TECHNIQUES MODERNES DE PROGRAMMATION CONCURRENTE

COURS 1
ARCHITECTURES MULTICŒUR

Par Jean-Pierre Lozi
Basé sur les cours de
Gaël Thomas

MODULE « TECHNIQUES MODERNES DE PROGRAMMATION CONCURRENTE »

CE MODULE

- Techniques Modernes de Programmation Concurrente

CE MODULE

- Techniques Modernes de Programmation Concurrente
- *Première mouture, tout est sujet à modifications !*

CE MODULE

- Techniques Modernes de Programmation Concurrente
- *Première mouture, tout est sujet à modifications !*
- 7 séances, les mardis après-midi, de 14h00 jusqu'à 17h45 au plus tard

CE MODULE

- **Techniques Modernes de Programmation Concurrente**
- *Première mouture, tout est sujet à modifications !*
- **7 séances, les mardis après-midi, de 14h00 jusqu'à 17h45 au plus tard**
- **Première partie (4 séances) :** Jean-Pierre Lozi (moi-même)
  « Exploiter la puissance des architectures multicœur »

CE MODULE

- **Techniques Modernes de Programmation Concurrente**
- *Première mouture, tout est sujet à modifications !*
- **7 séances, les mardis après-midi, de 14h00 jusqu'à 17h45 au plus tard**
- **Première partie (4 séances) :** Jean-Pierre Lozi (moi-même)
  « Exploiter la puissance des architectures multicœur »
  - ***Séance 1 (20/09) :*** 1h30 de cours sur les architectures multicœur + 2h TP

CE MODULE

- **Techniques Modernes de Programmation Concurrente**
- *Première mouture, tout est sujet à modifications !*
- **7 séances, les mardis après-midi, de 14h00 jusqu'à 17h45 au plus tard**
- **Première partie (4 séances) :** Jean-Pierre Lozi (moi-même)
  « Exploiter la puissance des architectures multicœur »
  - ***Séance 1 (20/09) :*** 1h30 de cours sur les architectures multicœur + 2h TP
  - ***Séance 2 (27/09) :*** 1h30 de cours sur les algorithmes de verrouillage + 2h TP

CE MODULE

- **Techniques Modernes de Programmation Concurrente**
- *Première mouture, tout est sujet à modifications !*
- **7 séances, les mardis après-midi, de 14h00 jusqu'à 17h45 au plus tard**
- **Première partie (4 séances) :** Jean-Pierre Lozi (moi-même)
  « Exploiter la puissance des architectures multicœur »
  - ***Séance 1 (20/09) :*** 1h30 de cours sur les architectures multicœur + 2h TP
  - ***Séance 2 (27/09) :*** 1h30 de cours sur les algorithmes de verrouillage + 2h TP
  - ***Séance 3 (11/10) :*** 1h30 de cours sur l'algorithmique non bloquante + 2h TP

CE MODULE

- **Techniques Modernes de Programmation Concurrente**
- *Première mouture, tout est sujet à modifications !*
- **7 séances, les mardis après-midi, de 14h00 jusqu'à 17h45 au plus tard**
- **Première partie (4 séances) :** Jean-Pierre Lozi (moi-même)
  « Exploiter la puissance des architectures multicœur »
  - ***Séance 1 (20/09) :*** 1h30 de cours sur les architectures multicœur + 2h TP
  - ***Séance 2 (27/09) :*** 1h30 de cours sur les algorithmes de verrouillage + 2h TP
  - ***Séance 3 (11/10) :*** 1h30 de cours sur l'algorithmique non bloquante + 2h TP
  - ***Séance 4 (18/10) :*** 45 minutes de présentation sur les mémoires transactionnelles + 45 minutes de présentation de travaux de recherche sur le scheduler Linux + 3h TP

CE MODULE

- **Deuxième partie (3 séances) :** Erick Gallesio

  « Outils de haut niveau pour programmation multi-cœurs »

CE MODULE

- **Deuxième partie (3 séances) :** Erick Gallesio

  « Outils de haut niveau pour programmation multi-cœurs »

  - Clojure, Go ? Lui demander pour plus de détails ☺

CE MODULE

- **Deuxième partie (3 séances) :** Erick Gallesio

  « Outils de haut niveau pour programmation multi-cœurs »

  - Clojure, Go ? Lui demander pour plus de détails ☺

- **Evaluation :**
  - 50% de la note = certains TPs seront relevés (tous?)
  - 50% de la note = examen final le 15/11 (2h)

CE MODULE

- **Deuxième partie (3 séances) :** Erick Gallesio

  « Outils de haut niveau pour programmation multi-cœurs »

  - Clojure, Go ? Lui demander pour plus de détails ☺

- **Evaluation :**
  - 50% de la note = certains TPs seront relevés (tous?)
  - 50% de la note = examen final le 15/11 (2h)

- **ECTS :** 2

CE MODULE

- **Deuxième partie (3 séances) :** Erick Gallesio

« Outils de haut niveau pour programmation multi-cœurs »

  - Clojure, Go ? Lui demander pour plus de détails ☺

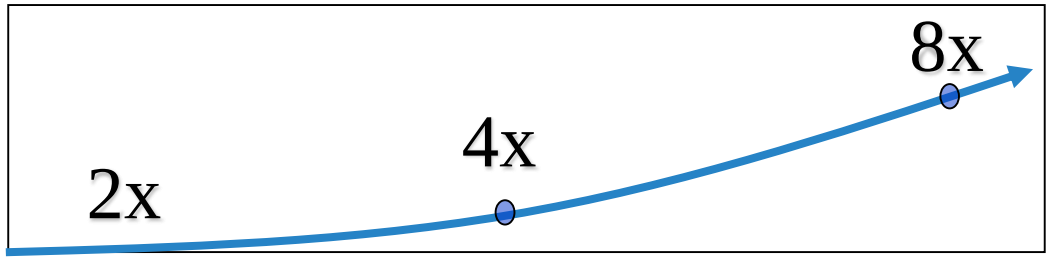
- **Evaluation :**
  - 50% de la note = certains TPs seront relevés (tous?)
  - 50% de la note = examen final le 15/11 (2h)
- **ECTS :** 2
- **Prérequis :**
  - Savoir programmer en C et Java
  - Avoir quelques notions de système et d'architecture…

ARCHITECTURES MULTICŒUR

ÉVOLUTION DES MACHINES MONO-CŒUR

Accélération : suit une « fausse » loi de Moore

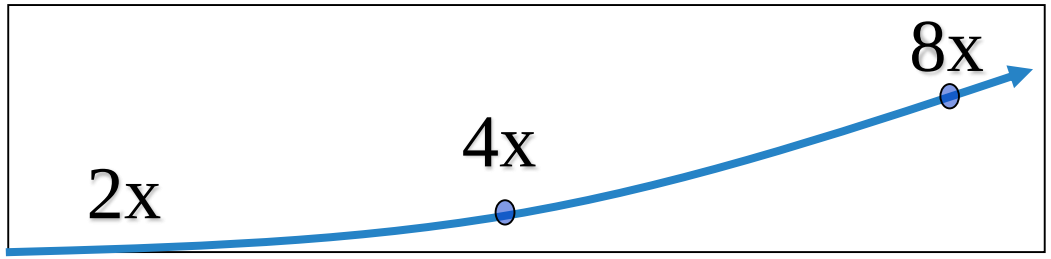
Performances « doublent » tous les 18 mois ?



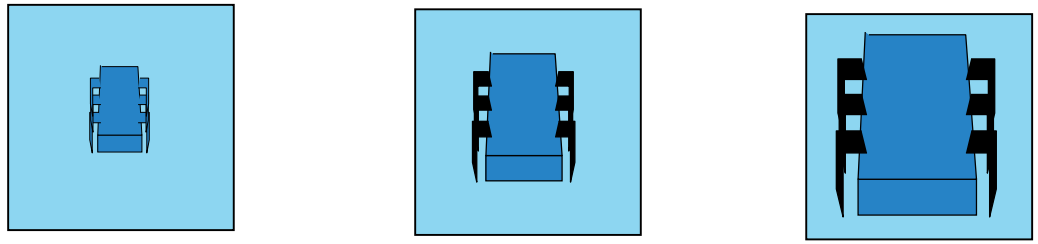
Temps

ÉVOLUTION DES MACHINES MONO-CŒUR

Accélération : suit une « fausse » loi de Moore
Performances « doublent » tous les 18 mois ?



Évolution traditionnelle des mono-processeurs :
Circuits de plus en plus « gros/complexes » (de plus en plus de transistors) gravés sur support de la même taille
« Vraie » loi de Moore : nb transistors double tous les 2 ans



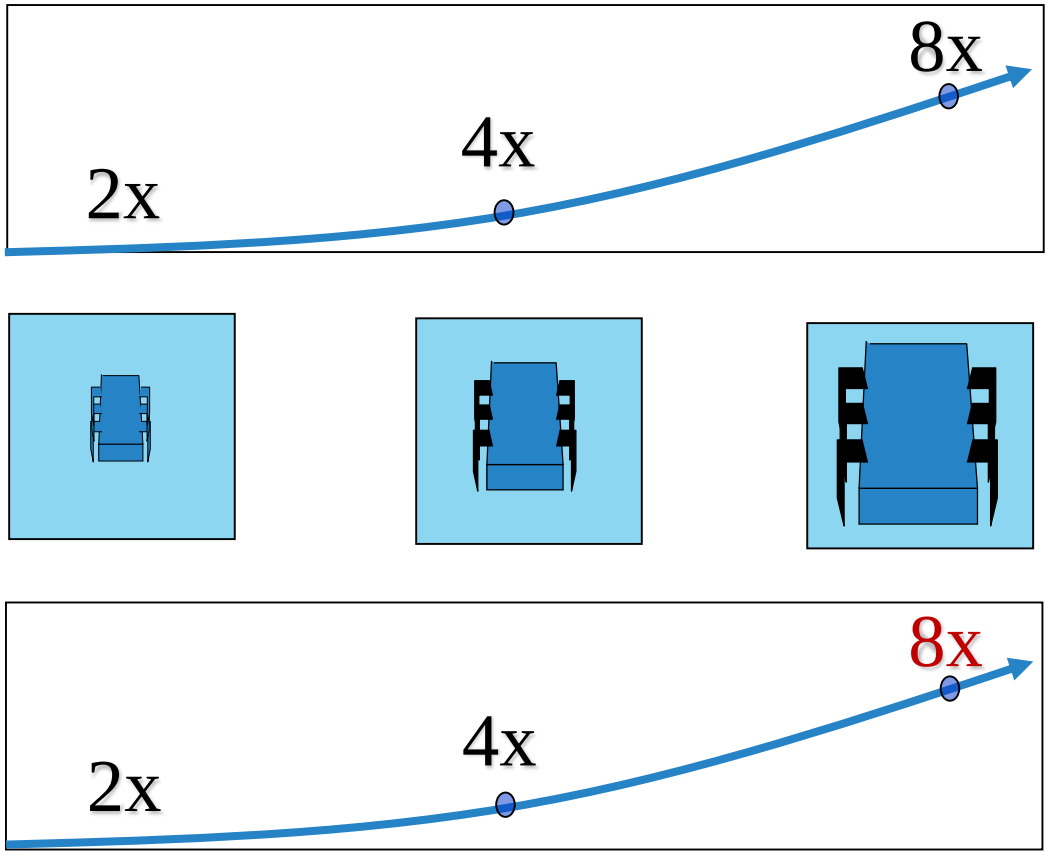
Temps

ÉVOLUTION DES MACHINES MONO-CŒUR

Accélération : suit une « fausse » loi de Moore
Performances « doublent » tous les 18 mois ?

Évolution traditionnelle des mono-processeurs :
Circuits de plus en plus « gros/complexes » (de plus en plus de transistors) gravés sur support de la même taille
« Vraie » loi de Moore : nb transistors double tous les 2 ans

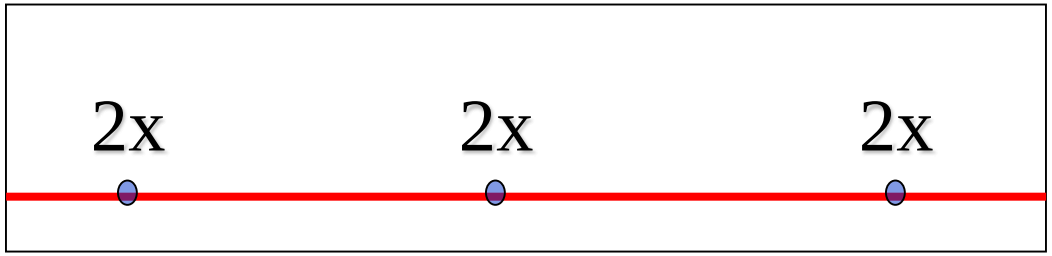
Consommation énergétique / chauffe:
Double aussi tous les 18 mois !
Proportionnel à la fréquence…
Atteint des niveaux problématiques dès les années 2000 !



ÉVOLUTION DES MACHINES MULTICŒUR

Accélération par cœur : pratiquement constante

En pratique augmente encore un peu quand même…

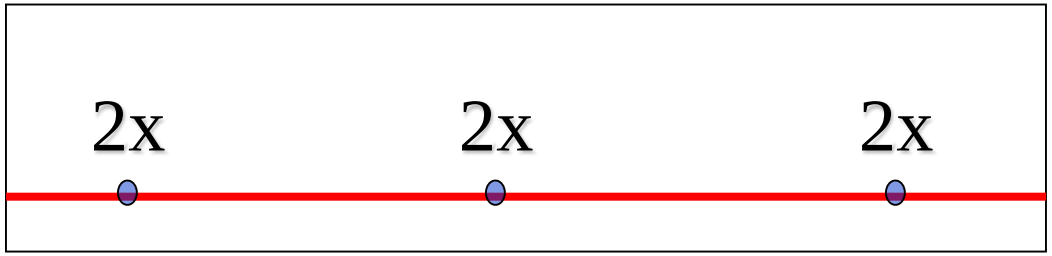


Temps

ÉVOLUTION DES MACHINES MULTICŒUR

Accélération par cœur : pratiquement constante

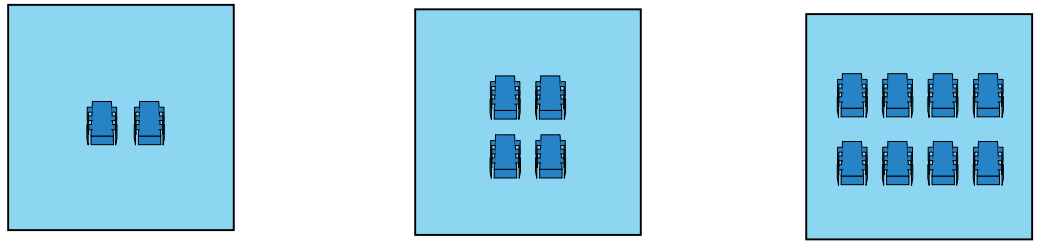
En pratique augmente encore un peu quand même…



Évolution actuelle des processeurs:

De plus en plus de cœurs dans un processeur

Transistors « en plus » utilisés pour de nouveaux cœurs!



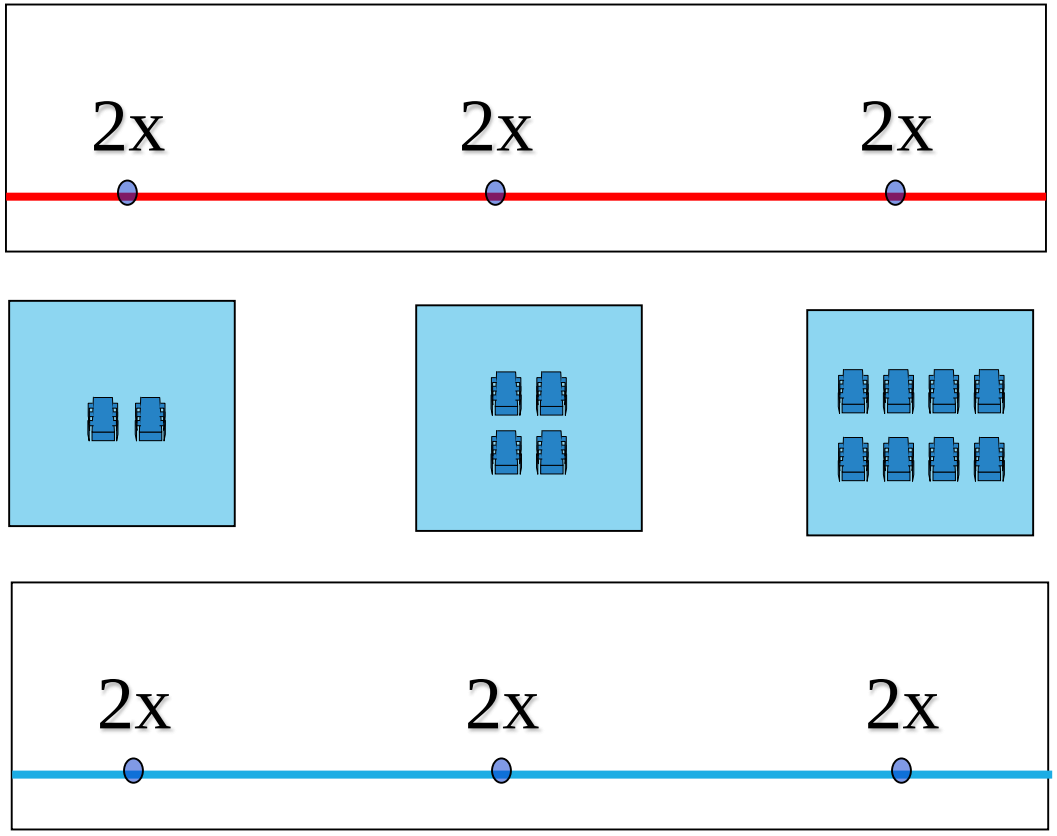
Temps

ÉVOLUTION DES MACHINES MULTICŒUR

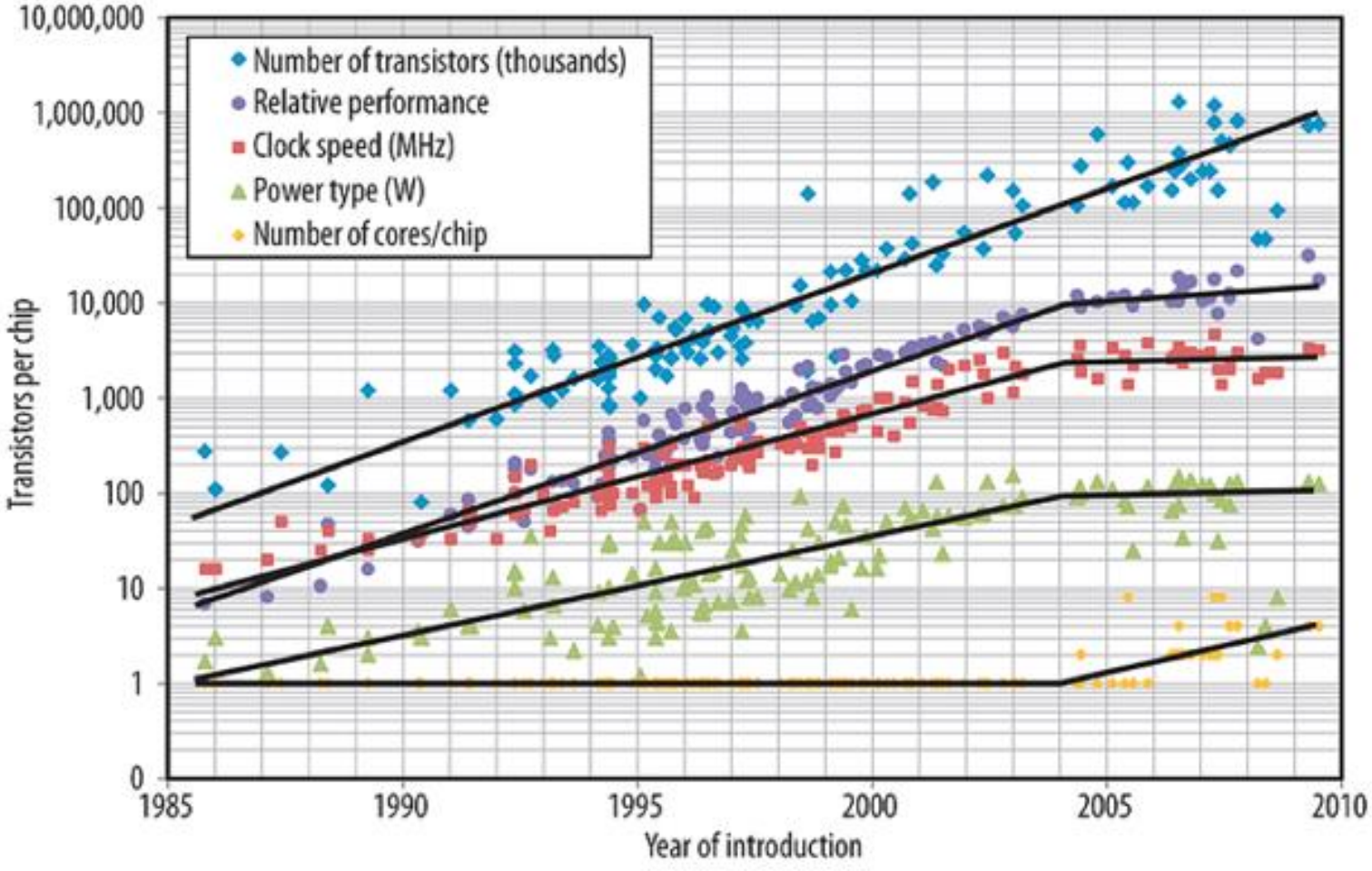
Accélération par cœur : pratiquement constante
En pratique augmente encore un peu quand même…

Évolution actuelle des processeurs:
De plus en plus de cœurs dans un processeur
Transistors « en plus » utilisés pour de nouveaux cœurs!

Consommation énergétique / chauffe:
À peu près constante… À chaque fois, on alimente deux fois plus de cœurs deux fois plus petits, à fréquence constante!



ÉVOLUTION DES MACHINES MULTICŒUR



CHALLENGES ET OPTIMISATIONS

- Problème principal : faire passer les programmes à l'échelle !

CHALLENGES ET OPTIMISATIONS

- Problème principal : faire passer les programmes à l'échelle !
  - On met 10 cœurs au lieu d'un, puissance théorique multipliée par 10 !

CHALLENGES ET OPTIMISATIONS

- Problème principal : faire passer les programmes à l'échelle !
  - On met 10 cœurs au lieu d'un, puissance théorique multipliée par 10 !
  - En pratique, programmes ne passent pas à l'échelle !

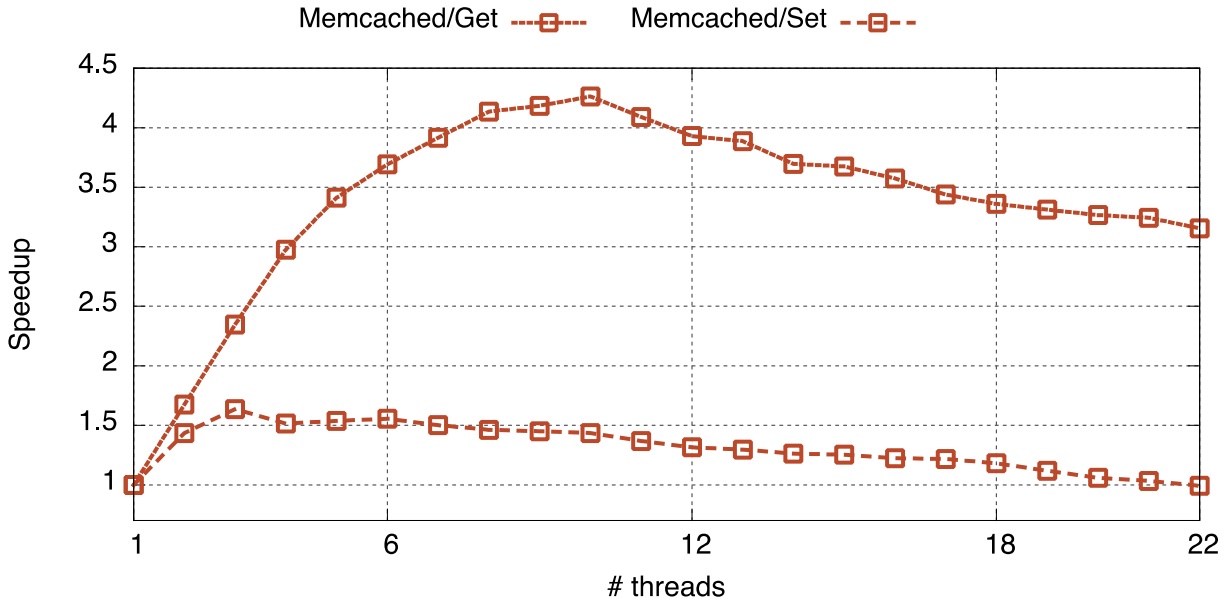
CHALLENGES ET OPTIMISATIONS

- Problème principal : faire passer les programmes à l'échelle !
  - On met 10 cœurs au lieu d'un, puissance théorique multipliée par 10 !
  - En pratique, programmes ne passent pas à l'échelle !
  - Perfs augmentent avec le nombre de cœurs, puis stagnent ou s'effondrent…

CHALLENGES ET OPTIMISATIONS

- Problème principal : faire passer les programmes à l'échelle !
  - On met 10 cœurs au lieu d'un, puissance théorique multipliée par 10 !
  - En pratique, programmes ne passent pas à l'échelle !
  - Perfs augmentent avec le nombre de cœurs, puis stagnent ou s'effondrent…

- Courbe de scalabilité classique :



CHALLENGES ET OPTIMISATIONS

- Problème principal : faire passer les programmes à l'échelle !
  - **Remplacez une personne par un robot 100 fois plus rapide :** tâche effectuée 100x plus vite

CHALLENGES ET OPTIMISATIONS

- Problème principal : faire passer les programmes à l'échelle !
  - **Remplacez une personne par un robot 100 fois plus rapide :** tâche effectuée 100x plus vite
  - **Remplacez une personne par 100 personnes :** répartition des tâches, communications, gestion…
    - *Enorme overhead !*

CHALLENGES ET OPTIMISATIONS

- **Problème principal : faire passer les programmes à l'échelle !**
  - **Remplacez une personne par un robot 100 fois plus rapide :** tâche effectuée 100x plus vite
  - **Remplacez une personne par 100 personnes :** répartition des tâches, communications, gestion…
    - *Enorme overhead !*
  - **En pratique, scalabilité pendant un moment jusqu'à atteinte d'un bottleneck !**
    - Forte contention sur un verrou/une ressource par exemple.

CHALLENGES ET OPTIMISATIONS

- **Problème principal : faire passer les programmes à l'échelle !**
  - **Remplacez une personne par un robot 100 fois plus rapide :** tâche effectuée 100x plus vite
  - **Remplacez une personne par 100 personnes :** répartition des tâches, communications, gestion…
    - *Enorme overhead !*
  - **En pratique, scalabilité pendant un moment jusqu'à atteinte d'un bottleneck !**
    - Forte contention sur un verrou/une ressource par exemple.
- **Comment optimiser ?**
  - **Améliorer la synchronisation, e.g. :**

CHALLENGES ET OPTIMISATIONS

- **Problème principal : faire passer les programmes à l'échelle !**
  - **Remplacez une personne par un robot 100 fois plus rapide :** tâche effectuée 100x plus vite
  - **Remplacez une personne par 100 personnes :** répartition des tâches, communications, gestion…
    - *Enorme overhead !*
  - **En pratique, scalabilité pendant un moment jusqu'à atteinte d'un bottleneck !**
    - Forte contention sur un verrou/une ressource par exemple.
- **Comment optimiser ?**
  - **Améliorer la synchronisation, e.g. :**
    - Réduire la taille des sections critiques

CHALLENGES ET OPTIMISATIONS

- **Problème principal : faire passer les programmes à l'échelle !**
  - **Remplacez une personne par un robot 100 fois plus rapide :** tâche effectuée 100x plus vite
  - **Remplacez une personne par 100 personnes :** répartition des tâches, communications, gestion…
    - *Enorme overhead !*
  - **En pratique, scalabilité pendant un moment jusqu'à atteinte d'un bottleneck !**
    - Forte contention sur un verrou/une ressource par exemple.
- **Comment optimiser ?**
  - **Améliorer la synchronisation, e.g. :**
    - Réduire la taille des sections critiques
    - Utiliser des verrous plus réactifs

CHALLENGES ET OPTIMISATIONS

- **Problème principal : faire passer les programmes à l'échelle !**
  - **Remplacez une personne par un robot 100 fois plus rapide :** tâche effectuée 100x plus vite
  - **Remplacez une personne par 100 personnes :** répartition des tâches, communications, gestion…
    - *Enorme overhead !*
  - **En pratique, scalabilité pendant un moment jusqu'à atteinte d'un bottleneck !**
    - Forte contention sur un verrou/une ressource par exemple.
- **Comment optimiser ?**
  - **Améliorer la synchronisation, e.g. :**
    - Réduire la taille des sections critiques
    - Utiliser des verrous plus réactifs
    - Supprimer les verrous entièrement ! (algorithmique non bloquante)

CHALLENGES ET OPTIMISATIONS

- Comment optimiser ?
  - **Améliorer la localité des données :** threads travaillant sur mêmes données plus rapides si partagent mêmes caches de bas niveau, de haut niveau, mêmes bancs mémoire, etc.

CHALLENGES ET OPTIMISATIONS

- Comment optimiser ?
  - **Améliorer la localité des données :** threads travaillant sur mêmes données plus rapides si partagent mêmes caches de bas niveau, de haut niveau, mêmes bancs mémoire, etc.
    - Limiter partages/accès mémoire inutiles ou redondants
    - Migrations/réplications mémoire, placement des threads

CHALLENGES ET OPTIMISATIONS

- Comment optimiser ?
  - **Améliorer la localité des données :** threads travaillant sur mêmes données plus rapides si partagent mêmes caches de bas niveau, de haut niveau, mêmes bancs mémoire, etc.
    - Limiter partages/accès mémoire inutiles ou redondants
    - Migrations/réplications mémoire, placement des threads
  - **Améliorer l'ordonnancement :** au niveau du noyau, peu de contrôle

CHALLENGES ET OPTIMISATIONS

- Comment optimiser ?
  - **Améliorer la localité des données :** threads travaillant sur mêmes données plus rapides si partagent mêmes caches de bas niveau, de haut niveau, mêmes bancs mémoire, etc.
    - Limiter partages/accès mémoire inutiles ou redondants
    - Migrations/réplications mémoire, placement des threads
  - **Améliorer l'ordonnancement :** au niveau du noyau, peu de contrôle
  - **Supprimer certains « bugs » concurrents :** false sharing, convois avec certains verrous…

CHALLENGES ET OPTIMISATIONS

- Comment optimiser ?
  - **Améliorer la localité des données :** threads travaillant sur mêmes données plus rapides si partagent mêmes caches de bas niveau, de haut niveau, mêmes bancs mémoire, etc.
    - Limiter partages/accès mémoire inutiles ou redondants
    - Migrations/réplications mémoire, placement des threads
  - **Améliorer l'ordonnancement :** au niveau du noyau, peu de contrôle
  - **Supprimer certains « bugs » concurrents :** false sharing, convois avec certains verrous…

- Pas que les perfs !
  - **Problématiques d'énergie** de plus en plus importantes !

CHALLENGES ET OPTIMISATIONS

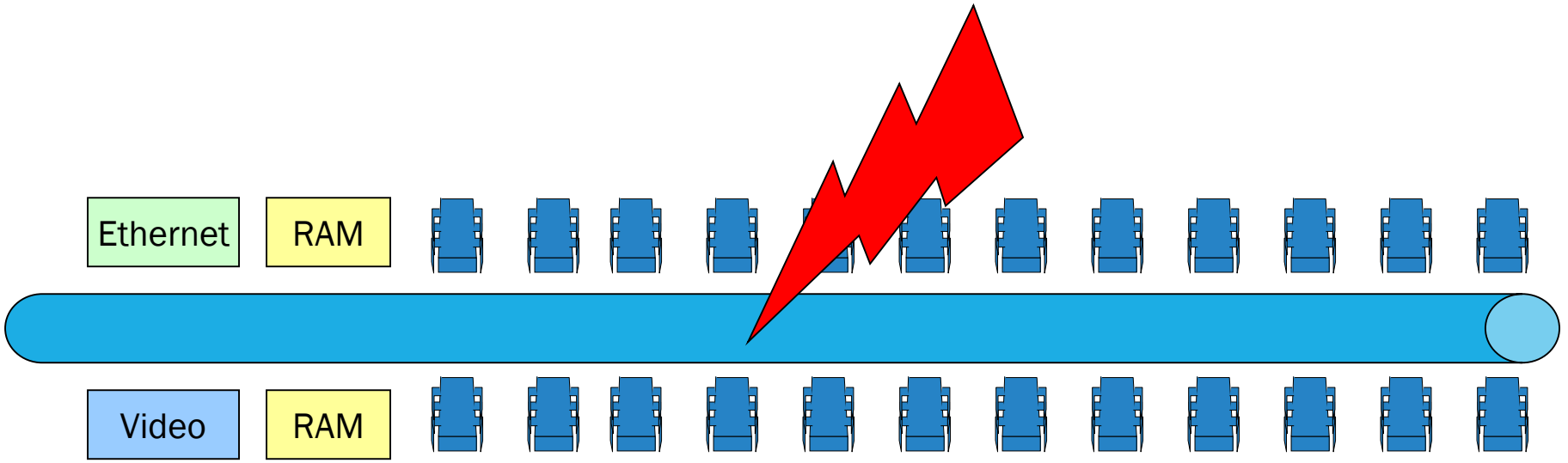
- Comment optimiser ?
  - **Améliorer la localité des données :** threads travaillant sur mêmes données plus rapides si partagent mêmes caches de bas niveau, de haut niveau, mêmes bancs mémoire, etc.
    - Limiter partages/accès mémoire inutiles ou redondants
    - Migrations/réplications mémoire, placement des threads
  - **Améliorer l'ordonnancement :** au niveau du noyau, peu de contrôle
  - **Supprimer certains « bugs » concurrents :** false sharing, convois avec certains verrous…

- Pas que les perfs !
  - **Problématiques d'énergie** de plus en plus importantes !
  - **Et de coût :** gros serveur ou cluster de Raspberry Pi ?

ARCHITECTURE D'UN MULTICŒUR

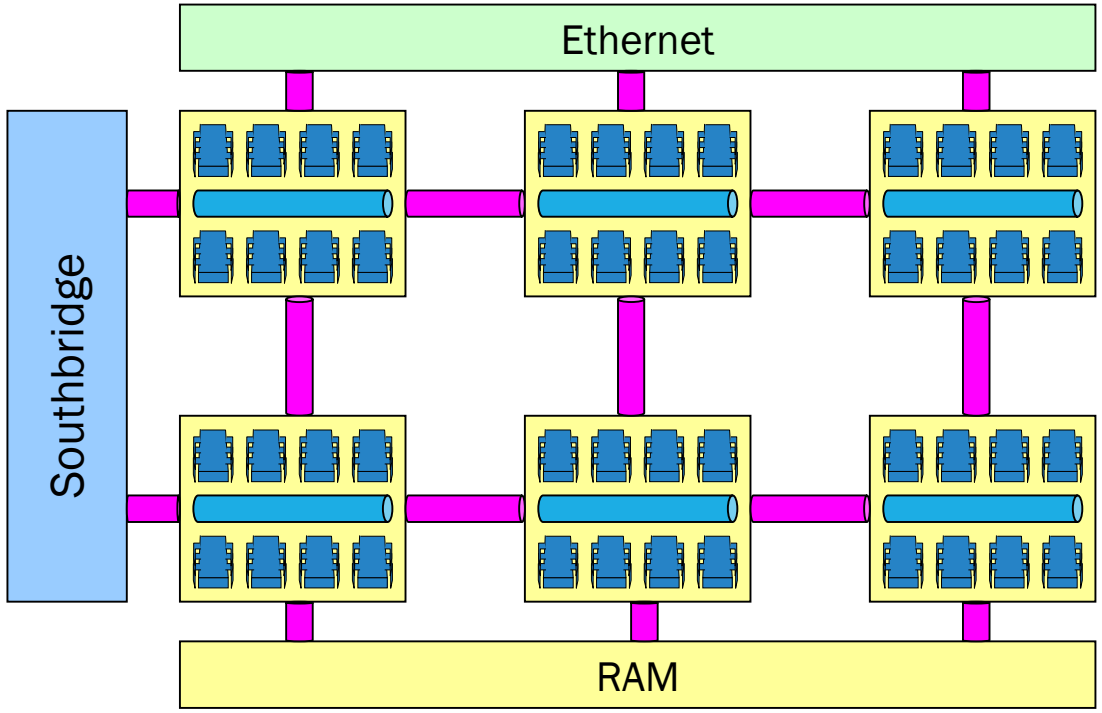
- Impossible d'utiliser une topologie classique à base de bus !

Pas assez de bande passante/trop de collisions !



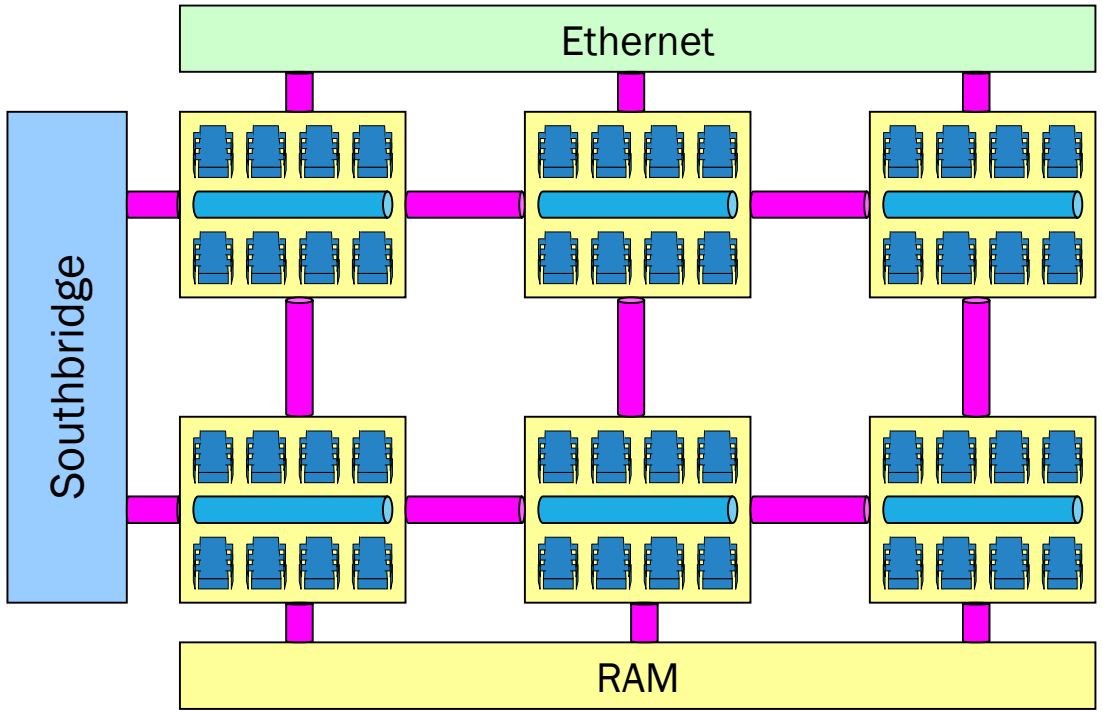
ARCHITECTURE D'UN MULTICŒUR

- Pour passer à l'échelle, il faut changer la topologie…



ARCHITECTURE D'UN MULTICŒUR

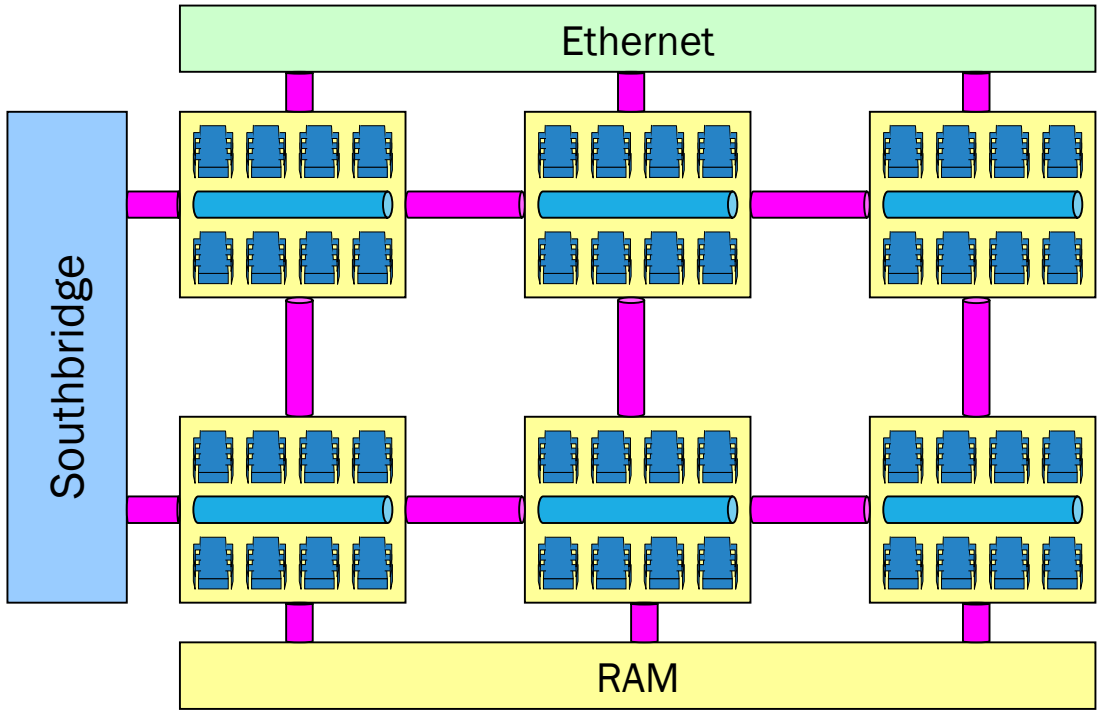
- Pour passer à l'échelle, il faut changer la topologie…



- Au lieu d'un simple bus, on a un graphe

ARCHITECTURE D'UN MULTICŒUR

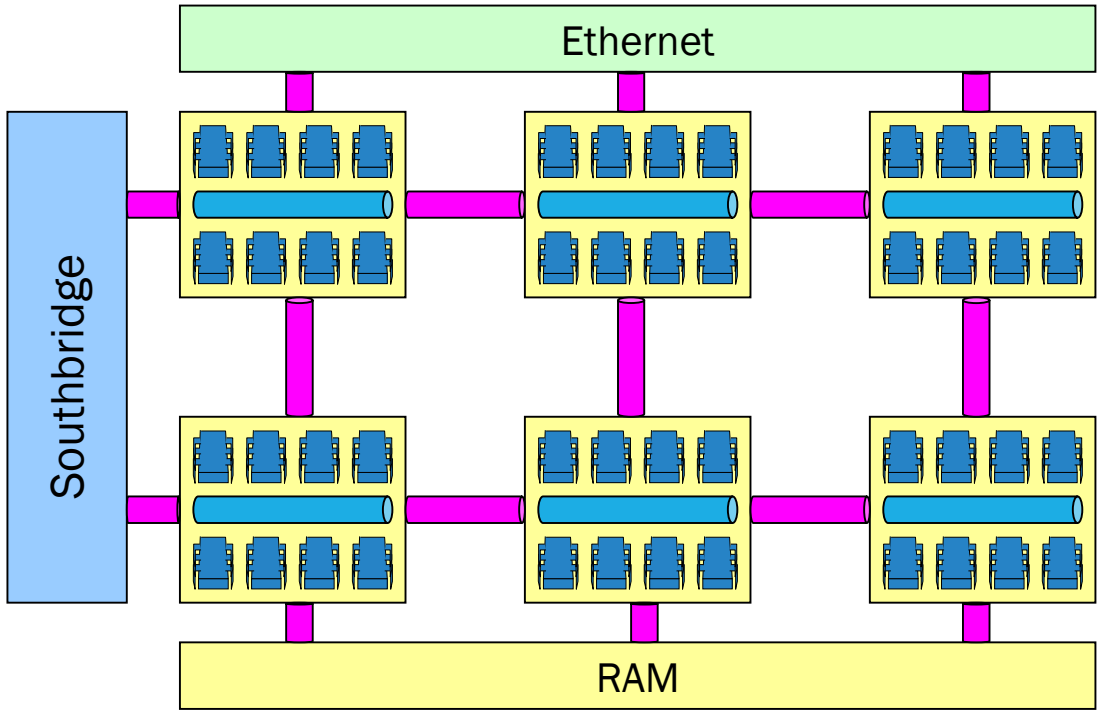
- **Pour passer à l'échelle, il faut changer la topologie…**



- Au lieu d'un simple bus, on a un graphe
- **Pas forcément un graphe complet !**
  - En fait de plus en plus rare…

ARCHITECTURE D'UN MULTICŒUR

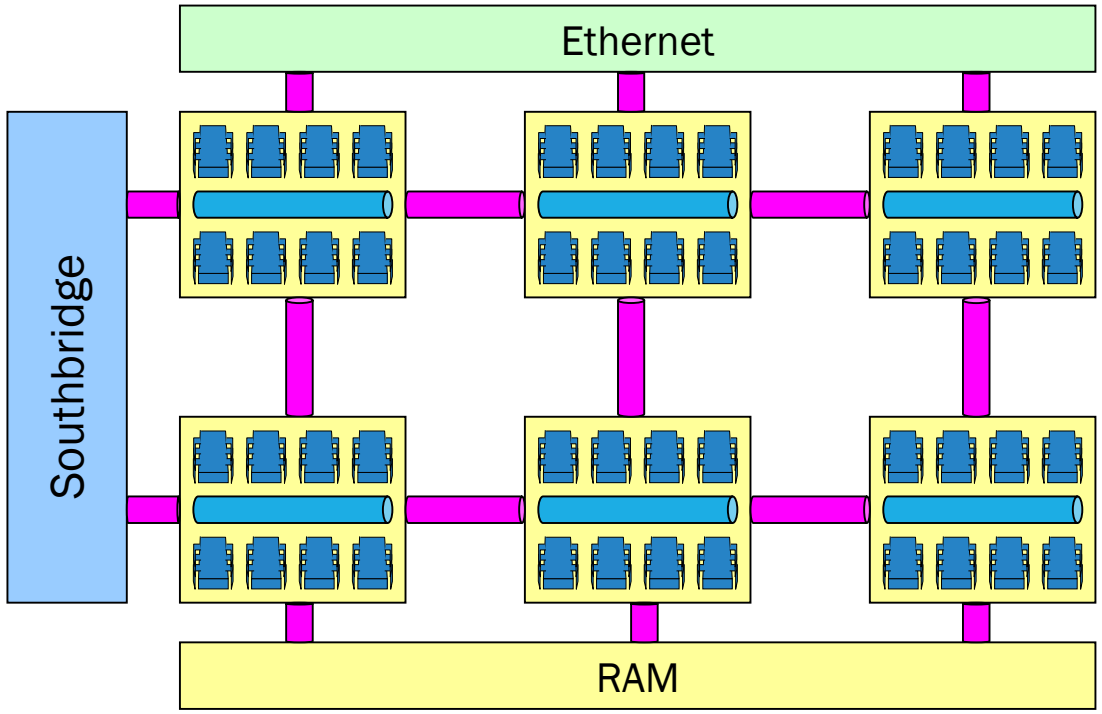
- **Pour passer à l'échelle, il faut changer la topologie…**



- Au lieu d'un simple bus, on a un graphe
- **Pas forcément un graphe complet !**
  - En fait de plus en plus rare…
- **Southbridge :** USB, IDE, BIOS, PCI…

ARCHITECTURE D'UN MULTICŒUR

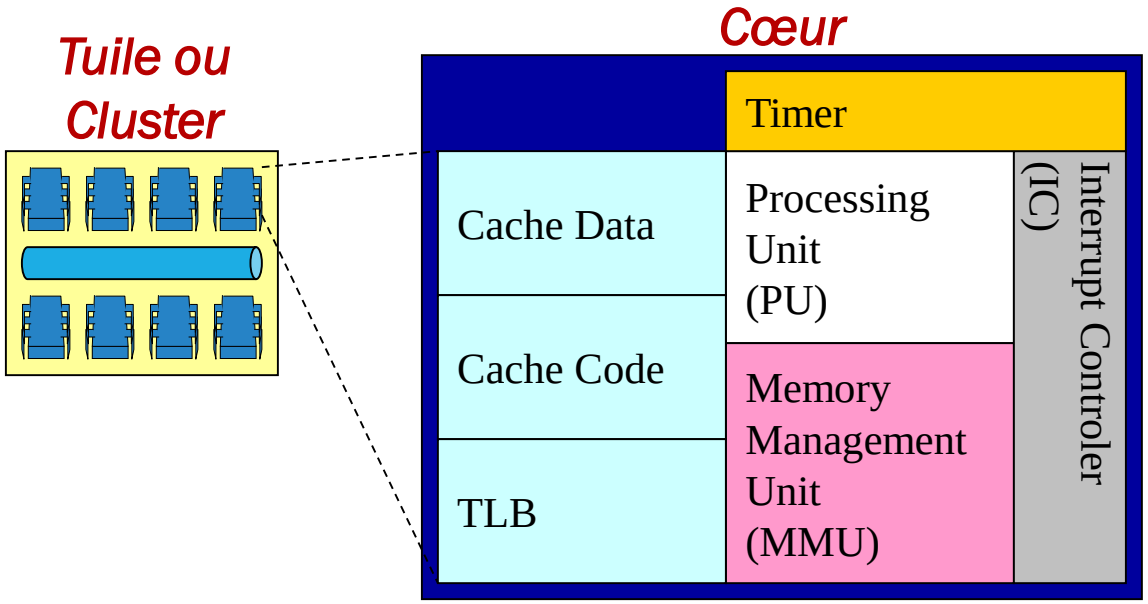
- **Pour passer à l'échelle, il faut changer la topologie…**



- Au lieu d'un simple bus, on a un graphe
- **Pas forcément un graphe complet !**
  - En fait de plus en plus rare…
- **Southbridge :** USB, IDE, BIOS, PCI…
- **Northbridge:** RAM et PCI-Express
  - Tend à disparaitre : controlleurs RAM directement sur processeurs…

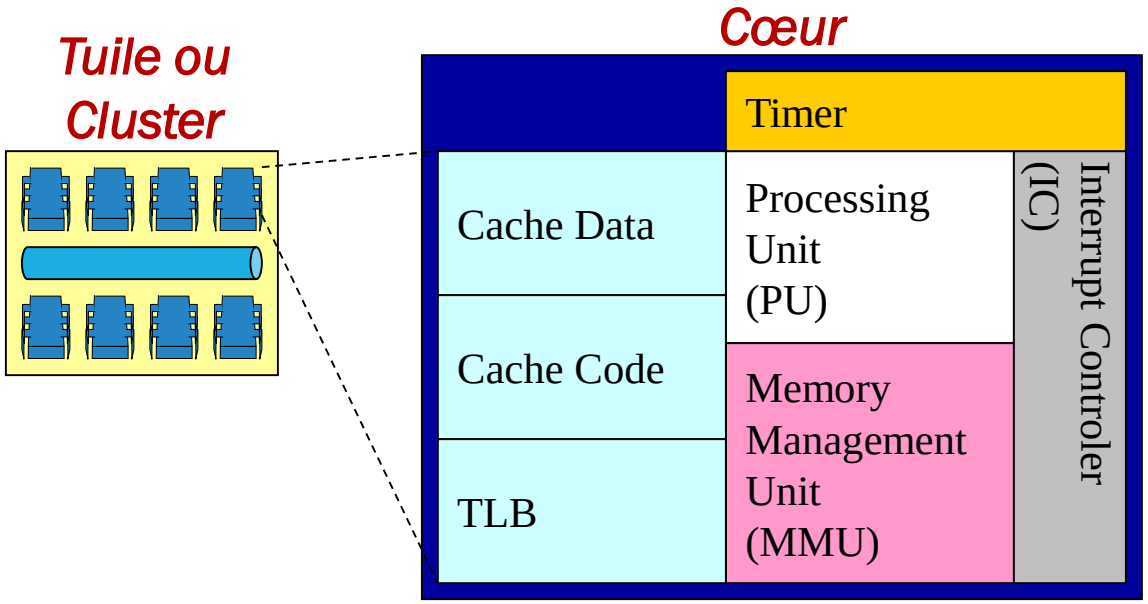
UN PEU DE TERMINOLOGIE…

- Pour passer à l'échelle, il faut changer la topologie…



UN PEU DE TERMINOLOGIE…

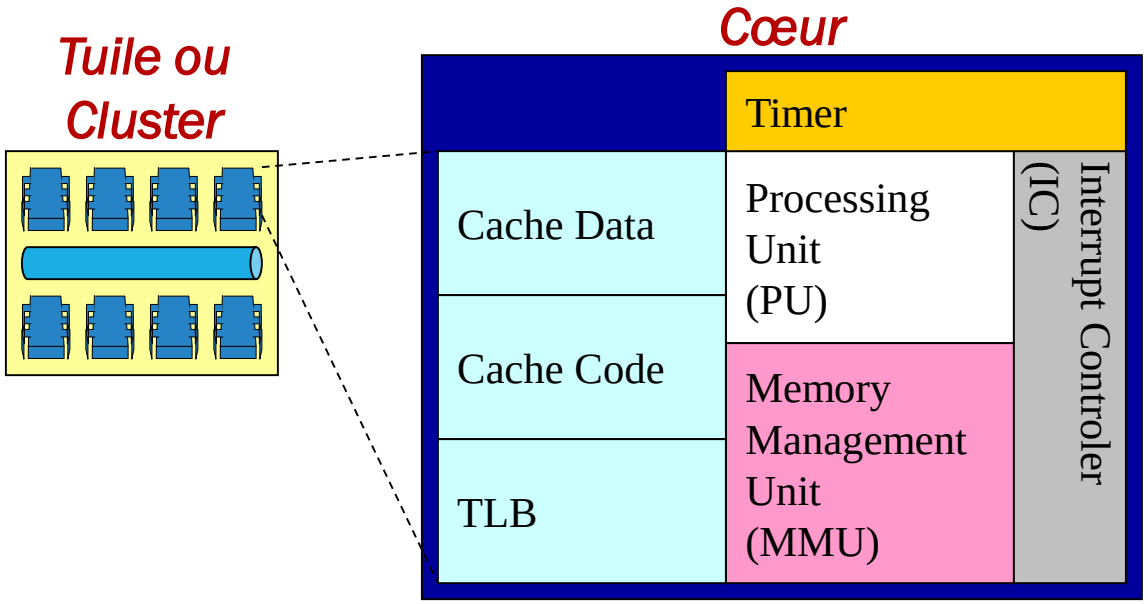
- Pour passer à l'échelle, il faut changer la topologie…



- MMU = contrôle les accès mémoire
  - Traduction d'adresses, protection…

UN PEU DE TERMINOLOGIE…

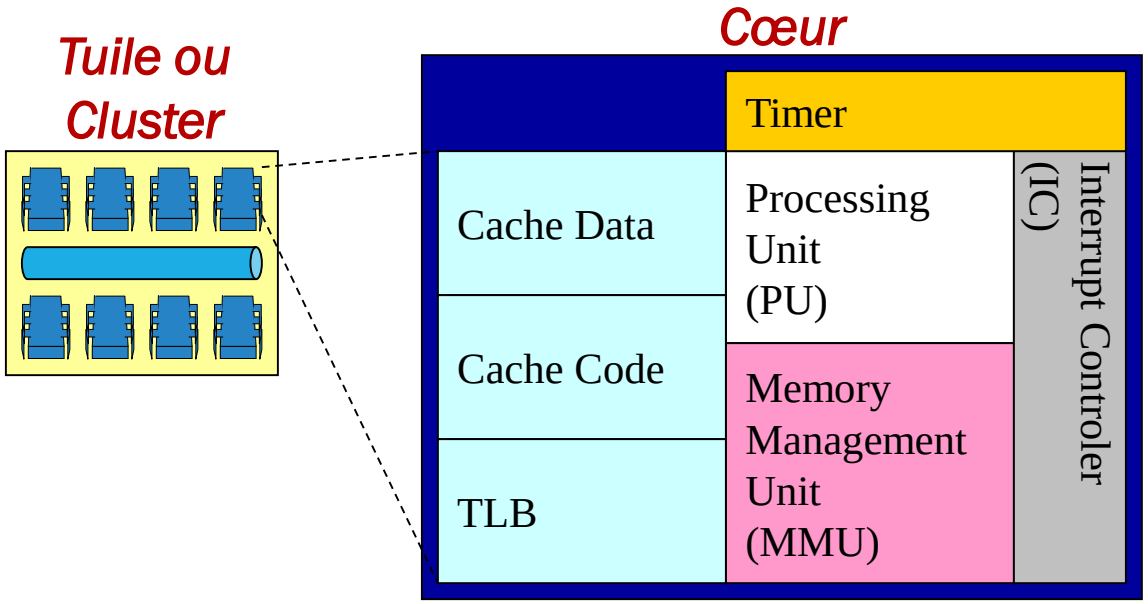
- Pour passer à l'échelle, il faut changer la topologie…



- MMU = contrôle les accès mémoire
  - Traduction d'adresses, protection…
- TLB = Translation Lookaside Buffer
  - Permet la traduction rapide d'adresses virtuelles vers physiques sans consulter la table des pages…

UN PEU DE TERMINOLOGIE…

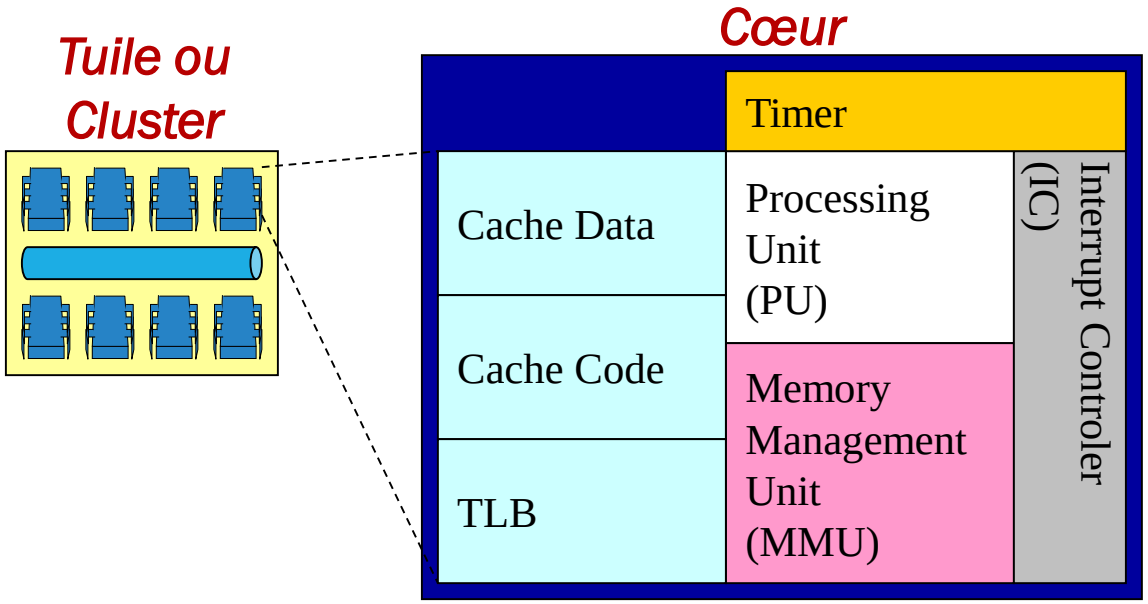
- Pour passer à l'échelle, il faut changer la topologie…



- MMU = contrôle les accès mémoire
  - Traduction d'adresses, protection…
- TLB = Translation Lookaside Buffer
  - Permet la traduction rapide d'adresses virtuelles vers physiques sans consulter la table des pages…
- Cache: parfois divisé entre code et données

UN PEU DE TERMINOLOGIE…

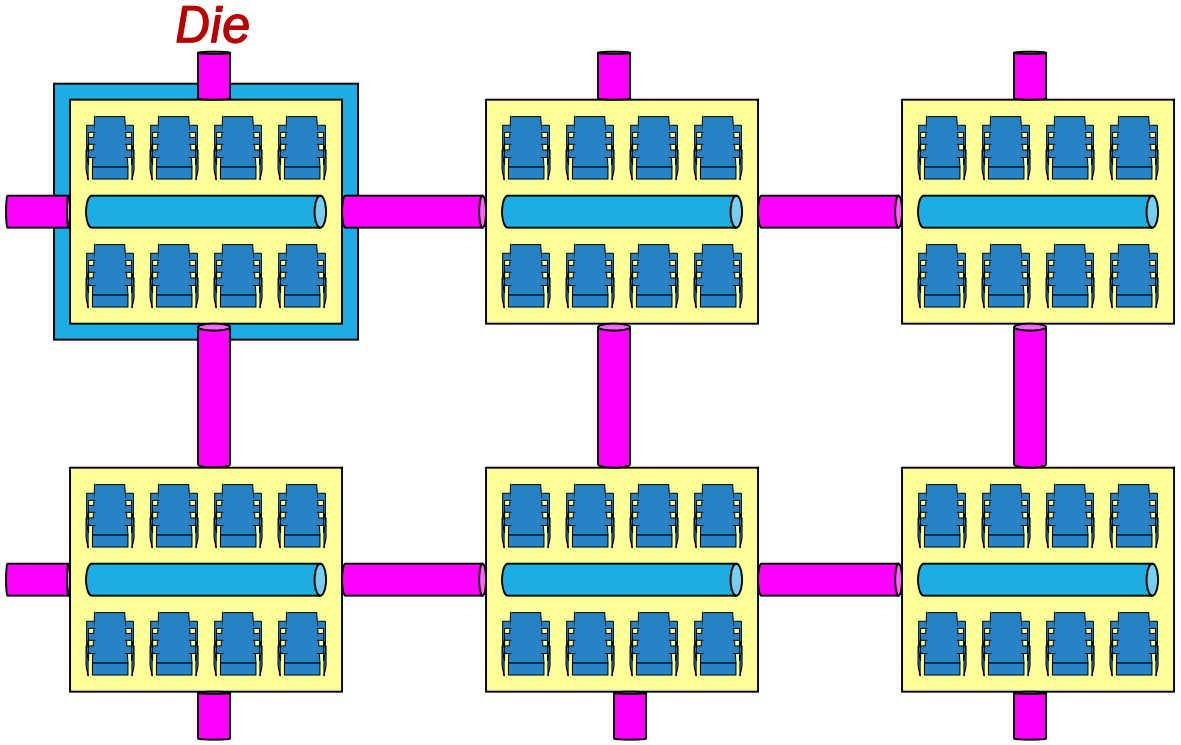
- Pour passer à l'échelle, il faut changer la topologie…



- MMU = contrôle les accès mémoire
  - Traduction d'adresses, protection…
- TLB = Translation Lookaside Buffer
  - Permet la traduction rapide d'adresses virtuelles vers physiques sans consulter la table des pages…
- Cache: parfois divisé entre code et données
- Interrupt Controller: gère les interruptions (system timer, USB, ethernet)…

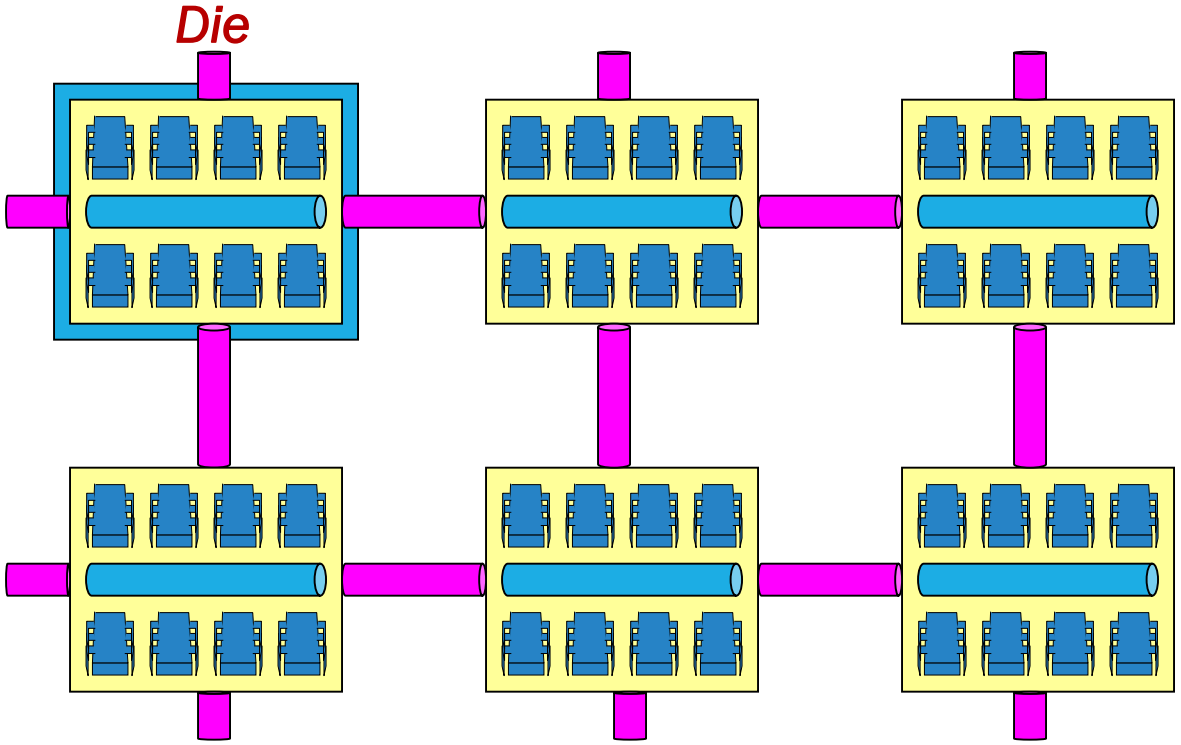
UN PEU DE TERMINOLOGIE…

- **Die:** ensemble de clusters correspondant à un circuit intégré unique



UN PEU DE TERMINOLOGIE…

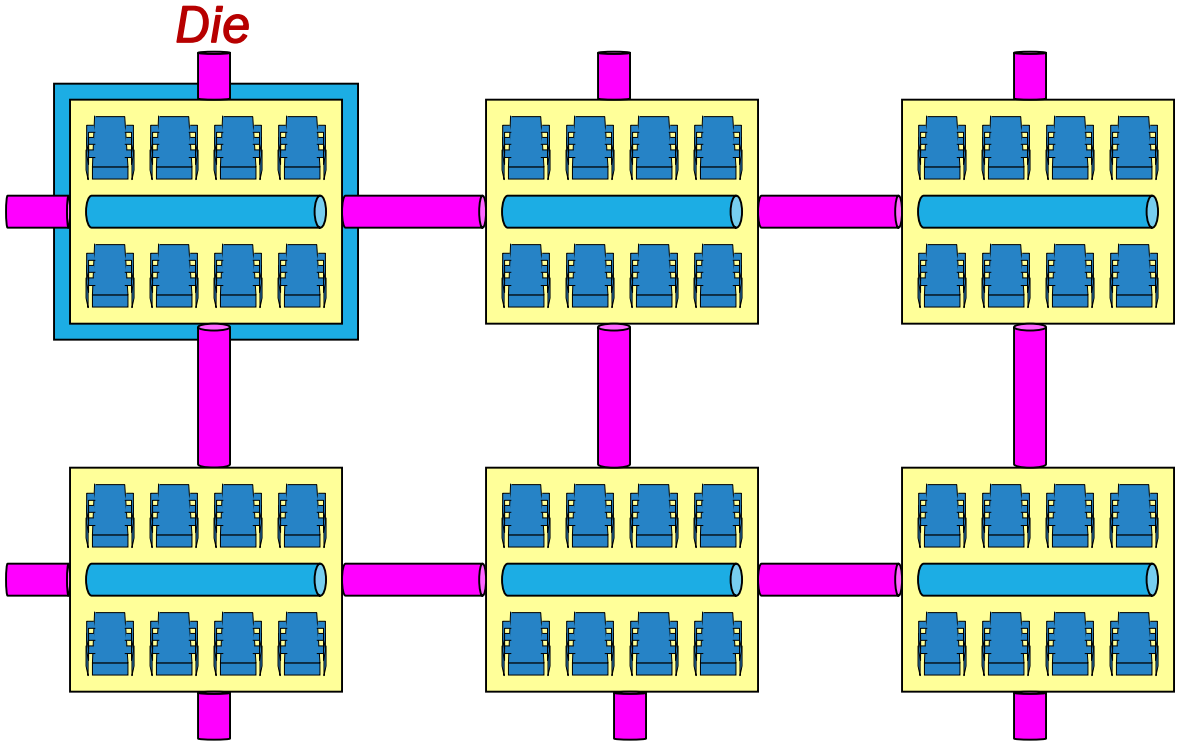
- **Die:** ensemble de clusters correspondant à un circuit intégré unique



- Souvent die = processeur

UN PEU DE TERMINOLOGIE…

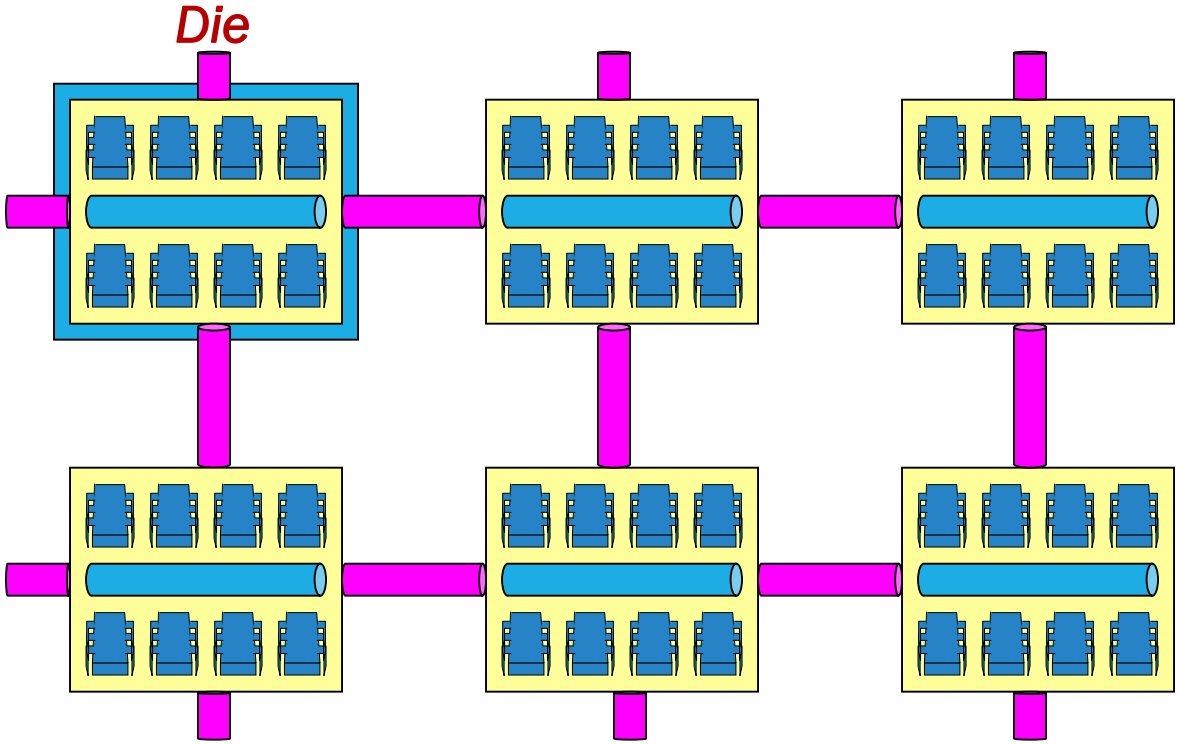
- **Die:** ensemble de clusters correspondant à un circuit intégré unique



- Souvent die = processeur
- **Pas toujours le cas :**
  - E.g. sur une machine à base d'Opterons 6172 à 12 cœurs chacun, chaque CPU a deux dies de 6 cœurs

UN PEU DE TERMINOLOGIE…

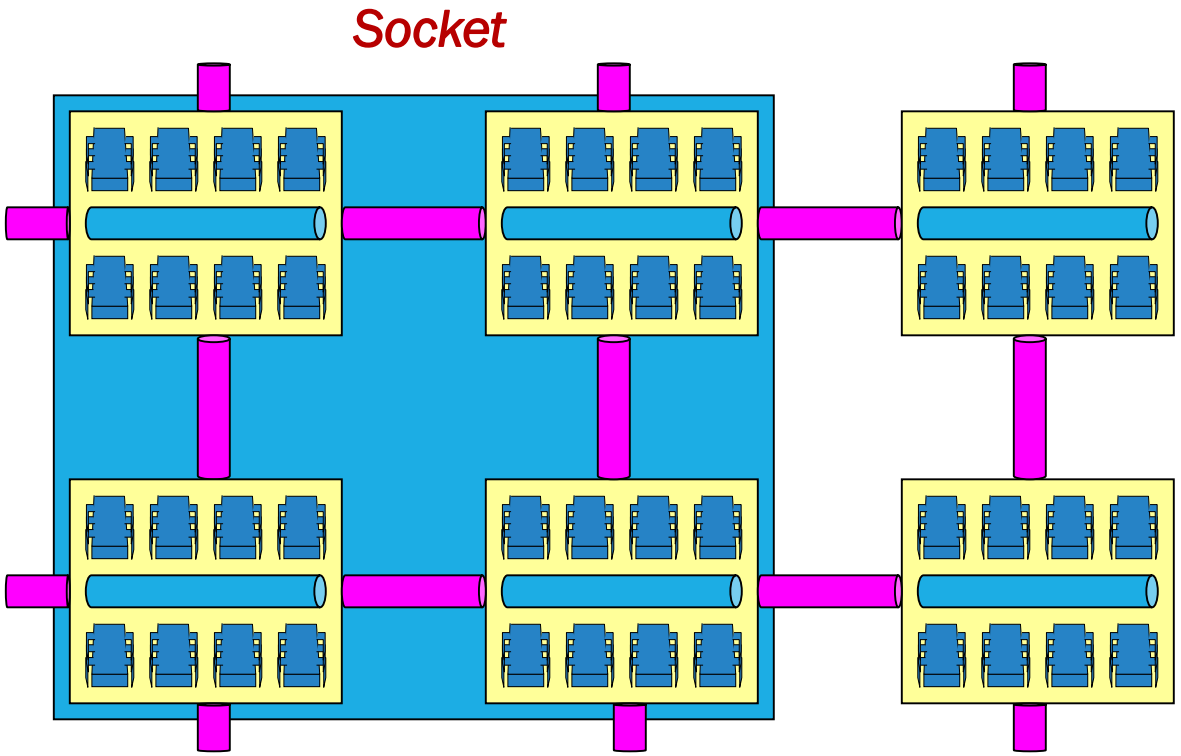
- **Die:** ensemble de clusters correspondant à un circuit intégré unique



- Souvent die = processeur
- **Pas toujours le cas :**
  - E.g. sur une machine à base d'Opterons 6172 à 12 cœurs chacun, chaque CPU a deux dies de 6 cœurs
  - Chaque die a sa propre hiérarchie de cache, son propre contrôlleur mémoire, etc. Fonctionne vraiment comme deux CPUs séparés…

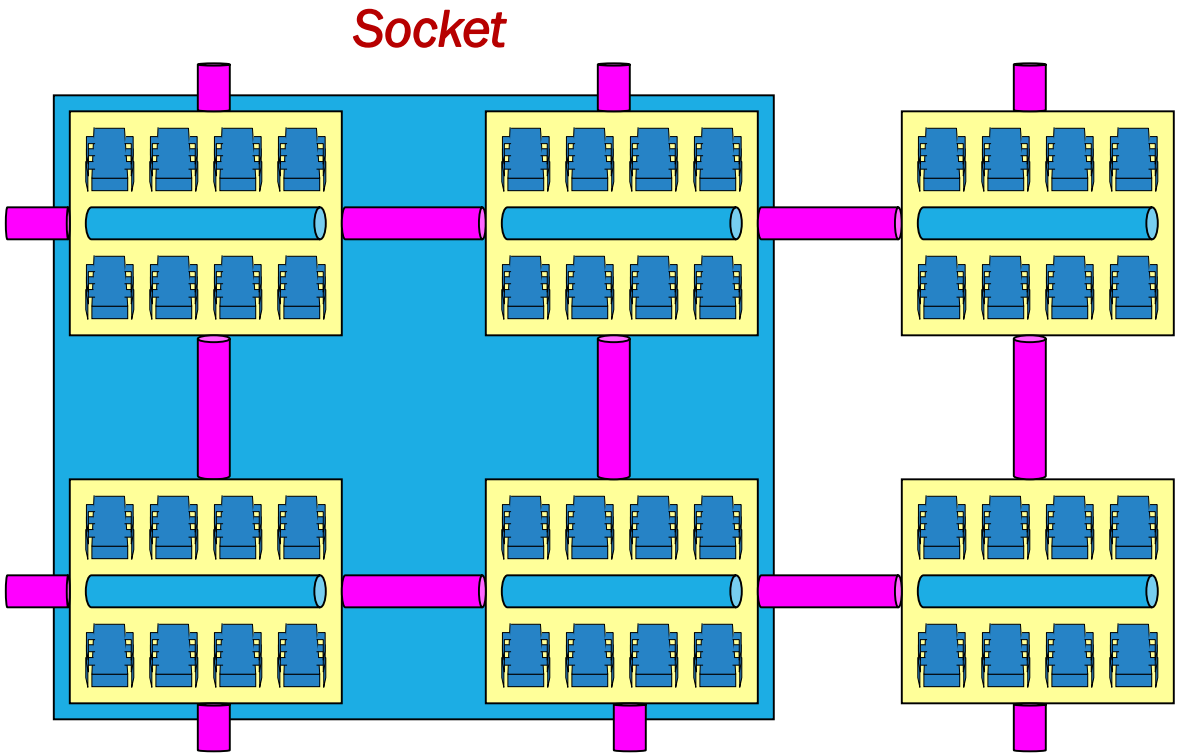
UN PEU DE TERMINOLOGIE…

- **Package/Socket:** ensemble de dies identiques intégrés sur un même support (proco)



UN PEU DE TERMINOLOGIE…

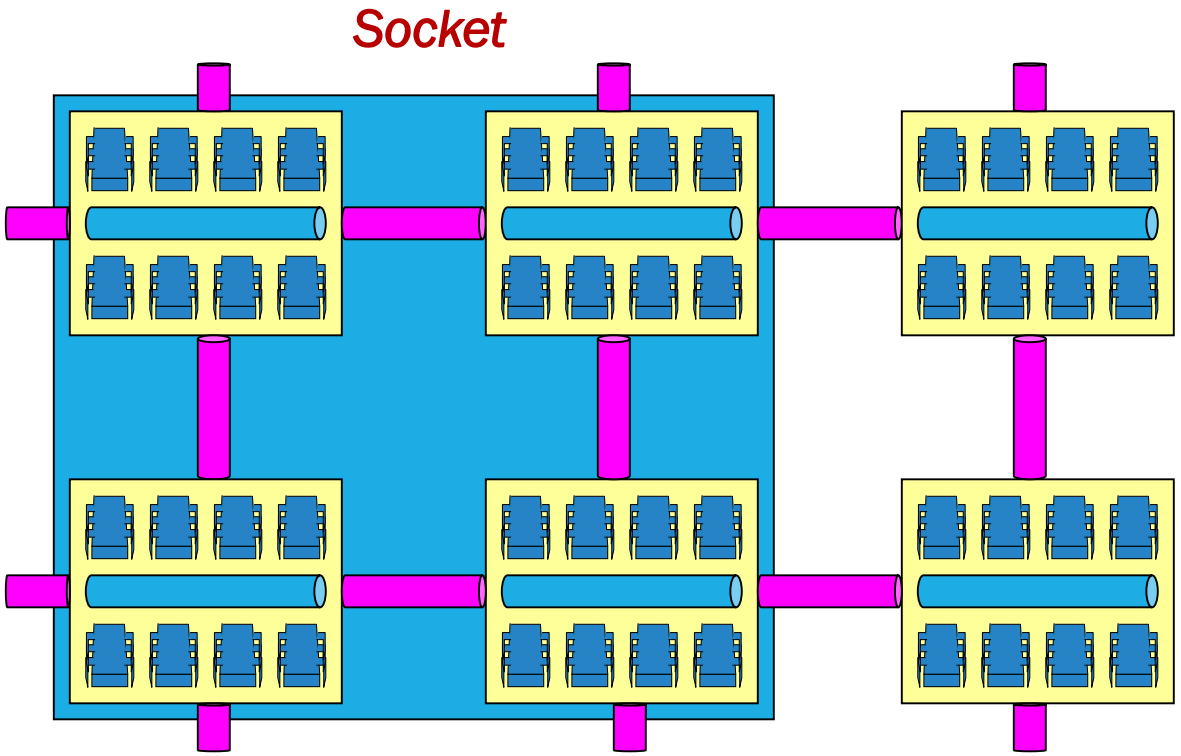
- **Package/Socket:** ensemble de dies identiques intégrés sur un même support (proco)



- C'est "ce qu'on achète"

UN PEU DE TERMINOLOGIE…

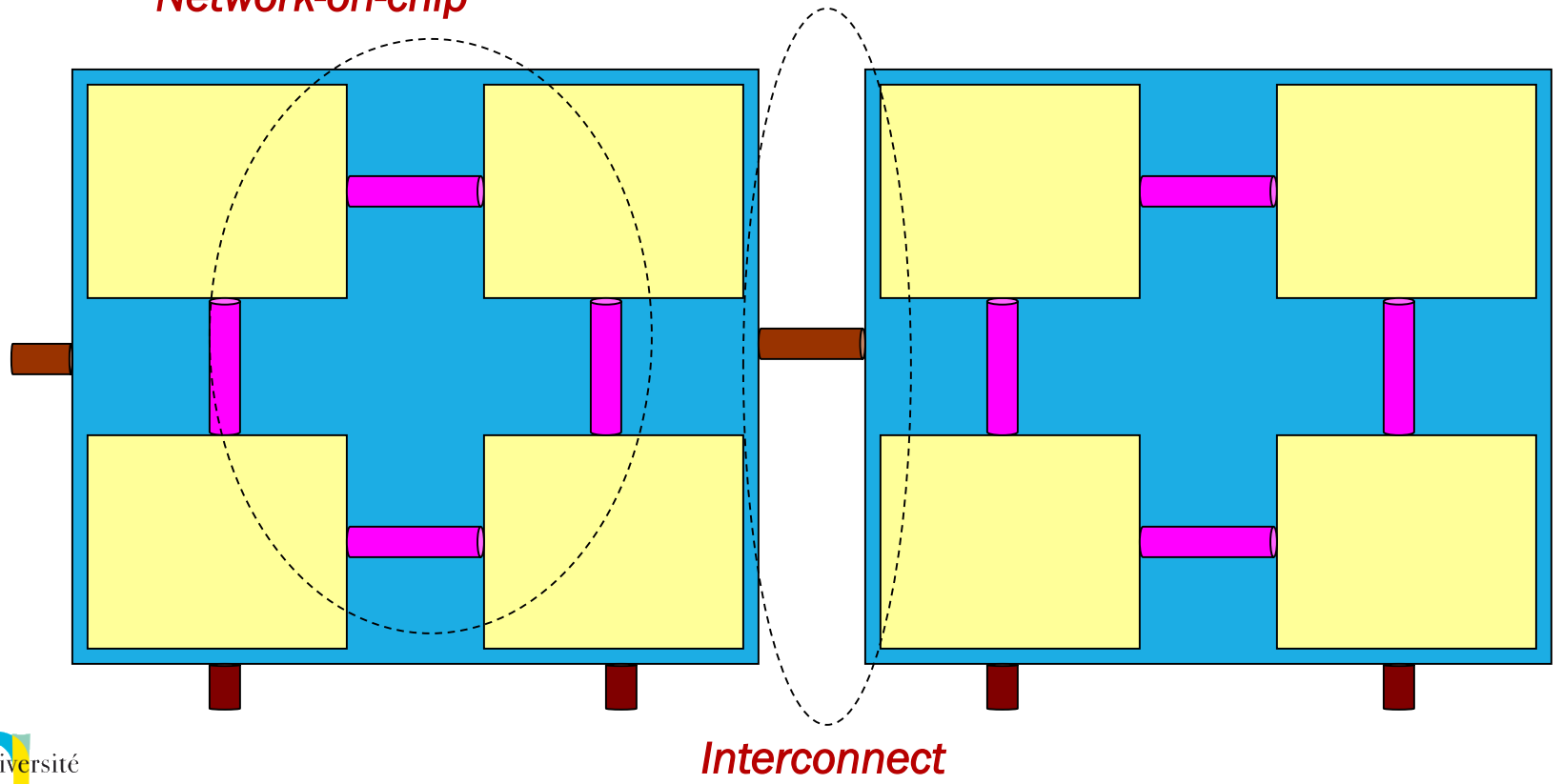
- **Package/Socket:** ensemble de dies identiques intégrés sur un même support (proco)



- **C'est "ce qu'on achète"**
- Cartes mères souvent à 1, 2, 4, ou 8 sockets…

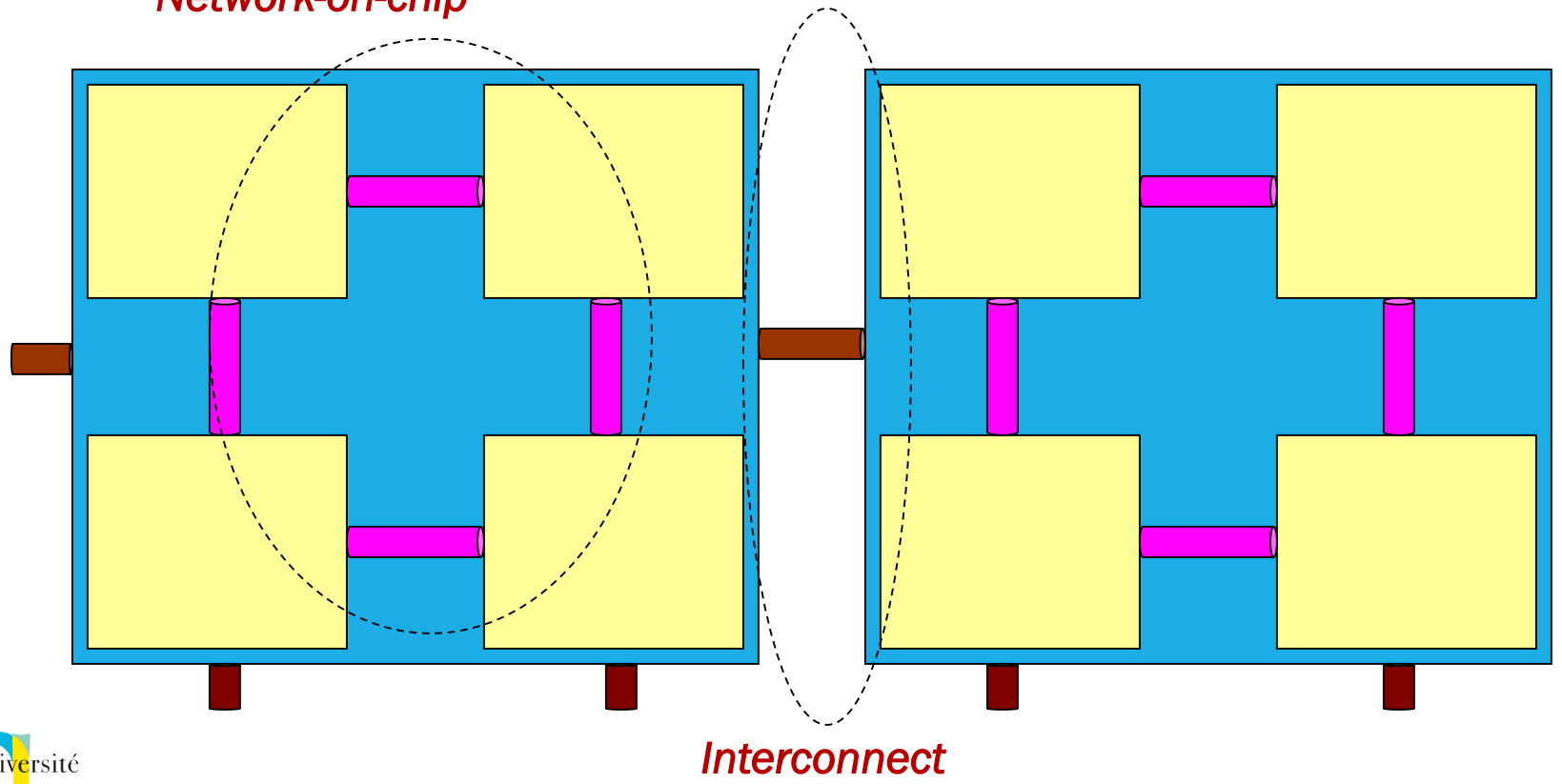
UN PEU DE TERMINOLOGIE…

Network-on-Chip (NOC) si sur un seul die.
Interconnect si entre die.
Peut mixer les deux…



UN PEU DE TERMINOLOGIE…

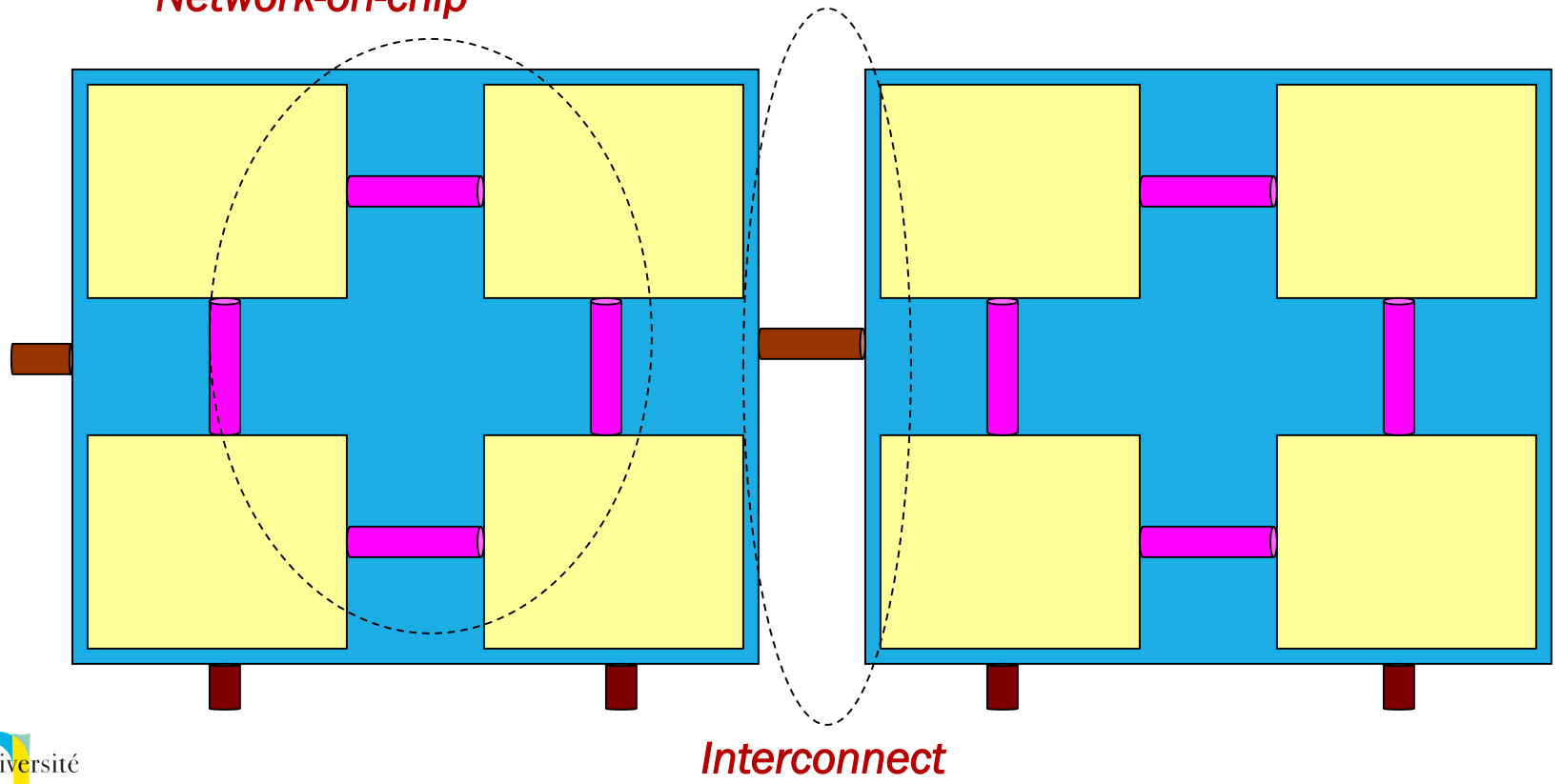
Network-on-Chip (NOC) si sur un seul die.
Interconnect si entre die.
Peut mixer les deux…



- **AMD:** HyperTransport (HT)

UN PEU DE TERMINOLOGIE…

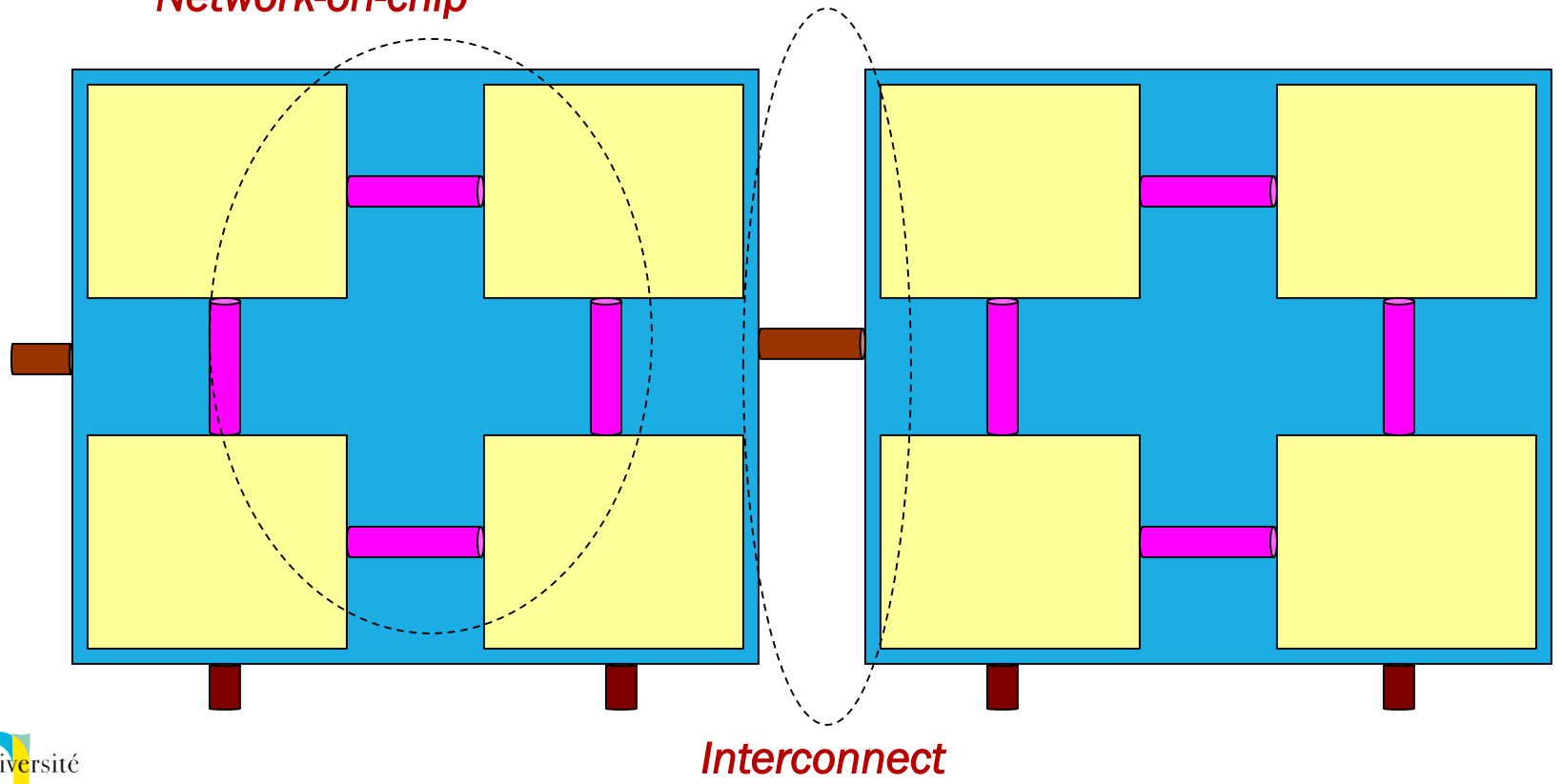
Network-on-Chip (NOC) si sur un seul die.
Interconnect si entre die.
Peut mixer les deux…



- **AMD:** HyperTransport (HT)
- **Intel:** QuickPath interconnect (QPI)

UN PEU DE TERMINOLOGIE…

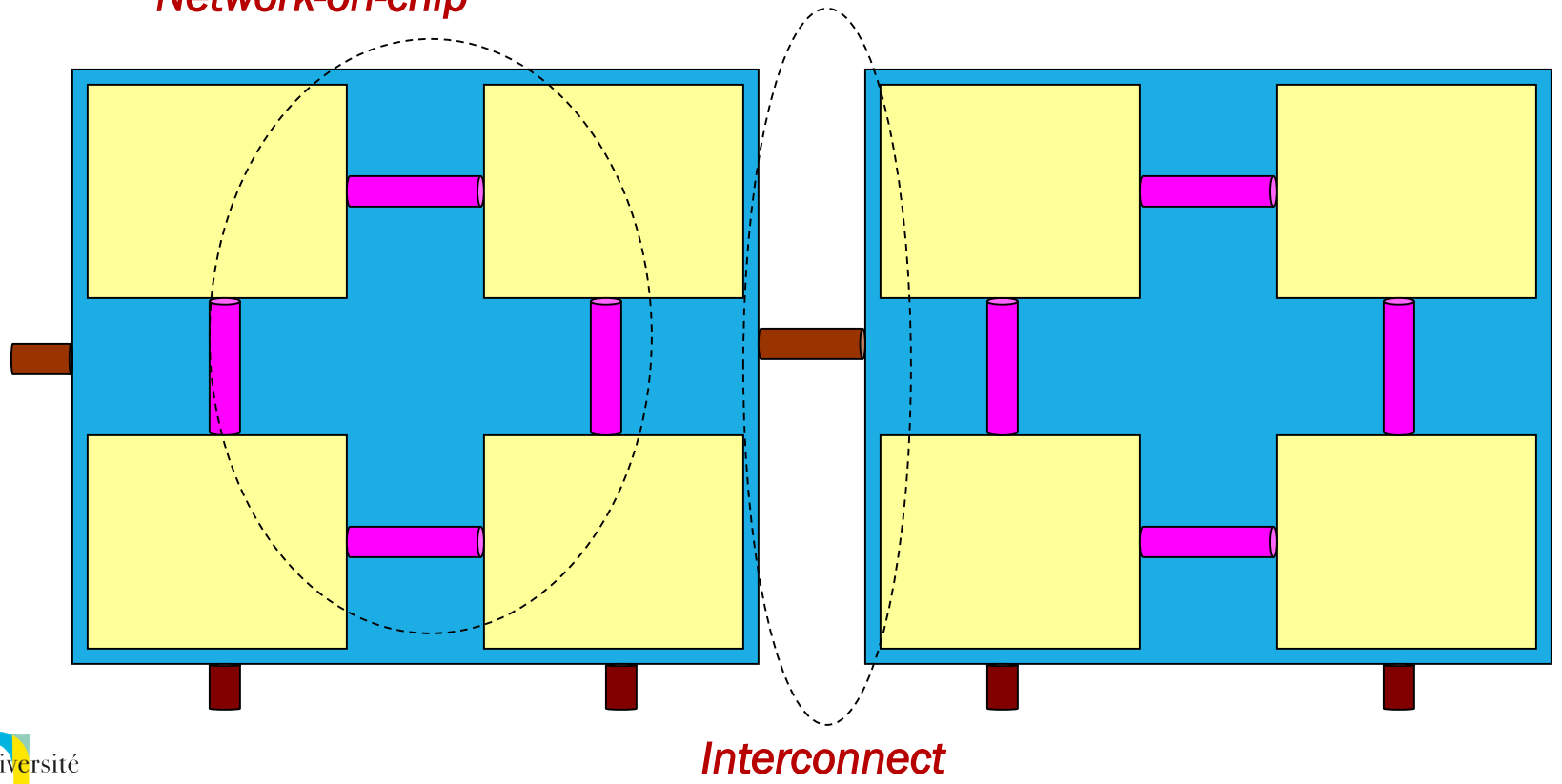
Network-on-Chip (NOC) si sur un seul die.
Interconnect si entre die.
Peut mixer les deux…



- **AMD:** HyperTransport (HT)
- **Intel:** QuickPath interconnect (QPI)
- **Packet-oriented:** contrairement à un bus, véritable réseau en packet-switching…

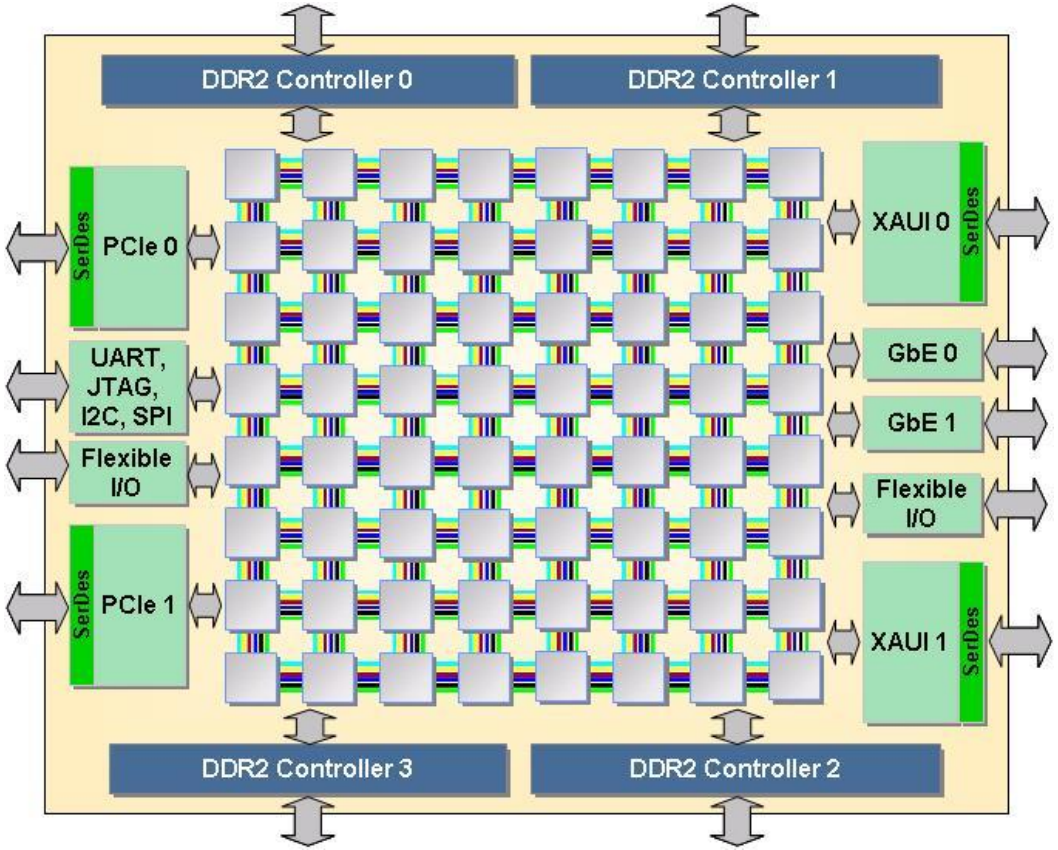
UN PEU DE TERMINOLOGIE…

Network-on-Chip (NOC) si sur un seul die.
Interconnect si entre die.
Peut mixer les deux…

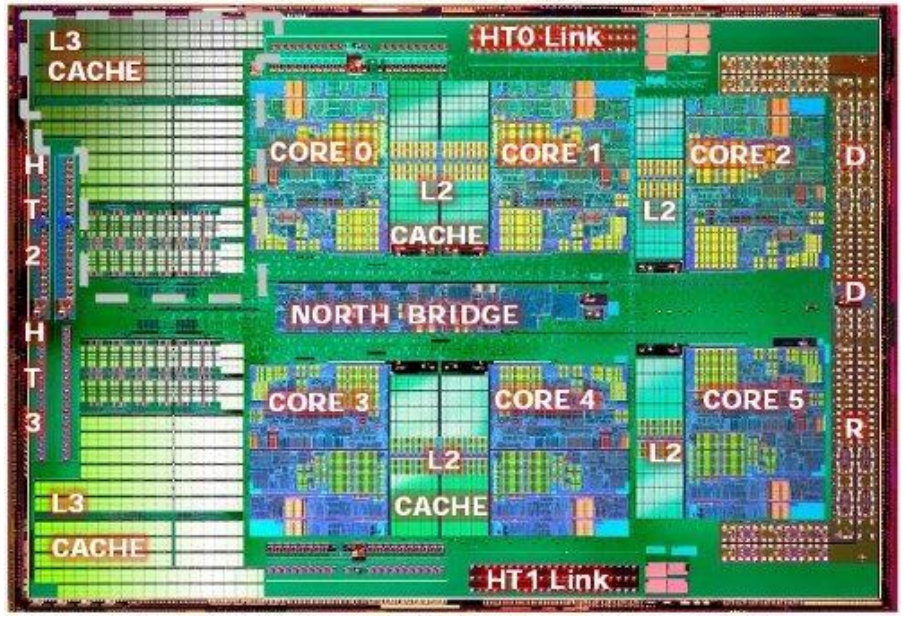


- **AMD:** HyperTransport (HT)
- **Intel:** QuickPath interconnect (QPI)
- **Packet-oriented:** contrairement à un bus, véritable réseau en packet-switching…
- Un paquet transite souvent sur plusieurs liens, e.g. pour récupérer des données stockées dans le cache d'un processeur distant…

EXEMPLES DE MULTICŒUR



64 cœurs de Tilera (unique die)

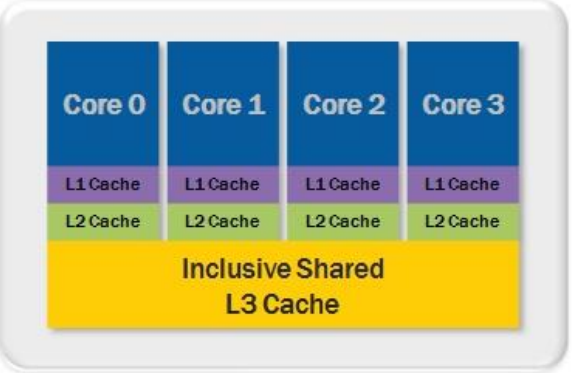


Opteron 6172, 12 cœurs, 4 CPU

(6 cœurs * 2 dies * 4 sockets = 48)

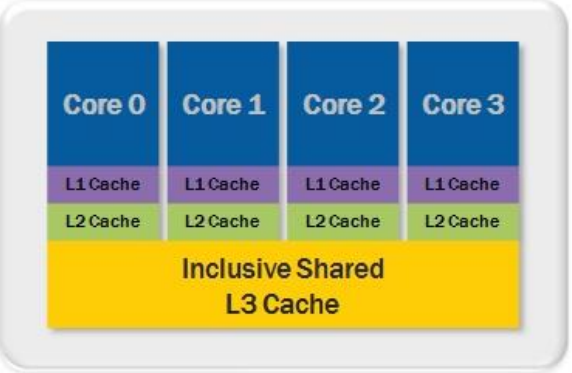
Université Nice Sophia Antipolis

COHÉRENCENCE DE CACHE



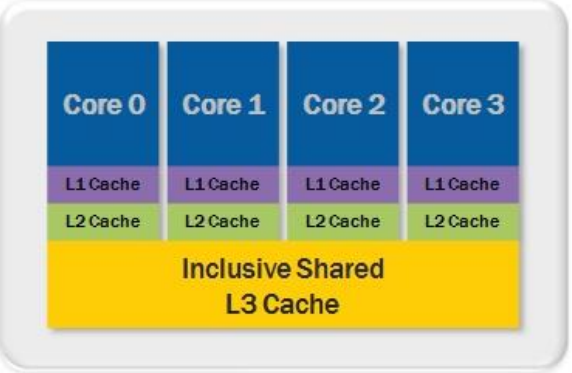
- **Deux types d'architectures :** à cohérence de cache ou sans cohérence de cache

COHÉRENCENCE DE CACHE



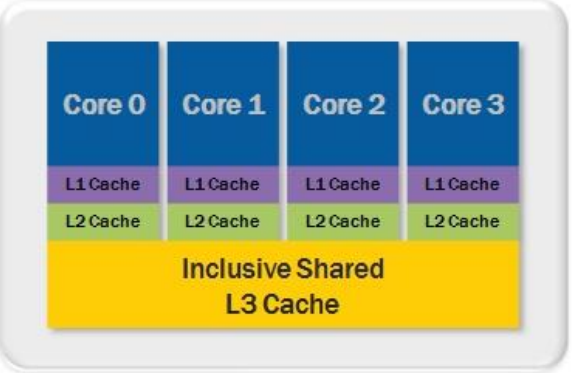
- **Deux types d'architectures :** à cohérence de cache ou sans cohérence de cache
- **Dans les architectures à cohérence de cache,** les cœurs ont une vision de la mémoire qui est « la même » pour tout le monde (à peu près, peut nécessiter certaines barrières mémoires dans certains cas).

COHÉRENCENCE DE CACHE



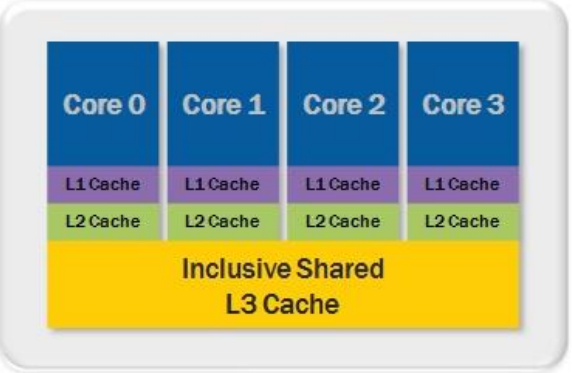
- **Deux types d'architectures :** à cohérence de cache ou sans cohérence de cache
- **Dans les architectures à cohérence de cache,** les cœurs ont une vision de la mémoire qui est « la même » pour tout le monde (à peu près, peut nécessiter certaines barrières mémoires dans certains cas).
  - C'est le cas de la plupart des architectures actuelles.

COHÉRENCENCE DE CACHE



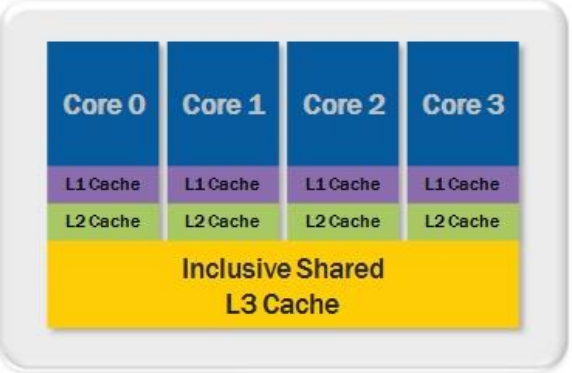
- **Deux types d'architectures :** à cohérence de cache ou sans cohérence de cache
- **Dans les architectures à cohérence de cache,** les cœurs ont une vision de la mémoire qui est « la même » pour tout le monde (à peu près, peut nécessiter certaines barrières mémoires dans certains cas).
  - C'est le cas de la plupart des architectures actuelles.
- **Chaque cœur a ses propres caches.** Si on modifie une variable en cache et qu'un CPU distant veut y accéder, ¿que pasa?

COHÉRENCENCE DE CACHE



- **Deux types d'architectures :** à cohérence de cache ou sans cohérence de cache
- **Dans les architectures à cohérence de cache,** les cœurs ont une vision de la mémoire qui est « la même » pour tout le monde (à peu près, peut nécessiter certaines barrières mémoires dans certains cas).
  - C'est le cas de la plupart des architectures actuelles.
- **Chaque cœur a ses propres caches.** Si on modifie une variable en cache et qu'un CPU distant veut y accéder, ¿que pasa?
  - Protocole qui garantit que les lignes de caches distantes sont invalidées…

COHÉRENCENCE DE CACHE



- **Deux types d'architectures :** à cohérence de cache ou sans cohérence de cache
- **Dans les architectures à cohérence de cache,** les cœurs ont une vision de la mémoire qui est « la même » pour tout le monde (à peu près, peut nécessiter certaines barrières mémoires dans certains cas).
  - C'est le cas de la plupart des architectures actuelles.
- **Chaque cœur a ses propres caches.** Si on modifie une variable en cache et qu'un CPU distant veut y accéder, ¿que pasa?
  - Protocole qui garantit que les lignes de caches distantes sont invalidées…
- **Bien sûr, assurer la cohérence de manière transparente a un coût…**

COHÉRENCENCE DE CACHE

- Lorsqu'on a un très grand nombre de cœurs, on peut vouloir supprimer le surcoût du protocole de cohérence de cache !

COHÉRENCENCE DE CACHE

- Lorsqu'on a un très grand nombre de cœurs, on peut vouloir supprimer le surcoût du protocole de cohérence de cache !
  - Architectures sans cohérence de cache…

COHÉRENCENCE DE CACHE

- **Lorsqu'on a un très grand nombre de cœurs, on peut vouloir supprimer le surcoût du protocole de cohérence de cache !**
  - **Architectures sans cohérence de cache…**
- **Idée :** chaque cœur a ses propres données et ses propres caches, et pour accéder à une donnée d'un autre cœur, on demande sa valeur par message…

COHÉRENCENCE DE CACHE

- **Lorsqu'on a un très grand nombre de cœurs, on peut vouloir supprimer le surcoût du protocole de cohérence de cache !**
  - **Architectures sans cohérence de cache…**
- **Idée :** chaque cœur a ses propres données et ses propres caches, et pour accéder à une donnée d'un autre cœur, on demande sa valeur par message…
- **Fonctionne exactement comme une grappe d'ordinateurs en réseau…**

COHÉRENCENCE DE CACHE

- **Lorsqu'on a un très grand nombre de cœurs, on peut vouloir supprimer le surcoût du protocole de cohérence de cache !**
  - **Architectures sans cohérence de cache…**
- **Idée :** chaque cœur a ses propres données et ses propres caches, et pour accéder à une donnée d'un autre cœur, on demande sa valeur par message…
- **Fonctionne exactement comme une grappe d'ordinateurs en réseau…**
- **Le futur ?** Qui sait, prédit depuis des années, en pratique encore rare/exotique.

COHÉRENCENCE DE CACHE

- **Lorsqu'on a un très grand nombre de cœurs, on peut vouloir supprimer le surcoût du protocole de cohérence de cache !**
  - **Architectures sans cohérence de cache…**
- **Idée :** chaque cœur a ses propres données et ses propres caches, et pour accéder à une donnée d'un autre cœur, on demande sa valeur par message…
- **Fonctionne exactement comme une grappe d'ordinateurs en réseau…**
- **Le futur ?** Qui sait, prédit depuis des années, en pratique encore rare/exotique.
- Certaines architectures entre les deux : voir Tilera…

COHÉRENCENCE DE CACHE

- **Lorsqu'on a un très grand nombre de cœurs, on peut vouloir supprimer le surcoût du protocole de cohérence de cache !**
  - **Architectures sans cohérence de cache…**
- **Idée :** chaque cœur a ses propres données et ses propres caches, et pour accéder à une donnée d'un autre cœur, on demande sa valeur par message…
- **Fonctionne exactement comme une grappe d'ordinateurs en réseau…**
- **Le futur ?** Qui sait, prédit depuis des années, en pratique encore rare/exotique.
- Certaines architectures entre les deux : voir Tilera…
- **Dans ce cours, on ne considèrera que les architectures à cohérence de cache !**

PROTOCOLES DE COHÉRENCE DE CACHE : MSI

- **Protocole le plus basique:** MSI = trois états « Modified, Shared, Invalid »

PROTOCOLES DE COHÉRENCE DE CACHE : MSI

- **Protocole le plus basique:** MSI = trois états « Modified, Shared, Invalid »
- **Modified :** Seule copie valide, cœur est son propriétaire. Copie en mémoire peut être ou non valide (cohérence non assurée). Si ligne évincée, doit être recopiée en mémoire.

PROTOCOLES DE COHÉRENCE DE CACHE : MSI

- **Protocole le plus basique:** MSI = trois états « Modified, Shared, Invalid »
- **Modified :** Seule copie valide, cœur est son propriétaire. Copie en mémoire peut être ou non valide (cohérence non assurée). Si ligne évincée, doit être recopiée en mémoire.
- **Shared :** Ligne dans le cache sans jamais avoir été modifiée. D'autres caches peuvent héberger une copie de cette ligne. Cohérence assurée.

PROTOCOLES DE COHÉRENCE DE CACHE : MSI

- **Protocole le plus basique:** MSI = trois états « Modified, Shared, Invalid »
- **Modified :** Seule copie valide, cœur est son propriétaire. Copie en mémoire peut être ou non valide (cohérence non assurée). Si ligne évincée, doit être recopiée en mémoire.
- **Shared :** Ligne dans le cache sans jamais avoir été modifiée. D'autres caches peuvent héberger une copie de cette ligne. Cohérence assurée.
- **Invalid :** Ligne non valide. L'information qui y est rangée n'est pas digne de confiance. Tout accès donne lieu à un *miss*.

PROTOCOLES DE COHÉRENCE DE CACHE : MSI

- **Protocole le plus basique:** MSI = trois états « Modified, Shared, Invalid »
- **Modified :** Seule copie valide, cœur est son propriétaire. Copie en mémoire peut être ou non valide (cohérence non assurée). Si ligne évincée, doit être recopiée en mémoire.
- **Shared :** Ligne dans le cache sans jamais avoir été modifiée. D'autres caches peuvent héberger une copie de cette ligne. Cohérence assurée.
- **Invalid :** Ligne non valide. L'information qui y est rangée n'est pas digne de confiance. Tout accès donne lieu à un *miss*.
- **Lecture d'une ligne de cache :** Si M/S, direct. Sinon, le protocole s'assure que pas dans un état M ailleurs. Si oui, cœur distant en M écrit en mémoire et passe en S. Cœur local obtient la ligne localement en S.

PROTOCOLES DE COHÉRENCE DE CACHE : MSI

- Ecriture d'une ligne de cache :
  - Si M, direct.

PROTOCOLES DE COHÉRENCE DE CACHE : MSI

- **Ecriture d'une ligne de cache :**
  - Si M, direct.
  - Si S, on invalide toutes les autres versions qui ont la ligne en S, si nécessaire. Puis on passe en M et écrit. (On sait que pas d'autre cache a la ligne en M).

PROTOCOLES DE COHÉRENCE DE CACHE : MSI

- **Ecriture d'une ligne de cache :**
  - Si M, direct.
  - Si S, on invalide toutes les autres versions qui ont la ligne en S, si nécessaire. Puis on passe en M et écrit. (On sait que pas d'autre cache a la ligne en M).
  - Si I, plus ou moins la même chose, mais on invalide les M aussi (qui écrivent dans la RAM ou transfèrent la donnée pour instruction qui read+write).

PROTOCOLES DE COHÉRENCE DE CACHE : MSI

- **Ecriture d'une ligne de cache :**
  - Si M, direct.
  - Si S, on invalide toutes les autres versions qui ont la ligne en S, si nécessaire. Puis on passe en M et écrit. (On sait que pas d'autre cache a la ligne en M).
  - Si I, plus ou moins la même chose, mais on invalide les M aussi (qui écrivent dans la RAM ou transfèrent la donnée pour instruction qui read+write).

	M	S	I
M	✗	✗	✓
S	✗	✓	✓
I	✓	✓	✓

PROTOCOLES DE COHÉRENCE DE CACHE : MSI

- **Ecriture d'une ligne de cache :**
  - Si M, direct.
  - Si S, on invalide toutes les autres versions qui ont la ligne en S, si nécessaire. Puis on passe en M et écrit. (On sait que pas d'autre cache a la ligne en M).
  - Si I, plus ou moins la même chose, mais on invalide les M aussi (qui écrivent dans la RAM ou transfèrent la donnée pour instruction qui read+write).

- **Problème 1 de MSI :** écriture de ligne de cache présente dans un seul cache (local) en S nécessite un broadcast. Or, très courant…

	M	S	I
M	✗	✗	✓
S	✗	✓	✓
I	✓	✓	✓

PROTOCOLES DE COHÉRENCE DE CACHE : MSI

- **Ecriture d'une ligne de cache :**
  - Si M, direct.
  - Si S, on invalide toutes les autres versions qui ont la ligne en S, si nécessaire. Puis on passe en M et écrit. (On sait que pas d'autre cache a la ligne en M).
  - Si I, plus ou moins la même chose, mais on invalide les M aussi (qui écrivent dans la RAM ou transfèrent la donnée pour instruction qui read+write).

- **Problème 1 de MSI :** écriture de ligne de cache présente dans un seul cache (local) en S nécessite un broadcast. Or, très courant…
- **Problème 2 de MSI :** lecture de ligne de cache présente en M ailleurs. Nécessite une invalidation, un write-back, une lecture de la mémoire…

	M	S	I
M	✗	✗	✓
S	✗	✓	✓
I	✓	✓	✓

PROTOCOLES DE COHÉRENCE DE CACHE : MESI

- **Problème 1 de MSI :** écriture de ligne de cache présente dans un seul cache (local) en S nécessite un broadcast. Or, très courant…

PROTOCOLES DE COHÉRENCE DE CACHE : MESI

- **Problème 1 de MSI :** écriture de ligne de cache présente dans un seul cache (local) en S nécessite un broadcast. Or, très courant…
- **Solution:** couper l'état M en état E (« Exclusive ») et M.
  - Si Modified, seule copie valide, et RAM pas à jour.
  - Si Exclusive, seule copie valide, et RAM à jour.

	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

PROTOCOLES DE COHÉRENCE DE CACHE : MESI

- **Problème 1 de MSI :** écriture de ligne de cache présente dans un seul cache (local) en S nécessite un broadcast. Or, très courant…
- **Solution:** couper l'état M en état E (« Exclusive ») et M.
  - Si Modified, seule copie valide, et RAM pas à jour.
  - Si Exclusive, seule copie valide, et RAM à jour.
- Du coup, lorsqu'on lit une variable pour la première fois, elle est en E au lieu d'être en S. **Si on la modifie, pas de broadcast pour passer en M.**

	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

PROTOCOLES DE COHÉRENCE DE CACHE : MESI

- **Problème 1 de MSI :** écriture de ligne de cache présente dans un seul cache (local) en S nécessite un broadcast. Or, très courant…
- **Solution:** couper l'état M en état E (« Exclusive ») et M.
  - Si Modified, seule copie valide, et RAM pas à jour.
  - Si Exclusive, seule copie valide, et RAM à jour.
- Du coup, lorsqu'on lit une variable pour la première fois, elle est en E au lieu d'être en S. **Si on la modifie, pas de broadcast pour passer en M.**
- On peut passer en M directement, on sait qu'on est seuls à manipuler la donnée…

	M	E	S	I
M	✗	✗	✗	✓
E	✗	✗	✗	✓
S	✗	✗	✓	✓
I	✓	✓	✓	✓

PROTOCOLES DE COHÉRENCE DE CACHE : MOSI

- **Problème 2 de MSI :** lecture de ligne de cache présente en M ailleurs. Nécessite une invalidation, un write-back, une lecture de la mémoire… Ou en S. Nécessite d'aller lire la mémoire…

PROTOCOLES DE COHÉRENCE DE CACHE : MOSI

- **Problème 2 de MSI :** lecture de ligne de cache présente en M ailleurs. Nécessite une invalidation, un write-back, une lecture de la mémoire… Ou en S. Nécessite d'aller lire la mémoire…
- **Solution:** ajout de l'état O, « Owned »
  - Un des cœurs responsable de la ligne, **possède la dernière valeur** et **peut l'envoyer directement**

	M	O	S	I
M	✗	✗	✗	✓
O	✗	✗	✓	✓
S	✗	✓	✓	✓
I	✓	✓	✓	✓

PROTOCOLES DE COHÉRENCE DE CACHE : MOSI

- **Problème 2 de MSI :** lecture de ligne de cache présente en M ailleurs. Nécessite une invalidation, un write-back, une lecture de la mémoire… Ou en S. Nécessite d'aller lire la mémoire…
- **Solution:** ajout de l'état O, « Owned »
  - Un des cœurs responsable de la ligne, **possède la dernière valeur** et **peut l'envoyer directement**
  - Autres processeurs peuvent avoir la ligne en Shared
    - *Modif de la ligne peut mettre à jour ou invalider ces lignes…*

	M	O	S	I
M	✗	✗	✗	✓
O	✗	✗	✓	✓
S	✗	✓	✓	✓
I	✓	✓	✓	✓

PROTOCOLES DE COHÉRENCE DE CACHE : MOSI

- **Problème 2 de MSI :** lecture de ligne de cache présente en M ailleurs. Nécessite une invalidation, un write-back, une lecture de la mémoire… Ou en S. Nécessite d'aller lire la mémoire…
- **Solution:** ajout de l'état O, « Owned »
  - Un des cœurs responsable de la ligne, **possède la dernière valeur** et **peut l'envoyer directement**
  - Autres processeurs peuvent avoir la ligne en Shared
    - *Modif de la ligne peut mettre à jour ou invalider ces lignes…*
- Du coup, lorsqu'on lit une ligne de cache cohérente ou pas distante, si possible elle se trouve en Owned (vrai si lue une fois).

	M	O	S	I
M	✗	✗	✗	✓
O	✗	✗	✓	✓
S	✗	✓	✓	✓
I	✓	✓	✓	✓

PROTOCOLES DE COHÉRENCE DE CACHE : MOSI

- **Problème 2 de MSI :** lecture de ligne de cache présente en M ailleurs. Nécessite une invalidation, un write-back, une lecture de la mémoire… Ou en S. Nécessite d'aller lire la mémoire…
- **Solution:** ajout de l'état O, « Owned »
  - Un des cœurs responsable de la ligne, **possède la dernière valeur** et **peut l'envoyer directement**
  - Autres processeurs peuvent avoir la ligne en Shared
    - *Modif de la ligne peut mettre à jour ou invalider ces lignes…*
- Du coup, lorsqu'on lit une ligne de cache cohérente ou pas distante, si possible elle se trouve en Owned (vrai si lue une fois).
- ***On demande donc au cœur qui a la ligne de cache en Owned d'envoyer la valeur, en évitant de passer par la RAM…***

	M	O	S	I
M	✗	✗	✗	✓
O	✗	✗	✓	✓
S	✗	✓	✓	✓
I	✓	✓	✓	✓

PROTOCOLES DE COHÉRENCE DE CACHE : MOESI

- **Le plus courant de nos jours :** MOESI qui combine MESI et MOSI…

PROTOCOLES DE COHÉRENCE DE CACHE : MOESI

	M	E	S	I	F
M	✗	✗	✗	✓	✗
E	✗	✗	✗	✓	✗
S	✗	✗	✓	✓	✓
I	✓	✓	✓	✓	✓
F	✗	✗	✓	✓	✗

- **Le plus courant de nos jours :** MOESI qui combine MESI et MOSI…
  - *D'autres protocoles existent, comme MESIF, utilisé e.g. sur des SPARC*

PROTOCOLES DE COHÉRENCE DE CACHE : MOESI

	M	E	S	I	F
M	✗	✗	✗	✓	✗
E	✗	✗	✗	✓	✗
S	✗	✗	✓	✓	✓
I	✓	✓	✓	✓	✓
F	✗	✗	✓	✓	✗

- **Le plus courant de nos jours :** MOESI qui combine MESI et MOSI…
  - *D'autres protocoles existent, comme MESIF, utilisé e.g. sur des SPARC*

- **Au final, on a :**

	Modified	Owned	Exclusive	Shared	Invalid
Mémoire centrale	Incohérente	Incohérente	Cohérente	Cohérente	Cohérente
Répliquées	Non	Oui	Non	Oui	Indéfini

PROTOCOLES DE COHÉRENCE DE CACHE : MOESI

	M	E	S	I	F
M	✗	✗	✗	✓	✗
E	✗	✗	✗	✓	✗
S	✗	✗	✓	✓	✓
I	✓	✓	✓	✓	✓
F	✗	✗	✓	✓	✗

- **Le plus courant de nos jours :** MOESI qui combine MESI et MOSI…
  - *D'autres protocoles existent, comme MESIF, utilisé e.g. sur des SPARC*
- **Au final, on a :**

	Modified	Owned	Exclusive	Shared	Invalid
Mémoire centrale	Incohérente	Incohérente	Cohérente	Cohérente	Cohérente
Répliquées	Non	Oui	Non	Oui	Indéfini

- **Principe de transition : équivalent à un verrou lecteur/écrivain**
  - **Etats M et E :** verrou en écriture
  - **Etats E et S :** verrou en lecture
  - **Etat I :** pas de verrou

PROTOCOLES DE COHÉRENCE DE CACHE : MOESI

	M	E	S	I	F
M	✗	✗	✗	✓	✗
E	✗	✗	✗	✓	✗
S	✗	✗	✓	✓	✓
I	✓	✓	✓	✓	✓
F	✗	✗	✓	✓	✗

- **Le plus courant de nos jours :** MOESI qui combine MESI et MOSI…
  - *D'autres protocoles existent, comme MESIF, utilisé e.g. sur des SPARC*

- **Au final, on a :**

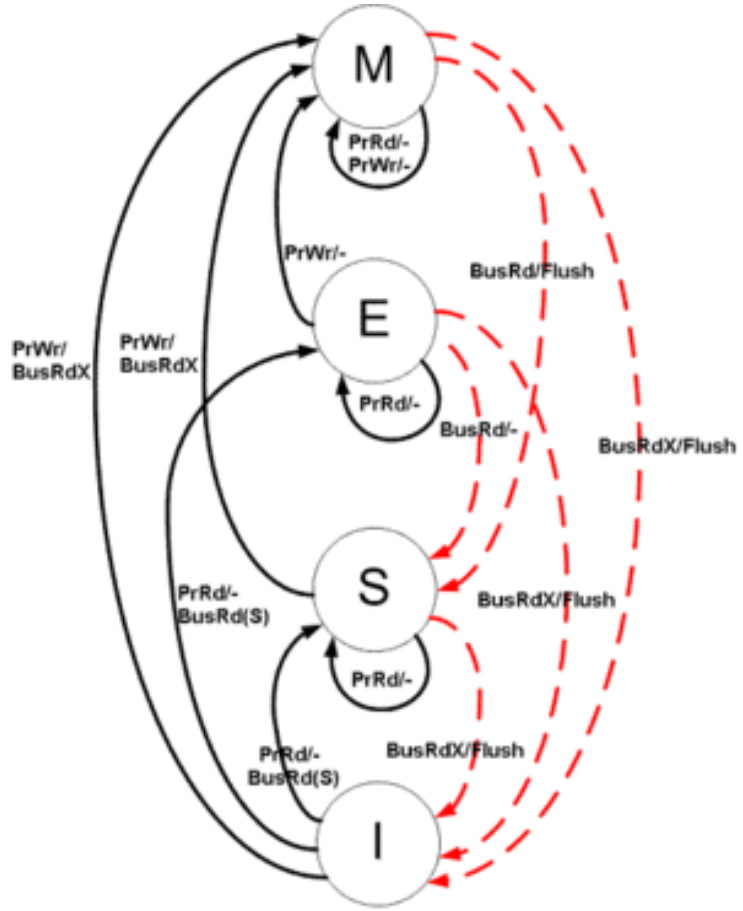
	Modified	Owned	Exclusive	Shared	Invalid
Mémoire centrale	Incohérente	Incohérente	Cohérente	Cohérente	Cohérente
Répliquées	Non	Oui	Non	Oui	Indéfini

- **Principe de transition : équivalent à un verrou lecteur/écrivain**
  - **Etats M et E :** verrou en écriture
  - **Etats E et S :** verrou en lecture
  - **Etat I :** pas de verrou

- **Duplication des états verrou en lecture ou écriture pour indiquer si la ligne est sale !**

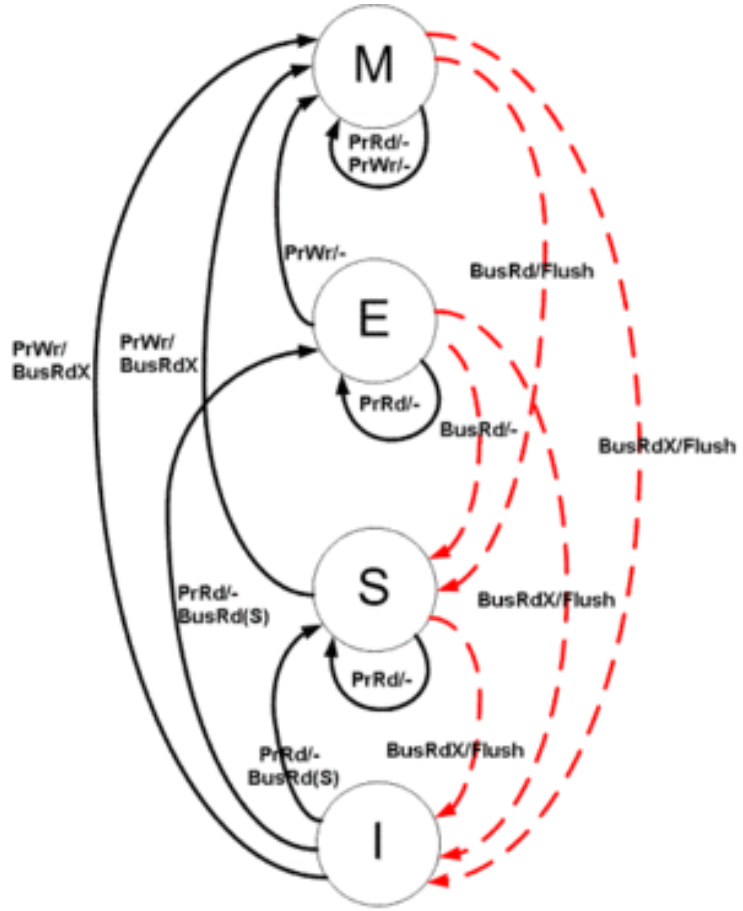
PROTOCOLES DE COHÉRENCE DE CACHE

- **Protocoles de cohérence de cache :** un peu « prise de tête »



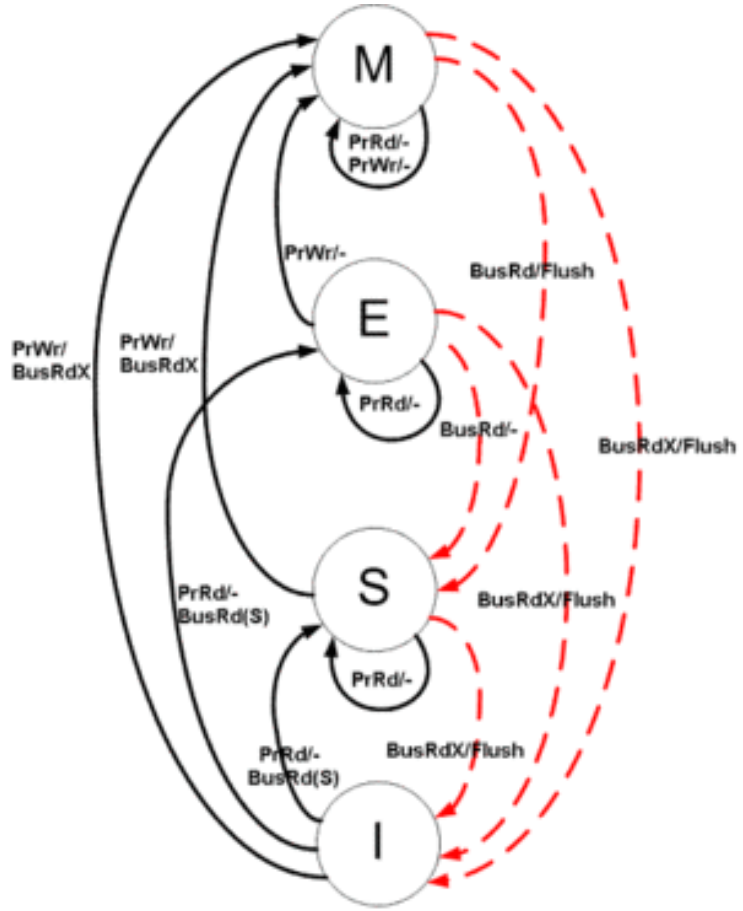
PROTOCOLES DE COHÉRENCE DE CACHE

- **Protocoles de cohérence de cache :** un peu « prise de tête »
- **Wikipedia a de très bons articles pour chacun :**
  https://en.wikipedia.org/wiki/MSI_protocol
  https://en.wikipedia.org/wiki/MESI_protocol
  https://en.wikipedia.org/wiki/MOSI_protocol
  https://en.wikipedia.org/wiki/MOESI_protocol
  https://en.wikipedia.org/wiki/MESIF_protocol



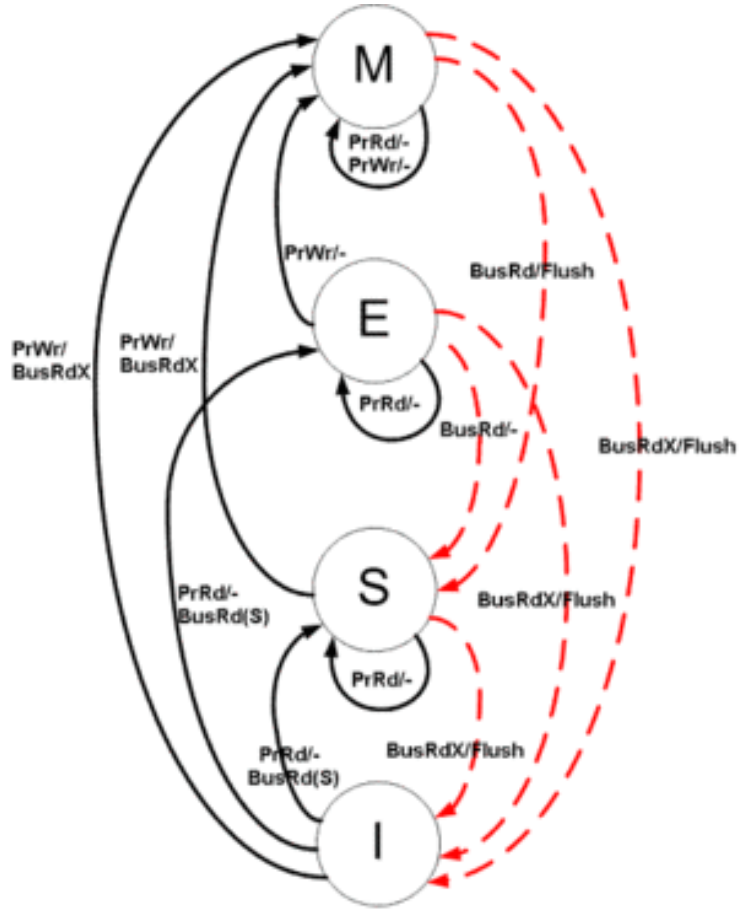
PROTOCOLES DE COHÉRENCE DE CACHE

- **Protocoles de cohérence de cache :** un peu « prise de tête »
- **Wikipedia a de très bons articles pour chacun :**
  https://en.wikipedia.org/wiki/MSI_protocol
  https://en.wikipedia.org/wiki/MESI_protocol
  https://en.wikipedia.org/wiki/MOSI_protocol
  https://en.wikipedia.org/wiki/MOESI_protocol
  https://en.wikipedia.org/wiki/MESIF_protocol
- **Protocoles sont machines à états**



PROTOCOLES DE COHÉRENCE DE CACHE

- **Protocoles de cohérence de cache :** un peu « prise de tête »
- **Wikipedia a de très bons articles pour chacun :**
  https://en.wikipedia.org/wiki/MSI_protocol
  https://en.wikipedia.org/wiki/MESI_protocol
  https://en.wikipedia.org/wiki/MOSI_protocol
  https://en.wikipedia.org/wiki/MOESI_protocol
  https://en.wikipedia.org/wiki/MESIF_protocol



- **Protocoles sont machines à états**
- **N'expliquent pas forcément *comment* les CPUs peuvent les implémenter…**
  - E.g., comment marche une invalidation ? Comment récupère-t-on l'owner d'une ligne ?

INVALIDATIONS

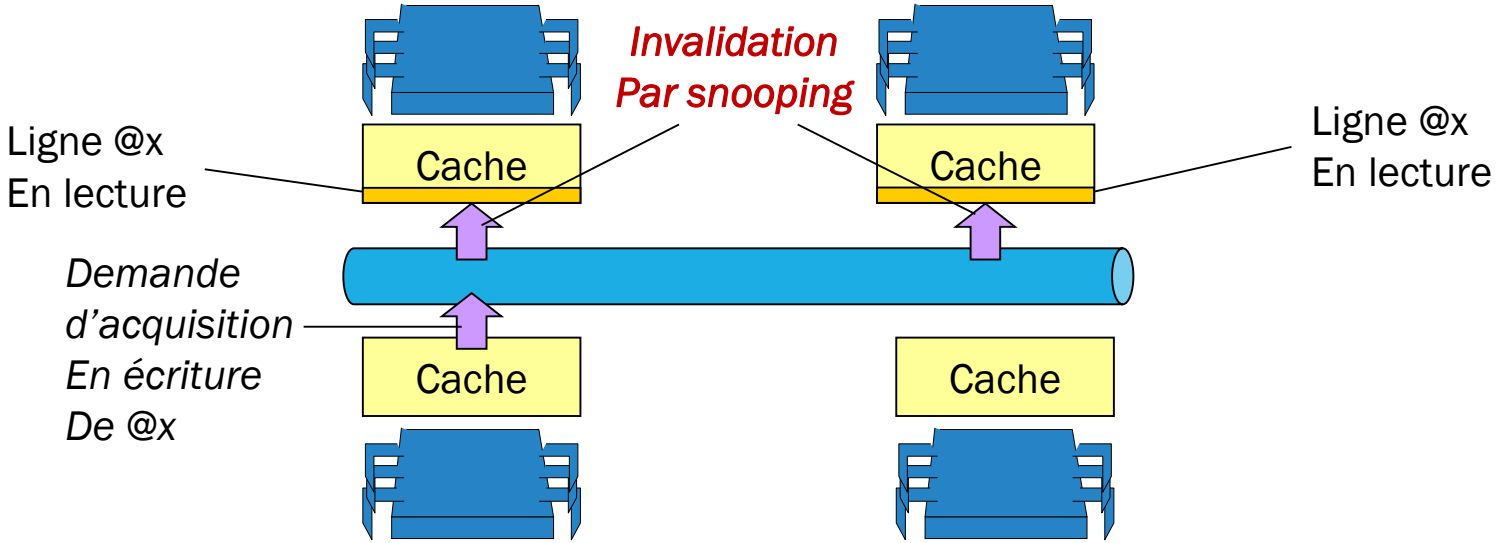
- **Problème :** comment invalider une ligne de cache ?

INVALIDATIONS

- **Problème :** comment invalider une ligne de cache ?
- **Classiquement avec un bus unique :** changement d'état via « snooping »

INVALIDATIONS

- **Problème :** comment invalider une ligne de cache ?
- **Classiquement avec un bus unique :** changement d'état via « snooping »
- I.e., espionnage des demandes de lecture/écriture :

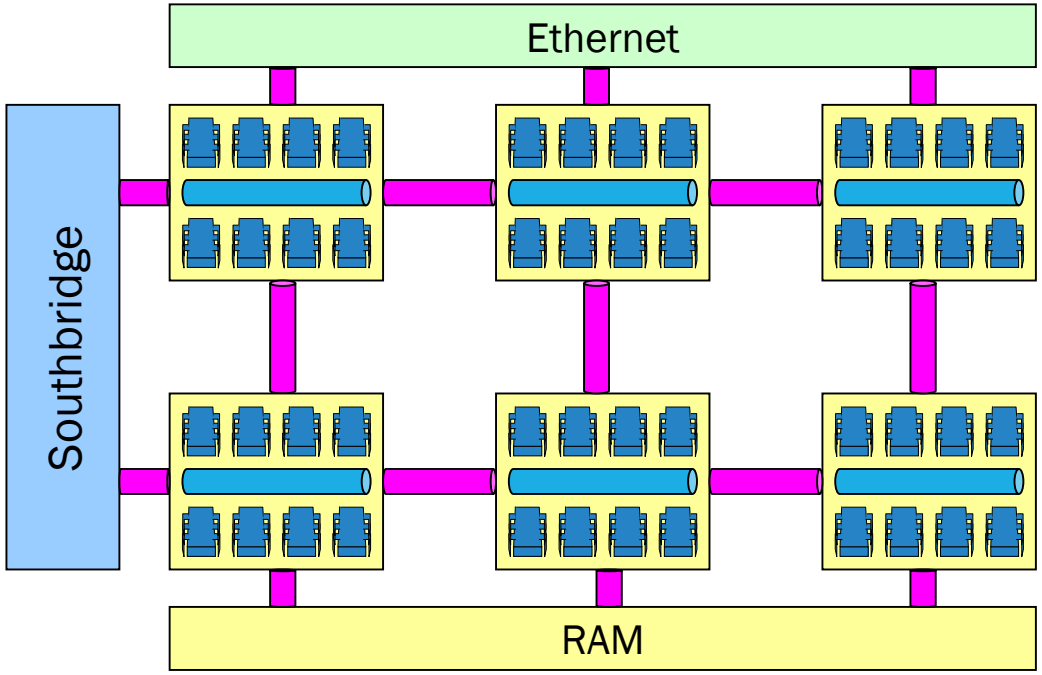


INVALIDATIONS

- **Problème :** comment invalider une ligne de cache ?

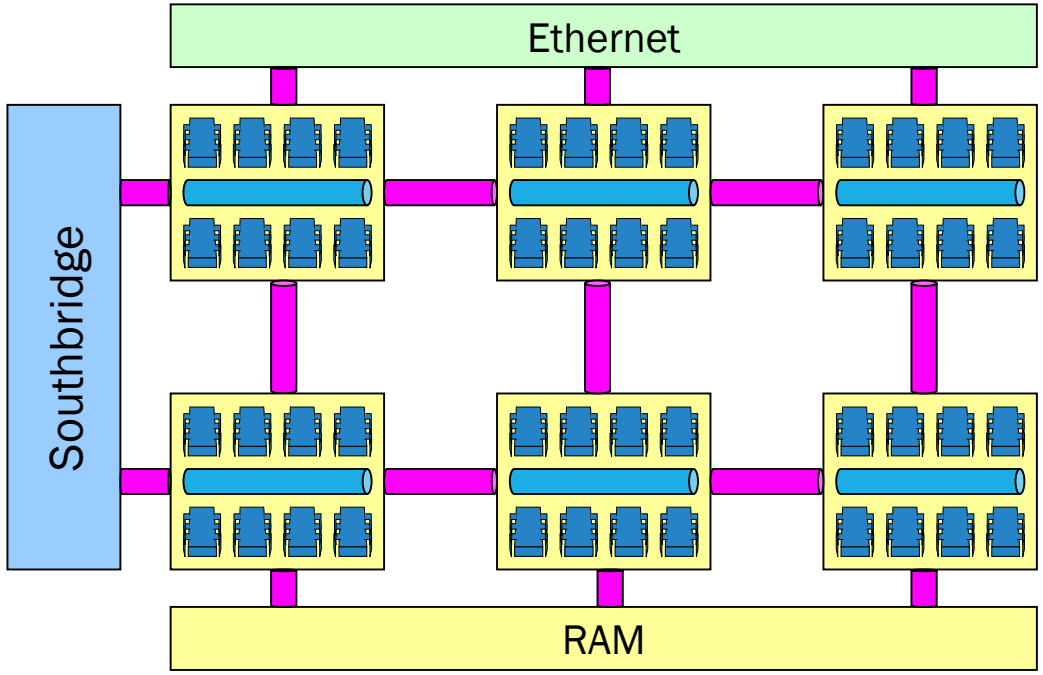
INVALIDATIONS

- **Problème :** comment invalider une ligne de cache ?
- **Snooping ne passe pas à l'échelle car nécessite un broadcast !**



INVALIDATIONS

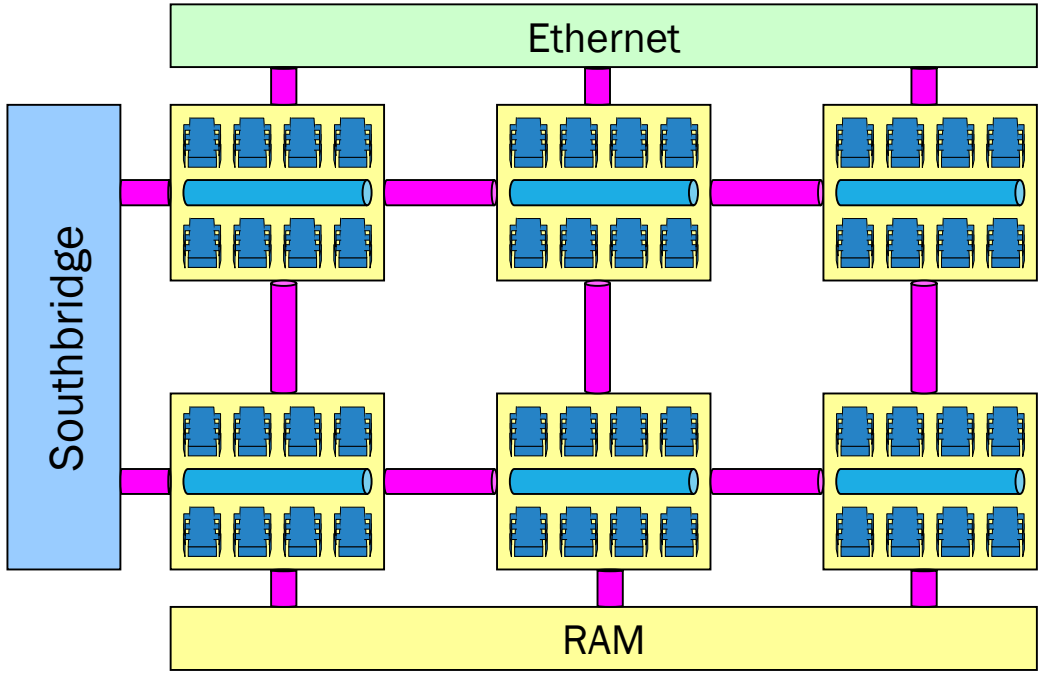
- **Problème :** comment invalider une ligne de cache ?
- **Snooping ne passe pas à l'échelle car nécessite un broadcast !**



- **Solution :** une table associant adresse physique avec cœurs qui cachent la ligne invalidations adressées explicitement aux cœurs qui cachent la ligne

INVALIDATIONS

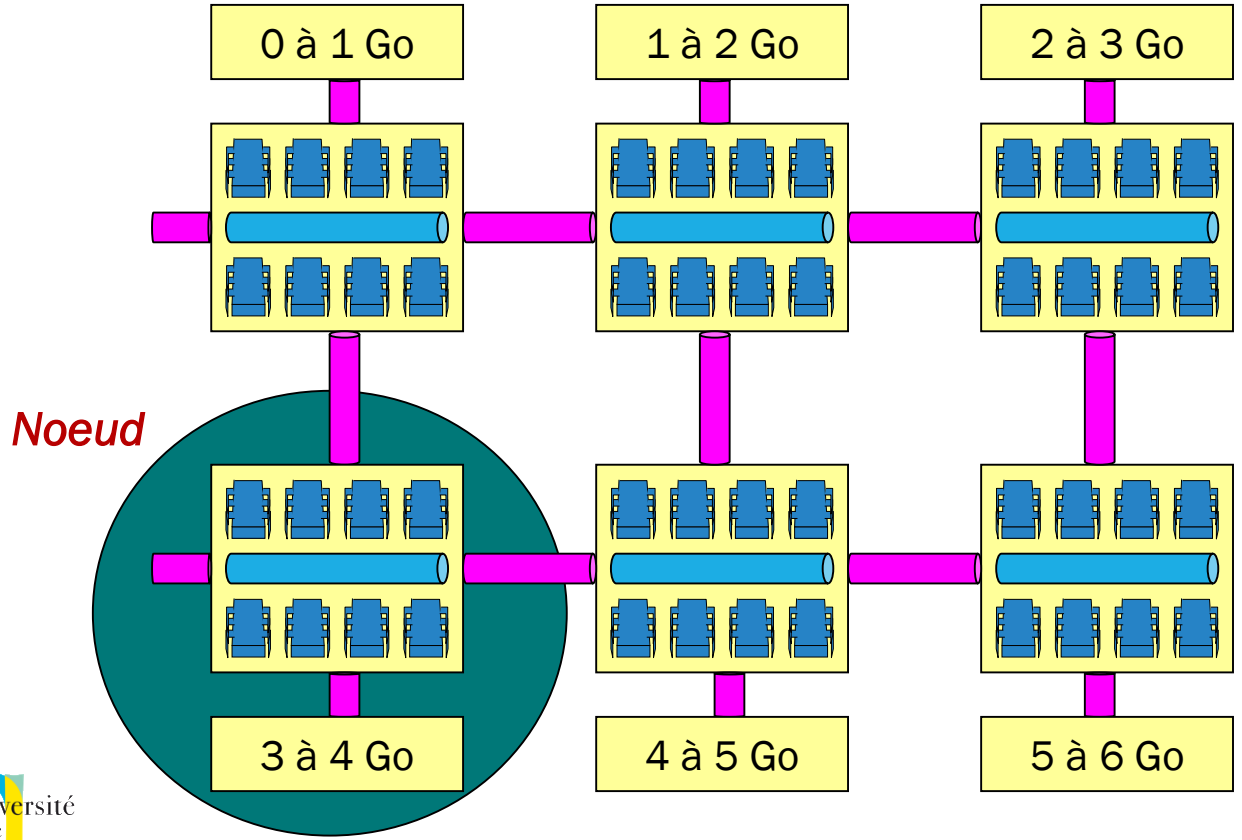
- **Problème :** comment invalider une ligne de cache ?
- **Snooping ne passe pas à l'échelle car nécessite un broadcast !**



- **Solution :** une table associant adresse physique avec cœurs qui cachent la ligne invalidations adressées explicitement aux cœurs qui cachent la ligne
- Problème : où placer cette table ?

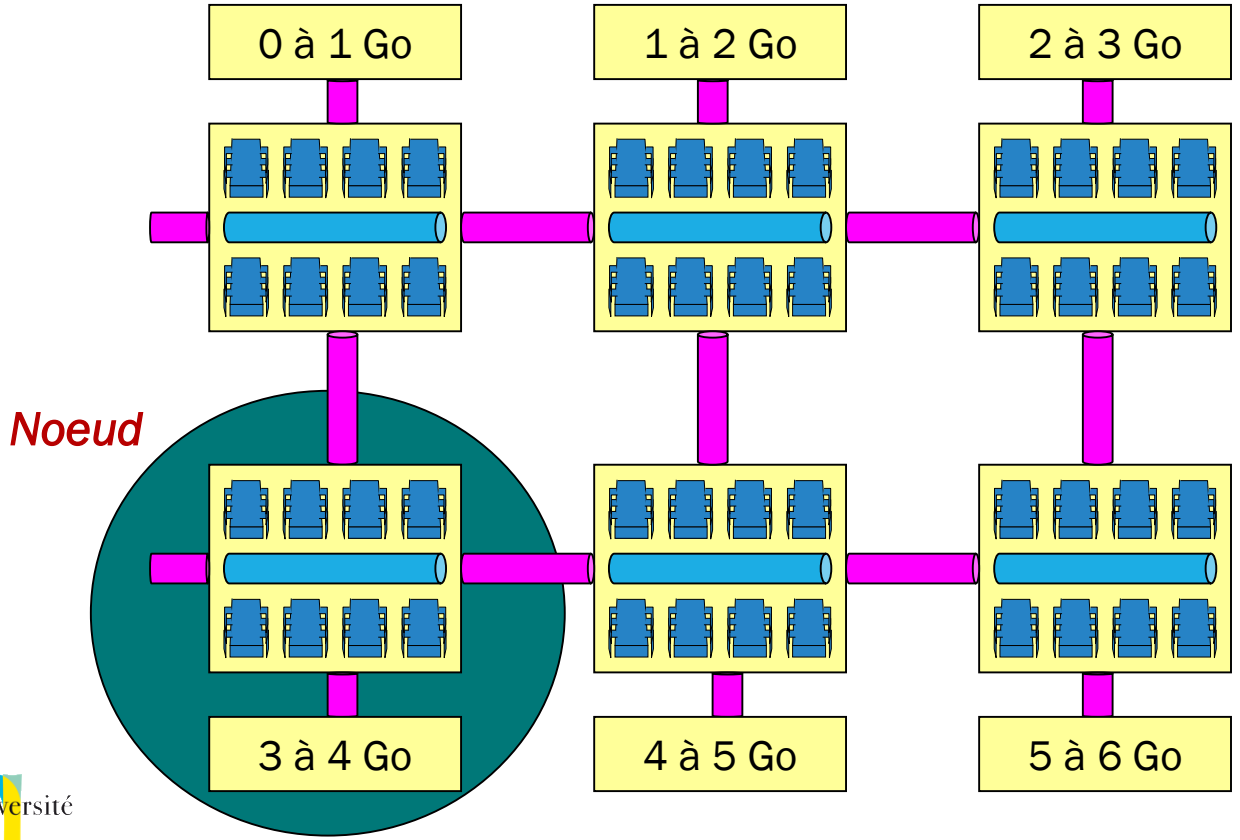
INVALIDATIONS

- **Solution :** partitionnement de l'espace d'adressage physique !



INVALIDATIONS

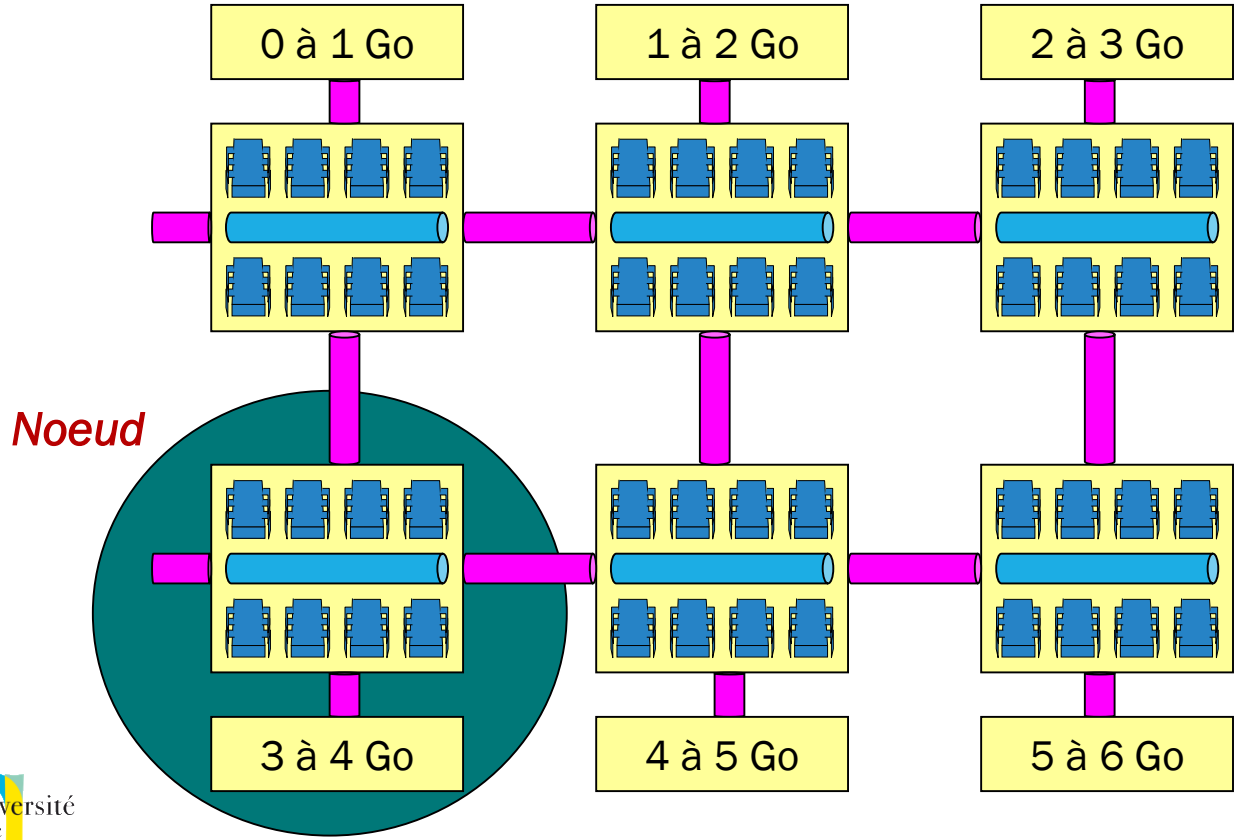
- **Solution :** partitionnement de l'espace d'adressage physique !



- Notion de **nœud** = ensemble de clusters gérant une partition mémoire

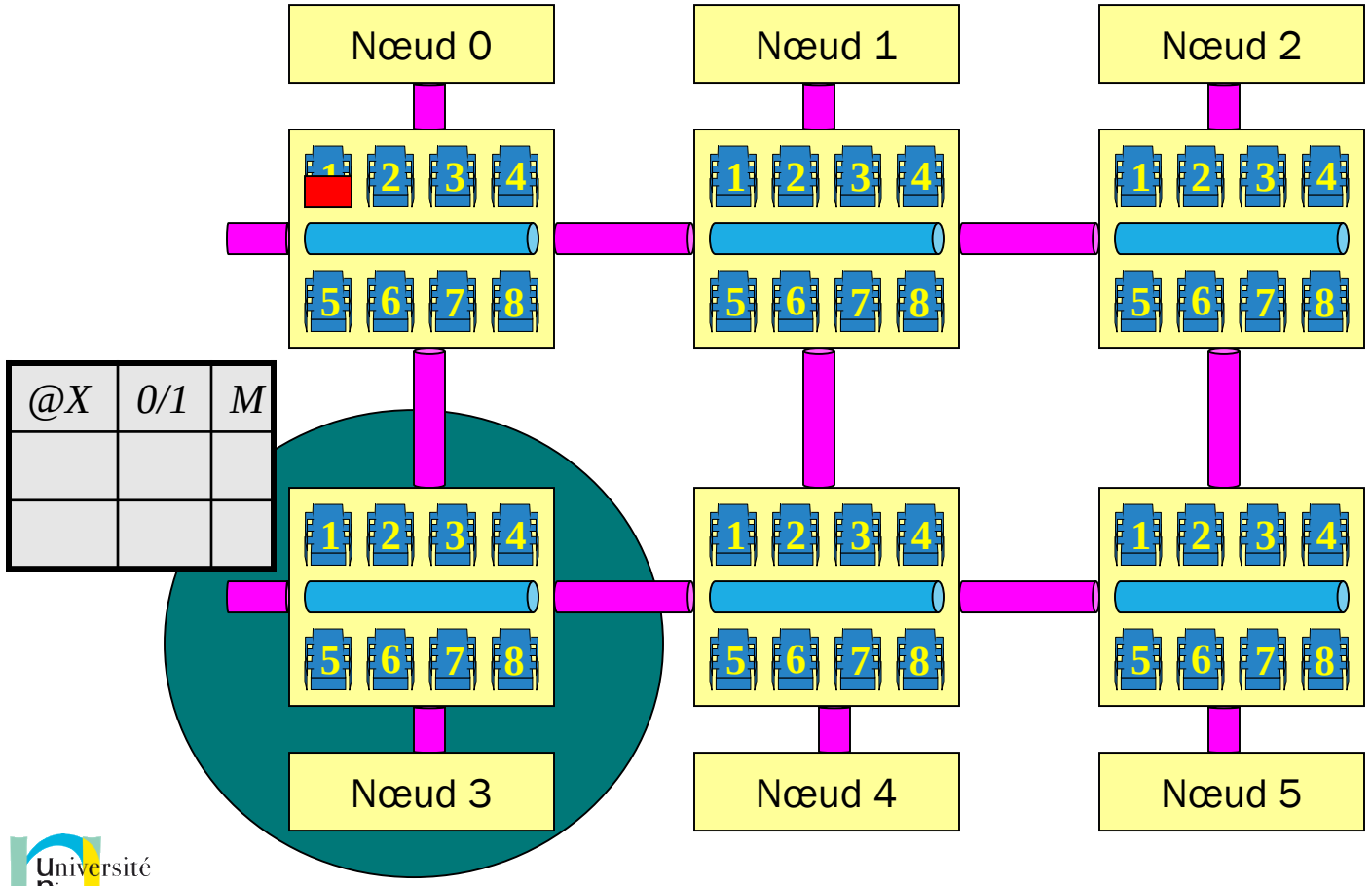
INVALIDATIONS

- **Solution :** partitionnement de l'espace d'adressage physique !



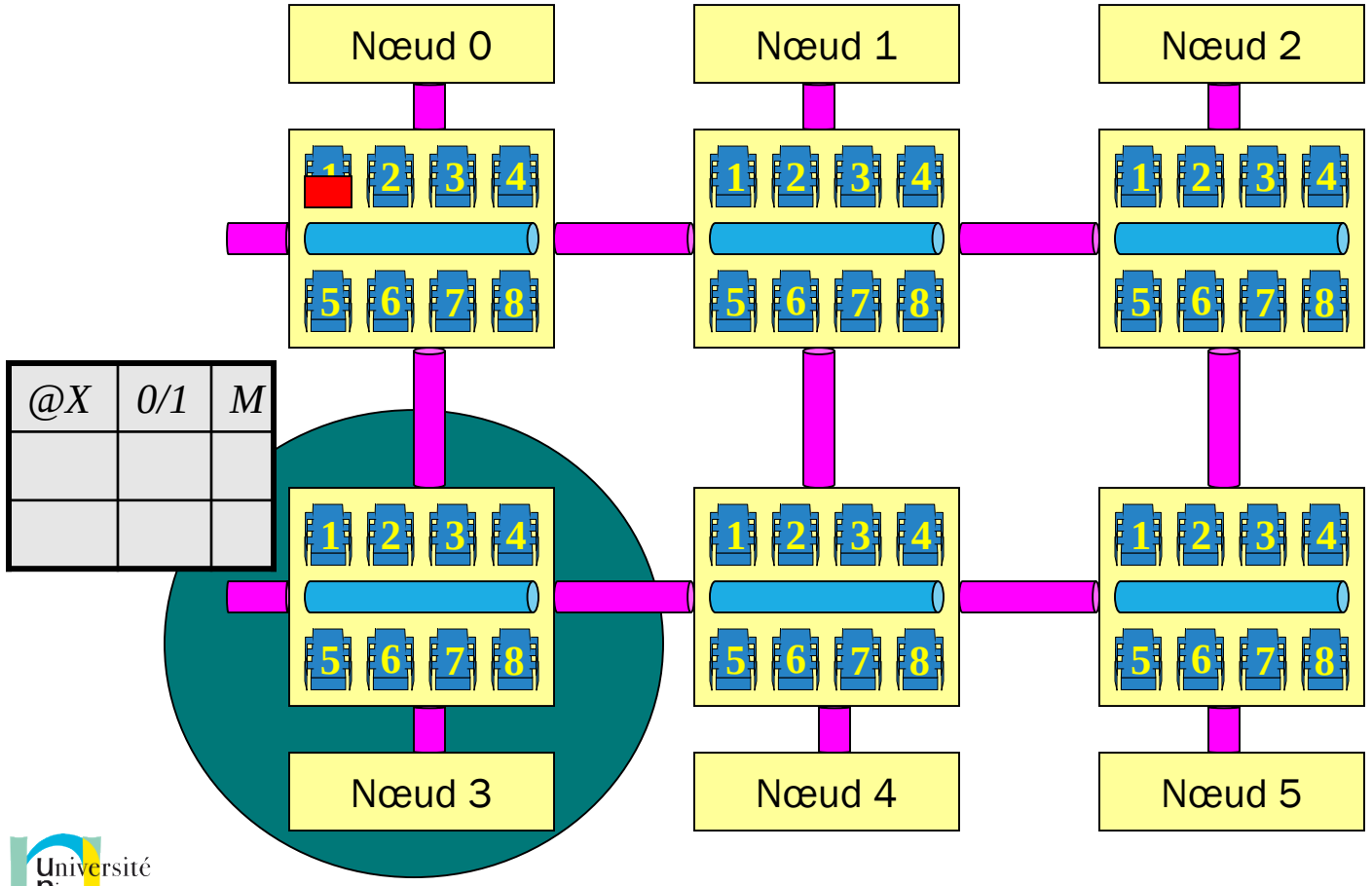
- Notion de **nœud** = ensemble de clusters gérant une partition mémoire
- **En général : un nœud = un die** ou **un nœud = un cluster**

INVALIDATIONS



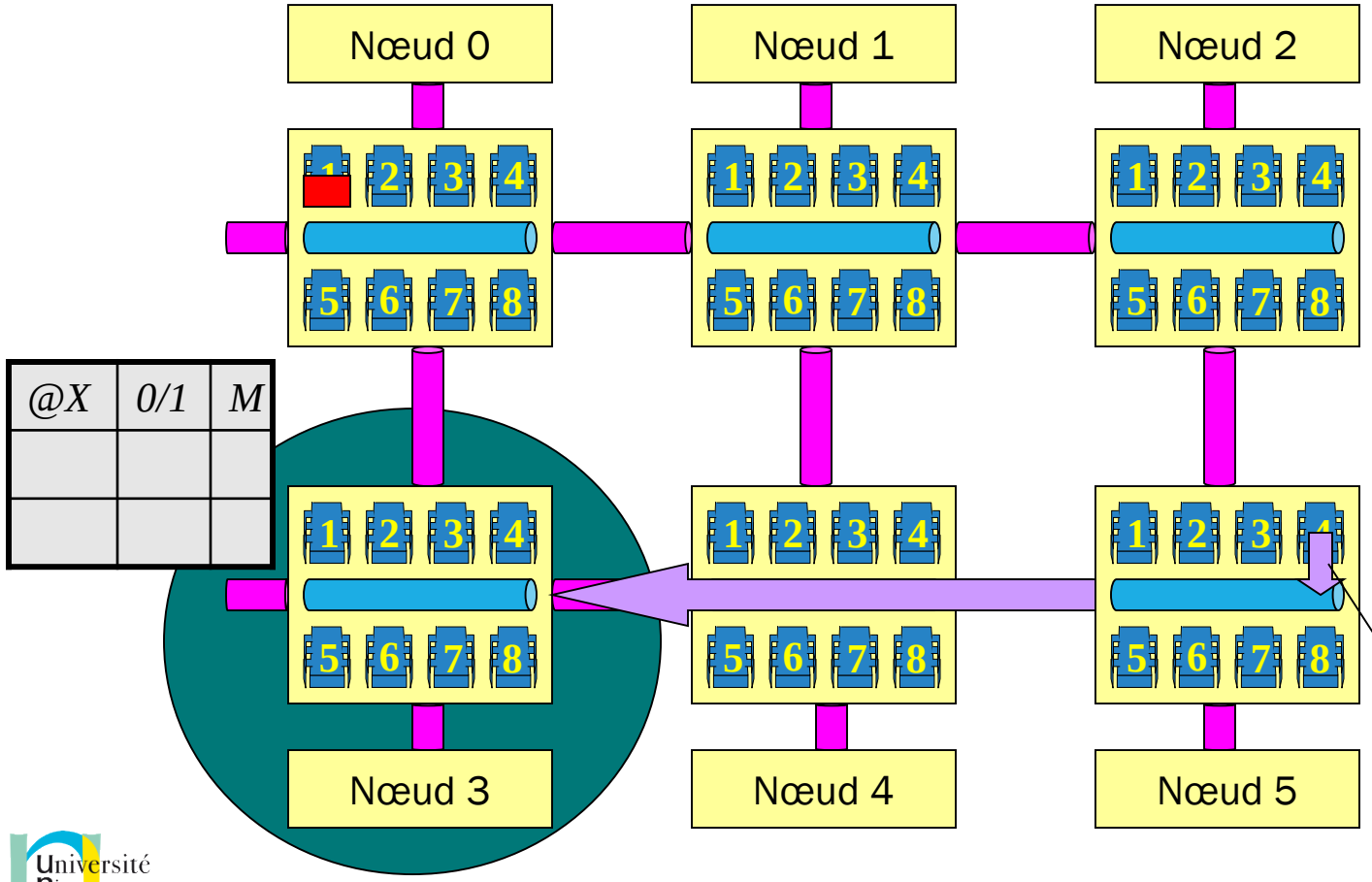
- **Exemple :**
  - Lecture de @X, par 5/4
  - Adresse gérée par nœud 3
  - Ligne de cache en M sur 0/1

INVALIDATIONS



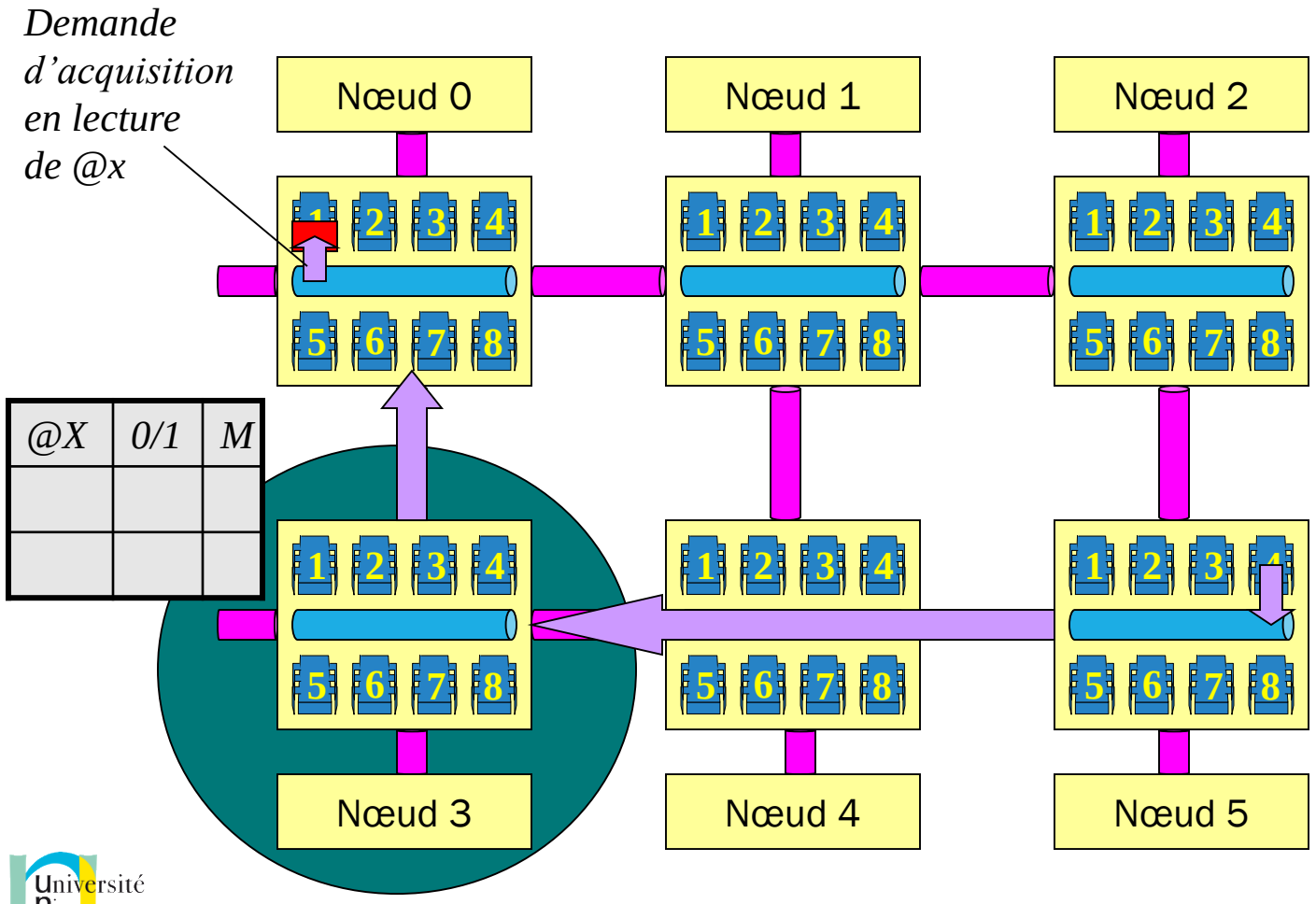
- **Exemple :**
  - Lecture de @X, par 5/4
  - Adresse gérée par nœud 3
  - Ligne de cache en M sur 0/1
- *La table du nœud 3 sait où la ligne de cache se trouve.*

INVALIDATIONS



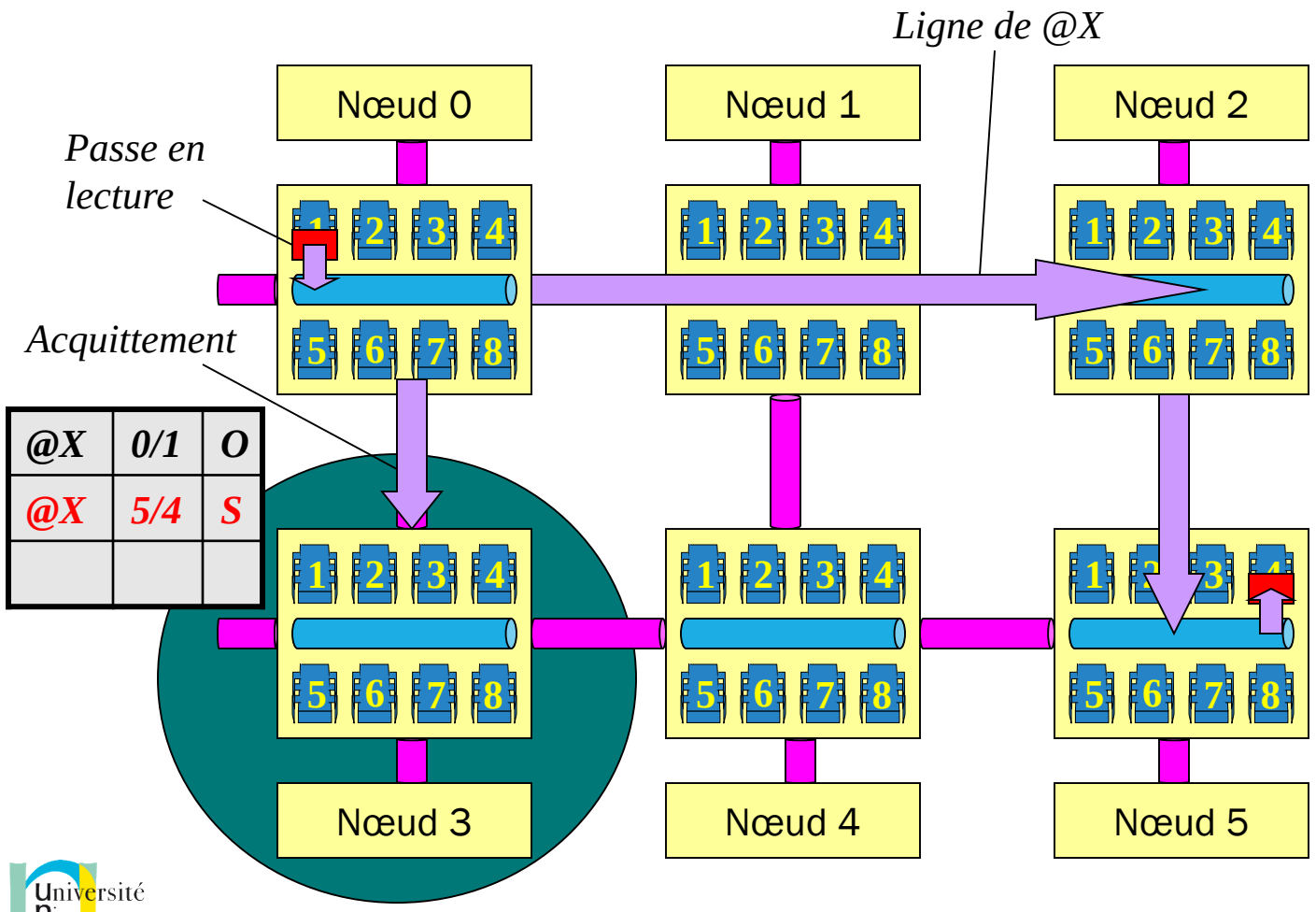
- **Exemple :**
  - Lecture de @X, par 5/4
  - Adresse gérée par nœud 3
  - Ligne de cache en M sur 0/1
- *Demande d'acquisition en lecture routée vers le nœud 3…*

INVALIDATIONS



- **Exemple :**
  - Lecture de @X, par 5/4
  - Adresse gérée par nœud 3
  - Ligne de cache en M sur 0/1
- *Le nœud 3 fait suivre au nœud 1 (car état M)…*

INVALIDATIONS



- **Exemple :**
  - Lecture de @X, par 5/4
  - Adresse gérée par nœud 3
  - Ligne de cache en M sur 0/1
- **La ligne sur 0/1 passe en état Owner, en accord avec le nœud 3 (msg+ack)**
- ***Puis, elle est envoyée à 5/4 directement, sans passer par la RAM !***
- *5/4 possède la ligne en Shared, suffisant pour la lecture*

UMA VS. NUMA

- **Le nombre de cœurs augmentant, généralement *réellement* partitionnée :**
  - Barrettes de RAM différentes pour chaque nœud, un contrôleur mémoire par nœud !

UMA VS. NUMA

- **Le nombre de cœurs augmentant, généralement *réellement* partitionnée :**
  - Barrettes de RAM différentes pour chaque nœud, un contrôleur mémoire par nœud !
  - **On parle de NUMA : Non-Uniform Memory Architecture…**

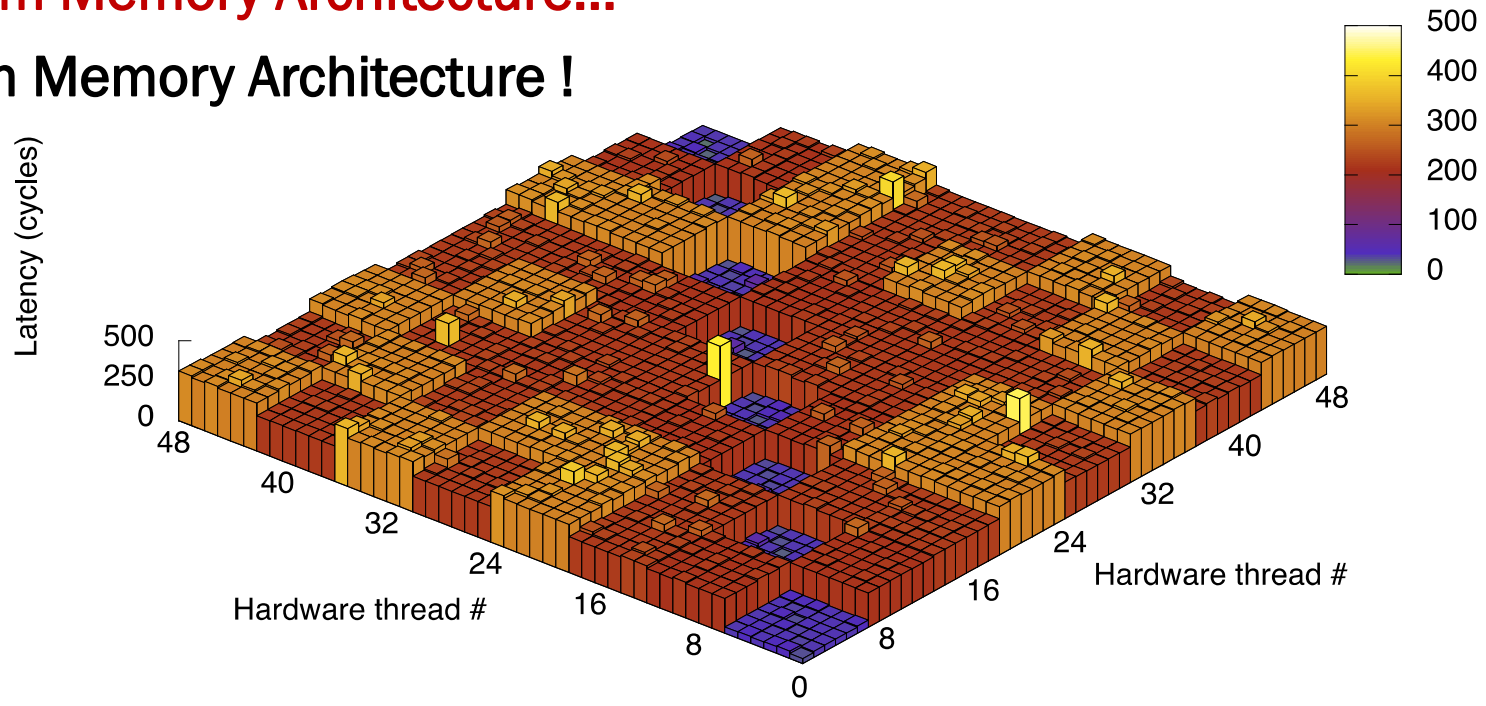
UMA VS. NUMA

- **Le nombre de cœurs augmentant, généralement *réellement* partitionnée :**
  - Barrettes de RAM différentes pour chaque nœud, un contrôleur mémoire par nœud !
  - **On parle de NUMA : Non-Uniform Memory Architecture…**
  - **En opposition au UMA : Uniform Memory Architecture !**

UMA VS. NUMA

- **Le nombre de cœurs augmentant, généralement *réellement* partitionnée :**
  - Barrettes de RAM différentes pour chaque nœud, un contrôleur mémoire par nœud !
  - **On parle de NUMA : Non-Uniform Memory Architecture…**
  - **En opposition au UMA : Uniform Memory Architecture !**

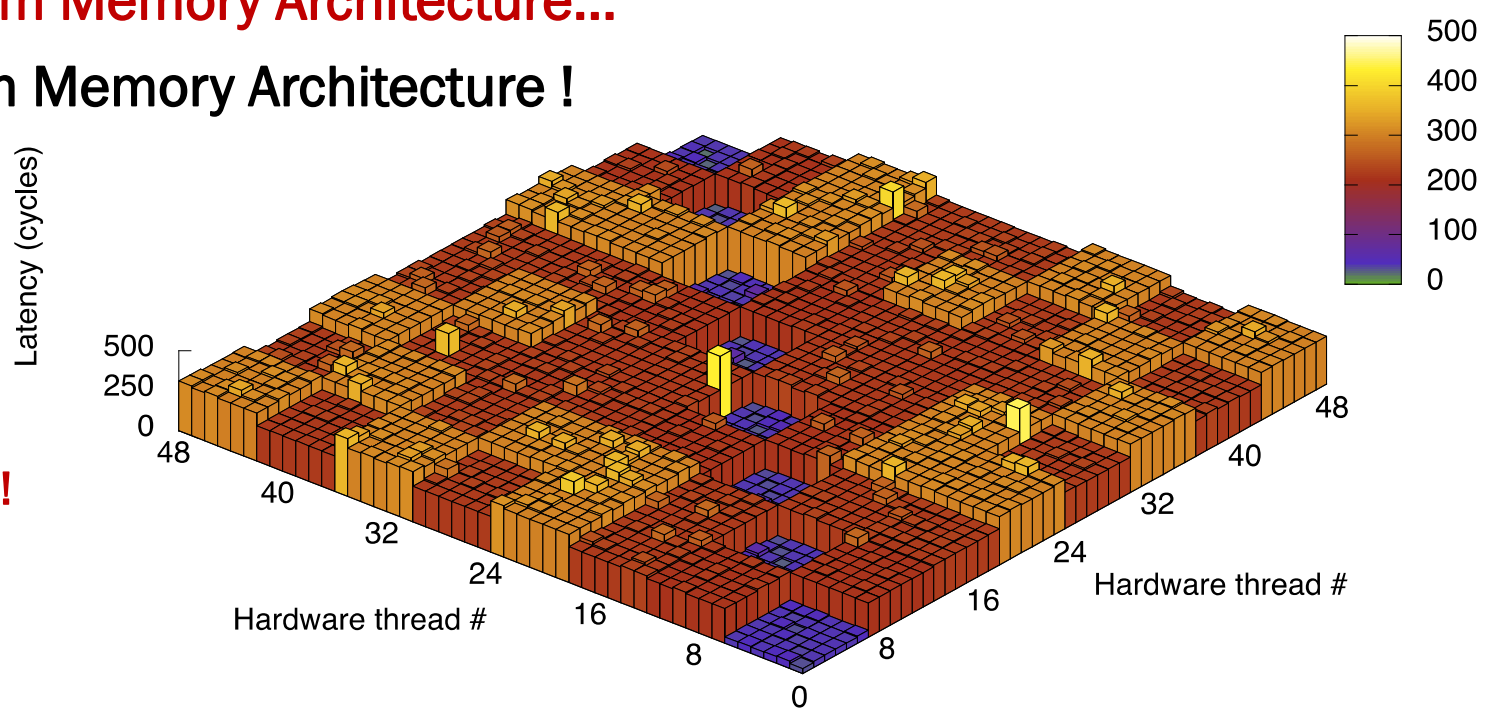
- **Différentes latences !**
  - Deux types d'accès : local et distant…



UMA VS. NUMA

- **Le nombre de cœurs augmentant, généralement *réellement* partitionnée :**
  - Barrettes de RAM différentes pour chaque nœud, un contrôleur mémoire par nœud !
  - **On parle de NUMA : Non-Uniform Memory Architecture…**
  - **En opposition au UMA : Uniform Memory Architecture !**

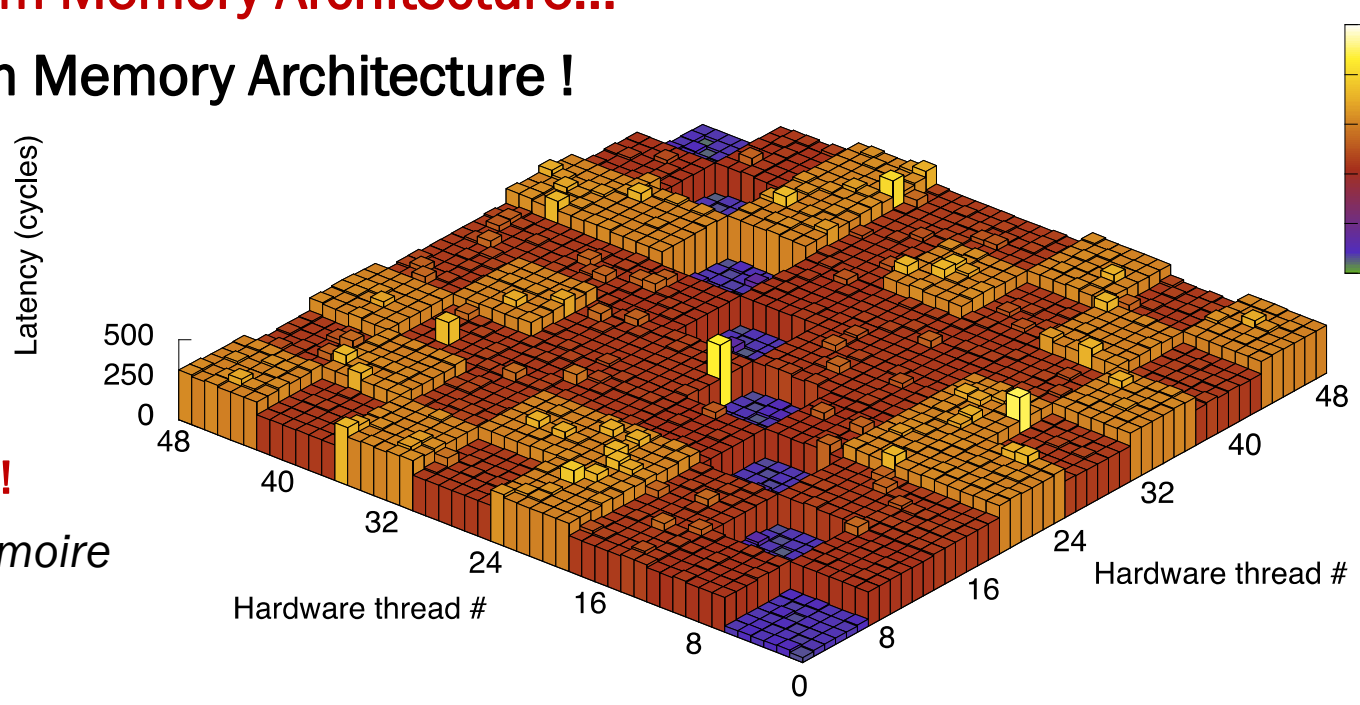
- **Différentes latences !**
  - Deux types d'accès : local et distant…
  - **En fait pire que ça : niveaux multiples !**



UMA VS. NUMA

- **Le nombre de cœurs augmentant, généralement *réellement* partitionnée :**
  - Barrettes de RAM différentes pour chaque nœud, un contrôleur mémoire par nœud !
  - **On parle de NUMA : Non-Uniform Memory Architecture…**
  - **En opposition au UMA : Uniform Memory Architecture !**

- **Différentes latences !**
  - Deux types d'accès : local et distant…
  - **En fait pire que ça : niveaux multiples !**
  - *Autant que possible : utiliser de la mémoire allouée localement…*

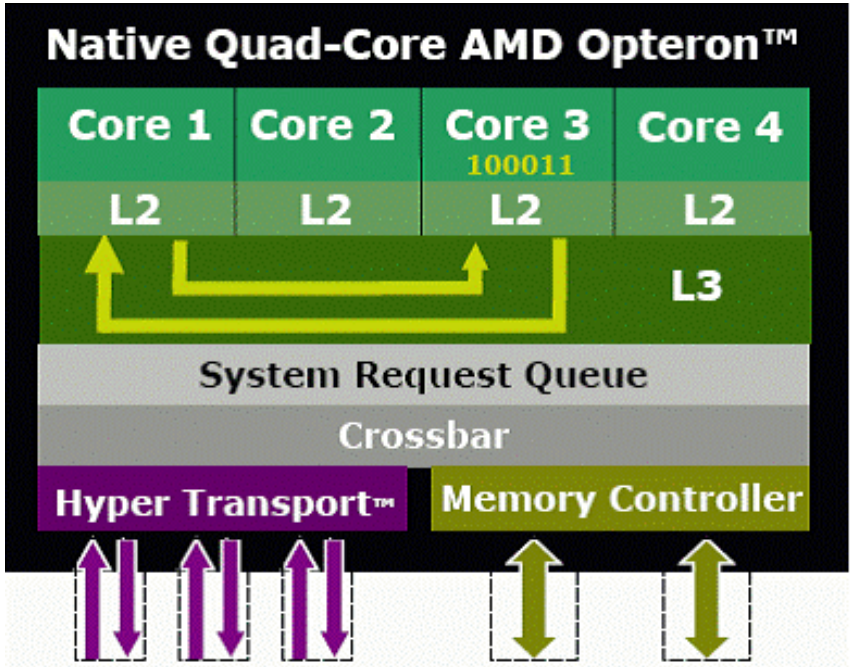


UMA VS. NUMA

- Exemple: comment deux cœurs qui partagent un cache communiquent-ils ?

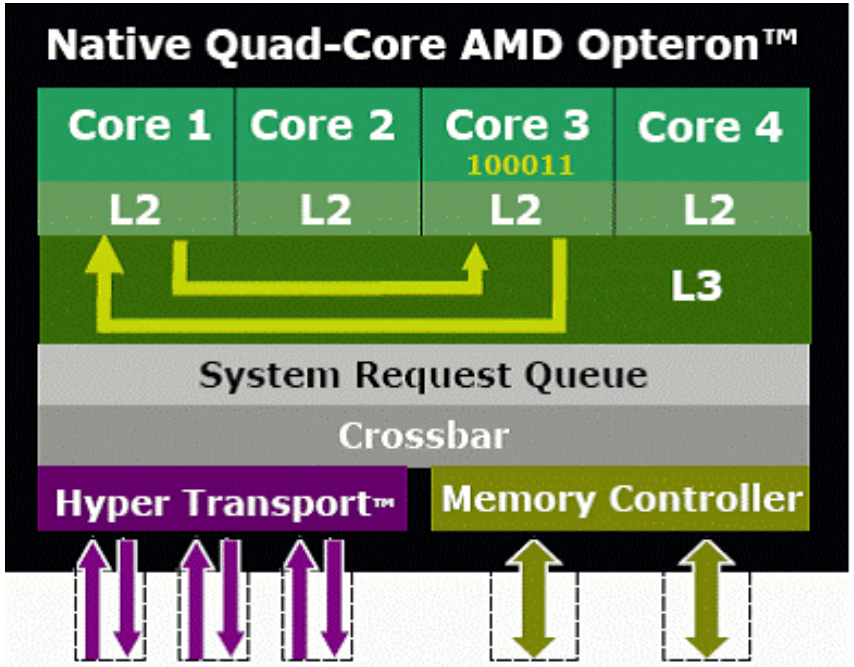
UMA VS. NUMA

- **Exemple: comment deux cœurs qui partagent un cache communiquent-ils ?**
  - Même chose, sauf que pas besoin de passer par l'interconnect !



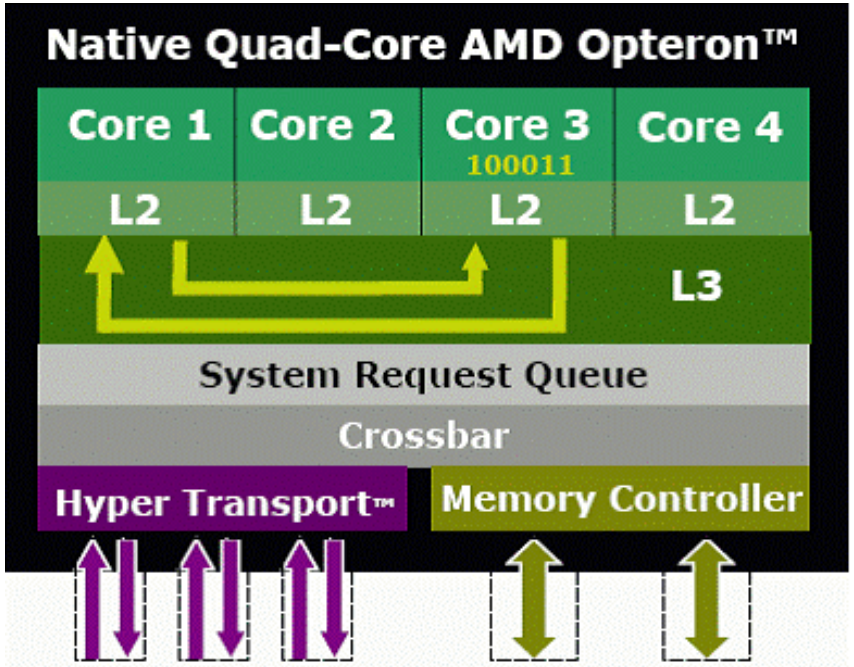
UMA VS. NUMA

- **Exemple: comment deux cœurs qui partagent un cache communiquent-ils ?**
  - Même chose, sauf que pas besoin de passer par l'interconnect !
  - Passage par le cache de niveau le plus bas qu'ils ont en commun…



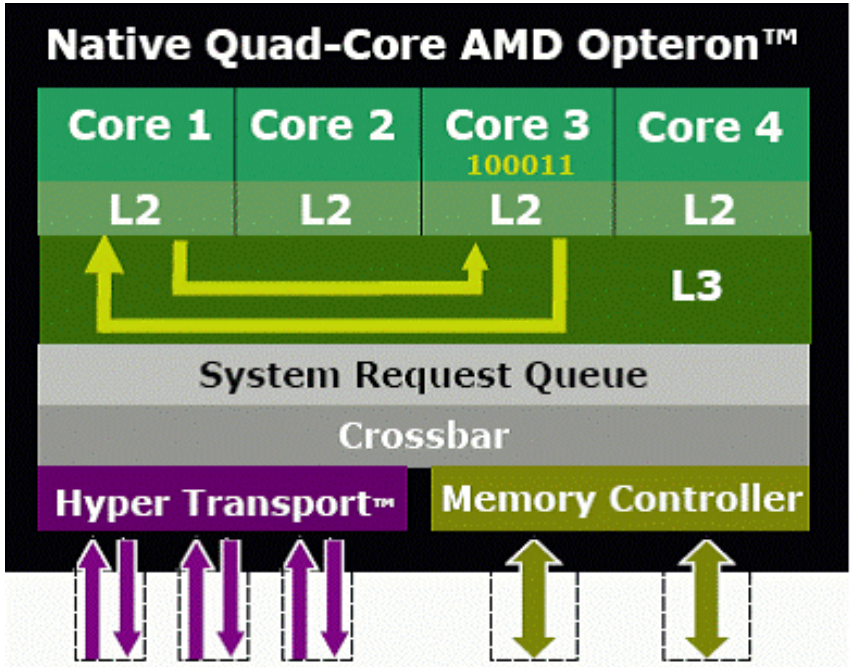
UMA VS. NUMA

- **Exemple: comment deux cœurs qui partagent un cache communiquent-ils ?**
  - Même chose, sauf que pas besoin de passer par l'interconnect !
  - Passage par le cache de niveau le plus bas qu'ils ont en commun…
- **Mais même dans ce cas, si mémoire allouée sur nœud distant, toujours besoin d'accéder à sa table !**



UMA VS. NUMA

- **Exemple: comment deux cœurs qui partagent un cache communiquent-ils ?**
  - Même chose, sauf que pas besoin de passer par l'interconnect !
  - Passage par le cache de niveau le plus bas qu'ils ont en commun…
- **Mais même dans ce cas, si mémoire allouée sur nœud distant, toujours besoin d'accéder à sa table !**
- ***Il faut donc faire très attention à ce que les allocations soient locales…***



GESTION DE LA MÉMOIRE SOUS LINUX

- **Objectif :** augmenter la localité d'accès (i.e. accès plutôt à la mémoire locale au nœud)

GESTION DE LA MÉMOIRE SOUS LINUX

- **Objectif :** augmenter la localité d'accès (i.e. accès plutôt à la mémoire locale au nœud)
- **Trois principes :**
  - Eviter de migrer un processus d'un nœud sur l'autre
  - Essayer de maintenir les threads d'un processus sur le même nœud

} *En pratique, ces deux points mal gérés…*

GESTION DE LA MÉMOIRE SOUS LINUX

- **Objectif :** augmenter la localité d'accès (i.e. accès plutôt à la mémoire locale au nœud)
- **Trois principes :**
  - Eviter de migrer un processus d'un nœud sur l'autre
  - Essayer de maintenir les threads d'un processus sur le même nœud

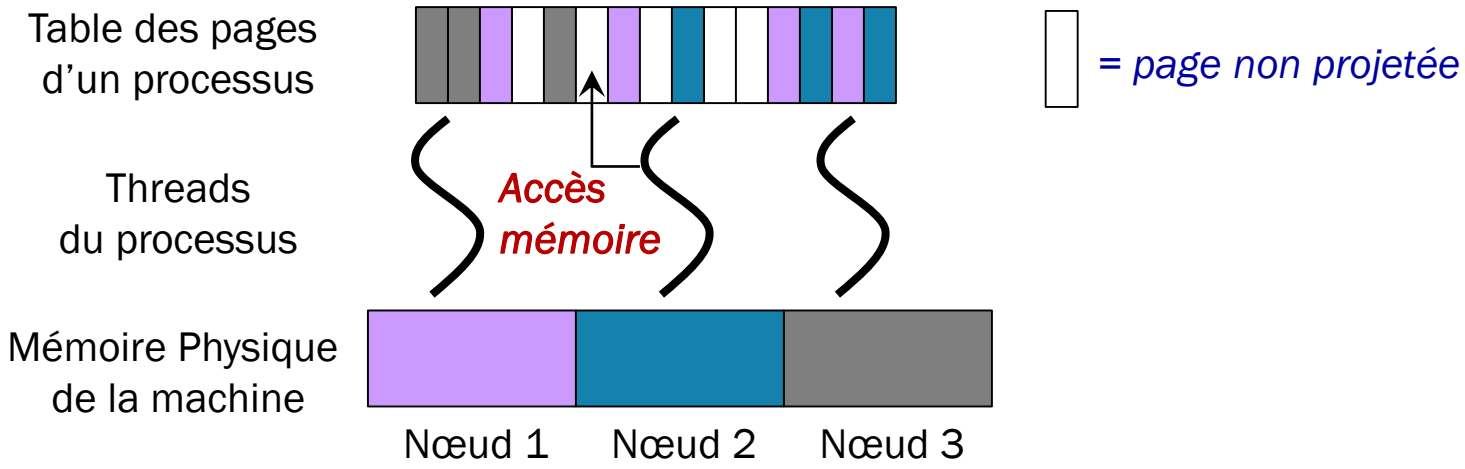
  } *En pratique, ces deux points mal gérés…*

  - **First-touch :** lors du premier accès à une page non projetée en mémoire, la page physique est allouée sur le nœud local.

GESTION DE LA MÉMOIRE SOUS LINUX

- **Objectif :** augmenter la localité d'accès (i.e. accès plutôt à la mémoire locale au nœud)
- **Trois principes :**
  - Eviter de migrer un processus d'un nœud sur l'autre
  - Essayer de maintenir les threads d'un processus sur le même nœud
  - **First-touch :** lors du premier accès à une page non projetée en mémoire, la page physique est allouée sur le nœud local.

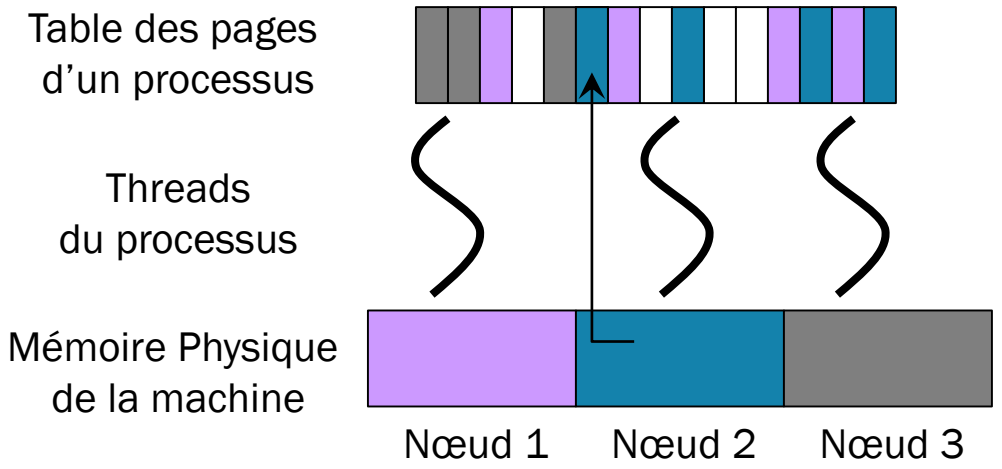
En pratique, ces deux points mal gérés…



GESTION DE LA MÉMOIRE SOUS LINUX

- **Objectif :** augmenter la localité d'accès (i.e. accès plutôt à la mémoire locale au nœud)
- **Trois principes :**
  - Eviter de migrer un processus d'un nœud sur l'autre
  - Essayer de maintenir les threads d'un processus sur le même nœud
  - **First-touch :** lors du premier accès à une page non projetée en mémoire, la page physique est allouée sur le nœud local.

En pratique, ces deux points mal gérés…



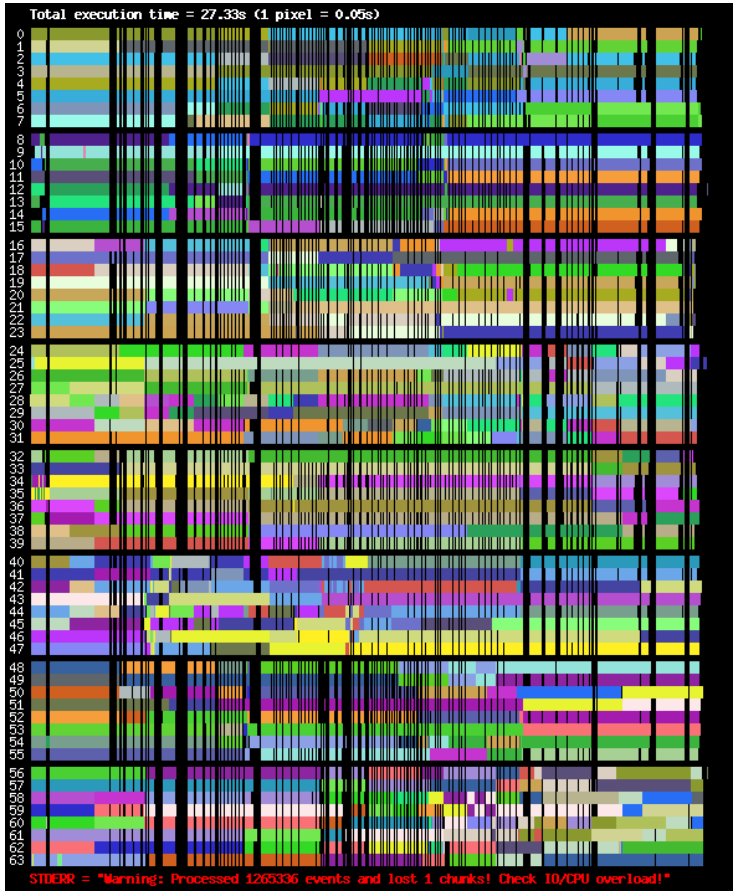
Allocation à partir du nœud 2 car thread s'exécute sur nœud 2

GESTION DE LA MÉMOIRE SOUS LINUX

- **En pratique :** gestion de la mémoire sous Linux pas très efficace sous NUMA

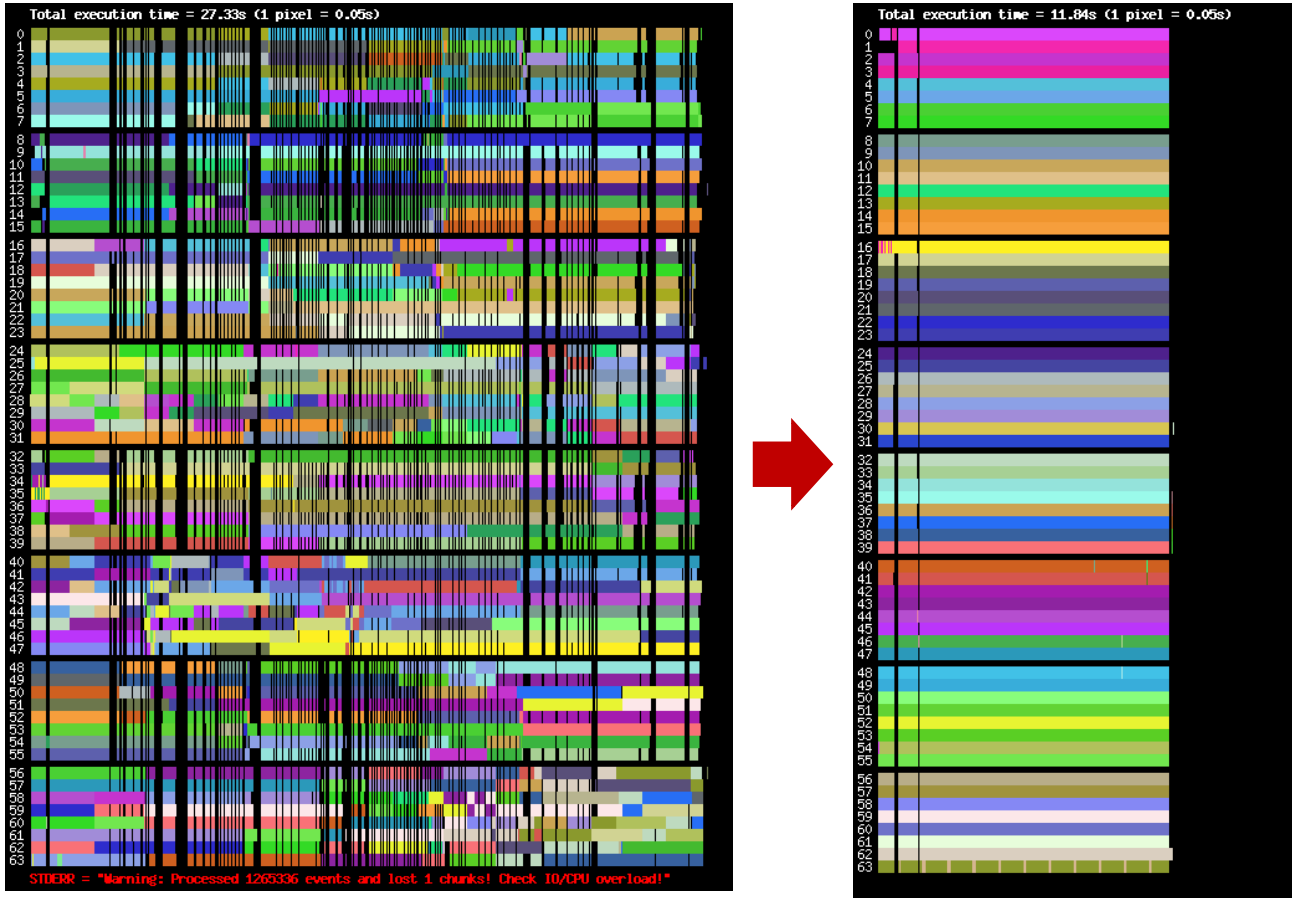
GESTION DE LA MÉMOIRE SOUS LINUX

- **En pratique :** gestion de la mémoire sous Linux pas très efficace sous NUMA
- **Linux migre les threads trop facilement**



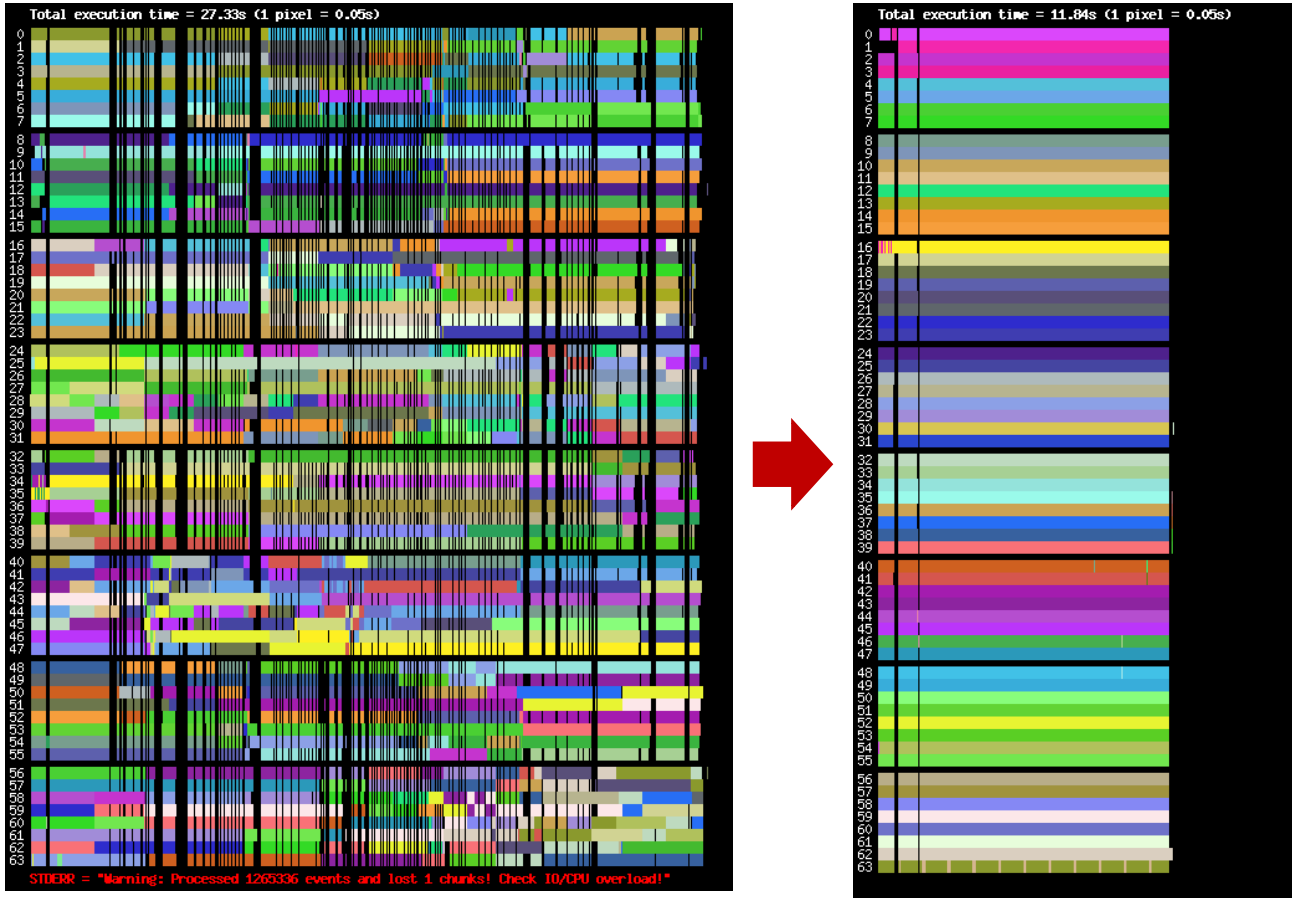
GESTION DE LA MÉMOIRE SOUS LINUX

- **En pratique :** gestion de la mémoire sous Linux pas très efficace sous NUMA
- **Linux migre les threads trop facilement**
- **Généralement plus efficace de fixer les threads sur des cœurs…**



GESTION DE LA MÉMOIRE SOUS LINUX

- **En pratique :** gestion de la mémoire sous Linux pas très efficace sous NUMA
- **Linux migre les threads trop facilement**
- **Généralement plus efficace de fixer les threads sur des cœurs…**
- **…et de faire attention à l'endroit où les allocations sont faites !**



GESTION DE LA MÉMOIRE SOUS LINUX

Quelques fonctions et commandes utiles…

GESTION DE LA MÉMOIRE SOUS LINUX

Quelques fonctions et commandes utiles…

- **Placement des processus et des threads :**
  - `setaffinity(<ensemble E de cœurs>)` : oblige l'ordonnanceur à placer les threads du processus sur un des cœurs de E.

GESTION DE LA MÉMOIRE SOUS LINUX

Quelques fonctions et commandes utiles…

- **Placement des processus et des threads :**
  - `setaffinity(<ensemble E de cœurs>)` : oblige l'ordonnanceur à placer les threads du processus sur un des cœurs de E.
  - `pthread_setaffinity(ensemble E de cœurs)` : idem, mais pour un thread.

GESTION DE LA MÉMOIRE SOUS LINUX

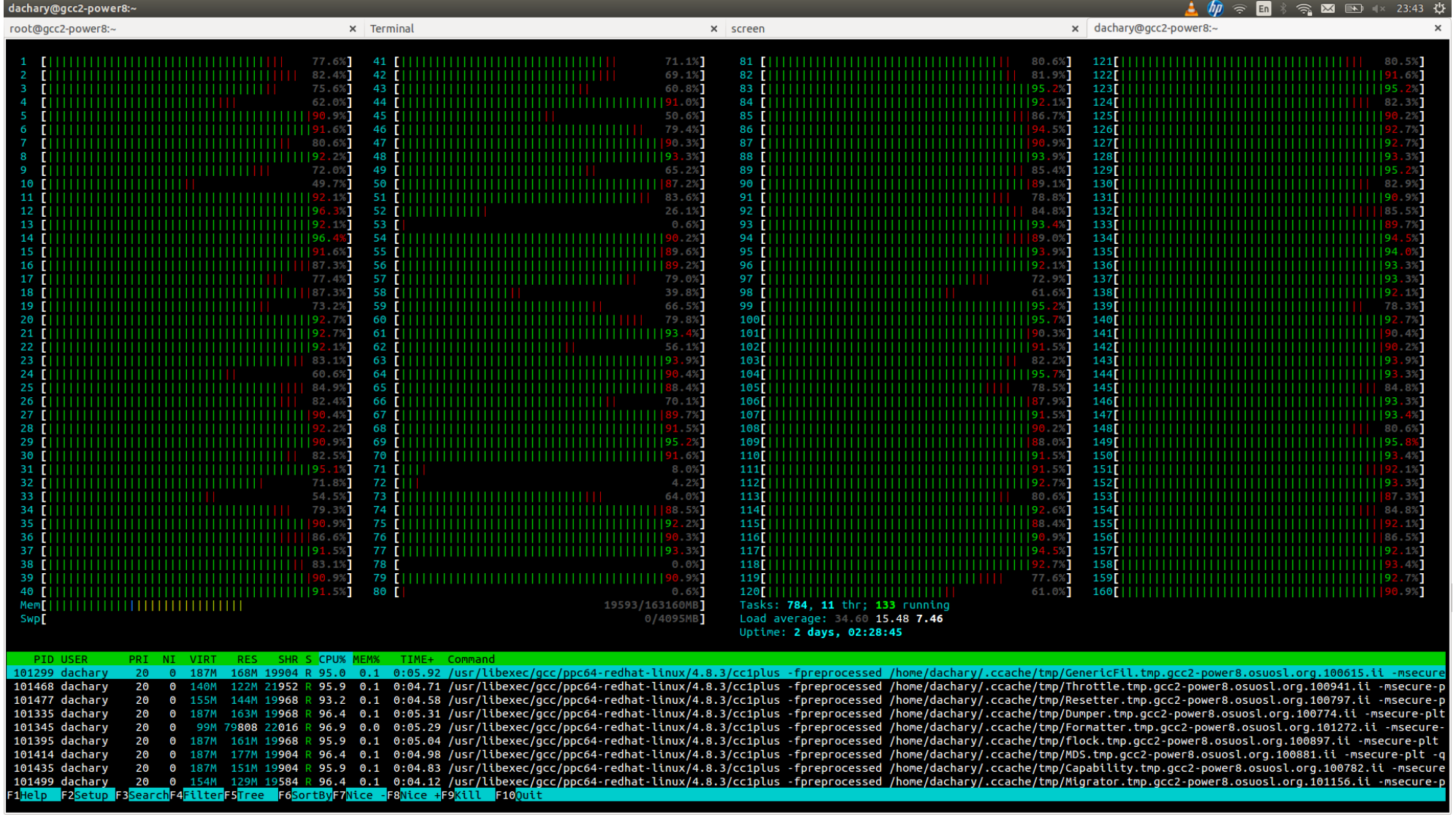
Quelques fonctions et commandes utiles…

- **Placement des processus et des threads :**
  - `setaffinity(<ensemble E de cœurs>)` : oblige l'ordonnanceur à placer les threads du processus sur un des cœurs de E.
  - `pthread_setaffinity(ensemble E de cœurs)` : idem, mais pour un thread.
  - Peut se faire en ligne de commande avec `taskset`, e.g. :
    `taskset –c 0-3 ./benchmark`
    Lance benchmark en le plaçant sur les cœurs 0 à 3.

GESTION DE LA MÉMOIRE SOUS LINUX

Quelques fonctions et commandes utiles…

- **Placement des processus et des threads :**
  - `setaffinity(<ensemble E de cœurs>)` : oblige l'ordonnanceur à placer les threads du processus sur un des cœurs de E.
  - `pthread_setaffinity(ensemble E de cœurs)` : idem, mais pour un thread.
  - Peut se faire en ligne de commande avec `taskset`, e.g. :
    `taskset –c 0-3 ./benchmark`
    Lance benchmark en le plaçant sur les cœurs 0 à 3.
- **Utiliser htop au lieu de top !** (si installé…)
  - Bien plus facile de voir où les threads tournent…



threads du .

- **Utiliser htop au lieu de top !** (si installé...)
  - Bien plus facile de voir où les threads tournent...

GESTION DE LA MÉMOIRE SOUS LINUX

Quelques fonctions et commandes utiles…

GESTION DE LA MÉMOIRE SOUS LINUX

Quelques fonctions et commandes utiles…

- **Placement de pages (utile en NUMA seulement) :**
  - mbind(<plage d'adresses virtuelle>, <ensemble de nœuds>) : demande à n'allouer la mémoire physique des pages qu'à partir de l'ensemble de nœuds

GESTION DE LA MÉMOIRE SOUS LINUX

Quelques fonctions et commandes utiles…

- **Placement de pages (utile en NUMA seulement) :**
  - mbind(<plage d'adresses virtuelle>, <ensemble de nœuds>) : demande à n'allouer la mémoire physique des pages qu'à partir de l'ensemble de nœuds
  - madvise(<plage d'adresses virtuelle>, <flag>) : demande à libérer les pages physique de la plage d'adresse (permet de migrer).

GESTION DE LA MÉMOIRE SOUS LINUX

Quelques fonctions et commandes utiles…

- **Placement de pages (utile en NUMA seulement) :**
  - mbind(<plage d'adresses virtuelle>, <ensemble de nœuds>) : demande à n'allouer la mémoire physique des pages qu'à partir de l'ensemble de nœuds
  - madvise(<plage d'adresses virtuelle>, <flag>) : demande à libérer les pages physique de la plage d'adresse (permet de migrer).
- **Allocation mémoire (utile en NUMA seulement) :**
  - numa_alloc_onnode(<taille>, <nœud N>) : malloc() sur le nœud N

GESTION DE LA MÉMOIRE SOUS LINUX

Quelques fonctions et commandes utiles…

- **Placement de pages (utile en NUMA seulement) :**
  - mbind(<plage d'adresses virtuelle>, <ensemble de nœuds>) : demande à n'allouer la mémoire physique des pages qu'à partir de l'ensemble de nœuds
  - madvise(<plage d'adresses virtuelle>, <flag>) : demande à libérer les pages physique de la plage d'adresse (permet de migrer).
- **Allocation mémoire (utile en NUMA seulement) :**
  - numa_alloc_onnode(<taille>, <nœud N>) : malloc() sur le nœud N
  - numa_alloc_local(<taille>) : pareil mais sur le nœud local

GESTION DE LA MÉMOIRE SOUS LINUX

Quelques fonctions et commandes utiles…

- **Placement de pages (utile en NUMA seulement) :**
  - mbind(<plage d'adresses virtuelle>, <ensemble de nœuds>) : demande à n'allouer la mémoire physique des pages qu'à partir de l'ensemble de nœuds
  - madvise(<plage d'adresses virtuelle>, <flag>) : demande à libérer les pages physique de la plage d'adresse (permet de migrer).
- **Allocation mémoire (utile en NUMA seulement) :**
  - numa_alloc_onnode(<taille>, <nœud N>) : malloc() sur le nœud N
  - numa_alloc_local(<taille>) : pareil mais sur le nœud local
  - Fonctions de libnuma, il y en a beaucoup d'autres…

LE « FALSE SHARING »

- Unité minimale gérée par le protocole de cohérence de cache : lignes de cache

LE « FALSE SHARING »

- **Unité minimale gérée par le protocole de cohérence de cache : lignes de cache**
- Contiennent plusieurs variables : 64 ou 128 octets, i.e. 8 à 16 long long (`int64_t`)

LE « FALSE SHARING »

- **Unité minimale gérée par le protocole de cohérence de cache : lignes de cache**
- Contiennent plusieurs variables : 64 ou 128 octets, i.e. 8 à 16 long long (int64_t)
- Supposons qu'on a (déclaration globale) :

```
struct foo { int x; int y; };
```

LE « FALSE SHARING »

- **Unité minimale gérée par le protocole de cohérence de cache : lignes de cache**
- Contiennent plusieurs variables : 64 ou 128 octets, i.e. 8 à 16 long long (`int64_t`)
- Supposons qu'on a (déclaration globale) :

```
struct foo { int x; int y; };
```

- **Si thread 1/cœur 0 écrit souvent dans x, et thread 2/cœur 40 écrivent souvent dans y :**
  - La ligne de cache fait du ping-pong entre les caches du cœur 0 et 40 !
  - À chaque écriture, la ligne de cache doit être envoyée, puis changement d'état vers Owner…
  - Alors que travail sur des variables indépendantes, tout ça inutile !

LE « FALSE SHARING »

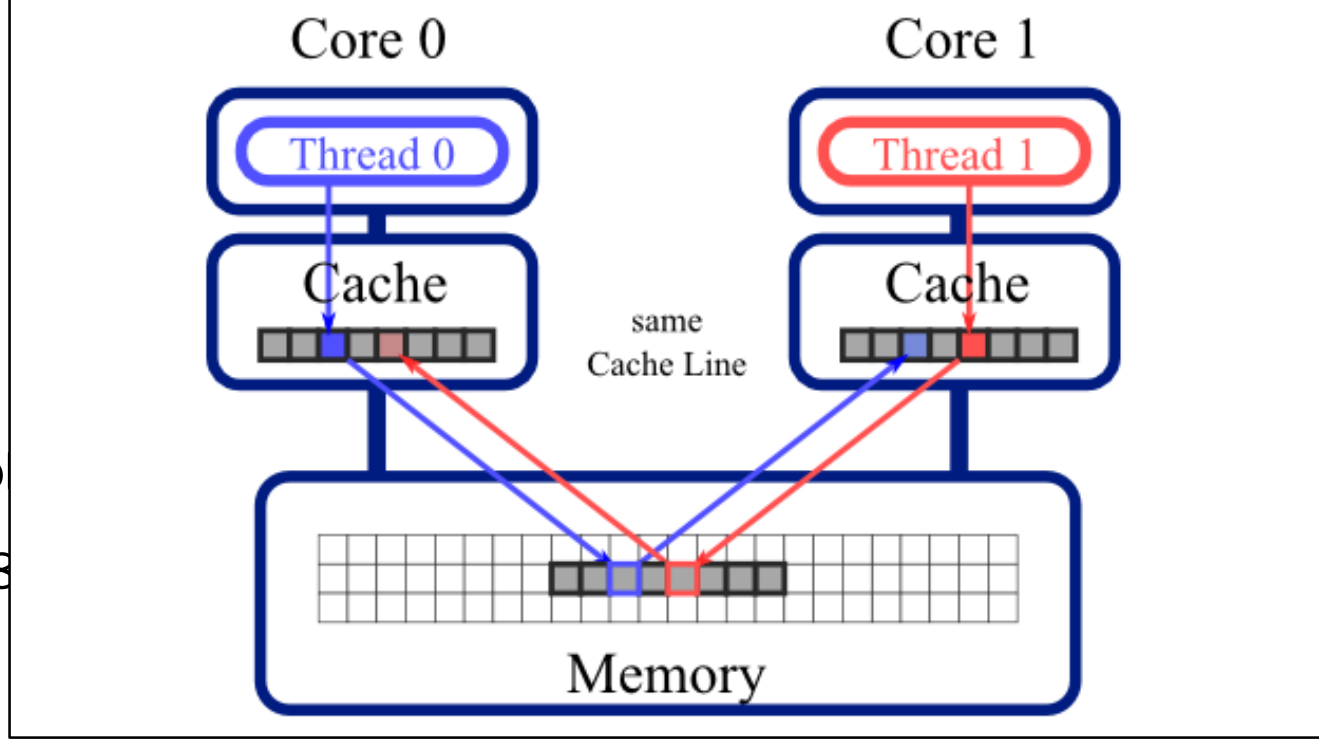
- **Unité minimale gérée par le protocole de cohérence de cache : lignes de cache**
- Contiennent plusieurs variables : 64 ou 128 octets, i.e. 8 à 16 long long (int64_t)
- Supposons qu'on a (déclaration globale) :

```
struct foo { int x; int y; };
```

- **Si thread 1/cœur 0 écrit souvent dans x, et thread 2/cœur 40 écrivent souvent dans y :**
  - **La ligne de cache fait du ping-pong entre les caches du cœur 0 et 40 !**
  - À chaque écriture, la ligne de cache doit être envoyée, puis changement d'état vers Owner…
  - Alors que travail sur des variables indépendantes, tout ça inutile !
- **Solution : ajouter du padding…**

```
struct foo { int x; char padding[64]; int y; };
```

LE « FALSE SHARING »



- **Unité minimale gérée par le protocole de co**
- Contiennent plusieurs variables : 64 ou 128
- Supposons qu'on a (déclaration globale) :

```
struct foo { int x; int y; };
```

- **Si thread 1/cœur 0 écrit souvent dans x, et thread 2/cœur 40 écrivent souvent dans y :**
  - **La ligne de cache fait du ping-pong entre les caches du cœur 0 et 40 !**
  - À chaque écriture, la ligne de cache doit être envoyée, puis changement d'état vers Owner…
  - Alors que travail sur des variables indépendantes, tout ça inutile !
- **Solution : ajouter du padding…**

```
struct foo { int x; char padding[64]; int y; };
```

GARANTIES D'ORDRE

- **Lorsqu'on écrit du code concurrent, les instructions peuvent être réordonnées :**
  - Par le compilateur C, **ou par le hardware lui-même (out-of-order execution)**

GARANTIES D'ORDRE

- **Lorsqu'on écrit du code concurrent, les instructions peuvent être réordonnées :**
  - Par le compilateur C, **ou par le hardware lui-même (out-of-order execution)**
  - Besoin de garanties : si des lectures/écritures se croisent entre cœurs, que lit-on ?

GARANTIES D'ORDRE

- **Lorsqu'on écrit du code concurrent, les instructions peuvent être réordonnées :**
  - Par le compilateur C, **ou par le hardware lui-même (out-of-order execution)**
  - Besoin de garanties : si des lectures/écritures se croisent entre cœurs, que lit-on ?
- **Plusieurs modèles de cohérence :**
  - **Séquentielle :** toutes lectures et écritures effectuées dans l'ordre

GARANTIES D'ORDRE

- **Lorsqu'on écrit du code concurrent, les instructions peuvent être réordonnées :**
  - Par le compilateur C, **ou par le hardware lui-même (out-of-order execution)**
  - Besoin de garanties : si des lectures/écritures se croisent entre cœurs, que lit-on ?
- **Plusieurs modèles de cohérence :**
  - **Séquentielle :** toutes lectures et écritures effectuées dans l'ordre
  - **Relachée :** plusieurs niveaux, possibles réordonnancement des **lectures après lectures**, **lectures après écritures**, **écritures après lectures**, et/ou **écritures après écritures**

GARANTIES D'ORDRE

- **Lorsqu'on écrit du code concurrent, les instructions peuvent être réordonnées :**
  - Par le compilateur C, **ou par le hardware lui-même (out-of-order execution)**
  - Besoin de garanties : si des lectures/écritures se croisent entre cœurs, que lit-on ?
- **Plusieurs modèles de cohérence :**
  - **Séquentielle :** toutes lectures et écritures effectuées dans l'ordre
  - **Relachée :** plusieurs niveaux, possibles réordonnancement des **lectures après lectures**, **lectures après écritures**, **écritures après lectures**, et/ou **écritures après écritures**
  - **Faible :** tout peut être réordonné ! Besoin de barrières mémoire pour assurer tout ordre.

GARANTIES D'ORDRE

- **Lorsqu'on écrit du code concurrent, les instructions peuvent être réordonnées :**
  - Par le compilateur C, **ou par le hardware lui-même (out-of-order execution)**
  - Besoin de garanties : si des lectures/écritures se croisent entre cœurs, que lit-on ?
- **Plusieurs modèles de cohérence :**
  - **Séquentielle :** toutes lectures et écritures effectuées dans l'ordre
  - **Relachée :** plusieurs niveaux, possibles réordonnancement des **lectures après lectures**, **lectures après écritures**, **écritures après lectures**, et/ou **écritures après écritures**
  - **Faible :** tout peut être réordonné ! Besoin de barrières mémoire pour assurer tout ordre.
- **Sous x86, on a de la chance. Total Store Order (TSO), i.e. seul réordonnancement possible : écritures poussées après les lectures !**

GARANTIES D'ORDRE

- **Lorsqu'on écrit du code concurrent, les instructions peuvent être réordonnées :**
  - Par le compilateur C, **ou par le hardware lui-même (out-of-order execution)**
  - Besoin de garanties : si des lectures/écritures se croisent entre cœurs, que lit-on ?
- **Plusieurs modèles de cohérence :**
  - **Séquentielle :** toutes lectures et écritures effectuées dans l'ordre
  - **Relachée :** plusieurs niveaux, possibles réordonnancement des **lectures après lectures**, **lectures après écritures**, **écritures après lectures**, et/ou **écritures après écritures**
  - **Faible :** tout peut être réordonné ! Besoin de barrières mémoire pour assurer tout ordre.
- **Sous x86, on a de la chance. Total Store Order (TSO), i.e. seul réordonnancement possible : écritures poussées après les lectures !**
- **Attention : si vous voulez écrire du code portable, RISC, ARM, etc., ont une cohérence faible !**

Memory ordering in some architectures[7][8]

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC RMO	SPARC PSO	SPARC TSO	x86	x86 oostore	AMD64	IA-64	zSeries
Loads reordered after loads	Y	Y	Y	Y	Y				Y		Y	
Loads reordered after stores	Y	Y	Y	Y	Y				Y		Y	
Stores reordered after stores	Y	Y	Y	Y	Y	Y			Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with loads	Y	Y		Y	Y						Y	
Atomic reordered with stores	Y	Y		Y	Y	Y					Y	
Dependent loads reordered	Y											
Incoherent instruction cache pipeline	Y	Y		Y	Y	Y	Y	Y	Y		Y	Y

- **Plusieurs modèles de cohérence :**
  - **Séquentielle :** toutes lectures et écritures effectuées dans l'ordre
  - **Relachée :** plusieurs niveaux, possibles réordonnancement des **lectures après lectures**, **lectures après écritures**, **écritures après lectures**, et/ou **écritures après écritures**
  - **Faible :** tout peut être réordonné ! Besoin de barrières mémoire pour assurer tout ordre.
- **Sous x86, on a de la chance. Total Store Order (TSO), i.e. seul réordonnancement possible : écritures poussées après les lectures !**
- **Attention : si vous voulez écrire du code portable, RISC, ARM, etc., ont une cohérence faible !**

GARANTIES D'ORDRE

- **Un principe commun à tous les modèles mémoire (utile !) :**
  - *En local, toute lecture succédant à une écriture lit la dernière valeur écrite*

```
write x, value
read x -> lit la valeur value
```

GARANTIES D'ORDRE

- **Un principe commun à tous les modèles mémoire (utile !) :**
  - *En local, toute lecture succédant à une écriture lit la dernière valeur écrite*

```
write x, value
read x -> lit la valeur value
```

GARANTIES D'ORDRE

- Sous x86, on a de la chance. Total Store Order (TSO), i.e. seul réordonnancement possible : **écritures poussées après les lectures !** => *Qu'est-ce que ça veut dire ?*

GARANTIES D'ORDRE

- Sous x86, on a de la chance. Total Store Order (TSO), i.e. seul réordonnancement possible : **écritures poussées après les lectures !** => *Qu'est-ce que ça veut dire ?*
- *Si on a une écriture suivie par des lectures qui sont indépendantes de l'écriture, les lectures peuvent être réalisées avant l'écriture en attendant que l'écriture obtienne sa ligne de cache.*

GARANTIES D'ORDRE

- Sous x86, on a de la chance. Total Store Order (TSO), i.e. seul réordonnancement possible : **écritures poussées après les lectures !** => *Qu'est-ce que ça veut dire ?*
- *Si on a une écriture suivie par des lectures qui sont indépendantes de l'écriture, les lectures peuvent être réalisées avant l'écriture en attendant que l'écriture obtienne sa ligne de cache.*
- **Cas problématique :** initialement, [x] = [y] = 0

CPU0:	CPU1:
store [x]<--1	store [y]<--1
load r1<--[y]	load r2<--[x]

GARANTIES D'ORDRE

- Sous x86, on a de la chance. Total Store Order (TSO), i.e. seul réordonnancement possible : **écritures poussées après les lectures !** => *Qu'est-ce que ça veut dire ?*
- *Si on a une écriture suivie par des lectures qui sont indépendantes de l'écriture, les lectures peuvent être réalisées avant l'écriture en attendant que l'écriture obtienne sa ligne de cache.*
- **Cas problématique :** initialement, [x] = [y] = 0

CPU0:	CPU1:
store [x]<--1	store [y]<--1
load r1<--[y]	load r2<--[x]

Inversion possible sur CPU1 !

GARANTIES D'ORDRE

- Sous x86, on a de la chance. Total Store Order (TSO), i.e. seul réordonnancement possible : **écritures poussées après les lectures !** => *Qu'est-ce que ça veut dire ?*
- *Si on a une écriture suivie par des lectures qui sont indépendantes de l'écriture, les lectures peuvent être réalisées avant l'écriture en attendant que l'écriture obtienne sa ligne de cache.*
- **Cas problématique :** initialement, [x] = [y] = 0

CPU0:	CPU1:
store [x]<--1	store [y]<--1
load r1<--[y]	load r2<--[x]

Inversion possible sur CPU1 !

- Sans réordonnancement ici, on a {r1, r2} = {1, 0}, {0, 1} ou {1, 1}
  - « CPU1 avant », « CPU1 après », ou « entrelacé »

GARANTIES D'ORDRE

- Sous x86, on a de la chance. Total Store Order (TSO), i.e. seul réordonnancement possible : **écritures poussées après les lectures !** *=> Qu'est-ce que ça veut dire ?*

- *Si on a une écriture suivie par des lectures qui sont indépendantes de l'écriture, les lectures peuvent être réalisées avant l'écriture en attendant que l'écriture obtienne sa ligne de cache.*

- **Cas problématique :** initialement, [x] = [y] = 0

CPU0:	CPU1:
store [x]<--1	store [y]<--1
load r1<--[y]	load r2<--[x]

Inversion possible sur CPU1 !

- Sans réordonnancement ici, on a {r1, r2} = {1, 0}, {0, 1} ou {1, 1}
  - « CPU1 avant », « CPU1 après », ou « entrelacé »

- *Avec réordonnancement, on peut avoir {r1, r2} = {0, 0}… Etrange et imprévu !*

GARANTIES D'ORDRE

- Solution aux réordonnancements intempestifs : barrières mémoire (memory barriers/fences)

GARANTIES D'ORDRE

- Solution aux réordonnancements intempestifs : barrières mémoire (memory barriers/fences)
- **Premier type :** barrière mémoire soft, empêche réordonnancements du compilateur autour (gcc) :
  ```
  asm volatile("" ::: "memory");
  ```

GARANTIES D'ORDRE

- Solution aux réordonnancements intempestifs : barrières mémoire (memory barriers/fences)
- **Premier type :** barrière mémoire soft, empêche réordonnancements du compilateur autour (gcc) :
  `asm volatile("" ::: "memory");`
- **Deuxième type :** barrières mémoire hard
  - `_mm_mfence()` : empêche tout réordonnancement d'instructions hard autour de la barrière

GARANTIES D'ORDRE

- Solution aux réordonnancements intempestifs : barrières mémoire (memory barriers/fences)
- **Premier type :** barrière mémoire soft, empêche réordonnancements du compilateur autour (gcc) : `asm volatile("" ::: "memory");`
- **Deuxième type :** barrières mémoire hard
  - `_mm_mfence()` : empêche tout réordonnancement d'instructions hard autour de la barrière
  - `_mm_lfence()` : empêche réord. des lectures avant avec lectures/écritures après (LoadLoad + LoadStore)

GARANTIES D'ORDRE

- Solution aux réordonnancements intempestifs : barrières mémoire (memory barriers/fences)
- **Premier type :** barrière mémoire soft, empêche réordonnancements du compilateur autour (gcc) :
  `asm volatile("" ::: "memory");`
- **Deuxième type :** barrières mémoire hard
  - _mm_mfence() : empêche tout réordonnancement d'instructions hard autour de la barrière
  - _mm_lfence() : empêche réord. des lectures avant avec lectures/écritures après (LoadLoad + LoadStore)
  - _mm_sfence() : empêche réord. des écritures avant avec les écritures après (StoreStore)

GARANTIES D'ORDRE

- Solution aux réordonnancements intempestifs : barrières mémoire (memory barriers/fences)
- **Premier type :** barrière mémoire soft, empêche réordonnancements du compilateur autour (gcc) : `asm volatile("" ::: "memory");`
- **Deuxième type :** barrières mémoire hard
  - `_mm_mfence()` : empêche tout réordonnancement d'instructions hard autour de la barrière
  - `_mm_lfence()` : empêche réord. des lectures avant avec lectures/écritures après (LoadLoad + LoadStore)
  - `_mm_sfence()` : empêche réord. des écritures avant avec les écritures après (StoreStore)
  - **Du coup, `LFENCE` et `SFENCE` « inutiles » en x86 (on a besoin de StoreLoad) !** On n'utilise généralement que `MFENCE` (en fait il y a des subtilités, allez fouiller sur StackOverflow).

GARANTIES D'ORDRE

- Solution aux réordonnancements intempestifs : barrières mémoire (memory barriers/fences)
- **Premier type :** barrière mémoire soft, empêche réordonnancements du compilateur autour (gcc) : `asm volatile("" ::: "memory");`
- **Deuxième type :** barrières mémoire hard
  - `_mm_mfence()` : empêche tout réordonnancement d'instructions hard autour de la barrière
  - `_mm_lfence()` : empêche réord. des lectures avant avec lectures/écritures après (LoadLoad + LoadStore)
  - `_mm_sfence()` : empêche réord. des écritures avant avec les écritures après (StoreStore)
  - **Du coup, `LFENCE` et `SFENCE` « inutiles » en x86 (on a besoin de StoreLoad) !** On n'utilise généralement que `MFENCE` (en fait il y a des subtilités, allez fouiller sur StackOverflow).
- **Troisième type :** le builtin gcc `__sync_synchronize()`
  - **Barrière soft + hard (`MFENCE`). Empêche tout ordonnancement.**

GARANTIES D'ORDRE

- Attention : ne pas être paranoïaque avec les barrières hard !
  - Instructions coûteuses !

GARANTIES D'ORDRE

- Attention : ne pas être paranoïaque avec les barrières hard !
  - Instructions coûteuses !
  - Sur x86, en fait rarement utiles… sauf cas subtils !

GARANTIES D'ORDRE

- Attention : ne pas être paranoïaque avec les barrières hard !
  - Instructions coûteuses !
  - Sur x86, en fait rarement utiles… sauf cas subtils !
- Variables `volatile` :
  - Certains qu'elles sont toujours lues de la mémoire/caches et pas d'un registre

GARANTIES D'ORDRE

- Attention : ne pas être paranoïaque avec les barrières hard !
  - **Instructions coûteuses !**
  - Sur x86, en fait rarement utiles… sauf cas subtils !
- Variables `volatile` :
  - Certains qu'elles sont toujours lues de la mémoire/caches et pas d'un registre
    - *Peut paraître utile si on manipule une variable qui peut être modifiée d'ailleurs…*

GARANTIES D'ORDRE

- Attention : ne pas être paranoïaque avec les barrières hard !
  - **Instructions coûteuses !**
  - Sur x86, en fait rarement utiles… sauf cas subtils !
- Variables `volatile` :
  - Certains qu'elles sont toujours lues de la mémoire/caches et pas d'un registre
    - *Peut paraître utile si on manipule une variable qui peut être modifiée d'ailleurs…*
  - Pas de réordonnancement des lectures et écritures des variables `volatile`

GARANTIES D'ORDRE

- Attention : ne pas être paranoïaque avec les barrières hard !
  - **Instructions coûteuses !**
  - Sur x86, en fait rarement utiles… sauf cas subtils !
- Variables `volatile` :
  - Certains qu'elles sont toujours lues de la mémoire/caches et pas d'un registre
    - *Peut paraître utile si on manipule une variable qui peut être modifiée d'ailleurs…*
  - Pas de réordonnancement des lectures et écritures des variables `volatile`
- **En fait, on peut généralement se passer de** `volatile`**…**
  - Pour empêcher les réordonnancements, il vaut mieux utiliser une barrière au bon moment, plus rapide, et marche contre toutes les autres variables, pas seulement `volatile` !

GARANTIES D'ORDRE

- Attention : ne pas être paranoïaque avec les barrières hard !
  - **Instructions coûteuses !**
  - Sur x86, en fait rarement utiles… sauf cas subtils !
- Variables `volatile` :
  - Certains qu'elles sont toujours lues de la mémoire/caches et pas d'un registre
    - *Peut paraître utile si on manipule une variable qui peut être modifiée d'ailleurs…*
  - Pas de réordonnancement des lectures et écritures des variables `volatile`
- **En fait, on peut généralement se passer de** `volatile`**…**
  - Pour empêcher les réordonnancements, il vaut mieux utiliser une barrière au bon moment, plus rapide, et marche contre toutes les autres variables, pas seulement `volatile` !
  - **La barrière (même soft) force le flush des registres, donc pas de soucis !**

GARANTIES D'ORDRE

- Un cas où il peut être tentant d'utiliser **volatile** : attente active

```
while (my_variable != what_i_want)
        ;
```

GARANTIES D'ORDRE

- Un cas où il peut être tentant d'utiliser **volatile** : attente active

```
while (my_variable != what_i_want)
        ;
```

- Ici, my_variable stocké dans un registre, mise à jour distante manquée si non volatile…

GARANTIES D'ORDRE

- Un cas où il peut être tentant d'utiliser **volatile** : attente active

```
while (my_variable != what_i_want)
        ;
```

- Ici, my_variable stocké dans un registre, mise à jour distante manquée si non volatile…
- En fait, lorsqu'on fait de l'attente active, on doit *toujours* utiliser l'instruction **PAUSE** :

```
while (my_variable != what_i_want)
        _mm_pause();
```

GARANTIES D'ORDRE

- Un cas où il peut être tentant d'utiliser volatile : attente active

```
while (my_variable != what_i_want)
        ;
```

- Ici, my_variable stocké dans un registre, mise à jour distante manquée si non volatile…
- En fait, lorsqu'on fait de l'attente active, on doit *toujours* utiliser l'instruction PAUSE :

```
while (my_variable != what_i_want)
        _mm_pause();
```

- Il s'agit d'un « spin loop hint », qui entre autres, joue le rôle d'une barrière : registres flushés

GARANTIES D'ORDRE

- **Un cas où il peut être tentant d'utiliser volatile : attente active**

```
while (my_variable != what_i_want)
        ;
```

- Ici, my_variable stocké dans un registre, mise à jour distante manquée si non volatile…
- **En fait, lorsqu'on fait de l'attente active, on doit *toujours* utiliser l'instruction PAUSE :**

```
while (my_variable != what_i_want)
        _mm_pause();
```

- Il s'agit d'un « spin loop hint », qui entre autres, joue le rôle d'une barrière : registres flushés
- **Autres optimisations :** peut spinner moins vite, économiser de l'énergie, passer la main à un hyperthread, désactive la prédiction de branche pour éviter un flush du pipeline à la sortie…

AVANCÉES « RÉCENTES » EN OS MULTICŒUR

- **Linux (et autres OS à mémoire partagée) :**
  - Tous les cœurs partagent leur mémoire, communication par mémoire partagée/lock

AVANCÉES « RÉCENTES » EN OS MULTICŒUR

- **Linux (et autres OS à mémoire partagée) :**
  - Tous les cœurs partagent leur mémoire, communication par mémoire partagée/lock
  - Verrouillage à grain fin (plus de « Big Kernel Lock » !)

AVANCÉES « RÉCENTES » EN OS MULTICŒUR

- **Linux (et autres OS à mémoire partagée) :**
  - Tous les cœurs partagent leur mémoire, communication par mémoire partagée/lock
  - Verrouillage à grain fin (plus de « Big Kernel Lock » !)
  - Nombreuses optimisations (compteurs répliqués par cœur, RCU, etc.)

AVANCÉES « RÉCENTES » EN OS MULTICŒUR

- **Linux (et autres OS à mémoire partagée) :**
  - Tous les cœurs partagent leur mémoire, communication par mémoire partagée/lock
  - Verrouillage à grain fin (plus de « Big Kernel Lock » !)
  - Nombreuses optimisations (compteurs répliqués par cœur, RCU, etc.)
  - **Performances bonnes jusqu'à 48 cœurs ! [Boyd-Wickizer, OSDI 2010]**

AVANCÉES « RÉCENTES » EN OS MULTICŒUR

- **Linux (et autres OS à mémoire partagée) :**
  - Tous les cœurs partagent leur mémoire, communication par mémoire partagée/lock
  - Verrouillage à grain fin (plus de « Big Kernel Lock » !)
  - Nombreuses optimisations (compteurs répliqués par cœur, RCU, etc.)
  - **Performances bonnes jusqu'à 48 cœurs ! [Boyd-Wickizer, OSDI 2010]**
- **Multikernel (Barrelfish [Baumann, SOSP 2009], Helios [Nightingale, SOSP 2009]) :**
  - Pas de mémoire physique partagée entre les cœurs dans l'OS

AVANCÉES « RÉCENTES » EN OS MULTICŒUR

- **Linux (et autres OS à mémoire partagée) :**
  - Tous les cœurs partagent leur mémoire, communication par mémoire partagée/lock
  - Verrouillage à grain fin (plus de « Big Kernel Lock » !)
  - Nombreuses optimisations (compteurs répliqués par cœur, RCU, etc.)
  - **Performances bonnes jusqu'à 48 cœurs ! [Boyd-Wickizer, OSDI 2010]**
- **Multikernel (Barrelfish [Baumann, SOSP 2009], Helios [Nightingale, SOSP 2009]) :**
  - Pas de mémoire physique partagée entre les cœurs dans l'OS
  - **Communication par envoi de message (évite la contention sur lignes de cache)**

AVANCÉES « RÉCENTES » EN OS MULTICŒUR

- **Linux (et autres OS à mémoire partagée) :**
  - Tous les cœurs partagent leur mémoire, communication par mémoire partagée/lock
  - Verrouillage à grain fin (plus de « Big Kernel Lock » !)
  - Nombreuses optimisations (compteurs répliqués par cœur, RCU, etc.)
  - **Performances bonnes jusqu'à 48 cœurs ! [Boyd-Wickizer, OSDI 2010]**
- **Multikernel (Barrelfish [Baumann, SOSP 2009], Helios [Nightingale, SOSP 2009]) :**
  - Pas de mémoire physique partagée entre les cœurs dans l'OS
  - **Communication par envoi de message (évite la contention sur lignes de cache)**
- **Exemple de problème :** scheduler devient complexe quand nombreux cœurs…
  - Difficile d'écrire un scheduler prouvé correct, qui marche dans tous les cas…

AVANCÉES « RÉCENTES » EN OS MULTICŒUR

- **Linux (et autres OS à mémoire partagée) :**
  - Tous les cœurs partagent leur mémoire, communication par mémoire partagée/lock
  - Verrouillage à grain fin (plus de « Big Kernel Lock » !)
  - Nombreuses optimisations (compteurs répliqués par cœur, RCU, etc.)
  - **Performances bonnes jusqu'à 48 cœurs ! [Boyd-Wickizer, OSDI 2010]**
- **Multikernel (Barrelfish [Baumann, SOSP 2009], Helios [Nightingale, SOSP 2009]) :**
  - Pas de mémoire physique partagée entre les cœurs dans l'OS
  - **Communication par envoi de message (évite la contention sur lignes de cache)**
- **Exemple de problème :** scheduler devient complexe quand nombreux cœurs…
  - Difficile d'écrire un scheduler prouvé correct, qui marche dans tous les cas…
  - *Voir présentation au cours no. 4*