

# TECHNIQUES MODERNES DE PROGRAMMATION CONCURRENTE

## COURS 2

### ALGORITHMES DE VERROUILLAGE

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

# LIMITE À L'ACCELÉRATION

- Limite : la loi d'Amdahl

# LIMITE À L'ACCELÉRATION

- Limite : la loi d'Amdahl
- $p$  : pourcentage du code exécutable en parallèle
  - $(1 - p)$  : pourcentage du code exécuté en séquentiel
  - $p / n$  : exécution du code parallèle sur  $n$  cœurs
  - Temps d'exécution :  $(1 - p) + p / n$

# LIMITE À L'ACCELÉRATION

- Limite : la loi d'Amdahl
- $p$  : pourcentage du code exécutable en parallèle
  - $(1 - p)$  : pourcentage du code exécuté en séquentiel
  - $p / n$  : exécution du code parallèle sur  $n$  cœurs
  - Temps d'exécution :  $(1 - p) + p / n$
- Accélération maximale théorique :  $a = 1 / (1 - p + p/n)$ 
  - Limite pour  $n \rightarrow \infty$  :  $a \rightarrow 1 / (1 - p)$

# LIMITE À L'ACCELÉRATION

- Limite : la loi d'Amdahl
- $p$  : pourcentage du code exécutable en parallèle
  - $(1 - p)$  : pourcentage du code exécuté en séquentiel
  - $p / n$  : exécution du code parallèle sur  $n$  cœurs
  - Temps d'exécution :  $(1 - p) + p / n$
- Accélération maximale théorique :  $a = 1 / (1 - p + p/n)$ 
  - Limite pour  $n \rightarrow \infty$  :  $a \rightarrow 1 / (1 - p)$
- Application numérique :
  - $p = 0,25 \Rightarrow a \rightarrow 1 / 0,75 = 4$  quand  $n \rightarrow \infty$  (3,7 à 32 cœurs)
  - $p = 0,95 \Rightarrow a \rightarrow 1 / 0,05 = 20$  quand  $n \rightarrow \infty$  (12,55 à 32 cœurs, 17,42 à 128 cœurs)
  - $p = 0,96 \Rightarrow a \rightarrow 1 / 0,04 = 25$  quand  $n \rightarrow \infty$  (14,28 à 32 cœurs : +12.1%, 21,05 à 128 cœurs : +17.2% !)

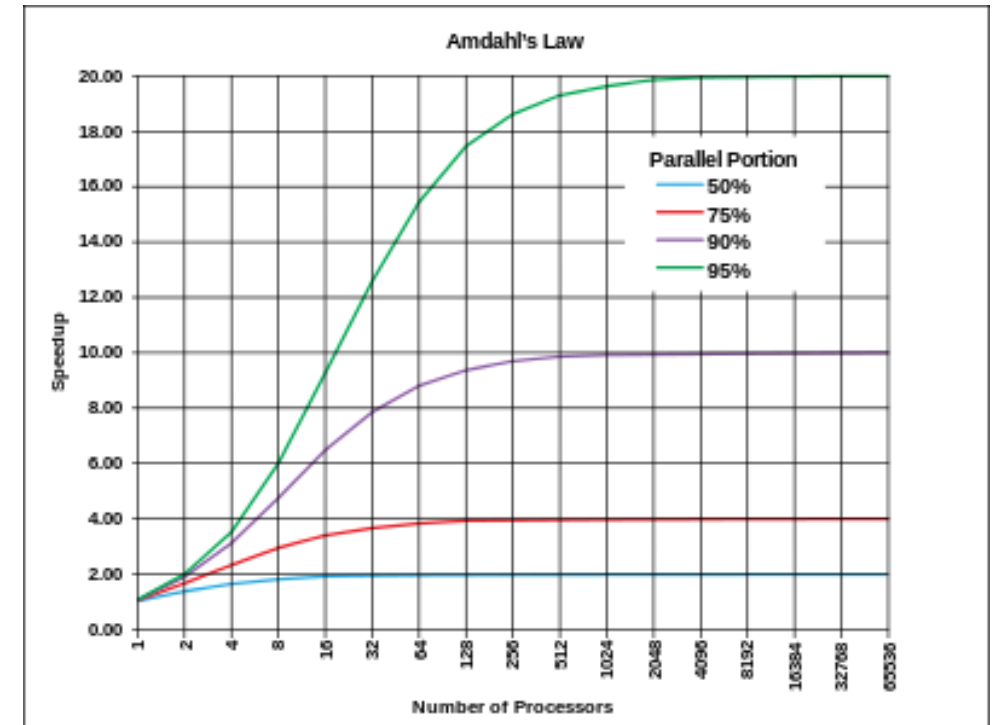
# LIMITE À L'ACCELÉRATION

- Limite : la loi d'Amdahl
- $p$  : pourcentage du code exécutable en parallèle
  - $(1 - p)$  : pourcentage du code exécuté en séquentiel
  - $p / n$  : exécution du code parallèle sur  $n$  cœurs
  - Temps d'exécution :  $(1 - p) + p / n$
- Accélération maximale théorique :  $a = 1 / (1 - p + p/n)$ 
  - Limite pour  $n \rightarrow \infty$  :  $a \rightarrow 1 / (1 - p)$
- Application numérique :
  - $p = 0,25 \Rightarrow a \rightarrow 1 / 0,75 = 4$  quand  $n \rightarrow \infty$  (3,7 à 32 cœurs)
  - $p = 0,95 \Rightarrow a \rightarrow 1 / 0,05 = 20$  quand  $n \rightarrow \infty$  (12,55 à 32 cœurs, 17,42 à 128 cœurs)
  - $p = 0,96 \Rightarrow a \rightarrow 1 / 0,04 = 25$  quand  $n \rightarrow \infty$  (14,28 à 32 cœurs : +12.1%, 21,05 à 128 cœurs : +17.2% !)

⇒ Ça vaut le coup de se battre pour paralléliser les quelques pourcents restants!

# LIMITE À L'ACCELÉRATION

- Limite : la loi d'Amdahl
- $p$  : pourcentage du code exécutable en parallèle
  - $(1 - p)$  : pourcentage du code exécuté en séquentiel
  - $p / n$  : exécution du code parallèle sur  $n$  cœurs
  - Temps d'exécution :  $(1 - p) + p / n$
- Accélération maximale théorique :  $a = 1 / (1 - p + p/n)$ 
  - Limite pour  $n \rightarrow \infty$  :  $a \rightarrow 1 / (1 - p)$
- Application numérique :
  - $p = 0,25 \Rightarrow a \rightarrow 1 / 0,75 = 4$  quand  $n \rightarrow \infty$  (3,7 à 32 cœurs)
  - $p = 0,95 \Rightarrow a \rightarrow 1 / 0,05 = 20$  quand  $n \rightarrow \infty$  (12,55 à 32 cœurs, 17,42 à 128 cœurs)
  - $p = 0,96 \Rightarrow a \rightarrow 1 / 0,04 = 25$  quand  $n \rightarrow \infty$  (14,28 à 32 cœurs : +12.1%, 21,05 à 128 cœurs : +17.2% !)



⇒ Ça vaut le coup de se battre pour paralléliser les quelques pourcents restants!

# COMMENT MIEUX PARALLÉLISER

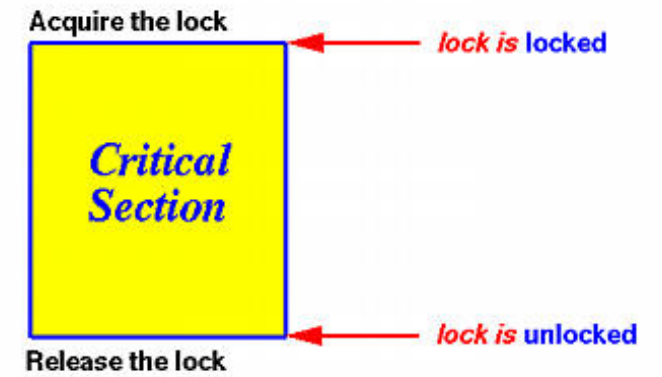
Traditionnellement, les applications parallèles se synchronisent avec des verrous



# COMMENT MIEUX PARALLÉLISER

Traditionnellement, les applications parallèles se synchronisent avec des verrous

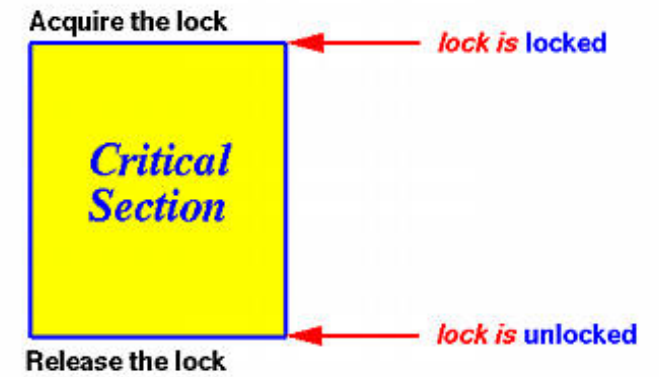
- **Option 1 : réduire la taille des sections critiques**
  - I.e., passer du *coarse-grained locking* à du *fine-grained locking*
  - En français, *verrouillage à gros grain*  $\Rightarrow$  *verrouillage à grain fin*



# COMMENT MIEUX PARALLÉLISER

Traditionnellement, les applications parallèles se synchronisent avec des verrous

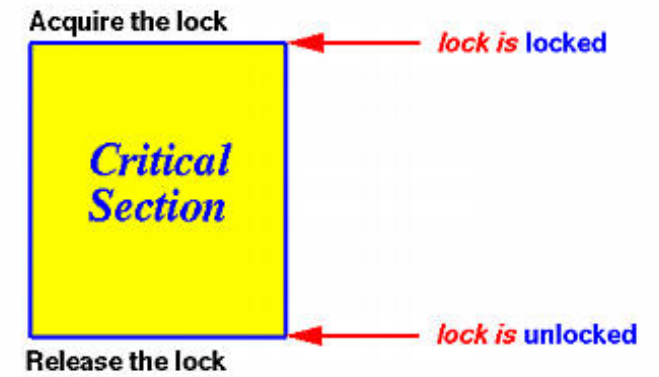
- **Option 1 : réduire la taille des sections critiques**
  - I.e., passer du *coarse-grained locking* à du *fine-grained locking*
  - En français, *verrouillage à gros grain*  $\Rightarrow$  *verrouillage à grain fin*
  - Augmente le  $p$  dans la loi d'Amdahl



# COMMENT MIEUX PARALLÉLISER

Traditionnellement, les applications parallèles se synchronisent avec des verrous

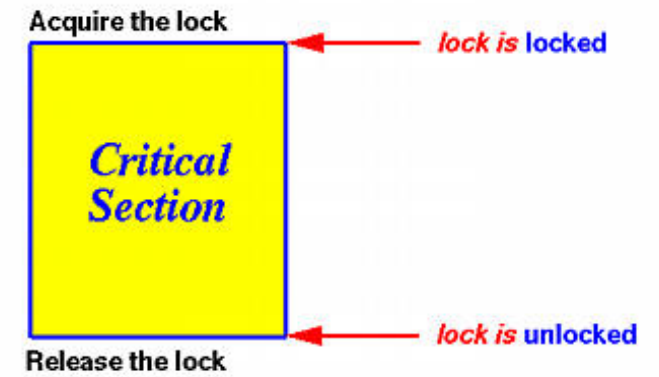
- **Option 1 : réduire la taille des sections critiques**
  - I.e., passer du *coarse-grained locking* à du *fine-grained locking*
  - En français, *verrouillage à gros grain*  $\Rightarrow$  *verrouillage à grain fin*
  - Augmente le  $p$  dans la loi d'Amdahl
  - Division de grosses sections critiques « simples » en de nombreuses petites sections critiques qui interagissent de manière complexe



# COMMENT MIEUX PARALLÉLISER

Traditionnellement, les applications parallèles se synchronisent avec des verrous

- **Option 1 : réduire la taille des sections critiques**
  - I.e., passer du *coarse-grained locking* à du *fine-grained locking*
  - En français, *verrouillage à gros grain*  $\Rightarrow$  *verrouillage à grain fin*
  - Augmente le  $p$  dans la loi d'Amdahl
  - Division de grosses sections critiques « simples » en de nombreuses petites sections critiques qui interagissent de manière complexe
  - **Travail nécessaire, mais complexe et pas vraiment de technique générale pour le faire...**
  - *Du coup, pas étudié directement dans ce cours, mais doit toujours être la première approche...*



# COMMENT MIEUX PARALLÉLISER

Traditionnellement, les applications parallèles se synchronisent avec des verrous

- **Option 2 : améliorer les algorithmes de verrouillage**
  - Pas tous les verrous équivalents !
  - Exemple simple : différence de réactivité entre verrous « qui dorment » et spinlocks...

# COMMENT MIEUX PARALLÉLISER

Traditionnellement, les applications parallèles se synchronisent avec des verrous

- **Option 2 : améliorer les algorithmes de verrouillage**
  - Pas tous les verrous équivalents !
  - **Exemple simple** : différence de réactivité entre verrous « qui dorment » et spinlocks...
  - Réduit aussi chemins critiques, car acquisition et relâchement de verrous sur chemin critique...

# COMMENT MIEUX PARALLÉLISER

Traditionnellement, les applications parallèles se synchronisent avec des verrous

- **Option 2 : améliorer les algorithmes de verrouillage**
  - Pas tous les verrous équivalents !
  - **Exemple simple** : différence de réactivité entre verrous « qui dorment » et spinlocks...
  - Réduit aussi chemins critiques, car acquisition et relâchement de verrous sur chemin critique...
- **Option 3 : écrire des algorithmes sans verrous, a.k.a. « non bloquants » !**
  - Possible grâce aux instructions atomiques des CPUs actuels : compare-and-swap, test-and-set...

# COMMENT MIEUX PARALLÉLISER

Traditionnellement, les applications parallèles se synchronisent avec des verrous

- **Option 2 : améliorer les algorithmes de verrouillage**
  - Pas tous les verrous équivalents !
  - **Exemple simple** : différence de réactivité entre verrous « qui dorment » et spinlocks...
  - Réduit aussi chemins critiques, car acquisition et relâchement de verrous sur chemin critique...
- **Option 3 : écrire des algorithmes sans verrous, a.k.a. « non bloquants » !**
  - Possible grâce aux instructions atomiques des CPUs actuels : compare-and-swap, test-and-set...
  - En fait, « micro » sections critiques optimisées en hard : impossible de faire plus petit !



# COMMENT MIEUX PARALLÉLISER

Traditionnellement, les applications parallèles se synchronisent avec des verrous

- **Option 2 : améliorer les algorithmes de verrouillage**
  - Pas tous les verrous équivalents !
  - **Exemple simple** : différence de réactivité entre verrous « qui dorment » et spinlocks...
  - Réduit aussi chemins critiques, car acquisition et relâchement de verrous sur chemin critique...
- **Option 3 : écrire des algorithmes sans verrous, a.k.a. « non bloquants » !**
  - Possible grâce aux instructions atomiques des CPUs actuels : compare-and-swap, test-and-set...
  - En fait, « micro » sections critiques optimisées en hard : impossible de faire plus petit !
  - Ne marche pas pour tous les algorithmes, mais si possible de trouver un algo non bloquant, idéal...

# COMMENT MIEUX PARALLÉLISER

Traditionnellement, les applications parallèles se synchronisent avec des verrous

- **Option 4 : changer complètement le modèle mémoire avec les mémoires transactionnelles**
  - **Idée** : remplacer les verrous par des transactions qu'on peut exécuter en parallèle

# COMMENT MIEUX PARALLÉLISER

Traditionnellement, les applications parallèles se synchronisent avec des verrous

- **Option 4 : changer complètement le modèle mémoire avec les mémoires transactionnelles**
  - **Idée** : remplacer les verrous par des transactions qu'on peut exécuter en parallèle
  - Si aucun conflit : commit. Si conflit : rollback ! Comme dans une base de données...

# COMMENT MIEUX PARALLÉLISER

Traditionnellement, les applications parallèles se synchronisent avec des verrous

- **Option 4 : changer complètement le modèle mémoire avec les mémoires transactionnelles**
  - **Idée** : remplacer les verrous par des transactions qu'on peut exécuter en parallèle
  - Si aucun conflit : commit. Si conflit : rollback ! Comme dans une base de données...
  - S'implémente en software, processeurs récents le gèrent partiellement en hard !

## INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS (INTEL® TSX) FASTER ACCESS TO DATA WITH THE SAP HANA® PLATFORM

Coordinates access  
to data

Threads without data  
conflicts are automatically  
executed in parallel.



**Simple and scalable**  
Data flows freely because the  
processor determines if threads  
need to serialize and performs  
serialization only when required.

# COMMENT MIEUX PARALLÉLISER

Traditionnellement, les applications parallèles se synchronisent avec des verrous

- **Option 4 : changer complètement le modèle mémoire avec les mémoires transactionnelles**
  - **Idée** : remplacer les verrous par des transactions qu'on peut exécuter en parallèle
  - Si aucun conflit : commit. Si conflit : rollback ! Comme dans une base de données...
  - S'implémente en software, processeurs récents le gèrent partiellement en hard !
  - Pas forcément plus rapide, mais plus simple...  
**Peut-être le futur ?**

**INTEL® TRANSACTIONAL SYNCHRONIZATION EXTENSIONS  
(INTEL® TSX) FASTER ACCESS TO DATA WITH THE SAP HANA® PLATFORM**

**Coordinates access  
to data**

Threads without data  
conflicts are automatically  
executed in parallel.



**Simple and scalable**  
Data flows freely because the  
processor determines if threads  
need to serialize and performs  
serialization only when required.

# OPTION 2 : AMÉLIORER LES ALGORITHMES DE VERROUILLAGE

# BLOCKING VS. SPINNING

- Deux types de verrous : mutexes ou à attente active

# BLOCKING VS. SPINNING

- Deux types de verrous : mutexes ou à attente active
- Mutexes : par défaut en général dans les bibliothèques
  - Par exemple : `pthread_mutex_lock()`





# BLOCKING VS. SPINNING

- Deux types de verrous : mutexes ou à attente active
- Mutexes : par défaut en général dans les bibliothèques
  - Par exemple : `pthread_mutex_lock()`
  - **Idée** : si un thread n'arrive pas à acquérir le verrou, il s'endort, réveil quand verrou libre



# BLOCKING VS. SPINNING

- Deux types de verrous : mutexes ou à attente active
- Mutexes : par défaut en général dans les bibliothèques
  - Par exemple : `pthread_mutex_lock()`
  - **Idée** : si un thread n'arrive pas à acquérir le verrou, il s'endort, réveil quand verrou libre
  - **Avantages** : consomme peu de ressources (pas d'attente active), marche en monocœur



# BLOCKING VS. SPINNING

- Deux types de verrous : mutexes ou à attente active
- Mutexes : par défaut en général dans les bibliothèques
  - Par exemple : `pthread_mutex_lock()`
  - **Idée** : si un thread n'arrive pas à acquérir le verrou, il s'endort, réveil quand verrou libre
  - **Avantages** : consomme peu de ressources (pas d'attente active), marche en monocœur
  - **Inconvénients** : context switch à chaque passage de verrou, gros overhead !



# BLOCKING VS. SPINNING

- Deux types de verrous : mutexes ou à attente active
- Mutexes : par défaut en général dans les bibliothèques
  - Par exemple : `pthread_mutex_lock()`
  - Idée : si un thread n'arrive pas à acquérir le verrou, il s'endort, réveil quand verrou libre
  - Avantages : consomme peu de ressources (pas d'attente active), marche en monocœur
  - Inconvénients : context switch à chaque passage de verrou, gros overhead !
- Verrous à attente active (= spinning, busy-waiting)
  - Partout dans le noyau Linux, par exemple : `spin_lock_irqsave()`



# BLOCKING VS. SPINNING

- Deux types de verrous : mutexes ou à attente active
- Mutexes : par défaut en général dans les bibliothèques
  - Par exemple : `pthread_mutex_lock()`
  - Idée : si un thread n'arrive pas à acquérir le verrou, il s'endort, réveil quand verrou libre
  - Avantages : consomme peu de ressources (pas d'attente active), marche en monocœur
  - Inconvénients : context switch à chaque passage de verrou, gros overhead !
- Verrous à attente active (= spinning, busy-waiting)
  - Partout dans le noyau Linux, par exemple : `spin_lock_irqsave()`
  - Idée : si un thread n'arrive pas à acquérir le verrou, attente active jusqu'à l'obtention



# BLOCKING VS. SPINNING

- Deux types de verrous : mutexes ou à attente active
- Mutexes : par défaut en général dans les bibliothèques
  - Par exemple : `pthread_mutex_lock()`
  - Idée : si un thread n'arrive pas à acquérir le verrou, il s'endort, réveil quand verrou libre
  - Avantages : consomme peu de ressources (pas d'attente active), marche en monocœur
  - Inconvénients : context switch à chaque passage de verrou, gros overhead !
- Verrous à attente active (= spinning, busy-waiting)
  - Partout dans le noyau Linux, par exemple : `spin_lock_irqsave()`
  - Idée : si un thread n'arrive pas à acquérir le verrou, attente active jusqu'à l'obtention
  - Avantage : très réactif ! Pour le code noyau et applications qui veulent exploiter au max les cœurs



# BLOCKING VS. SPINNING

- Deux types de verrous : mutexes ou à attente active
- Mutexes : par défaut en général dans les bibliothèques
  - Par exemple : `pthread_mutex_lock()`
  - Idée : si un thread n'arrive pas à acquérir le verrou, il s'endort, réveil quand verrou libre
  - Avantages : consomme peu de ressources (pas d'attente active), marche en monocœur
  - Inconvénients : context switch à chaque passage de verrou, gros overhead !
- Verrous à attente active (= spinning, busy-waiting)
  - Partout dans le noyau Linux, par exemple : `spin_lock_irqsave()`
  - Idée : si un thread n'arrive pas à acquérir le verrou, attente active jusqu'à l'obtention
  - Avantage : très réactif ! Pour le code noyau et applications qui veulent exploiter au max les cœurs
  - Inconvénients : dépensier en cycles/énergie (attente active), ne marche pas en monocœur



# BLOCKING VS. SPINNING

- Verrous hybrides?
  - **Spinning marche bien quand  $\#threads \leq \#cœurs$  (1 thread/cœur perdre cycles sans importance)**





# BLOCKING VS. SPINNING

- Verrous hybrides?
  - **Spinning marche bien quand  $\#threads \leq \#cœurs$  (1 thread/cœur perdre cycles sans importance)**
  - Si on veut économiser des ressources (cycles + énergie) : spinning un moment, puis on dort
    - Combien de temps ? Délai variable ? **Linear/exponential backoff** ? À voir...



# BLOCKING VS. SPINNING



- Verrous hybrides?
  - **Spinning marche bien quand  $\#threads \leq \#cœurs$  (1 thread/cœur perdre cycles sans importance)**
  - Si on veut économiser des ressources (cycles + énergie) : spinning un moment, puis on dort
    - Combien de temps ? Délai variable ? **Linear/exponential backoff** ? À voir...
  - Autre possibilité pour l'énergie : **monitor/mwait** au lieu de **spinner**
    - Instructions qui endorment et réveillent un cœur rapidement... Inutilisé mais pas de context switch

# BLOCKING VS. SPINNING



- Verrous hybrides?
  - **Spinning marche bien quand  $\#threads \leq \#cœurs$  (1 thread/cœur perdre cycles sans importance)**
  - Si on veut économiser des ressources (cycles + énergie) : spinning un moment, puis on dort
    - Combien de temps ? Délai variable ? **Linear/exponential backoff** ? À voir...
  - Autre possibilité pour l'énergie : **monitor/mwait** au lieu de **spinner**
    - Instructions qui endorment et réveillent un cœur rapidement... Inutilisé mais pas de context switch
  - Si on utilise des verrous à **attente active** : fall back sur mutexes si monocœur nécessaire !

# BLOCKING VS. SPINNING



- Verrous hybrides?
  - **Spinning marche bien quand  $\#threads \leq \#cœurs$  (1 thread/cœur perdre cycles sans importance)**
  - Si on veut économiser des ressources (cycles + énergie) : spinning un moment, puis on dort
    - Combien de temps ? Délai variable ? **Linear/exponential backoff** ? À voir...
  - Autre possibilité pour l'énergie : **monitor/mwait** au lieu de **spinner**
    - Instructions qui endorment et réveillent un cœur rapidement... Inutilisé mais pas de context switch
  - Si on utilise des verrous à attente active : fall back sur mutexes si monocœur nécessaire !
- **Notre but : exploiter les performances des multicœurs au maximum !**
  - On va donc explorer quelques algorithmes de verrouillages à attente active...





## INTERLUDE : OUTILS POUR L'ALGORITHMIQUE BAS NIVEAU (ALGOS DE VERROUS, ALGOS NON-BLOQUANTS)

# OUTILS POUR ALGORITHMIQUE BAS NIVEAU

- Ecrire algos de verrouillage, ou algos non-bloquants => on est à plus « bas niveau » que les verrous

# OUTILS POUR ALGORITHMIQUE BAS NIVEAU

- **Ecrire algos de verrouillage, ou algos non-bloquants => on est à plus « bas niveau » que les verrous**
- **Outils à notre disposition :**
  - Cohérence de cache, avec modèle mémoire et barrières (voir cours précédent)

# OUTILS POUR ALGORITHMIQUE BAS NIVEAU

- **Ecrire algos de verrouillage, ou algos non-bloquants => on est à plus « bas niveau » que les verrous**
- **Outils à notre disposition :**
  - Cohérence de cache, avec modèle mémoire et barrières (voir cours précédent)
  - Instructions classiques de lecture et d'écriture



# OUTILS POUR ALGORITHMIQUE BAS NIVEAU

- **Ecrire algos de verrouillage, ou algos non-bloquants => on est à plus « bas niveau » que les verrous**
- **Outils à notre disposition :**
  - Cohérence de cache, avec modèle mémoire et barrières (voir cours précédent)
  - Instructions classiques de lecture et d'écriture
  - **Appels système comme `futex()` sous Linux pour mutexes**
    - Besoin, e.g. de faire un context switch (OS!) selon la valeur d'une variable, atomiquement
    - *Utile que pour implem de mutexes/verrous hybrides... ne nous intéresse pas !*

# OUTILS POUR ALGORITHMIQUE BAS NIVEAU

- **Ecrire algos de verrouillage, ou algos non-bloquants => on est à plus « bas niveau » que les verrous**
- **Outils à notre disposition :**
  - Cohérence de cache, avec modèle mémoire et barrières (voir cours précédent)
  - Instructions classiques de lecture et d'écriture
  - **Appels système comme `futex()` sous Linux pour mutexes**
    - Besoin, e.g. de faire un context switch (OS!) selon la valeur d'une variable, atomiquement
    - *Utile que pour implem de mutexes/verrous hybrides... ne nous intéresse pas !*
  - **Instructions atomiques : permettent de combiner des opérations**
    - **Atomiquement : incrément, échange de valeurs... Section critique en hard!**

# OUTILS : RAPPELS SUR LE MODÈLE MÉMOIRE

- *Vu rapidement la semaine dernière... Mais soyons sur de partir avec les idées claires ! 😊*

# OUTILS : RAPPELS SUR LE MODÈLE MÉMOIRE

- *Vu rapidement la semaine dernière... Mais soyons sur de partir avec les idées claires ! 😊*
- Principe de base, commun à tous les modèles mémoire :
  - **En local, toute lecture succédant à une écriture lit la dernière valeur écrite**

write x, value

read x -> lit la valeur value

# OUTILS : RAPPELS SUR LE MODÈLE MÉMOIRE

- Sur x86, on a de la chance... Total Store Order (TSO) !

# OUTILS : RAPPELS SUR LE MODÈLE MÉMOIRE

- Sur x86, on a de la chance... Total Store Order (TSO) !
  - Lectures jamais réordonnancées entre elles
  - Ecritures jamais réordonnancées entre elles
  - Une écriture après une lecture pas réordonnancée

# OUTILS : RAPPELS SUR LE MODÈLE MÉMOIRE

- Sur x86, on a de la chance... Total Store Order (TSO) !
  - Lectures jamais réordonnancées entre elles
  - Ecritures jamais réordonnancées entre elles
  - Une écriture après une lecture pas réordonnancée
  - Lecture après écriture peut être réordonnancée si deux cases mémoires différentes !

# OUTILS : RAPPELS SUR LE MODÈLE MÉMOIRE

- Sur x86, on a de la chance... Total Store Order (TSO) !
  - Lectures jamais réordonnancées entre elles
  - Ecritures jamais réordonnancées entre elles
  - Une écriture après une lecture pas réordonnancée
  - **Lecture après écriture peut être réordonnancée si deux cases mémoires différentes !**

Initialement,  $x = y = 0$

**P1**

écrit  $x$ , 1

écrit  $y$ , 1

**P2**

lit  $y$

lit  $x$

Lectures non réordonnancées  
et écritures non réordonnancées

Si on lit  $y = 1$ , on lit  $x = 1$



# OUTILS : RAPPELS SUR LE MODÈLE MÉMOIRE

- Sur x86, on a de la chance... Total Store Order (TSO) !
  - Lectures jamais réordonnancées entre elles
  - Ecritures jamais réordonnancées entre elles
  - Une écriture après une lecture pas réordonnancée
  - **Lecture après écriture peut être réordonnancée si deux cases mémoires différentes !**

Initialement,  $x = y = 0$

**P1**

$x = 1$  (on écrit  $x$ )

$y = 1$  (on écrit  $y$ )

**P2**

$\text{while } (y == 0) \text{ \_mm\_pause();}$  (on lit  $y$ )  
 $\text{res} = x$  (on lit  $x$ )

Lectures non réordonnancées  
et écritures non réordonnancées

On a forcément  $\text{res} = 1$  (jamais 0)

# OUTILS : RAPPELS SUR LE MODÈLE MÉMOIRE

- Sur x86, on a de la chance... Total Store Order (TSO) !
  - Lectures jamais réordonnancées entre elles
  - Ecritures jamais réordonnancées entre elles
  - Une écriture après une lecture pas réordonnancée
  - **Lecture après écriture peut être réordonnancée si deux cases mémoires différentes !**

Initialement,  $x = y = 0$

**P1**

lit x

écrit y, 1

**P2**

lit y

écrit x, 1

*Ecritures après lectures pas réordonnancées*

**Si on lit  $y = 1$ , on lit  $x = 0$**

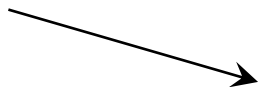
# OUTILS : RAPPELS SUR LE MODÈLE MÉMOIRE

- Sur x86, on a de la chance... Total Store Order (TSO) !
  - Lectures jamais réordonnancées entre elles
  - Ecritures jamais réordonnancées entre elles
  - Une écriture après une lecture pas réordonnancée
  - **Lecture après écriture peut être réordonnancée si deux cases mémoires différentes !**

Initialement,  $x = y = 0$

**P1**

res = x    (on lit x)  
y = 1    (on écrit y)



**P2**

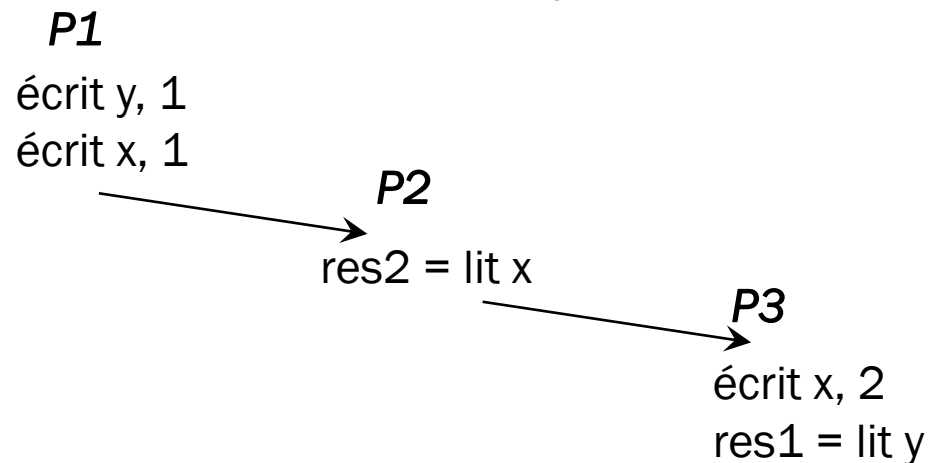
while (y == 0) \_mm\_pause(); (on lit y)  
x = 1    (on écrit x)

*Ecritures après lectures pas réordonnancées*  
**On a forcément res = 0 (jamais 1)**

# OUTILS : RAPPELS SUR LE MODÈLE MÉMOIRE

- Sur x86, on a de la chance... Total Store Order (TSO) !
  - Lectures jamais réordonnancées entre elles
  - Ecritures jamais réordonnancées entre elles
  - Une écriture après une lecture pas réordonnancée
  - **Lecture après écriture peut être réordonnancée si deux cases mémoires différentes !**

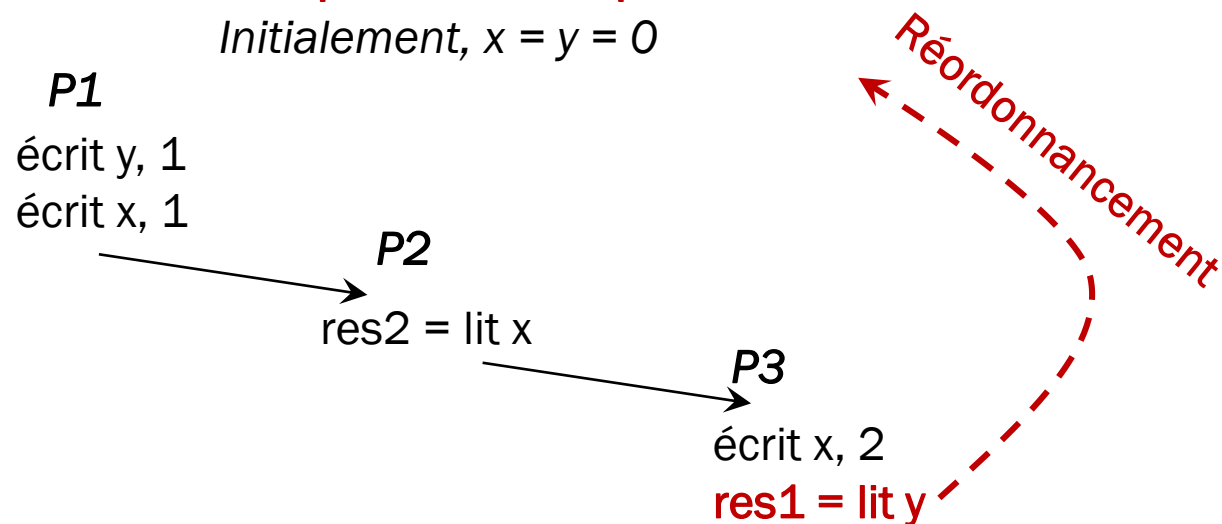
Initialement,  $x = y = 0$



*Ecritures après lectures pas réordonnancées*  
Si on lit res2 = 1, P2 après P1 et avant P3  
Du coup, res1 = 1?

# OUTILS : RAPPELS SUR LE MODÈLE MÉMOIRE

- Sur x86, on a de la chance... Total Store Order (TSO) !
  - Lectures jamais réordonnancées entre elles
  - Ecritures jamais réordonnancées entre elles
  - Une écriture après une lecture pas réordonnancée
  - **Lecture après écriture peut être réordonnancée si deux cases mémoires différentes !**



*Ecritures après lectures pas réordonnancées*

Si on lit  $res2 = 1$ , P2 après P1 et avant P3

Du coup,  $res1 = 1$ ?

**Pas assuré par TSO :  $res1$  peut valoir 0 !**

# OUTILS : RAPPELS SUR LE MODÈLE MÉMOIRE

- Sur x86, on a de la chance... Total Store Order (TSO) !
  - Lectures jamais réordonnancées entre elles
  - Ecritures jamais réordonnancées entre elles
  - Une écriture après une lecture pas réordonnancée
  - **Lecture après écriture peut être réordonnancée si deux cases mémoires différentes !**


Initialement,  $x = y = 0$

P1	P2
$x = 1$	$y = 1$
$res1 = y$	$res2 = x$
$res1 = 1, res2 = 1$	

P1	P2
$x = 1$	
$res1 = y$	
	$y = 1$
	$res2 = x$
$res1 = 0, res2 = 1$	

P1	P2
	$y = 1$
	$res2 = x$
$x = 1$	
$res1 = y$	
$res1 = 1, res2 = 0$	

P1	P2
$x = 1$	
$res1 = y$	
	$res2 = x$
	$y = 1$
$res1 = 0, res2 = 0$	



# OUTILS : RAPPELS SUR LE MODÈLE MÉMOIRE

- Au final, en TSO, problèmes de réordonnancement pas si courants... Mais du coup subtils.
  - Barrières `mfence()` si nécessaire.

# OUTILS : RAPPELS SUR LE MODÈLE MÉMOIRE

- Au final, en TSO, problèmes de réordonnancement pas si courants... Mais du coup subtils.
  - Barrières `mfence()` si nécessaire.
- Par contre, barrières compilateur utiles, même s'il assure aussi TSO
  - **Pour éviter de lire une variable stockée dans un registre !**
  - Pour lire variable modifiée ailleurs, soit volatile, soit barrière au moins soft
    - Incluse dans `_mm_pause()` si on fait de l'attente active...



# OUTILS : RAPPELS SUR LE MODÈLE MÉMOIRE

- Au final, en TSO, problèmes de réordonnancement pas si courants... Mais du coup subtils.
  - Barrières `mfence()` si nécessaire.
- Par contre, barrières compilateur utiles, même s'il assure aussi TSO
  - **Pour éviter de lire une variable stockée dans un registre !**
  - Pour lire variable modifiée ailleurs, soit volatile, soit barrière au moins soft
    - Incluse dans `_mm_pause()` si on fait de l'attente active...
- Code portable pour ARM, PPC, SPARC... ?
  - **Faire très attention (un peu prise de tête...).**

# OUTILS : INSTRUCTIONS ATOMIQUES

- **Atomic add** : addition atomique en mémoire (ajoute value à variable)  
old-value  $\leftarrow$  atomic-add(variable, value)  
*Existe en fait avec add, sub, or, and, xor, nand.*

# OUTILS : INSTRUCTIONS ATOMIQUES

- **Atomic add** : addition atomique en mémoire (ajoute value à variable)  
old-value  $\leftarrow$  atomic-add(variable, value)  
*Existe en fait avec add, sub, or, and, xor, nand.*
- **Atomic swap** : écrit une valeur en mémoire et renvoie son ancienne valeur  
old-value  $\leftarrow$  atomic-swap(variable, value)

# OUTILS : INSTRUCTIONS ATOMIQUES

- **Atomic add** : addition atomique en mémoire (ajoute value à variable)

old-value  $\leftarrow$  `atomic-add`(variable, value)

*Existe en fait avec add, sub, or, and, xor, nand.*

- **Atomic swap** : écrit une valeur en mémoire et renvoie son ancienne valeur

old-value  $\leftarrow$  `atomic-swap`(variable, value)

- **Atomic compare-and-swap** : compare avec old, si égal, fait un swap avec new

old-value  $\leftarrow$  `atomic-CAS`(variable, old, new)

```
atomic {  
    if (variable == old)  
        variable = new;  
    return old;  
    return variable;  
}
```

# OUTILS : INSTRUCTIONS ATOMIQUES

- Comment utiliser ces instructions atomiques ?
  - On peut utiliser de l'assembleur directement...

# OUTILS : INSTRUCTIONS ATOMIQUES

- Comment utiliser ces instructions atomiques ?
  - On peut utiliser de l'assembleur directement...
- **Atomic add** : instruction **lock xadd**

# OUTILS : INSTRUCTIONS ATOMIQUES

- Comment utiliser ces instructions atomiques ?
  - On peut utiliser de l'assembleur directement...
- **Atomic add** : instruction **lock** xadd
- **Atomic swap** : instruction xchg
  - Préfixe **lock** implicite pour cette instruction

# OUTILS : INSTRUCTIONS ATOMIQUES

- Comment utiliser ces instructions atomiques ?
  - On peut utiliser de l'assembleur directement...
- **Atomic add** : instruction **lock** xadd
- **Atomic swap** : instruction xchg
  - Préfixe **lock** implicite pour cette instruction
- **Atomic compare-and-swap** : **lock** cmpxchg



# OUTILS : INSTRUCTIONS ATOMIQUES

- Comment utiliser ces instructions atomiques ?
  - On peut utiliser de l'assembleur directement...
- **Atomic add** : instruction **lock** xadd
- **Atomic swap** : instruction xchg
  - Préfixe **lock** implicite pour cette instruction
- **Atomic compare-and-swap** : **lock** cmpxchg
- **En gros** : le préfixe **lock** assure l'atomicité si plusieurs cœurs
  - *Implémentation naïve sur machine simple : verrouillage du bus...*

# OUTILS : INSTRUCTIONS ATOMIQUES

- Utiliser de l'assembleur en C ? Inline assembly :

```
static inline void *xchg_64(void *ptr, void *x) {  
    asm volatile("xchgq %0,%1"  
        : "=r" ((unsigned long long) x)  
        : "m" (*(volatile long long *)ptr), "O" ((unsigned long long) x)  
        : "memory");  
    return x;  
}
```

# OUTILS : INSTRUCTIONS ATOMIQUES

- Utiliser de l'assembleur en C ? Inline assembly :

```
static inline void *xchg_64(void *ptr, void *x) {  
    asm volatile("xchgq %0,%1"  
        : "=r" ((unsigned long long) x)  
        : "m" (*(volatile long long *)ptr), "O" ((unsigned long long) x)  
        : "memory");  
    return x;  
}
```

- `asm volatile` ou `__asm__ volatile`

# OUTILS : INSTRUCTIONS ATOMIQUES

- Utiliser de l'assembleur en C ? Inline assembly :

```
static inline void *xchg_64(void *ptr, void *x) {  
    asm volatile("xchgq %0,%1"  
        : "=r" ((unsigned long long) x)  
        : "m" (*(volatile long long *)ptr), "O" ((unsigned long long) x)  
        : "memory");  
    return x;  
}
```

- `asm volatile` ou `__asm__ volatile`

- Premier « paramètre » : instruction avec %0, %1, %2, ... remplace les opérandes dans l'instruction.

# OUTILS : INSTRUCTIONS ATOMIQUES

- Utiliser de l'assembleur en C ? Inline assembly :

```
static inline void *xchg_64(void *ptr, void *x) {  
    asm volatile("xchgq %0,%1"  
        : "=r" ((unsigned long long) x)  
        : "m" (*(volatile long long *)ptr), "O" ((unsigned long long) x)  
        : "memory");  
    return x;  
}
```

- `asm volatile` ou `__asm__ volatile`

- Premier « paramètre » : instruction avec %0, %1, %2, ... remplace les opérandes dans l'instruction.
- : output operands, avec contraintes. E.g., « = » veut dire overwrite, « r » veut dire registre possible

# OUTILS : INSTRUCTIONS ATOMIQUES

- Utiliser de l'assembleur en C ? Inline assembly :

```
static inline void *xchg_64(void *ptr, void *x) {  
    asm volatile("xchgq %0,%1"  
        : "=r" ((unsigned long long) x)  
        : "m" (*(volatile long long *)ptr), "O" ((unsigned long long) x)  
        : "memory");  
    return x;  
}
```

- `asm volatile` ou `__asm__ volatile`

- Premier « paramètre » : instruction avec %0, %1, %2, ... remplace les opérandes dans l'instruction.
- : output operands, avec contraintes. E.g., « = » veut dire overwrite, « r » veut dire registre possible
- : input operands, avec contraintes. E.g., « m » = adresse mémoire, « O » même contrainte que op 0 (la première)

# OUTILS : INSTRUCTIONS ATOMIQUES

- Utiliser de l'assembleur en C ? Inline assembly :

```
static inline void *xchg_64(void *ptr, void *x) {  
    asm volatile("xchgq %0,%1"  
        : "=r" ((unsigned long long) x)  
        : "m" (*(volatile long long *)ptr), "O" ((unsigned long long) x)  
        : "memory");  
    return x;  
}
```

- `asm volatile` ou `__asm__ volatile`

- Premier « paramètre » : instruction avec %0, %1, %2, ... remplace les opérandes dans l'instruction.
- : output operands, avec contraintes. E.g., « = » veut dire overwrite, « r » veut dire registre possible
- : input operands, avec contraintes. E.g., « m » = adresse mémoire, « O » même contrainte que op 0 (la première)
- : clobbers, « memory » en gros flushes les registres (comme pour une barrière soft)
- Souvenez vous : `asm volatile("" ::: "memory");`

# OUTILS : INSTRUCTIONS ATOMIQUES

GCC offre des « atomic builtins » pour éviter d'avoir à écrire de l'assembleur...



# OUTILS : INSTRUCTIONS ATOMIQUES

GCC offre des « atomic builtins » pour éviter d'avoir à écrire de l'assembleur...

- **Atomic add** : `type __sync_fetch_and_add (type *ptr, type value, ...)`
  - Variante qui renvoie nouvelle valeur : `type __sync_add_and_fetch (type *ptr, type value, ...)`
  - Variantes avec add, sub, or, and, xor, et nand.

# OUTILS : INSTRUCTIONS ATOMIQUES

GCC offre des « atomic builtins » pour éviter d'avoir à écrire de l'assembleur...

- **Atomic add** : `type __sync_fetch_and_add (type *ptr, type value, ...)`
  - Variante qui renvoie nouvelle valeur : `type __sync_add_and_fetch (type *ptr, type value, ...)`
  - Variantes avec add, sub, or, and, xor, et nand.
- **Atomic swap** : `type __sync_lock_test_and_set (type *ptr, type value, ...)`
  - Vraiment bizarre ! « *This builtin, as described by Intel, is not a traditional test-and-set operation, but rather an atomic exchange operation. It writes value into \*ptr, and returns the previous contents of \*ptr.* »

# OUTILS : INSTRUCTIONS ATOMIQUES

GCC offre des « atomic builtins » pour éviter d'avoir à écrire de l'assembleur...

- **Atomic add** : `type __sync_fetch_and_add (type *ptr, type value, ...)`
  - Variante qui renvoie nouvelle valeur : `type __sync_add_and_fetch (type *ptr, type value, ...)`
  - Variantes avec add, sub, or, and, xor, et nand.
- **Atomic swap** : `type __sync_lock_test_and_set (type *ptr, type value, ...)`
  - Vraiment bizarre ! « *This builtin, as described by Intel, is not a traditional test-and-set operation, but rather an atomic exchange operation. It writes value into \*ptr, and returns the previous contents of \*ptr.* »
- **Atomic CAS** : `type __sync_val_compare_and_swap (type *ptr, type oldval, type newval, ...)`
  - **Version alternative** : `bool __sync_bool_compare_and_swap (bool *ptr, bool oldval, bool newval, ...)`
    - Renvoie 1 si l'échange à été fait, 0 sinon.
    - Revient à tester si la valeur de retour de la fonction de base est la même que oldval.

# OUTILS : INSTRUCTIONS ATOMIQUES

GCC offre des « atomic builtins » pour éviter d'avoir à écrire de l'assembleur...

- **Atomic add** : `type __sync_fetch_and_add (type *ptr, type value, ...)`
  - Variante qui renvoie nouvelle valeur : `type __sync_add_and_fetch (type *ptr, type value, ...)`
  - Variantes avec add, sub, or, and, xor, et nand.
- **Atomic swap** : `type __sync_lock_test_and_set (type *ptr, type value, ...)`
  - Vraiment bizarre ! « *This builtin, as described by Intel, is not a traditional test-and-set operation, but rather an atomic exchange operation. It writes value into \*ptr, and returns the previous contents of \*ptr.* »
- **Atomic CAS** : `type __sync_val_compare_and_swap (type *ptr, type oldval, type newval, ...)`
  - **Version alternative** : `bool __sync_bool_compare_and_swap (bool *ptr, bool oldval, bool newval, ...)`
    - Renvoie 1 si l'échange à été fait, 0 sinon.
    - Revient à tester si la valeur de retour de la fonction de base est la même que oldval.
- **Déjà vu barrière** : `__sync_synchronize()`...

# ET EN JAVA ?

- Modèle mémoire défini depuis Java 5...

# ET EN JAVA ?

- Modèle mémoire défini depuis Java 5...
- Tout réordonnancement possible, sauf :
  - Variable `volatile` : barrière mémoire à chaque accès (= lock)
    - Pas de réordonnancement avec accès à autres variables non-volatiles...
    - ***volatile bien plus fort qu'en C!***

# ET EN JAVA ?

- Modèle mémoire défini depuis Java 5...
- Tout réordonnancement possible, sauf :
  - Variable **volatile** : barrière mémoire à chaque accès (= lock)
    - Pas de réordonnancement avec accès à autres variables non-volatiles...
    - *volatile bien plus fort qu'en C!*
  - Entrée et sortie de section critique (avec **synchronize**) : barrière mémoire (= lock)
    - Pas surprenant, **synchronize** \*est\* un verrou...
    - En Java, **volatile** pareil qu'un **synchronize** sur la variable...

# ET EN JAVA ?

- **Piège classique** : éviter la prise de verrou lors d'une initialisation

```
if(!inited) {  
    synchronize(obj) {  
        if(!inited) {  
            bidule = new Bidule();  
            inited = true;  
        }  
    }  
}  
bidule.use();
```



# ET EN JAVA ?

- **Piège classique** : éviter la prise de verrou lors d'une initialisation

```
if(!inited) {  
    synchronize(obj) {  
        if(!inited) {  
            bidule = new Bidule();  
            inited = true;  
        }  
    }  
}  
bidule.use();
```

- Possible qu'un thread voie inited à true alors que bidule n'est pas encore initialisé
  - Du coup, appel à use() plante...

# ET EN JAVA ?

- **Piège classique** : éviter la prise de verrou lors d'une initialisation

```
if(!inited) {  
    synchronize(obj) {  
        if(!inited) {  
            bidule = new Bidule();  
            inited = true;  
        }  
    }  
}  
bidule.use();
```

- Possible qu'un thread voie inited à true alors que bidule n'est pas encore initialisé
  - Du coup, appel à use() plante...
- **Solution : marquer inited comme volatile !**

# ET EN JAVA ?

- **Piège classique** : éviter la prise de verrou lors d'une initialisation

```
if(!inited) {  
    synchronize(obj) {  
        if(!inited) {  
            bidule = new Bidule();  
            inited = true;  
        }  
    }  
}  
bidule.use();
```

- Possible qu'un thread voie inited à true alors que bidule n'est pas encore initialisé
  - Du coup, appel à use() plante...
- **Solution : marquer inited comme volatile !**
- **Remarque** : impossible en TSO, sauf si compilateur optimise le code.



**FIN DU (LONG) INTERLUDE  
ON EN REVIENT À :**

**OPTION 2 :  
AMÉLIORER LES ALGORITHMES DE VERROUILLAGE**

# LE SPINLOCK « DE BASE »

- Algorithme de verrou le plus simple au monde :

```
function lock(boolean *lock)
    while atomic_cas(lock, false, true)
        PAUSE();
```

```
function unlock(boolean *lock)
    *lock = false;
```

# LE SPINLOCK « DE BASE »

- Algorithme de verrou le plus simple au monde :

```
function lock(boolean *lock)
    while atomic_cas(lock, false, true)
        PAUSE();
```

```
function unlock(boolean *lock)
    *lock = false;
```

- Gros problème du spinlock « de base » : sature l'interconnect très facilement

# LE SPINLOCK « DE BASE »

- Algorithme de verrou le plus simple au monde :

```
function lock(boolean *lock)
    while atomic_cas(lock, false, true)
        PAUSE();
```

```
function unlock(boolean *lock)
    *lock = false;
```

- Gros problème du spinlock « de base » : sature l'interconnect très facilement
  - Nombreux threads qui font de l'attente active en parallèle à haute contention

# LE SPINLOCK « DE BASE »

- Algorithme de verrou le plus simple au monde :

```
function lock(boolean *lock)
    while atomic_cas(lock, false, true)
        PAUSE();
```

```
function unlock(boolean *lock)
    *lock = false;
```

- Gros problème du spinlock « de base » : sature l'interconnect très facilement
  - Nombreux threads qui font de l'attente active en parallèle à haute contention
  - Messages MOESI dans tous les sens pour amener la ligne de cache d'un cœur à l'autre
    - Effet « ping-pong » !



# LE SPINLOCK « DE BASE »

- Algorithme de verrou le plus simple au monde :

```
function lock(boolean *lock)
    while atomic_cas(lock, false, true)
        PAUSE();
```

```
function unlock(boolean *lock)
    *lock = false;
```

- Gros problème du spinlock « de base » : sature l'interconnect très facilement
  - Nombreux threads qui font de l'attente active en parallèle à haute contention
  - Messages MOESI dans tous les sens pour amener la ligne de cache d'un cœur à l'autre
    - Effet « ping-pong » !
  - À la sortie de chaque section critique, invalidations envoyées à tout le monde en même temps...

# LE SPINLOCK « DE BASE »

- Algorithme de verrou le plus simple au monde :

```
function lock(boolean *lock)
    while atomic_cas(lock, false, true)
        PAUSE();
```

```
function unlock(boolean *lock)
    *lock = false;
```

- Gros problème du spinlock « de base » : sature l'interconnect très facilement
  - Nombreux threads qui font de l'attente active en parallèle à haute contention
  - Messages MOESI dans tous les sens pour amener la ligne de cache d'un cœur à l'autre
    - Effet « ping-pong » !
  - À la sortie de chaque section critique, invalidations envoyées à tout le monde en même temps...
- Existe variante « test-and-test-and-set », perfs similaires sur architectures actuelles...

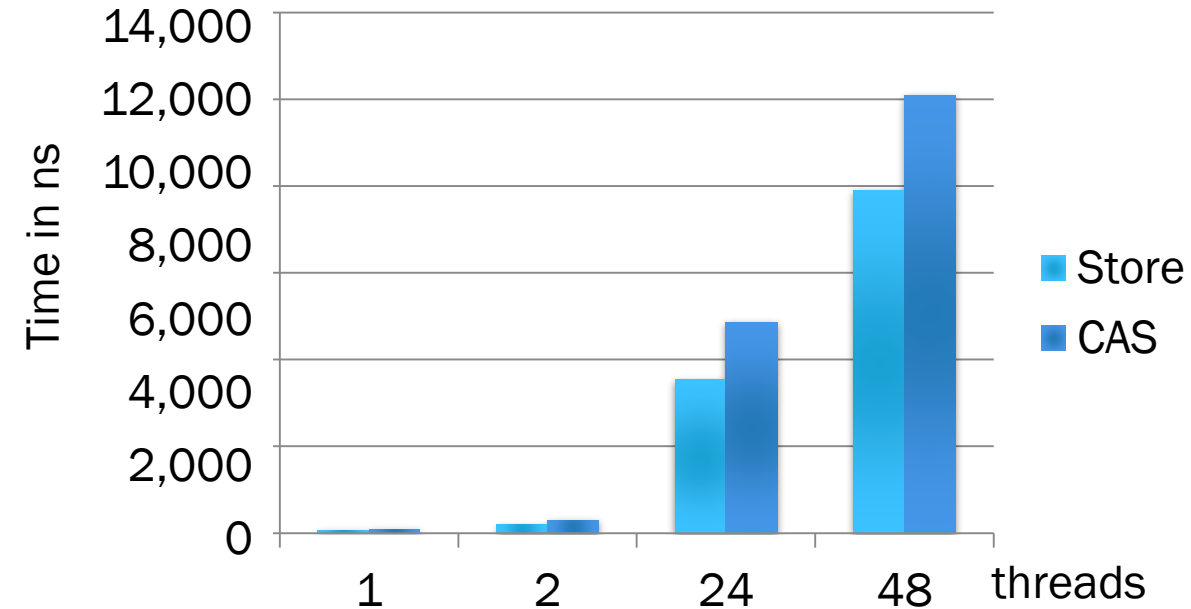
# LE SPINLOCK « DE BASE »

- Algorithme de verrou le plus simple au monde :

```
function lock(boolean *lock)
    while atomic_cas(lock, false, true)
        PAUSE();
```

```
function unlock(boolean *lock)
    *lock = false;
```

- Gros problème du spinlock « de base » : sature l'interconnect très facilement
  - Nombreux threads qui font de l'attente active en parallèle à haute contention
  - Messages MOESI dans tous les sens pour amener la ligne de cache d'un cœur à l'autre
    - Effet « ping-pong » !
  - À la sortie de chaque section critique, invalidations envoyées à tout le monde en même temps...
- Existe variante « test-and-test-and-set », perfs similaires sur architectures actuelles...



# MCS

- Problème du spinlock « de base » : **variable de synchro globale = bottleneck**

# MCS

- Problème du spinlock « de base » : **variable de synchro globale = bottleneck**
- Une solution : les verrous à liste, MCS ou CLH (similaire)

# MCS

- Problème du spinlock « de base » : **variable de synchro globale = bottleneck**
- Une solution : les verrous à liste, MCS ou CLH (similaire)
- On va étudier l'algorithme MCS (Mellor-Crummey & Scott, ASPLOS 1991)

# MCS

- Problème du spinlock « de base » : **variable de synchro globale = bottleneck**
- Une solution : les verrous à liste, MCS ou CLH (similaire)
- On va étudier l'algorithme MCS (Mellor-Crummey & Scott, ASPLOS 1991)
- **Principe** : éviter la contention sur une ligne de cache unique !
  - Chaque thread en attente s'ajoute dans une liste
  - Spin sur une variable dans son propre nœud, libéré par précédent quand unlock
  - **Une variable de synchro par thread !**

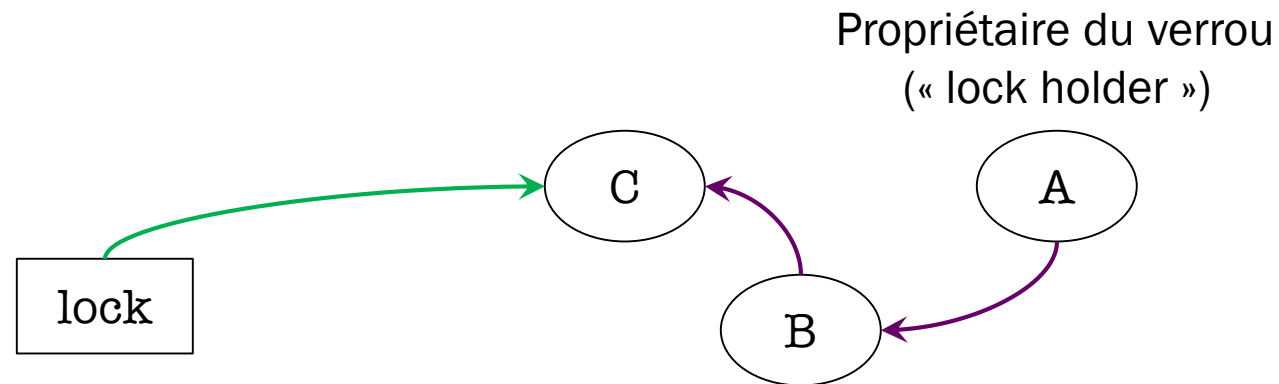
# MCS

- **Idée** : dernier thread qui demande le verrou stocké dans la variable lock
  - Si verrou pris, doit chaîner l'ancienne liste et attendre d'être libéré
  - Sinon, verrou directement acquis



# MCS

- **Idée** : dernier thread qui demande le verrou stocké dans la variable lock
  - Si verrou pris, doit chaîner l'ancienne liste et attendre d'être libéré
  - Sinon, verrou directement acquis
  - **lock est la tête de liste : permet de se mettre en queue.**
  - Les passages de verrous suivent les flèches violettes.



# MCS

```
struct node {  
    struct node* next;  
    int locked;  
};  
  
// À répliquer pour chaque verrou...  
struct node* lock;  
static __thread struct node my_node = {0, 1};
```

# MCS

```
struct node {  
    struct node* next;  
    int locked;  
};  
  
// À répliquer pour chaque verrou...  
struct node* lock;  
static __thread struct node my_node = {0, 1};  
Thread-local storage ! (GCC)
```

# MCS

```
void CS() {  
    // Lock  
    struct node* pred = SWAP (&lock, node);  
    if(pred) {  
        pred->next = my_node;  
        while (!my_node->locked)  
            PAUSE();  
    }  
  
    EXECUTE_CS();  
  
    // Unlock  
    if(CAS(&lock, my_node, 0) == 0)  
        return;  
    while(!my_node->next)  
        PAUSE();  
    my_node->next->locked = 1;  
}
```

```
struct node {  
    struct node* next;  
    int locked;  
};
```

*// À répliquer pour chaque verrou...*

```
struct node* lock;  
static __thread struct node my_node = {0, 1};
```

Thread-local storage ! (GCC)

# MCS

## D appelle CS()

```
void CS() {  
    struct node *node = &my_node;  
  
    // Lock  
    struct node* pred = SWAP (&lock, node);  
    if(pred) {  
        pred->next = node;  
        while (!node->locked)  
            PAUSE();  
    }  
}
```

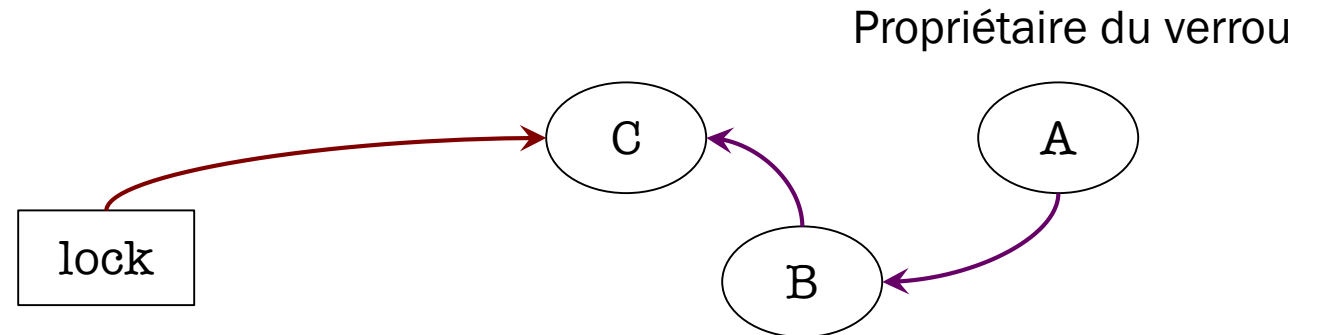
**EXECUTE\_CS();**

```
// Unlock  
if(CAS(&lock, node, 0) == 0)  
    return;  
while(!node->next)  
    PAUSE();  
node->next->locked = 1;  
}
```

```
struct node {  
    struct node* next;  
    int locked;  
};
```

*// À répliquer pour chaque verrou...*

```
struct node* lock;  
static __thread struct node my_node = {0, 1};
```



# MCS

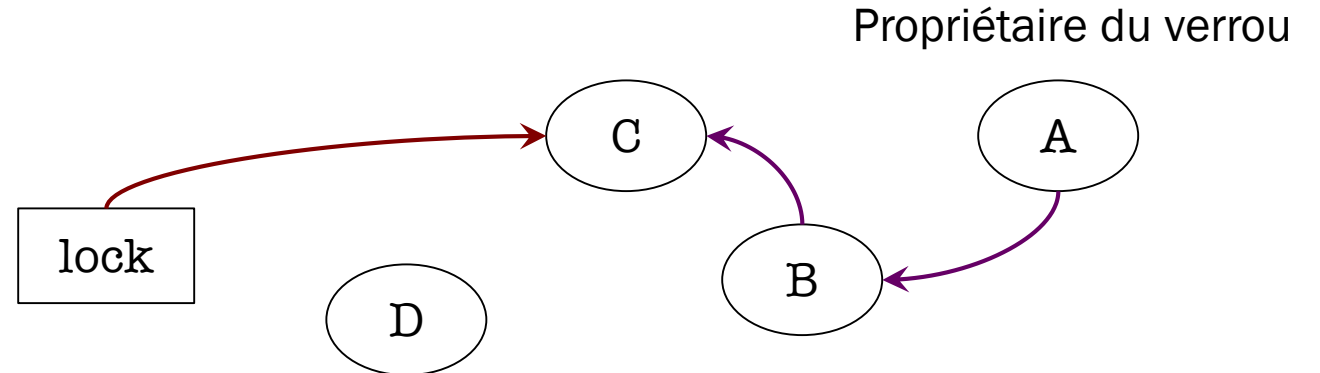
## D appelle CS()

```
void CS() {  
    struct node *node = &my_node;  
  
    // Lock  
    struct node* pred = SWAP (&lock, node);  
    if(pred) {  
        pred->next = node;  
        while (!node->locked)  
            PAUSE();  
    }  
  
    EXECUTE_CS();  
  
    // Unlock  
    if(CAS(&lock, node, 0) == 0)  
        return;  
    while(!node->next)  
        PAUSE();  
    node->next->locked = 1;  
}
```

```
struct node {  
    struct node* next;  
    int locked;  
};
```

*// À répliquer pour chaque verrou...*

```
struct node* lock;  
static __thread struct node my_node = {0, 1};
```



# MCS

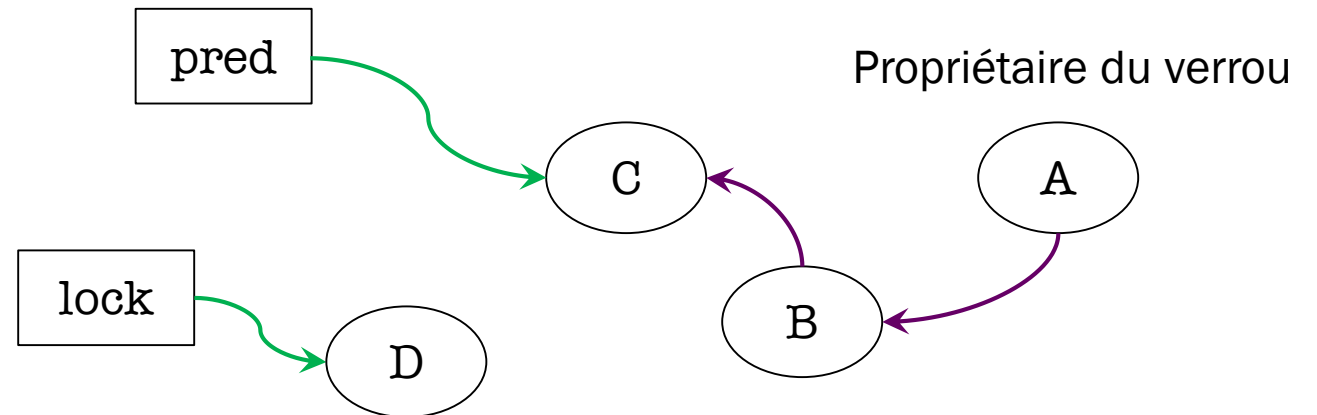
## D appelle CS()

```
void CS() {  
    struct node *node = &my_node;  
  
    // Lock  
    struct node* pred = SWAP (&lock, node);  
    if(pred) {  
        pred->next = node;  
        while (!node->locked)  
            PAUSE();  
    }  
  
    EXECUTE_CS();  
  
    // Unlock  
    if(CAS(&lock, node, 0) == 0)  
        return;  
    while(!node->next)  
        PAUSE();  
    node->next->locked = 1;  
}
```

```
struct node {  
    struct node* next;  
    int locked;  
};
```

*// À répliquer pour chaque verrou...*

```
struct node* lock;  
static __thread struct node my_node = {0, 1};
```



# MCS

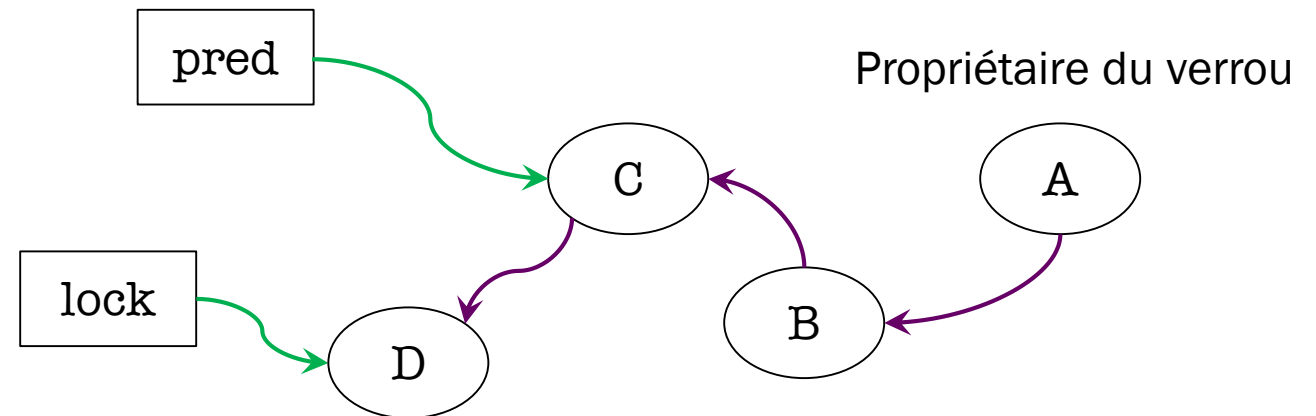
## D appelle CS()

```
void CS() {  
    struct node *node = &my_node;  
  
    // Lock  
    struct node* pred = SWAP (&lock, node);  
    if(pred) {  
        pred->next = node;  
        while (!node->locked) Chaînage en "deux pas"...  
            PAUSE();  
    }  
  
    EXECUTE_CS();  
  
    // Unlock  
    if(CAS(&lock, node, 0) == 0)  
        return;  
    while(!node->next)  
        PAUSE();  
    node->next->locked = 1;  
}
```

```
struct node {  
    struct node* next;  
    int locked;  
};
```

*// À répliquer pour chaque verrou...*

```
struct node* lock;  
static __thread struct node my_node = {0, 1};
```





# MCS

## D appelle CS()

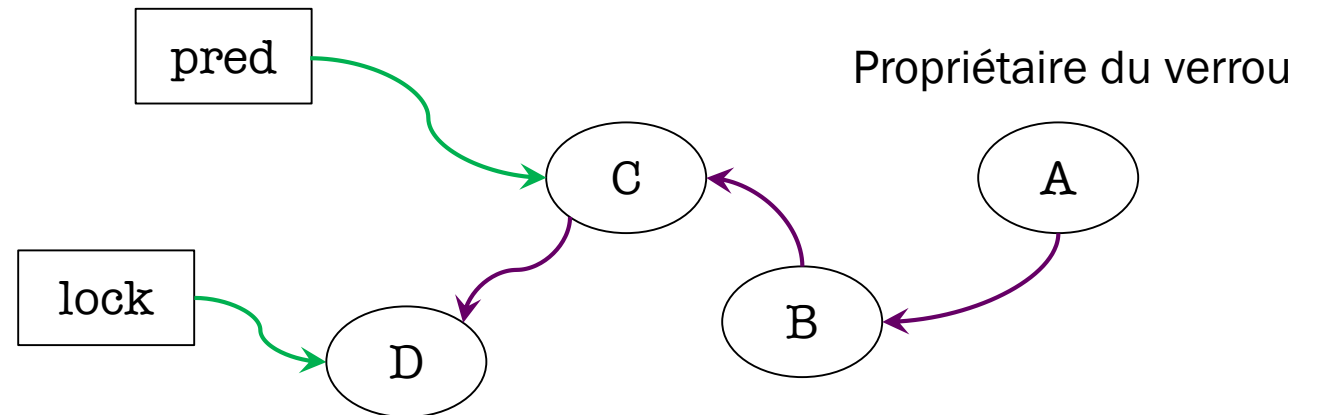
```
void CS() {  
    struct node *node = &my_node;  
  
    // Lock  
    struct node* pred = SWAP (&lock, node);  
    if(pred) {  
        pred->next = node;  
        while (!node->locked)  
            PAUSE();  
    }  
    EXECUTE_CS();  
  
    // Unlock  
    if(CAS(&lock, node, 0) == 0)  
        return;  
    while(!node->next)  
        PAUSE();  
    node->next->locked = 1;  
}
```

*Attente active sur variable  
de synchro du thread...*

```
struct node {  
    struct node* next;  
    int locked;  
};
```

*// À répliquer pour chaque verrou...*

```
struct node* lock;  
static __thread struct node my_node = {0, 1};
```



# MCS

## A fait son unlock

```
void CS() {  
    struct node *node = &my_node;  
  
    // Lock  
    struct node* pred = SWAP (&lock, node);  
    if(pred) {  
        pred->next = node;  
        while (!node->locked)  
            PAUSE();  
    }  
}
```

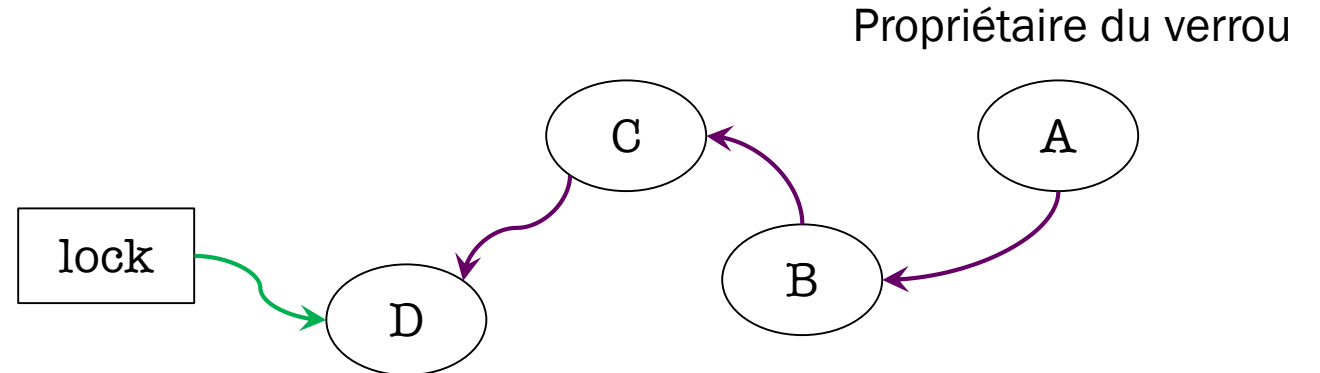
**EXECUTE\_CS();**

```
// Unlock  
if(CAS(&lock, node, 0) == 0)  
    return;  
while(!node->next)  
    PAUSE();  
node->next->locked = 1;  
}
```

```
struct node {  
    struct node* next;  
    int locked;  
};
```

*// À répliquer pour chaque verrou...*

```
struct node* lock;  
static __thread struct node my_node = {0, 1};
```



# MCS

## A fait son unlock

```
void CS() {  
    struct node *node = &my_node;  
  
    // Lock  
    struct node* pred = SWAP (&lock, node);  
    if(pred) {  
        pred->next = node;  
        while (!node->locked)  
            PAUSE();  
    }  
}
```

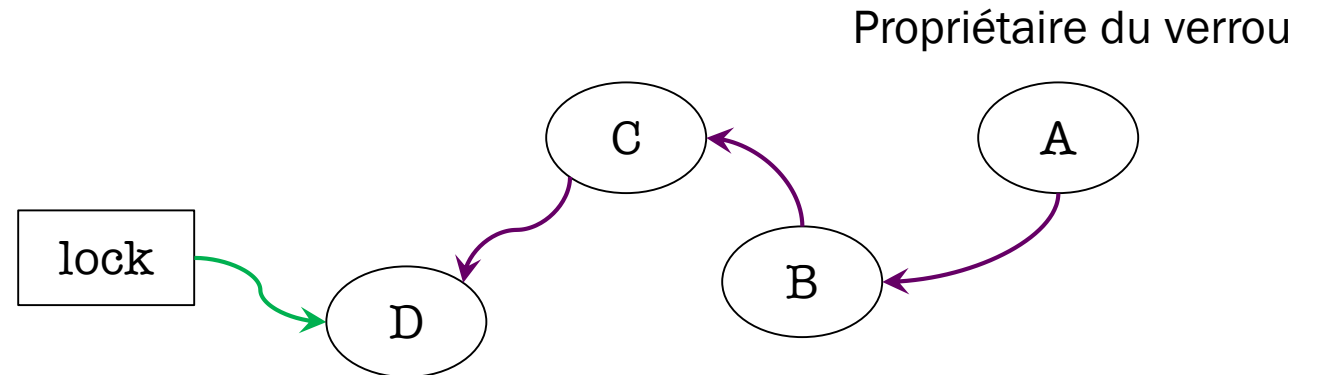
**EXECUTE\_CS();**

```
// Unlock  
if(CAS(&lock, node, 0) == 0)  
    return;  
while(!node->next)  
    PAUSE();  
node->next->locked = 1;  
}
```

```
struct node {  
    struct node* next;  
    int locked;  
};
```

*// À répliquer pour chaque verrou...*

```
struct node* lock;  
static __thread struct node my_node = {0, 1};
```



*Faux : A n'est pas le  
dernier lock holder...  
Il faut réveiller le suivant !*

# MCS

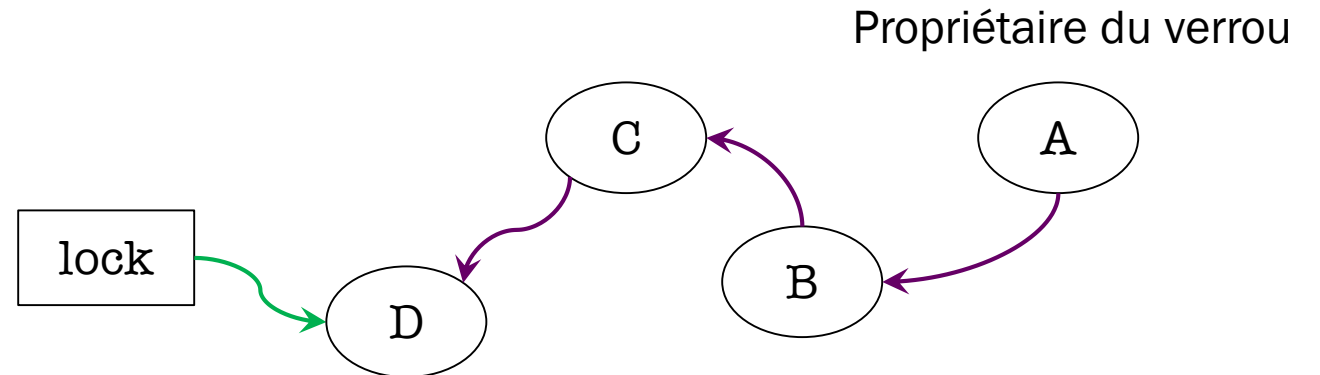
## A fait son unlock

```
void CS() {  
    struct node *node = &my_node;  
  
    // Lock  
    struct node* pred = SWAP (&lock, node);  
    if(pred) {  
        pred->next = node;  
        while (!node->locked)  
            PAUSE();  
    }  
  
    EXECUTE_CS();  
  
    // Unlock  
    if(CAS(&lock, node, 0) == 0)  
        return;  
    while(!node->next)  
        PAUSE();  
    node->next->locked = 1;  
}
```

```
struct node {  
    struct node* next;  
    int locked;  
};
```

// À répliquer pour chaque verrou...

```
struct node* lock;  
static __thread struct node my_node = {0, 1};
```



*Il se peut que le suivant soit entre les deux pas !  
Vrai ssi on n'est pas le dernier mais on n'a pas de suivant...  
Si nécessaire, courte attente active ! (pas ici)*

# MCS

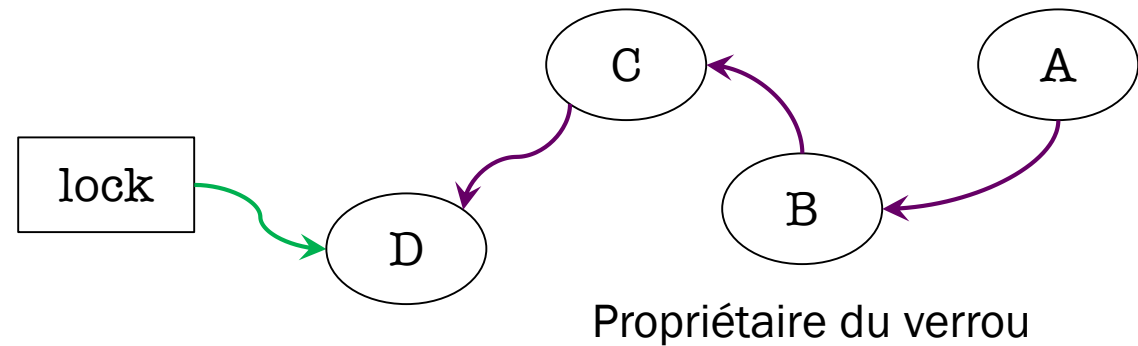
A fait son unlock

```
void CS() {  
    struct node *node = &my_node;  
  
    // Lock  
    struct node* pred = SWAP (&lock, node);  
    if(pred) {  
        pred->next = node;  
        while (!node->locked)  
            PAUSE();  
    }  
  
    EXECUTE_CS();  
  
    // Unlock  
    if(CAS(&lock, node, 0) == 0)  
        return;  
    while(!node->next)  
        PAUSE();  
    node->next->locked = 1;  
}
```

```
struct node {  
    struct node* next;  
    int locked;  
};
```

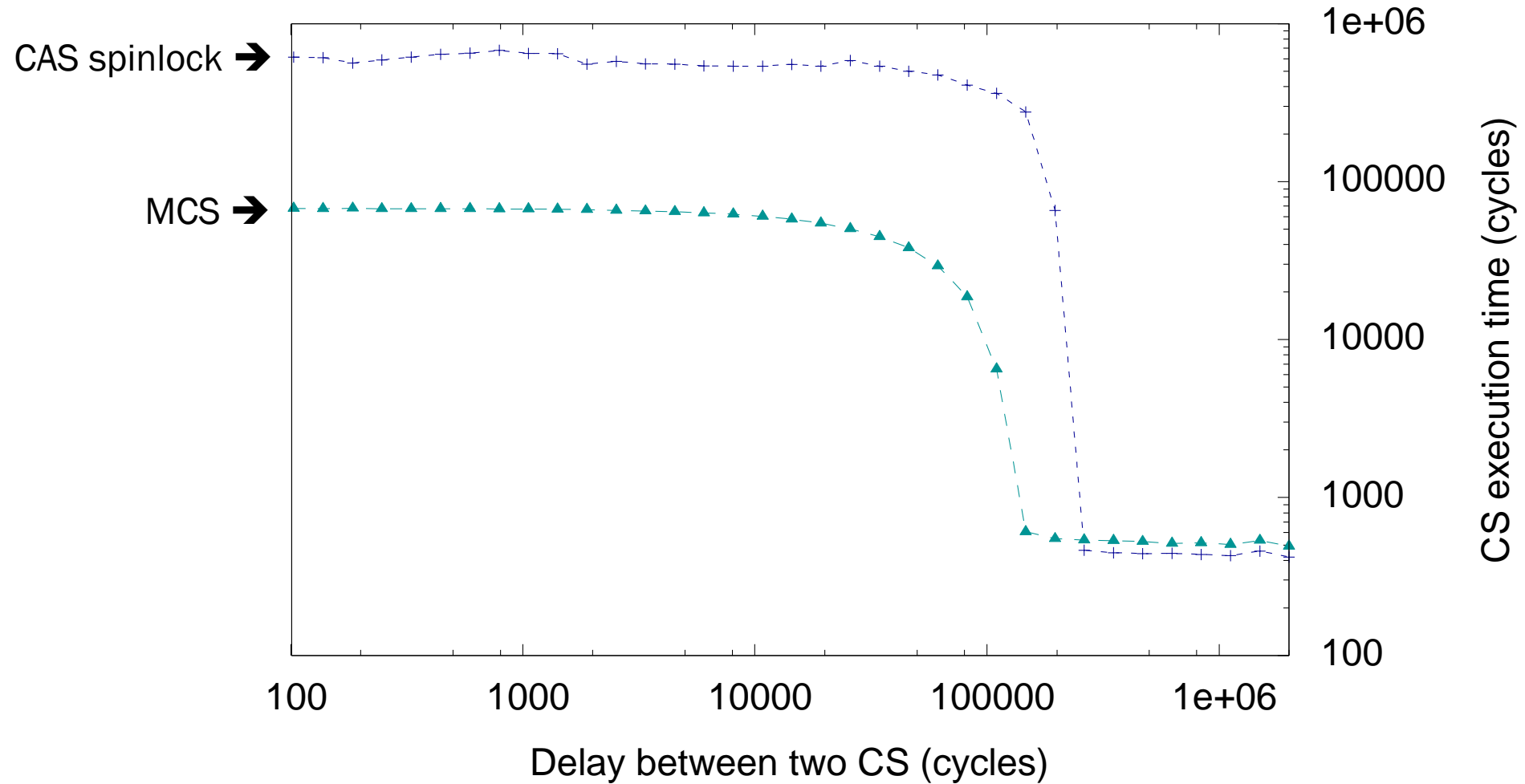
// À répliquer pour chaque verrou...

```
struct node* lock;  
static __thread struct node my_node = {0, 1};
```

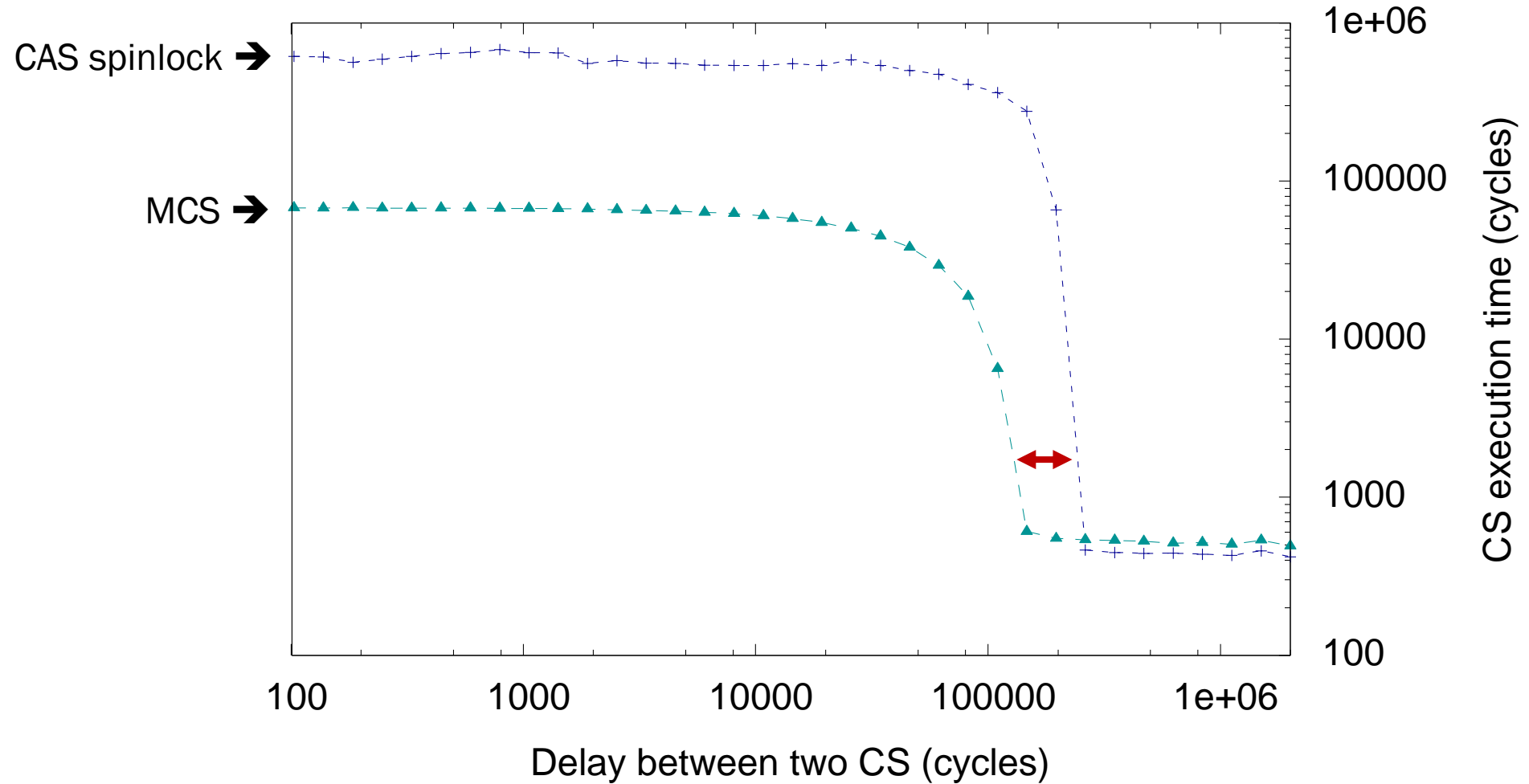


A sort B de sa boucle en mettant son locked à 0  
(Inutile de s'enlever de la liste)

# MCS



# MCS



# VERROUS HIÉRARCHIQUES

- MCS optimal ?
  - On passe encore le verrou d'un cœur à l'autre, sur le chemin critique



# VERROUS HIÉRARCHIQUES

- MCS optimal ?
  - On passe encore le verrou d'un cœur à l'autre, sur le chemin critique
  - Marche avec une invalidation/récupération de la ligne de cache

# VERROUS HIÉRARCHIQUES

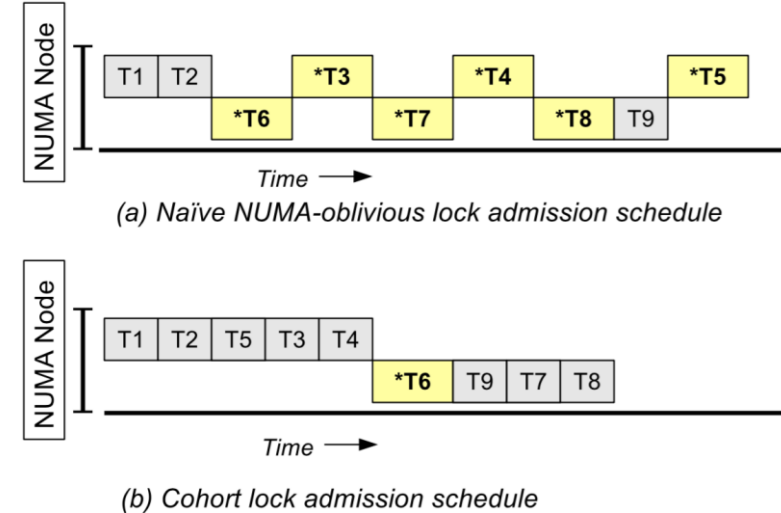
- MCS optimal ?
  - On passe encore le verrou d'un cœur à l'autre, sur le chemin critique
  - Marche avec une invalidation/récupération de la ligne de cache
  - Peut-être coûteux si les cœurs sont « loin » l'un de l'autre...

# VERROUS HIÉRARCHIQUES

- **MCS optimal ?**
  - On passe encore le verrou d'un cœur à l'autre, sur le chemin critique
  - Marche avec une invalidation/récupération de la ligne de cache
  - Peut-être coûteux si les cœurs sont « loin » l'un de l'autre...
- **Pour faire mieux, verrous hiérarchiques (ou « NUMA »)**
  - **Constatation** : plus rapide de passer le verrou à un voisin sur un nœud...

# VERROUS HIÉRARCHIQUES

- MCS optimal ?
  - On passe encore le verrou d'un cœur à l'autre, sur le chemin critique
  - Marche avec une invalidation/récupération de la ligne de cache
  - Peut-être coûteux si les cœurs sont « loin » l'un de l'autre...
- Pour faire mieux, verrous hiérarchiques (ou « NUMA »)
  - **Constatation** : plus rapide de passer le verrou à un voisin sur un nœud...
  - **Idée** : passer en priorité le verrou aux voisins du nœud pendant un moment, puis suivant



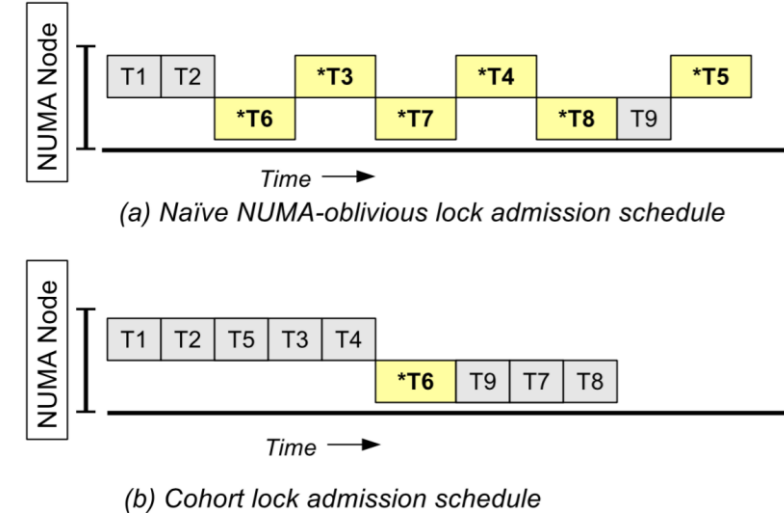
# VERROUS HIÉRARCHIQUES

- MCS optimal ?

- On passe encore le verrou d'un cœur à l'autre, sur le chemin critique
- Marche avec une invalidation/récupération de la ligne de cache
- Peut-être coûteux si les cœurs sont « loin » l'un de l'autre...

- Pour faire mieux, verrous hiérarchiques (ou « NUMA »)

- **Constatation** : plus rapide de passer le verrou à un voisin sur un nœud...
- **Idée** : passer en priorité le verrou aux voisins du nœud pendant un moment, puis suivant
- On échange de la « fairness » temporairement pour des performances...



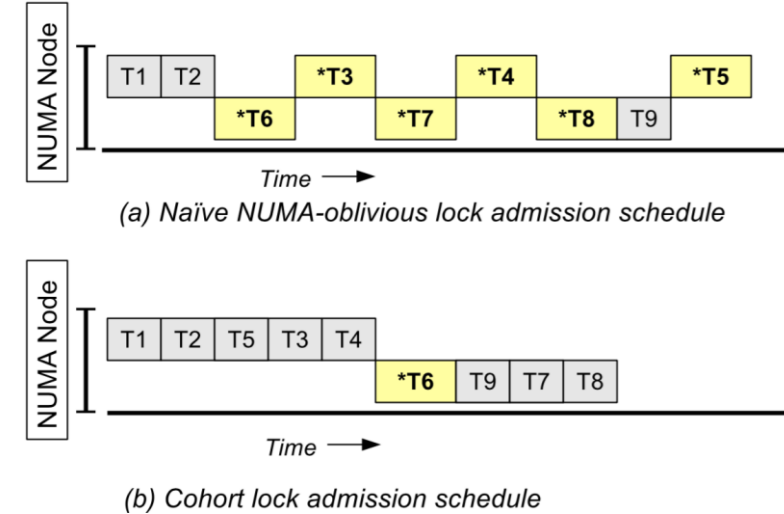
# VERROUS HIÉRARCHIQUES

- MCS optimal ?

- On passe encore le verrou d'un cœur à l'autre, sur le chemin critique
- Marche avec une invalidation/récupération de la ligne de cache
- Peut-être coûteux si les cœurs sont « loin » l'un de l'autre...

- Pour faire mieux, verrous hiérarchiques (ou « NUMA »)

- **Constatation** : plus rapide de passer le verrou à un voisin sur un nœud...
- **Idée** : passer en priorité le verrou aux voisins du nœud pendant un moment, puis suivant
- On échange de la « fairness » temporairement pour des performances...
- *Pas si simple* : éviter la famine !



# VERROUS HIÉRARCHIQUES : UN PEU DE LECTURE...

- Un des premiers : HBO (Hierarchical Backoff Lock)
  - *Radovic et al, Hierarchical Backoff Locks for Nonuniform Communication Architectures, HPCA '03*

# VERROUS HIÉRARCHIQUES : UN PEU DE LECTURE...

- **Un des premiers** : HBO (Hierarchical Backoff Lock)
  - *Radovic et al, Hierarchical Backoff Locks for Nonuniform Communication Architectures, HPCA '03*
- **Verrous à liste** : versions hiérarchiques de MCS et CLH
  - *H-CLH : Luchangko et al, « A hierarchical CLH queue lock », Euro-Par '06*
  - *Hierarchical MCS : Chabbi et al, « High performance locks for multi-level NUMA systems », PPOPP '15*



# VERROUS HIÉRARCHIQUES : UN PEU DE LECTURE...

- **Un des premiers** : HBO (Hierarchical Backoff Lock)
  - *Radovic et al, Hierarchical Backoff Locks for Nonuniform Communication Architectures, HPCA '03*
- **Verrous à liste** : versions hiérarchiques de MCS et CLH
  - *H-CLH : Luchangko et al, « A hierarchical CLH queue lock », Euro-Par '06*
  - *Hierarchical MCS : Chabbi et al, « High performance locks for multi-level NUMA systems », PPOPP '15*
- **Cohort locks** : construction universelle pour créer des verrous hiérarchiques !
  - **Concept** : prenez deux algos de verrou non-hiérarchiques qui satisfont des propriétés basiques...
    - ...et utilisez-les pour construire votre verrou hiérarchique !

# VERROUS HIÉRARCHIQUES : UN PEU DE LECTURE...

- **Un des premiers** : HBO (Hierarchical Backoff Lock)
  - *Radovic et al, Hierarchical Backoff Locks for Nonuniform Communication Architectures, HPCA '03*
- **Verrous à liste** : versions hiérarchiques de MCS et CLH
  - *H-CLH : Luchangko et al, « A hierarchical CLH queue lock », Euro-Par '06*
  - *Hierarchical MCS : Chabbi et al, « High performance locks for multi-level NUMA systems », PPOPP '15*
- **Cohort locks** : construction universelle pour créer des verrous hiérarchiques !
  - **Concept** : prenez deux algos de verrou non-hiérarchiques qui satisfont des propriétés basiques...
    - ...et utilisez-les pour construire votre verrou hiérarchique !
  - **Idée** : l'un des algos de verrouillage sert de verrou local au nœud (pour le passage rapide)...
    - ...et l'autre sert à protéger le passage de nœud en nœud.

# VERROUS HIÉRARCHIQUES : UN PEU DE LECTURE...

- **Un des premiers** : HBO (Hierarchical Backoff Lock)
  - *Radovic et al, Hierarchical Backoff Locks for Nonuniform Communication Architectures, HPCA '03*
- **Verrous à liste** : versions hiérarchiques de MCS et CLH
  - *H-CLH : Luchangko et al, « A hierarchical CLH queue lock », Euro-Par '06*
  - *Hierarchical MCS : Chabbi et al, « High performance locks for multi-level NUMA systems », PPOPP '15*
- **Cohort locks** : construction universelle pour créer des verrous hiérarchiques !
  - **Concept** : prenez deux algos de verrou non-hiérarchiques qui satisfont des propriétés basiques...
    - ...et utilisez-les pour construire votre verrou hiérarchique !
  - **Idée** : l'un des algos de verrouillage sert de verrou local au nœud (pour le passage rapide)...
    - ...et l'autre sert à protéger le passage de nœud en nœud.
  - Parait simple, mais belle exécution ! Papier « fun » à lire...
    - *Dice et al, « Lock cohorting: a general technique for designing NUMA locks », PPOPP '12*

# VERROUS « À COMBINATEUR »

- Même avec les verrous hiérarchiques, communication entre chaque CS
  - Lorsqu'un thread réveille le suivant, même en local

# VERROUS « À COMBinateur »

- Même avec les verrous hiérarchiques, communication entre chaque CS
  - Lorsqu'un thread réveille le suivant, même en local
- À votre avis, possible de faire mieux ?

# VERROUS « À COMBINATEUR »

- Même avec les verrous hiérarchiques, communication entre chaque CS
  - Lorsqu'un thread réveille le suivant, même en local
- À votre avis, possible de faire mieux ?
- Oui, avec les verrous « à combineur » :
  - Quand verrou pris, threads se mettent en queue, comme avec MCS...

# VERROUS « À COMBINA TEUR »

- Même avec les verrous hiérarchiques, communication entre chaque CS
  - Lorsqu'un thread réveille le suivant, même en local
- **À votre avis, possible de faire mieux ?**
- Oui, avec les verrous « à combine teur » :
  - Quand verrou pris, threads se mettent en queue, comme avec MCS...
  - **Idée géniale** : quand le lock holder a fini avec sa CS... Il exécute les CS des autres à leur place !

# VERROUS « À COMBINA TEUR »

- Même avec les verrous hiérarchiques, communication entre chaque CS
  - Lorsqu'un thread réveille le suivant, même en local
- **À votre avis, possible de faire mieux ?**
- Oui, avec les verrous « à combine teur » :
  - Quand verrou pris, threads se mettent en queue, comme avec MCS...
  - **Idée géniale** : quand le lock holder a fini avec sa CS... Il exécute les CS des autres à leur place !
  - **Exécution de plein de CS les unes après les autres sans synchro !**



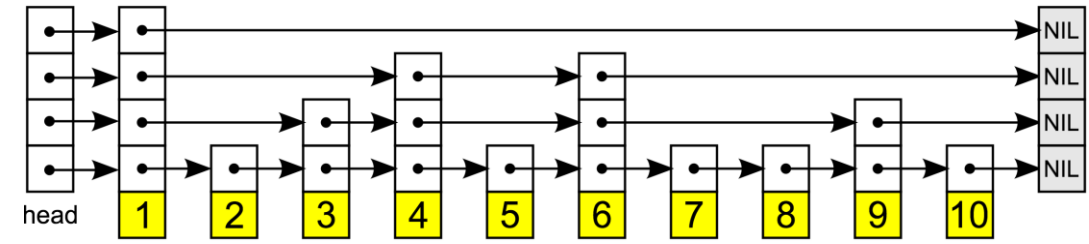
# VERROUS « À COMBINA TEUR »

- Même avec les verrous hiérarchiques, communication entre chaque CS
  - Lorsqu'un thread réveille le suivant, même en local
- **À votre avis, possible de faire mieux ?**
- Oui, avec les verrous « à combine teur » :
  - Quand verrou pris, threads se mettent en queue, comme avec MCS...
  - **Idée géniale** : quand le lock holder a fini avec sa CS... Il exécute les CS des autres à leur place !
  - **Exécution de plein de CS les unes après les autres sans synchro !**
  - Puis, il passe la main à un autre...

# VERROUS « À COMBINATEUR »

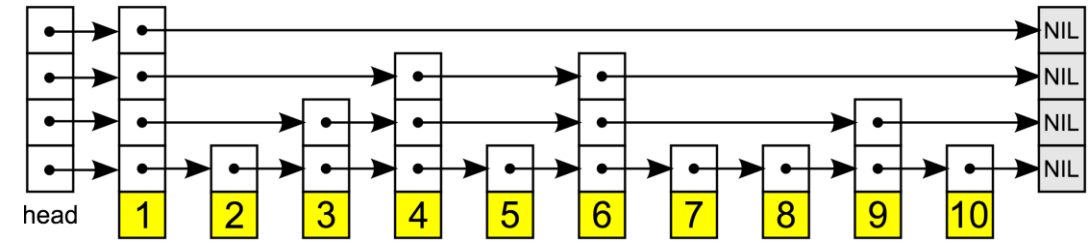
- Même avec les verrous hiérarchiques, communication entre chaque CS
  - Lorsqu'un thread réveille le suivant, même en local
- **À votre avis, possible de faire mieux ?**
- Oui, avec les verrous « à combinateur » :
  - Quand verrou pris, threads se mettent en queue, comme avec MCS...
  - **Idée géniale** : quand le lock holder a fini avec sa CS... Il exécute les CS des autres à leur place !
  - **Exécution de plein de CS les unes après les autres sans synchro !**
  - Puis, il passe la main à un autre...
  - *Un peu subtil, car il faut passer un pointeur de fonction, migrer le contexte, etc...*
  - **Idée originale** : Oyama Lock (vieux papier indigeste, pas très fun)

# VERROUS « À COMBINA TEUR »



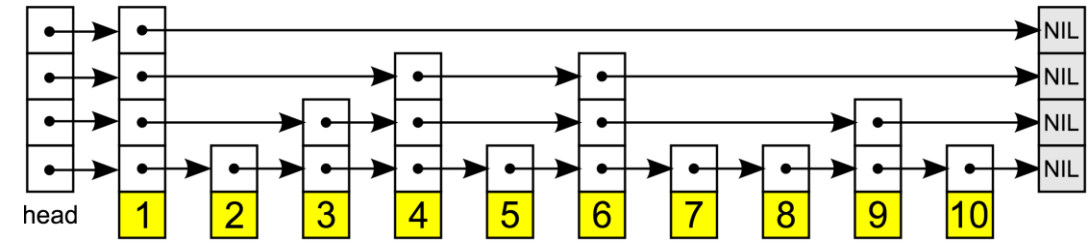
- Pour certaines structures de données :
  - Plus rapide d'effectuer une opération avec  $n$  éléments que  $n$  opérations avec un éléments.

# VERROUS « À COMBINA TEUR »



- Pour certaines structures de données :
  - Plus rapide d'effectuer une opération avec  $n$  éléments que  $n$  opérations avec un éléments.
  - **Par exemple :** une skiplist (liste à enjambements) à priorité
    - Possible de faire  $k$  opérations « remove-min » d'un coup en  $O(k + \log n)$
    - Alors que faire une opération « remove-min »  $k$  fois =  $O(k \log n)$

# VERROUS « À COMBINATEUR »



- Pour certaines structures de données :
  - Plus rapide d'effectuer une opération avec  $n$  éléments que  $n$  opérations avec un éléments.
  - **Par exemple :** une skiplist (liste à enjambements) à priorité
    - Possible de faire  $k$  opérations « remove-min » d'un coup en  $O(k + \log n)$
    - Alors que faire une opération « remove-min »  $k$  fois =  $O(k \log n)$
- Du coup, autre idée géniale :
  - **Le combineur regarde la liste de CS qu'il a à exécuter et combine les opérations !**
  - Réduit la complexité !

# VERROUS « À COMBINA TEUR »

- Autre avantage des verrous « à combine teur » :
  - CS protègent généralement les mêmes variables partagées...

# VERROUS « À COMBINA TEUR »

- Autre avantage des verrous « à combine teur » :
  - CS protègent généralement les mêmes variables partagées...
  - ...en exécutant plein de CS au même endroit, pas besoin de les transférer !

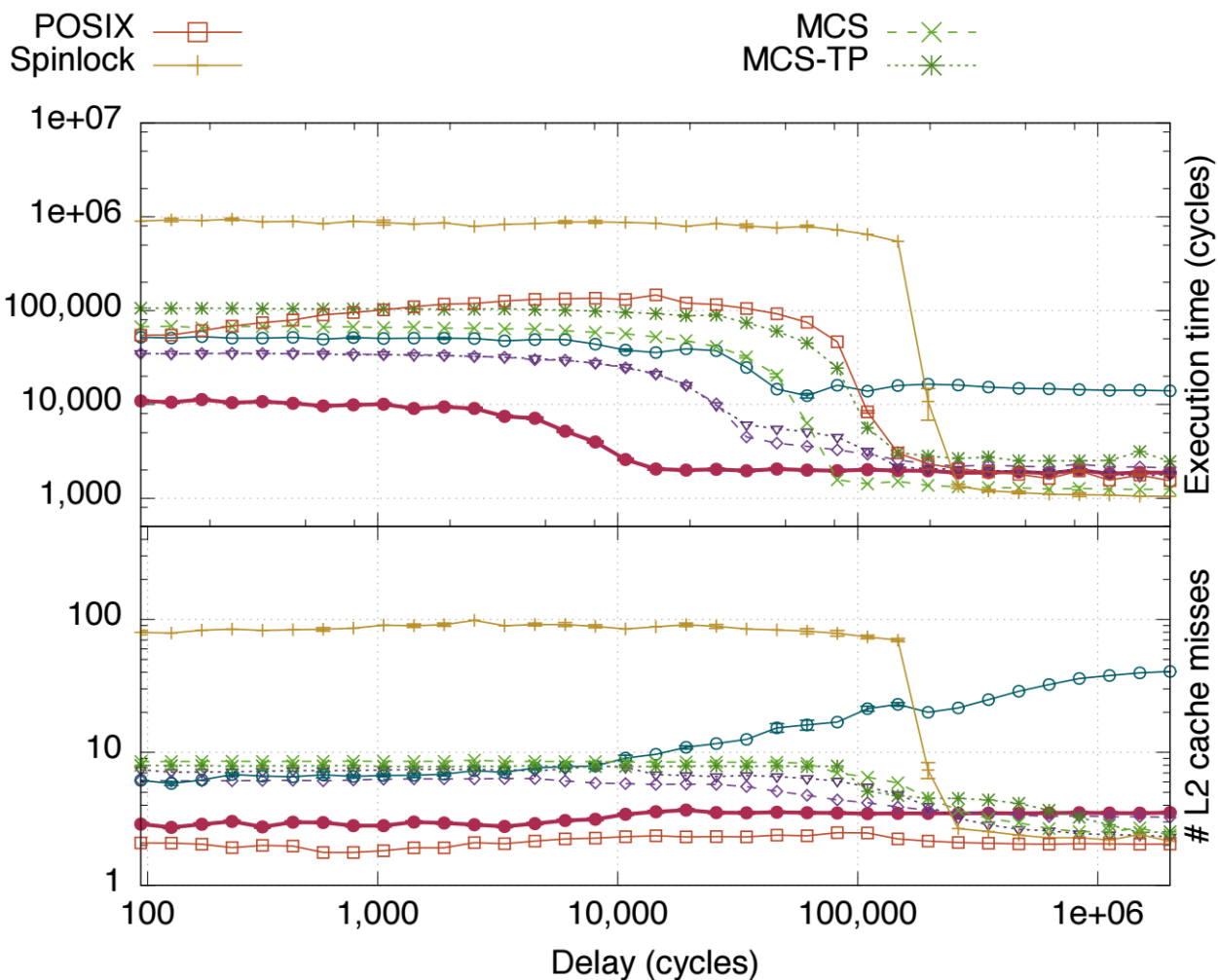
# VERROUS « À COMBINA TEUR »

- Autre avantage des verrous « à combine teur » :
  - CS protègent généralement les mêmes variables partagées...
  - ...en exécutant plein de CS au même endroit, pas besoin de les transférer !
  - **Localité des données améliorées !**

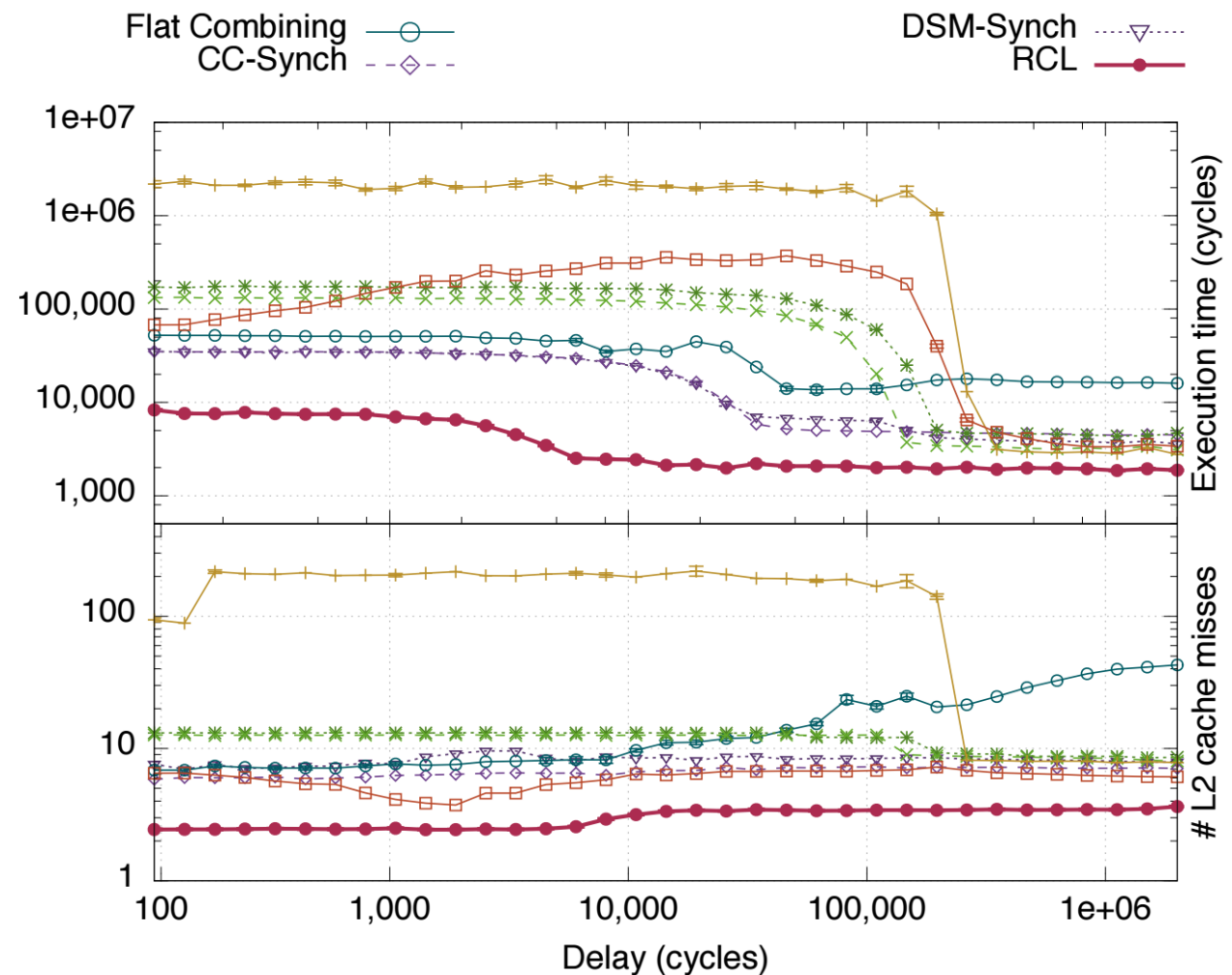


# VERROUS « À COMBINEUR »

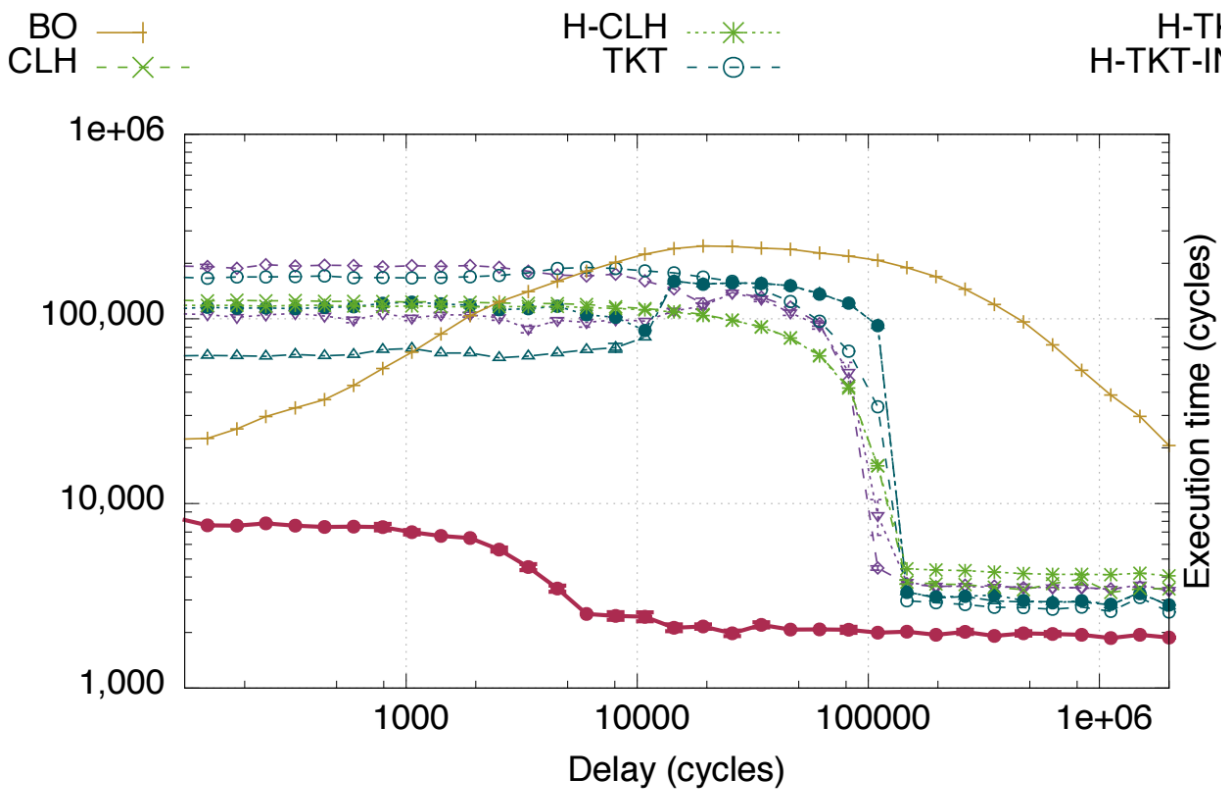
- Autre avantage des verrous « à combineur » :
  - CS protègent généralement les mêmes variables partagées...
  - ...en exécutant plein de CS au même endroit, pas besoin de les transférer !
  - **Localité des données améliorées !**
- Algorithmes existants : Flat Combining, Remote Core Locking, CC-Synch/DSM-Synch
  - Hendler et al, « Flat combining and the synchronization-parallelism tradeoff », SPAA '10
  - Fatourou et al, « Revisiting the combining synchronization technique », PPOPP '12



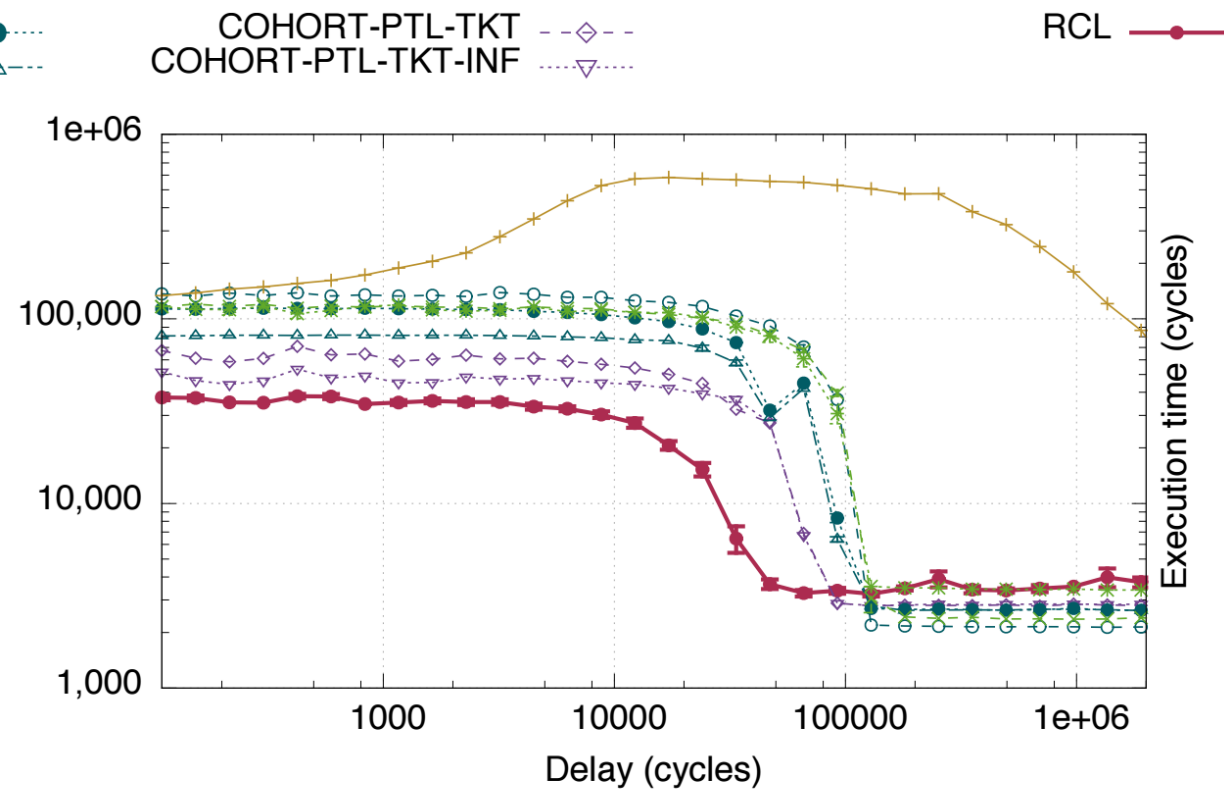
(a) One shared cache line per CS



(b) Five shared cache lines per CS



(a) Magnycours-48



(b) Niagara2-128

# VERROUS : CONCLUSION

- Plein d'algos... Et on n'a pas tout vu ! E.g., verrous à ticket

# VERROUS : CONCLUSION

- Plein d'algos... Et on n'a pas tout vu ! E.g., verrous à ticket
- Certains algos récents... Domaine de recherche actif !
  - Si vous voulez faire une thèse 😊

# VERROUS : CONCLUSION

- Plein d'algos... Et on n'a pas tout vu ! E.g., verrous à ticket
- Certains algos récents... Domaine de recherche actif !
  - Si vous voulez faire une thèse 😊
- Perfs complexes, dépendent du hardware, des cas d'utilisation...
  - Manque une bonne modélisation...
  - ...à nouveau, si vous voulez faire une thèse 😊

# VERROUS : CONCLUSION

- Plein d'algos... Et on n'a pas tout vu ! E.g., verrous à ticket
- Certains algos récents... Domaine de recherche actif !
  - Si vous voulez faire une thèse 😊
- Perfs complexes, dépendent du hardware, des cas d'utilisation...
  - Manque une bonne modélisation...
  - ...à nouveau, si vous voulez faire une thèse 😊
- Si sections critiques réduites au max, et bon algo de verrou... Comment encore améliorer ?
  - Si algo le permet, algorithmique lock-free !
  - Voir prochain cours...