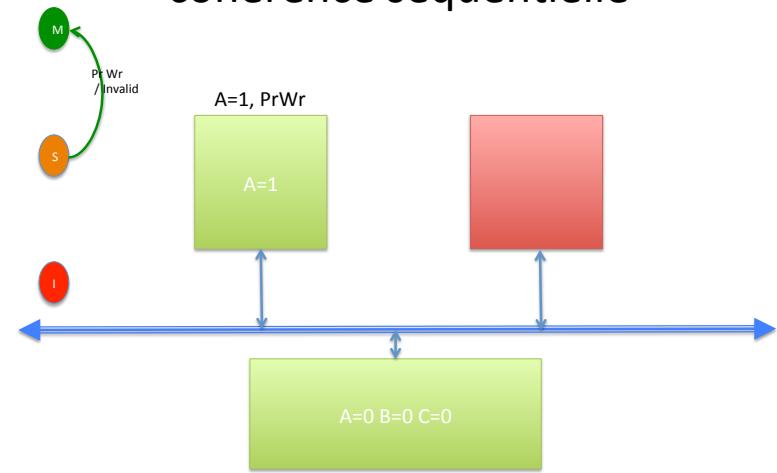
Implémentation de la cohérence séquentielle



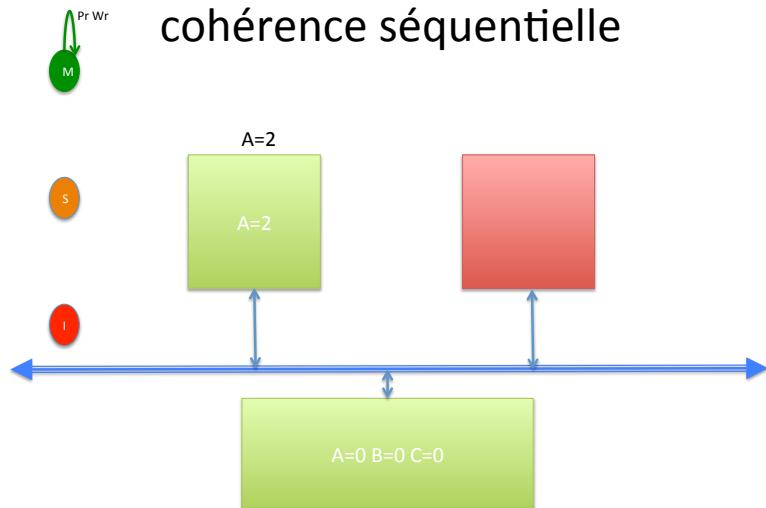
Implémentation de la cohérence séquentielle



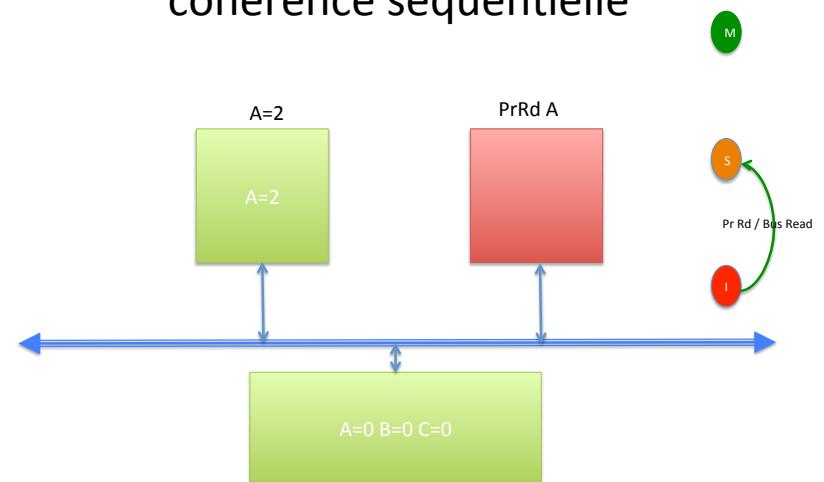
Implémentation de la cohérence séquentielle



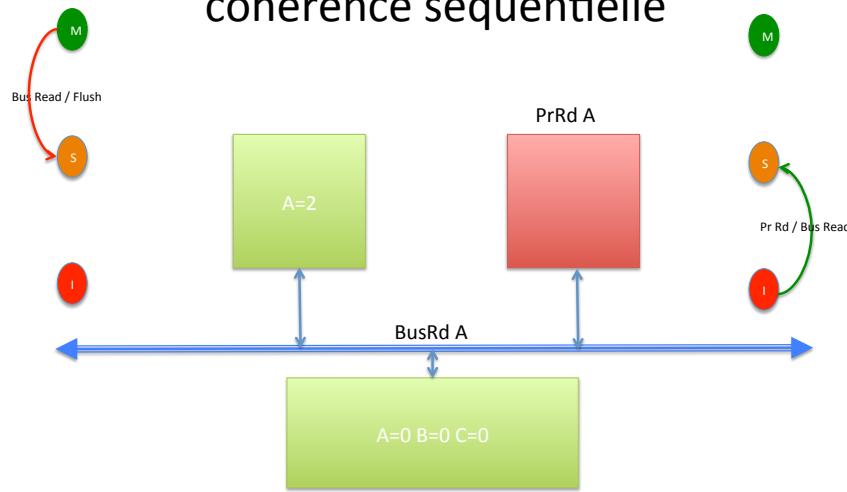
Implémentation de la cohérence séquentielle



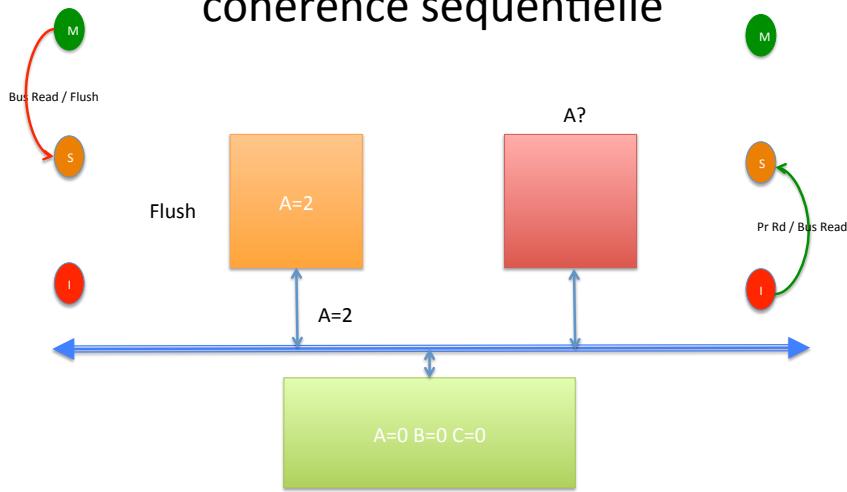
Implémentation de la cohérence séquentielle



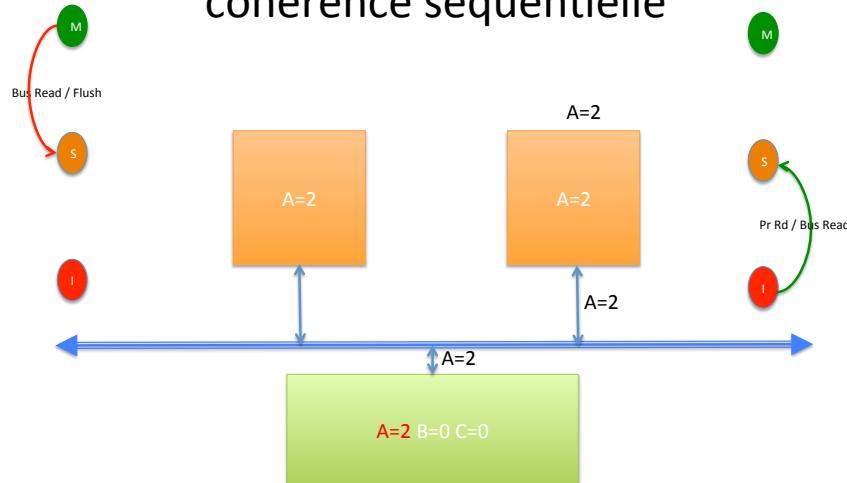
Implémentation de la cohérence séquentielle



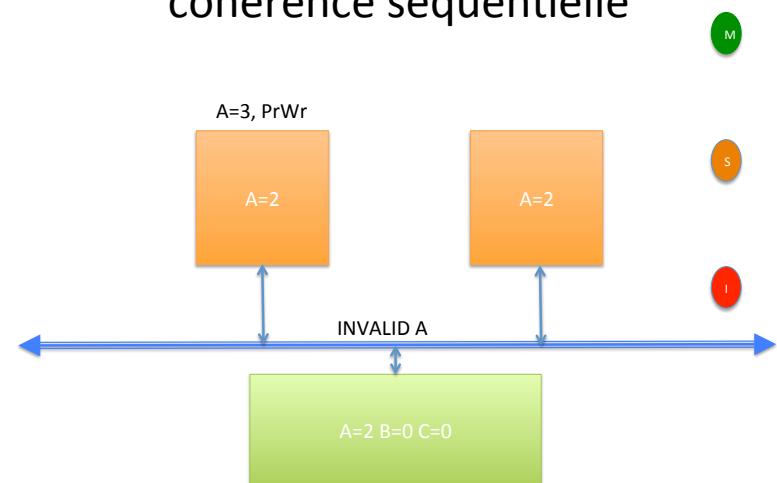
Implémentation de la cohérence séquentielle



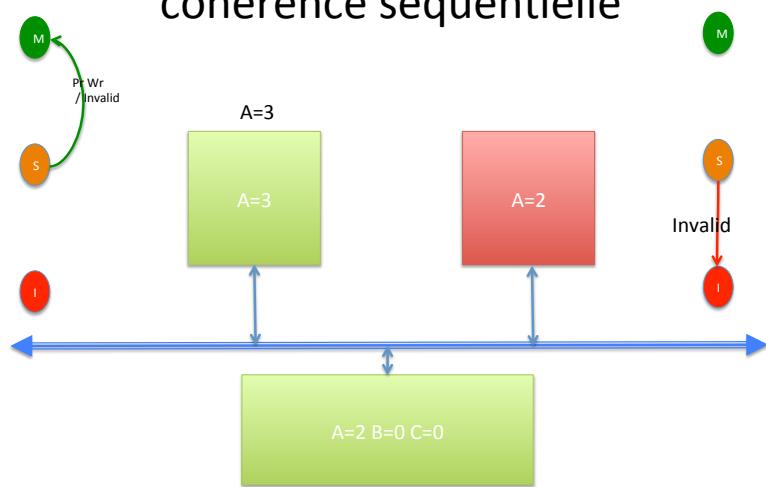
Implémentation de la cohérence séquentielle



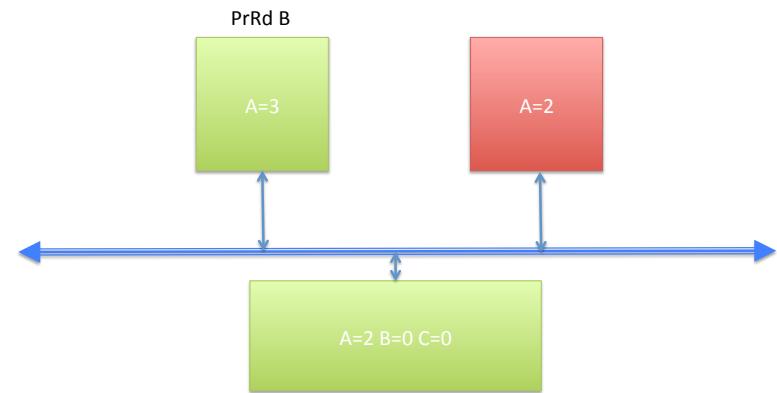
Implémentation de la cohérence séquentielle



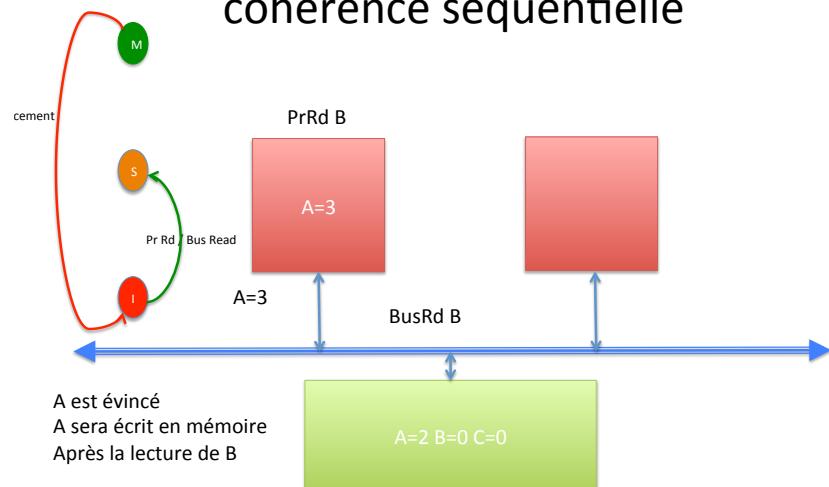
Implémentation de la cohérence séquentielle



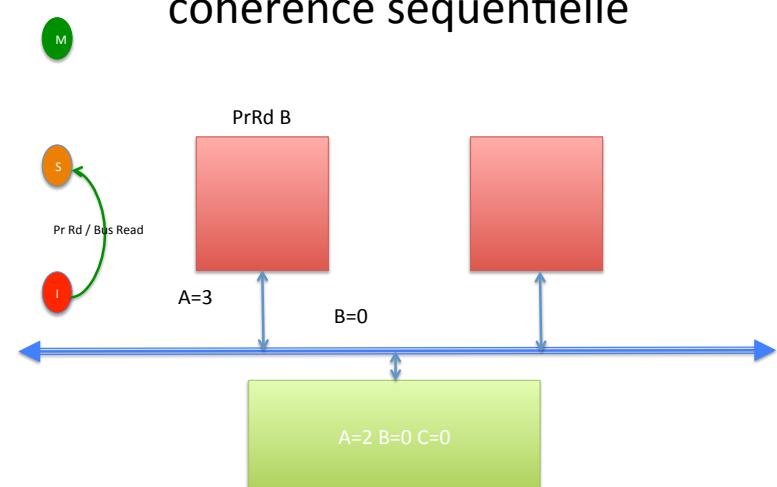
Implémentation de la cohérence séquentielle



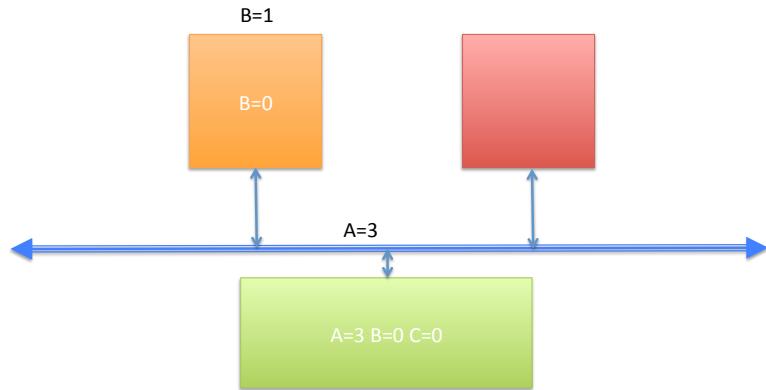
Implémentation de la cohérence séquentielle



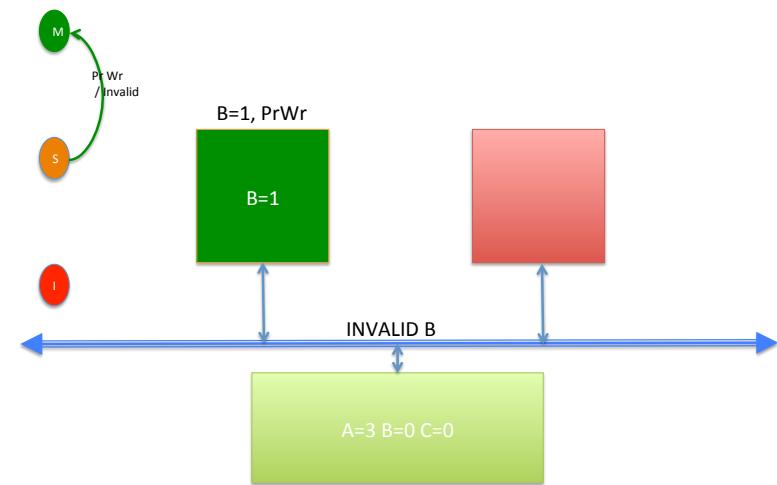
Implémentation de la cohérence séquentielle



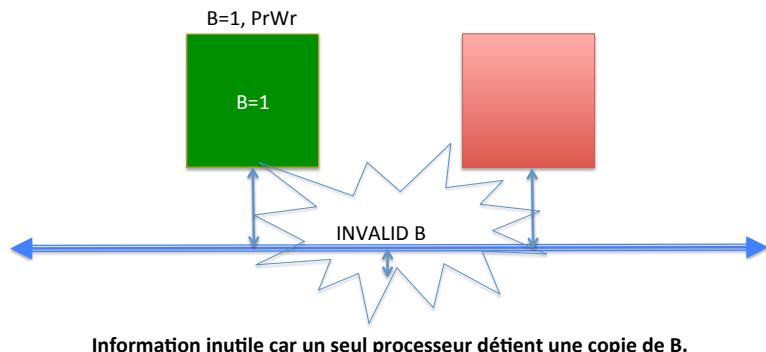
Implémentation de la cohérence séquentielle



Optimisation MSI

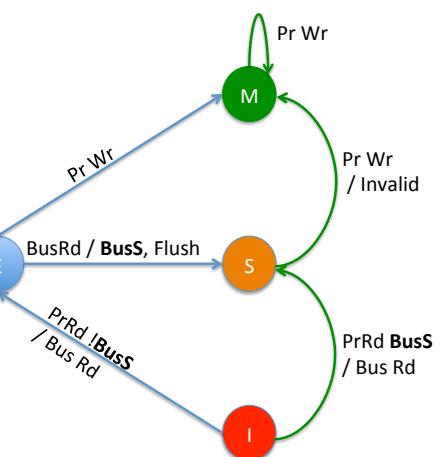


Optimisation MSI



Protocole MESI Modified Exclusive Shared Invalid

- Optimisation de MSI
 - Ajout de l'état Exclusive
 - Seul détenteur d'une copie propre
 - Permet une transition silencieuse si variable non partagée
- Nécessite de savoir si la valeur provient d'un cache ou de la mémoire
 - BusS : la variable vient d'un cache
 - !BusS : la variable vient de la mémoire



Instructions atomiques

- Protéger une opération sans prendre de mutex
 - Enchaînement « lecture opération écriture »
 - Concerne des objets de quelques octets
- Réalisations matérielles
 - Bloquer le bus
 - Passer en *modified* et conserver la donnée jusqu'à l'écriture (bloque le cache)
 - Passer en *modified* et observer si la donnée est toujours là au moment de l'écriture (ne bloque pas le cache)
- builtin gcc

```
_sync_fetch_and_add(type *ptr, type value, ...)  
type _sync_val_compare_and_swap(type *ptr, type oldval type newval, ...)
```
- ...

Les instructions atomiques en C11 <stdatomic.h>

```
_Bool atomic_compare_exchange_strong( volatile A* obj, C* expected, C desired );
_Bool atomic_compare_exchange_weak( volatile A *obj, C* expected, C desired );
_Bool atomic_compare_exchange_strong_explicit( volatile A* obj, C* expected, C desired,
                                              memory_order succ, memory_order fail );
_Bool atomic_compare_exchange_weak_explicit( volatile A *obj, C* expected, C desired,
                                            memory_order succ, memory_order fail );
```

- *obj* : pointer to the atomic object to test and modify
- *expected* : pointer to the value expected to be found in the atomic object
- *desired* : the value to store in the atomic object if it is as expected
- *succ* : the memory synchronization ordering for the read-modify-write operation if the comparison succeeds. All values are permitted.
- *fail* : the memory synchronization ordering for the load operation if the comparison fails. Cannot be memory_order_release or memory_order_acq_rel and cannot specify stronger ordering than succ

Les instructions atomiques en C11

```
_Bool atomic_compare_exchange_strong( volatile A* obj, C* expected, C desired );
_Bool atomic_compare_exchange_weak( volatile A *obj, C* expected, C desired );
_Bool atomic_compare_exchange_strong_explicit( volatile A* obj, C* expected, C desired,
                                              memory_order succ, memory_order fail );
_Bool atomic_compare_exchange_weak_explicit( volatile A *obj, C* expected, C desired,
                                            memory_order succ, memory_order fail );
```

Atomically compares the value pointed to by *obj* with the value pointed to by *expected*, and if those are equal, replaces the former with *desired* (performs read-modify-write operation). Otherwise, loads the actual value pointed to by *obj* into **expected* (performs load operation).

The memory models for the read-modify-write and load operations are *succ* and *fail* respectively. The (1-2) versions use *memory_order_seq_cst* by default.

The weak forms ((1) and (3)) of the functions are allowed to fail spuriously, that is, act as if **obj* != **expected* even if they are equal. When a compare-and-exchange is in a loop, the weak version will yield better performance on some platforms. When a weak compare-and-exchange would require a loop and a strong one would not, the strong one is preferable.

Cohérence mémoire

MODE	Explanation
<i>memory_order_relaxed</i>	Relaxed ordering: there are no constraints on reordering of memory accesses around the atomic variable
<i>memory_order_consume</i>	Consume operation: no reads in the current thread dependent on the value currently loaded can be reordered before this load. This ensures that writes to dependent variables in other threads that release the same atomic variable are visible in the current thread. On most platforms, this affects compiler optimization only.
<i>memory_order_acquire</i>	Acquire operation: no reads in the current thread can be reordered before this load. This ensures that all writes in other threads that release the same atomic variable are visible in the current thread.
<i>memory_order_release</i>	Release operation: no writes in the current thread can be reordered after this store. This ensures that all writes in the current thread are visible in other threads that acquire the same atomic variable.
<i>memory_order_acq_rel</i>	Acquire-release operation: no reads in the current thread can be reordered before this load as well as no writes in the current thread can be reordered after this store. The operation is read-modify-write operation. It is ensured that all writes in another threads that release the same atomic variable are visible before the modification and the modification is visible in other threads that acquire the same atomic variable.
<i>memory_order_seq_cst</i>	Sequential ordering. The operation has the same semantics as acquire-release operation, and additionally has sequentially-consistent operation ordering.

Le problème du faux partage *False sharing*

- 2 variables indépendantes peuvent être sur la même ligne de cache

int x,y;

Cette ligne de cache peut faire du *ping-pong* entre deux processeurs.

```
int Tab[nb_threads] ;
```

```
Void * thread_fun(void *p)
```

```
{
```

```
while(..)
```

```
{
```

```
Tab[id] += 1;
```

- Solutions

- Utiliser des variable locales au thread (allocation dans la pile)
- Faire du bourrage d'octets (padding)
- Utiliser des directives d'alignement
 - int x __attribute__((__aligned__(64));
 - Voir C11

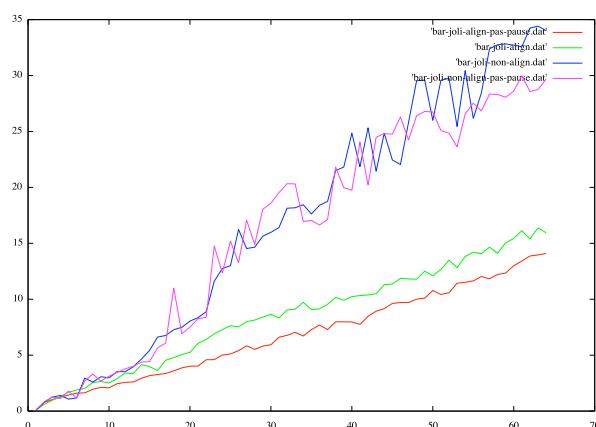
Modélisation d'une barrière

```
volatile int nb = 64;
volatile int cycle; // __attribute__((__aligned__(128)));
volatile int cpt; // __attribute__((__aligned__(128)));


...
for (i=0; i < 10000; i++)
{
    k = __sync_fetch_and_add(&cpt,-1);

    if(k == 1)
    {
        cpt=nb;
        cycle++;
    }
    else
        while(i==cycle){
            // __asm__ __volatile__("pause" :: :);
        }
}
```

Influence du false sharing et de pause
sur une barrière en fonction du nombre de threads



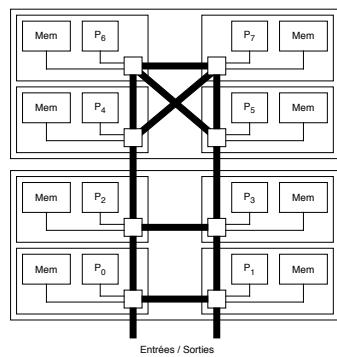
SMP conclusion

- Machines relativement faciles à programmer
 - Comme en séquentiel grâce au protocole MESI
- Mémoire centralisée = goulet d'étranglement
 - Contention, latence mémoire importante
 - Limitation des performances par le débit mémoire
 - Limitation du nombre de processeurs par le débit
 - Il faut donc optimiser l'utilisation du cache
 - Cold miss: premier accès à une variable.
 - Capacity miss: le cache est complet. *Travailler sur la localité (pavage)*
 - Conflict miss: le cache n'est pas complet mais son associativité trop faible. *Revoir l'alignement*
 - True sharing miss: défaut nécessaire à la communication entre les threads. *Limiter les synchronisations*
 - False sharing miss: défaut inutile. *Revoir l'alignement*
- Programmation lock-free (via des opérations atomiques)
 - *The art of multiprocessor programming*, par M. Herlihy & N. Shavit chez Morgan Kaufman
- L'apparition du multicœur a condamné cette approche pour les multiprocesseurs.
 - Trop de coeurs par processeur

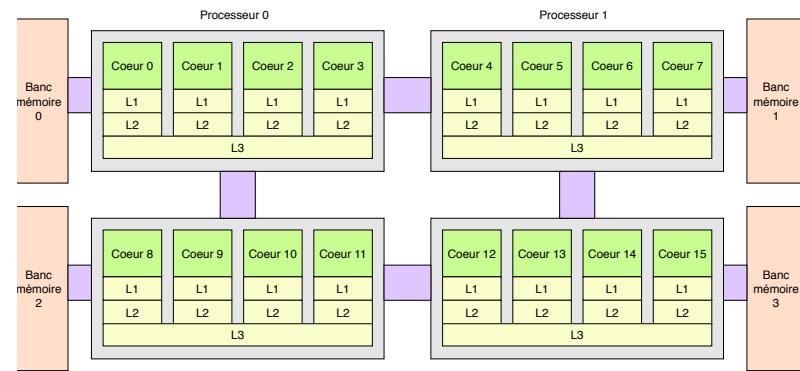
NUMA

non uniform memory access

- Nœud NUMA
 - Briques : processeurs + mémoires
 - Réseau de communication inter nœud
 - La latence mémoire dépend de la distance entre le processeur et la mémoire en jeu.
 - Disponibilité
 - Calculateurs des années 90 (SGI, IBM, ...)
 - AMD : opteron + hypertransport 2005
 - Intel : Nehalem + QuickPath Interconnect (2008)

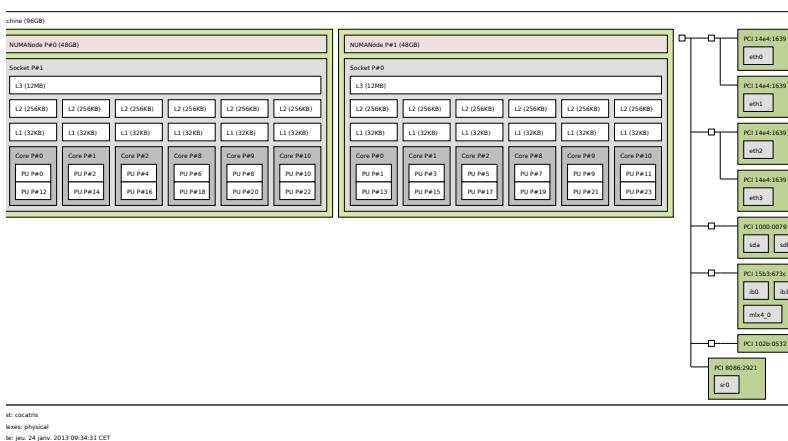


Impact NUMA opteron 4 x 4



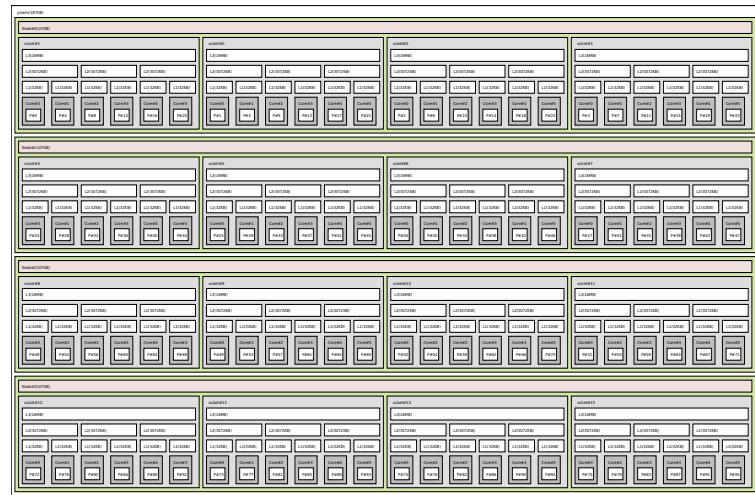
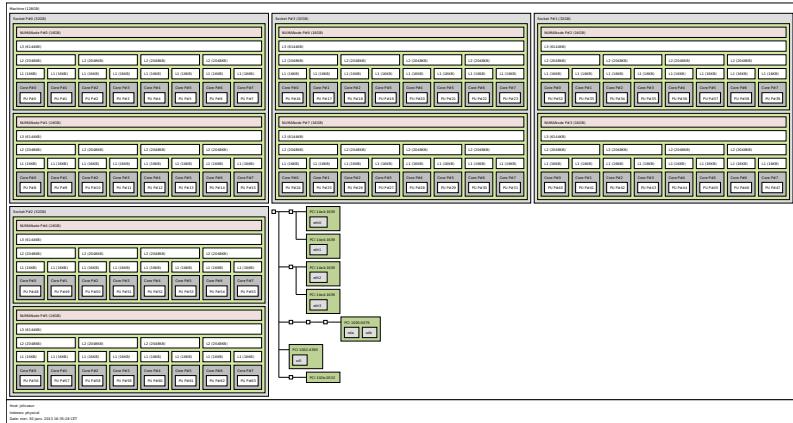
Type d'accès	Accès local	Accès au banc voisin	Accès au banc opposé
Lecture	83 ns	98 ns ($\times 1,18$)	117 ns ($\times 1,41$)
Ecriture	142 ns	177 ns ($\times 1,25$)	208 ns ($\times 1,46$)

cocatrix



Boursouf

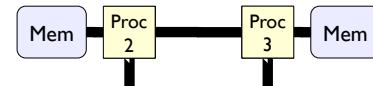
jolicoeur



NUMA

- Le problème de la contention mémoire
- Outils et techniques pour le placement mémoire
- Réalisation matérielle
 - MESI sur NUMA
 - Répertoire

Contention mémoire



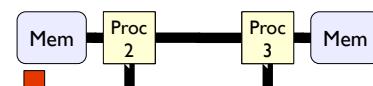
- ▶ Application synthétique



Local: 3621 Mo/s

- ▶ Deux placements mémoire différents
- ▶ Allouer localement

- ▶ Répartir les pages mémoire sur les nœuds voisins

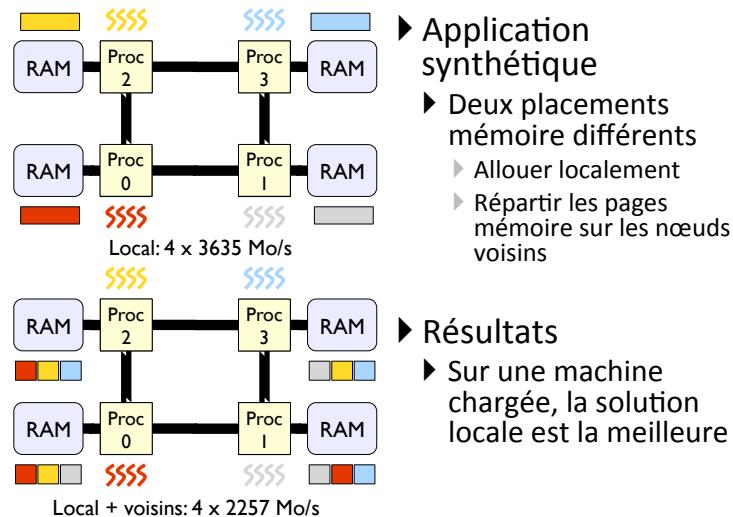


Local + voisins: 3940 Mo/s

Résultats

- ▶ Sur une machine non chargée, la solution répartie est la meilleure

Le problème de la contention mémoire



Outils pour le placement

Libnuma (linux)

- Placement des threads
 - int sched_setaffinity(pid_t pid, size_t cpusetsize, cpu_set_t *mask);
 - Politiques placement mémoire
 - Local, distant, interleave
 - Numactl en ligne de commande
- Allocation first touch
 - Allocation paresseuse des pages
 - Placement réalisé à la première utilisation
 - Suivant la politique définie
 - Par défaut la page est placée sur le nœud numa du cœur ayant causé l'allocation
 - Nécessite la connaissance de l'architecture pour faire des placements optimaux

Placement des threads

GNU / LINUX

Définition d'un masque précisant les numéros des cœurs où le thread pourra être exécuté

```
int p[P];
pthread_t t[P];
pthread_attr_t attr[P];

for(i = 0; i<P; i++)
{
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(%616,&cpuset);
    pthread_attr_init(&attr[i]);
    pthread_attr_setaffinity_np(&attr[i], sizeof cpuset, &cpuset);
    p[i]=i;
    pthread_create(&t[i],&attr[i],thread_function,&p[i]);
}
```

**OMP_PROC_BIND=true
OMP_CPU_AFFINITY="0 3 1-2 4-15:2"**

Optimisation mémoire

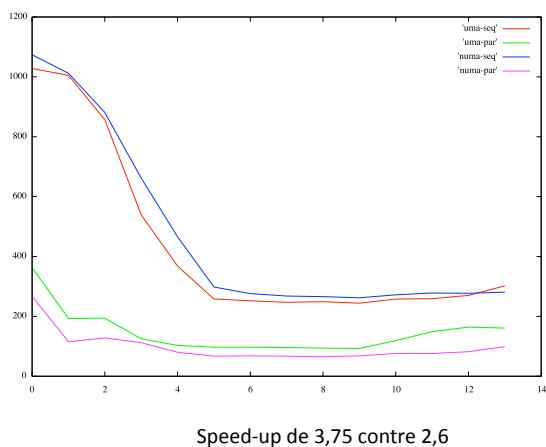
Initialisation de la matrice par les threads sur le cœur où il seront exécutés

```
void *init_par(void *p){
    int numero =*(int*)p;
    int i,j;

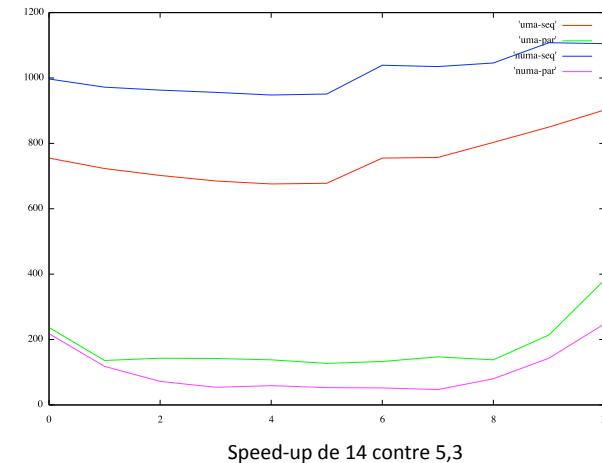
    int debut = ((N/P)*numero);
    int fin= ((N/P)*(numero+1));

    for (i=debut;i<fin;i++)
        for (j=0;j<N;j++)
            C[i][j]=B[i][j]=A[i][j] = ... ;
}
```

Application au problème « film » miro 16 threads



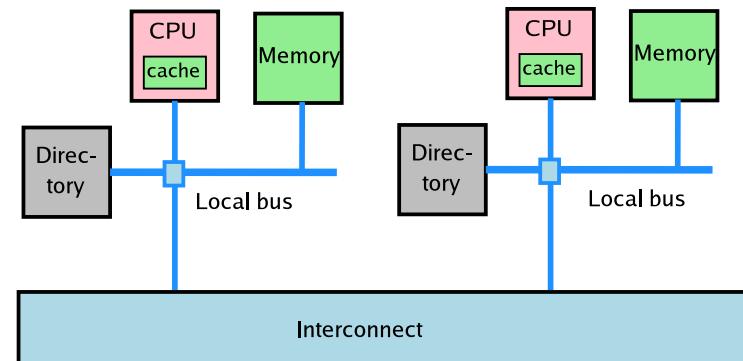
Application au problème « film » boursouf 48 threads



Stratégie Placement thread / mémoire

- Stratégie de placement thread / mémoire
 - Déterminer les dimensions où l'application à le comportement le plus régulier possible
 - Découper le travail en tâches équilibrées et les affecter de façon statique aux threads
 - Réaliser l'allocation physique des pages
 - Boucle d'initialisation à blanc réalisé par chaque thread
 - Lancement de l'application
- Commentaires :
 - Stratégie bien comprise par les programmeurs (OpenMP) un peu expérimentés
 - Ne s'applique pas aux problèmes irréguliers
 - « move on next touch »
 - Ne tient pas compte de la charge mémoire des nœuds
 - Ni de la consommation de la bande passante

Répertoire NUMA



Répertoire NUMA

Cache line data (e.g. 64 bytes)	Owner	0	1	2	3	4	5	6	7
	0	☒							
	3			☒	☒	☒			
	5	☒					☒		
	3	☒		☒					
	1	☒	☒			☒	☒		

...

Memory itself
(usually DRAM)

Node ID of
owner of
cache line

1 bit per node
indicating
presence of
line

MESI sur Opteron & Nehalem

	Clean/Dirty	Unique?	Can Write?	Can Forward?	Can Silent Transition to	Comments
Modified	Dirty	Yes	Yes	Yes		Must writeback to share or replace
Exclusive	Clean	Yes	Yes	Yes	MSIF	Transitions to M on write
Shared	Clean	No	No	No	I	Does not forward
Invalid	NA	NA	NA	NA		Cannot Read
Forwarding	Clean	Yes	No	Yes	SI	Must invalidate other copies to write

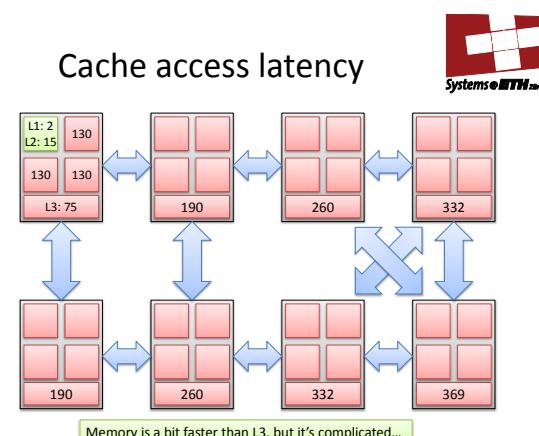
	Clean/Dirty	Unique?	Can Write?	Can Forward?	Can Silent Transition to	Comments
Modified	Dirty	Yes	Yes	Yes	O	Can share without writeback
Owned	Dirty	Yes	Yes	Yes		Must writeback to transition
Exclusive	Clean	Yes	Yes	Yes	MSI	Transitions to M on write
Shared	Either	No	No	No	I	Shared can be dirty or clean
Invalid	NA	NA	NA	NA		Cannot Read

	Clean/Dirty	Unique?	Can Write?	Can Forward?	Can Silent Transition to	Comments
Modified	Dirty	Yes	Yes	Yes		Must writeback to share or replace
Exclusive	Clean	Yes	Yes	Yes	MSI	Transitions to M on write
Shared	Clean	No	No	Yes	I	Shared implies clean, can forward
Invalid	NA	NA	NA	NA		Cannot Read

MOESI / MESIF

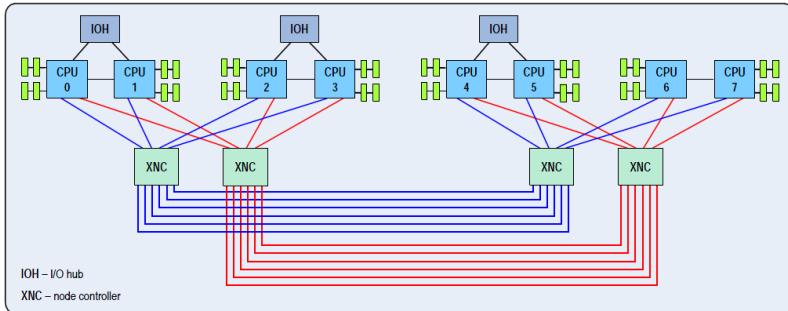
- O - Owned :**
 - La ligne de cache est valide.
 - Cette ligne est peut être partagée par d'autres caches qui eux sont dans l'état Shared.
 - La valeur contenue par la mémoire principale est incorrecte.
 - Ce cache est responsable de l'écriture en mémoire de la ligne.
- F – Fowarding**
 - La ligne de cache est valide et inchangée par rapport à la mémoire.
 - Cette ligne est peut être partagée par d'autres caches qui eux sont dans l'état Shared.
 - Ce cache est responsable de la transmission de la ligne aux autres caches

Opteron 8 sockets



HP Prema - Nehalem

In 8-socket glueless systems, snoops consume 50 to 65% of QPI bandwidth.
In the HP PREMA Architecture with smart CPU caching, coherency snoops and responses consume only 10 to 20% of QPI bandwidth and that of the HP fabric.



The HP implementation provides latency for local memory access comparable to a 4-socket glueless system and 30% lower latency when compared to an 8-socket glueless system.

Conclusion NUMA

- Le tournevis
 - Solution universelle et performante
 - Mais coûteuse en temps de développement et en matière grise
 - Cercle de plus en plus restreint d'experts
- Portabilité des performances
 - Utiliser des bibliothèques développées par des experts
 - Paramétrées par l'architecture de la machine cible
 - Ou basées sur des techniques d'*auto-tuning*
 - adaptation automatiquement au matériel via des techniques d'apprentissage
 - Utiliser des environnements de programmation parallèle
 - Programmer en exprimant de façon structuré le parallélisme
 - Parallelisme imbriqué, graphe de tâches
 - Déléguer l'ordonnancement à un support d'exécution
 - Placement dynamique des tâches / flux
 - Possibilité de contraindre le placement

Il s'agit de séparer la génération du parallélisme de son ordonnancement

Conclusion Archi

- ILP :
 - Limites atteintes il y a 10 ans
 - Consommation énergétique
 - Complexité des circuits
 - Autres approches
 - EPIC (itanium) : impasse, les compilateurs ne sont pas assez puissants.
 - Succès des processeurs *in order* (ARM) dans le domaine de l'embarqué
- Multicœur
 - Incontournable : que faire du silicium ?
 - Développement des architectures hétérogènes
 - Quelques gros cœurs pour l'ILP
 - Beaucoup de petits cœurs pour le TLP
- Quel avenir pour l'approche mémoire commune ?
 - Bugs difficiles
 - Performances pénalisées par la cohérence séquentielle

Comparaison pour 32 threads machine boursouf

	Mémoire sur le nœud 0	Mémoire répartie	rapport
Add séquentiel	1149	1878	0,61
Add // statique	632	57	11,09
Add // dynamique	535	425	1,26
Sum séquentiel	238	477	0,50
Sum // var part.	14974	13635	1,10
Sum // var local	207	23	9,00

Résultats expérimentaux

nehalem salle 008

$$C[i][j] = \cos(A[i][j]) + \sin(B[i][j]);$$

Distribution statique des lignes par blocs

Nb threads	base	Optim. placement		Optim. mémoire		Placement + mémoire	
1	5533,15	0,98	5534,75	0,98	5428,23	1,00	5404,59
2	2819,41	1,92	2811,09	1,92	2766,00	1,95	2747,78
4	1450,32	3,73	1419,52	3,81	1400,10	3,86	1373,94
8	709,62	7,62	724,34	7,46	697,68	7,75	685,75
16	494,69	10,93	460,49	11,74	433,69	12,48	435,69
32	453,92	11,91	454,95	11,88	435,89	12,40	444,82
64	457,33	11,82	466,93	11,57	438,73	12,32	443,83
							12,18

Optimisation placement : on fixe les threads sur les cœurs

Optimisation mémoire : on réparti le tableau sur les 2 sockets

Répertoire NUMA

AMD HT-ASSIST : 2^{14} entrées

