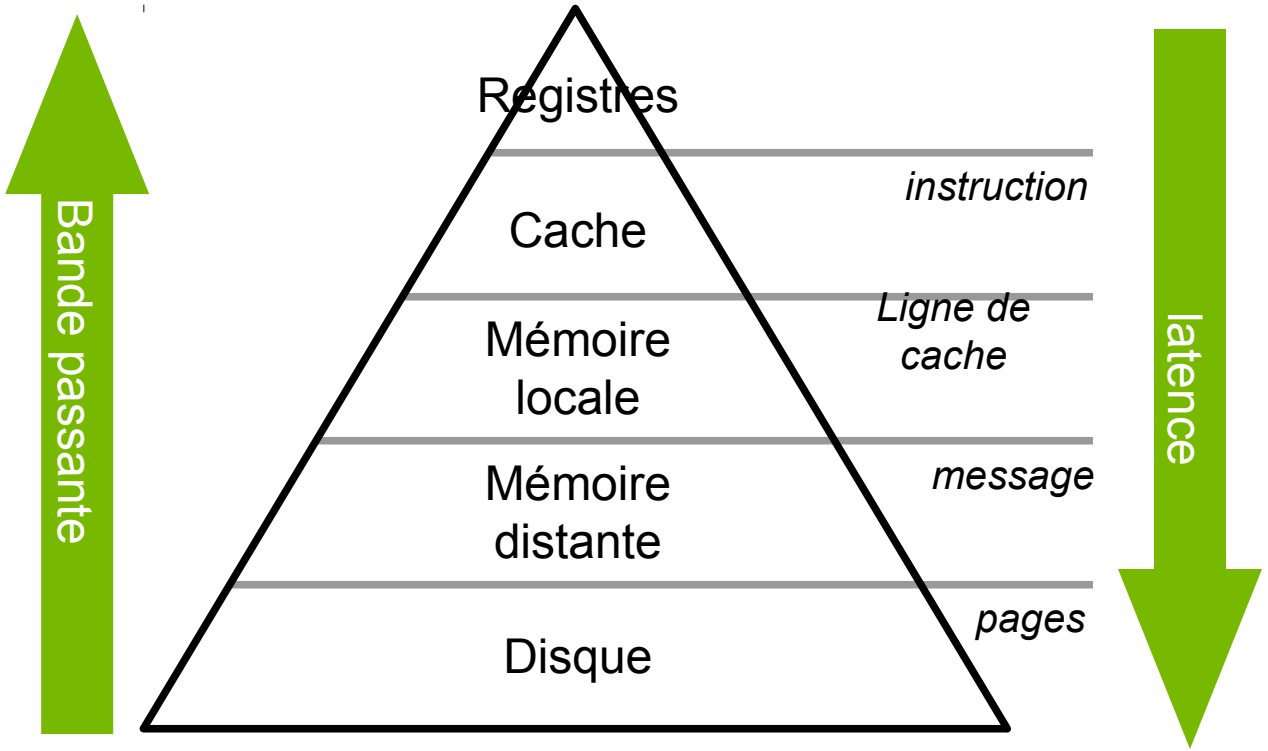
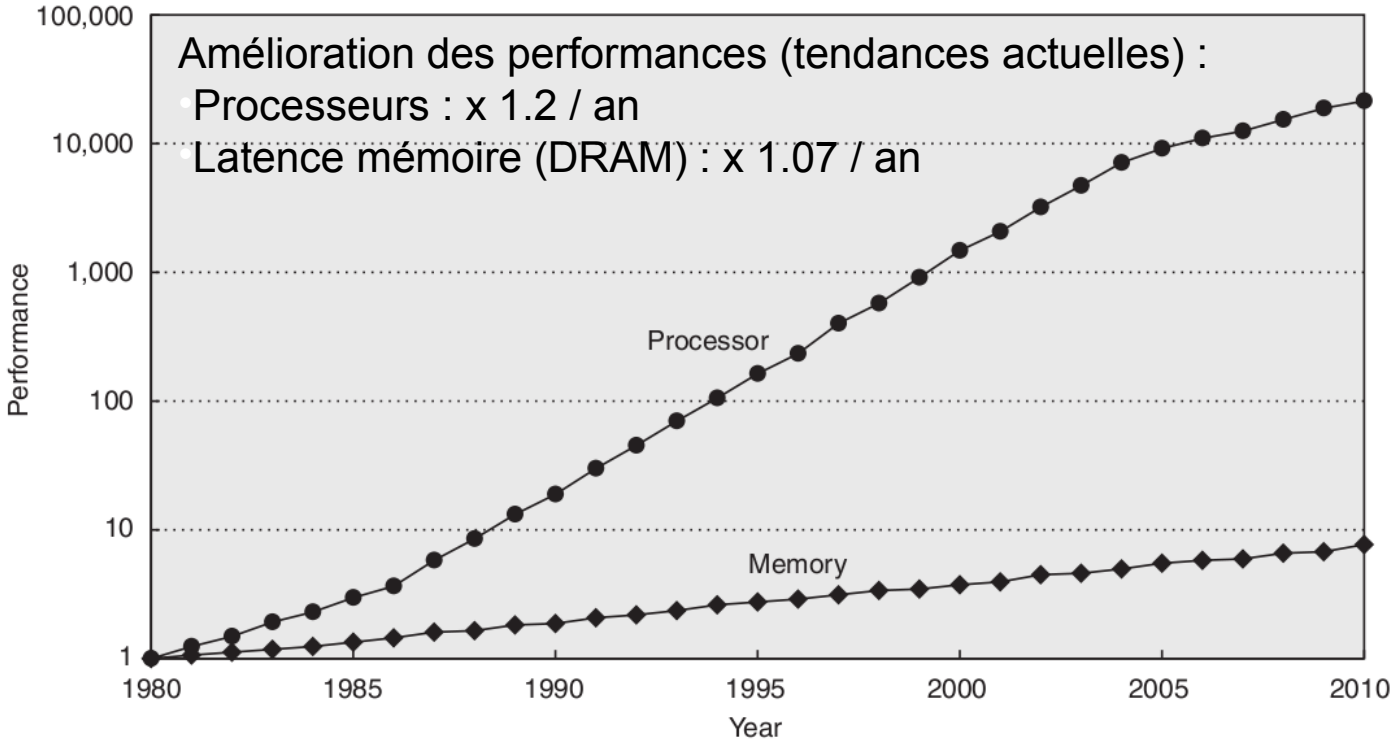
Elements de microarchitecture

2. Mémoire

Hiérarchie mémoire



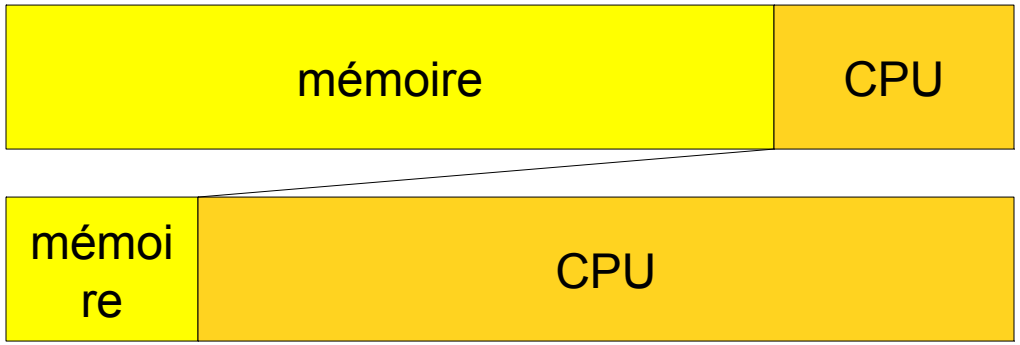
« Hitting the memory wall »



- Bill Wulf and Sally McKee (1994) : « *Hitting the Memory Wall: Implications of the Obvious* »
  - La croissance exponentielle de l'écart entre la performance des mémoires et processeurs va conduire rapidement à ce que les performances des codes ne soient uniquement dominées par les performances des mémoires.

Motivations

- On cherche un équilibre entre accès mémoires et calculs



On ne fait que déplacer un bottleneck

- Les codes actuels sont à tendance « memory-bound » plutôt que « CPU-bound »
  - Les bandes passantes ont tendance à augmenter
    - Mais le nombre de cœurs aussi !
  - Augmenter la taille des caches : cher, limité
  - On doit optimiser son code : « *cache-reuse* »
  - Les fréquences à diminuer ou stagner (conso.)

Accès mémoires

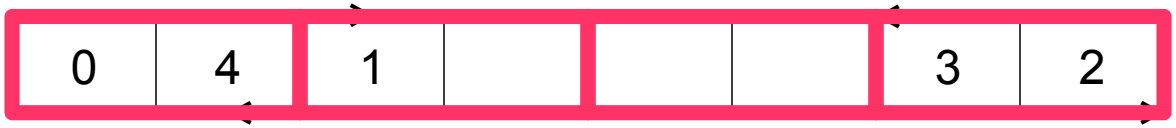
- Il est question des **accès mémoires** (les données)
  - Pour l'utilisateur, les métriques pertinentes sont :
    - Le temps d'accès à une donnée : la latence (temps ou nombre de cycles)
    - Le débit (GiB/secondes)
- Quels facteurs influent sur les performances des accès mémoire
  - Localisation : cache ? Mémoire centrale ? Pis, disques !
  - Si les données sont en RAM
  - La manière dont elles sont accédées :
    - Au travers d'un chipset, directement par le processeur ? Via le processeur voisin ?
      - → Les caractéristiques propres de chacun de ces éléments et de leur agencement (population des bancs mémoire par ex.)

Débit ou latence ?

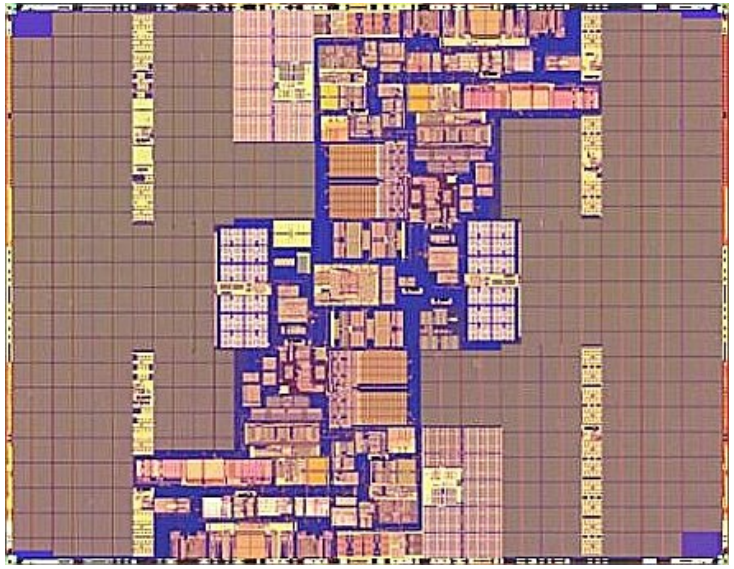
- Quel est la métrique la plus pertinente ?
  - Ca dépend !
    - Accès aléatoires : latence
      - → Accès au cache
      - → Accès à la mémoire centrale
    - Accès séquentiels : débit

Latence Mémoire

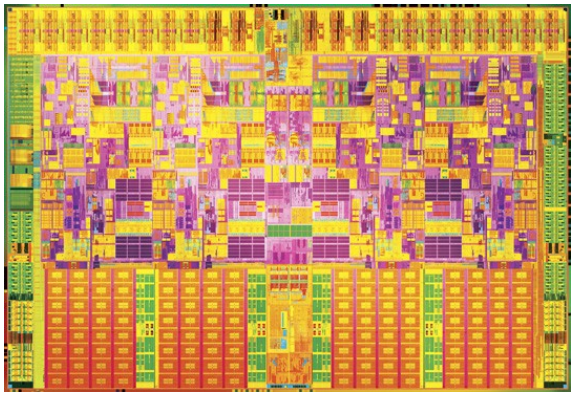
- Accès aléatoire de la mémoire
  - C'est le cas de certains codes applicatifs : multiples niveaux d'indirection (maillage, CFD).
  - L'accès à la mémoire coûte cher : 200 cycles est un bon ordre de grandeur ...
  - Pas forcément évident : il faut « flouer » le « prefetch » (software et hardware)
    - « *chase-pointer strategy* » : naviguer dans une liste chaînée de pointeur.
    - Attention à certaine options d'optim. des compilateurs (prefetch, code mort).
  - Unité : nombre de cycle, nanoseconde, parfois le Gups/sec (« *giga updates per second* »)



Cache



Intel IA64 Montecito (1.6 GHz)

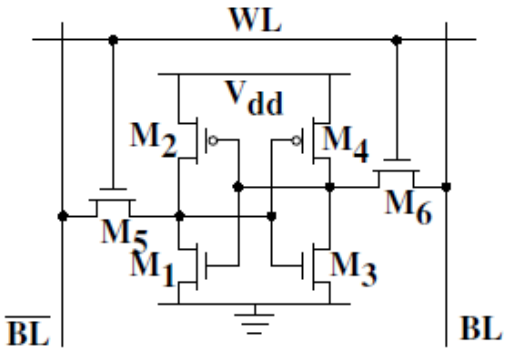


Intel Xeon Nehalem (3.2 GHz)

- Ex. Accès cache : 15 cycles – Accès Mémoire : 200 cycles
  - Accès à 100 éléments, 100 fois
    - Sans cache : 2 000 000 cycles
    - Avec cache : 168 500 cycles
  - > 91 % de gain !

SRAM ou DRAM ?

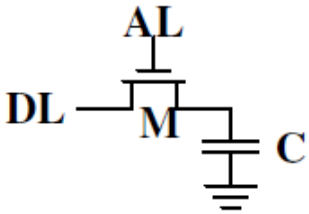
- **SRAM**



Static RAM

- Avantage
  - Rapidité
  - Stabilité (pas de rafraîchissement)
- Désavantage
  - Coût
  - Densité

- **DRAM**



Dynamic RAM

- Avantage
  - Côut
  - Densité
- Désavantage
  - Latence/Débit
  - Besoin d'un rafraichissement périodique

Cache

- Localité temporelle ou spatiale des données
- Design parameters
- Capacity (size)
- Line size
  - Banking
- Coherency
  - Protocol (Ex. MESI), “snooping”
- Associativity
  - Direct-mapped
  - Set-associative
  - Fully associative
- Block replacement policy
  - LRU, LFU, FIFO, random
- Write policy
  - Write-back, write-through (write buffer)
- Allocate-on-write-miss policy

- Victim buffer
- Cache unification
- Prefetching

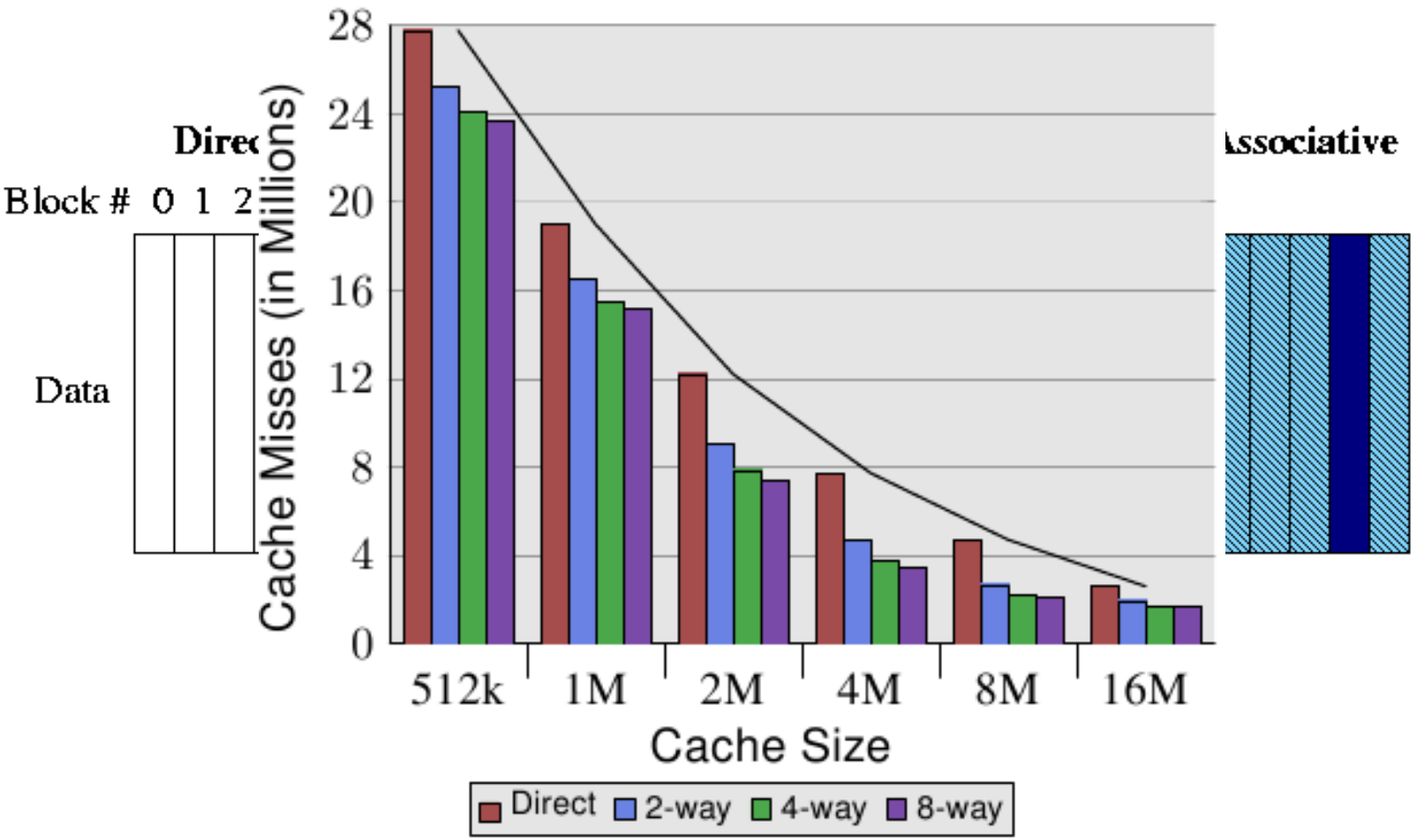
Cache

- Les caches sont des éléments matériels complexes.
  Cette complexité est cachée au programmeur.

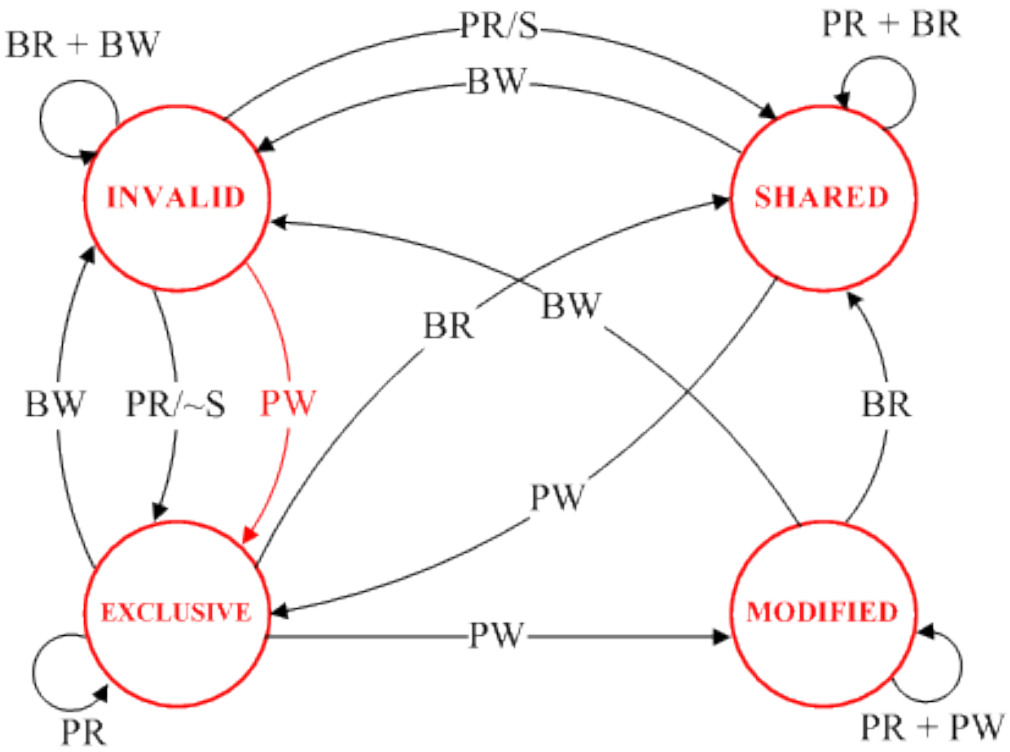
- Par contre il existe des règles de bonne usage des caches
  - Il faut travailler sur la réutilisation des données
  - Leur localité

- Le programmeur peut lui même gérer la cohérence de cache
  - “Explicit cache control”
  - Utiliser par les compilateur dans les phases d’optimisation
  - Exemple d’instruction ou principe :
    - → prefetching
    - → “non temporal stores” ou “streaming stores”
    - → fence
    - → flush

Cache | Associativité



Protocol MESI



PR = processor read
PW = processor write
S/~S = shared/NOT shared

BR = observed bus read
BW = observed bus write

Alignement des données

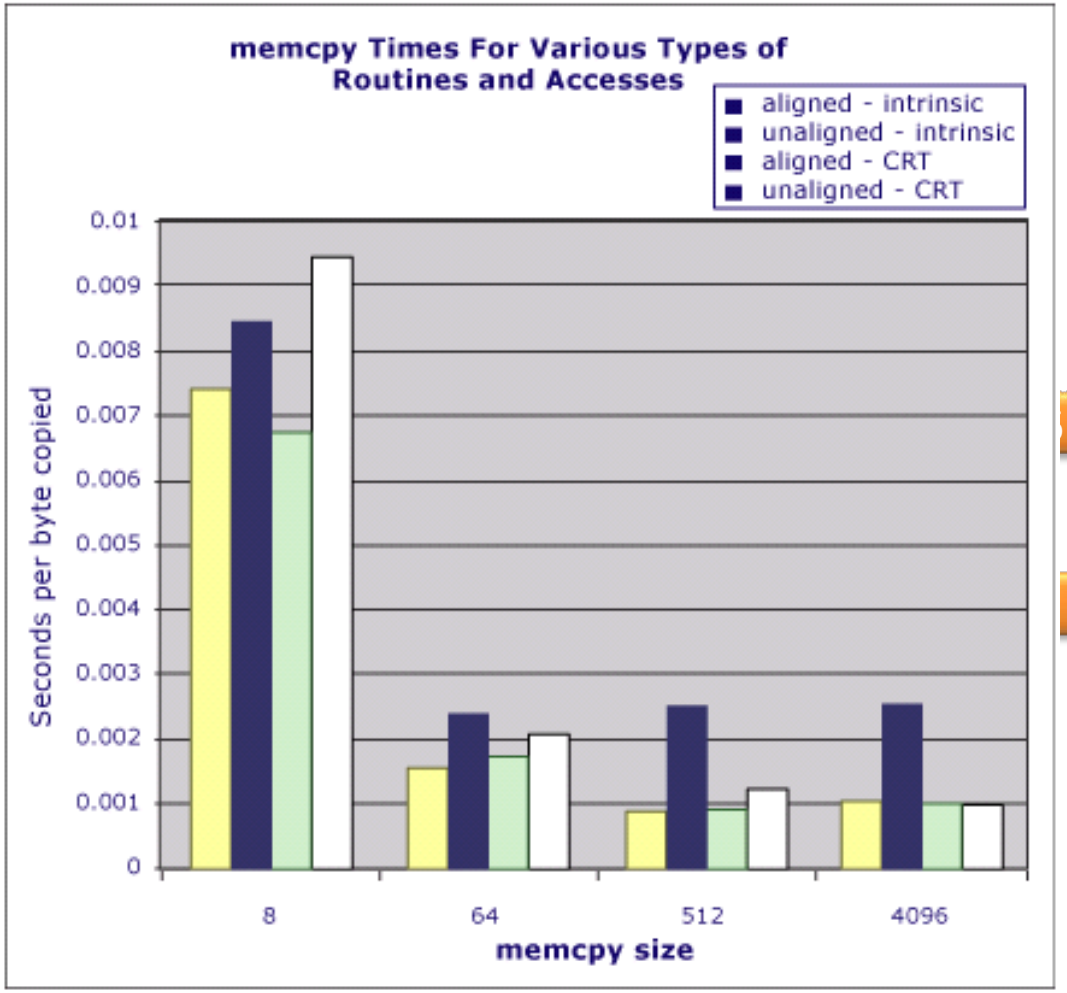
Lignes de

Ali

Non-ali

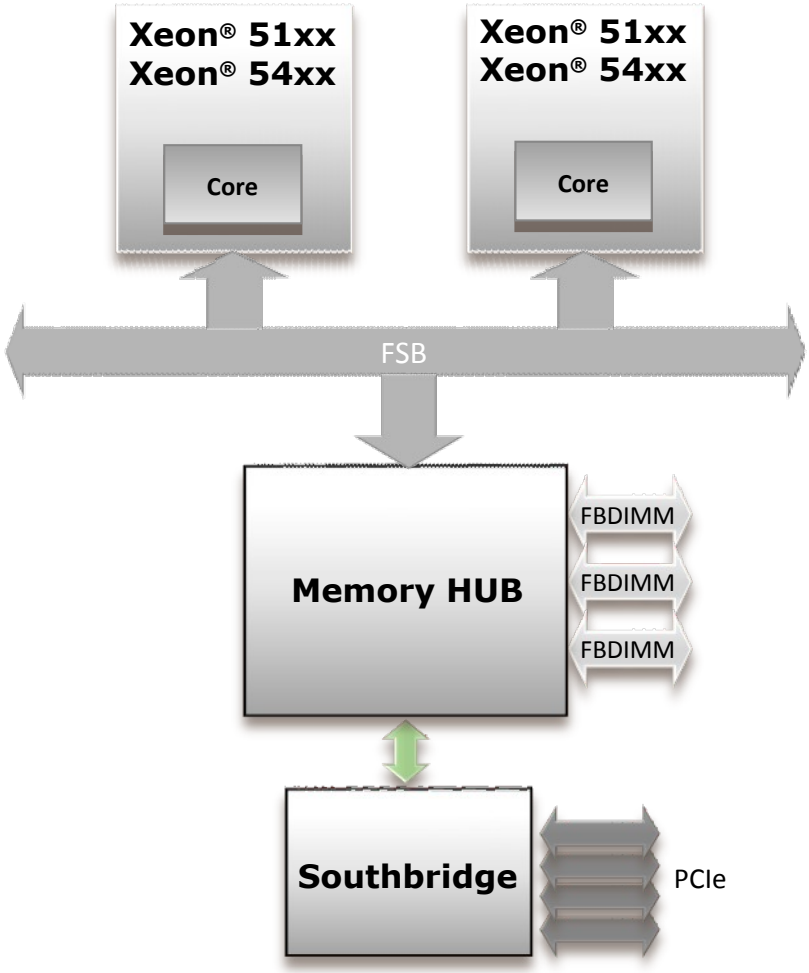
Ali

Non-ali

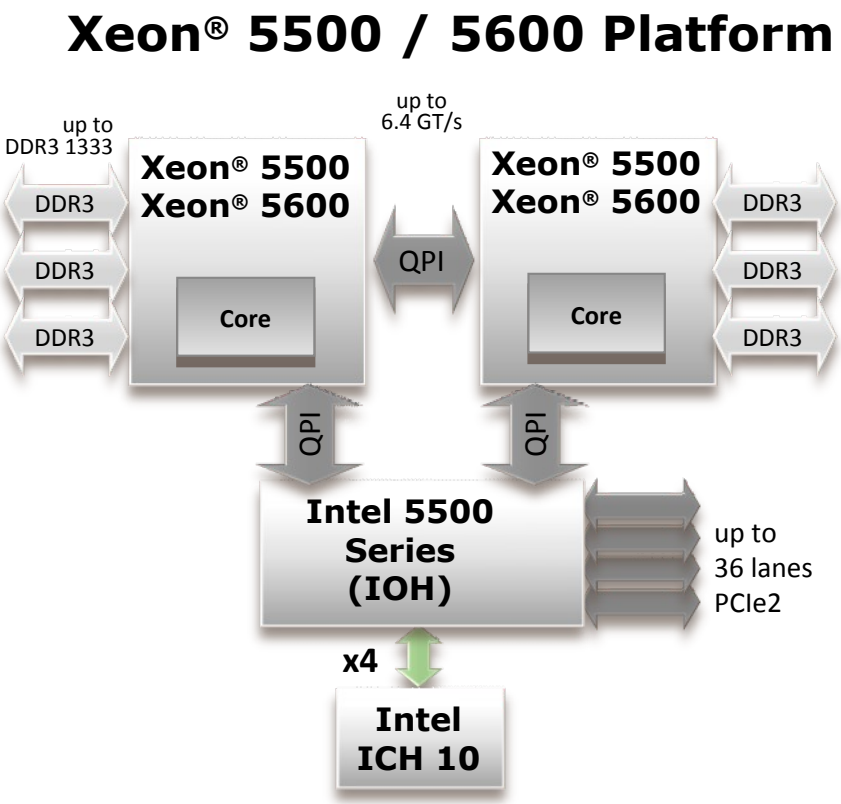


16

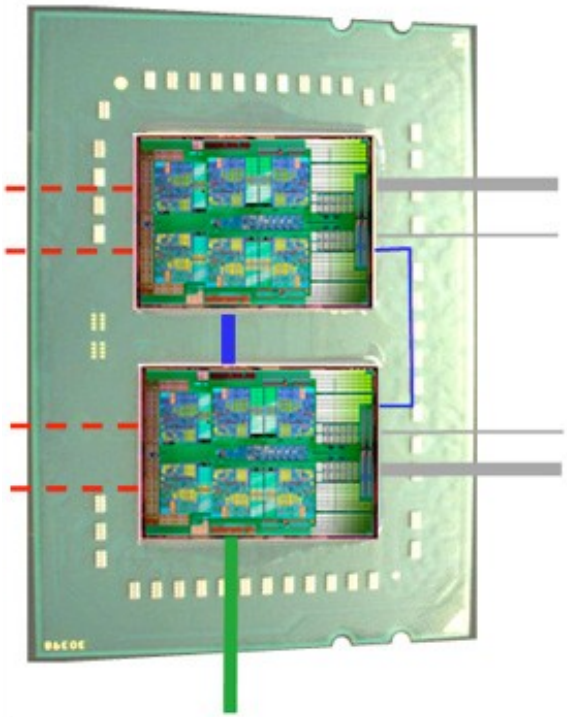
Configuration non-NUMA avec FSB



Configuration NUMA

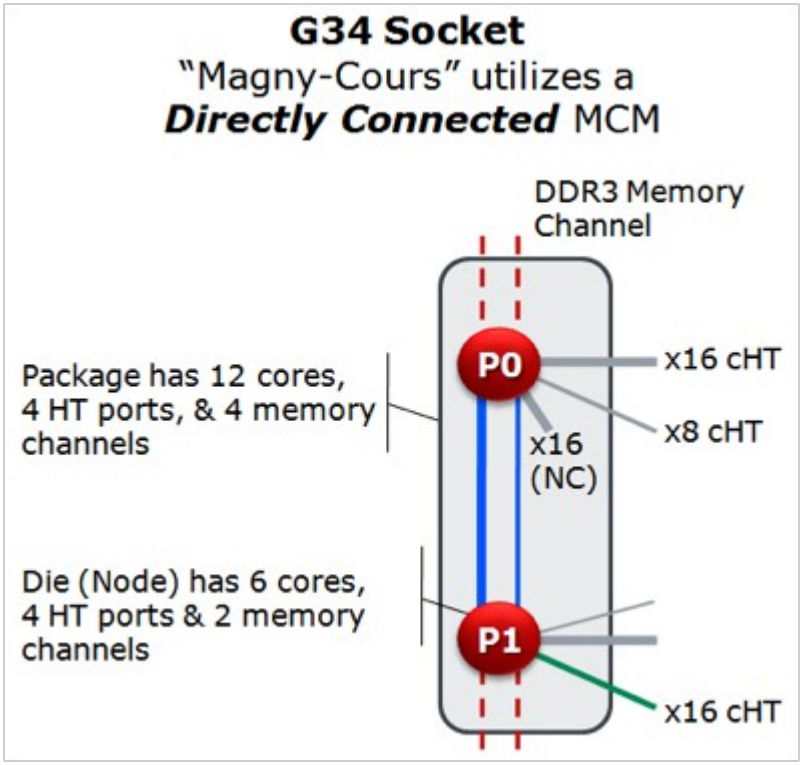


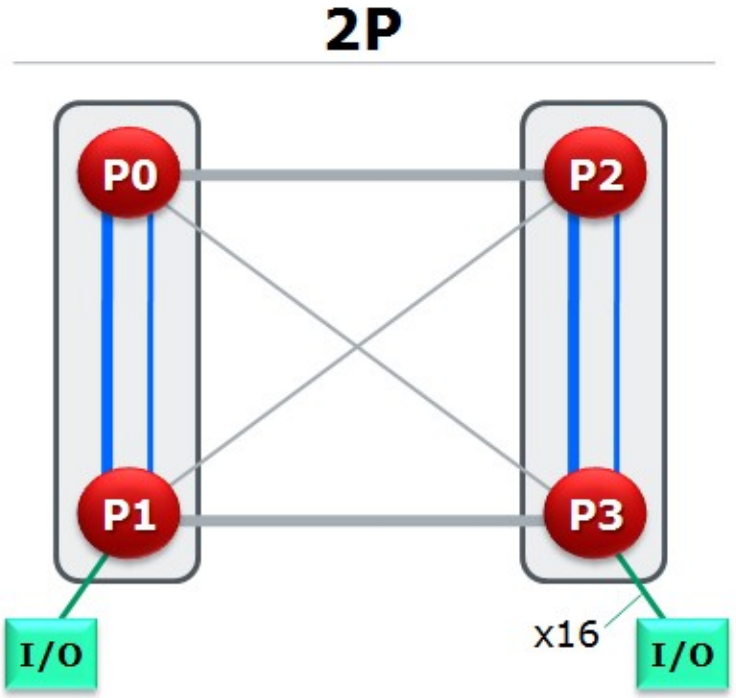
Configuration NUMA



AMD "Magny Cours"

Configuration NUMA



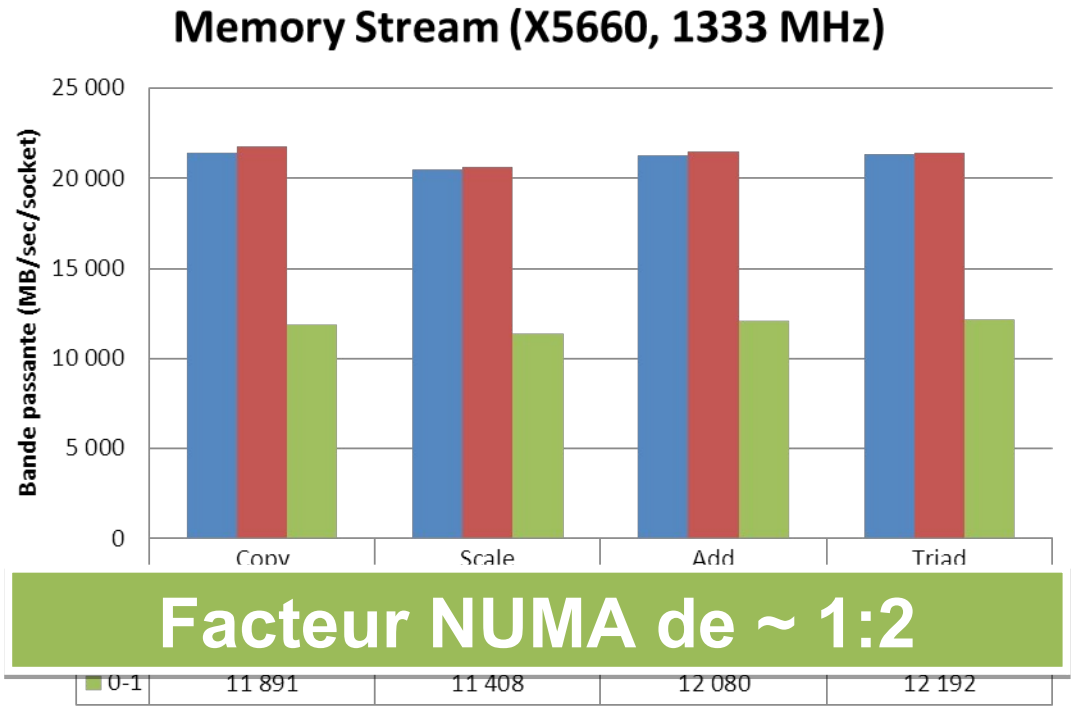


NUMA : bande passante mémoire

- Linux possède tout ce qu'il faut pour générer le NUMA

conf. n/m :

```
numactl –membind=n –cpubind=m ./cmd arg1, arg2, ...
```



NUMA : bande passante mémoire

- Linux possède tout ce qu'il faut pour gérer le NUMA
  - Au niveau noyau (extension cpuset)
  - Au niveau utilisateur :
    - libnuma
      - API relativement simple
      - Commande utilisateur : `numactl` (cpu et mémoire)
    - Commande `tasket` (binding et process binding)
    - API posix: `sched_{set,get}affinity ()`

- Connaître la topologie de la machine (noeuds de calcul)
  - `/proc/cpuinfo, /sys/devices/systems/{node,cpu}`
  - Outils livrés avec les piles MPI : `cpuinfo`, `hwloc`, …
  - `numactl –h`
  - Tout cela est basé sur l'instruction assembleur `cpuid`

NUMA : bande passante mémoire

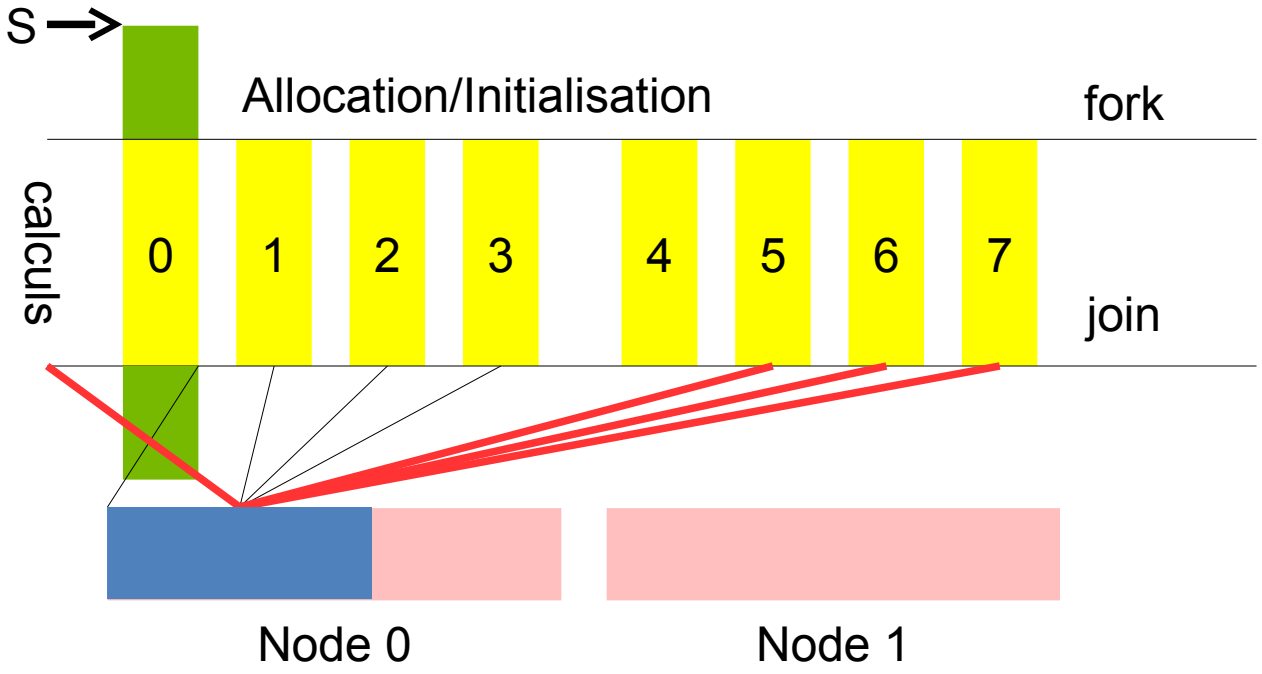
- Linux possède tout ce qu'il faut pour gérer le NUMA
  - Au niveau noyau (extension cpuset)
  - Au niveau utilisateur :
    - lib
      -
      -
    - C
    - Al
- Conn l)
  - /pr
  - Outi
  - numactl -H
  - Tout cela est basé sur l'instruction assembleur `cpuid`

Binder vos process
Placer vos process
Gérer la localité de la mémoire

N'utiliser pas l'hyperthreading
N'utiliser pas le turbomode ou équivalent (quoique …)

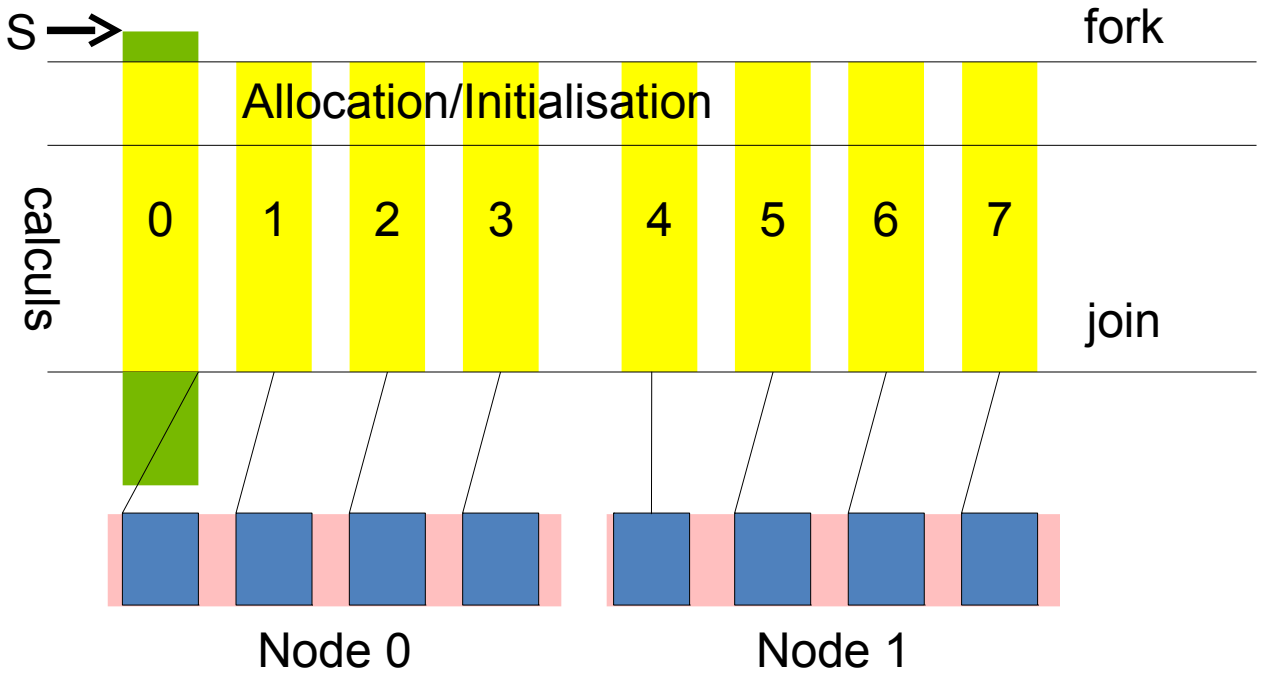
Effet du NUMA

- Processus OpenMP 8 threads, placement « compact »

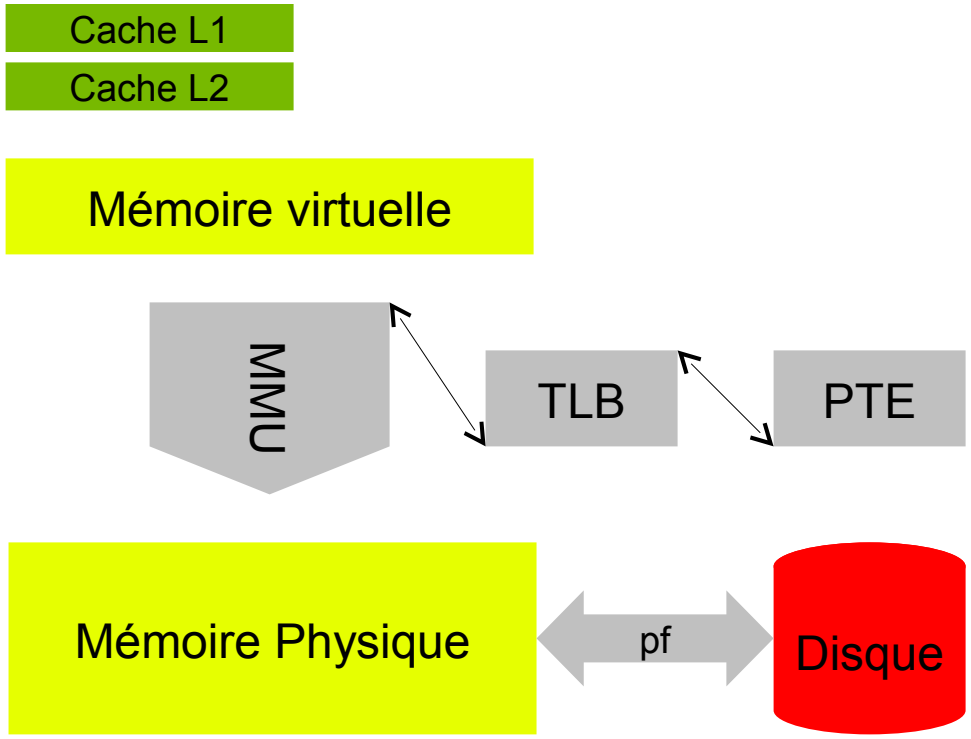


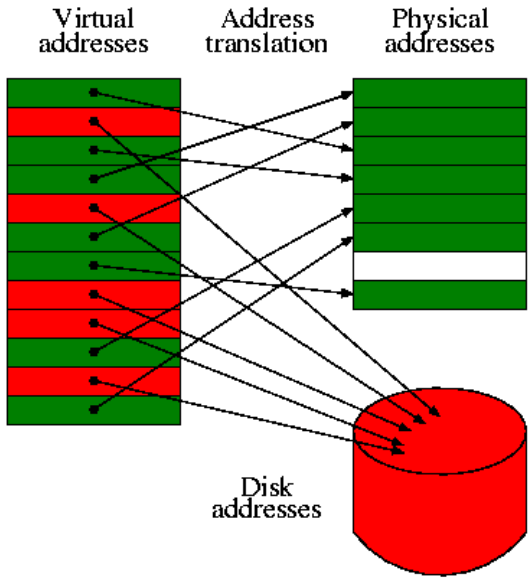
Effet NUMA

- Processus OpenMP 8 threads, placement « compact »



Mémoire virtuelle



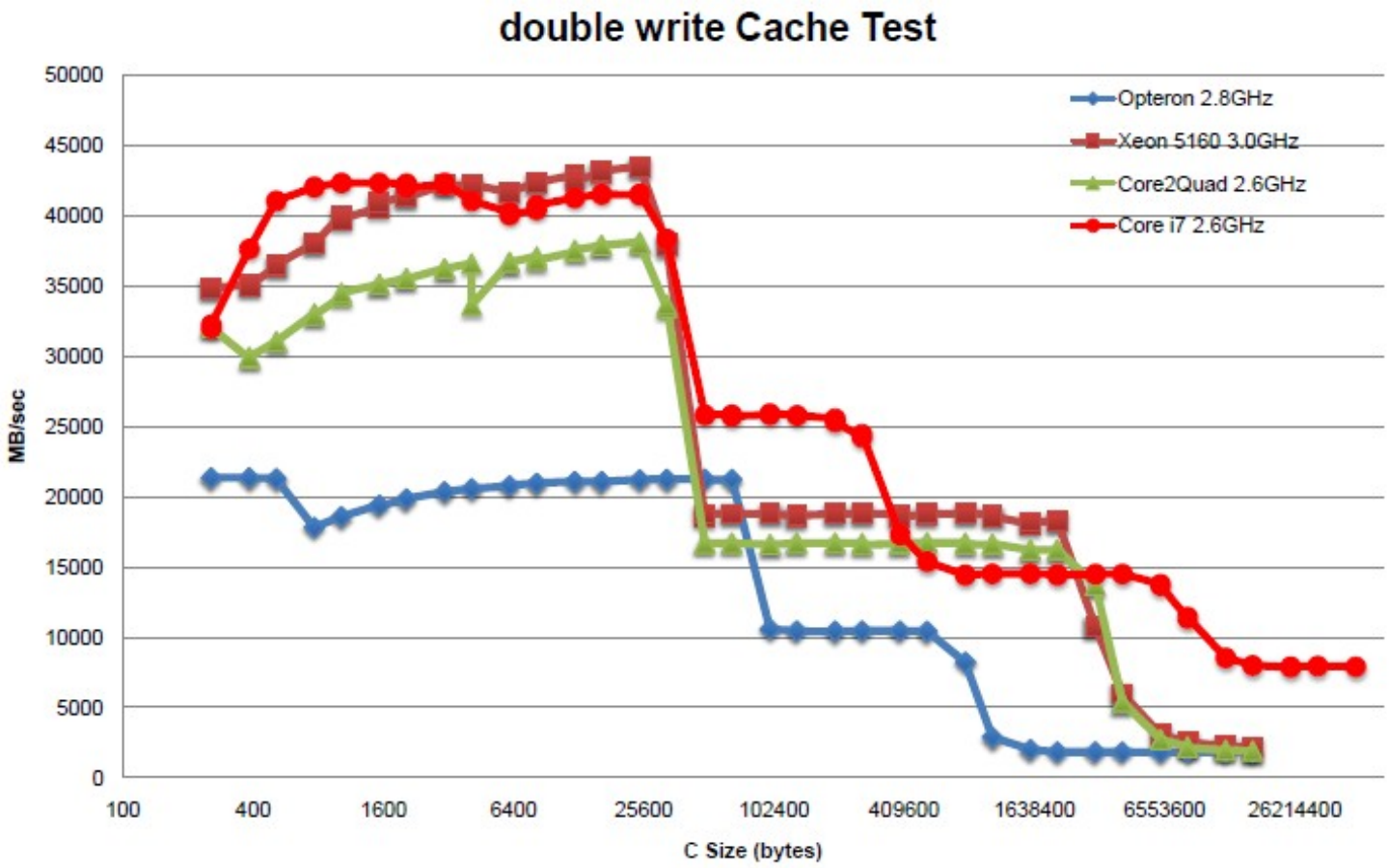


- Clairement, accéder à une donnée en mémoire est une chose complexe. Remonter aux débits et aux latences à partir des seules caractéristiques des barrettes est mission (quasi-)impossible.
  → **Démarche empirique : benchmarks !**

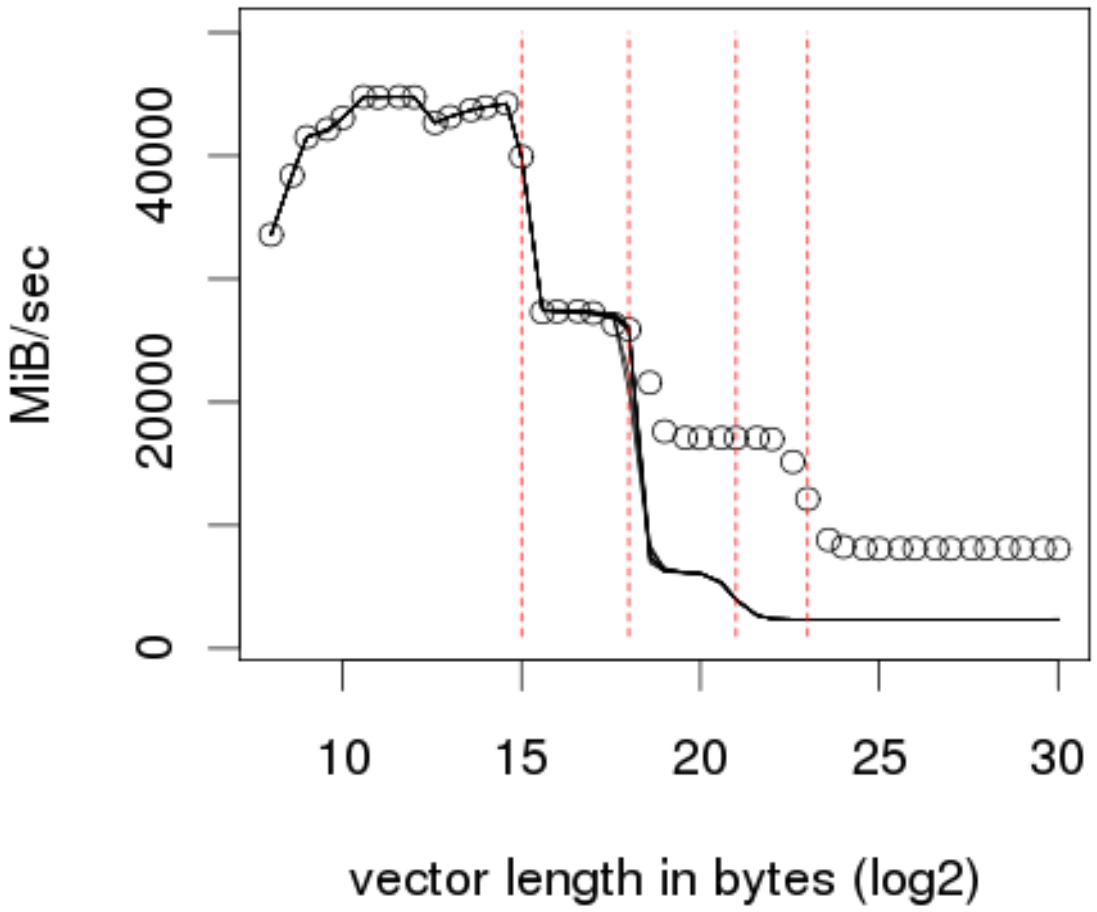
Cachebench (llcbench)

- http://icl.cs.utk.edu/projects/llcbench/
- Mesure de la bande passante pour différente taille de tableau (256 bytes à plusieurs dizaine de MiBytes) et différents « pattern »
  - Découverte de la hiérarchie mémoire
- Portage :
  - Immédiat
  - C
- Execution :
  - Immédiat

Hiérarchie de cache



Hiérarchie de cache



Intel X5570 et mémoire 1066 MHz

- L1 32KB, L2 256 KB, L3 8M (partagée)

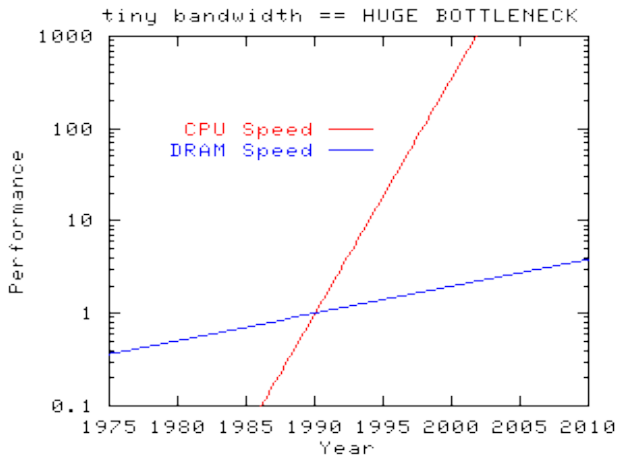
Stream benchmark

- http://www.cs.virginia.edu/stream/

John McCalpin

« The STREAM benchmark is a simple synthetic benchmark program that measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for simple vector kernels. »

- Benchmark accidentel ! Développer pour comprendre les différences de performance entre deux architectures pour un code météo.
- Indispensable dans sa trousse-à-outil
- Accès mémoires séquentiels
  - Portage :
    - → Aucune difficulté
    - → Fortran ou C (OpenMP)
  - Exécution : aucune difficulté
  - Auto-vérifiant (petite taille)



Stream benchmark | comment mesurer une bande passante ?

Operations		# FOP	#MEMOP	Bytes transférés/iter
Copy	A[i]=B[i]		2	16
Scale	A[i]=αB[i]	1	2	16
Add	A[i]=B[i]+C[i]	1	3	24
Triad	A[i]=B[i]+αC[i]	2	3	24

```
T0=time()
for i=0,N
    call Op(i)
endfor
T1=time()
```

$$\text{Bande Passante} = \frac{N \times \langle \text{nombre de bytes transférés} \rangle}{T1-T0}$$

Stream benchmark

```
-------------------------------------------------------------
STREAM version $Revision: 5.9 $
-------------------------------------------------------------
Array size = 20000000, Offset = 0
Total memory required = 457.8 MB.
Each test is run 10 times, but only
the *best* time for each is used.
-------------------------------------------------------------
Function      Rate (MB/s)   Avg time     Min time     Max time
Copy:         3718.0250       0.0862       0.0861       0.0864
Scale:        3772.0257       0.0850       0.0848       0.0852
Add:          4138.4692       0.1162       0.1160       0.1167
Triad:        4149.5237       0.1160       0.1157       0.1164
-------------------------------------------------------------
Solution Validates
-------------------------------------------------------------
```

lat_mem_rd (LMBench)

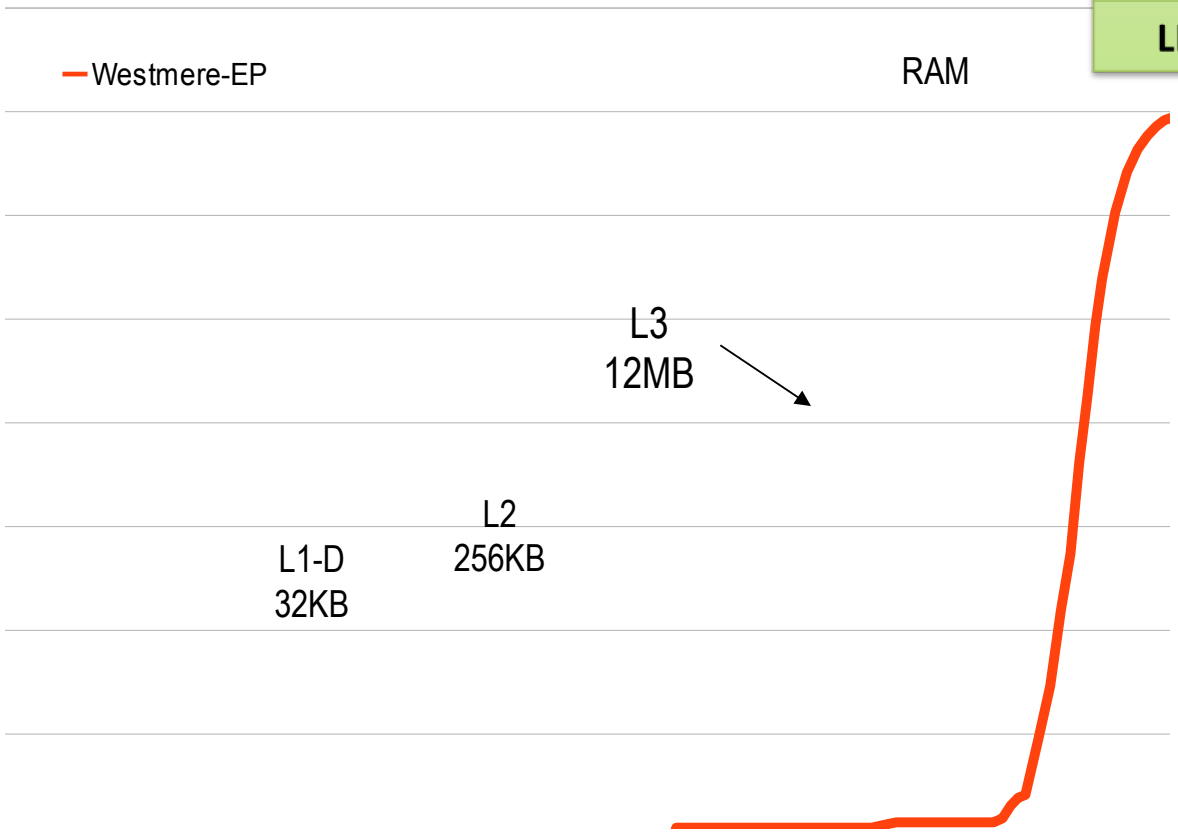
- http://www.bitmover.com/lmbench/get_lmbench.html
- Latence d'accès (en nanosecondes) aux données au travers de la hiérarchie de cache
- Portage
  - C
  - Makefile, immédiat
- Execution
  - Simple

lat_mem_rd (LMBench)

- LMBench3.0
- parametres:

```
-N 1 -P 1 256M 512
```

Cache level	Latencies	
	NHM	WST
L1-D	4 cycles	4 cycles
MLC	10 cycles	10 cycles
LLC	38 cycles	43 cycles



Contenu de la boîte à outils « mémoire »

- Hiérarchie :
  - Cachebench (llcbench)
  - mem_lat_rd (LMBench)
- Latence d'accès :
  - mem_lat_rd
  - HPCC randomAccess (random)
- Débit :
  - Stream

    Permet de détecter d'enventuels problèmes de mémoire
- Autres :
  - Memtest
  - Surpport de mcelog (kernel)