Calcul Haute Performance

Quentin Meunier - Adrien Cassagne

Multi-Thread et Synchronisation (1)

Quentin Meunier

Laboratoire d'Informatique de Paris 6
Équipe Alsoc
4 Place Jussieu, 75252 Paris, France

Polytech, EISE5, HPC

Généralités sur les threads

Rappels

- Un thread = un fil d'exécution
- Modèle de programmation à mémoire partagée plus intuitif que le passage de messages
- Transition plus facile depuis un système monocoeur
- Moyen de paralléliser un programme : avoir plusieurs threads
- Les threads d'un programme peuvent exécuter la même fonction ou des fonctions différentes
- Tous les threads partagent le même espace d'adressage (i.e. une même adresse désigne le même emplacement mémoire pour tous les threads)
- $\Rightarrow$ Certaines structures de données sont donc partagées
- $\Rightarrow$ Nécessité de synchroniser les accès sur ces structures à l'aide de primitives pour faire des opérations qui ont du sens à haut niveau

Motivation

- Exemple : Comment incrémenter une variable partagée de manière concurrente (ou *thread safe*) ?
- Pourquoi l'approche naïve ne marche pas, i.e. simplement faire a++ dans chaque thread ?
- a++ est traduit en un `load`, un `add` et un `store` : 2 opérations mémoire

```
# T0
# $10 contient l'adresse
# de a
lw     $8, 0($10)
addiu  $8, $8, 1


sw     $8, 0($10)
```

```
# T1
# $12 contient l'adresse
# de a

lw     $8, 0($12)
addiu  $8, $8, 1

sw     $8, 0($12)
```

- Après exécution des deux threads, une seule incrémentation a eu lieu

Types de synchronisation

- Types de synchronisation les plus utilisés : exclusion mutuelle et barrières (selon mon expérience en tout cas)
- Exclusion mutuelle
  - Introduit la notion de **section critique** : portion de code exécutée à un moment donné par au plus un thread
  - Une section critique est typiquement protégée par une variable de synchronisation
  - Permet facilement l'incrémentation de variables partagées de manière concurrente
- Synchronisation par barrière
  - Une barrière correspond à un endroit du code qu'il faut que tous les threads atteignent avant qu'ils puissent tous continuer à s'exécuter

Approche générale de la synchronisation

Composantes de la synchronisation

- Acquisition
  - Acquérir le droit à la synchronisation (entrée d'une section critique, aller au-delà d'un évènement)
- Algorithme d'attente
  - Attendre que la synchronisation devienne disponible si elle ne l'est pas
- Libération
  - Permettre à d'autres threads d'acquérir le droit à la synchronisation
- Remarque : l'algorithme d'attente est indépendant de la synchronisation

Les algorithmes d'attente

Caractéristiques

- Bloquant
  - Les threads en attente sont désordonnancés, seul le thread qu'il faut est réveillé avec le lock pris quand c'est son tour
  - Surcout élevé
  - Permet au coeur de faire autre chose
- Attente active
  - Les threads en attente testent en permanence une case mémoire jusqu'à ce que sa valeur change
  - Le thread libérant le verrou écrit la case
  - Surcout plus faible, mais consomme les ressources du coeur
  - Peut provoquer du traffic réseau
- L'attente active est meilleure quand :
  - Le surcout d'ordonnancement est plus grand que le temps d'attente estimé
  - Les ressources processeurs ne sont pas nécessaires pour d'autres tâches
  - Le blocage par l'ordonnanceur n'est pas possible (ex : dans certaines parties du noyau de l'OS)
- Méthodes hybrides : attente active puis blocage

Le problème de l'exclusion mutuelle

- Empêcher l'accès concurrent à plus d'un thread à une section de code donnée

Pourquoi est-ce un problème ?

- Pour la même raison que celle qui nous empêchait d'incrémenter une variable partagée avec a++;

```
lock_acquire:
    lw   $8, 0($4) # $4 contient l'adresse du lock
    bne  $8, $0, lock_acquire
    li   $8, 1
    sw   $8, 0($4)
    jr   $31
```

- La lecture (test) et l'écriture (prise du lock) ne sont pas atomiques
- Deux threads peuvent prendre le lock en même temps

Le problème de l'exclusion mutuelle

Types d'implémentation des solutions

- Uniquement logicielle
  - Algorithmes complexes (ex : Dekker, Peterson)
  - Contraintes supplémentaires (ex : le nombre de threads doit être connu à l'avance)
- Avec l'aide du matériel
  - Pas standard : différents types de primitives de synchronisation selon les architectures
  - Exemple de primitives : Test&Set, Compare-and-Swap, LL/SC

Système multi-coeurs à mémoire partagée

- Même avec des techniques de synchronisation, problème sémantique : quel est le comportement en cas de lectures / écritures concurrentes ?
- Exemple :

Initialement, le pointeur `head` et les pointeurs `my_task` valent NULL

T0

```
while (have_task(task_list)) {
  task = get_task(task_list);
  task->data = ...;
  add(task_queue, task);
}
head = task_queue->head;
```

T1, T2, ..., Tn-1

```
while (my_task == NULL) {
  <begin critical section>
  if (head != NULL) {
    my_task = head;
    head = head->next;
  }
  <end critical section>
}
my_data = my_task->data;
```

- **Quelles sont les valeurs possibles pour `my_data` ?**

Modèle de consistance

- Intuitivement, une lecture devrait retourner la "dernière" valeur écrite
- Difficile à définir de façon précise
- 3 ordres différents pour les opérations mémoire :
  - Ordre du programme : ordre des instructions écrites par le programmeur
  - Ordre d'exécution : ordre des références mémoire – différent du précédent, par exemple à cause des optimisations du compilateur et des exécutions *out-of-order*
  - Ordre perçu : ordre, propre à un thread, de la perception des opérations mémoire des autres threads

Définition

- Un **modèle de consistance** est une spécification des comportements de la mémoire autorisés, tels que vus par les threads ($\Leftrightarrow$ coeurs) : ordre perçu
- Il s'agit d'une spécification sur la vue du programmeur $\Rightarrow$ Pas d'hypothèse sur le matériel

Impact du modèle de consistance

- Sur le logiciel de haut-niveau (le programmeur)
  - Validité de la synchronisation entre les threads
- Sur le logiciel de bas-niveau (le programmeur ou le compilateur)
  - Validité du code de synchronisation bas-niveau (ex : bibliothèque de locks)
- Sur le matériel
  - Réordonnancement possible ou non des instructions dans le processeur, des transactions sur le bus, des accès mémoire dans les caches ou la mémoire
  - Spécification du protocole de cohérence de cache
- Sur la performance : supporter un modèle de consistance fort peut être plus complexe en matériel (ou alors interdire certaines optimisations $\Rightarrow$ dans ce cas plus simple), coute plus cher en temps, et interdit des optimisations du compilateur
- Sur la portabilité : le même code n'est pas valide sur toutes les architectures de processeur

Le modèle de consistance séquentiel

Définition générale

- Un système supporte la **consistance séquentielle** si toutes les opérations mémoire apparaissent s'effectuer atomiquement, et les opérations émises par un seul processeur apparaissent s'effectuer dans l'ordre du programme

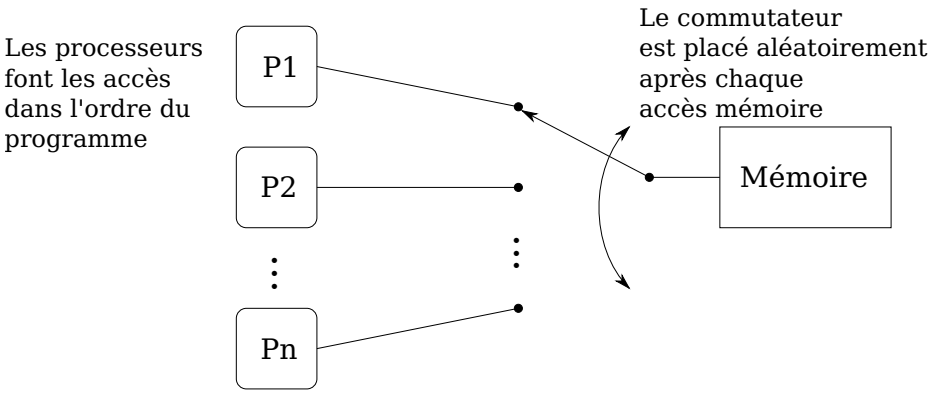
- Pour un programme séquentiel, il suffit de maintenir les dépendances de données pour assurer la consistance séquentielle
- Le compilateur et le matériel peuvent donc exécuter une liste d'instructions dans un ordre différent de l'ordre du programme
- Exemple : un cache *write-through* peut envoyer un miss sans attendre la fin de l'écriture précédente (et les deux peuvent se doubler) ; il suffit de bloquer le miss si une écriture est en cours sur la même ligne.

Le modèle de consistance séquentiel

Définition pour un système multicoeur

- Le résultat de n'importe quelle exécution est le même que si toutes les opérations de tous les threads étaient effectuées dans un ordre séquentiel (quelconque), et les opérations de chaque thread individuellement apparaissent dans cette séquence dans l'ordre spécifié par son programme.

- Exemple d'un système simple qui garantirait la consistance séquentielle : système sans cache et une seule mémoire avec un "commutateur"



- Ordre total obtenu en entrelaçant les accès des différents coeurs
- Les opérations mémoire de tous les threads aparaissent comme si elles étaient atomiques les unes par rapport aux autres
- Conserve l'intuition du programmeur

Le modèle de consistance séquentiel

Remarque 1

- L'ordre des accès mémoire n'est pas nécessairement le même entre 2 exécutions

Remarque 2

- Le modèle de consistance mémoire séquentiel ne protège pas contre les **race conditions**

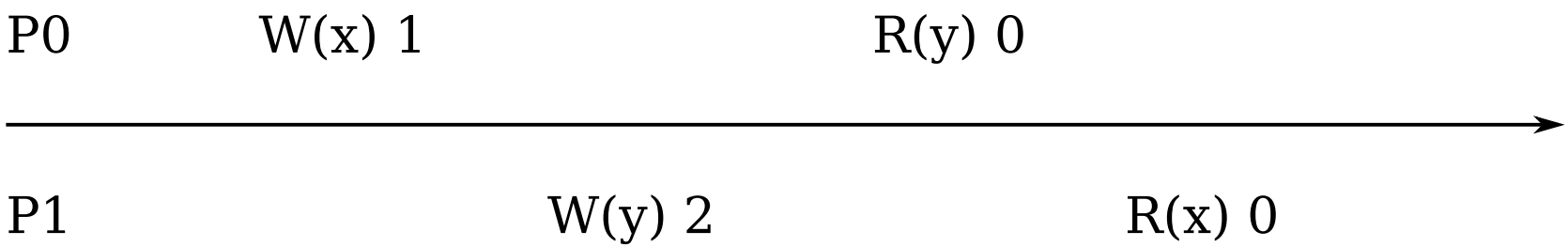
Définition

- Une **race condition** est une erreur dans l'écriture d'un programme, suite à laquelle le résultat/bon fonctionnement d'une application dépend de l'entrelacement de plusieurs accès mémoire concurrents

- Exemple de *race condition* : incrémentation d'une variable partagée sans précaution (exemple vu en introduction)
- N'importe quel entrelacement dans lequel le 2^e load a lieu avant le 1^{er} store donnera un résultat incorrect : les opérations mémoire sont atomiques, mais pas la séquence des 2
- Remarque : la plupart du temps, le résultat de l'exécution sera correct

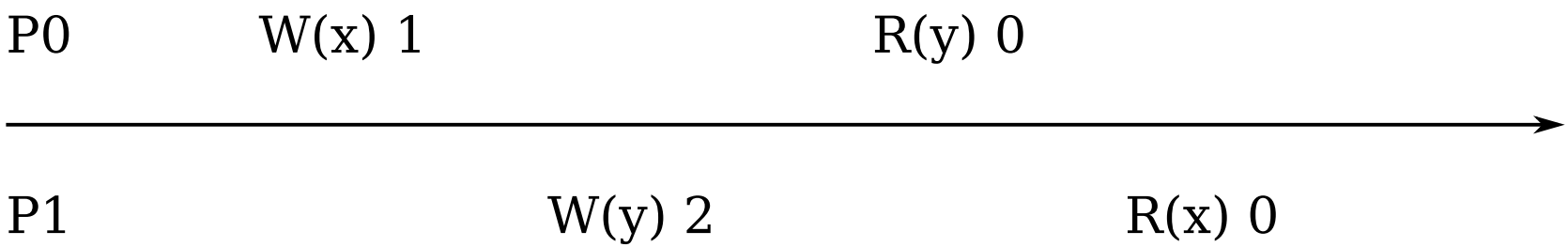
Exemple

- L'exécution suivante est-elle valide pour le modèle de consistance séquentiel ? (initialement `x = y = 0`)



Exemple

- L'exécution suivante est-elle valide pour le modèle de consistance séquentiel ? (initialement `x = y = 0`)



- Pas d'ordre séquentiel équivalent à un entrelacement :
  - `R(y) 0` doit être avant `W(y) 2`
  - Par ailleurs, `W(x) 1` doit être avant `R(y) 0` et `W(y) 2` avant `R(x) 0` (ordre du programme) $\Rightarrow$ `W(x) 1` doit être avant `R(x) 0`
  - Or `R(x) 0` doit être avant `W(x) 1` : contradiction
- Les exécutions `(R(x) 0, R(y) 2)`, `(R(x) 1, R(y) 0)` et `(R(x) 1, R(y) 2)` sont valides pour le modèle séquentiel

Implémentation de la consistance séquentielle

- Maintenir la consistance séquentielle dans une architecture multicoeur est difficile
  - Pour faire simple : la mémoire est constituée d'une hiérarchie de caches distribués
- De plus, cela ajoute beaucoup d'indirections et d'attentes qui empêchent une exécution efficace (i.e. ce serait beaucoup trop lent)
- En pratique, les architectures multicoeur n'implémentent pas la consistance séquentielle
- $\Rightarrow$ Il existe un certain nombre de modèles de consistance plus faibles que le modèle séquentiel
- $\Rightarrow$ Les synchronisations doivent en conséquence être adaptées avec des primitives spécifiques
- $\Rightarrow$ Plus généralement, le problème est partiellement repoussé vers le logiciel, qui ne peut plus en faire abstraction

Le modèle Local

Définition

- L'ordre perçu pour les opérations mémoire locales à un thread est l'ordre du programme

- Ne dit rien sur les opérations mémoire des autres threads
- Tous les modèles (utiles) sont plus forts que le modèle local et plus faibles que le modèle séquentiel

Le modèle Slow

Définition

- Les lectures doivent retourner une valeur précédemment écrite. Une fois qu'une valeur à été lue, aucune valeur précédemment écrite par le thread qui a écrit la valeur lue ne peut être retournée. Les écritures d'un thread doivent être visibles immédiatement à lui-même.

- Exemple :

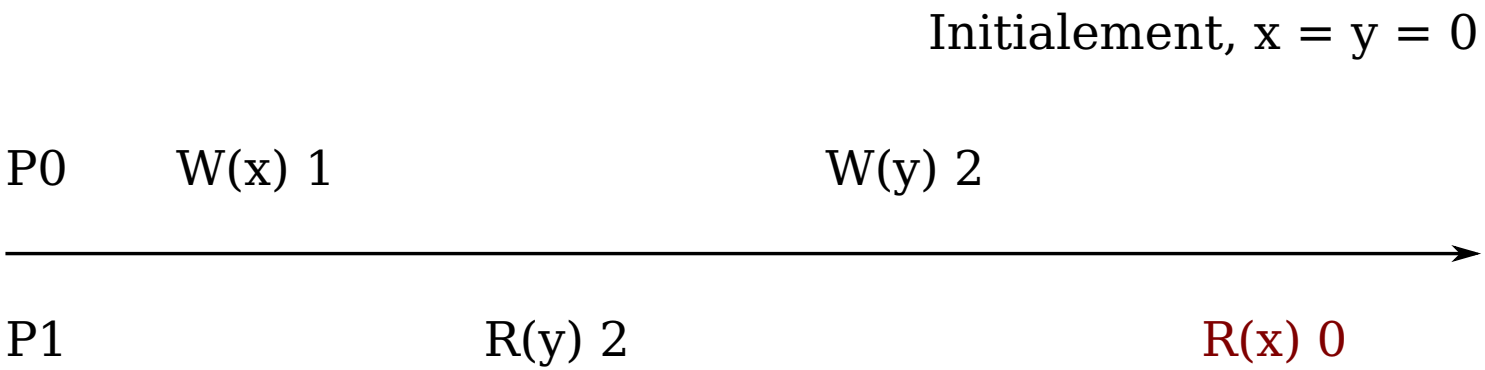
- $\Rightarrow$ Valide pour Local mais invalide pour Slow

Le modèle PRAM

Définition

- Les écritures effectuées par un même thread sont vues par les autres threads dans l'ordre dans lequel elles ont été effectuées, mais les écritures venant de threads différents peuvent être vues dans des ordres différents par différents threads

- Exemple :



- $\Rightarrow$ Valide pour Slow mais invalide pour PRAM

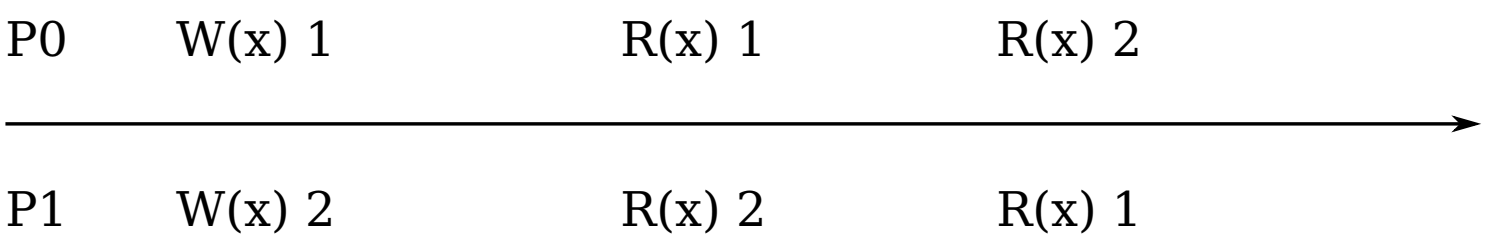
Le modèle Cache

Définition

- Toutes les écritures au même emplacement mémoire sont effectuées dans un ordre séquentiel

- Ne dit rien pour des écritures à des emplacements mémoire différents
- Exemple :

Initialement, x = y = 0



- $\Rightarrow$ Valide pour Slow mais invalide pour Cache

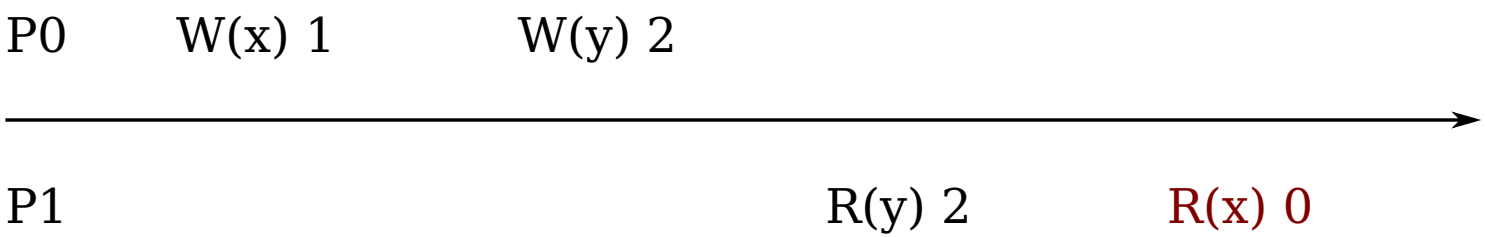
Le modèle Processeur

Définition

- L'exécution est consistente PRAM et toutes les écritures au même emplacement sont vues dans le même ordre par tous les threads

- Combinaison de PRAM et Cache
- Exemple :

Initialement, x = y = 0



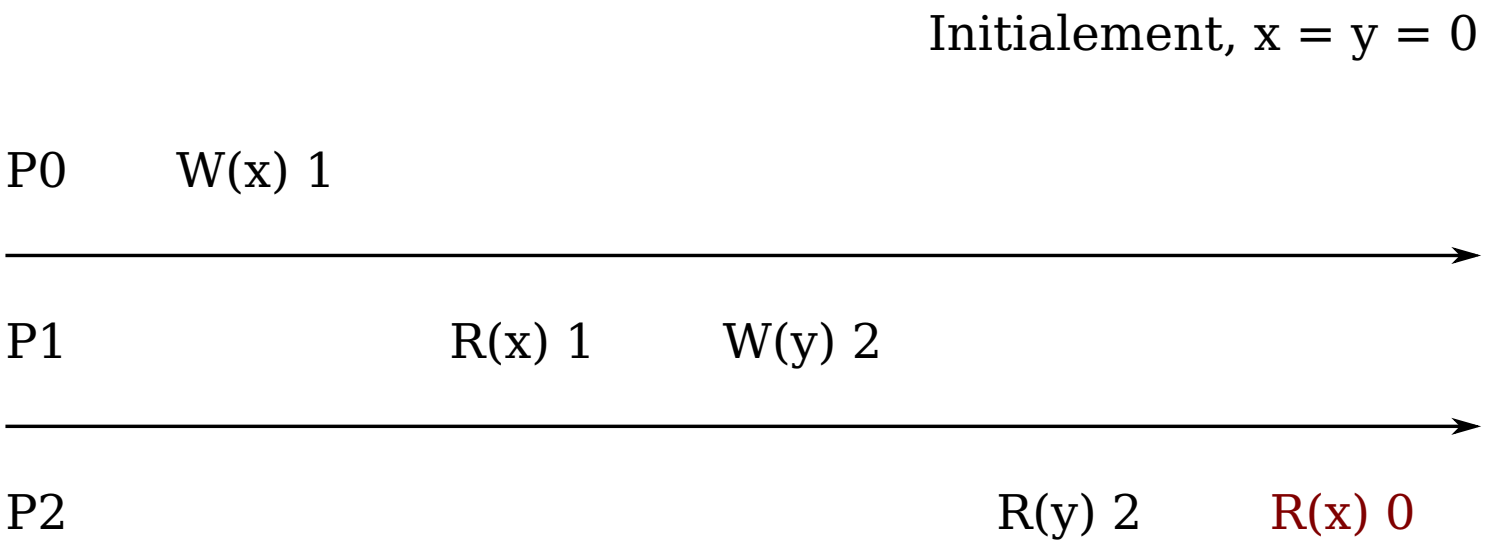
- $\Rightarrow$ Valide pour cache (les emplacements mémoire sont différents) mais invalide pour PRAM (et donc Processeur)

Le modèle Causal

Définition

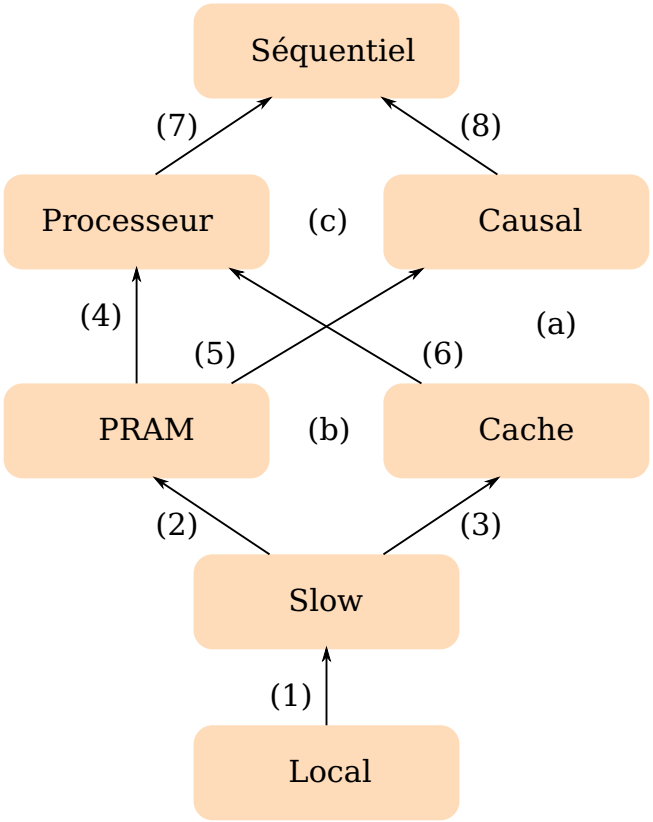
- Pour chaque thread, les opérations de ce thread et toutes les écritures "connues" de ce thread (dépendances RAW) apparaissent pour ce thread dans un ordre qui respecte la causalité.

- Exemple :



- ⇒ Valide pour Processeur (PRAM car pas le même thread + Cache car pas la meme addresse), mais invalide pour causal (et donc séquentiel)

Graphe de relations entre ces modèles



Autre approche : Modèle “Weak”

Idée

- Les accès aux variables globales de synchronisation (locks) sont fortement ordonnés : introduction de la notion de *variable de synchronisation*

Définition du modèle de consistence Weak

- Tous les accès aux variables de synchronisation sont vus dans le même ordre (ordre séquentiel) par tous les threads
- Tous les autres accès peuvent être vus dans des ordres différents par différents threads
- L'ensemble des variables accédées entre 2 opérations de synchronisation doit être le même pour tous les threads (i.e. une opération de synchronisation ne peut pas être réordonnée vis-à-vis d'autres opérations mémoire)

Modèle "Weak"

Conséquences

- Il ne peut pas y avoir d'accès à une variable de synchronisation tant qu'il y a des opérations d'écriture en cours
- Il ne peut pas y avoir de nouvelles opérations de lecture et d'écriture lancées quand une opération de synchronisation a été commencée et n'est pas terminée

En résumé

- Les accès aux variables de synchronisation empêchent le reordonnancement et la consistance séquentielle est assurée pour ces variables
- $\Rightarrow$ C'est ça qui est souvent utilisé en pratique car cela ne fait pas d'hypothèses sur le modèle implémenté par le matériel, et les synchronisations sont à la charge du programmeur
- $\Rightarrow$ Existence de primitives spécifiques : on appelle ces instructions des instructions de "barrière mémoire"
- Exemple : `sync` en mips, `mfence` en x86
- Dans les exemples de la suite du cours, l'hypothèse du modèle séquentiel sera implicite, mais il faut garder en tête qu'il faut souvent ajouter des appels à ces primitives "à la main"

Modèle “Weak” : exemple

Producer P0

```
lock(l);
read(status);
write(data0);
write(data1);
write(data2);
write(status);
unlock(l);
```

Pas de réordonnancement

Réordonnancement possible

Pas de réordonnancement

Consumer P1

```
lock(l);
read(status);
read(data0);
read(data1);
read(data2);
write(status);
unlock(l);
```

Pas de réordonnancement

Réordonnancement possible

Pas de réordonnancement

Instruction Test & Set

- La valeur de la case mémoire est lue dans un registre (comme pour un `load`)
- La constante 1 est écrite dans la case mémoire
- Peut être utilisé pour réaliser un lock :
  - Prise du lock réussie si la valeur chargée dans le registre est 0
  - Pour relâcher le lock, il suffit d'écrire la valeur 0

```
lock_acquire:
    tas $8, 0($4) # n'existe pas en Mips
    bne $8, $0, lock_acquire
    jr $31
```

Instruction Compare-and-Swap

- Sémantique

```
bool cas(int * addr, int old_val, int new_val) {
    <atomic>
    if (*addr == old_val) {
        *addr = new_val;
        return true;
    }
    else {
        return false;
    }
    <end atomic>
}
```

Exercice

- Donner le code des fonctions `lock_acquire(int * lock)` et `lock_release(int * lock)` en C en utilisant la primitive cas

Instructions LL/SC

- LL(x) (*Load Link*) : Lecture à l'adresse x avec un effet de bord
- SC(x, v) (*Store Conditional*) : Écriture de la valeur v à l'adresse x s'il n'y a pas eu d'autre écriture (éventuellement, à l'adresse x) ou de SC depuis le LL(x) fait par ce processeur
  - Renvoie succès (1) ou échec (0) pour en informer le processeur
  - En Mips, `sc $x, 0($y)`, avec résultat dans `$x`

Exercice

- Donner le code des fonctions `lock_acquire` et `lock_release` en assembleur mips en utilisant les instructions LL/SC

Instruction Mips sync

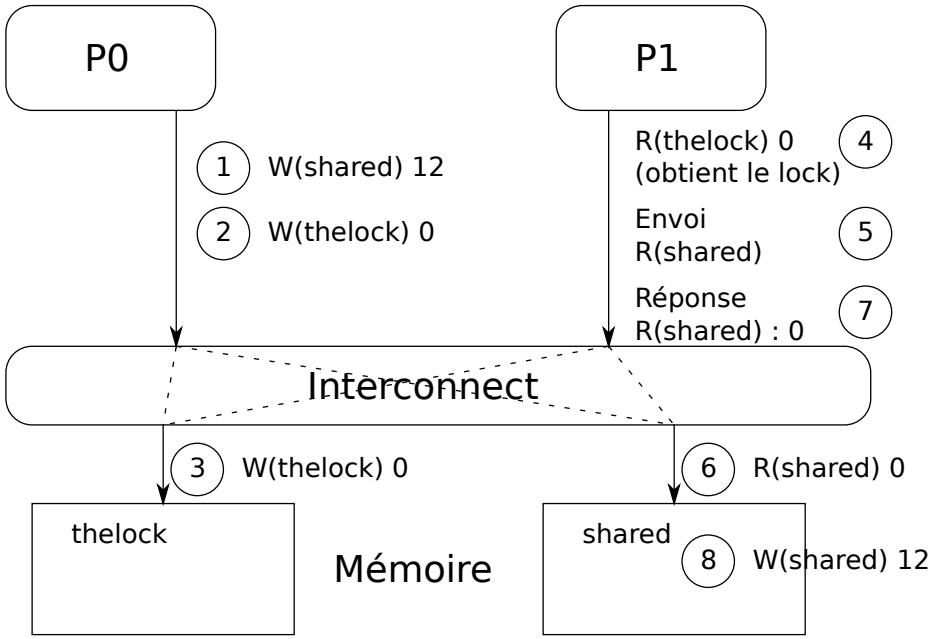
- Synchronise tous les accès mémoire en cours pour le processeur qui l'exécute
- $\Rightarrow$ Bloque l'exécution jusqu'à ce que toutes les écritures en cours soient terminées et visibles des autres coeurs

Exercice

- Montrer avec un exemple que la primitive `lock_release()` écrite précédemment peut nécessiter l'utilisation de `sync` dans certains modèles de consistance mémoire

Exemple de problème sans consistance séquentielle et sans instruction de barrière mémoire

- Hypothèse : architecture à mémoire distribuée




```
// T0
// initialement
// shared = 0
...
// section
// critique
shared = 12;
// relâche le lock
// fin section
// critique
*thelock = 0;
```

```
// T1
lock(thelock);
// entrée section
// critique :
// doit "voir"
// l'écriture de
// shared
mavar = shared;
```

API Posix

- Locks à attente passive : `mutex_lock`

Thread main

```
// Variables globales
pthread_mutex_t mutex;
int a = 0;
// Dans le main
  ...
  pthread_mutex_init(&mutex, NULL);
  // Création des threads
```

Phase parallèle

```
...
pthread_mutex_lock(&mutex);
a += 1;
pthread_mutex_unlock(&mutex);
```

- Locks à attente active : `spin_lock`

Thread main

```
// variables globales
pthread_spinlock_t lock;
int a = 0;
// Dans le main
  ...
  pthread_spin_init(&lock,
       PTHREAD_PROCESS_PRIVATE);
  // Création des threads
```

Phase parallèle

```
...
pthread_spin_lock(&lock);
a += 1;
pthread_spin_unlock(&lock);
```

Locks à ticket

Problème courant des spinlocks

- Souvent, les implémentations de spinlock ne respectent pas une prise de lock correspondant à l'ordre d'appel à la fonction `spin_lock`

Idée des locks à ticket

- Spin lock qui conserve l'ordre FIFO entre la demande et l'obtention du lock
- Besoin de 2 variables

```
typedef struct {
  int now_serving;
  int next_ticket;
} tlock;
...
ticket_init(tlock * lock) {
  lock->now_serving = 0;
  lock->next_ticket = 0;
}
```

```
void ticket_lock(tlock * lock) {
    // repose sur la primitive
    // atomique fetch_and_inc
    int my_ticket = fetch_and_inc(&lock->
        next_ticket);
    while (my_ticket != lock->now_serving);
}

void ticket_unlock(tlock * lock) {
    lock->now_serving += 1;
}
```

Locks MCS

Problème avec les ticket locks

- Lors d'un relâchement de lock, une invalidation (ou une mise à jour) est envoyée à tous les caches qui sont en attente du lock
- $\Rightarrow$ Sérialisation de tous les miss au niveau du cache L2 : ne passe pas bien à l'échelle quand on augmente le nombre de coeurs

Idée des locks MCS

- Chaque thread boucle sur une variable privée et non partagée

```
typedef struct _node {
  struct _node * next;
  bool is_locked;
} mcs_node;

typedef struct {
  mcs_node * queue;
} mcs_lock;
```

```
void lock(mcs_lock * lock, mcs_node * mynode) {
  mynode->next = NULL;
  // opération fetch_and_store atomique
  mcs_node * pred = fetch_and_store(&lock->queue
       , mynode);
  if (pred != NULL) {
    mynode->is_locked = true;
    pred->next = mynode;
    while (mynode->is_locked);
  }
}
```

Locks MCS (suite)

```
void unlock(mcs_lock * lock, mcs_node
      * mynode) {
  if (mynode->next == NULL) {
    // compare_and_swap
    if (cas(&lock->queue, mynode,
        NULL)) {
      // lock remis à NULL
      return;
    }
    else {
      // un thread s'est enregistré
      // mais n'avait pas encore
      // affecté le champ next au
      // moment du test
      while (mynode->next == NULL);
    }
  }
  mynode->next->is_locked = false;
}
```

Remarques

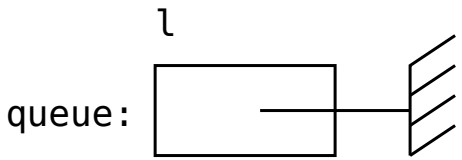
- Nécessite un mécanisme d'allocation de noeuds (il peut s'agir de variables en pile – un des rares cas où la sémantique de partage de pile est utile)
- Les primitives `fetch_and_store` et `compare_and_swap` peuvent être réalisées à partir des instructions LL/SC
- Question subsidiaire : où faut-il placer des barrières mémoire pour garantir un fonctionnement correct même avec un modèle de consistance slow ?

Locks MCS : Exemple

```
// Thread 0
mcs_node mynode_0;
lock(l, &mynode_0);
.
.
.
.
unlock(l, &mynode_0);
.
.
```

```
// Thread 1
.
.
mcs_node mynode_1;
lock(l, &mynode_1);
.
.
.
unlock(l, &mynode_1);
.
```

```
// Thread 2
.
.
.
.
mcs_node mynode_2;
lock(l, &mynode_2);
.
.
unlock(l, &mynode_2);
```

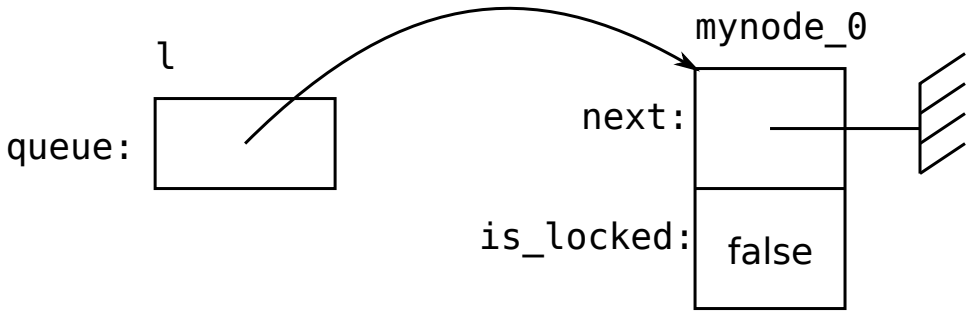


Locks MCS : Exemple

```
// Thread 0
mcs_node mynode_0;
lock(l, &mynode_0);
.
.
.
unlock(l, &mynode_0);
.
.
```

```
// Thread 1
.
.
mcs_node mynode_1;
lock(l, &mynode_1);
.
.
.
unlock(l, &mynode_1);
.
```

```
// Thread 2
.
.
.
.
mcs_node mynode_2;
lock(l, &mynode_2);
.
.
unlock(l, &mynode_2);
```

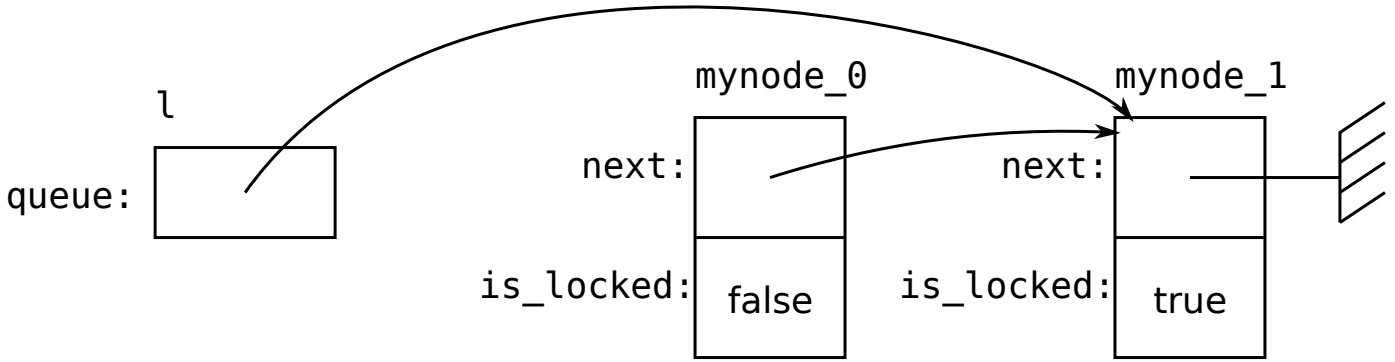


Locks MCS : Exemple

```
// Thread 0
mcs_node mynode_0;
lock(l, &mynode_0);
.
.
.
.
unlock(l, &mynode_0);
.
.
```

```
// Thread 1
.
.
mcs_node mynode_1;
lock(l, &mynode_1);
.
.
.
unlock(l, &mynode_1);
.
```

```
// Thread 2
.
.
.
.
mcs_node mynode_2;
lock(l, &mynode_2);
.
.
unlock(l, &mynode_2);
```

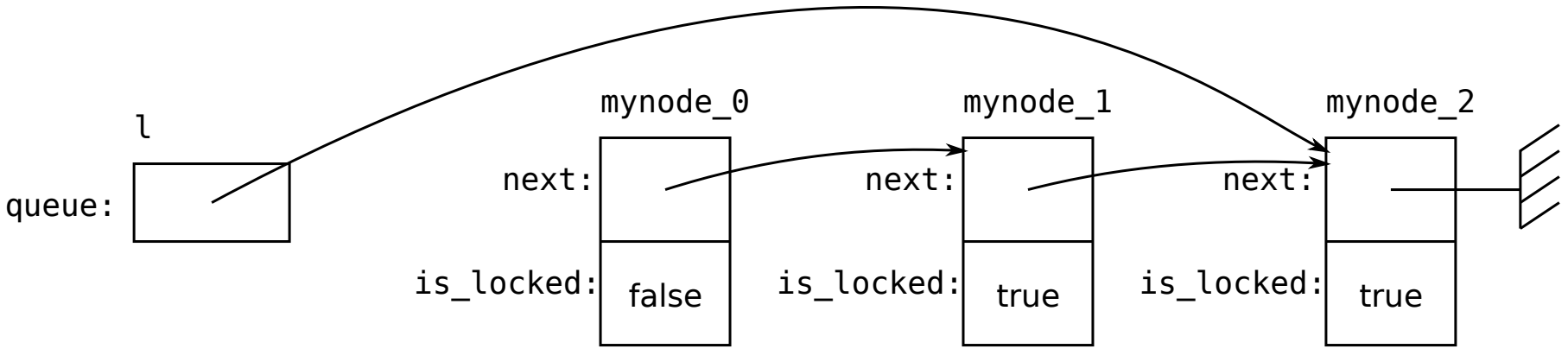


Locks MCS : Exemple

```
// Thread 0
mcs_node mynode_0;
lock(l, &mynode_0);
.
.
.
.
unlock(l, &mynode_0);
.
.
```

```
// Thread 1
.
.
mcs_node mynode_1;
lock(l, &mynode_1);
.
.
.
unlock(l, &mynode_1);
.
```

```
// Thread 2
.
.
.
.
mcs_node mynode_2;
lock(l, &mynode_2);
.
.
unlock(l, &mynode_2);
```

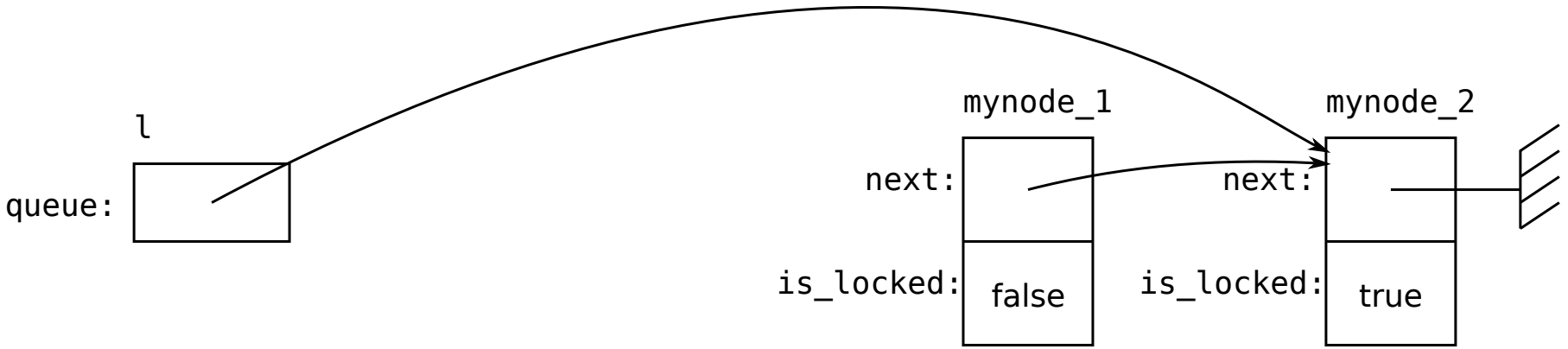


Locks MCS : Exemple

```
// Thread 0
mcs_node mynode_0;
lock(l, &mynode_0);
.
.
.
.
unlock(l, &mynode_0);
.
.
```

```
// Thread 1
.
.
mcs_node mynode_1;
lock(l, &mynode_1);
.
.
.
unlock(l, &mynode_1);
.
```

```
// Thread 2
.
.
.
.
mcs_node mynode_2;
lock(l, &mynode_2);
.
.
unlock(l, &mynode_2);
```

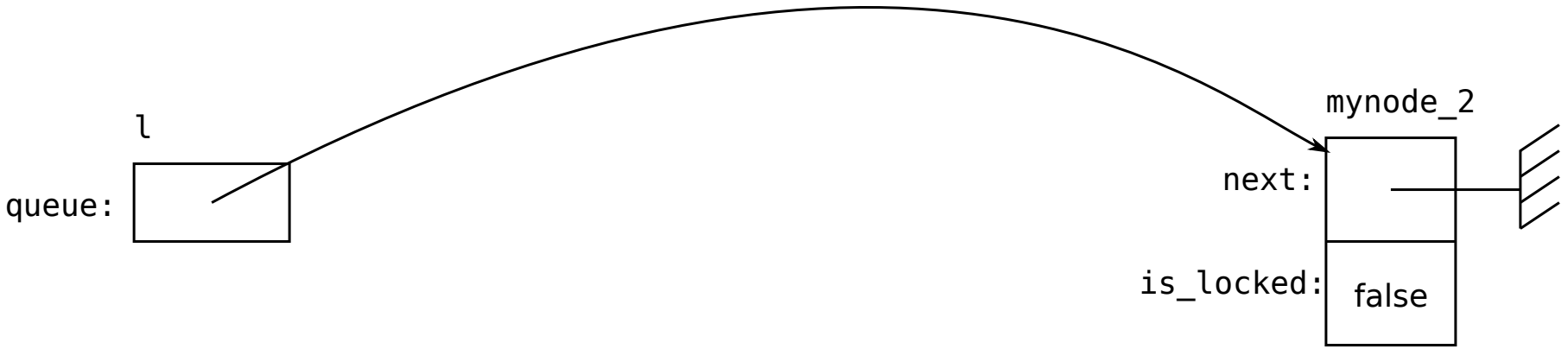


Locks MCS : Exemple

```
// Thread 0
mcs_node mynode_0;
lock(l, &mynode_0);
.
.
.
.
unlock(l, &mynode_0);
.
.
```

```
// Thread 1
.
.
mcs_node mynode_1;
lock(l, &mynode_1);
.
.
.
unlock(l, &mynode_1);
.
```

```
// Thread 2
.
.
.
.
mcs_node mynode_2;
lock(l, &mynode_2);
.
.
unlock(l, &mynode_2);
```

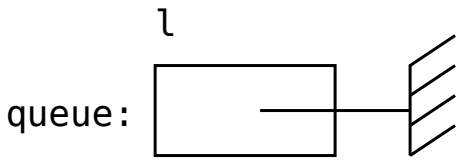


Locks MCS : Exemple

```
// Thread 0
mcs_node mynode_0;
lock(l, &mynode_0);
.
.
.
.
unlock(l, &mynode_0);
.
.
```

```
// Thread 1
.
.
mcs_node mynode_1;
lock(l, &mynode_1);
.
.
.
unlock(l, &mynode_1);
.
```

```
// Thread 2
.
.
.
.
mcs_node mynode_2;
lock(l, &mynode_2);
.
.
unlock(l, &mynode_2);
```



Locks et deadlocks

Hypothèses

- Soit une structure de liste chainée protégée par un lock
- Toute modification de la liste (insertion, suppression) requiert la prise du lock

```
typedef struct _node {
  struct _node * next;
  void * data;
} node;

typedef struct _liste {
  lock_t lock;
  node * head;
} liste;
```

```
void insert(liste * l, node * n) {
  lock(&l->lock);
  insert_nolock(l, n);
  unlock(&l->lock);
}

void remove(liste * l, node * n) {
  lock(&l->lock);
  remove_nolock(l, n);
  unlock(&l->lock);
}
```

Question

- Comment faire une fonction `void move(liste * l0, liste * l1, node * n)` qui déplace un élement d'une liste à une autre de manière atomique ?

Locks et deadlocks

Idée

- Locker `l0` puis locker `l1`

```
void move(liste * l0, liste * l1, node * n) {
  // précondition : n appartient à l0
  lock(&l0->lock);
  lock(&l1->lock);
  remove_nolock(&l0->lock, n);
  insert_nolock(&l1->lock, n);
  unlock(&l1->lock);
  unlock(&l0->lock);
}
```

Problème

- Que se passe-t-il si les deux appels suivants ont lieu en même temps : `move(l0, l1, n0);` et `move(l1, l0, n1);` ?

Locks et deadlocks

Idée

- Locker `l0` puis locker `l1`

```
void move(liste * l0, liste * l1, node * n) {
  // précondition : n appartient à l0
  lock(&l0->lock);
  lock(&l1->lock);
  remove_nolock(&l0->lock, n);
  insert_nolock(&l1->lock, n);
  unlock(&l1->lock);
  unlock(&l0->lock);
}
```

Problème

- Que se passe-t-il si les deux appels suivants ont lieu en même temps : `move(l0, l1, n0);` et `move(l1, l0, n1);` ?
- Solution : il faut locker selon un ordre total, par exemple selon l'ordre défini par l'adresse des structures

Barrières : de quoi a-t-on besoin ?

La structure barrière doit contenir (pour N threads)

- un lock
- le nombre de threads (valeur d'initialisation)
- un compteur (initialisé à la valeur d'initialisation)
- ...Let's go :

```
typedef struct _barrier_t {
  lock_t lock;
  int n; // num_threads
  int cpt; // init à 0
} barrier_t;
```

```
void barrier(barrier_t * b) {
  lock(&b->lock);
  b->cpt += 1;
  if (b->cpt == b->n) {
    b->cpt = 0;
    unlock(&b->lock);
  }
  else {
    unlock(&b->lock);
    while (b->cpt != b->n);
  }
}
```

Barrières : de quoi a-t-on besoin ?

La structure barrière doit contenir (pour N threads)

- un lock
- le nombre de threads (valeur d'initialisation)
- un compteur (initialisé à la valeur d'initialisation)
- ...Let's go :

```
typedef struct _barrier_t {
  lock_t lock;
  int n; // num_threads
  int cpt; // init à 0
} barrier_t;
```

```
void barrier(barrier_t * b) {
  lock(&b->lock);
  b->cpt += 1;
  if (b->cpt == b->n) {
    b->cpt = 0;
    unlock(&b->lock);
  }
  else {
    unlock(&b->lock);
    while (b->cpt != b->n);
  }
}
```

- Problème... ?

Barrières : de quoi a-t-on besoin ?

- Une solution : ajouter un champ `flag`

```
typedef struct _barrier_t {
  lock_t lock;
  int n; // num_threads
  int cpt; // init à 0
  int flag;
} barrier_t;
```

```
void barrier(barrier_t * b) {
  lock(&b->lock);
  if (b->cpt == 0) {
    // le premier arrivant reset le flag
    b->flag = 0;
  }
  b->cpt += 1;
  if (b->cpt == b->n) {
    unlock(&b->lock);
    // le dernier arrivant set le flag
    b->cpt = 0;
    b->flag = 1;
  }
  else {
    unlock(&b->lock);
    while (b->flag == 0);
  }
}
```

Barrières : de quoi a-t-on besoin ?

- Une solution : ajouter un champ `flag`

```
typedef struct _barrier_t {
  lock_t lock;
  int n; // num_threads
  int cpt; // init à 0
  int flag;
} barrier_t;
```

```
void barrier(barrier_t * b) {
  lock(&b->lock);
  if (b->cpt == 0) {
    // le premier arrivant reset le flag
    b->flag = 0;
  }
  b->cpt += 1;
  if (b->cpt == b->n) {
    unlock(&b->lock);
    // le dernier arrivant set le flag
    b->cpt = 0;
    b->flag = 1;
  }
  else {
    unlock(&b->lock);
    while (b->flag == 0);
  }
}
```

- Problème... ?

Barrières : de quoi a-t-on besoin ?

- Une solution : ajouter une variable privée par thread, `local` (initialisée à 0 pour tous les threads), qui définit la valeur du flag à attendre

```
typedef struct _barrier_t {
  lock_t lock;
  int n; // num_threads
  int cpt; // init à 0
  int flag; // init à 0
} barrier_t;
```

```
void barrier(barrier_t * b) {
  local = !local;
  lock(&b->lock);
  b->cpt += 1;
  if (b->cpt == b->n) {
    // le dernier arrivant set le flag
    unlock(&b->lock);
    b->cpt = 0;
    b->flag = local;
  }
  else {
    unlock(&b->lock);
    while (b->flag != local);
  }
}
```

Barrières : de quoi a-t-on besoin ?

- Une solution : ajouter une variable privée par thread, `local` (initialisée à 0 pour tous les threads), qui définit la valeur du flag à attendre

```
typedef struct _barrier_t {
  lock_t lock;
  int n; // num_threads
  int cpt; // init à 0
  int flag; // init à 0
} barrier_t;
```

```
void barrier(barrier_t * b) {
  local = !local;
  lock(&b->lock);
  b->cpt += 1;
  if (b->cpt == b->n) {
    // le dernier arrivant set le flag
    unlock(&b->lock);
    b->cpt = 0;
    b->flag = local;
  }
  else {
    unlock(&b->lock);
    while (b->flag != local);
  }
}
```

- Problème... ?

Barrières : de quoi a-t-on besoin ?

- Une solution : ajouter une variable privée par thread, `local` (initialisée à 0 pour tous les threads), qui définit la valeur du flag à attendre

```
typedef struct _barrier_t {
  lock_t lock;
  int n; // num_threads
  int cpt; // init à 0
  int flag; // init à 0
} barrier_t;
```

```
void barrier(barrier_t * b) {
  local = !local;
  lock(&b->lock);
  b->cpt += 1;
  if (b->cpt == b->n) {
    // le dernier arrivant set le flag
    unlock(&b->lock);
    b->cpt = 0;
    b->flag = local;
  }
  else {
    unlock(&b->lock);
    while (b->flag != local);
  }
}
```

- Problème... ?
- Pas cette fois-ci :-)

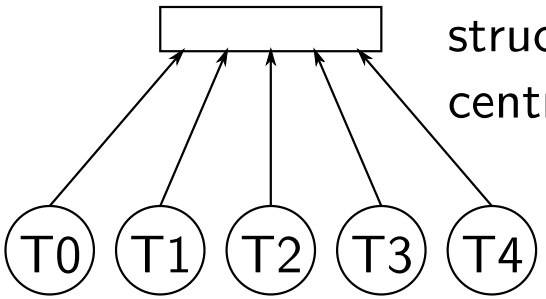
Barrières hiérarchiques

Problème posé par les barrières

- Contention élevée sur la structure de barrière
- Dès qu'un thread arrive, écriture ⇒ invalidation de tous les caches + miss
- Les barrières passent plutôt mal à l'échelle en termes de nombre de threads

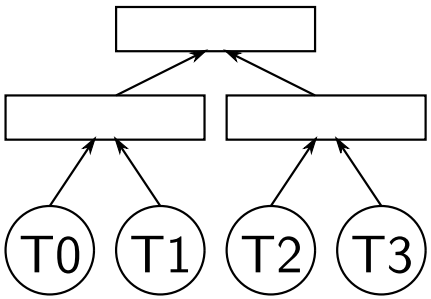
Idée pour limiter ce problème

- Utiliser une approche hiérarchique
- Particulièrement adapté sur les architectures NUMA



structure centralisée

structure hiérarchique distribuée



Kmeans

Description de l'algorithme

- Soit N points dans un espace de dimension D fini (typiquement un cube)
  - Par exemple, N = 10 000 et D = 3
- On cherche à agglomérer les points autour de C "clusters" (C < N)
  - Un cluster peut être vu comme un type de point particulier
  - Par exemple, C = 100
- Contraintes :
  - Un cluster doit être le centre de tous les points autour de lui
  - Chaque point doit appartenir au cluster le plus proche
- $\Rightarrow$ Processus itératif :
  - Calculer les points appartenant à chaque cluster
  - Recalculer les nouvelles coordonnées du cluster en fonction des nouveaux points y appartenant
  - Recommencer jusqu'à stabilité
- Dans l'implémentation considérée : variables entières

Kmeans : pseudo-code (1/2)

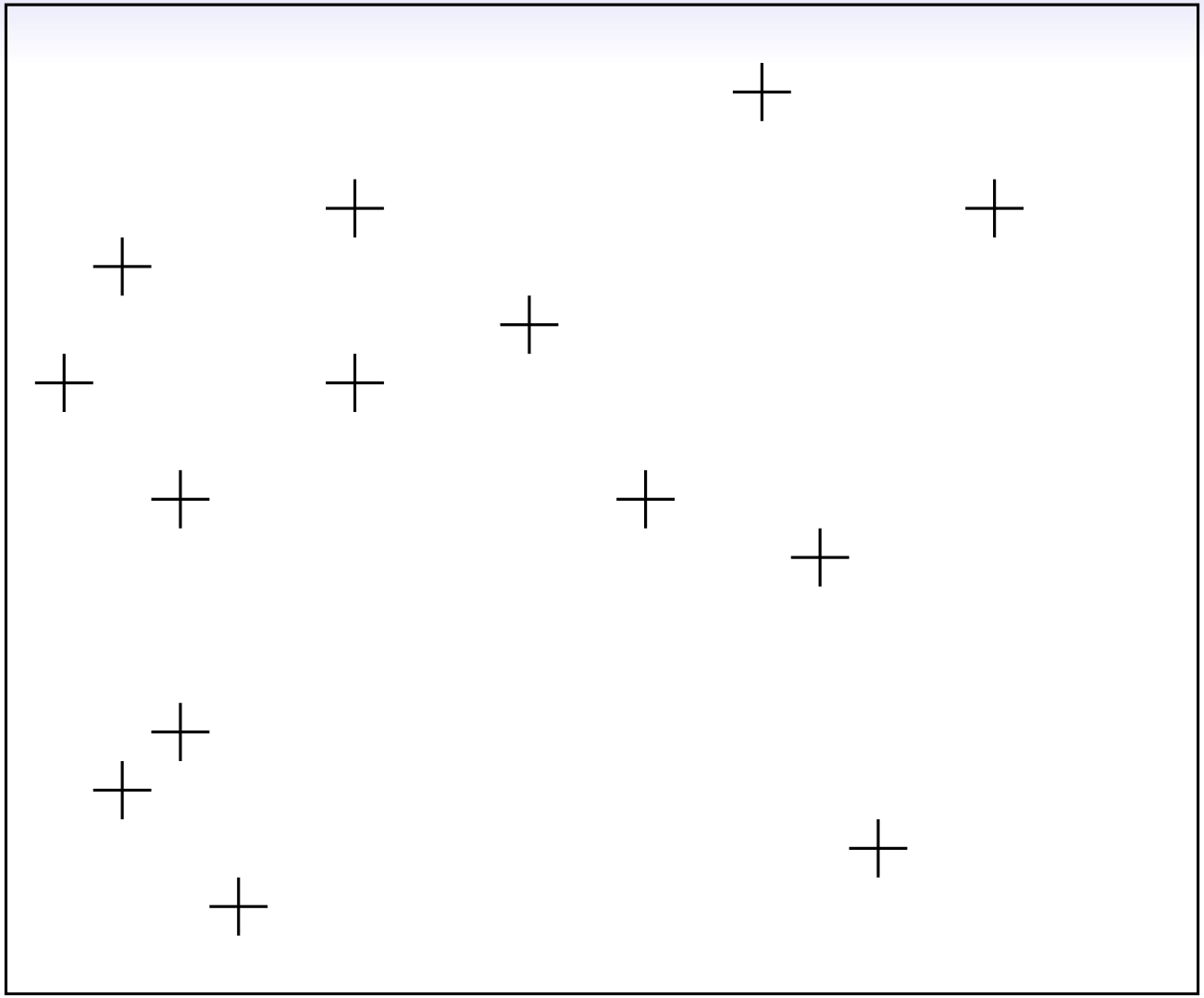
```
int ** points;         // points (num_points * dimension)
int ** clusters;       // clusters coordinates
int * point2cluster; // cluster to which a point belongs
...
  modified = true;
  while (modified) {
    modified = false;
    find_clusters();
    calc_means();
  }
...
void find_clusters(...) {
  ...
  for (int i = 0; i < num_points; i += 1) {
    for (int j = 0; j < num_clusters; j += 1) {
      if (distance(points[i], clusters[i]) < min) {
        min = distance(points[i], clusters[i]);
        min_idx = j;
      }
    }
    if (point2cluster[i] != min_idx) {
      point2cluster[i] = min_idx;
      modified = true;
    }
  }
}
```

Kmeans : pseudo-code (2/2)

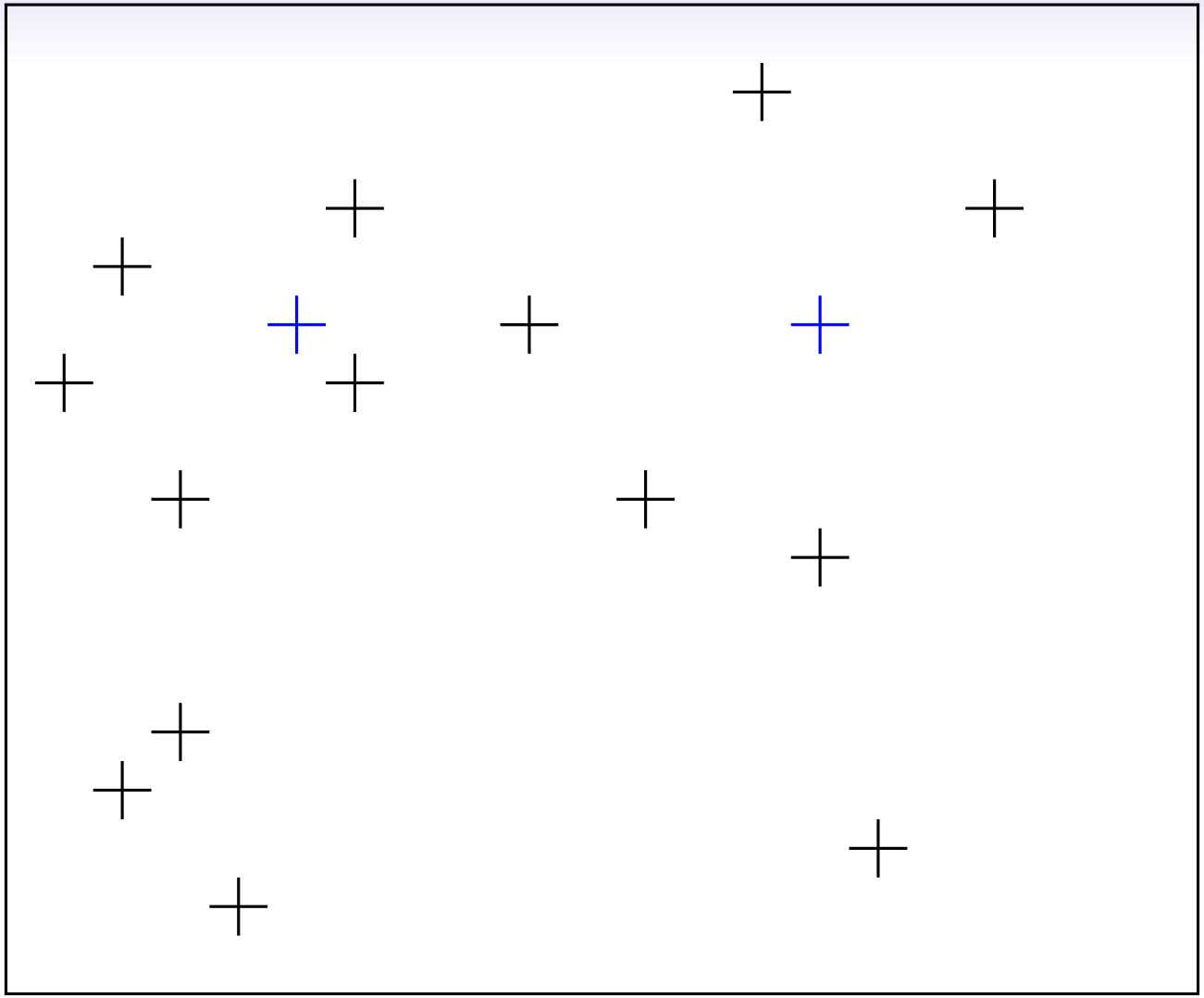
```
void calc_means(...) {
  ...
  for (int i = 0; i < num_clusters; i += 1) {
    // Calculer la moyenne des coordonnées des points du cluster
    cluster_size = 0;
    for (int j = 0; j < num_points; j += 1) {
      if (i == point2cluster[j]) {
        add_to_sum(sum, points[j]);
        cluster_size += 1;
      }
    }

    if (cluster_size != 0) {
      for (int j = 0; j < dim; j += 1) {
        clusters[i][j] = sum[j] / cluster_size;
      }
    }
  }
}
```

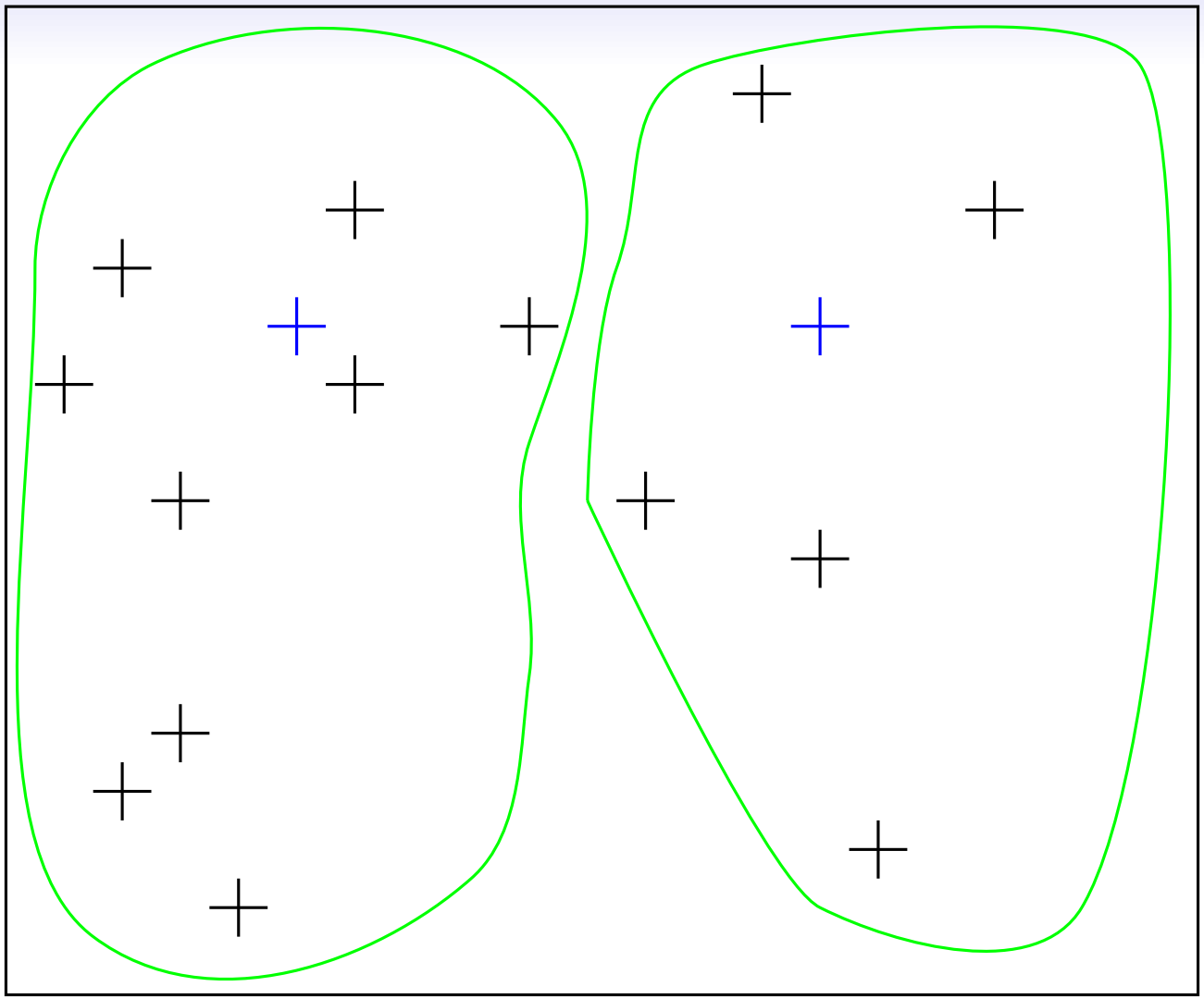
Kmeans : Illustration visuelle



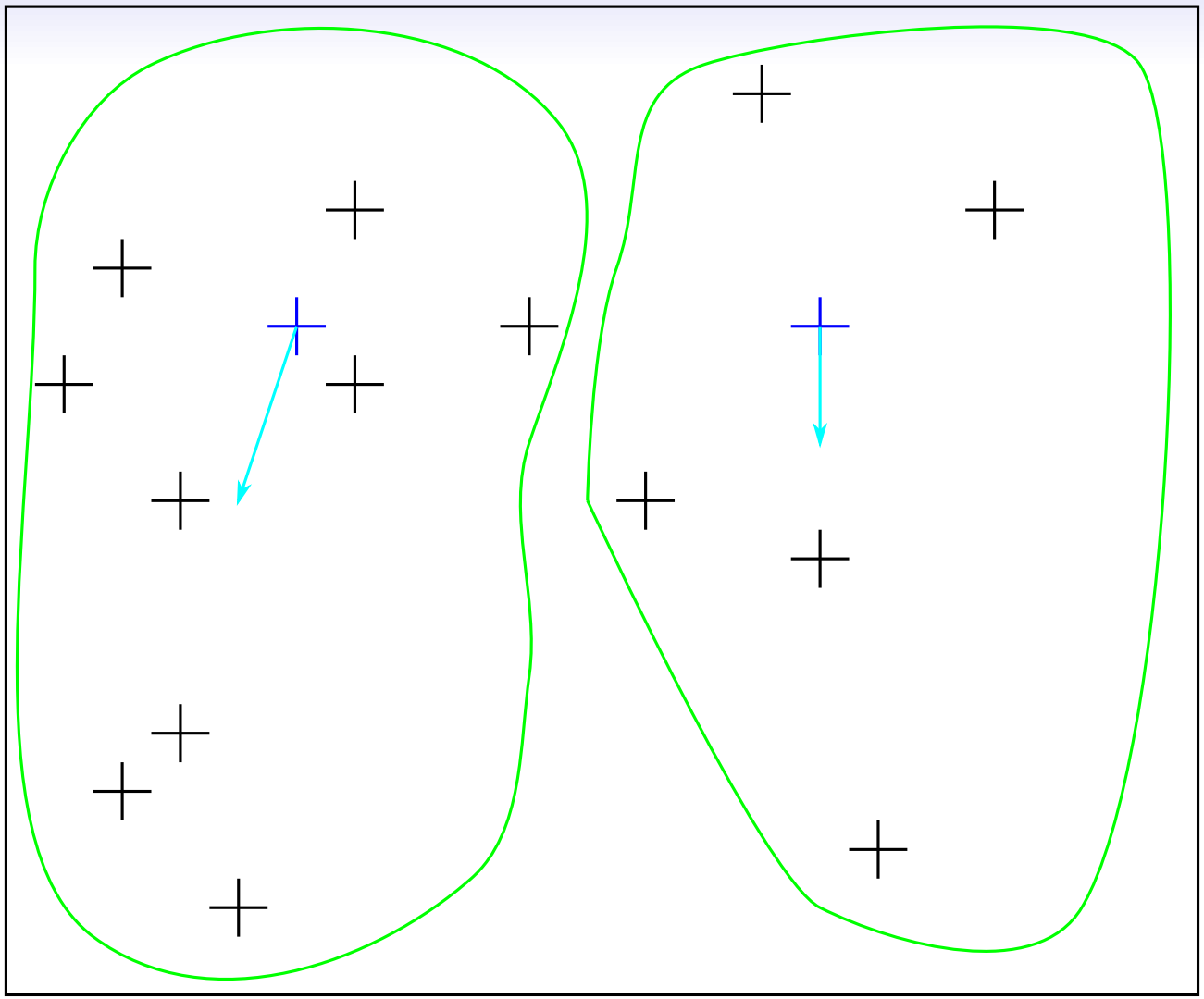
Kmeans : Illustration visuelle



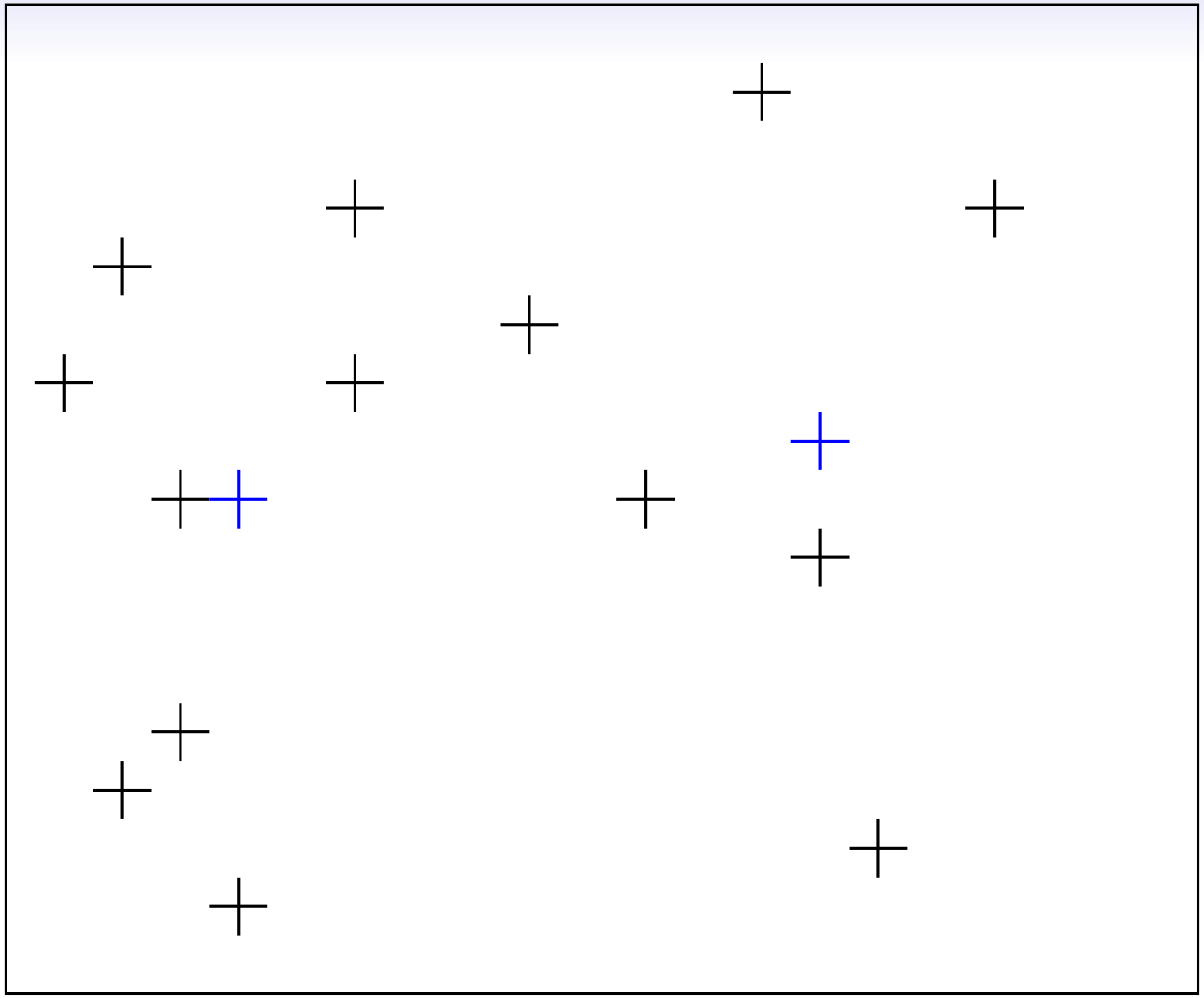
Kmeans : Illustration visuelle



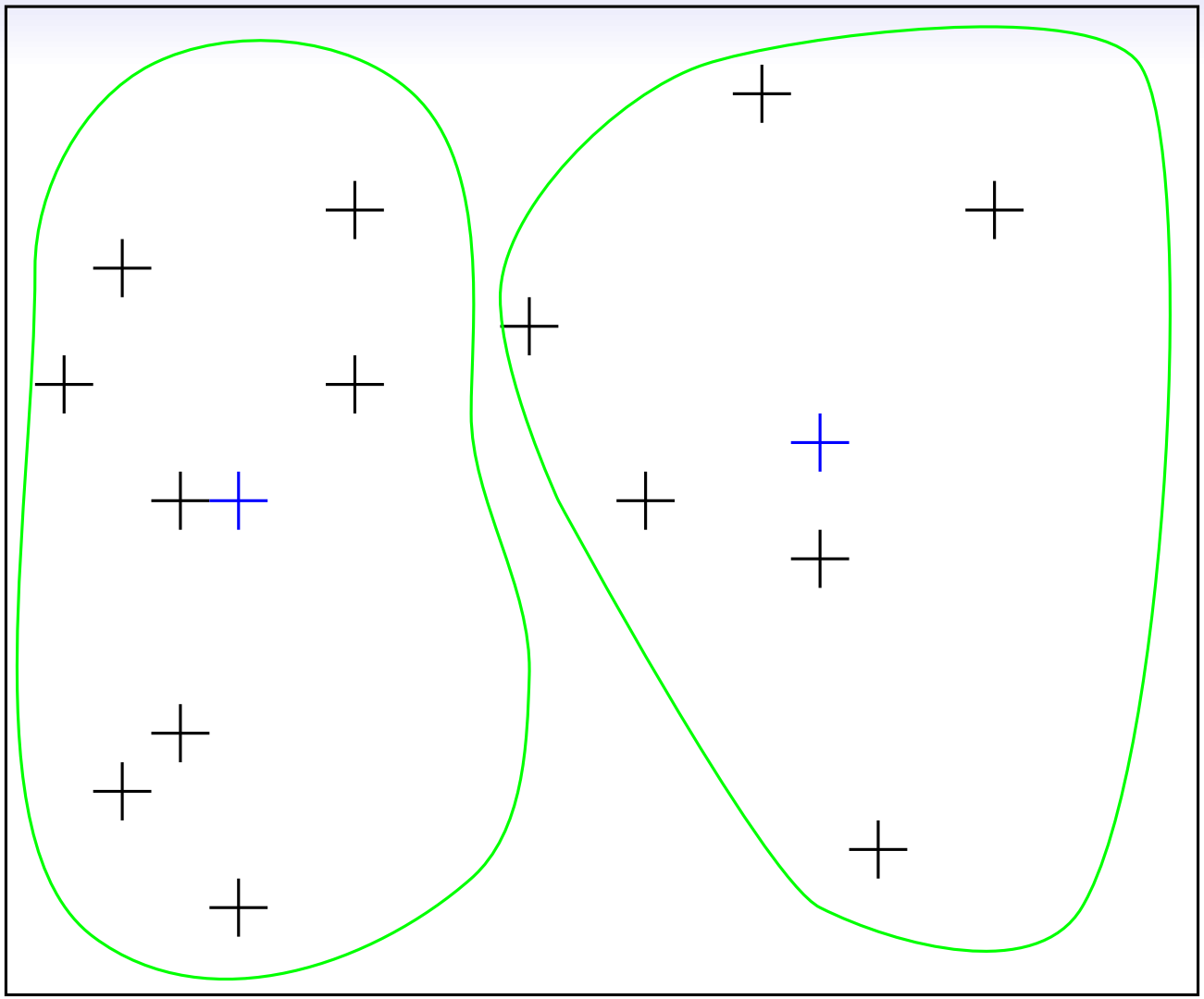
Kmeans : Illustration visuelle



Kmeans : Illustration visuelle



Kmeans : Illustration visuelle



Comment paralléliser l'application Kmeans ?

Comment paralléliser l'application Kmeans ?

Idées

- Découper les points à traiter par thread, pour le calcul du cluster pour chaque point
- Même chose pour les clusters, pour le calcul des nouvelles coordonnées
- Comment bien découper pour éviter des écarts ?

Comment paralléliser l'application Kmeans ?

Idées

- Découper les points à traiter par thread, pour le calcul du cluster pour chaque point
- Même chose pour les clusters, pour le calcul des nouvelles coordonnées
- Comment bien découper pour éviter des écarts ?
  - Ne pas faire tous les threads sauf 1 prennent `N / nb_threads` et le dernier thread tout le reste
  - Exemple : si 4 threads et 19 points : 4 points pour 3 des threads, 7 points pour le dernier thread
  - $\rightarrow$ Avoir au plus 1 point d'écart entre les threads

Comment paralléliser l'application Kmeans ?

Code de traitement

- Problème de la gestion des 2 boucles (`find_clusters` et `calc_means`)
- Implémentation naïve en pseudo-code (celle du bench) :

```
while (modified) {
  modified = false;
  pthread_create(find_clusters);
  pthread_join();
  pthread_create(calc_means);
  pthread_join();
}
```

- Problème : 2 créations et destructions de thread par itération
- Comment éviter cela ?

Comment paralléliser l'application Kmeans ?

Comment éviter d'avoir 2 créations et destructions de threads par itération ?

- Utiliser des barrières : les threads ne peuvent passer une barrière que quand il se trouvent au code correspondant

```
// Code d'un thread
while (modified) {
  barrier(); // attente que tous les threads aient lu modified
  modified = false;
  barrier(); // attente que tous les threads aient écrit modified
  find_clusters();
  barrier(); // attente que tous les points soient correctement rattachés
  calc_means();
  barrier(); // attente que toutes les nouvelles coordonnées des clusters
        soient re-calculées
}
```

- Remplace les 4 barrières implicites que causent `pthread_create` et `pthread_join` par 4 barrières explicites
  - Quand même beaucoup plus rapide
  - On pourrait même enlever une des 4 barrières...

Comment paralléliser l'application Kmeans ?

Comment créer des threads ?

- Bon ok, avec `pthread_create()`, mais encore...
- Idée : que le code marche quelque soit le nombre de threads (choisi de manière dynamique au début de l'application, par exemple sur la ligne de commande)
- $\Rightarrow$ Utilisation de macros pour permettre la factorisation du code (exemple en m4)

```
define(CREATE, `{
  for (int i = 1; i < ($2); i++) {
    int error = pthread_create(&pthread_table[i], NULL, (void * (*) (void *)) ($1), (void *) i);
    if (error != 0) {
      printf("*** Error in pthread_create\n");
      exit(-1);
    }
  }
  $1(0);
}')
define(WAIT_FOR_END, `{
  for (int i = 1; i < ($1); i++) {
    int error = pthread_join(pthread_table[i], NULL);
    if (error != 0) {
      printf("*** Error in pthread_join\n");
      exit(-1);
    }
  }
}')
```

Comment paralléliser l'application Kmeans ?

Passage des paramètres aux threads

- Conseil : regrouper tous les paramètres propres à un thread, et passer en paramètre au thread (un `void *`) soit le pointeur vers cette structure, soit l'index du thread, puis indexer un tableau global de structures avec
- En cas de tableau global, recopier le contenu dans des variables locales (éviter les *false sharing*)
- Exemple pour Kmeans :

```
typedef struct {
  int point_start_idx; // index du premier point à traiter
  int num_pts;         // nombre de points à traiter (pourrait être
                       // l'index du dernier point)
  int clus_start_idx;  // index du premier cluster
  int num_clusters;    // nombre de clusters à traiter
} thread_arg;
```

- Au début du thread (pour faire encore mieux, il faudrait passer un pointeur vers un `int`) :

```
void * find_clusters_and_calc_means(void * param) {
  long int tid = (long int) param;
  thread_arg * t_arg = &arg[tid];
```

Comment paralléliser l'application Kmeans ?

Comment mesurer le temps passé ?

- Quoi mesurer ?
  - Le temps total de l'application ?
  - Le temps passé dans les threads ?
  - Faut-il compter le temps des allocations et désallocations ?
- Comment mesurer le temps dans les threads ?
  - Temps moyen par thread ou temps total, i.e. du début du premier thread à la fin du dernier, c'est-à-dire max(fin) - min(début) ?
  - Rajout d'une barrière avant les endroits de mesure ? (Problème : rallonge le temps total)
- D'une manière générale, il faut préciser l'endroit de la mesure
- Les 4 endroits les plus utilisées :
  - Temps total
  - Avant la création des threads $\rightarrow$ après la fin de tous les threads
  - Début des threads $\rightarrow$ fin des threads
  - Début du calcul dans les threads $\rightarrow$ fin du calcul (exclut les allocations/désallocations)
- Souvent, on mesure le temps de calcul parallèle