NMV : Topologie mémoire

Julien Sopena - julien.sopena@lip6.fr
Gauthier Voron - gauthier.voron@lip6.fr

Topologie mémoire : pourquoi

Code source	Temps d'exécution

`for (i = 0; i < WIDTH; i++)`
`  for (j = 0; j < HEIGHT; j++)`
`  array[j * WIDTH + i]++;`
	16.64 secondes

`for (j = 0; j < HEIGHT; j++)`
`  for (i = 0; i < WIDTH; i++)`
`  array[j * WIDTH + i]++;`
	0.35 secondes

Topologie mémoire : pourquoi

Code source	Temps d'exécution

`for (i = 0; i < WIDTH; i++)`
`  for (j = 0; j < HEIGHT; j++)`
`  array[j * WIDTH + i]++;`
	16.64 secondes

`for (j = 0; j < HEIGHT; j++)`
`  for (i = 0; i < WIDTH; i++)`
`  array[j * WIDTH + i]++;`
	0.35 secondes

- Les deux codes sont fontionellement équivalents

Topologie mémoire : pourquoi

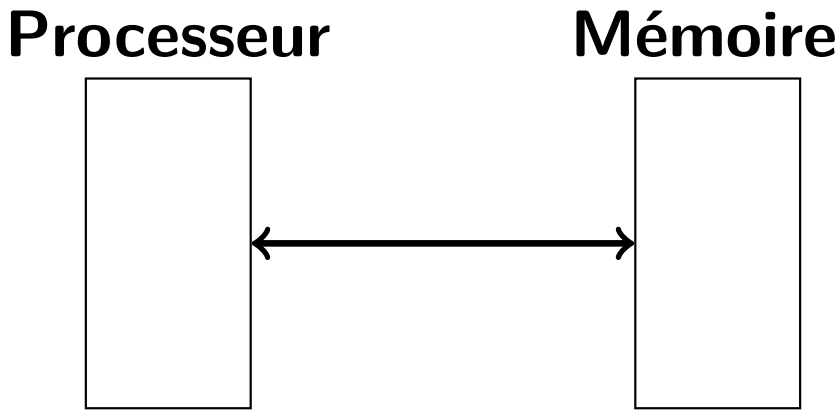
Code source	Temps d'exécution

`for (i = 0; i < WIDTH; i++)`
`  for (j = 0; j < HEIGHT; j++)`
`  array[j * WIDTH + i]++;`
	16.64 secondes

`for (j = 0; j < HEIGHT; j++)`
`  for (i = 0; i < WIDTH; i++)`
`  array[j * WIDTH + i]++;`
	0.35 secondes

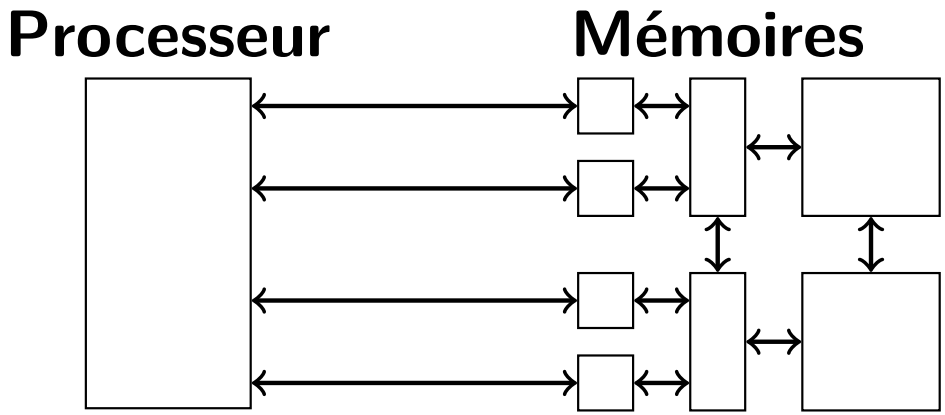
- Les deux codes sont fontionellement équivalents
- La seconde version exploite correctement la topologie mémoire

Topologie mémoire : quoi



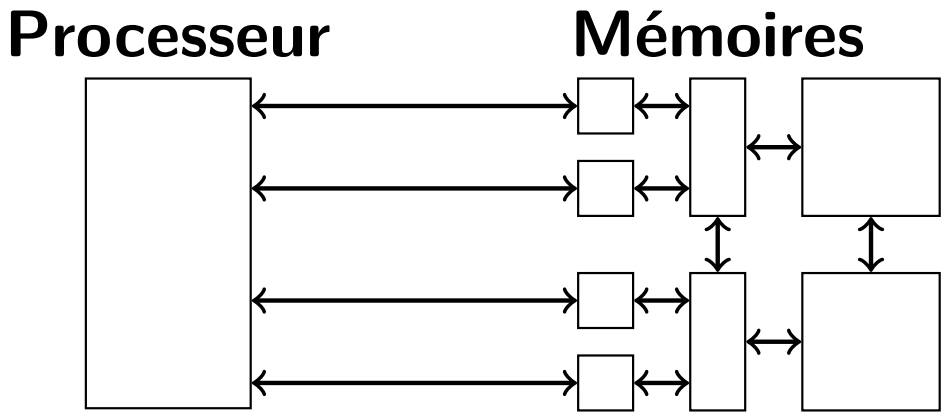
- Représentation simple d'un ordinateur : le processeur accède à la mémoire

Topologie mémoire : quoi



- Représentation simple d'un ordinateur : le processeur accède à la mémoire
- Il n'y a pas une mémoire unique mais plusieurs mémoires qui communiquent entre elles et avec le processeur

Topologie mémoire : quoi



- Représentation simple d'un ordinateur : le processeur accède à la mémoire
- Il n'y a pas une mémoire unique mais plusieurs mémoires qui communiquent entre elles et avec le processeur
- La topologie mémoire est la manière dont les différentes mémoires d'un système sont reliées et communiquent entre elles

Plan du cours

1. Multicœur et cohérence de caches
   - Caches multicœurs et cohérence séquentielle
   - Protocole MESI

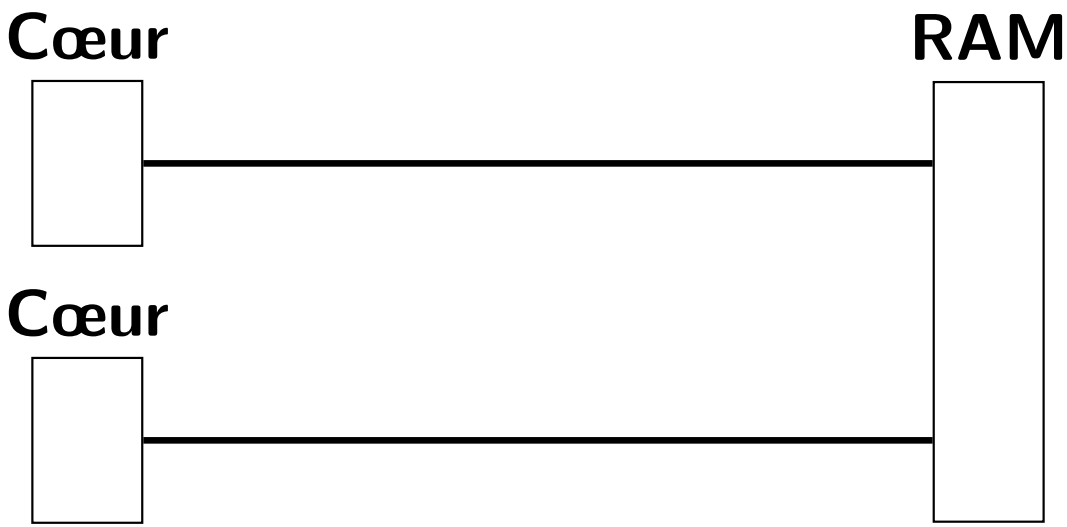
2. Architectures *Non Uniform Memory Access*
   - Contention matérielle et architecture NUMA
   - Cohérence NUMA et *cache directory*
   - Stratégies d'allocation mémoire

Plan du cours

1. Multicœur et cohérence de caches
   - Caches multicœurs et cohérence séquentielle
   - Protocole MESI
2. Architectures *Non Uniform Memory Access*
   - Contention matérielle et architecture NUMA
   - Cohérence NUMA et *cache directory*
   - Stratégies d'allocation mémoire

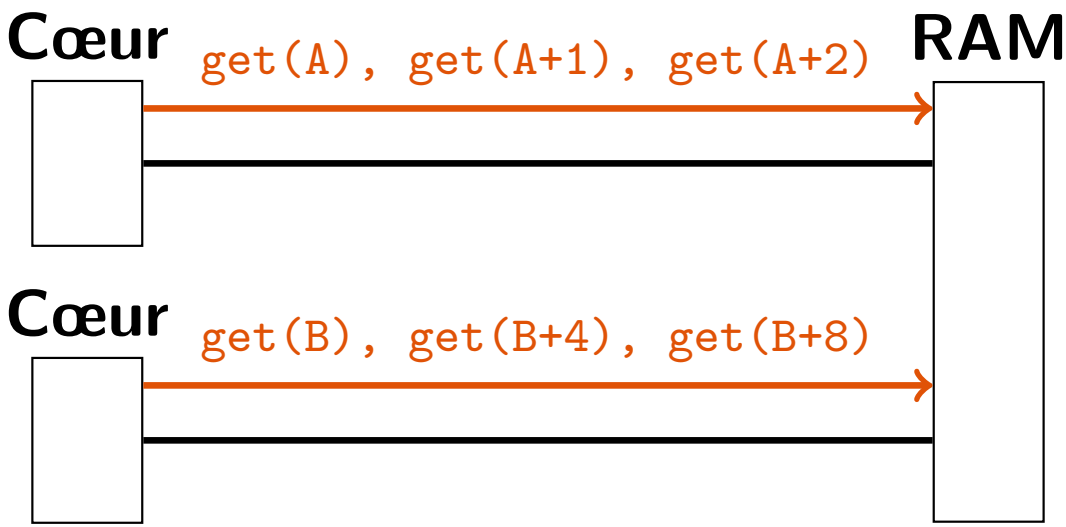
Architectures multicœurs et mémoire cache

- Les machines actuelles ont une architecture multicœur
- Chaque cœur accède à la mémoire indépendamment des autres cœurs



Architectures multicœurs et mémoire cache

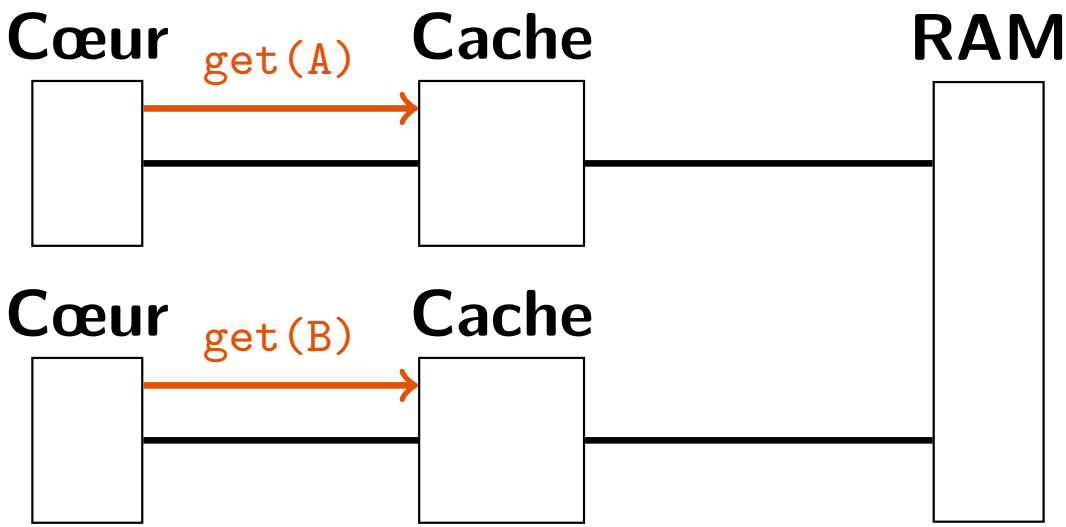
- Les machines actuelles ont une architecture multicœur
- Chaque cœur accède à la mémoire indépendamment des autres cœurs



- Chaque cœur suit son propre motif d'accès $\rightarrow$ peu de localité entre les cœurs

Architectures multicœurs et mémoire cache

- Les machines actuelles ont une architecture multicœur
- Chaque cœur accède à la mémoire indépendamment des autres cœurs



- Chaque cœur suit son propre motif d'accès $\rightarrow$ peu de localité entre les cœurs
- Chaque cœur dispose de son propre cache $\rightarrow$ bonne localité par cache

Cohérence séquentielle

- Les fonctions `on_core_0()` et `on_core_1()` s'exécutent en parallèle

```
A = 0;
B = 0;
```

```
void on_core_0(void) {
    A = 1;
    B = 1;
}
```

```
void on_core_1(void) {
    b = B;
    a = A;
}
```

- États possibles de `(a,b)` :

Cohérence séquentielle

- Les fonctions `on_core_0()` et `on_core_1()` s'exécutent en parallèle

```
A = 0;
B = 0;
```

```
void on_core_0(void) {
    A = 1;
    B = 1;
}
```

```
void on_core_1(void) {
    b = B;
    a = A;
}
```

- États possibles de `(a,b)` : `(0,0)`, `(1,0)`, `(1,1)`

Cohérence séquentielle

- Les fonctions `on_core_0()` et `on_core_1()` s'exécutent en parallèle

```
A = 0;
B = 0;
```

```
void on_core_0(void) {
    A = 1;
    B = 1;
}
```

```
void on_core_1(void) {
    b = B;
    a = A;
}
```

- États possibles de `(a,b)` : `(0,0)`, `(1,0)`, `(1,1)`
- L'état `(0,1)` est impossible avec un modèle de cohérence séquentielle

Cohérence séquentielle : définition

- Les fonctions `on_core_0()` et `on_core_1()` s'exécutent en parallèle

```
A = 0;
B = 0;
```

```
void on_core_0(void) {
    A = 1;
    B = 1;
}
```

```
void on_core_1(void) {
    b = B;
    a = A;
}
```

- États possibles de `(a,b)` : `(0,0)`, `(1,0)`, `(1,1)`
- L'état `(0,1)` est impossible avec un modèle de cohérence séquentielle

Une exécution est **cohérente séquentiellement** si
il existe un ordre total des opérations qui préserve l'ordre des instructions de chaque cœur et qui conduit au même résultat

Cohérence séquentielle : définition

- Les fonctions `on_core_0()` et `on_core_1()` s'exécutent en parallèle

```
A = 0;
B = 0;
```

```
void on_core_0(void) {
    A = 1;
    B = 1;
}
```

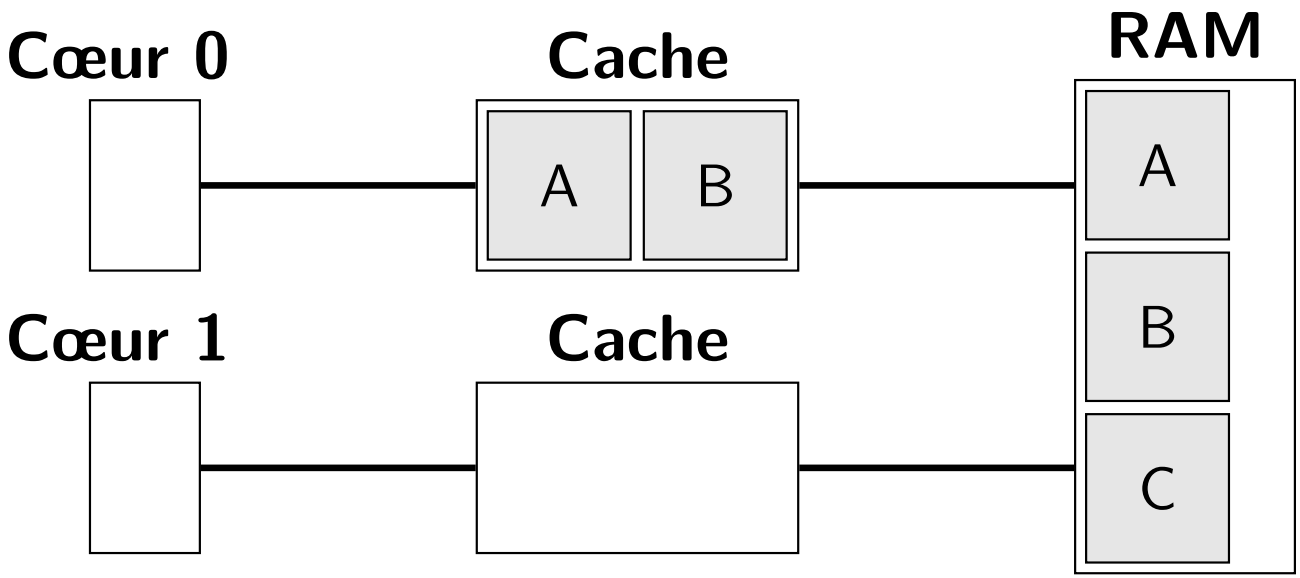
```
void on_core_1(void) {
    b = B;
    a = A;
}
```

- États possibles de `(a,b)` : `(0,0)`, `(1,0)`, `(1,1)`
- L'état `(0,1)` est impossible avec un modèle de cohérence séquentielle

> Une exécution est **cohérente séquentiellement** si il existe un ordre total des opérations qui préserve l'ordre des instructions de chaque cœur et qui conduit au même résultat

- La cohérence séquentielle paraît naturelle pour le programmeur

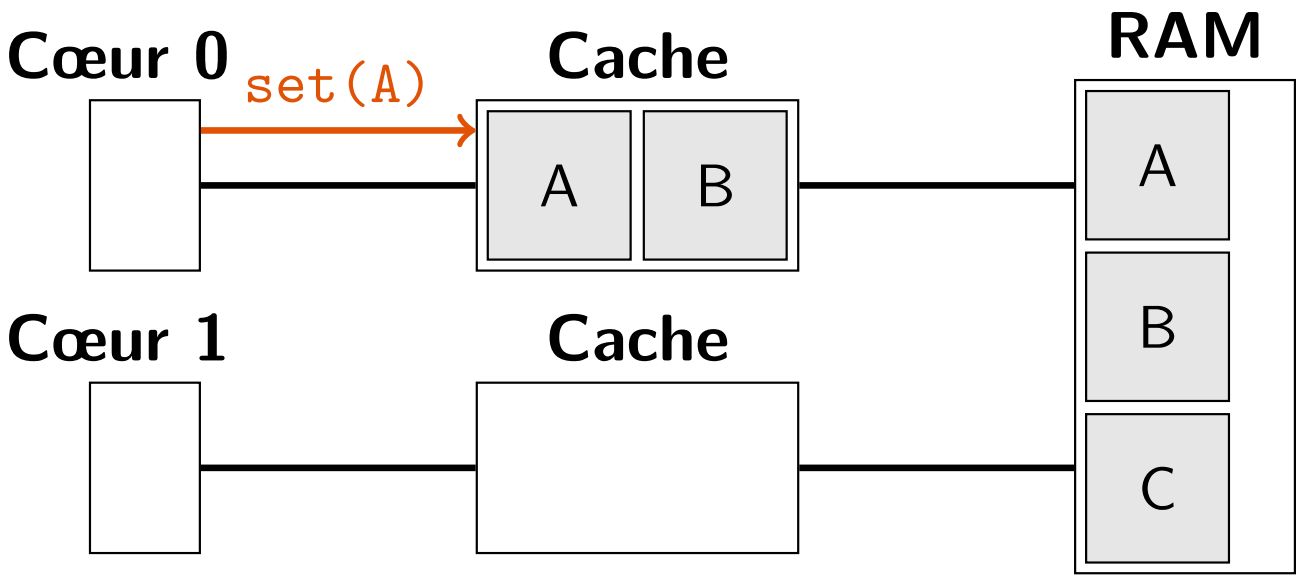
Cohérence séquentielle et cache multicœur




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

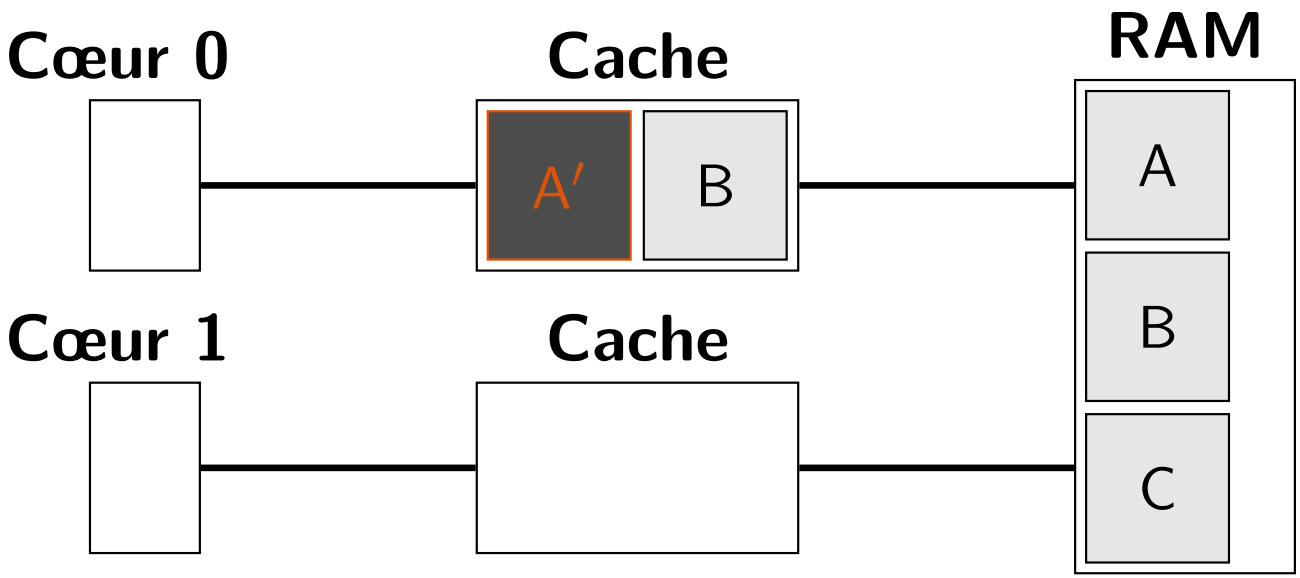
Cohérence séquentielle et cache multicœur




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

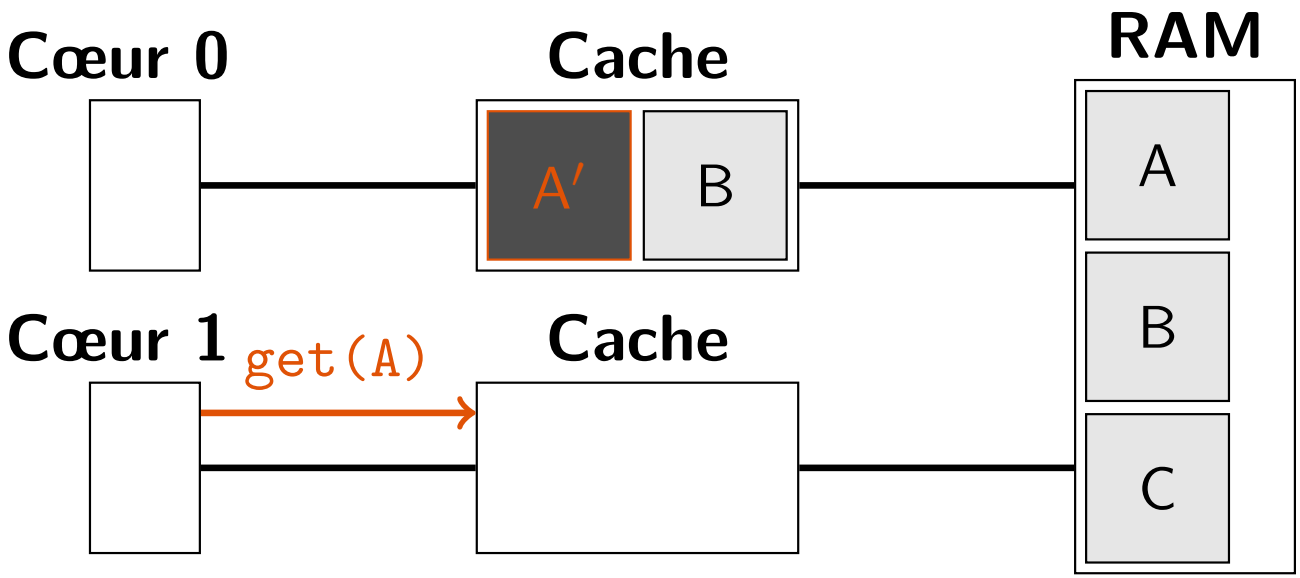
Cohérence séquentielle et cache multicœur




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

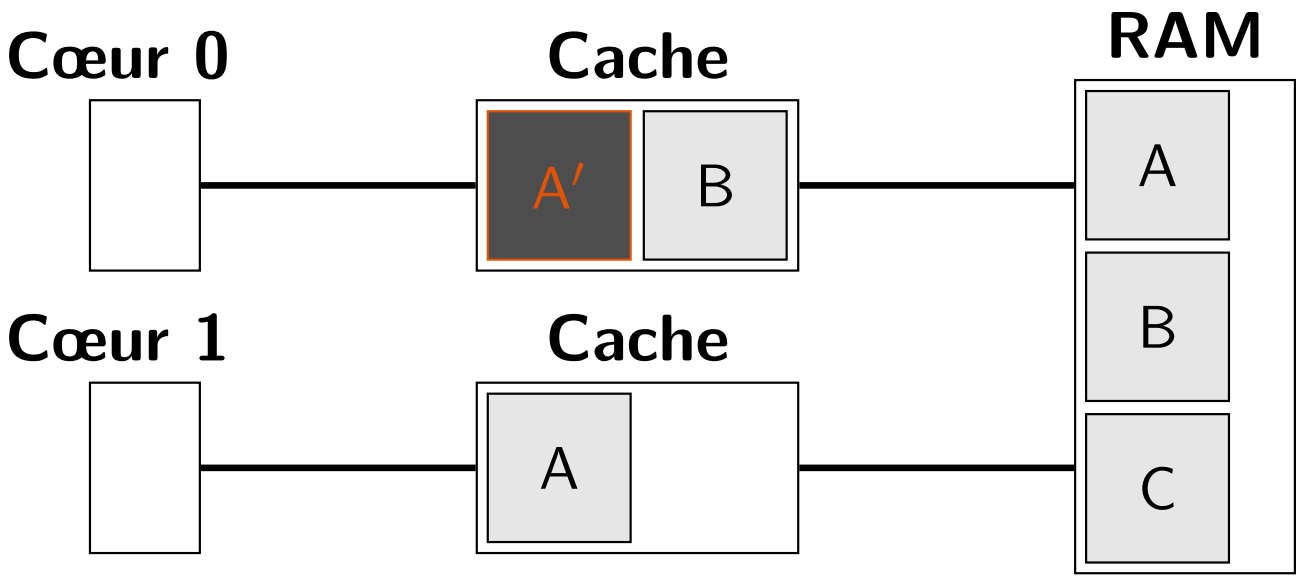



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée

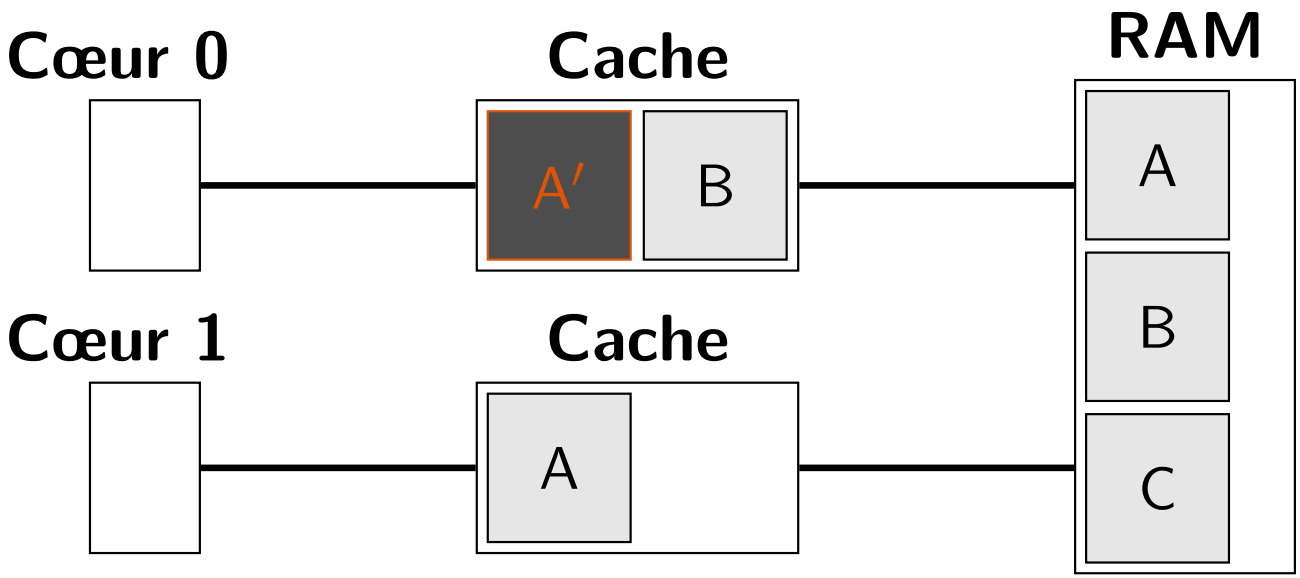



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici

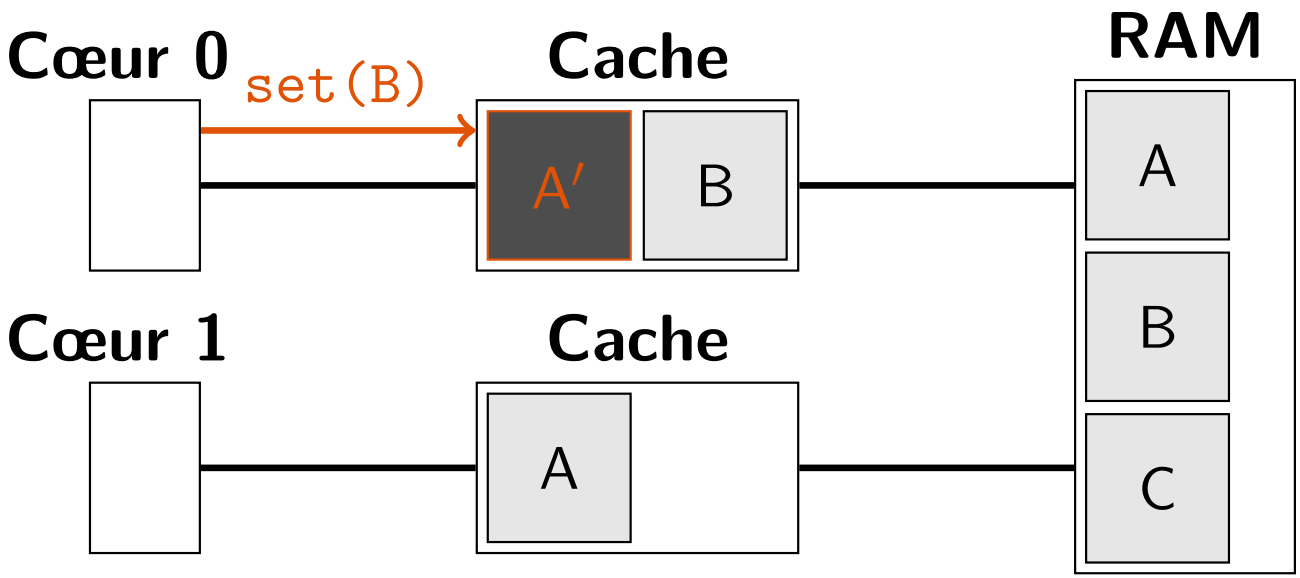



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici

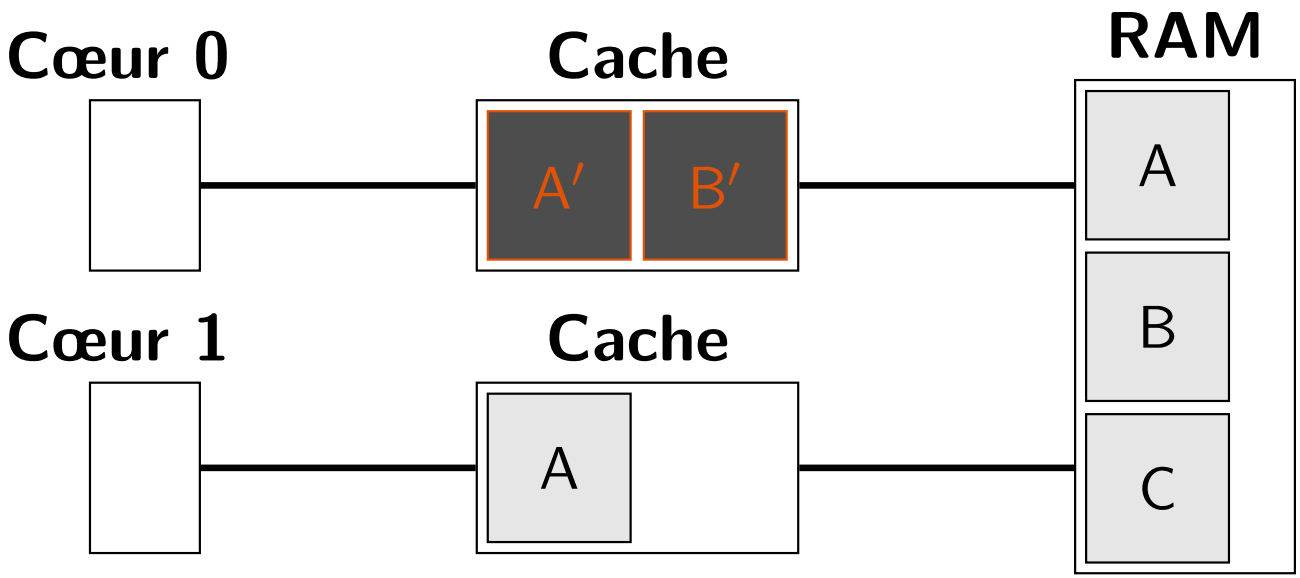



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée $\rightarrow$ pas de problème de cohérence ici

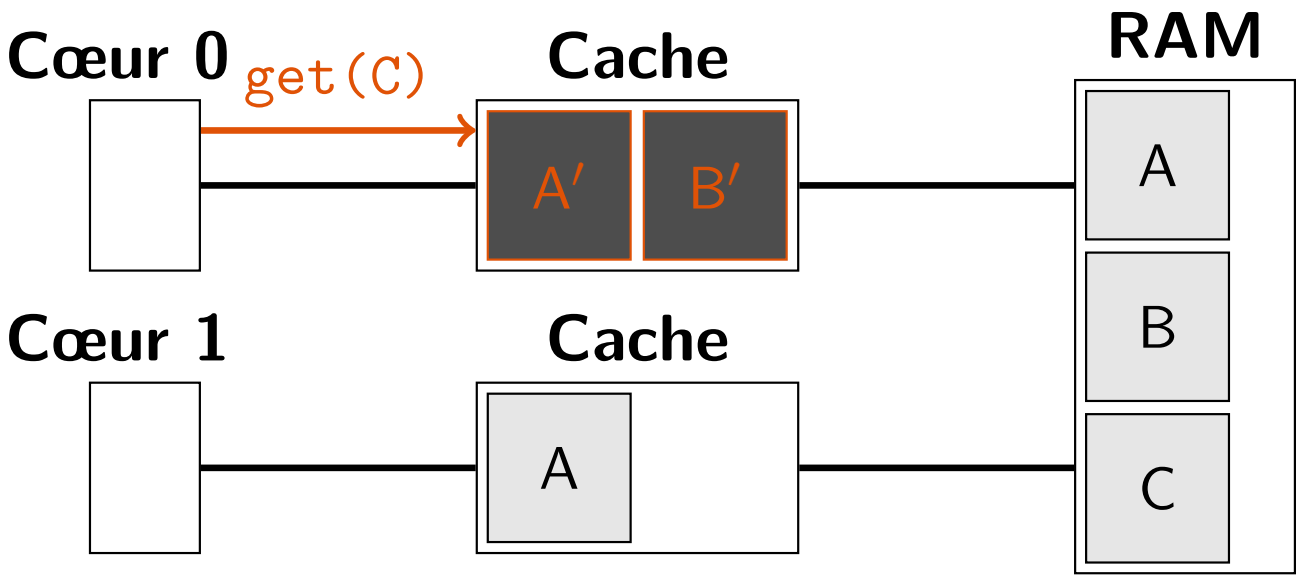



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici

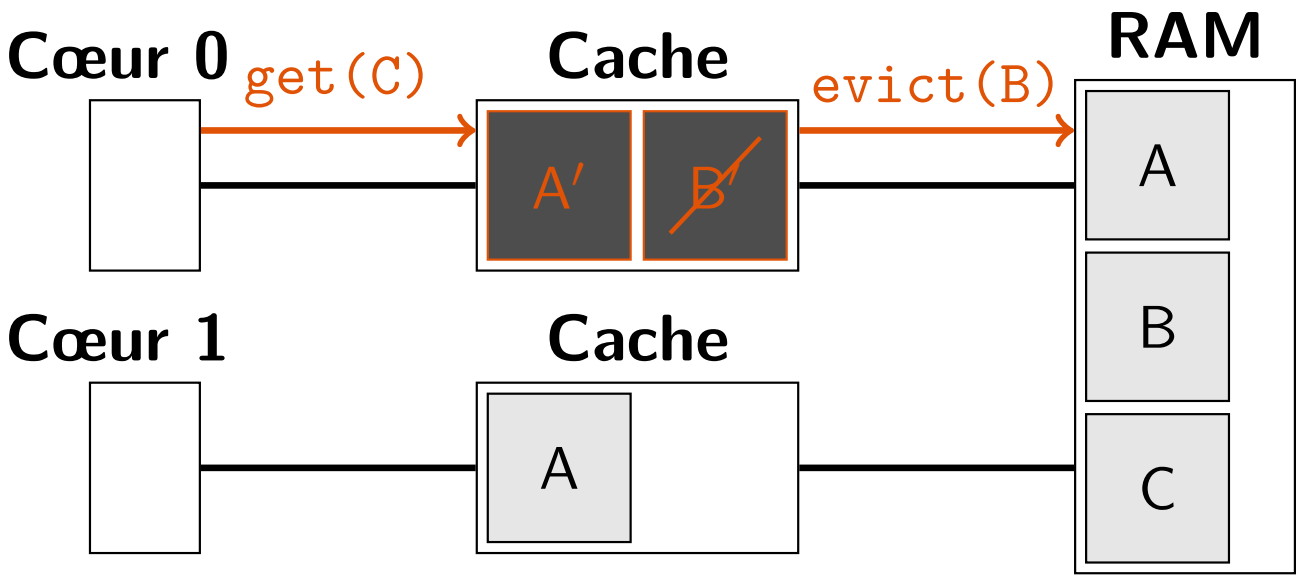



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici

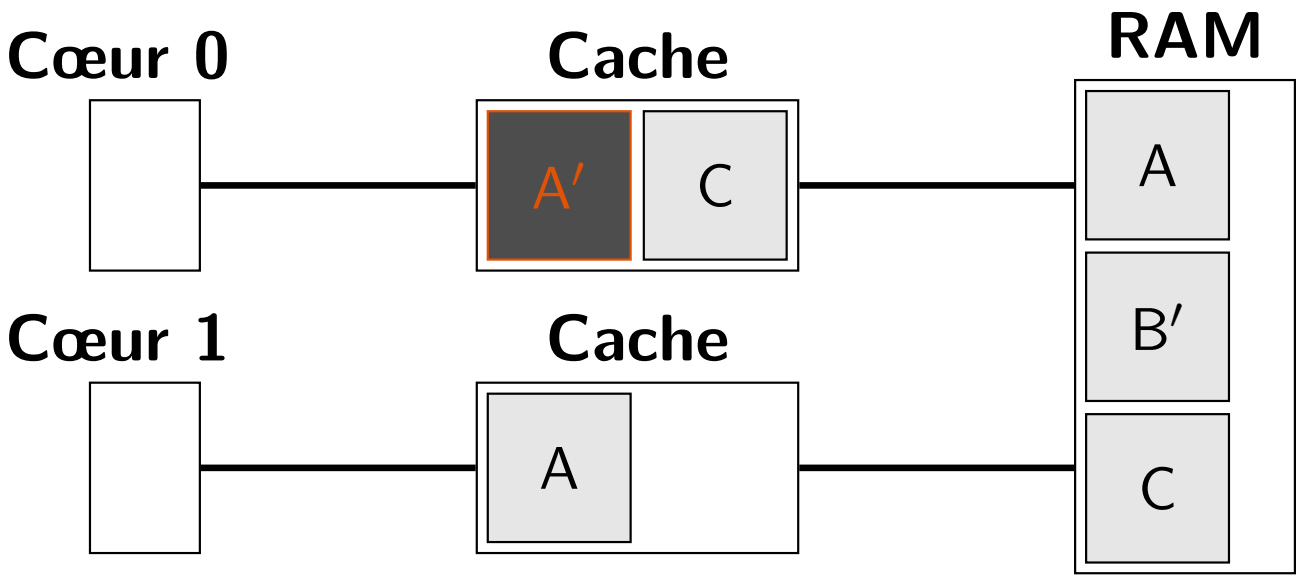



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici

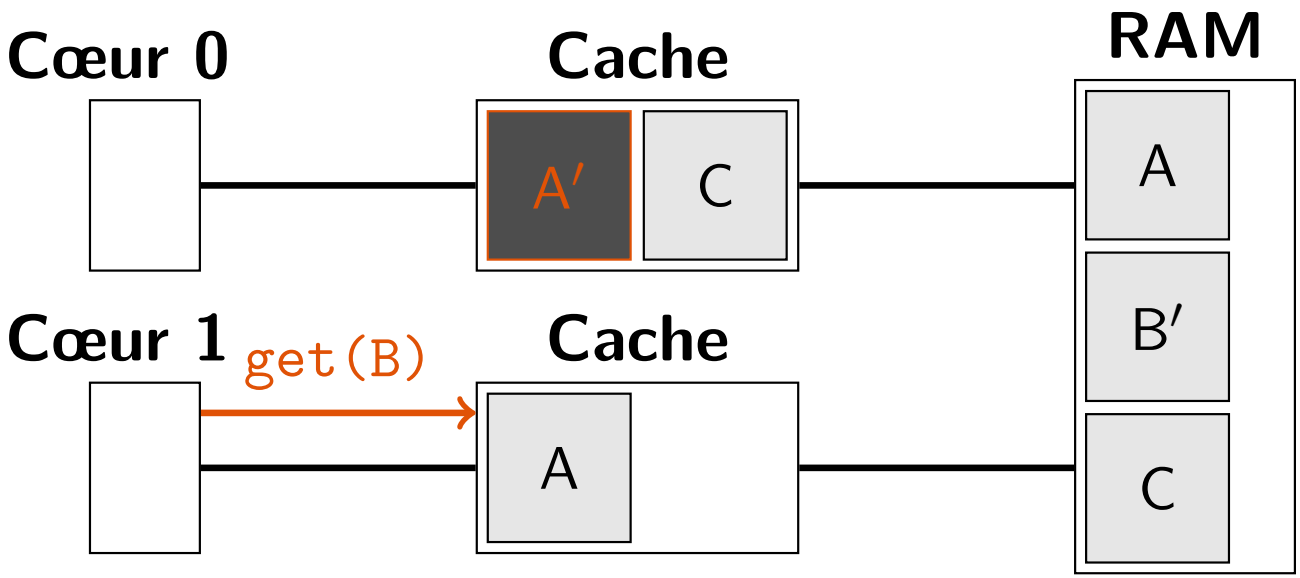



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici

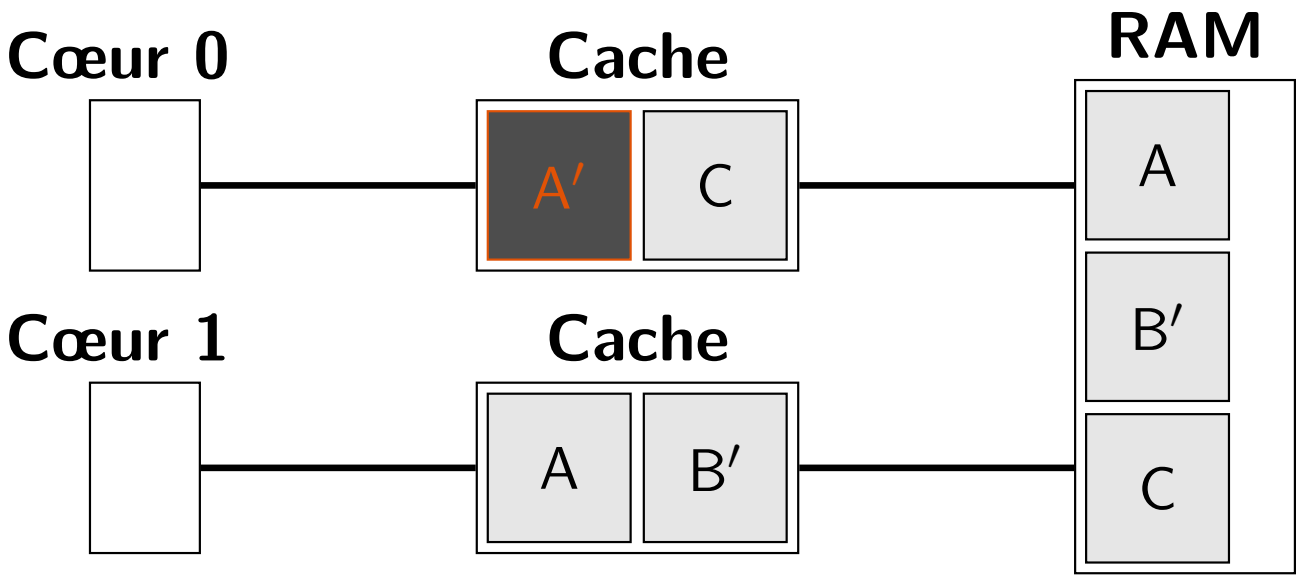



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici

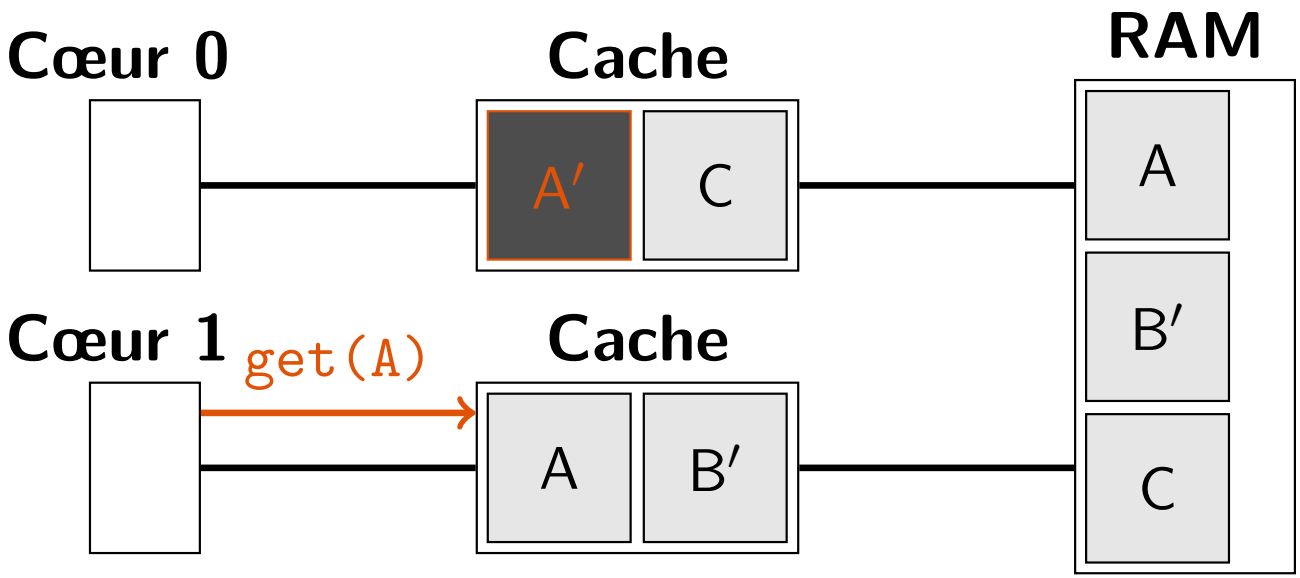



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici

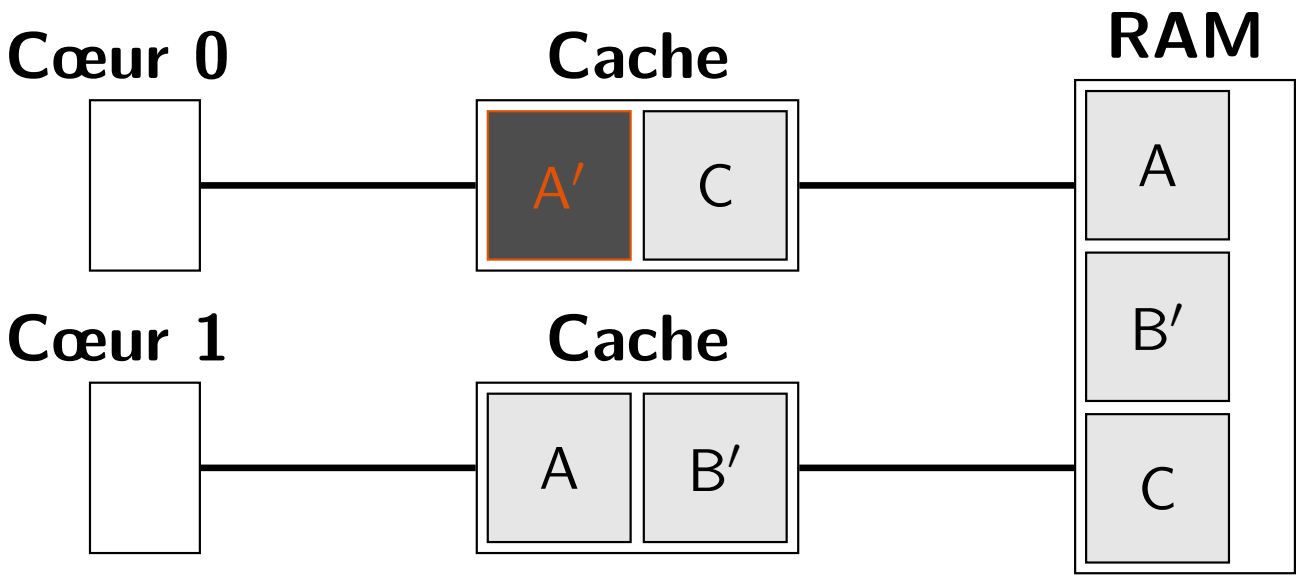



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée → pas de problème de cohérence ici
- Le cœur 1 lit `(a,b)` = `(0,1)`

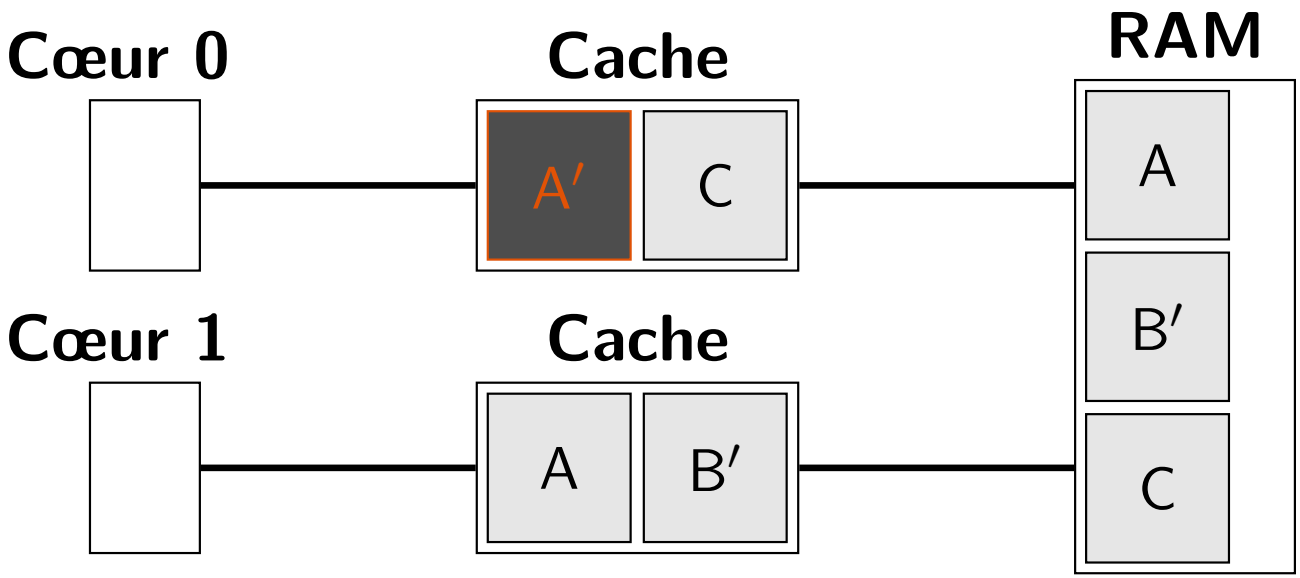



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Cohérence séquentielle et cache multicœur

- Le cœur 1 lit une vieille donnée $\rightarrow$ pas de problème de cohérence ici
- Le cœur 1 lit `(a,b)` $=$ `(0,1)` $\rightarrow$ incohérence séquentielle




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Caches multicœurs et cohérence séquentielle : résumé

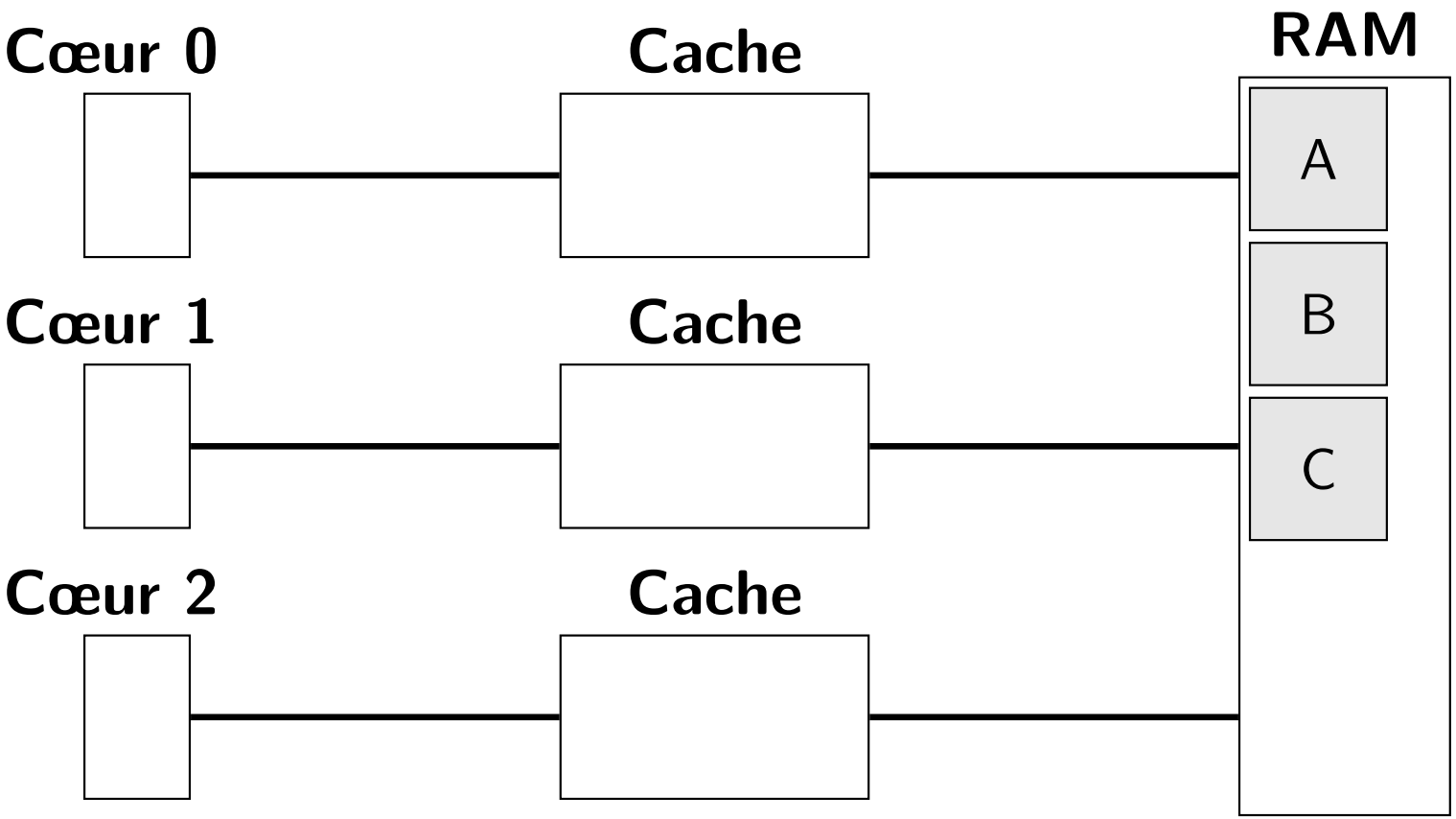
- Les architectures actuelles cherchent à guarantir la cohérence séquentielle des exécutions
  - Fonctionnement le plus naturel pour les programmeurs $\rightarrow$ incite les développeurs à utiliser l'architecture
  - Les instructions sur chaque cœur doivent "avoir l'air" de s'exécuter dans l'ordre
- Dans une architecture multicœur, chaque cœur a son propre cache
  - Les localités temporelle et spatiales sont spécifiques à chaque cœur
  - Pas de problème de multiplexage des ressources
- L'utilisation de mémoires cache totalement indépendantes brise la cohérence séquentielle

Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain

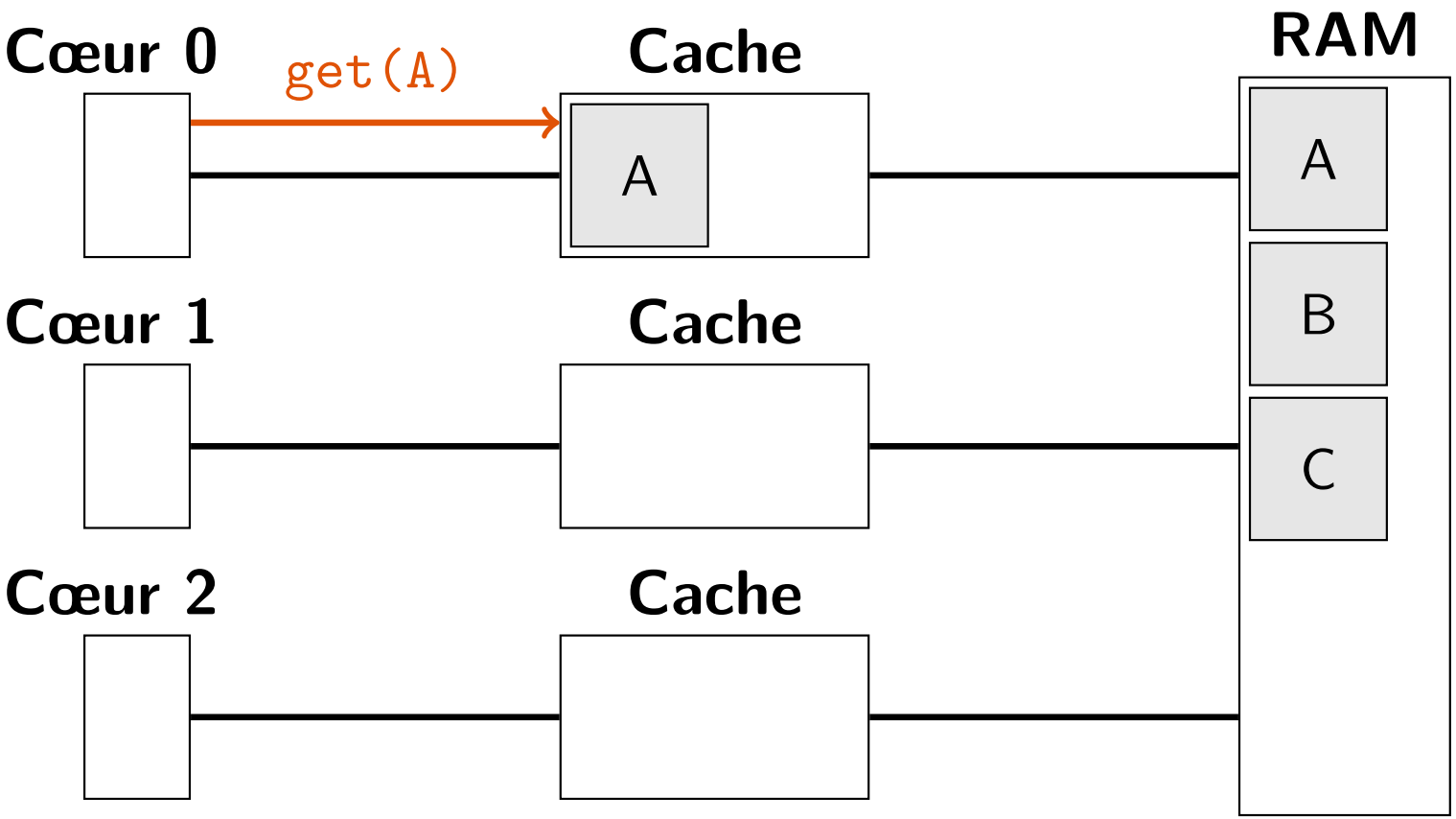
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : $\varnothing$



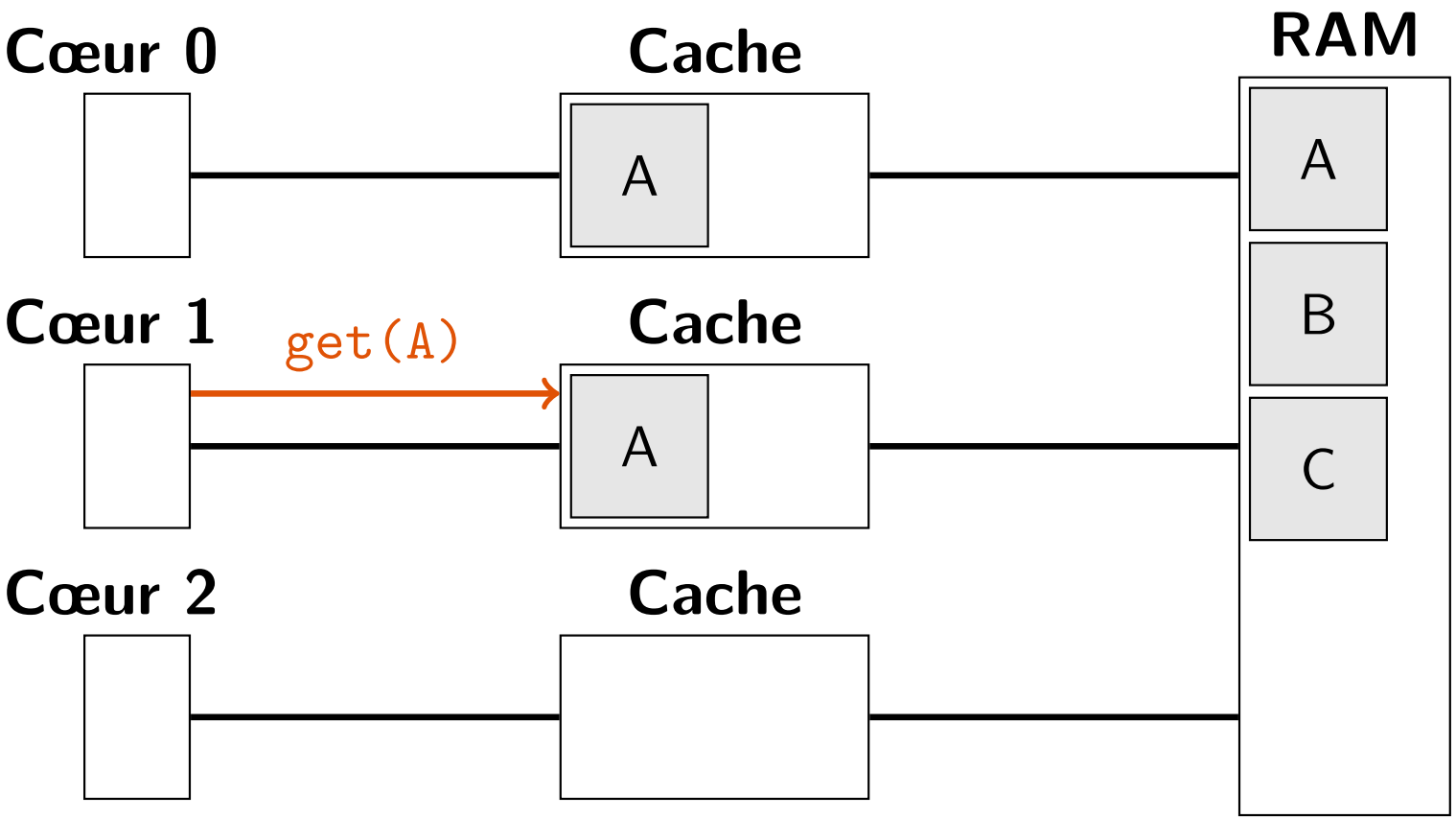
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : READ



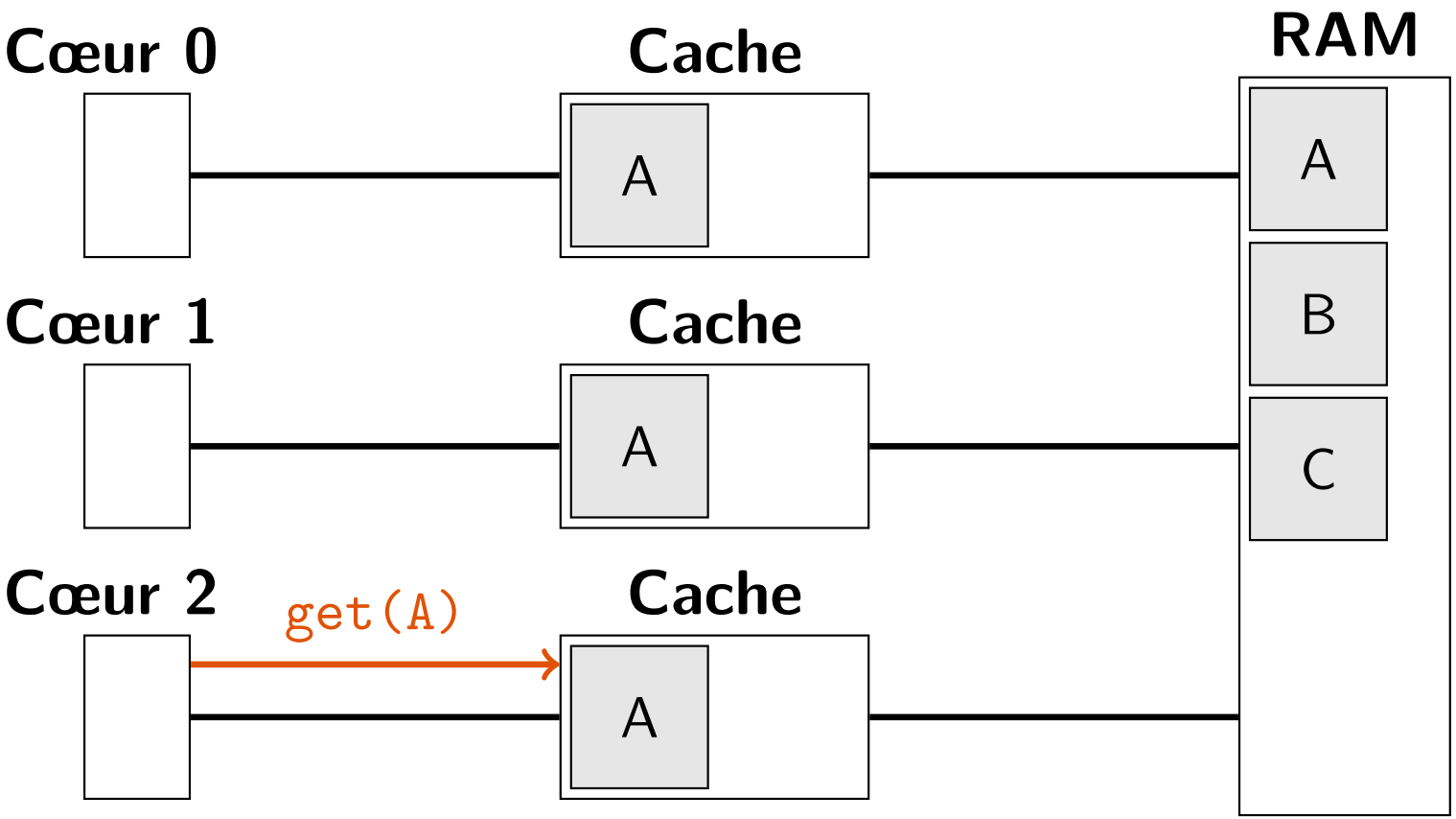
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : READ



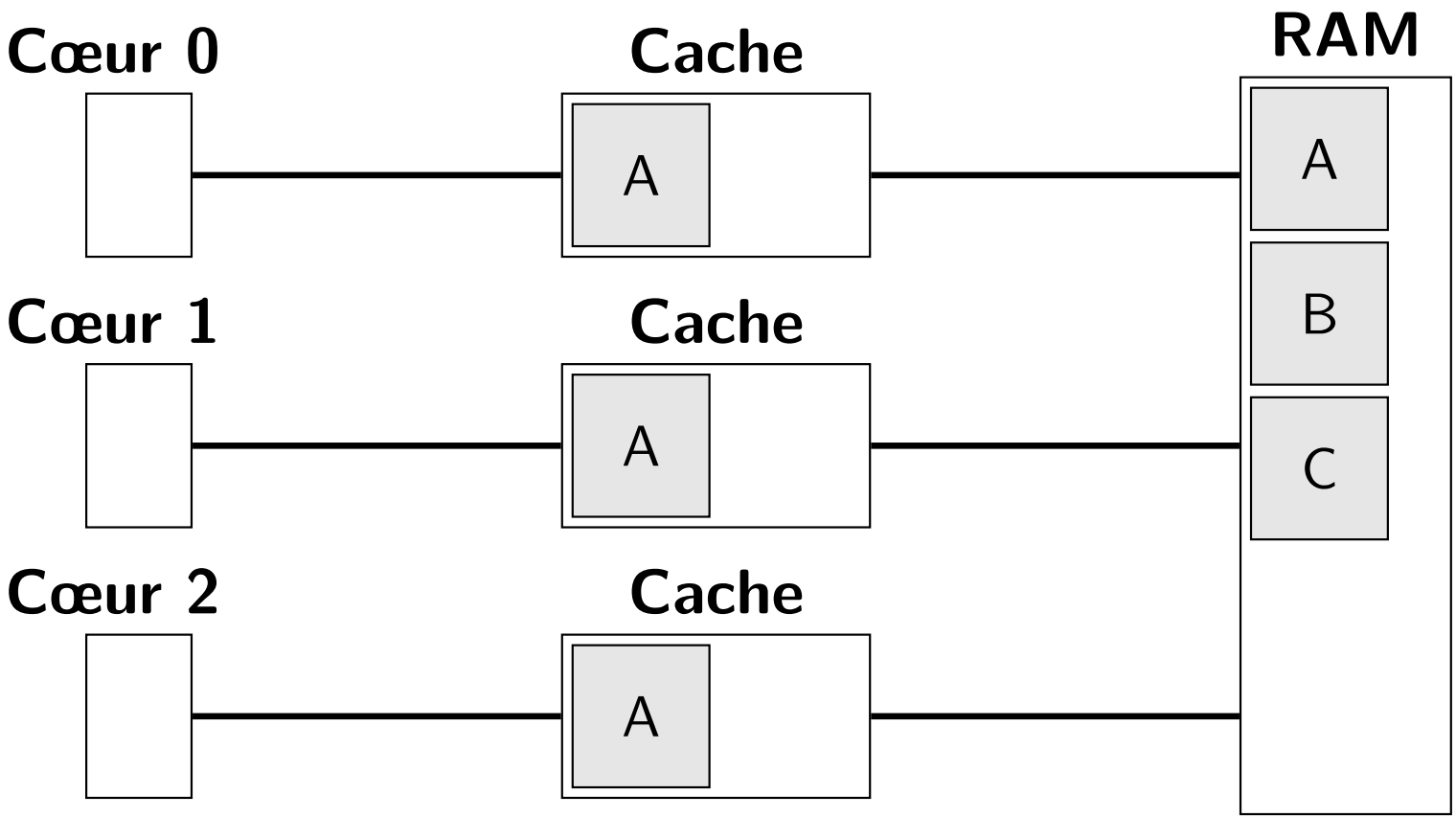
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : READ



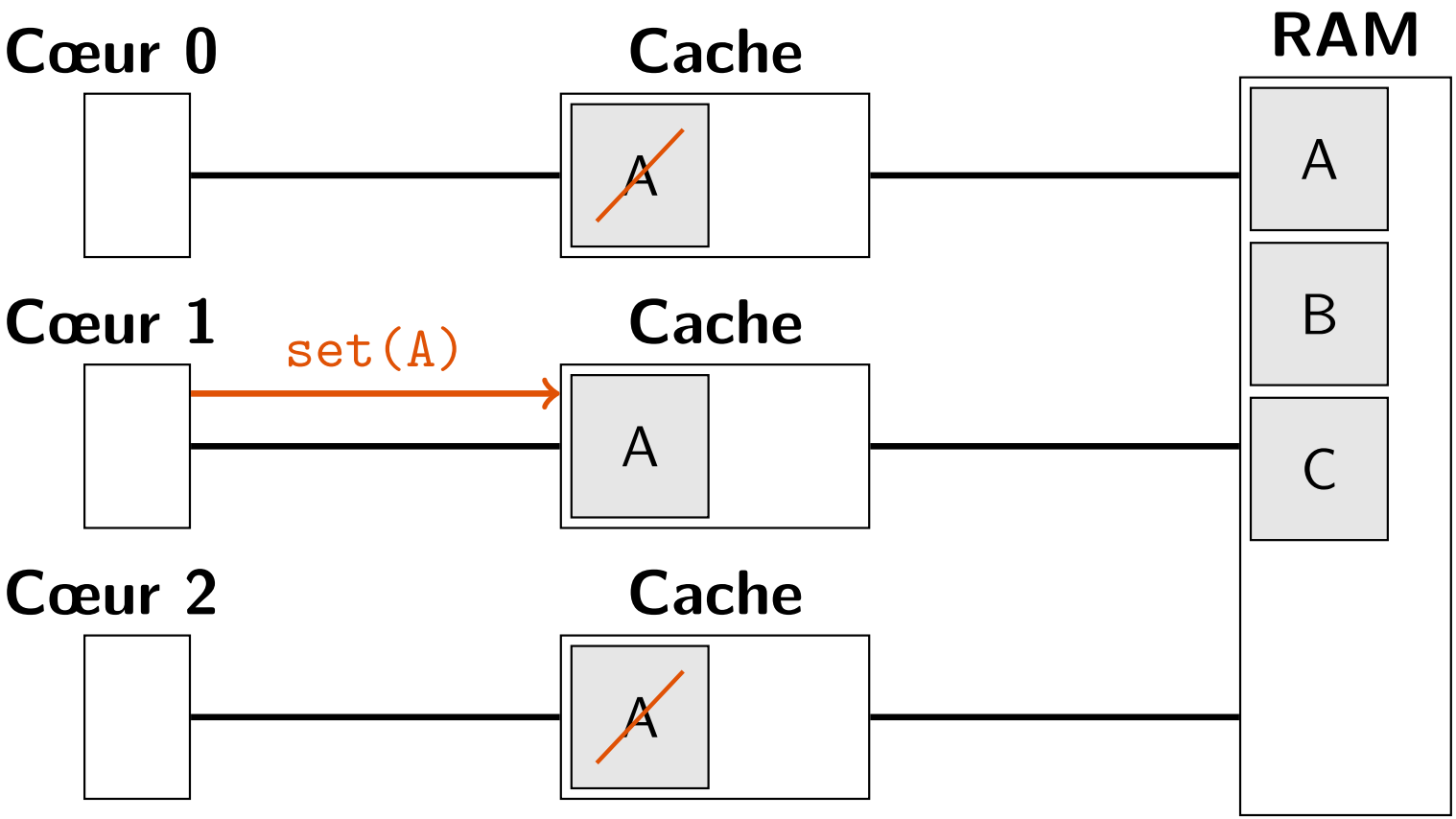
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : READ



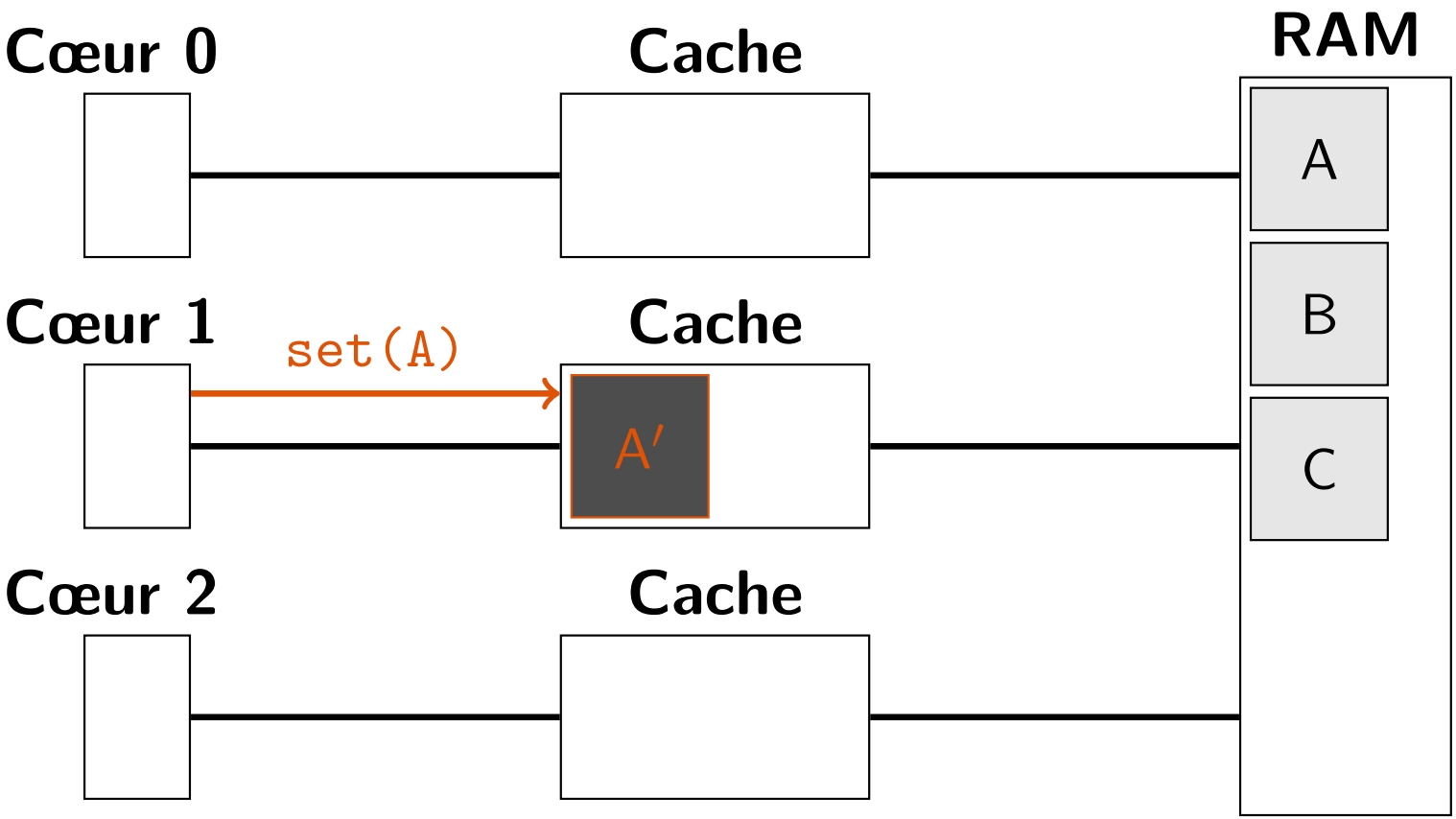
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : READ $\rightarrow$ WRITE



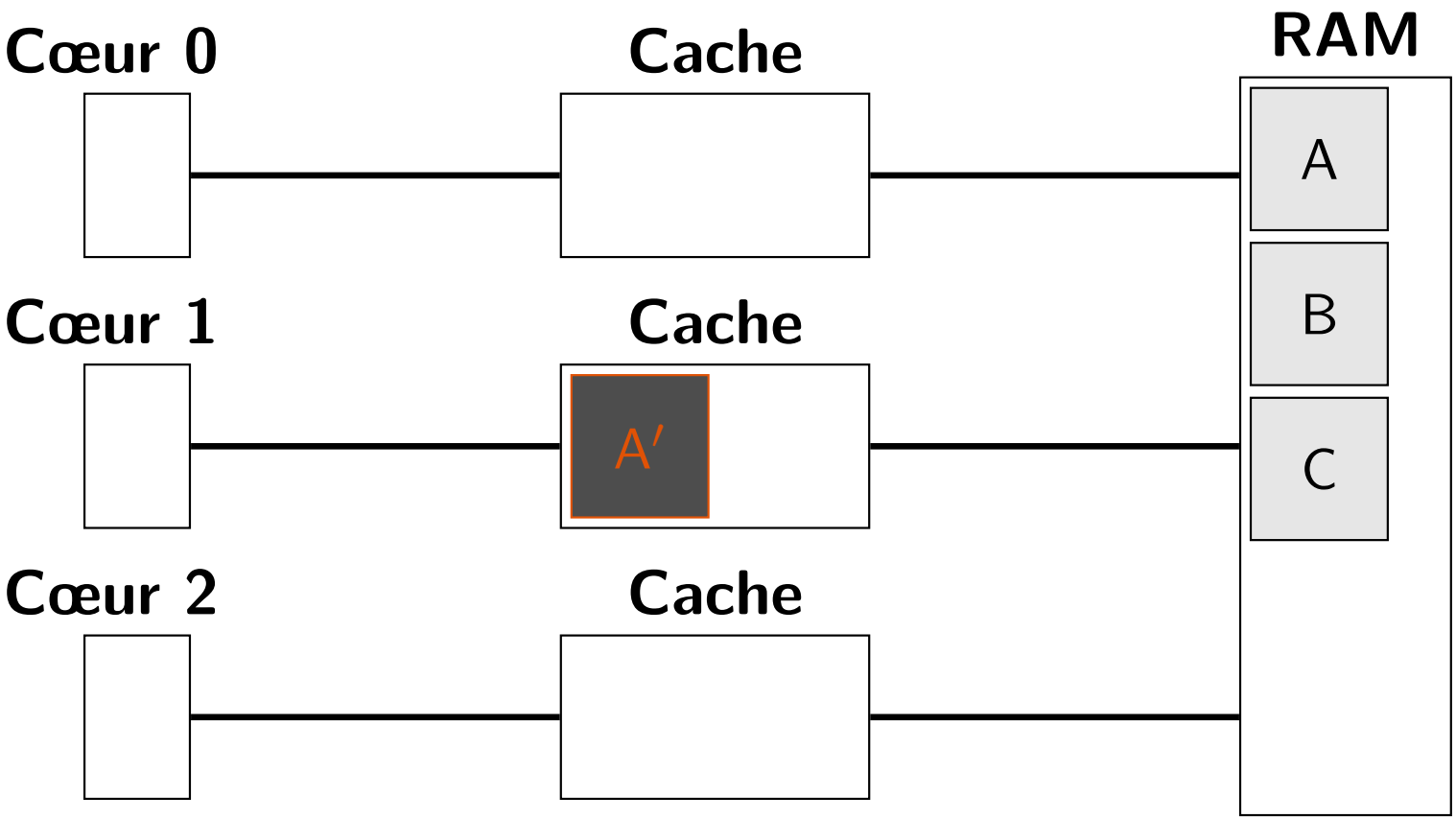
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : WRITE



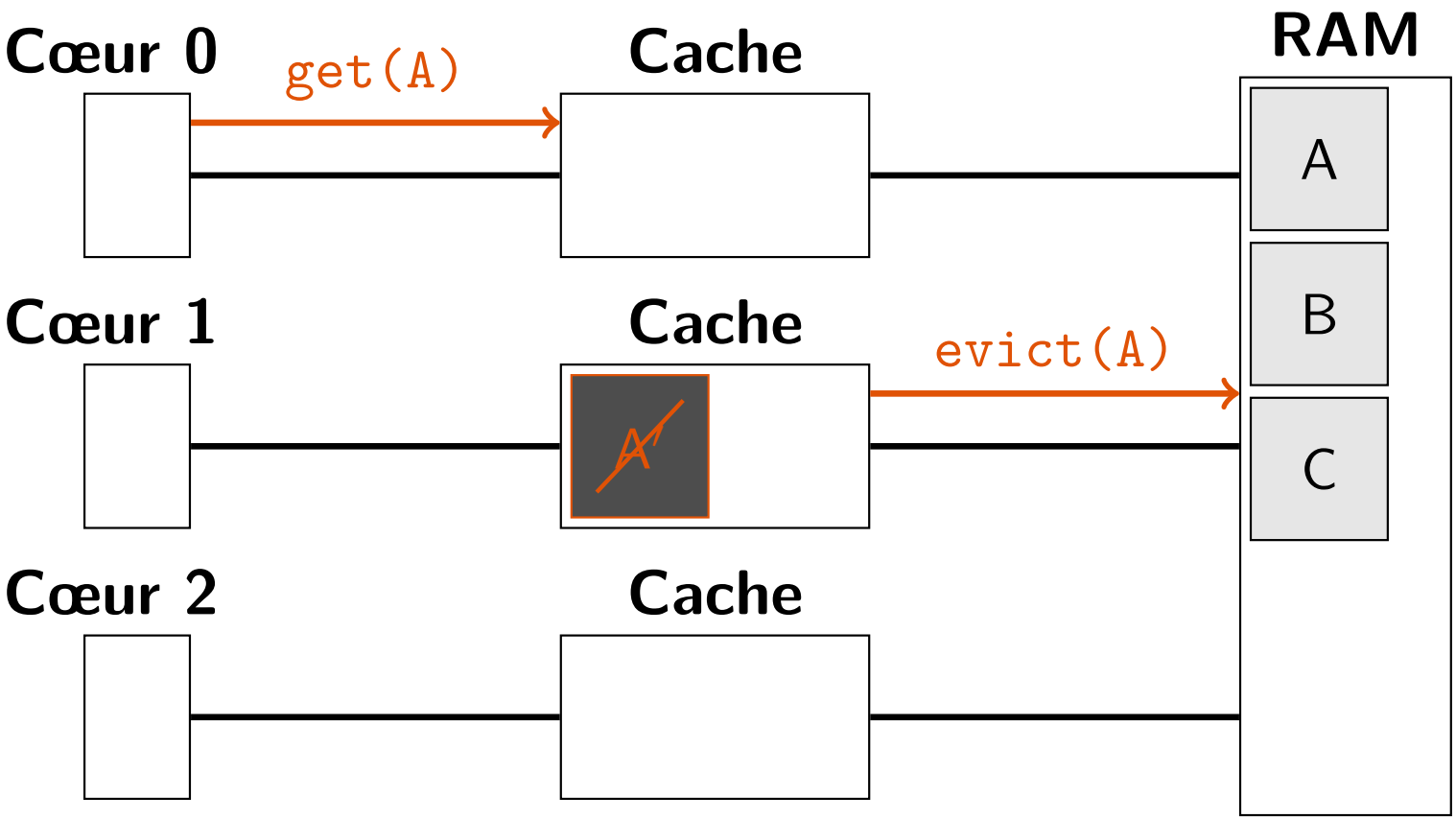
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : WRITE



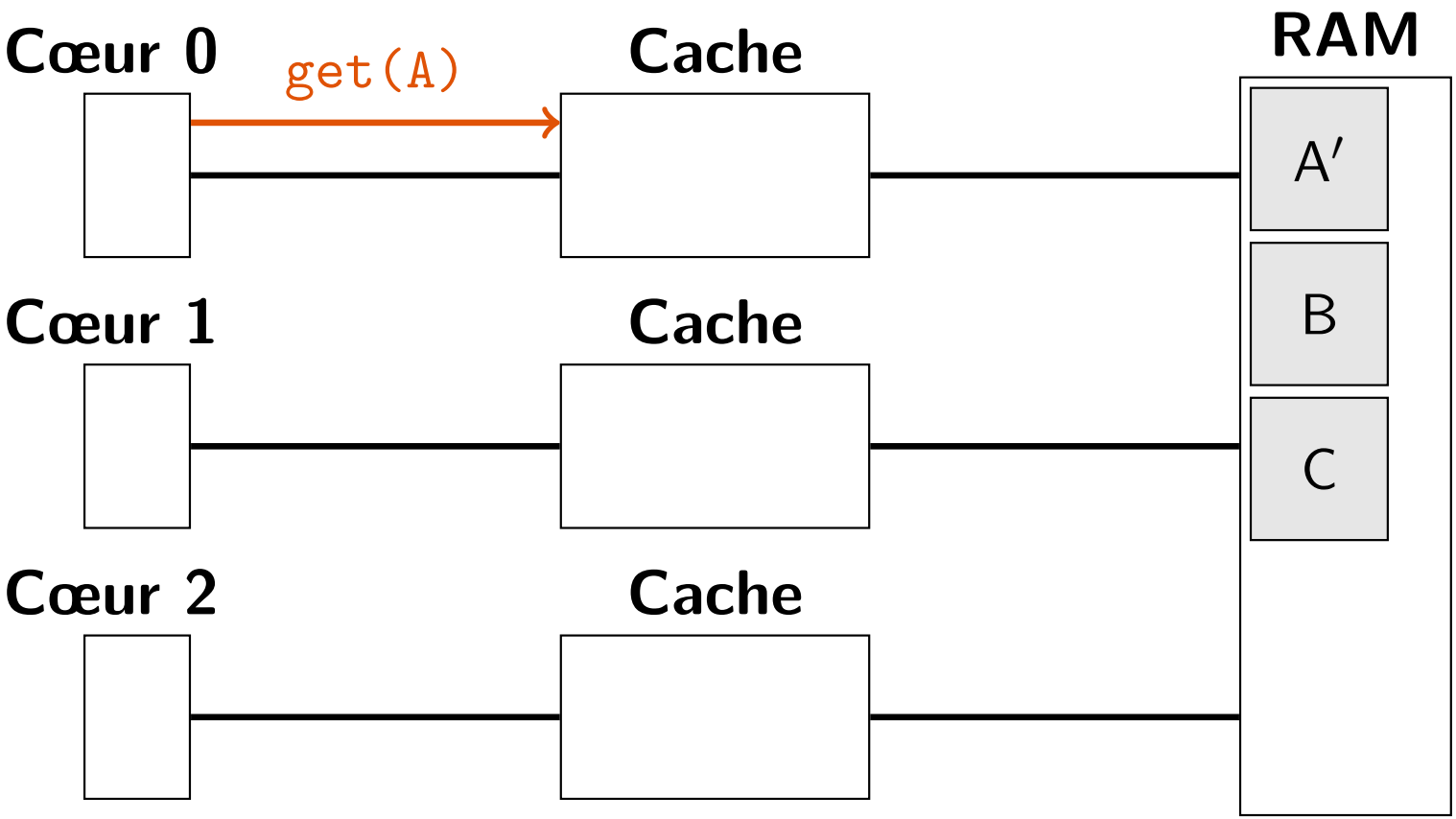
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : WRITE $\rightarrow$ READ



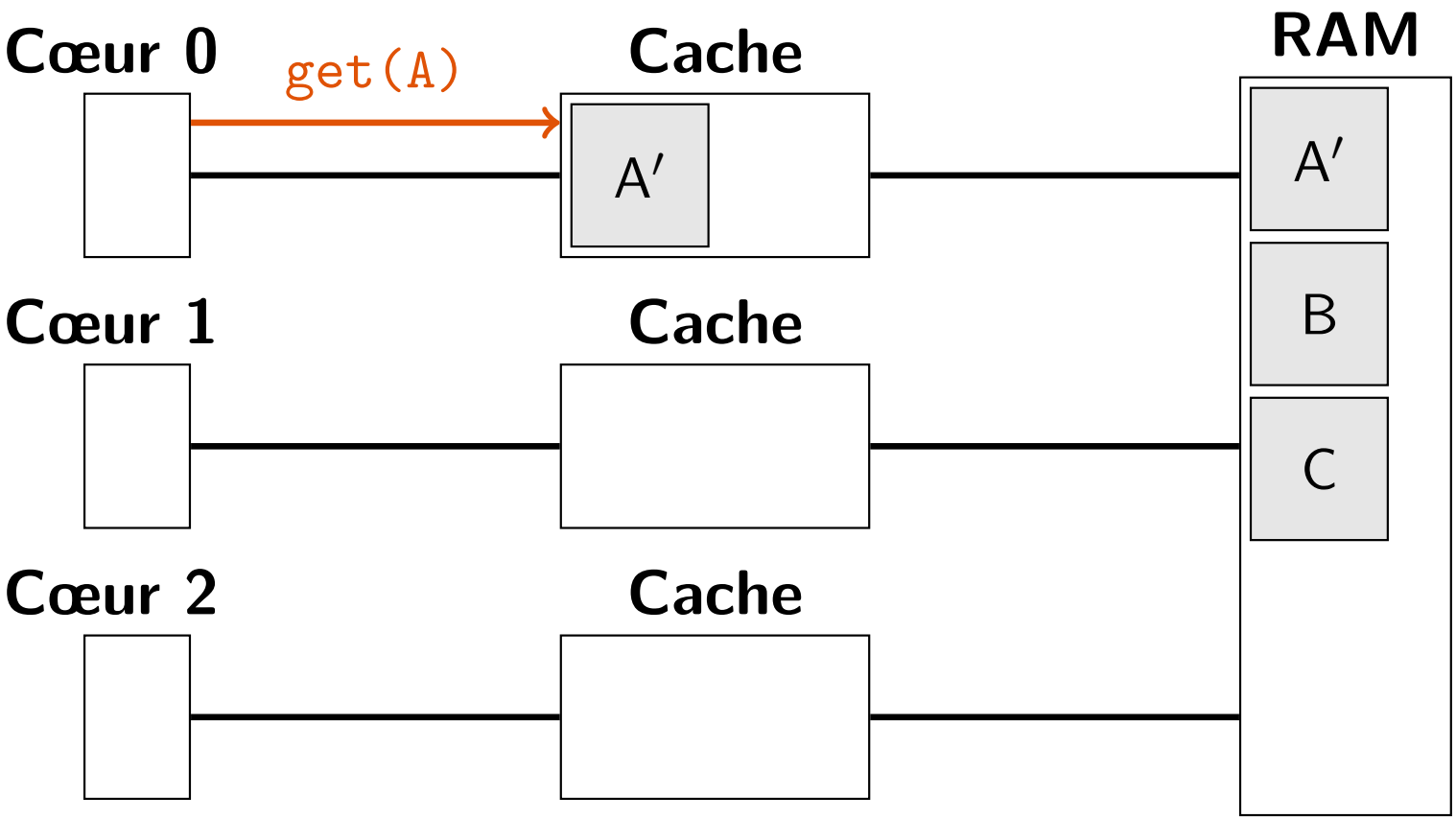
Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : WRITE $\rightarrow$ READ



Cohérence de cache et RW-lock distribué

- Solution possible pour assurer la cohérence séquentielle : toujours travailler sur la dernière version de chaque donnée
- Protéger chaque ligne de cache par un verrou lecteur/écrivain
- Verrou de `A` : READ

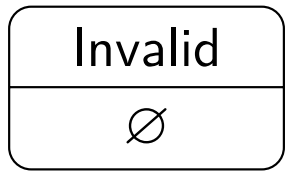


Le protocole MESI : une implémentation du RW-lock

- Chaque cache associe un état à chaque ligne

Le protocole MESI : une implémentation du RW-lock

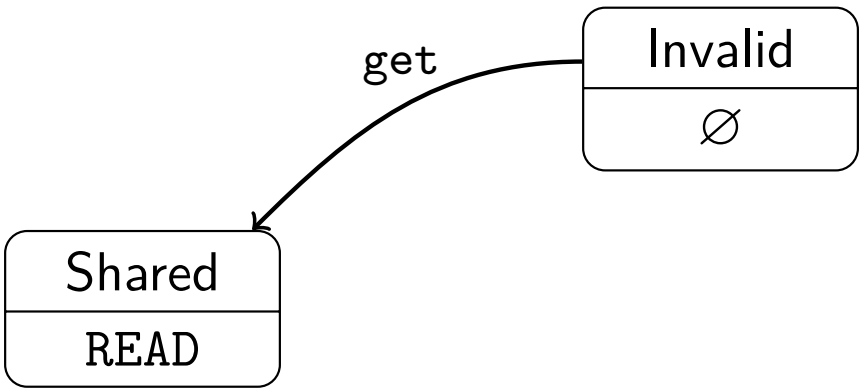
- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock



- Invalid : la ligne n'est pas stockée dans le cache

Le protocole MESI : une implémentation du RW-lock

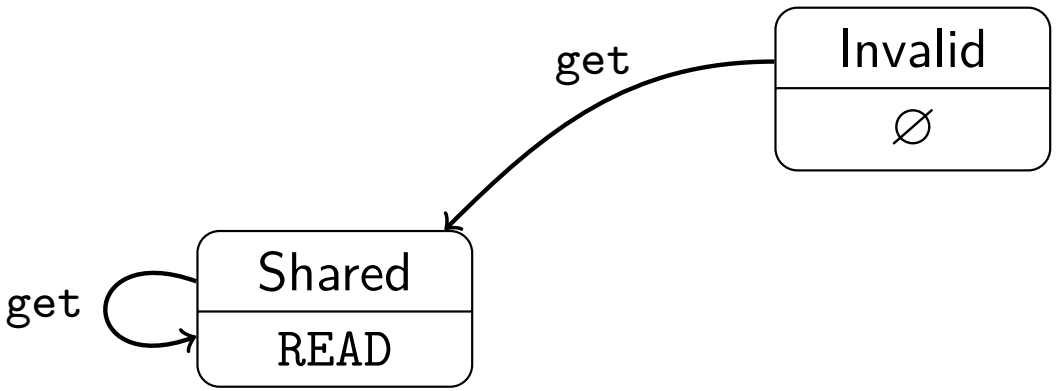
- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock



- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches

Le protocole MESI : une implémentation du RW-lock

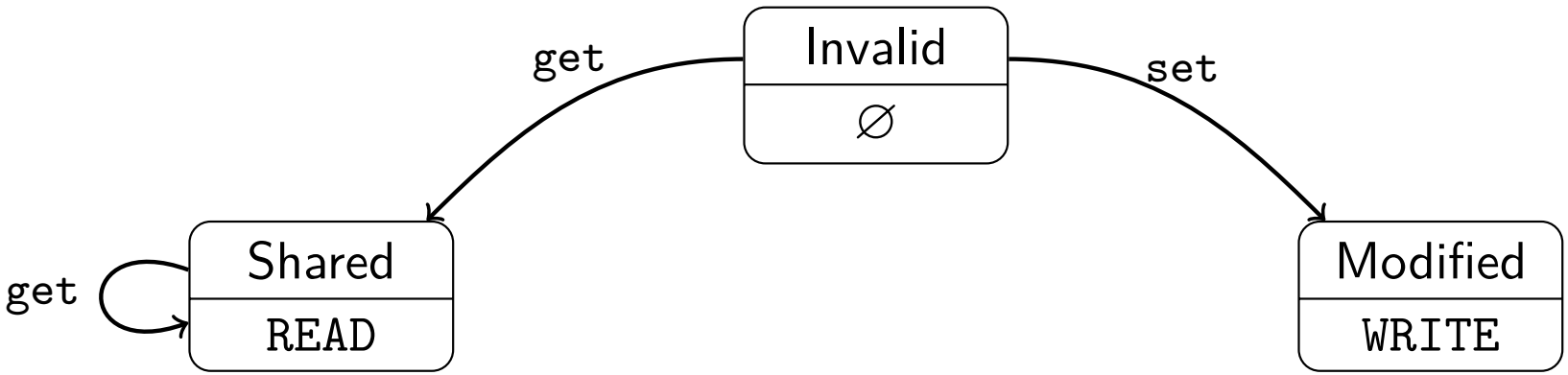
- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock



- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches

Le protocole MESI : une implémentation du RW-lock

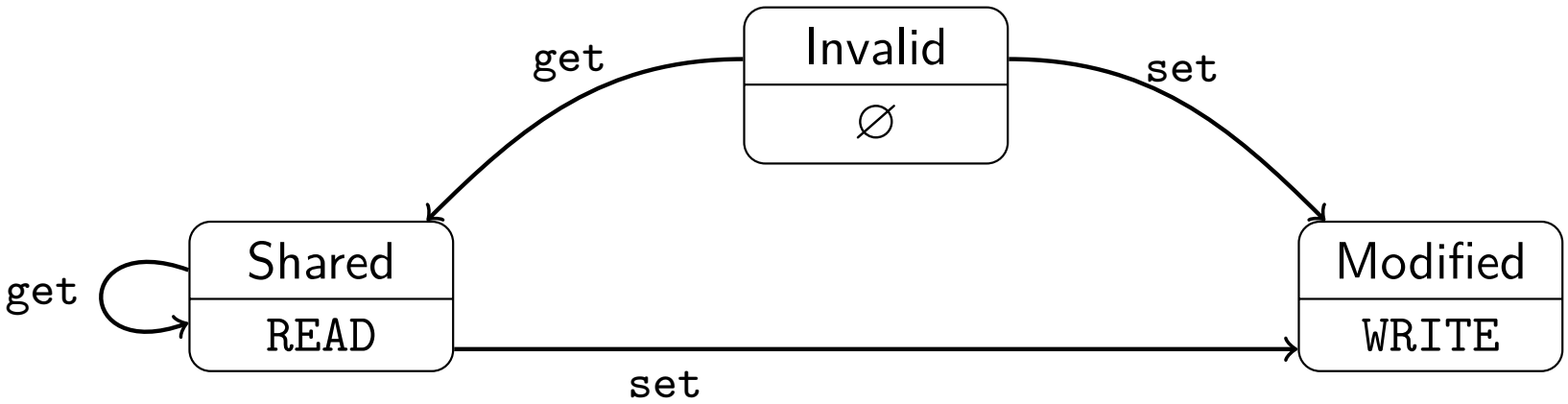
- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock



- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache

Le protocole MESI : une implémentation du RW-lock

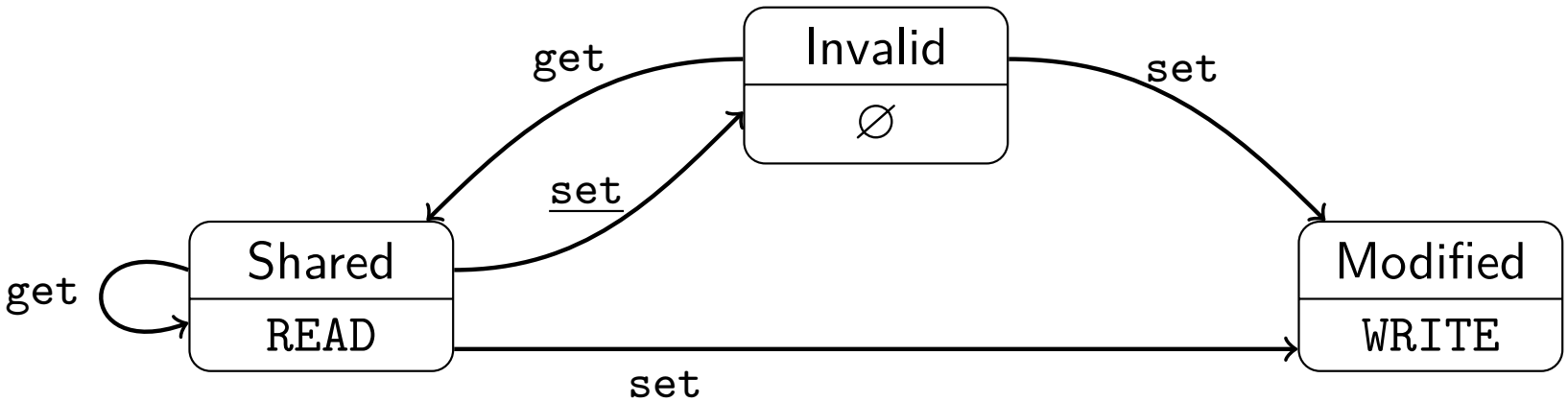
- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock



- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache

Le protocole MESI : une implémentation du RW-lock

- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour

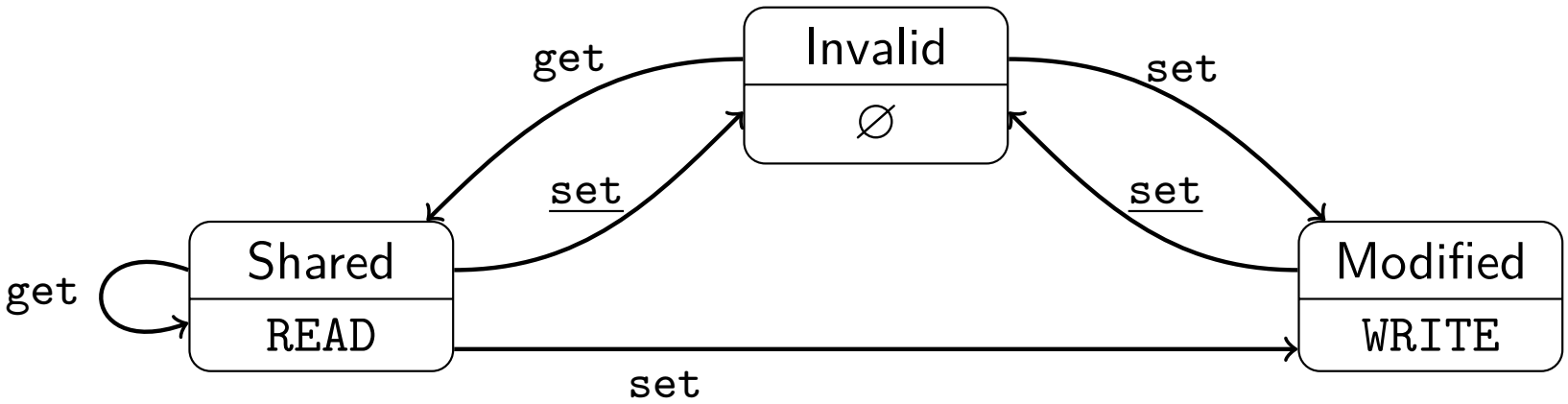


`opération locale`
`opération distante`

- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache

Le protocole MESI : une implémentation du RW-lock

- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour

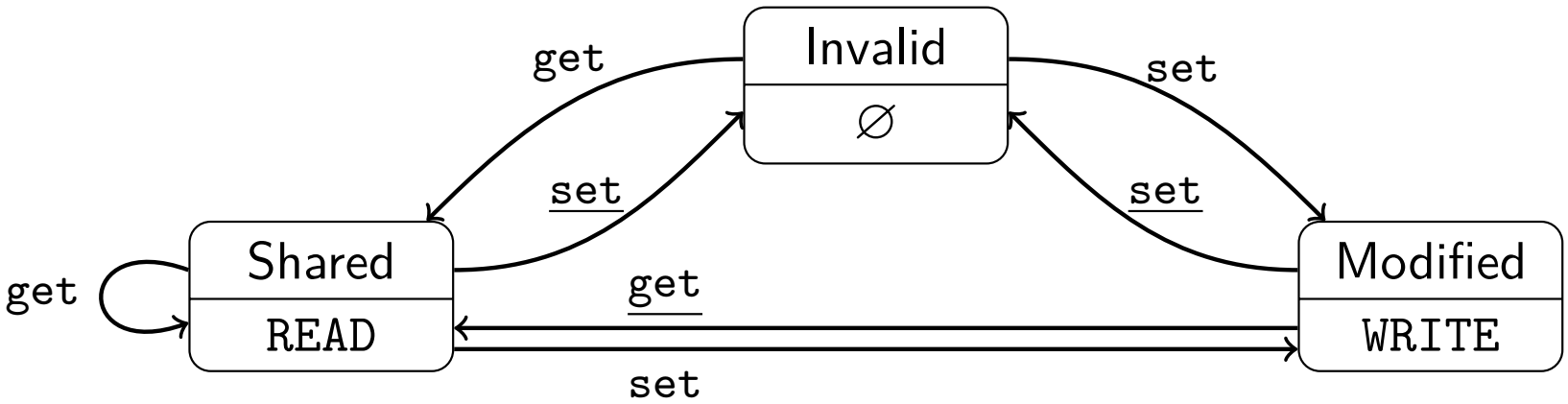


`opération locale`
`opération distante`

- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache

Le protocole MESI : une implémentation du RW-lock

- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour

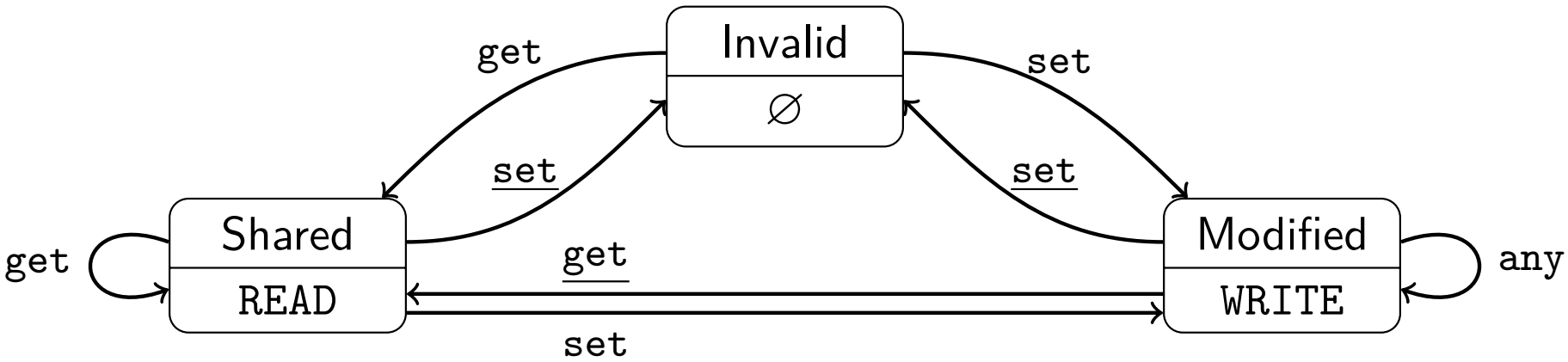


`opération locale`
`opération distante`

- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache

Le protocole MESI : une implémentation du RW-lock

- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour

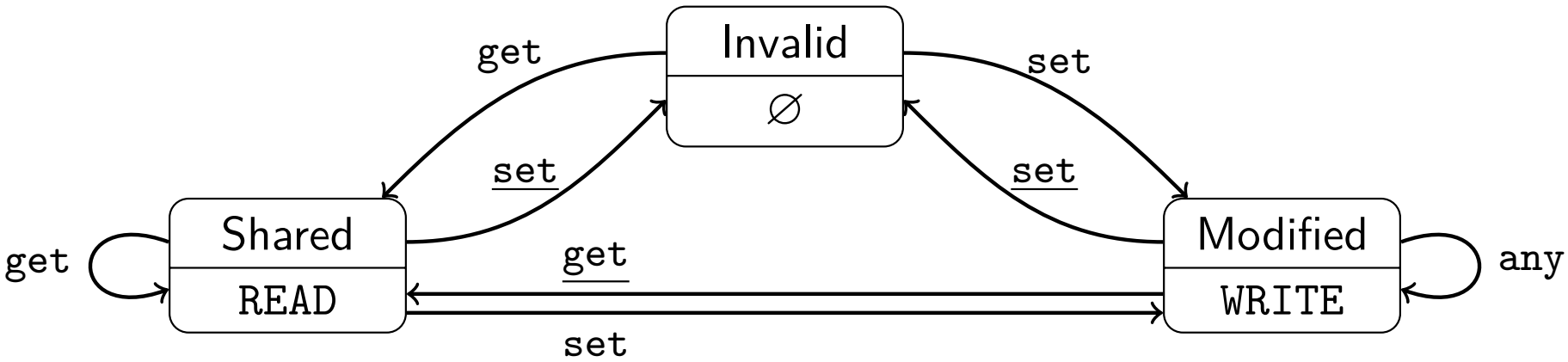


`opération locale`
`opération distante`

- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache

Le protocole MESI : une implémentation du RW-lock

- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour

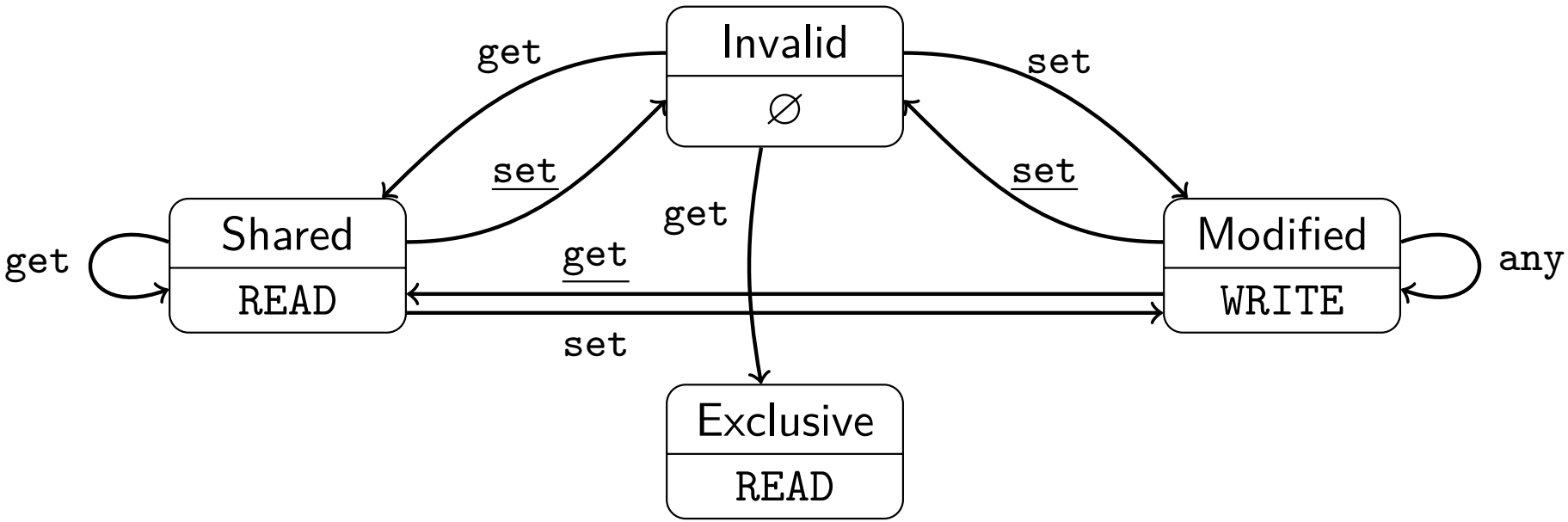


`opération locale`
`opération distante`

- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache
  - La ligne peut être *dirty* uniquement dans cet état

Le protocole MESI : une implémentation du RW-lock

- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour

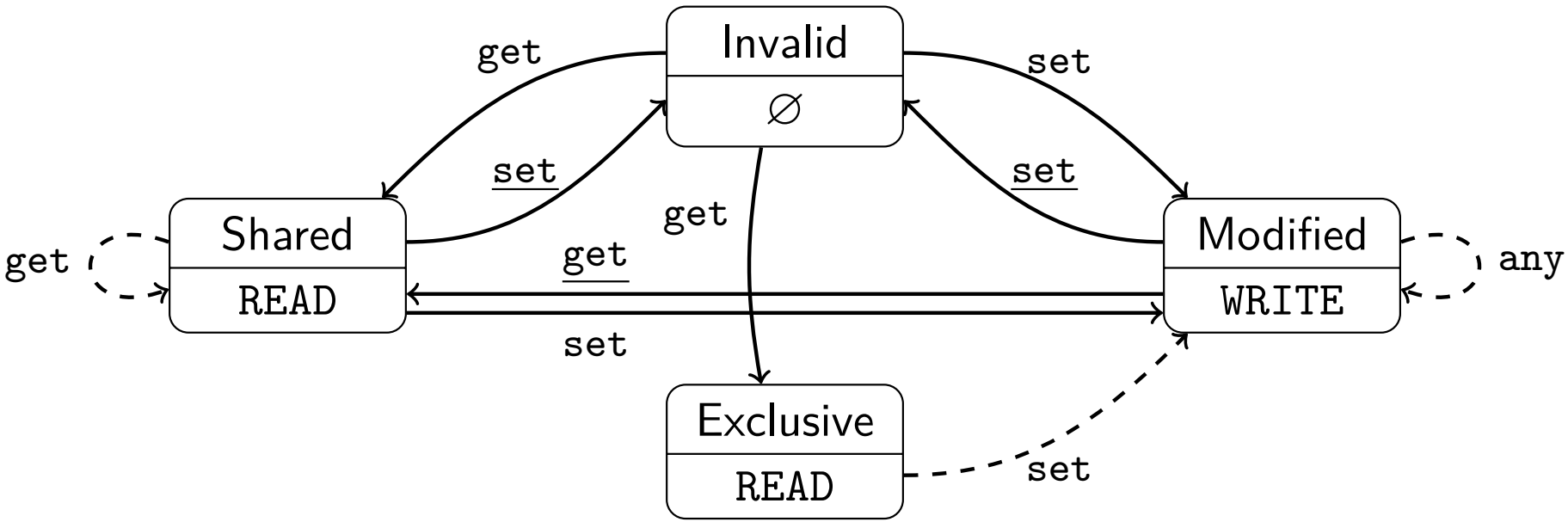


`opération locale`
`opération distante`

- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache
  - La ligne peut être *dirty* uniquement dans cet état
- Exclusive : la ligne est lue par un unique cache

Le protocole MESI : une implémentation du RW-lock

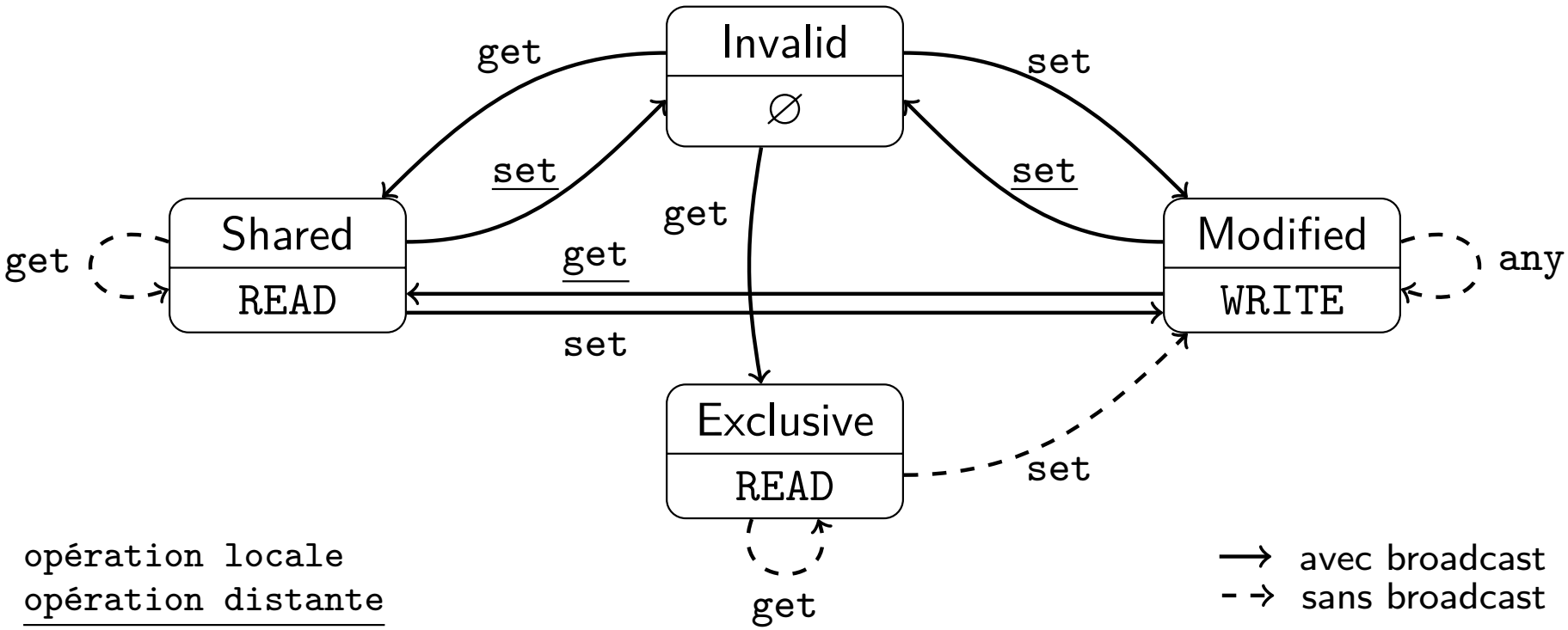
- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour



- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache
  - La ligne peut être *dirty* uniquement dans cet état
- Exclusive : la ligne est lue par un unique cache

Le protocole MESI : une implémentation du RW-lock

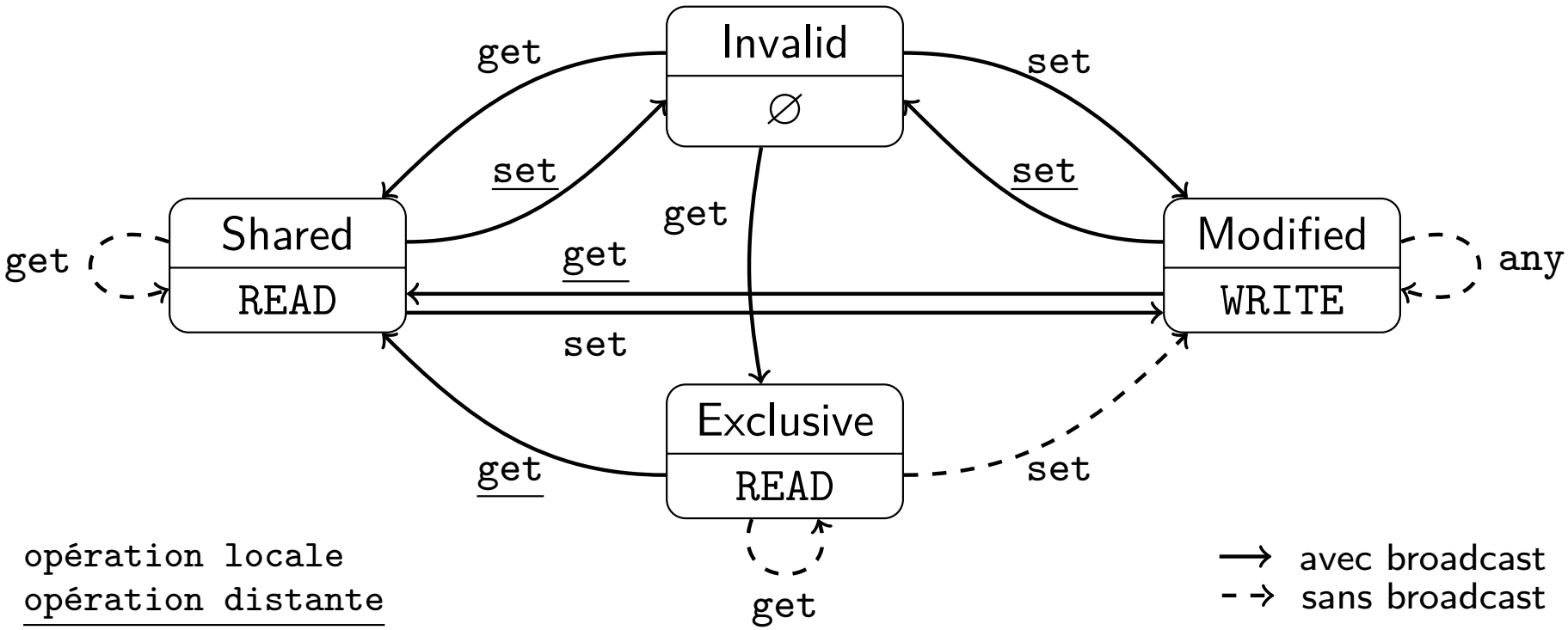
- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour



- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache
  - La ligne peut être *dirty* uniquement dans cet état
- Exclusive : la ligne est lue par un unique cache

Le protocole MESI : une implémentation du RW-lock

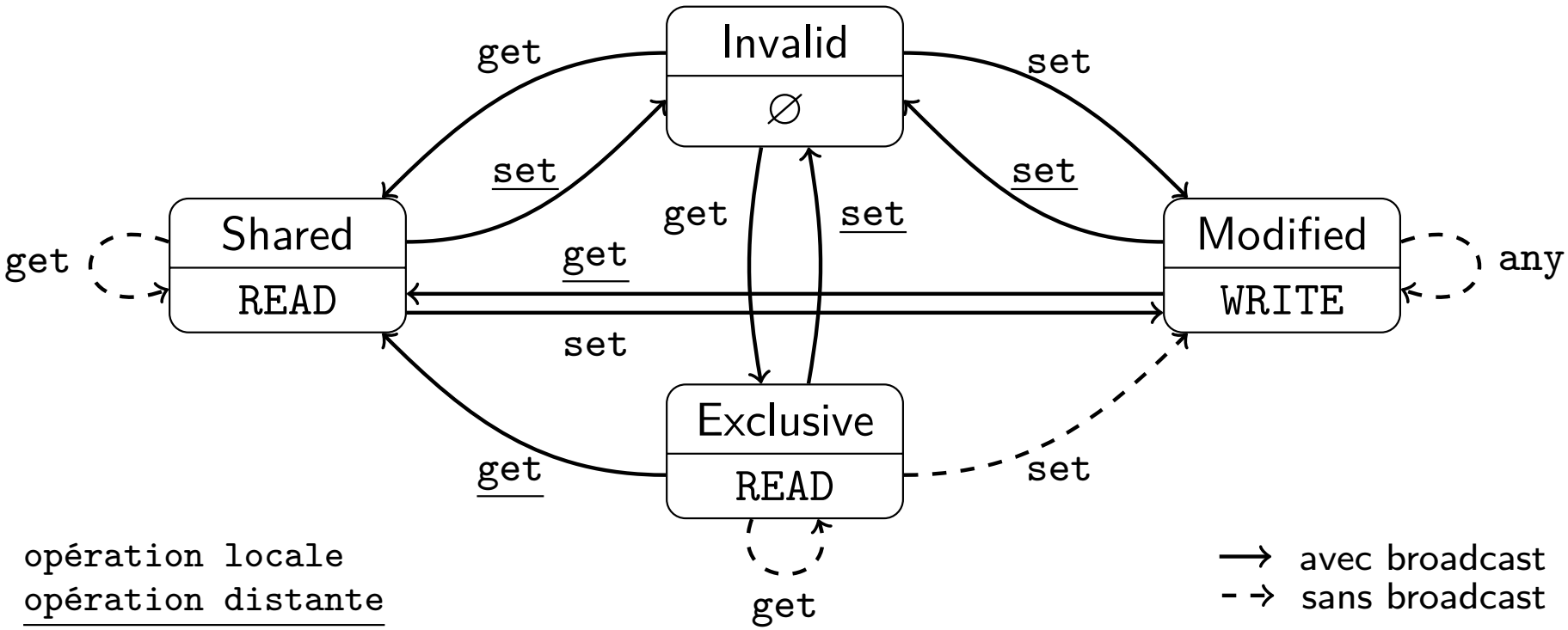
- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour



- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache
  - La ligne peut être *dirty* uniquement dans cet état
- Exclusive : la ligne est lue par un unique cache

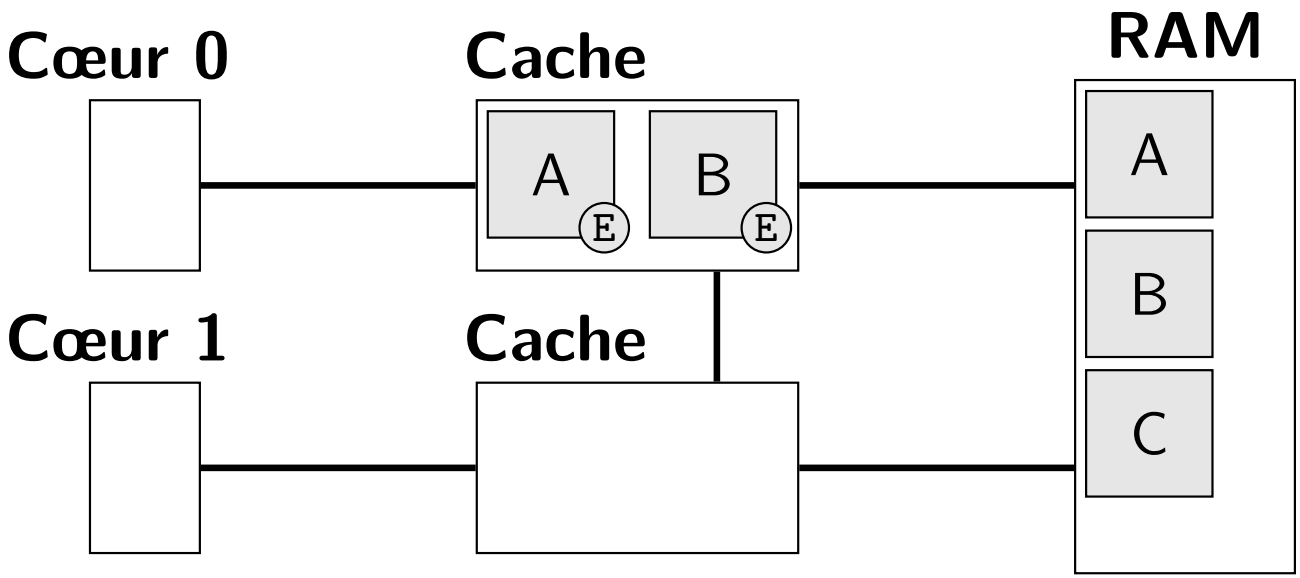
Le protocole MESI : une implémentation du RW-lock

- Chaque cache associe un état à chaque ligne
  - L'état d'une ligne indique l'état global du RW-lock
  - Les caches communiquent pour maintenir cet état à jour



- Invalid : la ligne n'est pas stockée dans le cache
- Shared : la ligne est lue par (potentiellement) plusieurs caches
- Modified : la ligne est lue/modifiée par un unique cache
  - La ligne peut être *dirty* uniquement dans cet état
- Exclusive : la ligne est lue par un unique cache

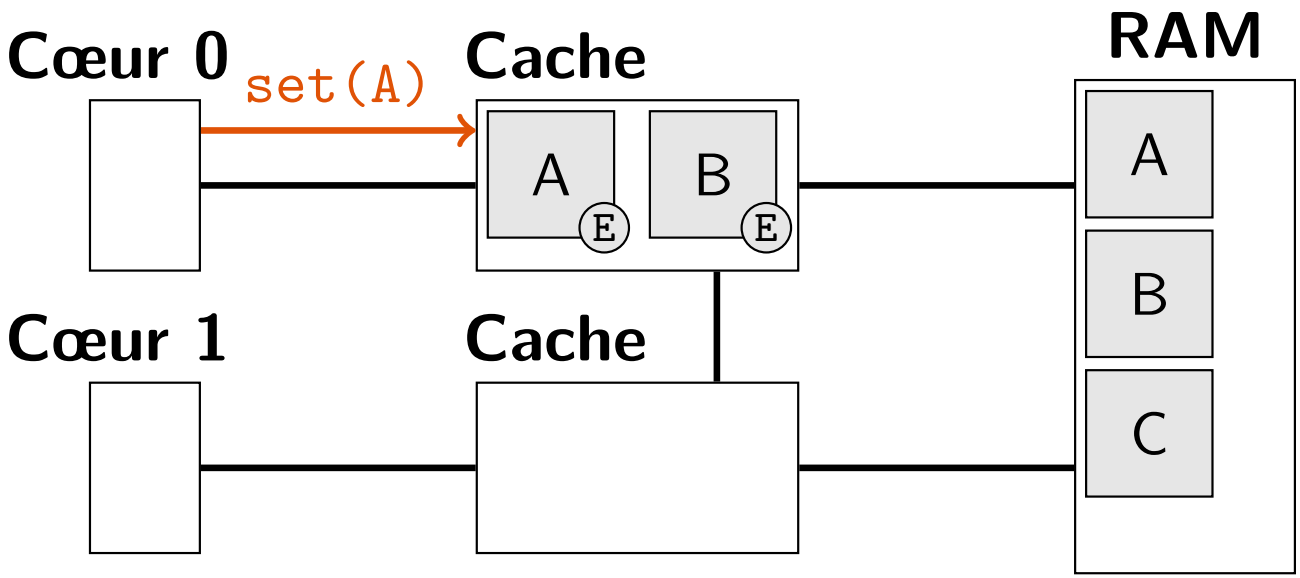
Protocol MESI : exemple




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

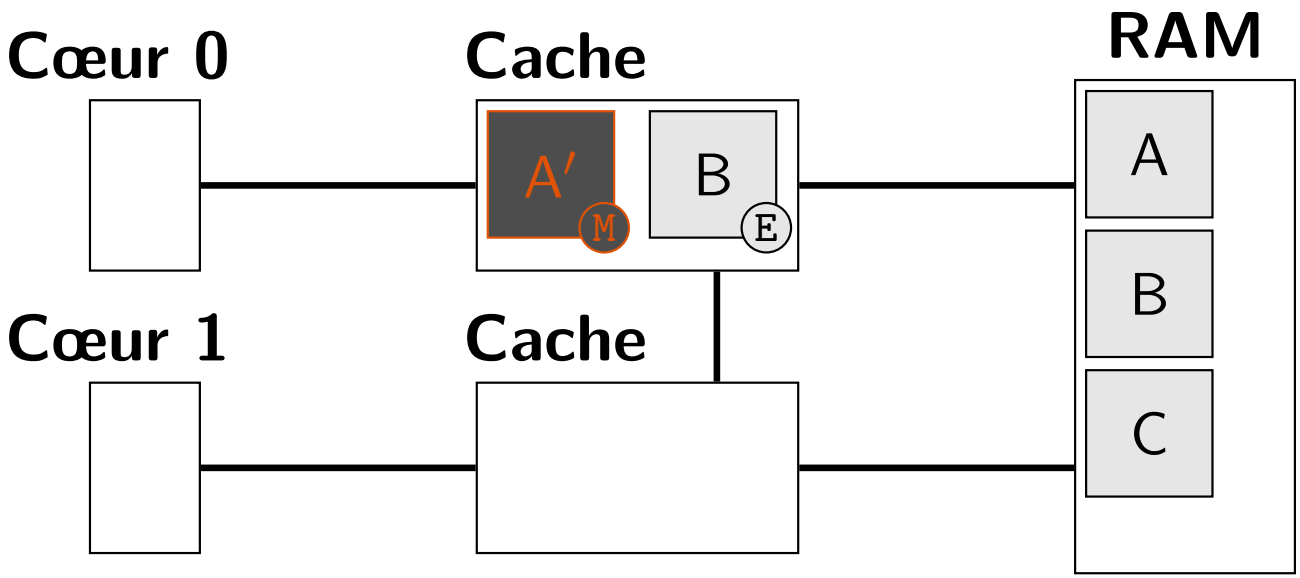
Protocol MESI : exemple




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

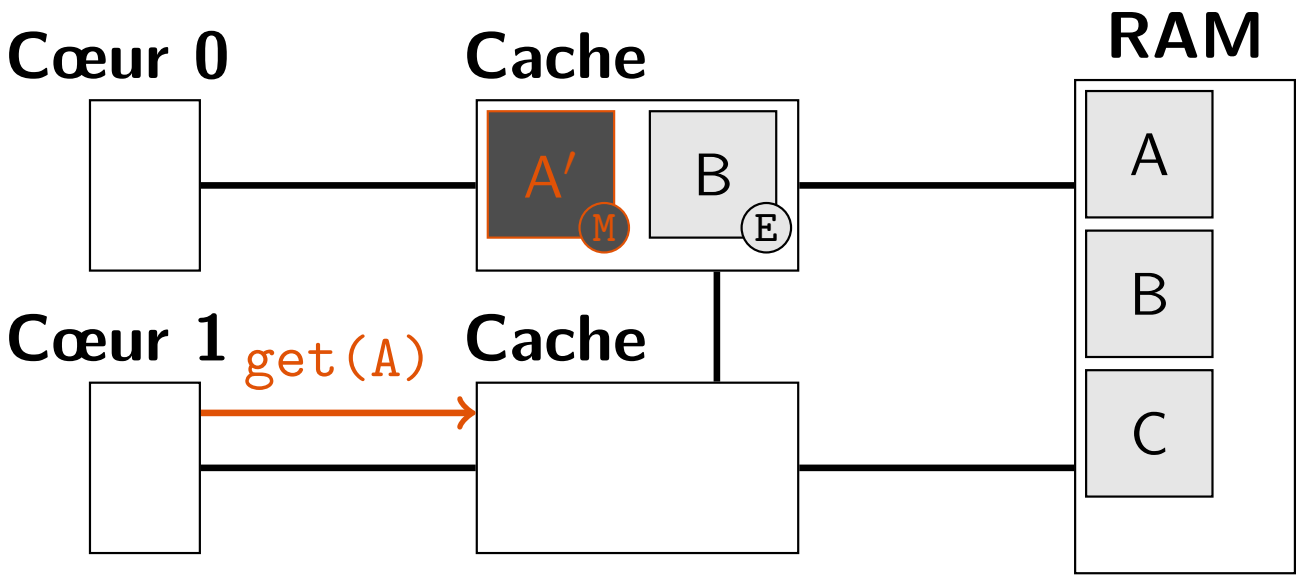
Protocol MESI : exemple




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

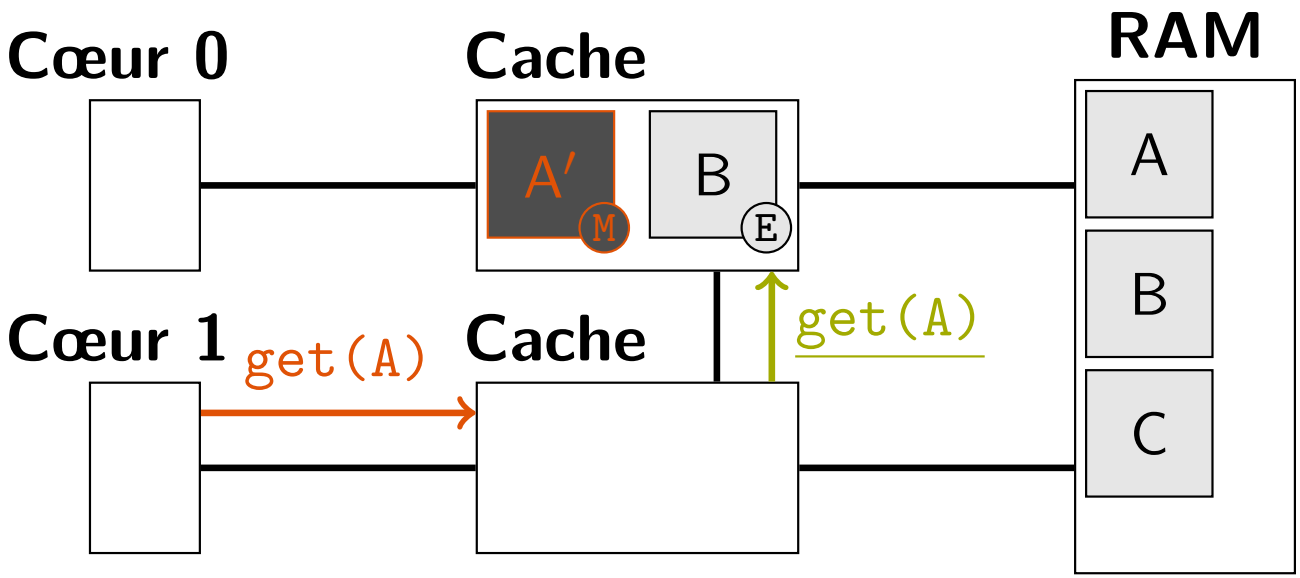
Protocol MESI : exemple




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

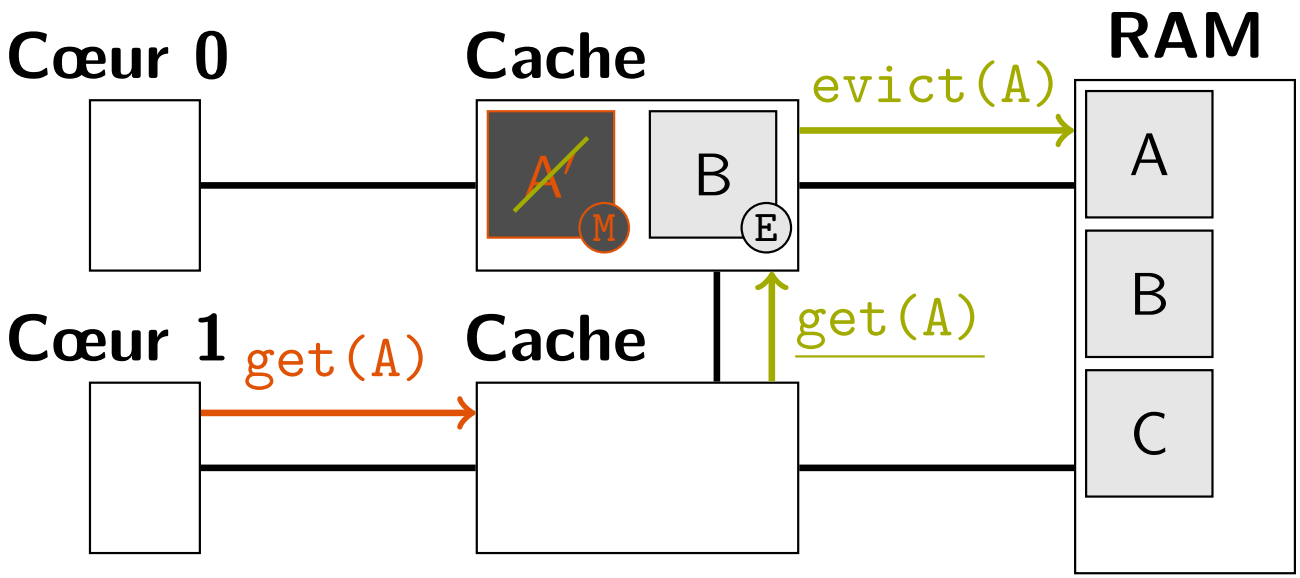
Protocol MESI : exemple




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

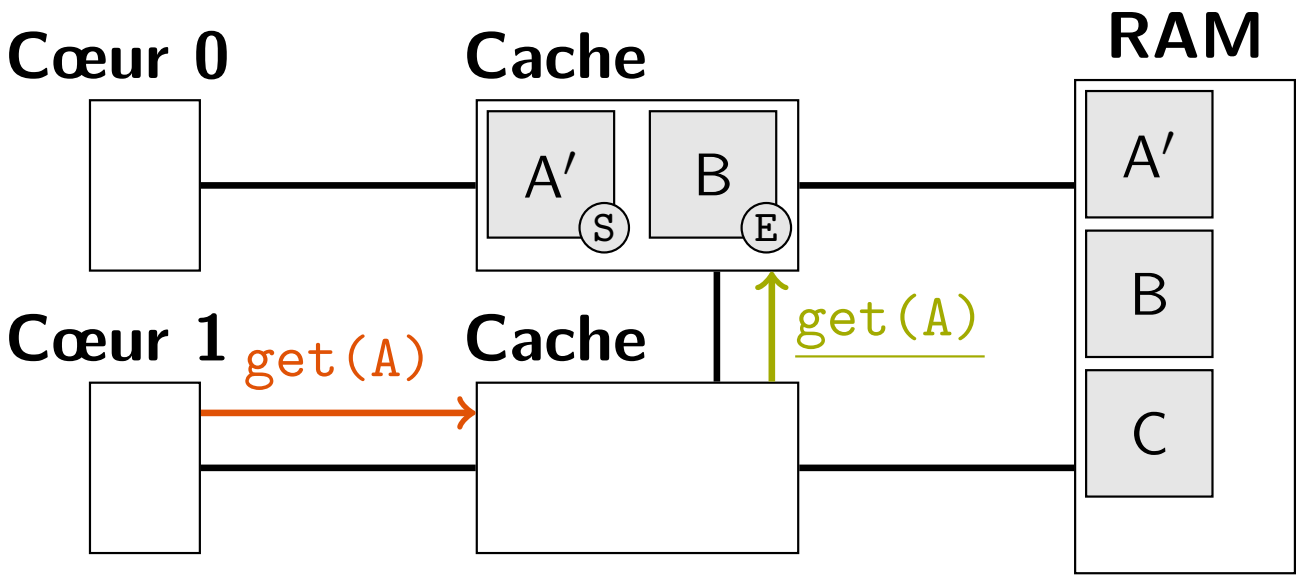
Protocol MESI : exemple




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

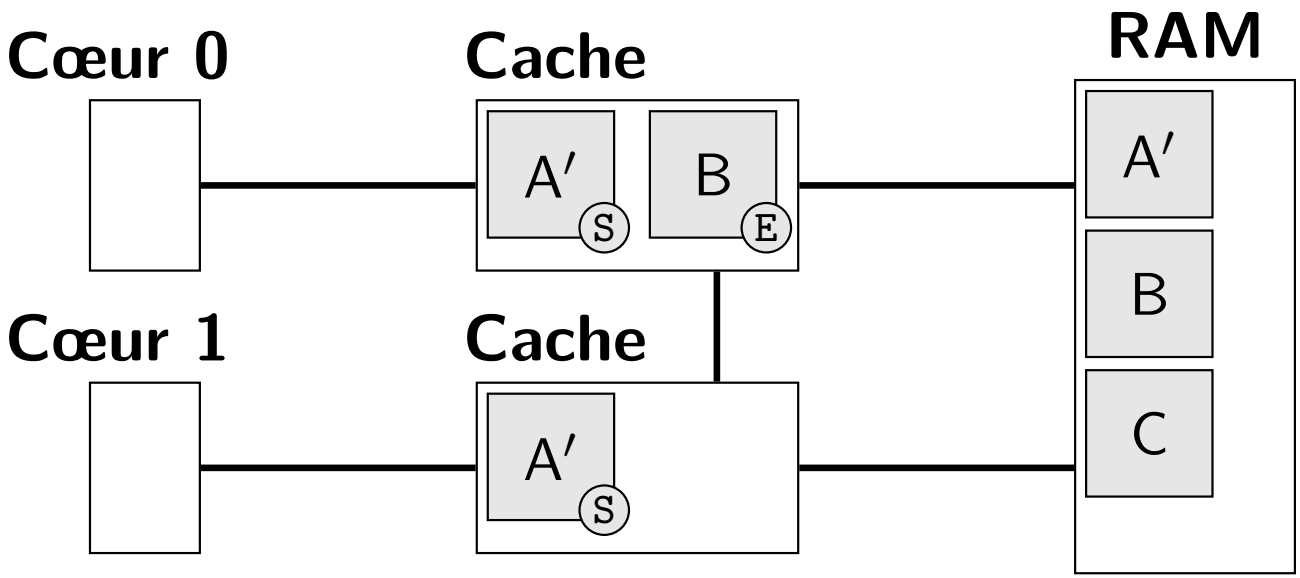



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

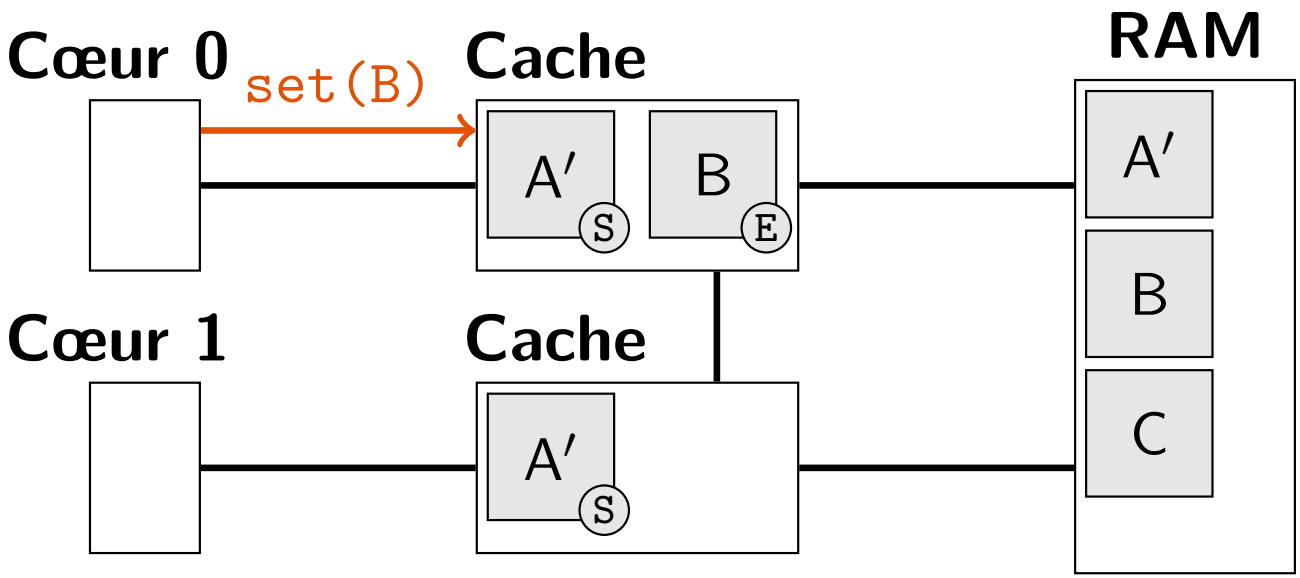



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

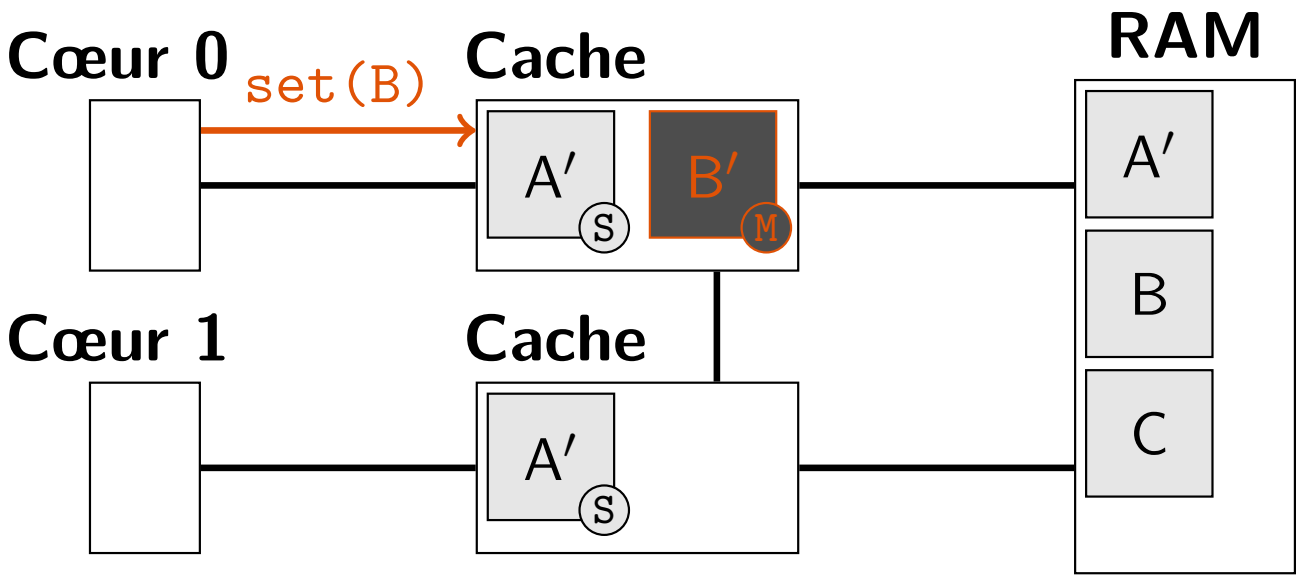



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

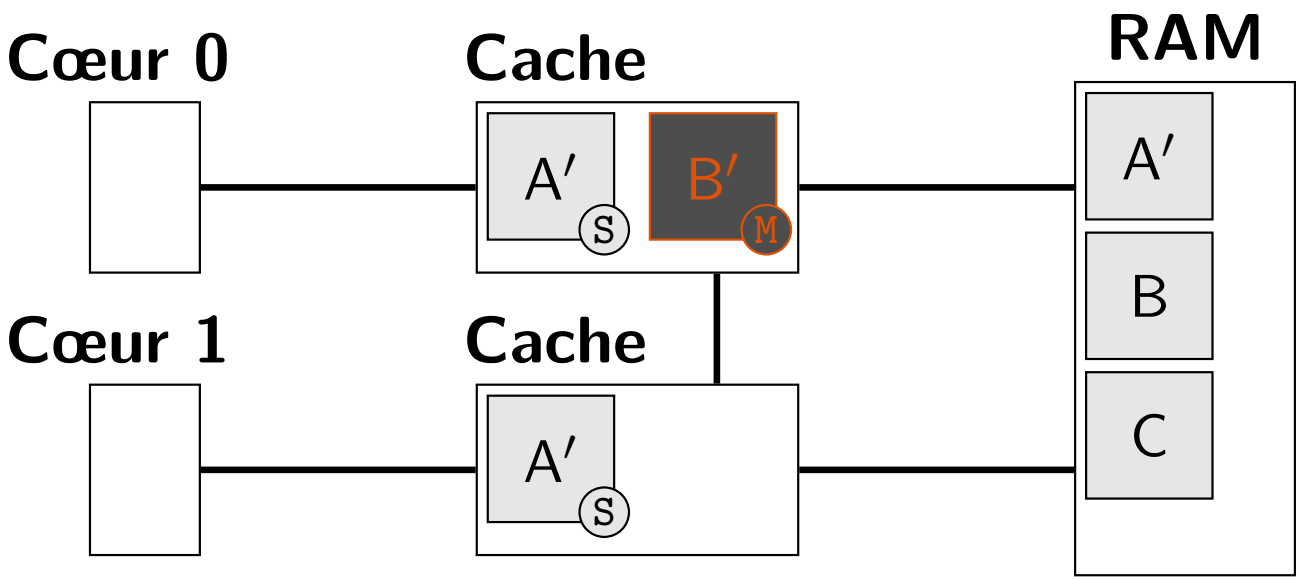



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

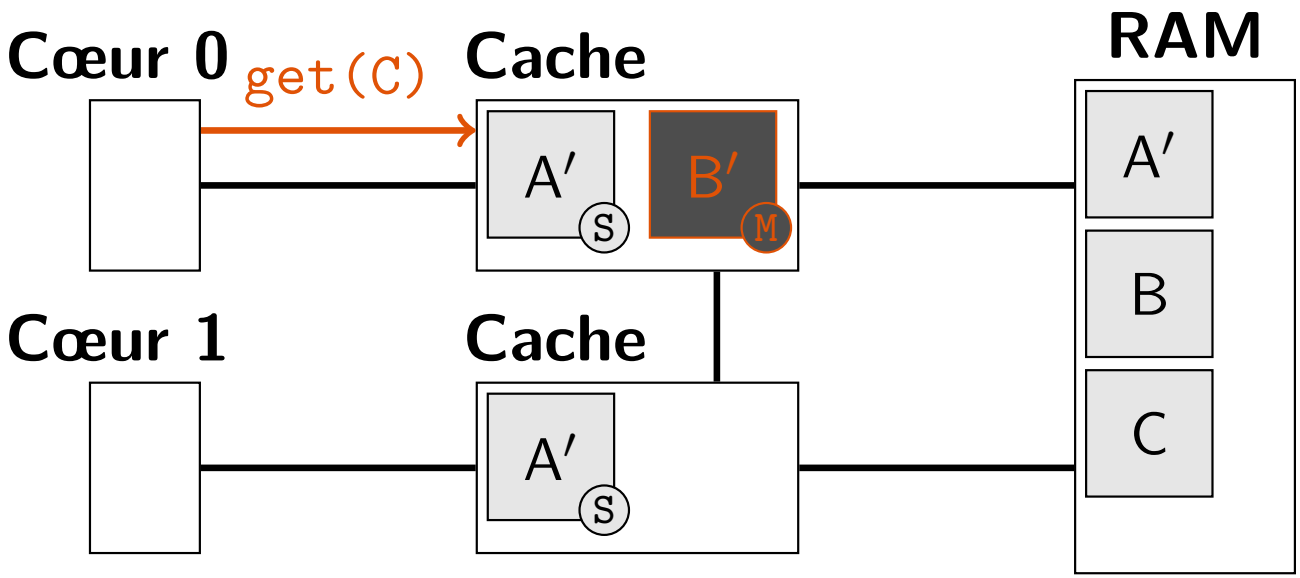



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

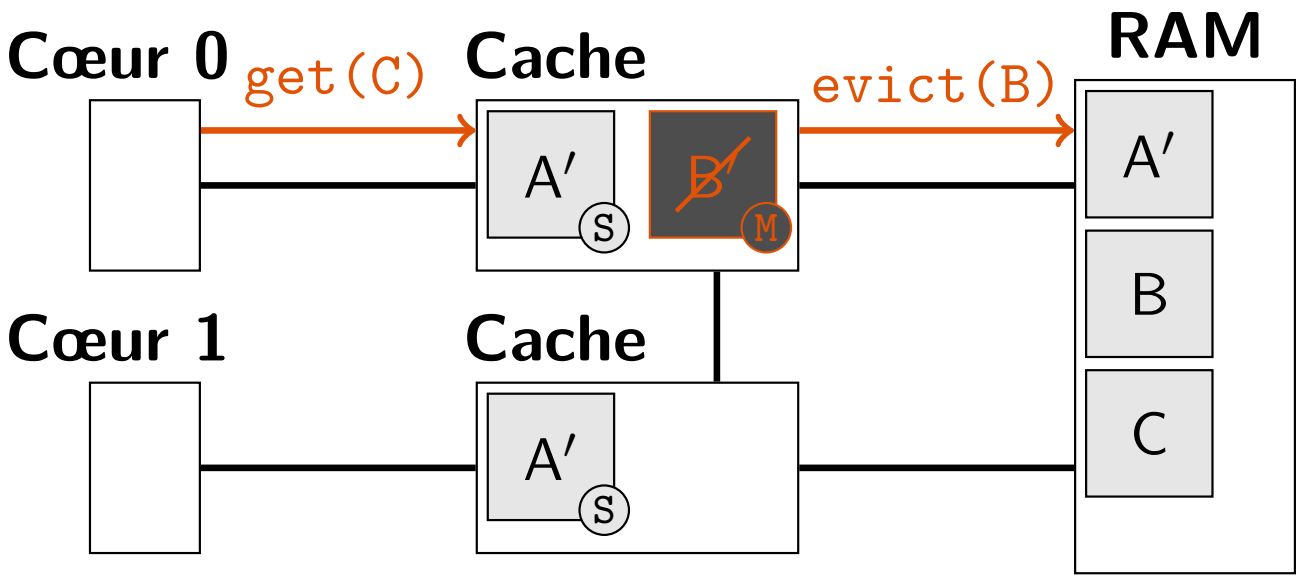



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

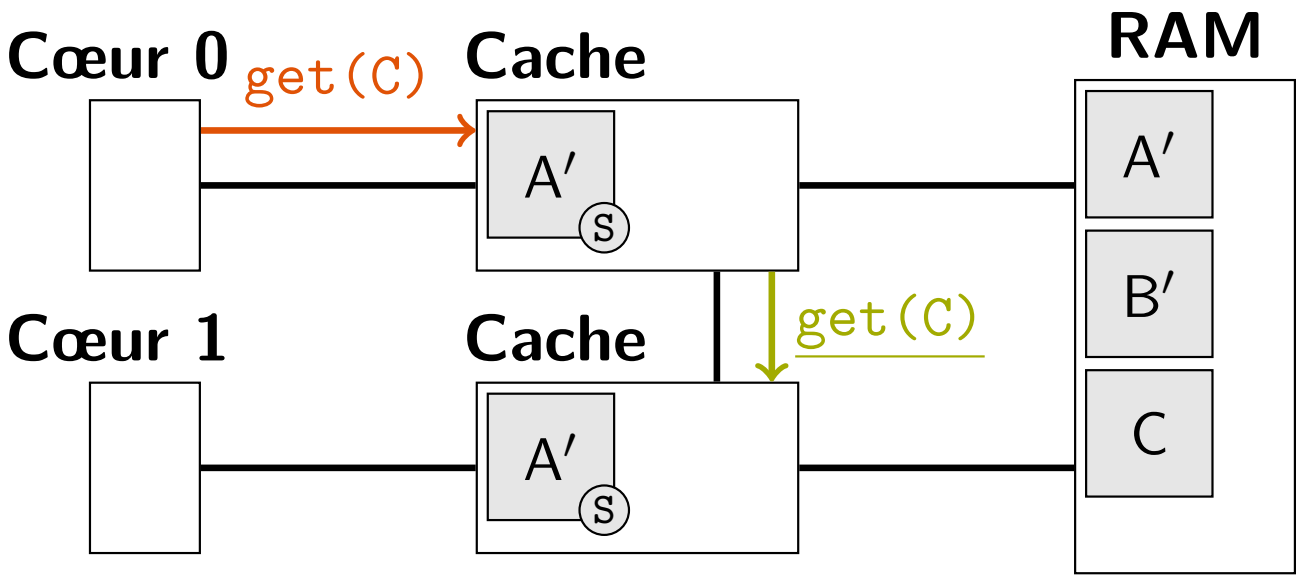



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

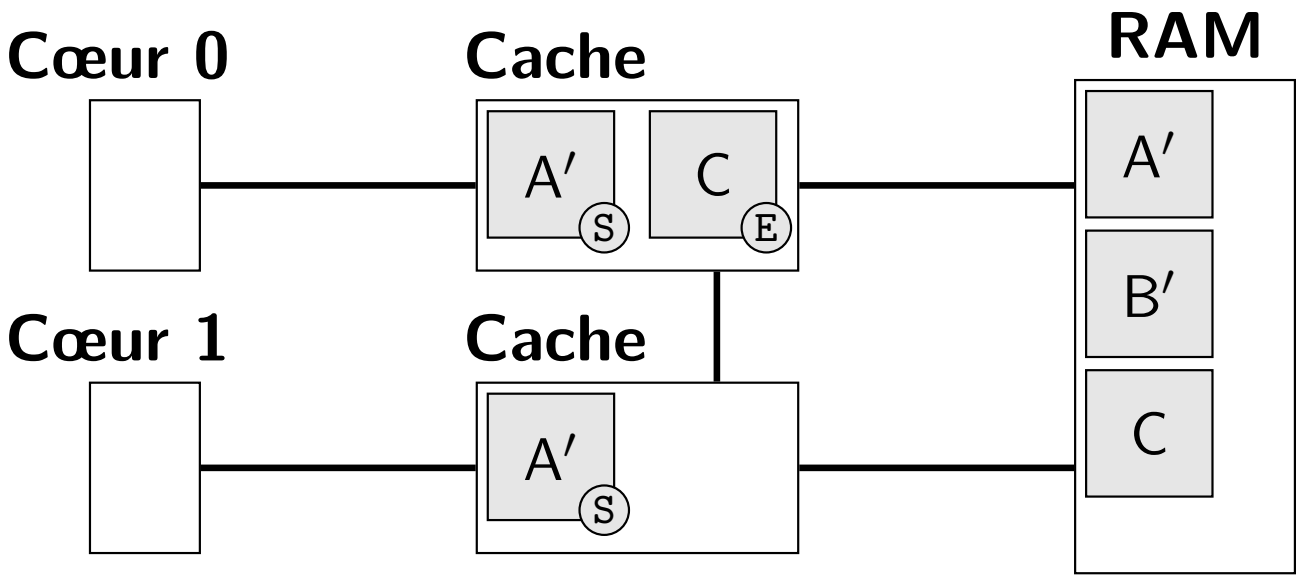



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

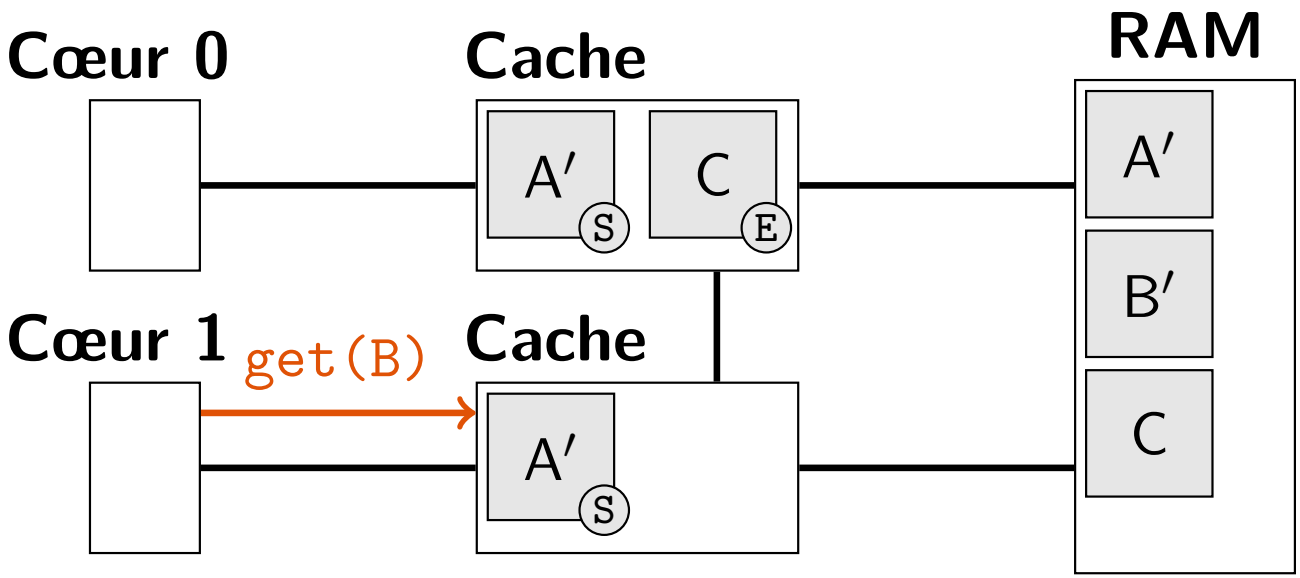



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

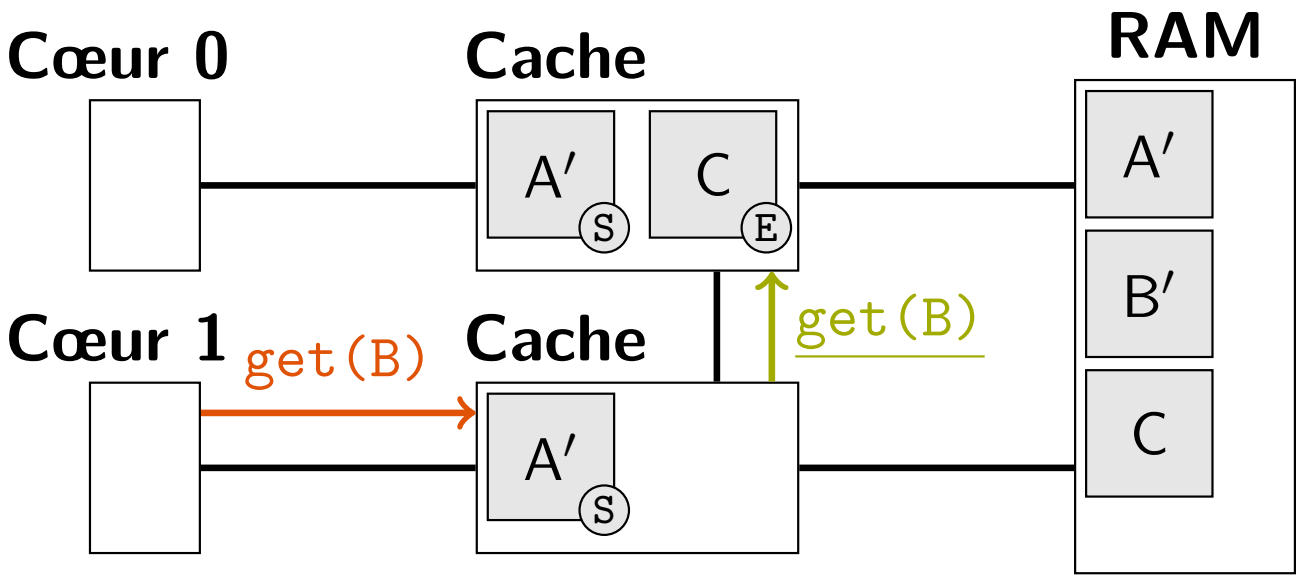



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

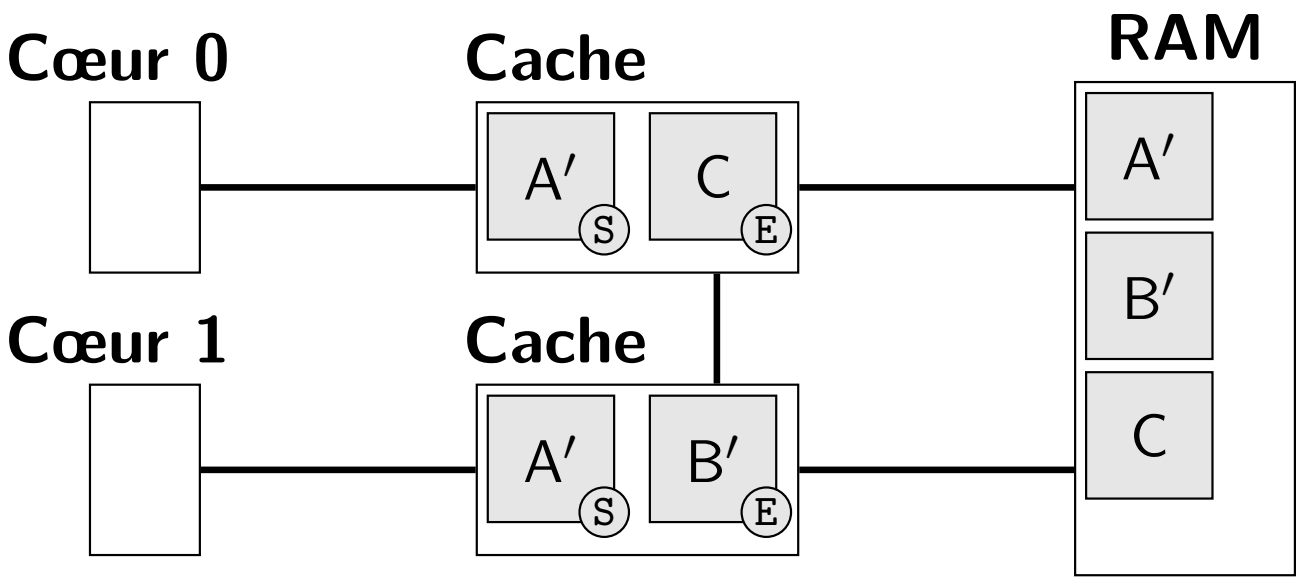



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

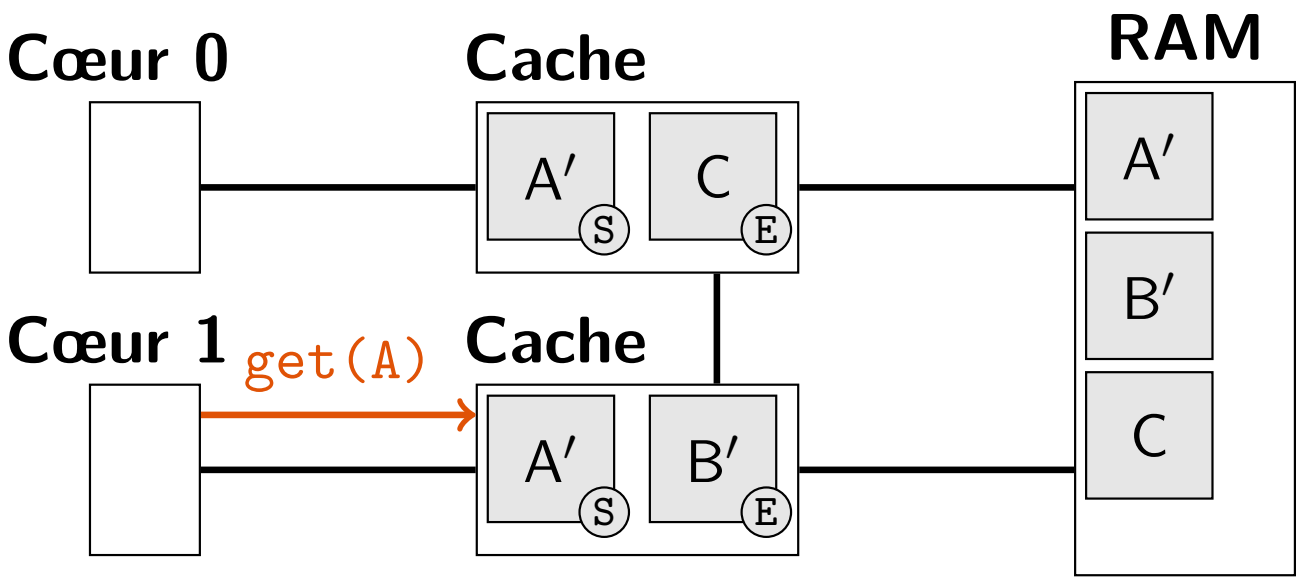



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

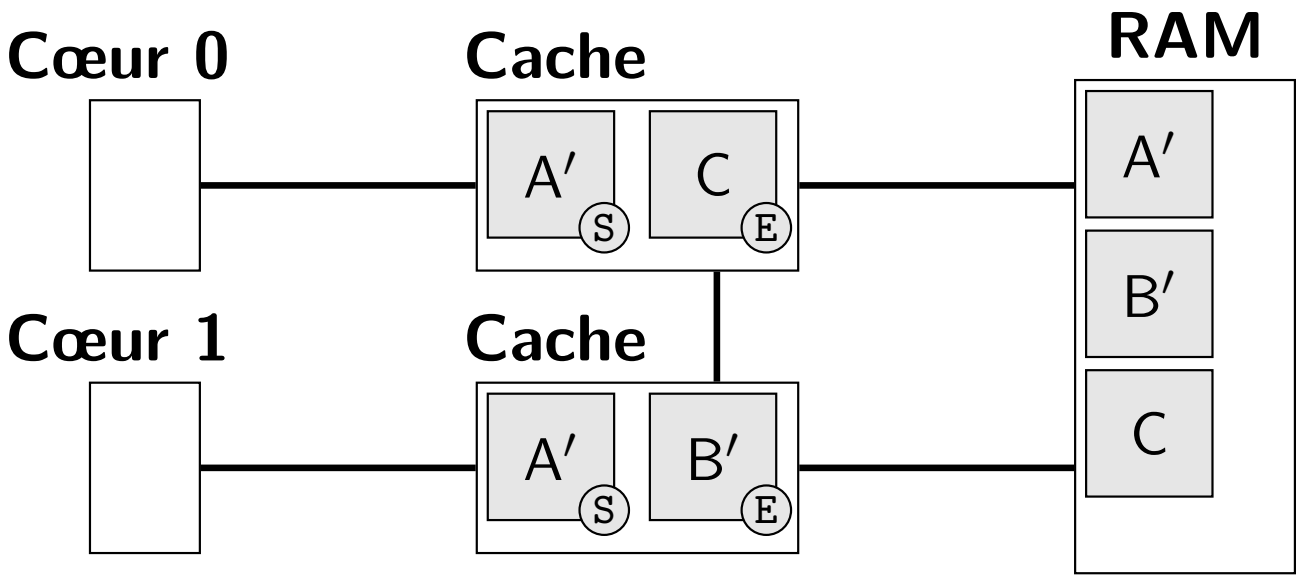



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne

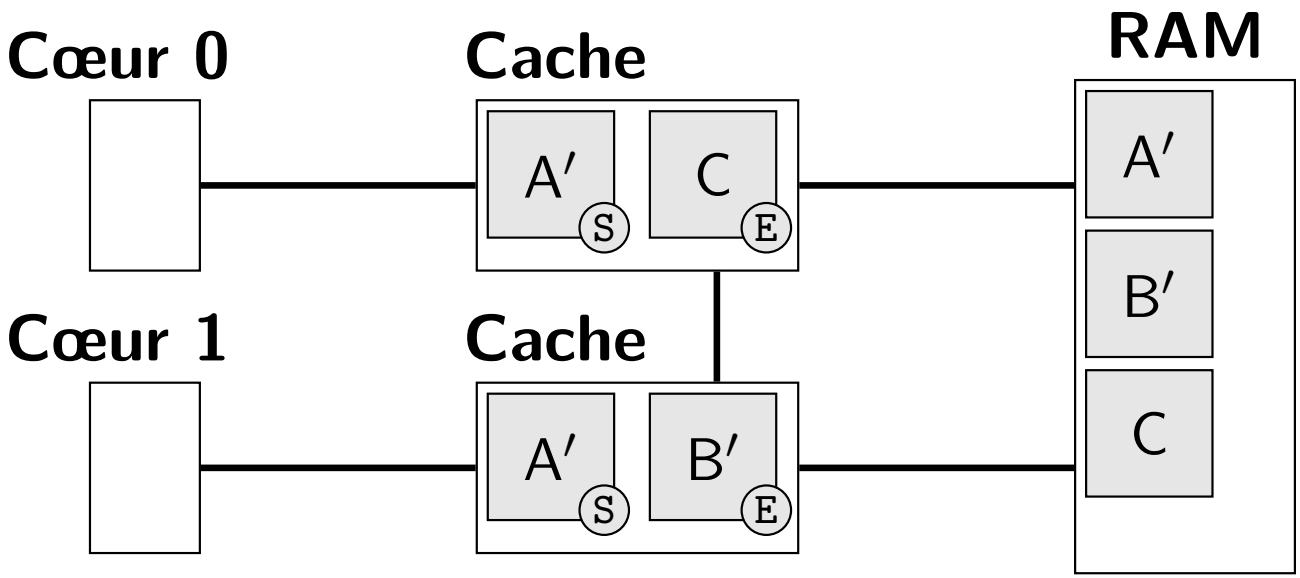



```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocol MESI : exemple

- Chaque cœur accède toujours à la dernière version de chaque ligne
- Par conséquent, la cohérence séquentielle est assurée




```
void on_core_0(void) {
    A = 1;
    B = 1;
    c = C;
}
```

```
void on_core_1(void) {
    x = A;
    b = B;
    a = A;
}
```

Protocole MESI : résumé

- Les mémoires cache actuelles assurent la cohérence séquentielle
  - Tout cœur travaille toujours sur la dernière version de chaque ligne
  - Chaque ligne est protégée par un verrou lecteur/écrivain distribué

- Une implémentation du verrou lecteur/écrivain est le protocole MESI
  - Le protocole MESI est implémenté par les caches Intel
  - Chaque ligne stockée en cache a un état
  - Invalid $= \varnothing$, Shared $=$ `READ`, Modified $=$ `WRITE`
  - Exclusive $=$ `READ` $\rightarrow$ optimisation en cas d'accesseur unique
  - Les caches maintiennent les états de chaque ligne à jour en communiquant

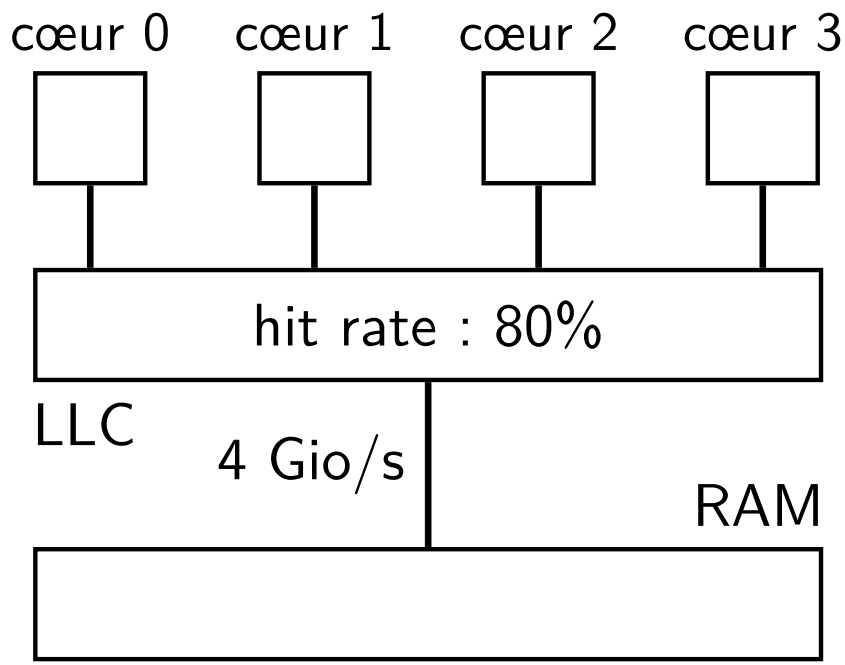
- Il existe plusieurs variantes de MESI $\rightarrow$ MOESI
  - Le protocole MOESI est implémenté par les caches AMD
  - Shared $=$ `READ` mais peut-être *dirty*
  - Owned $=$ `READ` mais *dirty* et responsable de la propagation en RAM
  - Une ligne Modified passe à Owned en cas de get distant

Plan du cours

1. Multicœur et cohérence de caches
   - Caches multicœurs et cohérence séquentielle
   - Protocole MESI
2. Architectures *Non Uniform Memory Access*
   - Contention matérielle et architecture NUMA
   - Cohérence NUMA et *cache directory*
   - Stratégies d'allocation mémoire

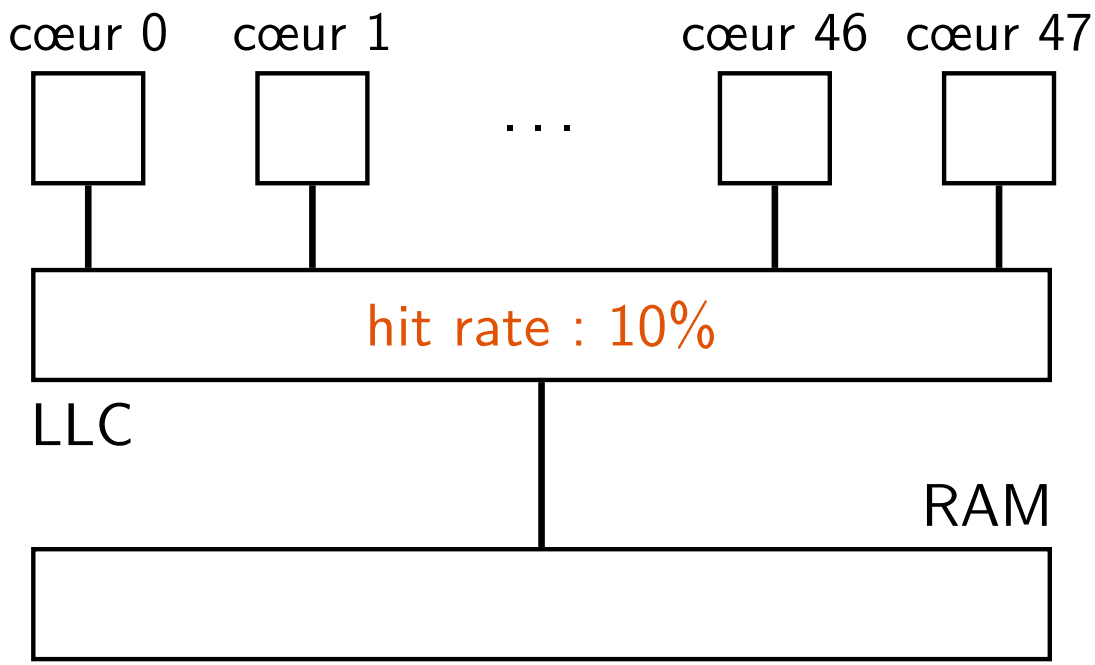
Multicœur et contention matérielle

- Entre 1 et $\sim$ 16 cœurs $\rightarrow$ problème de latence mémoire
  - Solution : mémoire cache $\rightarrow$ réduit la latence mémoire
  - Bonus : réduit le nombre d'accès à la mémoire principale



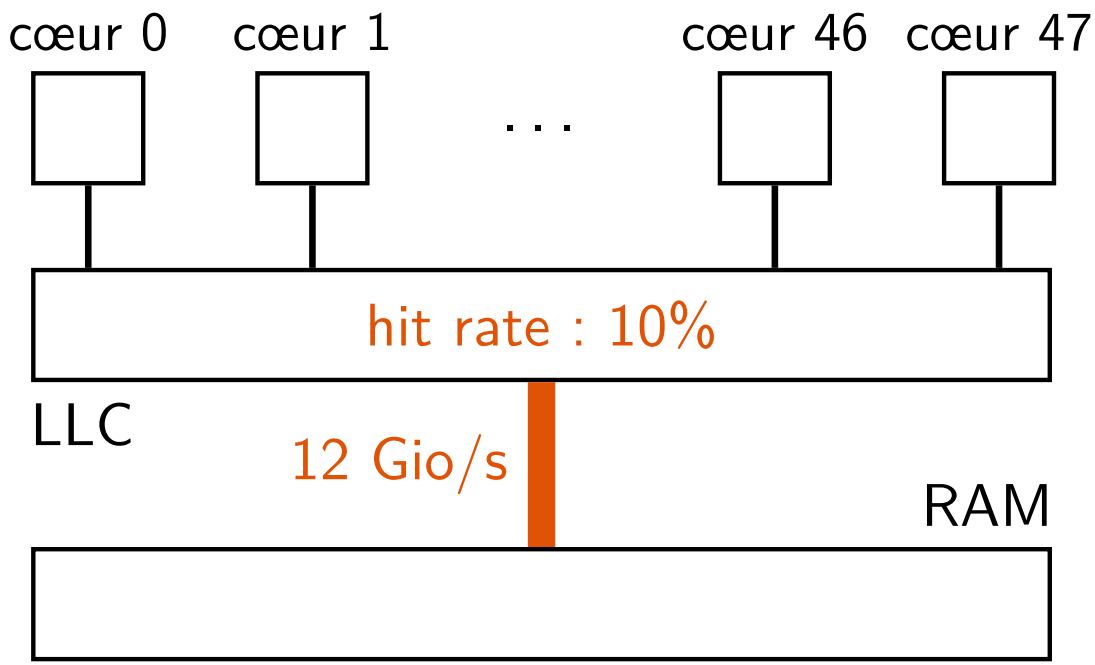
Multicœur et contention matérielle

- Entre 1 et $\sim$ 16 cœurs $\rightarrow$ problème de latence mémoire
  - Solution : mémoire cache $\rightarrow$ réduit la latence mémoire
  - Bonus : réduit le nombre d'accès à la mémoire principale
- Au delà de 16 cœurs
  - Faible localité dans les caches partagés



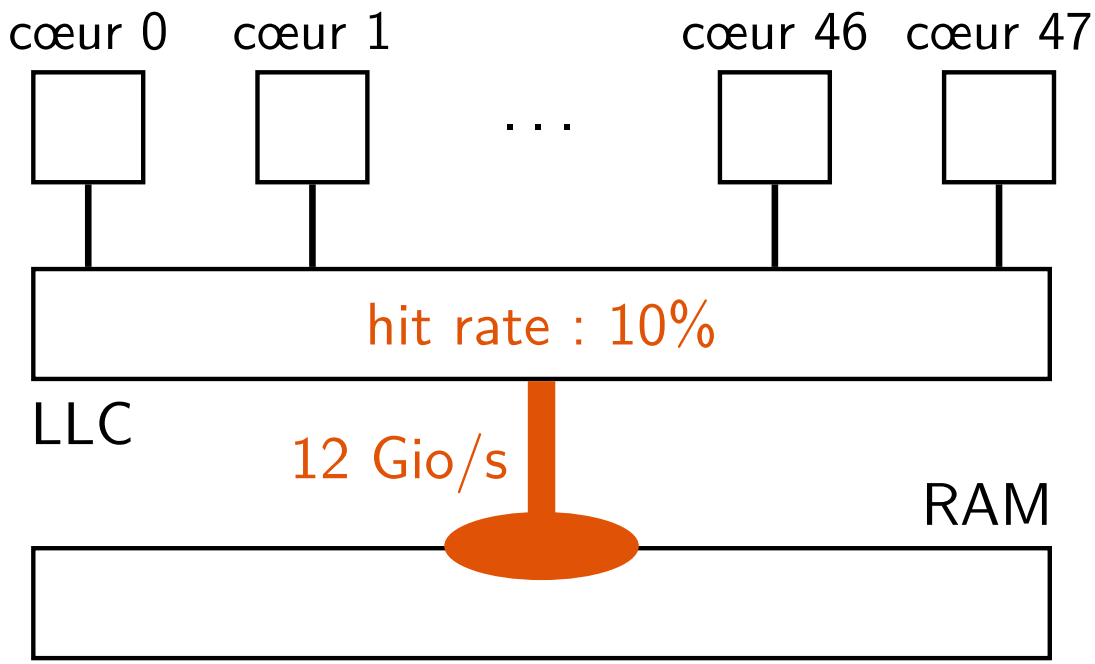
Multicœur et contention matérielle

- Entre 1 et $\sim$ 16 cœurs $\rightarrow$ problème de latence mémoire
  - Solution : mémoire cache $\rightarrow$ réduit la latence mémoire
  - Bonus : réduit le nombre d'accès à la mémoire principale
- Au delà de 16 cœurs
  - Faible localité dans les caches partagés
  - Augmente le nombre d'accès à la mémoire principale



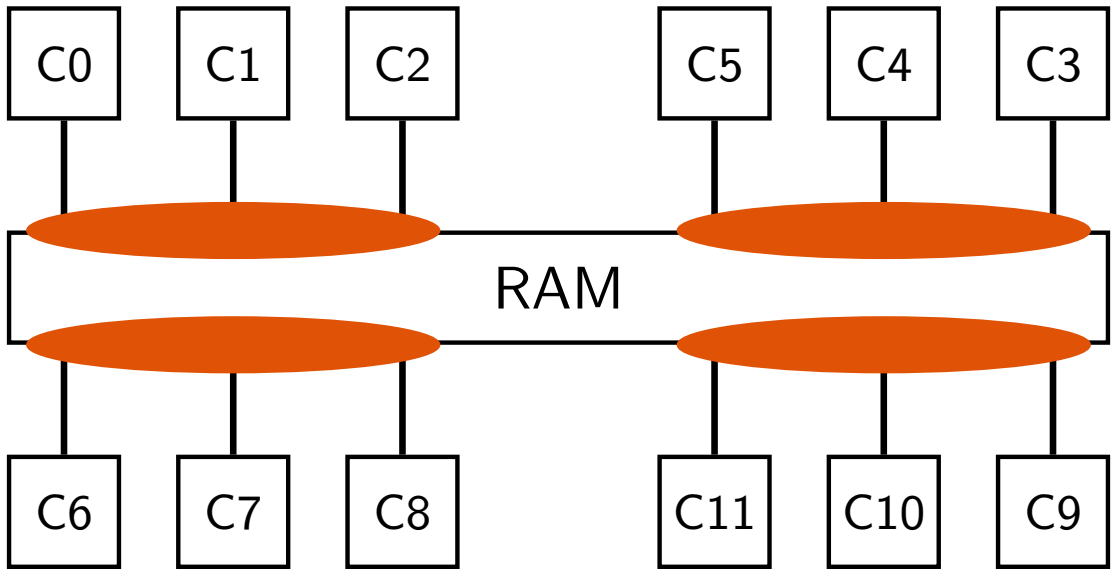
Multicœur et contention matérielle

- Entre 1 et $\sim$ 16 cœurs $\rightarrow$ problème de latence mémoire
  - Solution : mémoire cache $\rightarrow$ réduit la latence mémoire
  - Bonus : réduit le nombre d'accès à la mémoire principale
- Au delà de 16 cœurs $\rightarrow$ problème de débit mémoire
  - Faible localité dans les caches partagés
  - Augmente le nombre d'accès à la mémoire principale
  - Saturation de la mémoire principale



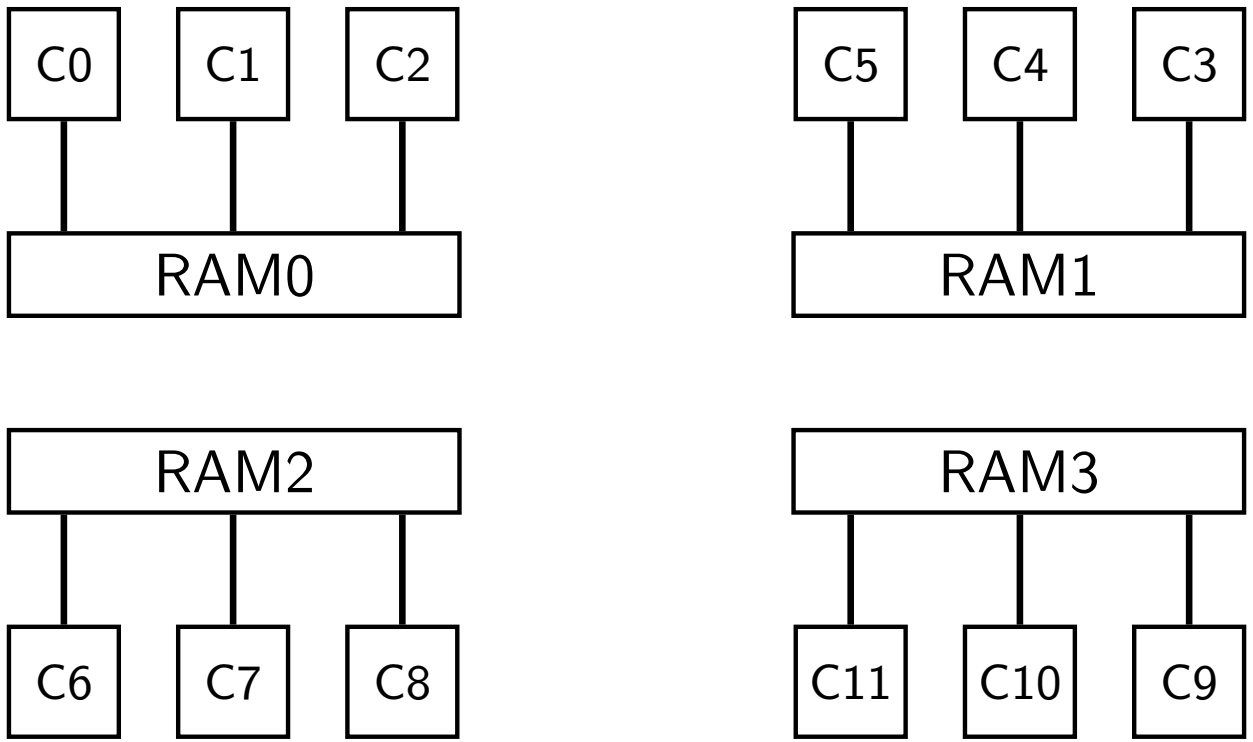
Architecture *Non Uniform Memory Access*

- Problème classique : trop de clients pour une seule ressource



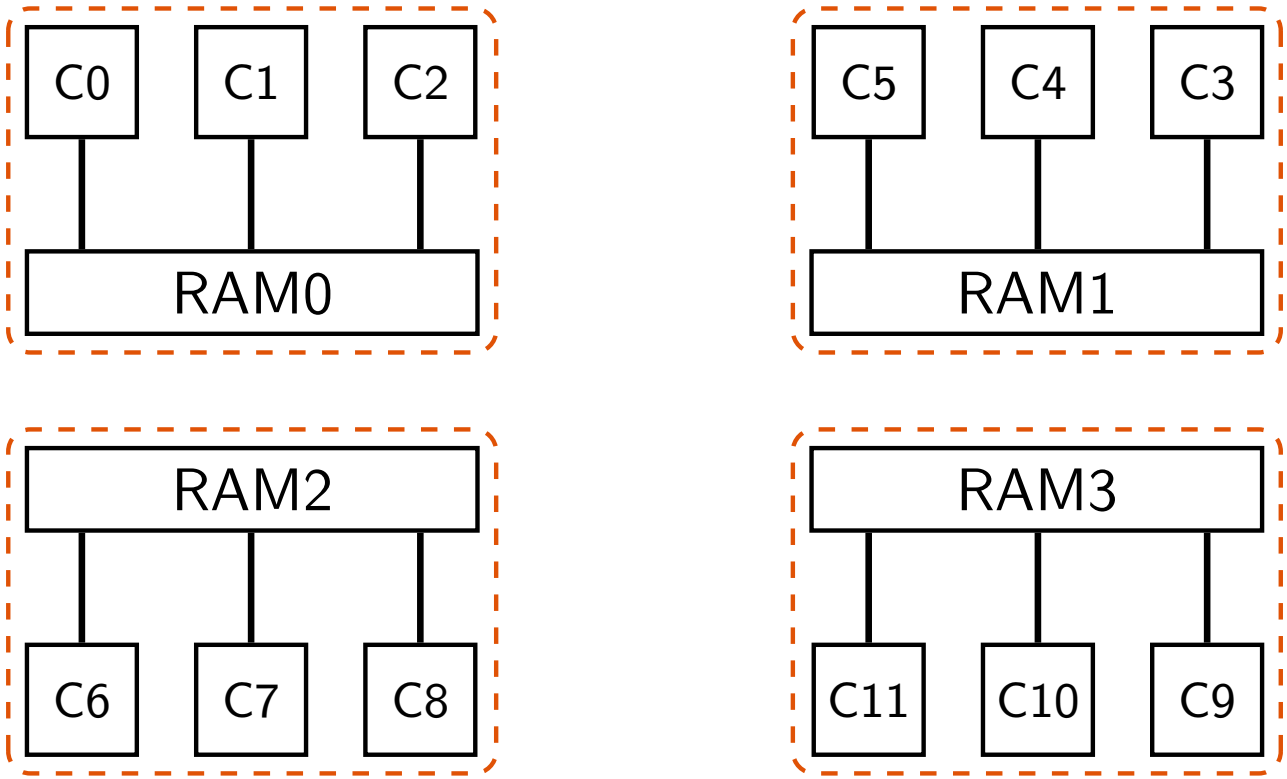
Architecture *Non Uniform Memory Access*

- Problème classique : trop de clients pour une seule ressource
- Solution classique : fragmenter / distribuer la ressource



Architecture *Non Uniform Memory Access*

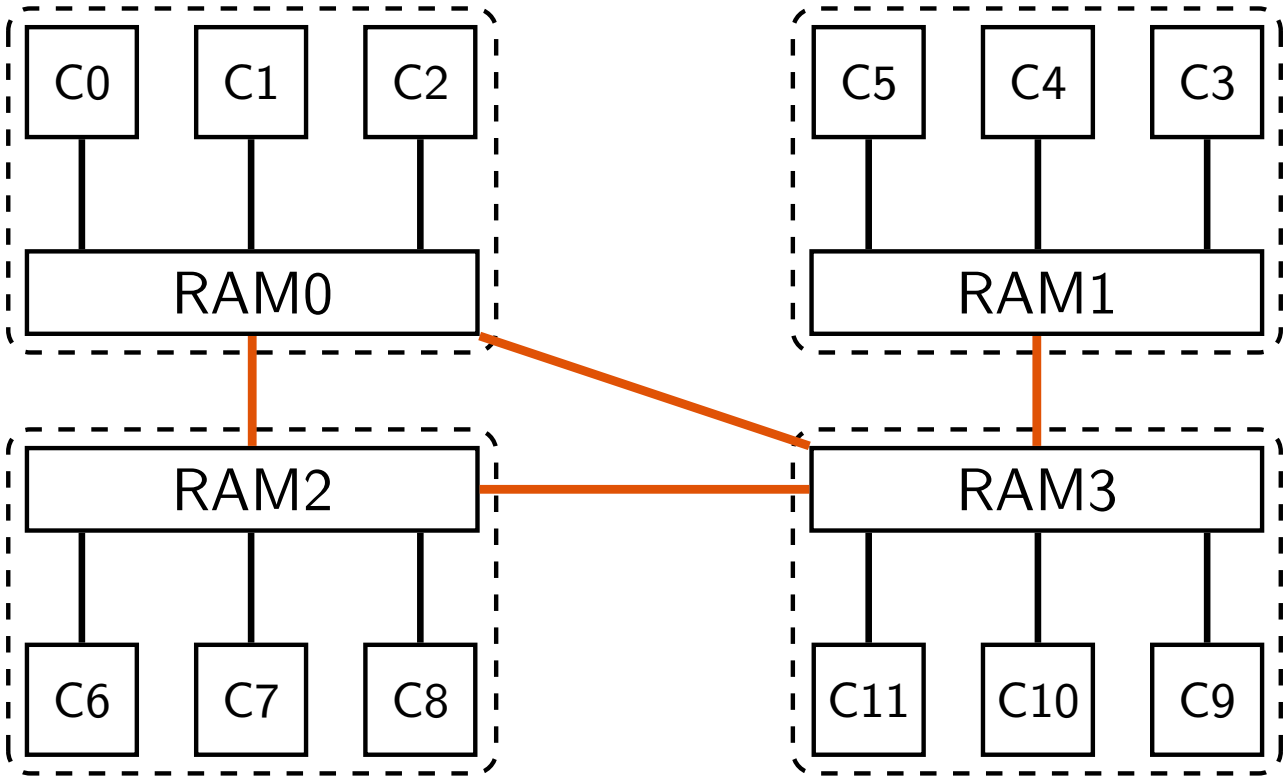
- Problème classique : trop de clients pour une seule ressource
- Solution classique : fragmenter / distribuer la ressource



- Chaque ensemble (cœurs + RAM) s'appelle un nœud

Architecture *Non Uniform Memory Access*

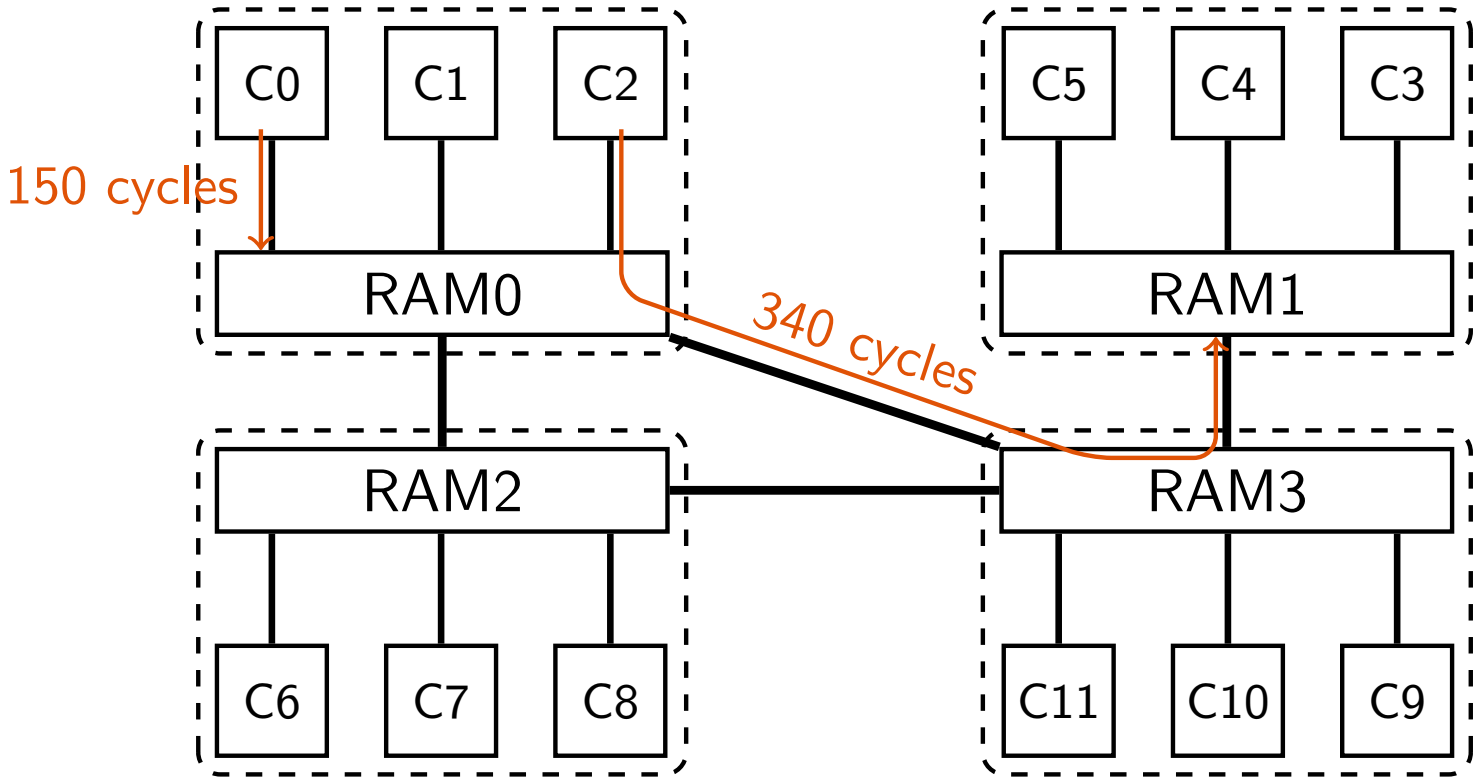
- Problème classique : trop de clients pour une seule ressource
- Solution classique : fragmenter / distribuer la ressource



- Chaque ensemble (cœurs + RAM) s'appelle un nœud
- Tout cœur peut accéder à n'importe quelle RAM via l'interconnect

Architecture *Non Uniform Memory Access*

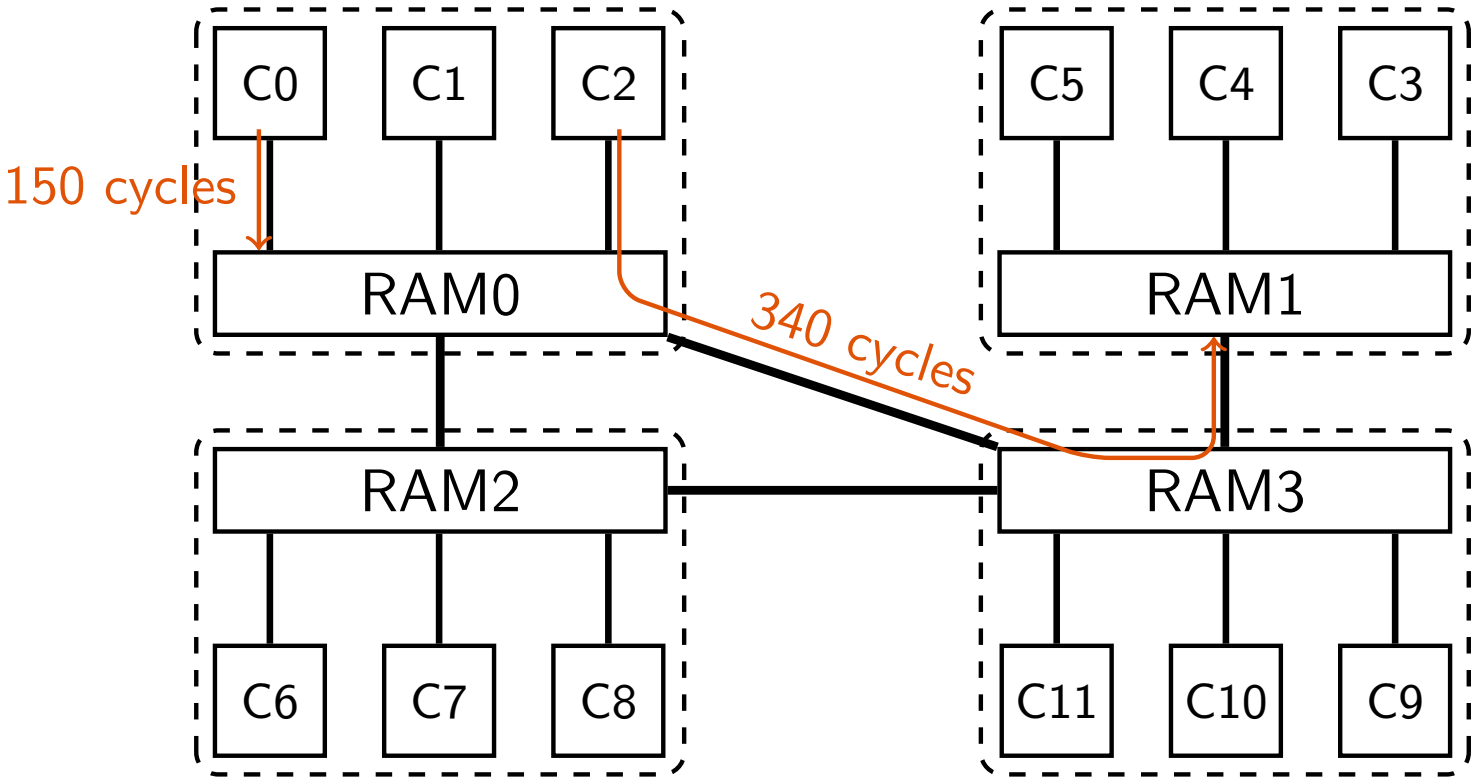
- Problème classique : trop de clients pour une seule ressource
- Solution classique : fragmenter / distribuer la ressource



- Chaque ensemble (cœurs + RAM) s'appelle un nœud
- Tout cœur peut accéder à n'importe quelle RAM via l'interconnect
- Accéder à la RAM locale est plus rapide qu'à une RAM distante

Architecture *Non Uniform Memory Access*

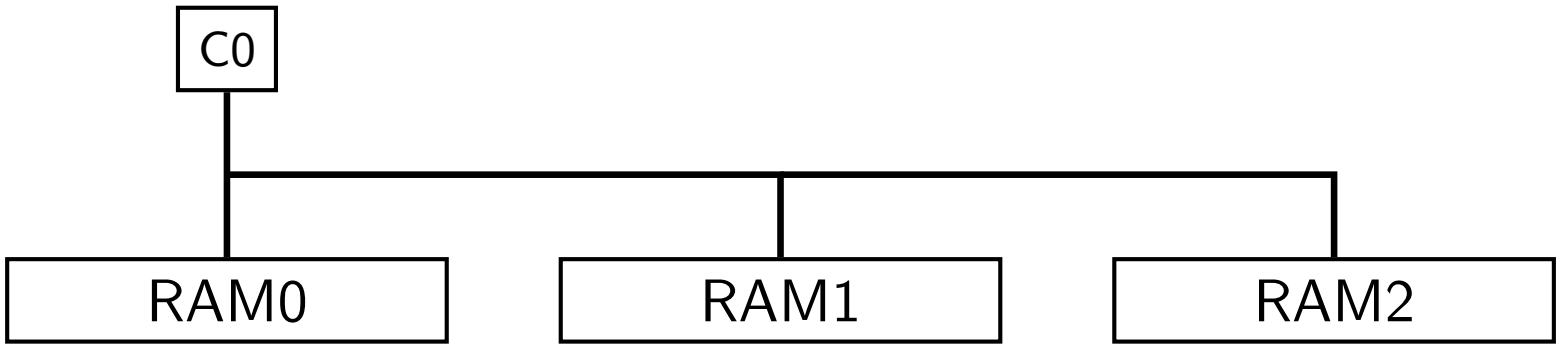
- Problème classique : trop de clients pour une seule ressource
- Solution classique : fragmenter / distribuer la ressource



- Chaque ensemble (cœurs + RAM) s'appelle un nœud NUMA
- Tout cœur peut accéder à n'importe quelle RAM via l'interconnect
- Accéder à la RAM locale est plus rapide qu'à une RAM distante
  - Les temps d'accès à la mémoire principale ne sont plus uniformes

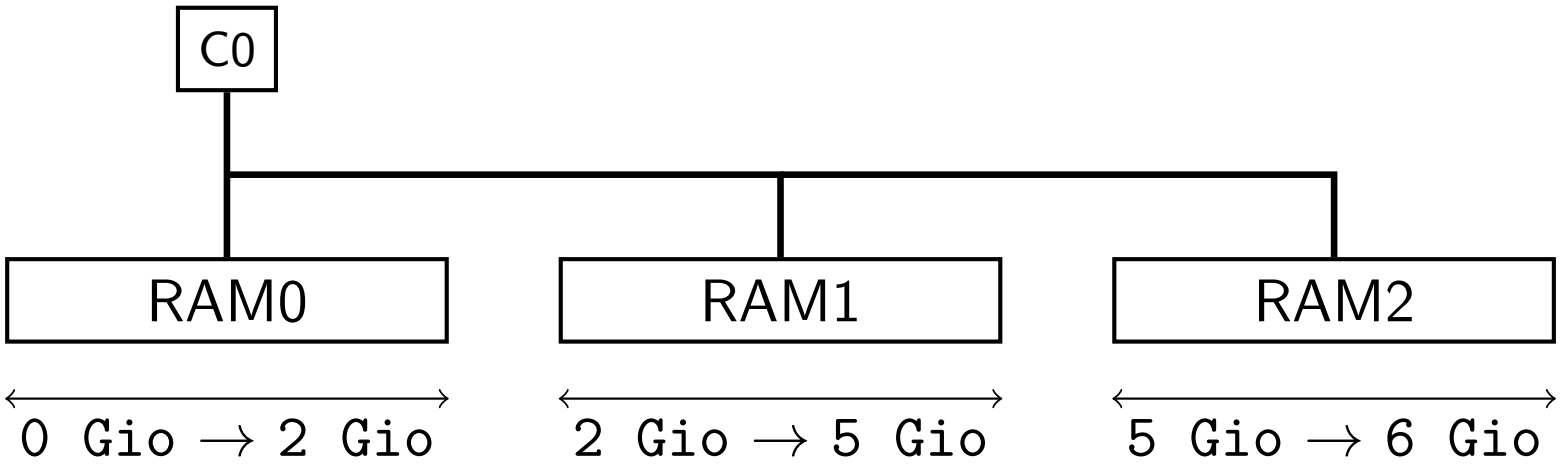
Architecture NUMA et adressage physique

- Objectif pour les constructueurs : compatibilité ascendante
  - Exécution possible du code legacy sur les nouvelles machines
- Le matériel doit présenter un espace d'adressage unique au logiciel



Architecture NUMA et adressage physique

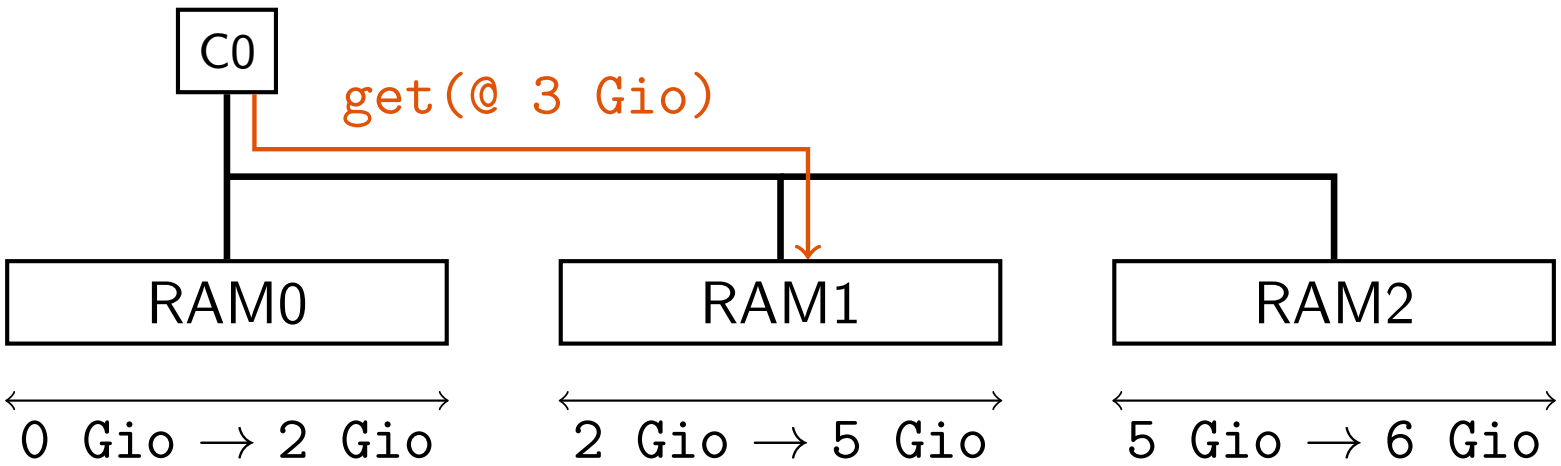
- Objectif pour les constructueurs : compatibilité ascendante
  - Exécution possible du code legacy sur les nouvelles machines
- Le matériel doit présenter un espace d'adressage unique au logiciel



- Les mémoires principales forment une partition de l'espace d'adressage physique

Architecture NUMA et adressage physique

- Objectif pour les constructueurs : compatibilité ascendante
  - Exécution possible du code legacy sur les nouvelles machines
- Le matériel doit présenter un espace d'adressage unique au logiciel



- Les mémoires principales forment une partition de l'espace d'adressage physique
- Le matériel route automatiquement les requêtes mémoire vers la bonne mémoire principale

Contention matérielle et architecture NUMA : résumé

- Augmentation du nombre de cœurs = augmentation de la pression mémoire → problème de débit mémoire
  - Au delà de $\sim$ 16 cœurs, la mémoire principale ne peut plus servir toutes les requêtes à temps → contention mémoire

- Solution : séparer le mémoire principale en plusieurs unités de mémoire indépendantes → architecture NUMA
  - Répartition de la charge mémoire
  - Le matériel (cœurs + mémoires) sont regroupés en nœuds NUMA
  - Les nœuds sont connectés entre eux par l'interconnect

- Une machine NUMA peut exécuter du code legacy
  - Chaque cœur peut accéder à l'ensemble de la mémoire
  - Le matériel présente un espace d'adressage physique unique au logiciel

- Le temps d'accès à la mémoire principale dépend du cœur et de l'adresse physique considérée

Cohérence de cache NUMA

- Un protocole de cohérence de cache assure la cohérence séquentielle
  - Les protocoles classiques (MESI, MOESI) utilise un RW-lock distribué
  - Le RW-lock est implémenté en associant un état à chaque ligne
  - Les caches maintiennent les états à jour avec des broadcasts

Cohérence de cache NUMA

- Un protocole de cohérence de cache assure la cohérence séquentielle
  - Les protocoles classiques (MESI, MOESI) utilise un RW-lock distribué
  - Le RW-lock est implémenté en associant un état à chaque ligne
  - Les caches maintiennent les états à jour avec des broadcasts

Cohérence de cache NUMA

- Un protocole de cohérence de cache assure la cohérence séquentielle
  - Les protocoles classiques (MESI, MOESI) utilise un RW-lock distribué
  - Le RW-lock est implémenté en associant un état à chaque ligne
  - Les caches maintiennent les états à jour avec des broadcasts

- Chaque broadcast passe par l'interconnect
  - Ajout de latence lors de l'acquisition d'une nouvelle ligne ($I \rightarrow *$)
  - Ajout de latence lors d'une mise à jour de l'état (sauf $E \rightarrow M$)
  - Consommation de bande passante

Cohérence de cache NUMA

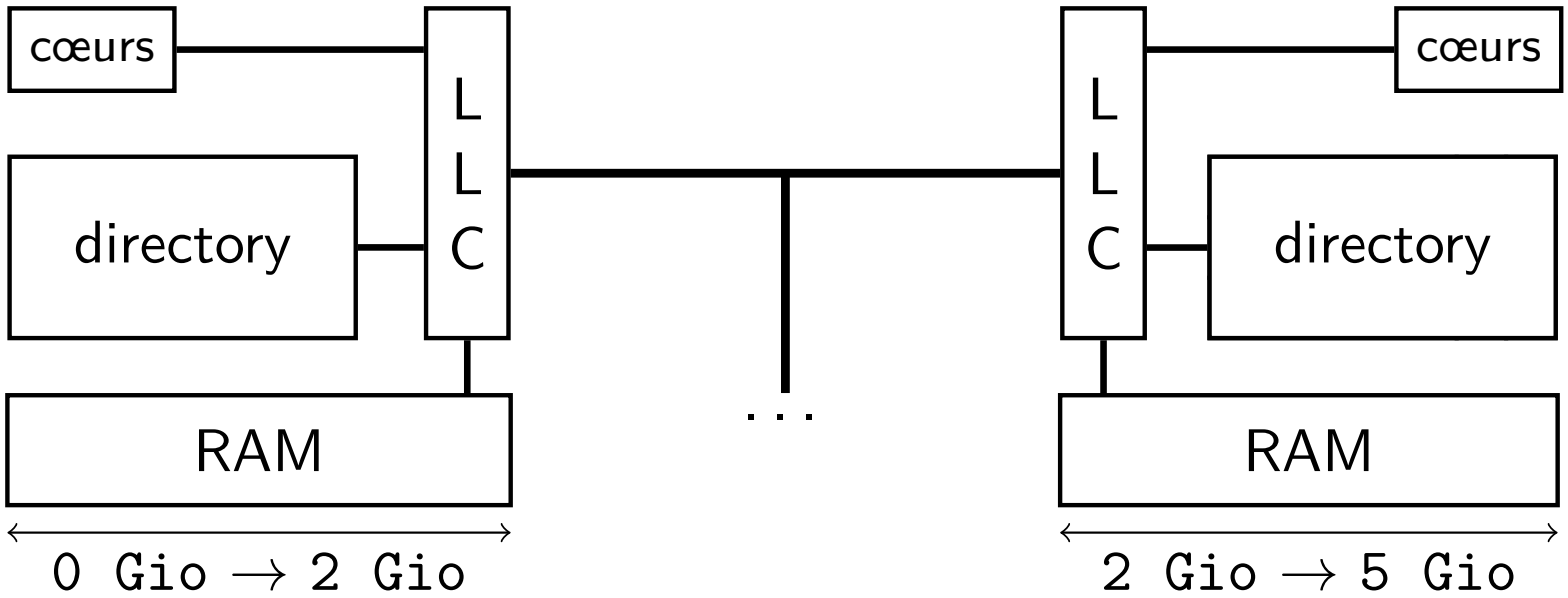
- Un protocole de cohérence de cache assure la cohérence séquentielle
  - Les protocoles classiques (MESI, MOESI) utilise un RW-lock distribué
  - Le RW-lock est implémenté en associant un état à chaque ligne
  - Les caches maintiennent les états à jour avec des broadcasts

- Chaque broadcast passe par l'interconnect
  - Ajout de latence lors de l'acquisition d'une nouvelle ligne (I $\rightarrow$ *)
  - Ajout de latence lors d'une mise à jour de l'état (sauf E $\rightarrow$ M)
  - Consommation de bande passante

- Simple à implémenter $\rightarrow$ conservation du protocole existant
  - Encore présent sur les vieilles machines AMD
  - Encore présent sur les machines Intel QPI

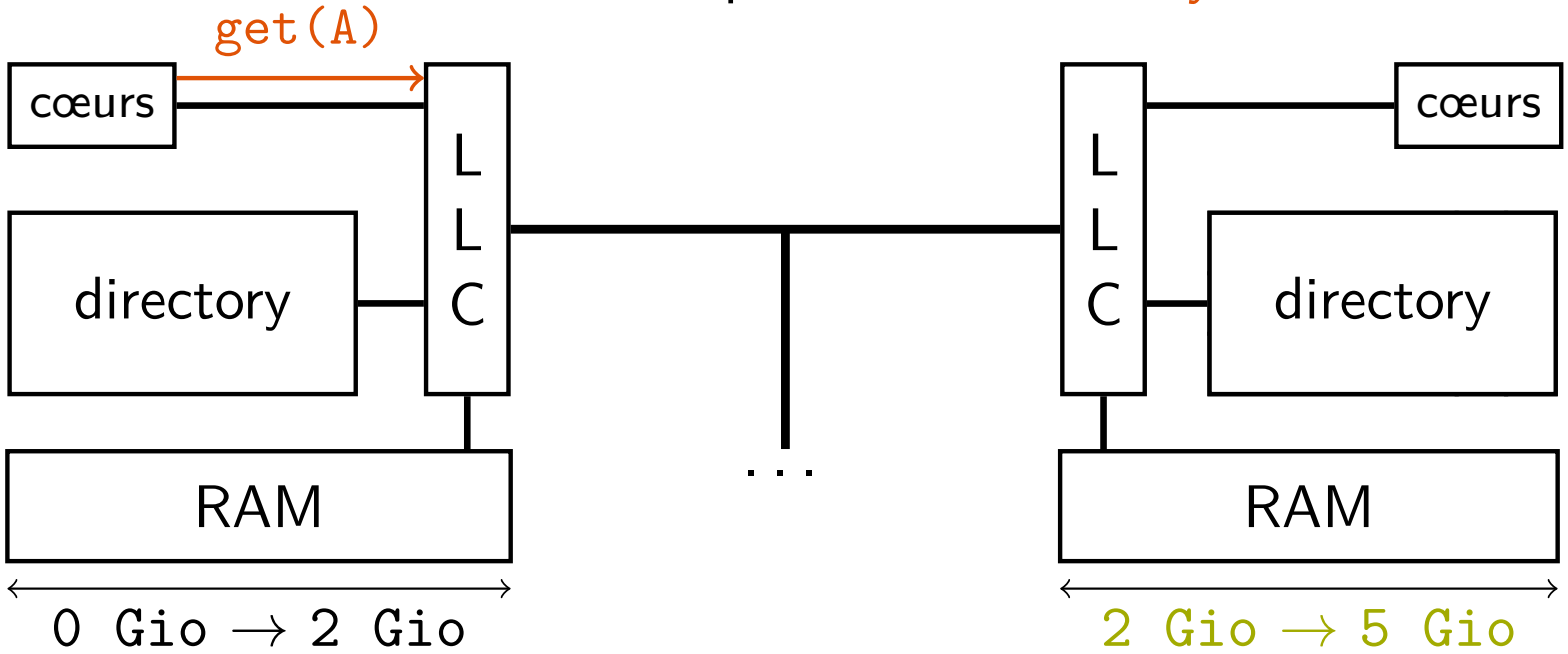
Home node et cache directory

- Pour réduire le nombre de broadcast entre les nœuds NUMA, les caches modernes utilisent des protocoles *directory based*



Home node et cache directory

- Pour réduire le nombre de broadcast entre les nœuds NUMA, les caches modernes utilisent des protocoles *directory based*

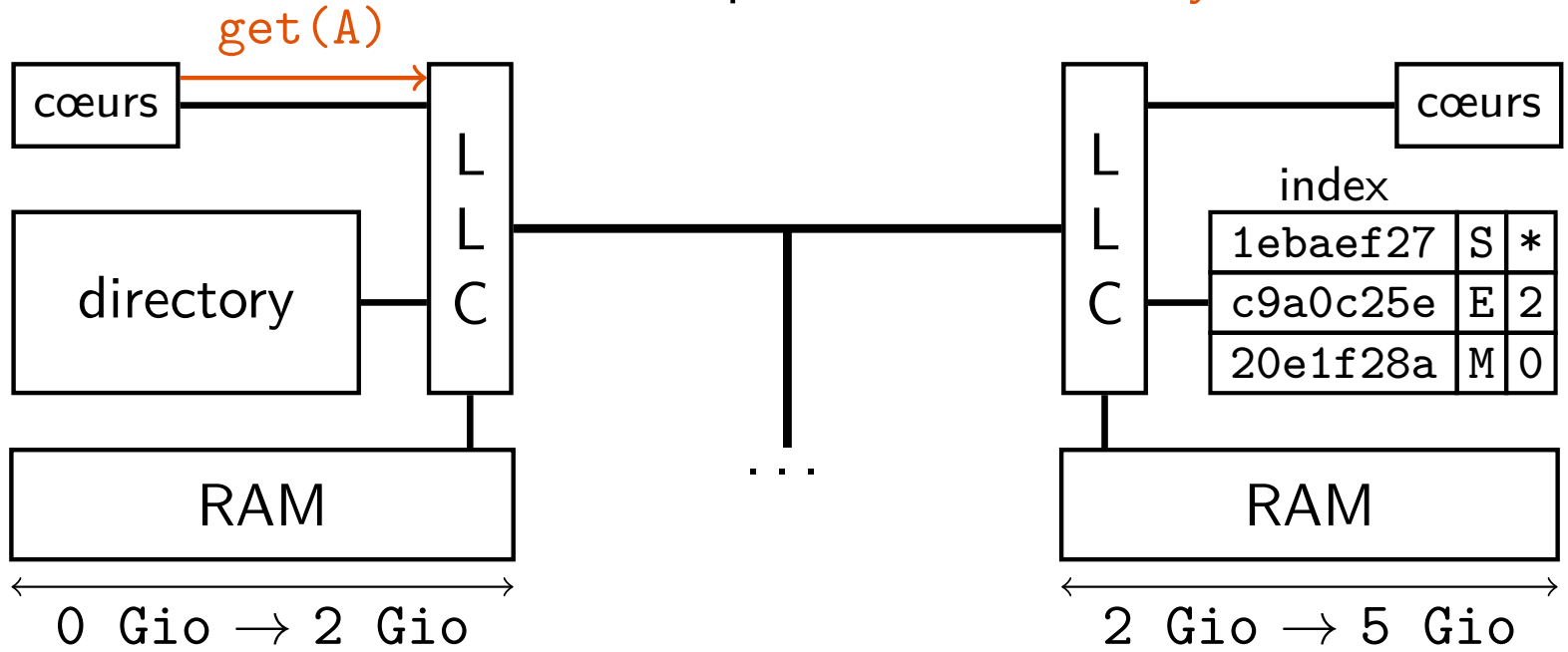


$A\varphi$ = 0x00000c9a0c25e $\simeq$ 3 Gio

- On définit, pour chaque ligne un *home node*
  - Le home node d'une ligne est déterminé par son adresse physique

Home node et cache directory

- Pour réduire le nombre de broadcast entre les nœuds NUMA, les caches modernes utilisent des protocoles *directory based*

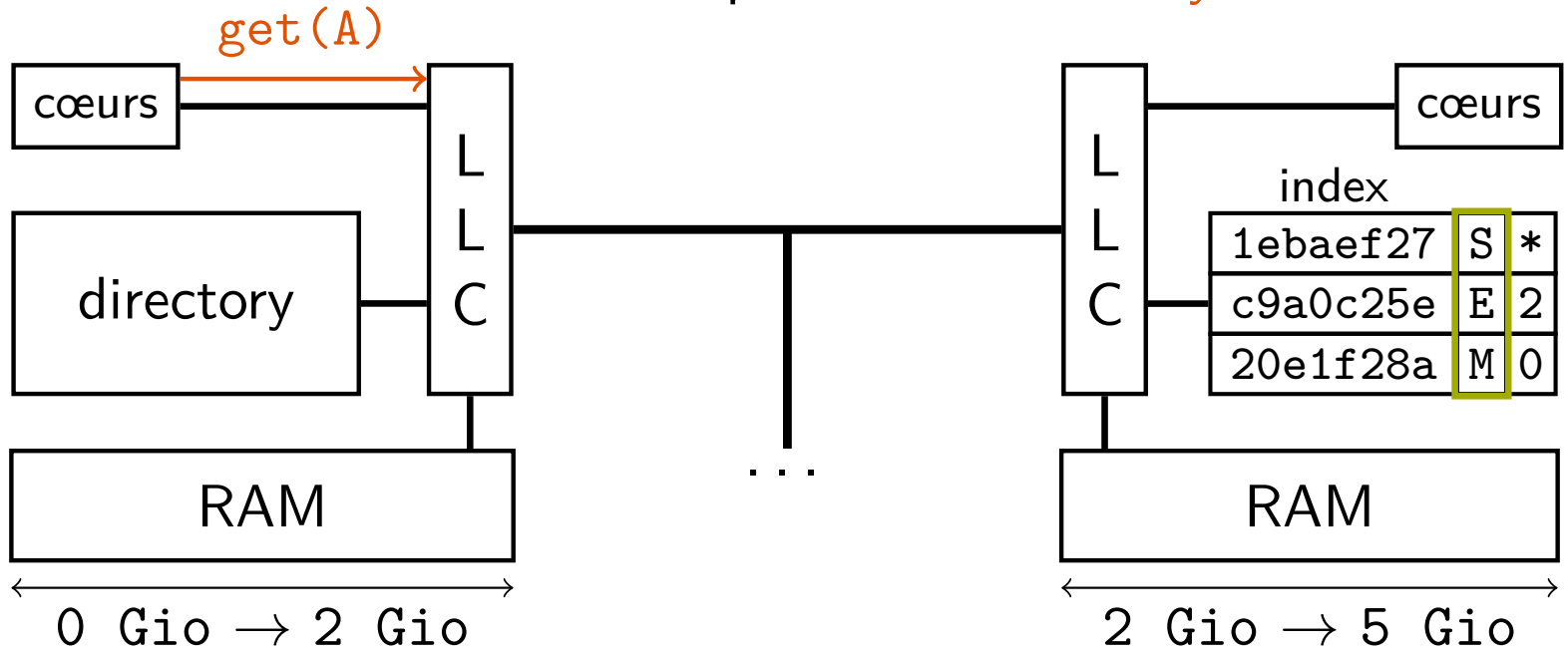


$A\varphi$ = 0x00000c9a0c25e $\simeq$ 3 Gio

- On définit, pour chaque ligne un *home node*
  - Le home node d'une ligne est déterminé par son adresse physique
- Le *home node* stocke pour chacune des lignes associées

Home node et cache directory

- Pour réduire le nombre de broadcast entre les nœuds NUMA, les caches modernes utilisent des protocoles *directory based*

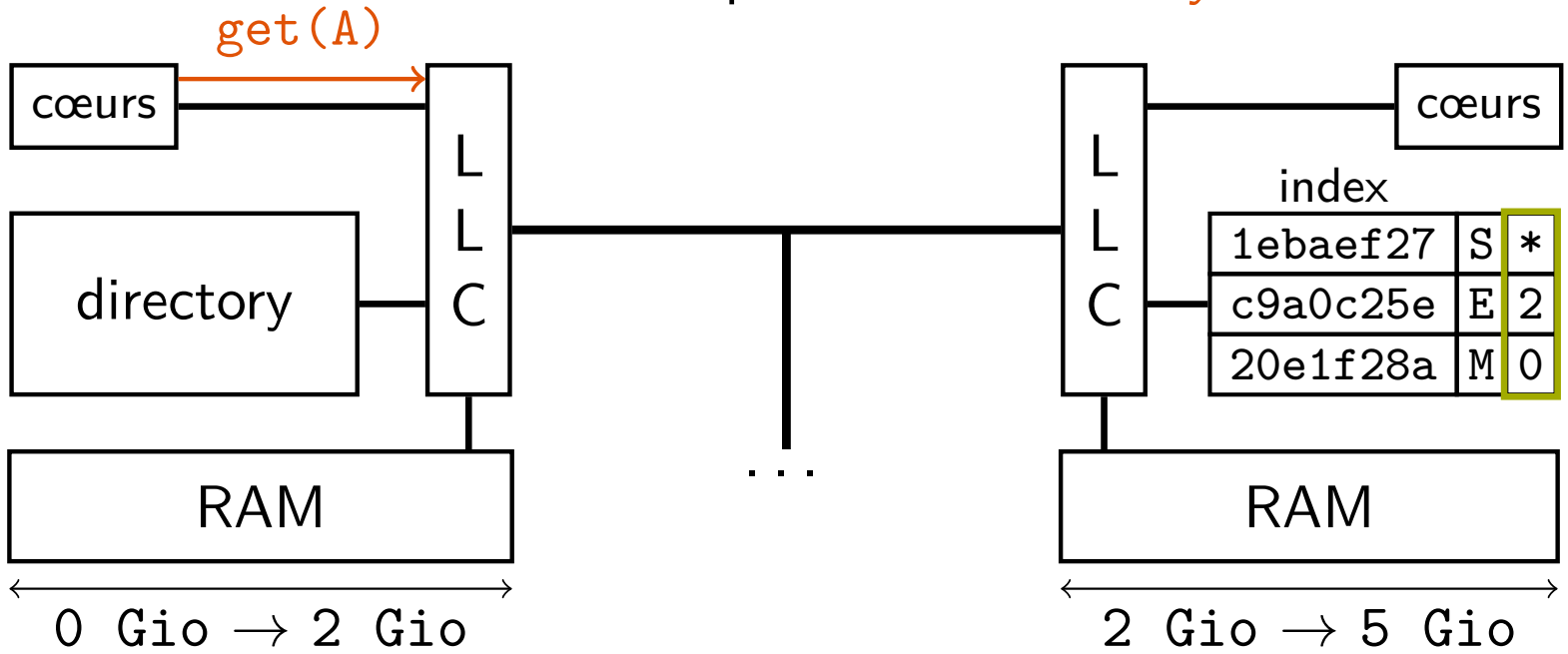


$A\varphi$ = 0x00000c9a0c25e $\simeq$ 3 Gio

- On définit, pour chaque ligne un *home node*
  - Le home node d'une ligne est déterminé par son adresse physique
- Le *home node* stocke pour chacune des lignes associées
  - L'état de la ligne → MESI, MOESI, états supplémentaires

Home node et cache directory

- Pour réduire le nombre de broadcast entre les nœuds NUMA, les caches modernes utilisent des protocoles *directory based*

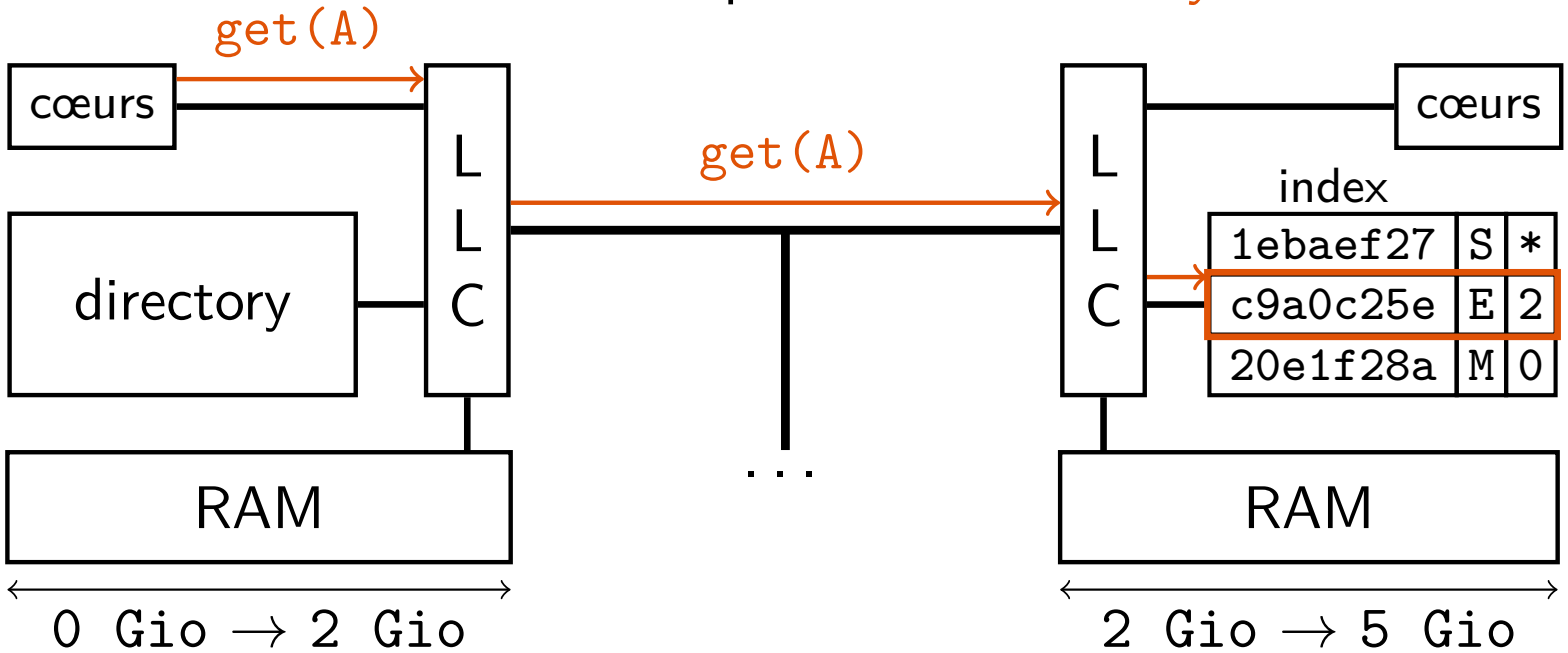


$A\varphi$ = 0x00000c9a0c25e $\simeq$ 3 Gio

- On définit, pour chaque ligne un *home node*
  - Le home node d'une ligne est déterminé par son adresse physique
- Le *home node* stocke pour chacune des lignes associées
  - L'état de la ligne → MESI, MOESI, états supplémentaires
  - Les nœuds propriétaires → qui détient la ligne en cache

Home node et cache directory

- Pour réduire le nombre de broadcast entre les nœuds NUMA, les caches modernes utilisent des protocoles *directory based*

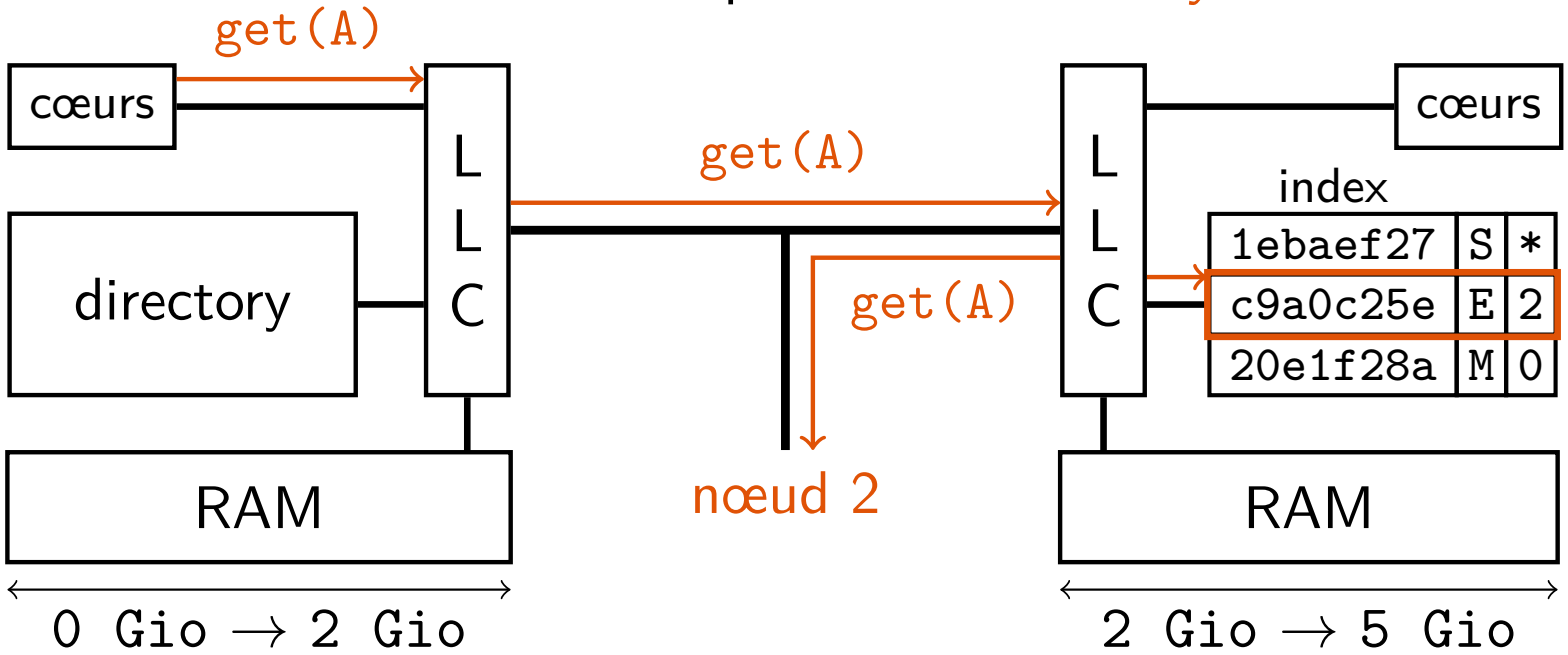


$A\varphi$ = 0x00000c9a0c25e $\simeq$ 3 Gio

- On définit, pour chaque ligne un *home node*
  - Le home node d'une ligne est déterminé par son adresse physique
- Le *home node* stocke pour chacune des lignes associées
  - L'état de la ligne → MESI, MOESI, états supplémentaires
  - Les nœuds propriétaires → qui détient la ligne en cache

Home node et cache directory

- Pour réduire le nombre de broadcast entre les nœuds NUMA, les caches modernes utilisent des protocoles *directory based*

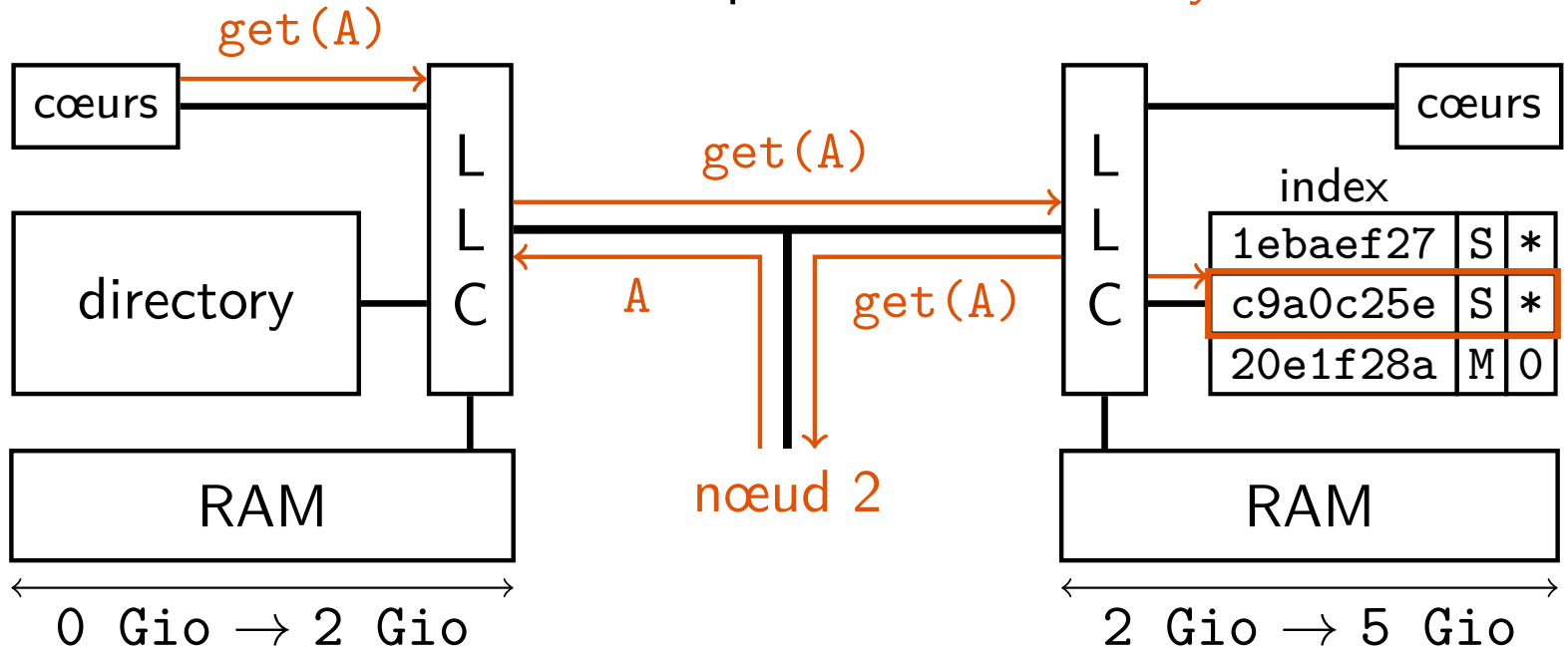


$A\varphi$ = 0x00000c9a0c25e $\simeq$ 3 Gio

- On définit, pour chaque ligne un *home node*
  - Le home node d'une ligne est déterminé par son adresse physique
- Le *home node* stocke pour chacune des lignes associées
  - L'état de la ligne → MESI, MOESI, états supplémentaires
  - Les nœuds propriétaires → qui détient la ligne en cache

Home node et cache directory

- Pour réduire le nombre de broadcast entre les nœuds NUMA, les caches modernes utilisent des protocoles *directory based*



$A\varphi$ = 0x00000c9a0c25e $\simeq$ 3 Gio

- On définit, pour chaque ligne un *home node*
  - Le home node d'une ligne est déterminé par son adresse physique
- Le *home node* stocke pour chacune des lignes associées
  - L'état de la ligne → MESI, MOESI, états supplémentaires
  - Les nœuds propriétaires → qui détient la ligne en cache

Home node et cache directory : complément

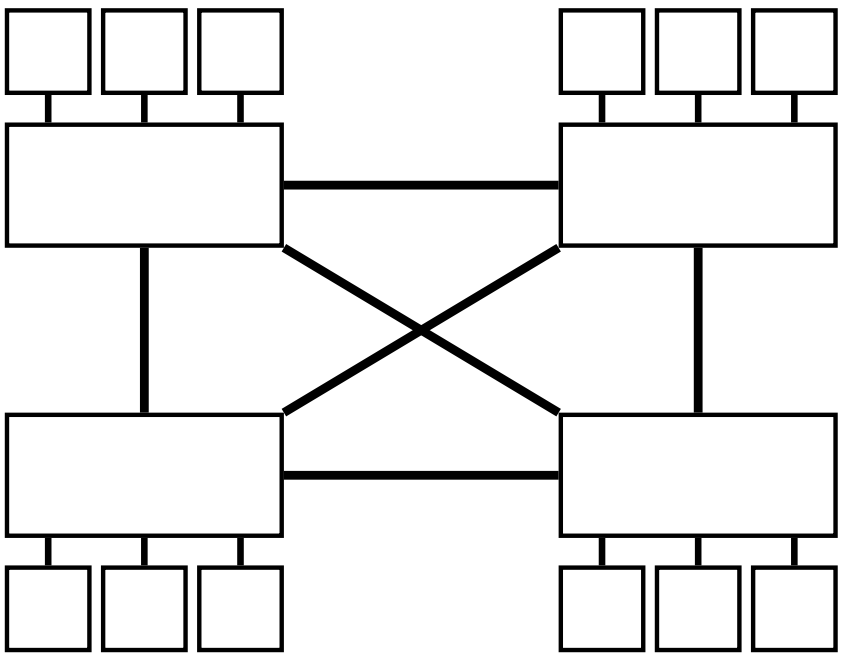
- Un protocole *directory based* réduit le nombre de broadcast entre les différents nœuds NUMA
  - Libère de la bande passante sur l'interconnect
  - Ne change rien aux broadcasts à l'intérieur d'un nœud NUMA
  - Le broadcast entre les nœuds reste utile (exemple : S $\rightarrow$ M)

- Un protocole *directory based* augmente le nombre de *hop* nécessaire à l'obtention d'une nouvelle ligne
  - Si la donnée n'est pas présente dans les caches locaux, il faut toujours passer par le home node
  - Ajoute de la latence pour la communication entre nœuds NUMA

- Plus complexe à implémenter mais généralement plus performant
  - Présent sur les machines AMD récentes (HT-Assist 3)
  - Présent sur les machines Intel récentes (Intel UPI)

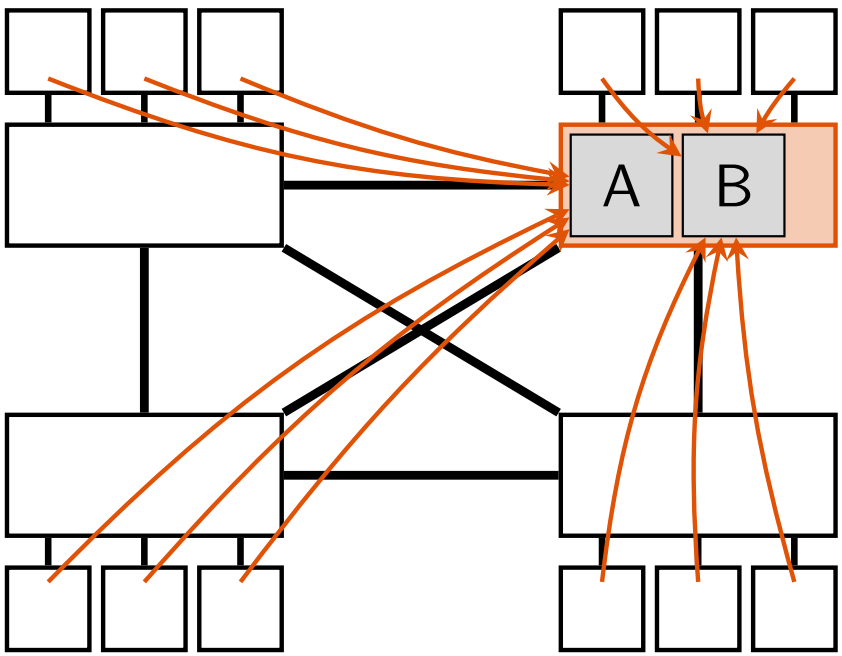
Architecture *Non Uniform Memory Access* et performance

- Dans une architecture NUMA, la performance du logiciel dépend du placement des tâches des données en mémoire physique



Architecture *Non Uniform Memory Access* et performance

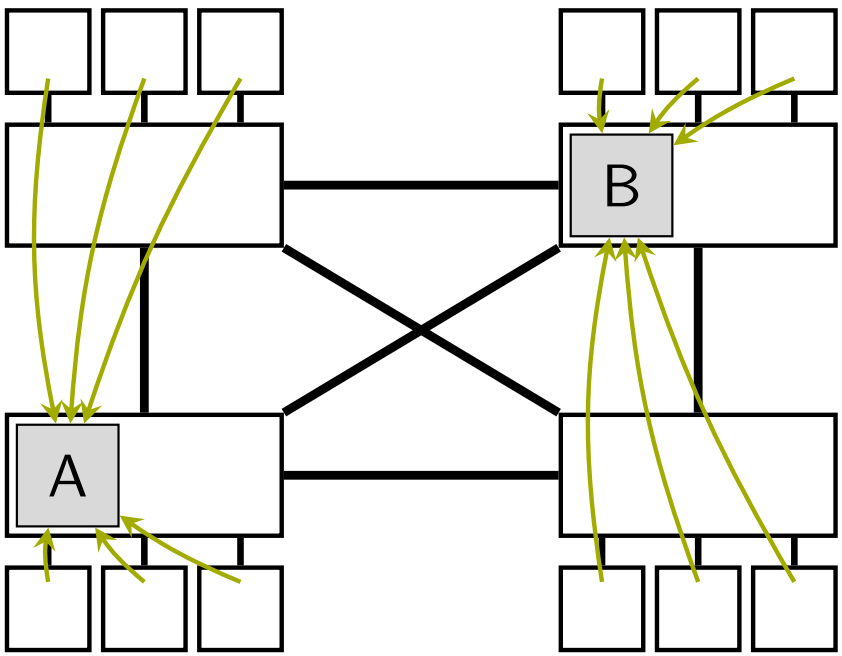
- Dans une architecture NUMA, la performance du logiciel dépend du placement des tâches des données en mémoire physique
  - Répartition de la charge mémoire



- Si tous les cœurs accèdent au même nœud
  - Saturation d'une mémoire principale
  - Problème de débit mémoire
  - Effondrement des performances

Architecture *Non Uniform Memory Access* et performance

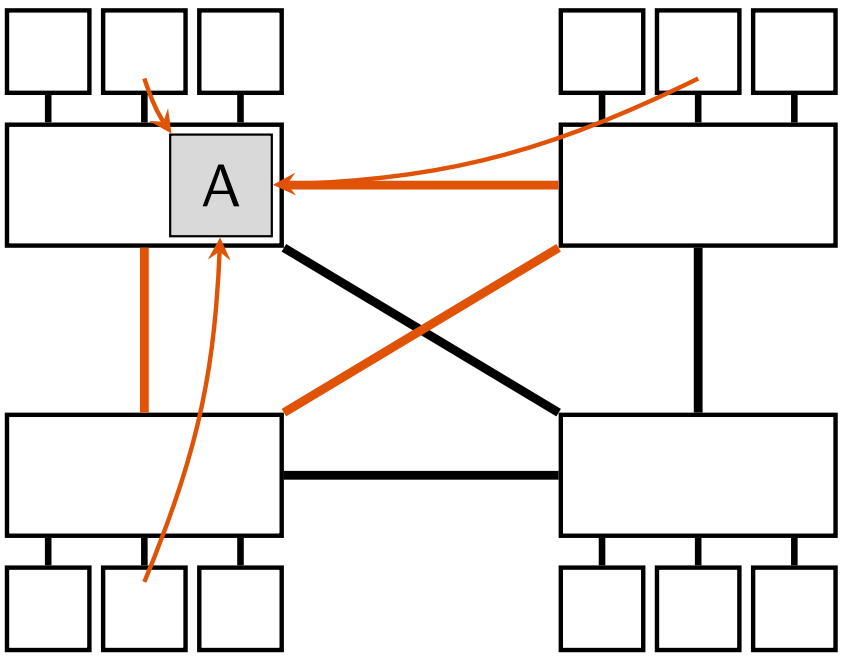
- Dans une architecture NUMA, la performance du logiciel dépend du placement des tâches des données en mémoire physique
  - Répartition de la charge mémoire



- Si tous les cœurs accèdent au même nœud
  - Saturation d'une mémoire principale
  - Problème de débit mémoire
  - Effondrement des performances
- Répartir les données dans différents nœuds
  - Changer l'emplacement physique des données

Architecture *Non Uniform Memory Access* et performance

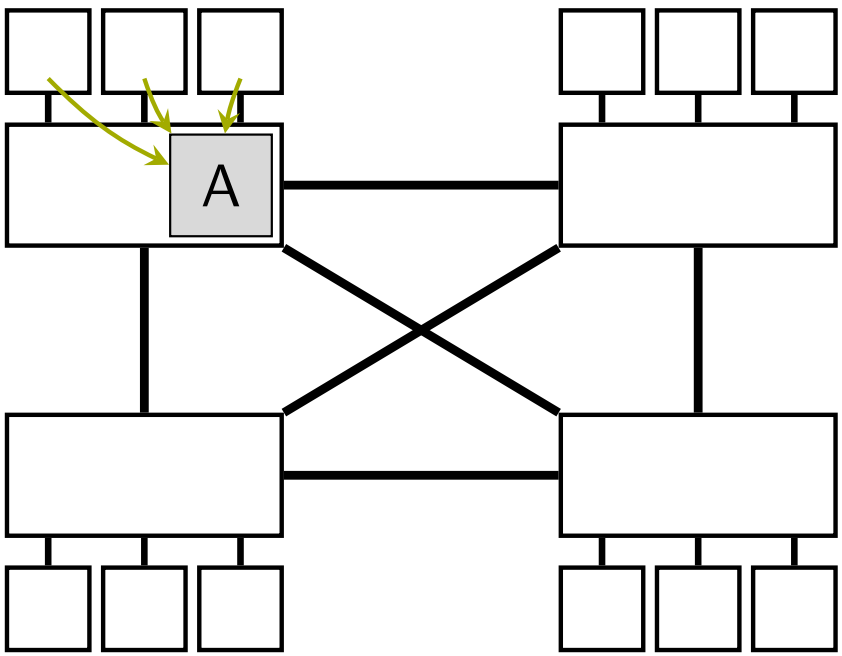
- Dans une architecture NUMA, la performance du logiciel dépend du placement des tâches des données en mémoire physique
  - Répartition de la charge mémoire
  - Limitation des transferts entre nœuds



- Transfert de ligne entre nœud coûteux
  - Latence d'accès mémoire cache supplémentaire
  - Consommation de bande passante interconnect
  - Effondrement des performances

Architecture *Non Uniform Memory Access* et performance

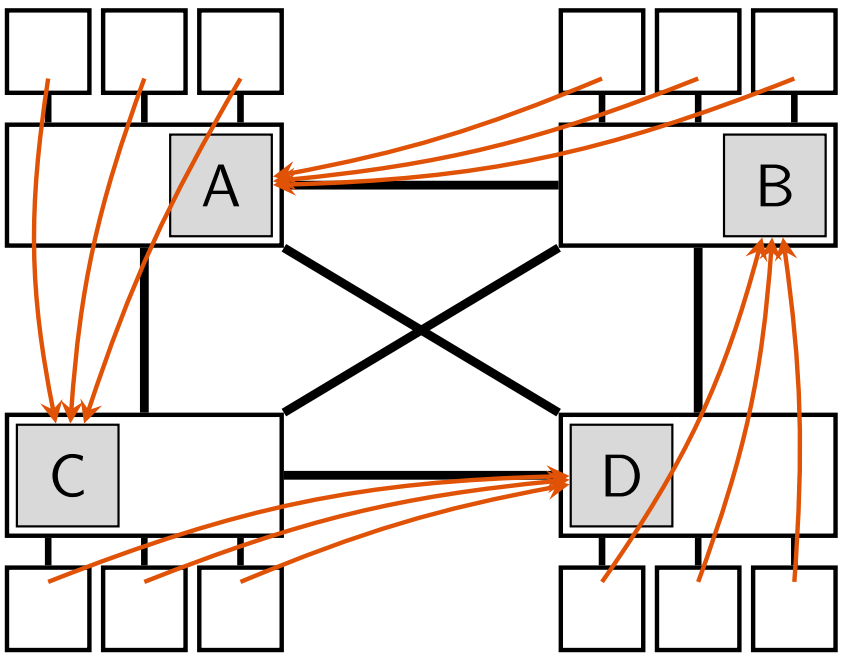
- Dans une architecture NUMA, la performance du logiciel dépend du placement des tâches des données en mémoire physique
  - Répartition de la charge mémoire
  - Limitation des transferts entre nœuds



- Transfert de ligne entre nœud coûteux
  - Latence d'accès mémoire cache supplémentaire
  - Consommation de bande passante interconnect
  - Effondrement des performances
- Communiquer localement
  - Identifier les tâches qui communiquent (données partagées)
  - Migrer ces tâches sur un même nœud

Architecture *Non Uniform Memory Access* et performance

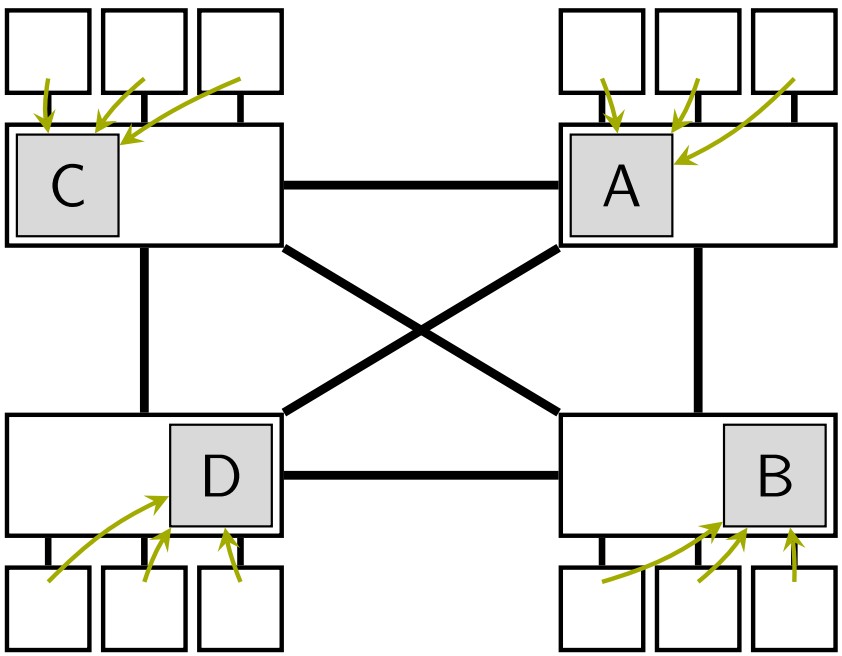
- Dans une architecture NUMA, la performance du logiciel dépend du placement des tâches des données en mémoire physique
  - Répartition de la charge mémoire
  - Limitation des transferts entre nœuds
  - Localité des accès mémoire



- Accès distant plus lent qu'accès local
  - Latence RAM supplémentaire
  - Diminution des performances

Architecture *Non Uniform Memory Access* et performance

- Dans une architecture NUMA, la performance du logiciel dépend du placement des tâches des données en mémoire physique
  - Répartition de la charge mémoire
  - Limitation des transferts entre nœuds
  - Localité des accès mémoire



- Accès distant plus lent qu'accès local
  - Latence RAM supplémentaire
  - Diminution des performances
- Accéder aux données locales
  - Changer l'emplacement des données
  - Migrer les tâches vers les données

Placement de ressources et performance

- Les performances dépendent du placement des tâches et des données entre elles et par rapport à la topologie mémoire

Placement de ressources et performance

- Les performances dépendent du placement des tâches et des données entre elles et par rapport à la topologie mémoire

- Créer les nouvelles tâches sur le bon nœud (au moment du `clone`)
  - Pas encore d'informations sur la nouvelle tâche
- Migrer périodiquement les tâches sur les bons nœuds
  - Changement de cache $\rightarrow$ perte de localité

Placement de ressources et performance

- Les performances dépendent du placement des tâches et des données entre elles et par rapport à la topologie mémoire

- Créer les nouvelles tâches sur le bon nœud (au moment du `clone`)
  - Pas encore d'informations sur la nouvelle tâche
- Migrer périodiquement les tâches sur les bons nœuds
  - Changement de cache $\rightarrow$ perte de localité

- Allouer les nouvelles données sur le bon nœud (au moment du

Placement de ressources et performance

- Les performances dépendent du placement des tâches et des données entre elles et par rapport à la topologie mémoire

- Créer les nouvelles tâches sur le bon nœud (au moment du `clone`)
  - Pas encore d'informations sur la nouvelle tâche
- Migrer périodiquement les tâches sur les bons nœuds
  - Changement de cache $\rightarrow$ perte de localité

- Allouer les nouvelles données sur le bon nœud (au moment du *first touch*)
  - Pas encore d'informations sur la nouvelle donnée
- Migrer périodiquement les données sur les bons nœuds
  - Coût du `memcpy` $\rightarrow$ bande passante, pollution de cache

Placement de ressources et performance

- Les performances dépendent du placement des tâches et des données entre elles et par rapport à la topologie mémoire

- Créer les nouvelles tâches sur le bon nœud (au moment du `clone`)
  - Pas encore d'informations sur la nouvelle tâche
- Migrer périodiquement les tâches sur les bons nœuds
  - Changement de cache $\rightarrow$ perte de localité

- Allouer les nouvelles données sur le bon nœud (au moment du *first touch*)
  - Pas encore d'informations sur la nouvelle donnée
- Migrer périodiquement les données sur les bons nœuds
  - Coût du `memcpy` $\rightarrow$ bande passante, pollution de cache

Placement de données : le *first touch*

- Intuition : la tâche qui alloue un espace en mémoire est souvent la tâche qui utilise cet espace → exemple : la pile
- Principe : allouer l'espace mémoire sur le nœud allocateur → le nœud qui touche l'espace mémoire en premier

```
void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

        if (!tree_contains(vaddr))
                segfault();

        current = current_node();
        paddr = allocate_page_on_node(current);
        map_page(vaddr, paddr);
}
```

Placement de données : le *first touch*

- Intuition : la tâche qui alloue un espace en mémoire est souvent la tâche qui utilise cet espace → exemple : la pile
- Principe : allouer l'espace mémoire sur le nœud allocateur → le nœud qui touche l'espace mémoire en premier

```
void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

        if (!tree_contains(vaddr))
                segfault();

        current = current_node();
        paddr = allocate_page_on_node(current);
        map_page(vaddr, paddr);
}
```

Placement de données : le *first touch*

- Intuition : la tâche qui alloue un espace en mémoire est souvent la tâche qui utilise cet espace → exemple : la pile
- Principe : allouer l'espace mémoire sur le nœud allocateur → le nœud qui touche l'espace mémoire en premier

```
void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

        if (!tree_contains(vaddr))
                segfault();

        current = current_node();
        paddr = allocate_page_on_node(current);
        map_page(vaddr, paddr);
}
```

- Politique par défault sous Linux
  - Configuration explicite avec `numactl --localalloc`

Placement de données : l'*interleaving*

- Observation : la répartition de la charge mémoire est le facteur le plus important pour les performances sur NUMA
- Principe : répartir équitablement les données sur tous les nœuds pour équilibrer la charge

```
void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

        if (!tree_contains(vaddr))
                segfault();

        random = choose_random_node();
        paddr = allocate_page_on_node(random);
        map_page(vaddr, paddr);
}
```

Placement de données : l'*interleaving*

- Observation : la répartition de la charge mémoire est le facteur le plus important pour les performances sur NUMA
- Principe : répartir équitablement les données sur tous les nœuds pour équilibrer la charge

```
void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

        if (!tree_contains(vaddr))
                segfault();

        random = choose_random_node();
        paddr = allocate_page_on_node(random);
        map_page(vaddr, paddr);
}
```

Placement de données : l'*interleaving*

- Observation : la répartition de la charge mémoire est le facteur le plus important pour les performances sur NUMA
- Principe : répartir équitablement les données sur tous les nœuds pour équilibrer la charge

```
void handle_page_fault(void *vaddr)
{
        paddr_t paddr;

        if (!tree_contains(vaddr))
                segfault();

        random = choose_random_node();
        paddr = allocate_page_on_node(random);
        map_page(vaddr, paddr);
}
```

- Politique la moins risquée
  - Configuration avec `numactl --interleave`

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    initialize_area(area);
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    initialize_area(area);
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    initialize_area(area);
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    initialize_area(area);
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    initialize_area(area);
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    initialize_area(area);
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    initialize_area(area);
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

- Exécution avec *first touch* :
- Exécution avec *interleaving* :

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    initialize_area(area);
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

- Exécution avec *first touch* : rapide
- Exécution avec *interleaving* : lente → mauvaise localité

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    initialize_area(area);

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    initialize_area(area);

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    initialize_area(area);

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    initialize_area(area);

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    initialize_area(area);

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    initialize_area(area);

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    initialize_area(area);

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

- Temps d'exécution avec *first touch* :
- Temps d'exécution avec *interleaving* :

Effet du placement mémoire : exercice

```
void worker(void *area)
{
    do_some_stuff(area);
}

void main(void)
{
    char *area = mmap(NULL, TASK * LEN, ...);
    int i;

    initialize_area(area);

    for (i = 0; i < TASK; i++)
        launch_worker(worker, area + i * LEN);

    wait_workers();
    use_result(area, TASK * LEN);
}
```

- Temps d'exécution avec *first touch* : très lente → contention mémoire
- Temps d'exécution avec *interleaving* : rapide

Stratégies d'allocation mémoire : résumé

- Pour s'exécuter efficacement sur une architecture NUMA, le logiciel doit maximiser plusieurs critères
  - Une bonne répartition de la charge mémoire
  - La limitation des transferts entre nœuds NUMA
  - Une bonne localité des accès mémoire

- Le logiciel peut agir sur le placement des données et des tâches $\rightarrow$ ici, on s'intéresse uniquement aux données
  - Placement initial des données $\rightarrow$ sur quel nœud allouer la mémoire
  - Migration des données après allocation $\rightarrow$ pas vu dans ce cours

- Stratégie d'allocation *first touch*
  - Allouer sur le même nœud que la tâche qui alloue
  - Performant quand chaque tâche alloue sa mémoire

- Stratégie d'allocation *interleaving*
  - Répartir équitablement les données entre les nœuds
  - Évite la contention mémoire $\rightarrow$ stratégie du moindre mal

Conclusion : multicœur et cohérence de caches

- Dans une architecture multicœur, chaque cœur a son propre cache
  - Du point de vue du logiciel, la mémoire doit garantir la cohérence séquentielle des accès
  - Les évictions de lignes de cache sont imprévisibles $\rightarrow$ incohérence séquentielle
  - Pour garantir la cohérence, le matériel assure que tout cache fournit toujours la dernière version connue de chaque ligne
  - Les caches maintiennent un RW-lock sur chaque ligne grâce à un protocole de cohérence de cache $\rightarrow$ MESI, MOESI

Conclusion : architectures *Non Uniform Memory Access*

- Quand le nombre de cœur augmente, la mémoire ne peut plus servir rapidement toutes les requêtes mémoire
  - Problème de débit d'accès à la mémoire principale
  - Dans les architectures NUMA, la mémoire principale est fragmentée en unités indépendantes
  - Une unité indépendante et les cœurs associés s'appelle un nœud NUMA
  - Le matériel présente un espace d'adressage physique unifié au logiciel

- Les protocoles de cohérence de cache classiques génèrent beaucoup de traffic quand le nombre de cœur est grand
  - Les protocoles modernes sont basés sur des *cache directories*
  - Chaque ligne a un *home node* associé $\rightarrow$ le *home node* indique quel nœud contacter pour obtenir le verrou d'une ligne

- La performance d'un logiciel sur un matériel NUMA dépend du placement des tâches et des données
  - Linux utilise des stratégies d'allocation pour placer les données
  - *First touch* : allouer au même endroit que la tâche qui alloue
  - *Interleaving* : allouer au hasard $\rightarrow$ équilibrage de charge