OpenMP suite

Charles Bouillaguet
charles.bouillaguet@lip6.fr

26 mars 2021

Retour sur un exemple

Programme

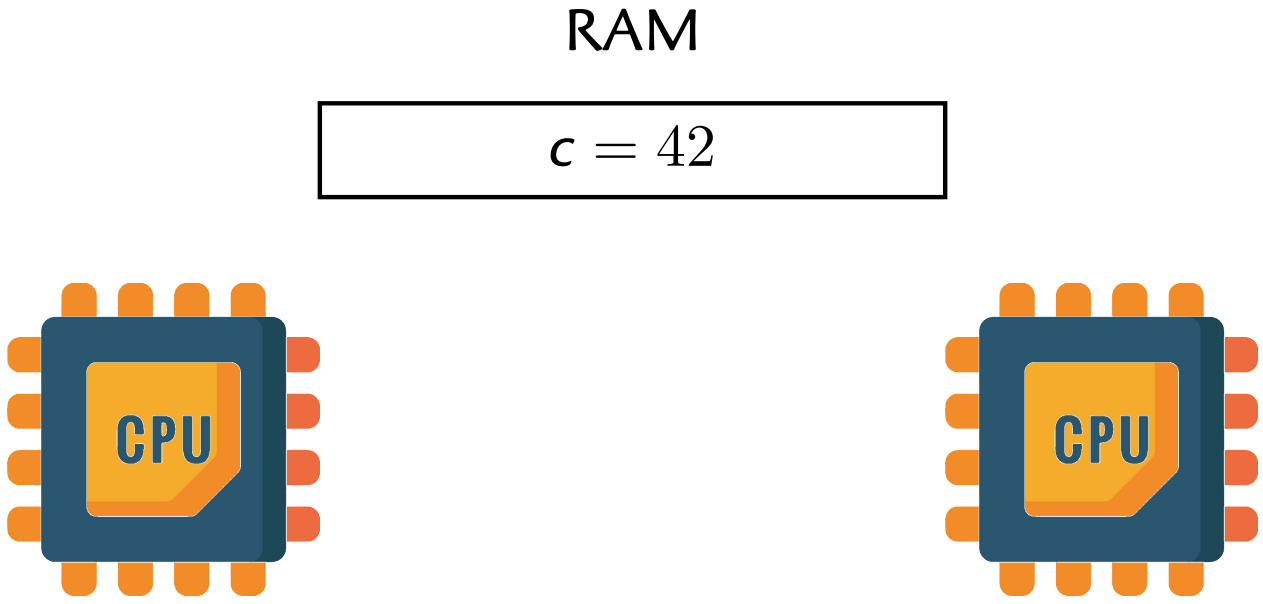
```c
#include <stdio.h>

int main()
{
  int c = 0;
  #pragma omp parallel for
  for (int i = 0; i < 100000; i++) {
        c++;
  }
  printf("c=%d\n", c);
}
```

Exécution

```
$ ./a.out
c=15074
```

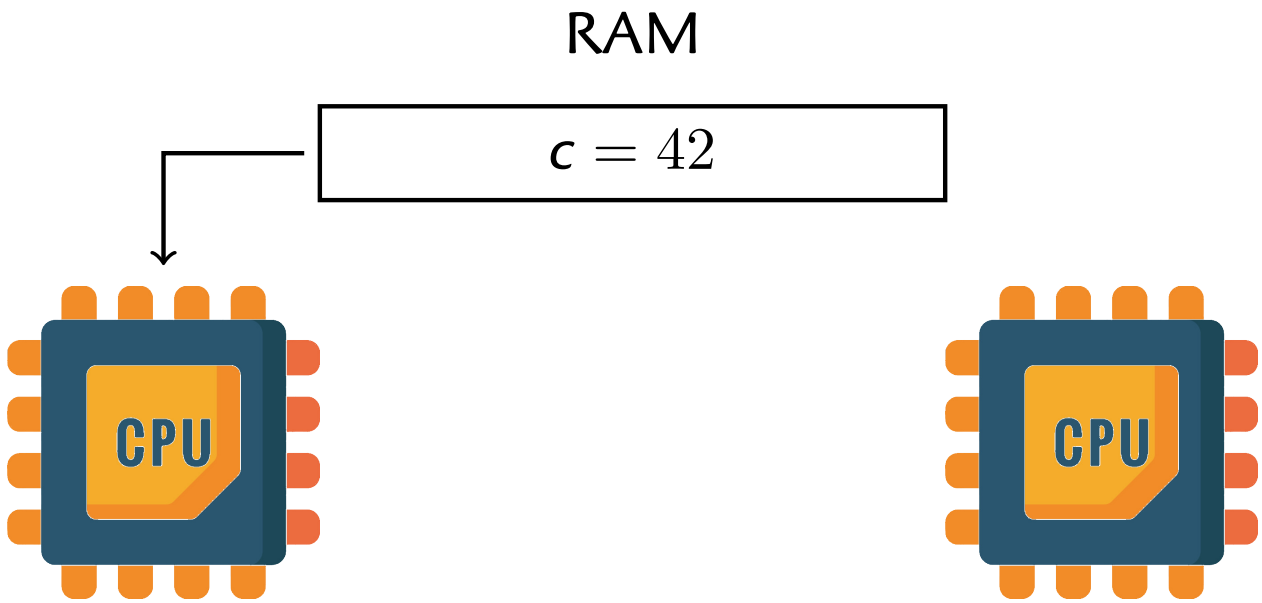
Entrelacement des incrémentations

Architectures *Load-Store*



Entrelacement des incrémentations

Architectures *Load-Store*

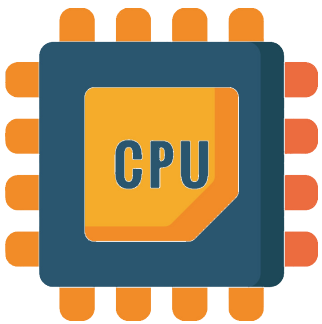


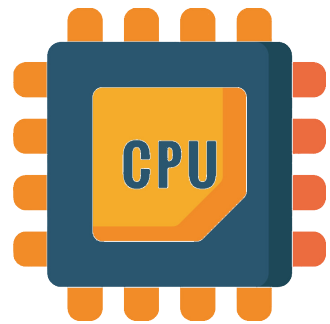
Entrelacement des incrémentations

Architectures *Load-Store*

RAM

$c = 42$

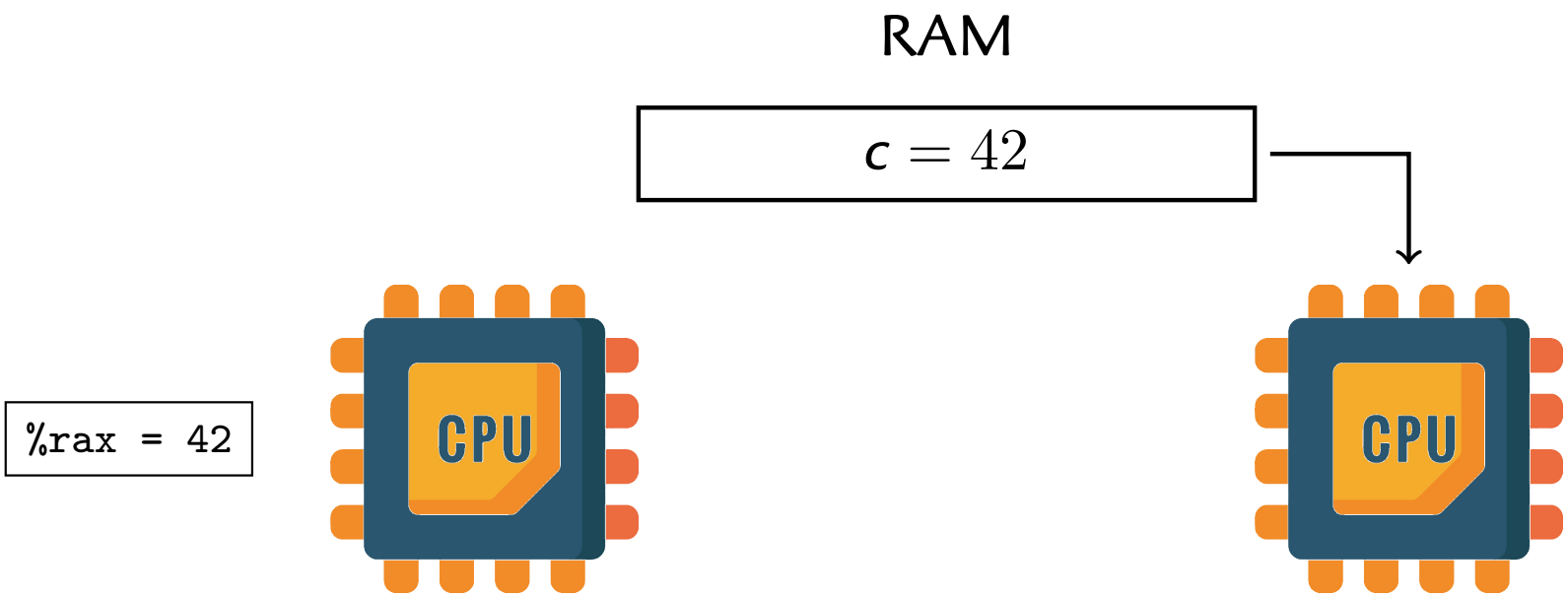





```
%rax = 42
```

Entrelacement des incrémentations

Architectures *Load-Store*

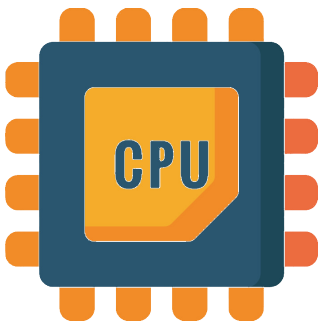


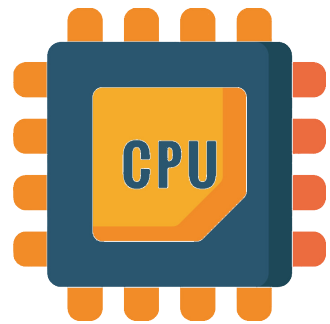
Entrelacement des incrémentations

Architectures *Load-Store*

RAM

$c = 42$





`%rax = 42`

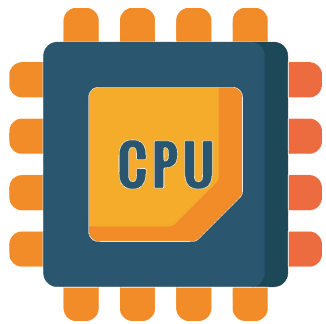
`%rax = 42`

Entrelacement des incrémentations

Architectures *Load-Store*

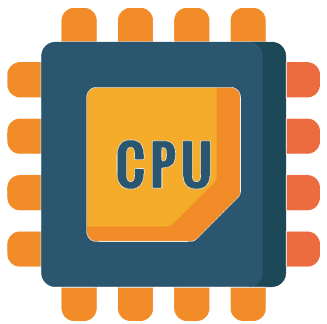
RAM

$c = 42$



`%rax = 42`

`add %rax, 1`



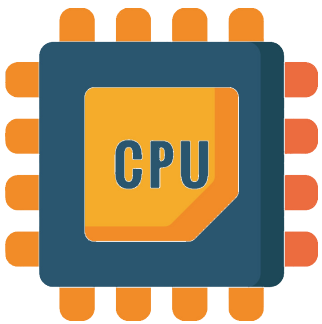
`%rax = 42`

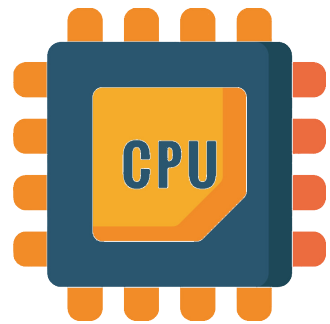
Entrelacement des incrémentations

Architectures *Load-Store*

RAM

$c = 42$





`%rax = 43`

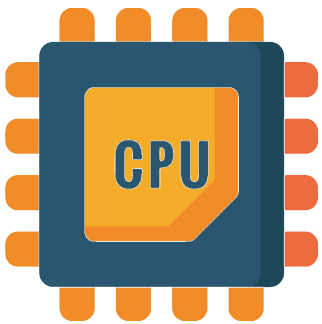
`%rax = 42`

Entrelacement des incrémentations

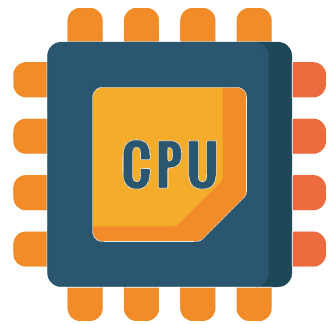
Architectures *Load-Store*

RAM

$c = 42$



`%rax = 43`



`%rax = 42`

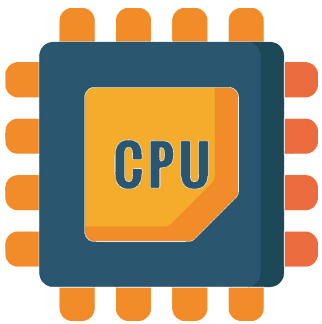
```
add %rax, 1
```

Entrelacement des incrémentations

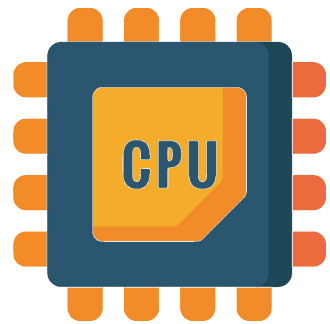
Architectures *Load-Store*

RAM

$c = 42$



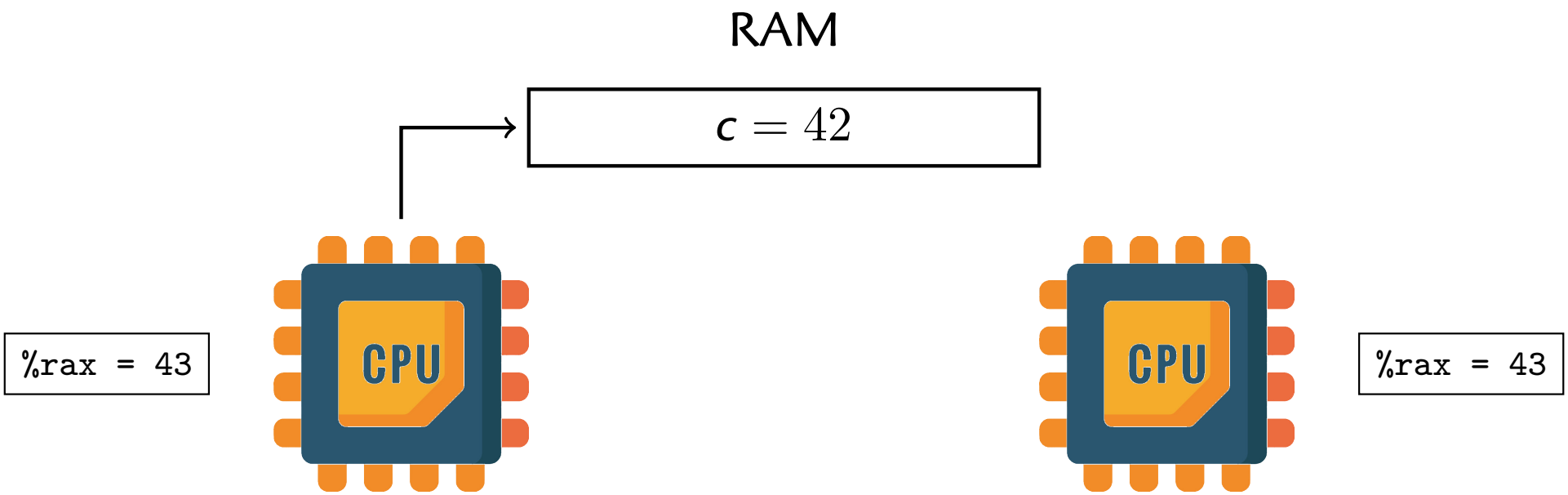
`%rax = 43`



`%rax = 43`

Entrelacement des incrémentations

Architectures *Load-Store*

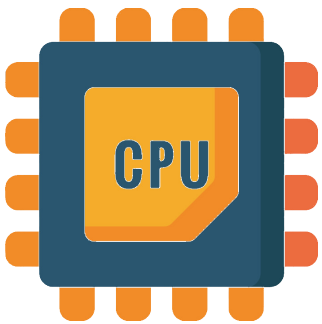


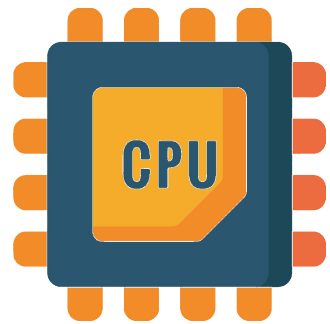
Entrelacement des incrémentations

Architectures *Load-Store*

RAM

$c = 43$



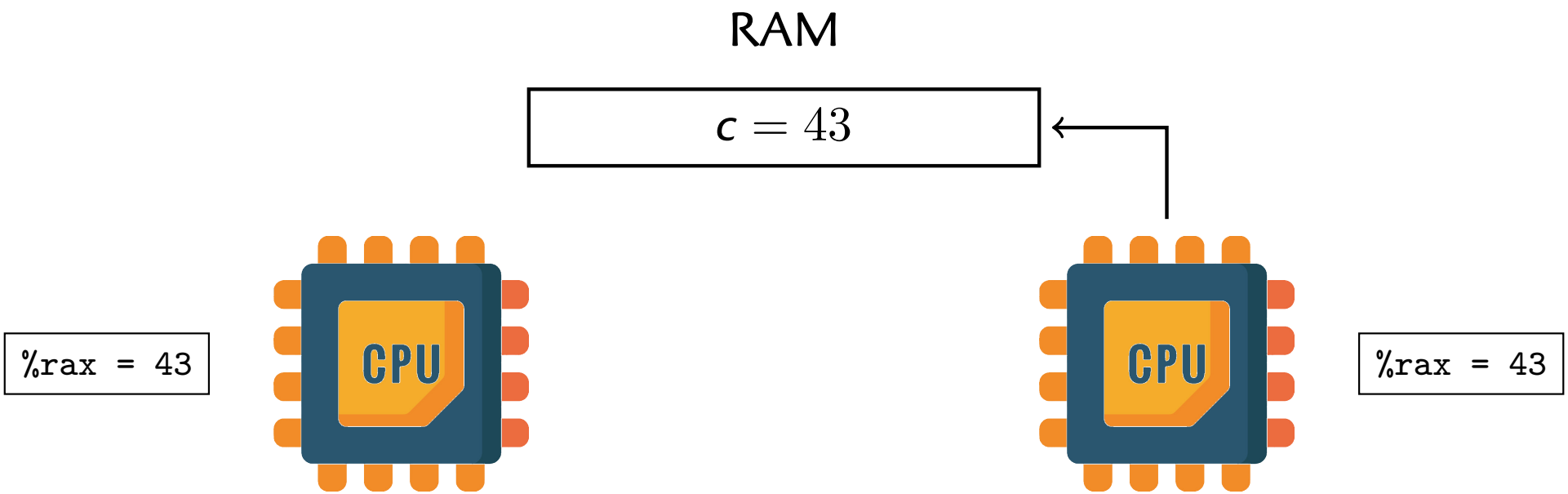


`%rax = 43`

`%rax = 43`

Entrelacement des incrémentations

Architectures *Load-Store*

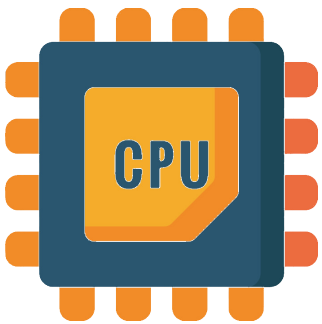


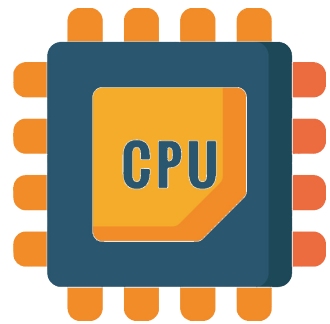
Entrelacement des incrémentations

Architectures *Load-Store*

RAM

$c = 43$





`%rax = 43`

`%rax = 43`

Mauvais

```c
#include <stdio.h>

int main()
{
  int c = 0;
  #pragma omp parallel for
  for (int i = 0; i < 100000; i++) {

    c++;
  }
  printf("c=%d\n", c);
}
```

Bon

```c
#include <stdio.h>

int main()
{
  int c = 0;
  #pragma omp parallel for
  for (int i = 0; i < 100000; i++) {
    #pragma omp atomic update
    c++;
  }
  printf("c=%d\n", c);
}
```

Architectures *load-store* :

`x++`, `x += 10`, etc. ne **sont pas atomiques**.

Règle d'or de la programmation multithreads

Tous les accès potentiellement conflictuels* aux variables partagées doivent être protégés (`atomic`, `critical`, ...).

* au moins l'un d'entre eux est une écriture.

`#pragma omp atomic` n'est pas la panacée

Exemple : somme des éléments d'un tableau

```
int sum = 0;
for (int i = 0; i < n; i++)
    sum += A[i];
```

$T = 5.95\text{s}$ $(n = 10^{10})$

```
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < n; i++)
    #pragma omp atomic
    sum += A[i];
```

```
int sum = 0;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < n; i++)

    sum += A[i];
```

(2 × Xeon Gold 6152 (« Skylake ») à 22 coeurs)

`#pragma omp atomic` n'est pas la panacée

Exemple : somme des éléments d'un tableau

```
int sum = 0;
for (int i = 0; i < n; i++)
    sum += A[i];
```

$T = 5.95\text{s}$ $(n = 10^{10})$

```
int sum = 0;
#pragma omp parallel for
for (int i = 0; i < n; i++)
    #pragma omp atomic
    sum += A[i];
```

$T \geq 200\text{s}$!!!

```
int sum = 0;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < n; i++)

    sum += A[i];
```

$T = 0.46\text{s}$ $(\times 12.9)$

(2 $\times$ Xeon Gold 6152 (« Skylake ») à 22 coeurs)

Histogramme (par ex. `numpy.histogram`)

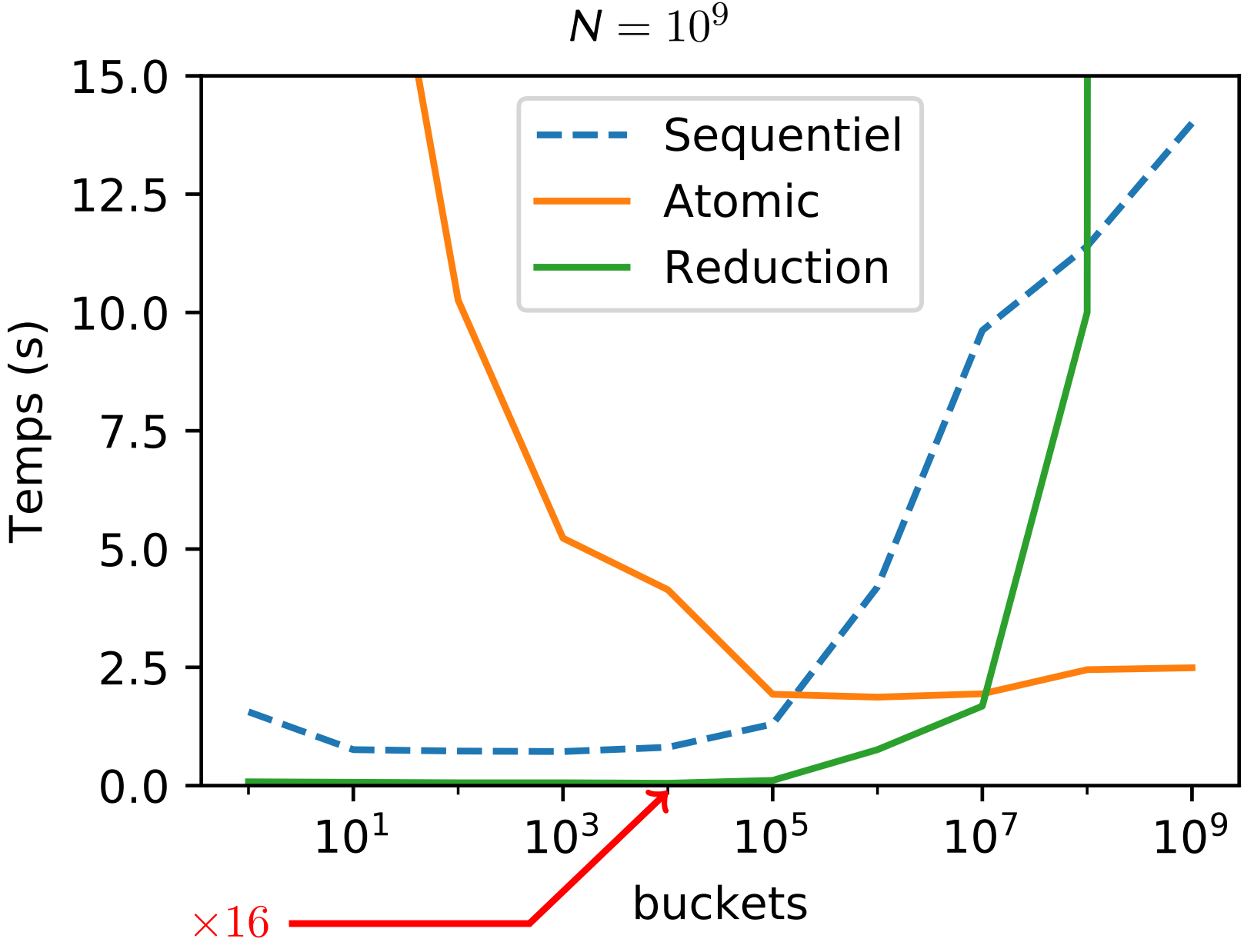
Plan A

```
void histogram(u32 A[], u64 n, u64 buckets, u64 H[])
{
    #pragma omp parallel for
    for (u64 i = 0; i < n; i++) {
        u64 x = (A[i] * buckets) >> 32;
        #pragma omp atomic
        H[x]++;
    }
}
```

Plan B

```
void histogram(u32 A[], u64 n, u64 buckets, u64 H[])
{
    #pragma omp parallel for reduction(+:H[0:buckets])
    for (u64 i = 0; i < n; i++) {
        u64 x = (A[i] * buckets) >> 32;
        H[x]++;
    }
}
```

Exemple : histogramme



You **must** follow the rule

You **must** follow the rule

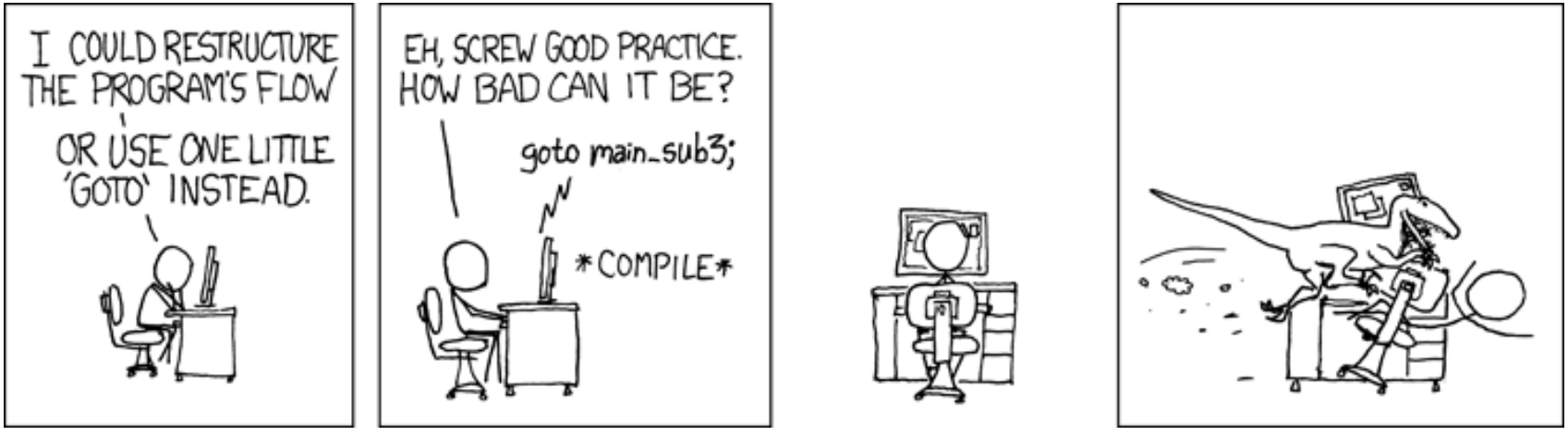
Or else...

You **must** follow the rule

Or else...

You will be living
In a **world of PAIN**

Don't follow the rules ?



https://xkcd.com/292/

La triste vérité...

- Barrière $\rightarrow$ attente
- Critical $\rightarrow$ séquentialisation
- Atomic $\rightarrow$ plus lent qu'un accès normal

Synchronisation $\rightarrow$ limite le passage à l'échelle.

$\Longrightarrow$ rôle important de la localité des données.

Tout le monde en passe inévitablement par là

- Bon, mais alors si on évite `i++`, ça va aller, non ?
- J'ai lu dans la doc de mon CPU que les lectures/écritures alignées étaient atomiques ; si on se tient à ça, ça va aller ?

Tout le monde en passe inévitablement par là

- Bon, mais alors si on évite `i++`, ça va aller, non ?
- J'ai lu dans la doc de mon CPU que les lectures/écritures alignées étaient atomiques ; si on se tient à ça, ça va aller ?

Crise d'adolescence

Laissons tomber la règle d'or.

Tout le monde en passe inévitablement par là

- Bon, mais alors si on évite `i++`, ça va aller, non ?
- J'ai lu dans la doc de mon CPU que les lectures/écritures alignées étaient atomiques ; si on se tient à ça, ça va aller ?

Crise d'adolescence

Laissons tomber la règle d'or.

Guru switch ⟶

Safety

World of PAIN

Le *Peterson Lock* (1981)

```
bool flag[2];
int turn;

void lock()
{
    int i = omp_get_thread_num();
    flag[i] = true;                          // I'm interested
    turn    = 1 - i;                         // you go first
    while (turn != i && flag[1-i]) {};       // wait 'til he's not interested
}                                            //    or its my turn

void unlock()
{
    int i = omp_get_thread_num();
    flag[i] = false;                         // I'm not interested
}
```

CLAIMS

- Exclusion mutuelle
- Pas de *deadlock*
- Starvation-free

LIVE DEMO

Guru problem #1 : **Trahison du compilateur**

- On avait :

```
turn = 1 - i;                          // you go first
while (turn != i && flag[1-i]) {}; // wait
```

- L'optimiseur « sait » que `turn != i`
- Donc ceci devient :

```
void lock()
{
    int i = omp_get_thread_num();
    flag[i] = true;                    // I'm interested
    while (flag[1-i]) {};              // wait
}
```

- Les deux threads appellent `lock()` en même temps...
- ... `flag[0] == flag[1] == true` ...
- Deadlock.

Guru problem #2 : ???

- Défaut d'exclusion mutuelle
- Les deux threads pénètrent simultanément dans la section critique
- Comment est-ce possible ?

Relations d'ordre sur les accès à la mémoire

- Accès mémoire :

  $W_i(x)a$: T_i écrit la valeur a dans la variable x

  $R_i(x)b$: T_i lit la variable x et la valeur b

- « *Program Order* » :

  $x \xrightarrow{po} y$: le code demande qu'on fasse x d'abord et y après

- « *Extended Communication Order* » :

  $W(x)a \xrightarrow{\texttt{rf}} R(x)a$: la lecture renvoie la valeur écrite

  $W(x)a \xrightarrow{\texttt{mo}} W(x)b$: l'écriture de a a lieu avant celle de b

  $R(x)a \xrightarrow{\texttt{rb}} W(x)b$: la lecture a lieu avant l'écriture

  par définition, $\xrightarrow{\texttt{rb}} = (\xrightarrow{\texttt{rf}})^{-1}; \xrightarrow{\texttt{mo}}$

Relations d'ordre sur les accès à la mémoire

- Accès mémoire :

$W_i(x)a$: T_i écrit la valeur a dans la variable x

$R_i(x)b$: T_i lit la variable x et la valeur b

- « *Program Order* » :

$x \xrightarrow{po} y$: le code demande qu'on fasse x d'abord et y après

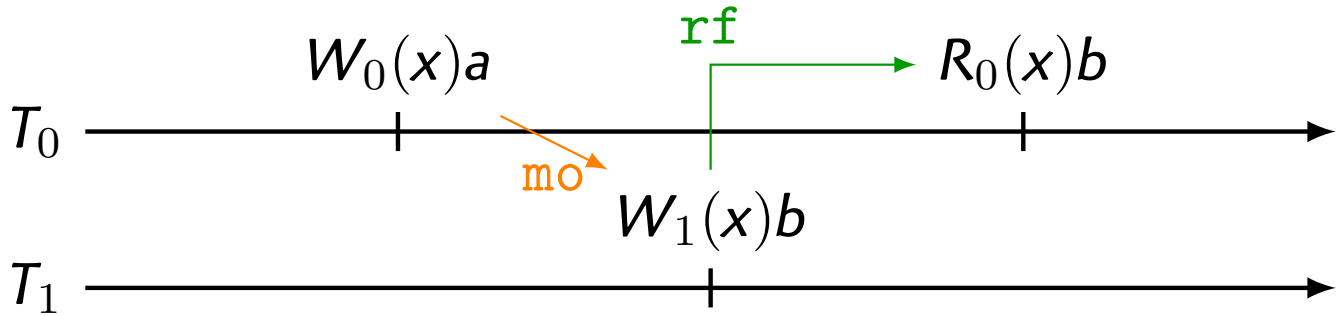
- « *Extended Communication Order* » :

$W(x)a \xrightarrow{\texttt{rf}} R(x)a$: la lecture renvoie la valeur écrite

$W(x)a \xrightarrow{\texttt{mo}} W(x)b$: l'écriture de a a lieu avant celle de b

$R(x)a \xrightarrow{\texttt{rb}} W(x)b$: la lecture a lieu avant l'écriture

par définition, $\xrightarrow{\texttt{rb}} = (\xrightarrow{\texttt{rf}})^{-1}; \xrightarrow{\texttt{mo}}$



Relations d'ordre entre accès à la mémoire

$W(x)a \xrightarrow{\texttt{rf}} R(x)a$: la lecture renvoie la valeur écrite

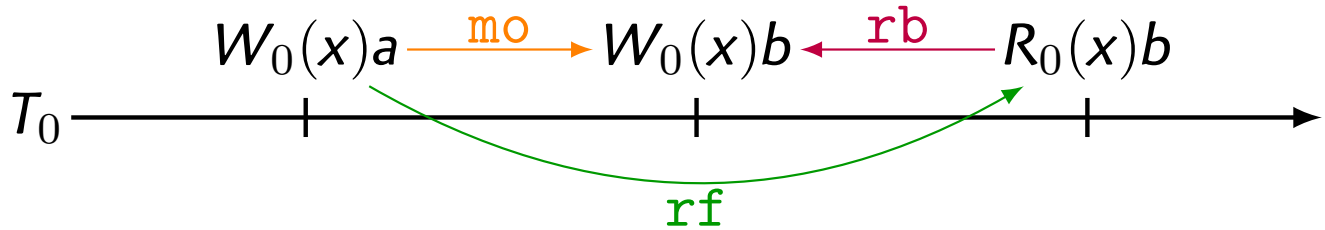
$W(x)a \xrightarrow{\texttt{mo}} W(x)b$: l'écriture de a a lieu avant celle de b

$R(x)a \xrightarrow{\texttt{rb}} W(x)b$: la lecture a lieu avant l'écriture

par définition, $\xrightarrow{\texttt{rb}} = (\xrightarrow{\texttt{rf}})^{-1}; \xrightarrow{\texttt{mo}}$

Intuition

- En principe, $\xrightarrow{\texttt{rf}}$, $\xrightarrow{\texttt{mo}}$, $\xrightarrow{\texttt{rb}}$ ne peuvent pas « contredire » $\xrightarrow{po}$
- On ne lit pas de valeur « périmée » ($\xrightarrow{\texttt{rb}}$ à l'envers)



Relations d'ordre entre accès à la mémoire

$W(x)a \xrightarrow{\texttt{rf}} R(x)a$: la lecture renvoie la valeur écrite

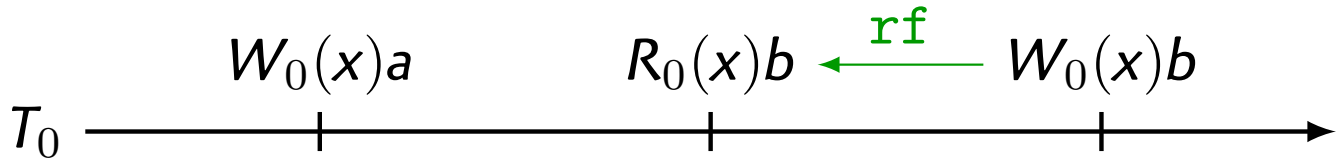
$W(x)a \xrightarrow{\texttt{mo}} W(x)b$: l'écriture de a a lieu avant celle de b

$R(x)a \xrightarrow{\texttt{rb}} W(x)b$: la lecture a lieu avant l'écriture

par définition, $\xrightarrow{\texttt{rb}} = (\xrightarrow{\texttt{rf}})^{-1}; \xrightarrow{\texttt{mo}}$

Intuition

- En principe, $\xrightarrow{\texttt{rf}}$, $\xrightarrow{\texttt{mo}}$, $\xrightarrow{\texttt{rb}}$ ne peuvent pas « contredire » $\xrightarrow{po}$
- On ne lit pas de valeur « périmée » ($\xrightarrow{\texttt{rb}}$ à l'envers)
- Pas de lecture « depuis le futur » ($\xrightarrow{\texttt{rf}}$ à l'envers)



Relations d'ordre entre accès à la mémoire

$W(x)a \xrightarrow{\texttt{rf}} R(x)a$: la lecture renvoie la valeur écrite

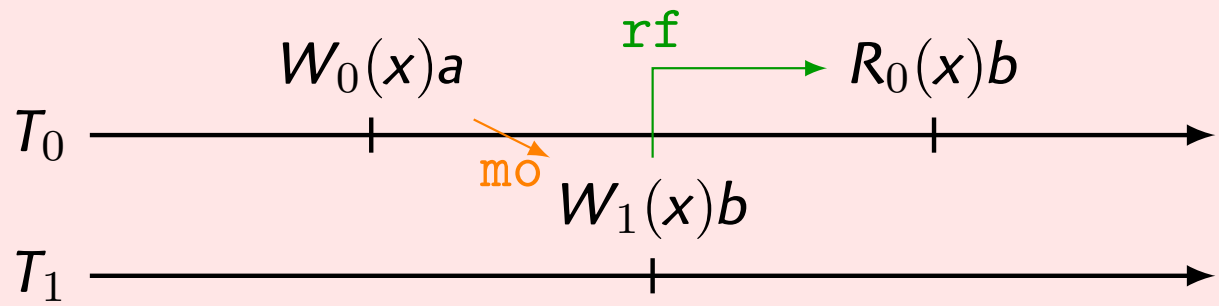
$W(x)a \xrightarrow{\texttt{mo}} W(x)b$: l'écriture de a a lieu avant celle de b

$R(x)a \xrightarrow{\texttt{rb}} W(x)b$: la lecture a lieu avant l'écriture

par définition, $\xrightarrow{\texttt{rb}} = (\xrightarrow{\texttt{rf}})^{-1}; \xrightarrow{\texttt{mo}}$

En séquentiel : $\xrightarrow{\texttt{rf}} \subseteq \xrightarrow{po}$, $\xrightarrow{\texttt{mo}} \subseteq \xrightarrow{po}$ et $\xrightarrow{\texttt{rb}} \subseteq \xrightarrow{po}$

C'est faux en parallèle



La *Sequential Consistency*

Intuition

Tout se passe comme si les accès mémoires étaient exécutées *séquentiellement* (dans un ordre qu'on ne connaît pas forcément).

La *Sequential Consistency*

Intuition

Tout se passe comme si les accès mémoires étaient exécutées *séquentiellement* (dans un ordre qu'on ne connaît pas forcément).

Definition (Sequential Consistency)

Un système parallèle est séquentiellement consistant si, pour chacune des exécutions possibles des threads auxquelles il peut aboutir, on peut construire un historique H :

- séquence totalement ordonnée
- contient une et une seule fois chaque accès mémoire
- compatible avec le code des threads (respecte $\xrightarrow{po}$)
- lecture de la dernière valeur écrite

Theorem

Sequential Consistency $\iff$ *pas de cycles avec* $\xrightarrow{po} \cup \xrightarrow{rf} \cup \xrightarrow{mo} \cup \xrightarrow{rb}$.

Le *Peterson Lock* (1981)

```
bool flag[2];
int turn;

void lock()
{
    int i = omp_get_thread_num();
    flag[i] = true;                          // I'm interested
    turn    = 1 - i;                         // you go first
    while (turn != i && flag[1-i]) {};       // wait 'til he's not interested
}                                            //    or its my turn

void unlock()
{
    int i = omp_get_thread_num();
    flag[i] = false;                         // I'm not interested
}
```

CLAIMS

- Exclusion mutuelle
- Pas de *deadlock*
- Starvation-free

Correction du *Peterson Lock*

Sequential Consistency

Tout se passe comme si les accès mémoires étaient exécutées *séquentiellement* (dans un ordre qu'on ne connaît pas forcément).

Theorem

Si la mémoire est séquentiellement consistante, alors le Peterson lock garantit l'exclusion mutuelle.

Preuve : ...

- Absurde : T_0 et T_1 appellent `lock()`, entrent dans la section critique.
- État initial : `flag[0] = false; flag[1] = false;`
- D'après le code :

$$T_0 : W_0(\texttt{flag[0]})\texttt{true} \xrightarrow{po} W_0(\texttt{turn})1 \xrightarrow{po} R_0(\texttt{turn})? \xrightarrow{po} R_0(\texttt{flag[1]})? \xrightarrow{po} CS_0$$

$$T_1 : W_1(\texttt{flag[1]})\texttt{true} \xrightarrow{po} W_1(\texttt{turn})0 \xrightarrow{po} R_1(\texttt{turn})? \xrightarrow{po} R_1(\texttt{flag[0]})? \xrightarrow{po} CS_1$$

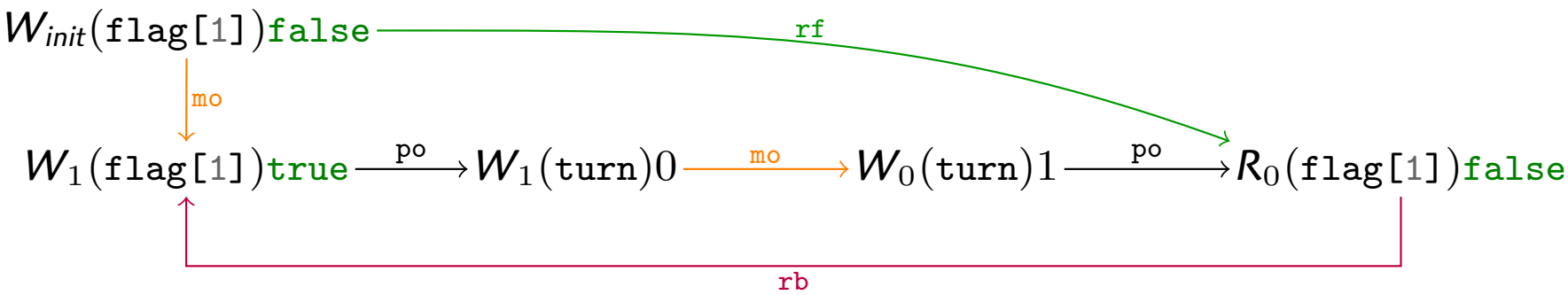
- Supposons que T_0 écrive `turn` en dernier :

$$W_1(\texttt{turn})0 \xrightarrow{\texttt{mo}} W_0(\texttt{turn})1.$$

- T_0 sort de la boucle, et `turn == 1`, donc forcément :

$$W_0(\texttt{turn})1 \xrightarrow{po} R_0(\texttt{turn})1 \xrightarrow{po} R_0(\texttt{flag[1]})\texttt{false}.$$

- Si on met tout ceci bout-à-bout :



- Cycle $\Rightarrow$ non-SC $\Rightarrow$ Contradiction !

Guru problem #2 : **défaut de *sequential consistency***

- Preuve : sequential consistency $\Rightarrow$ mutual exclusion
- Observation : ~~mutual exclusion~~
- Conséquence :

  - Mon laptop n'est pas séquentiellement consistent...

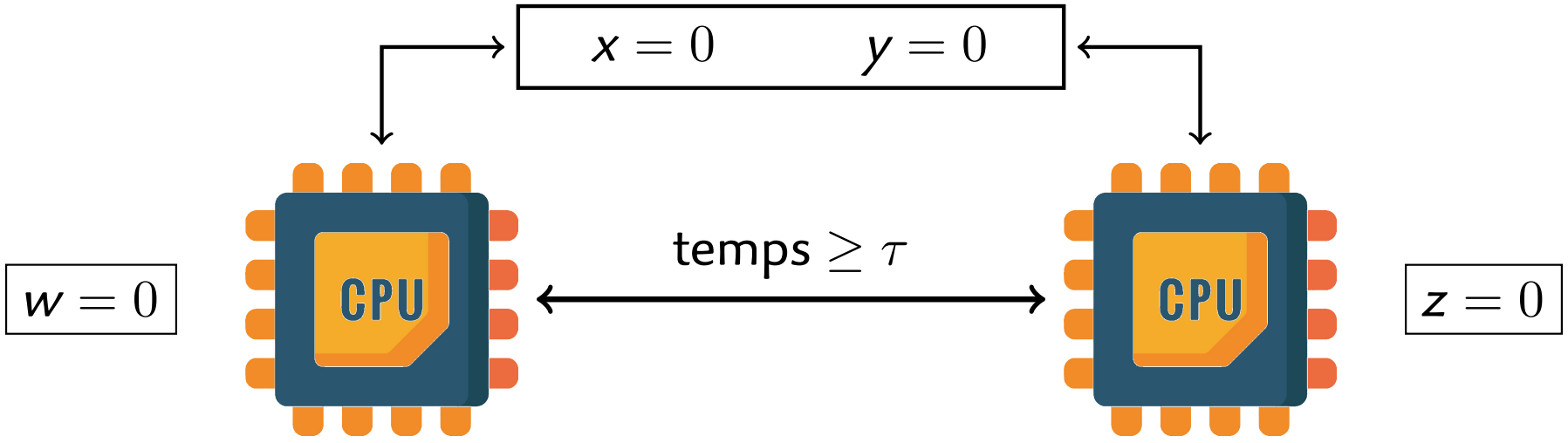
Guru problem #2 : **défaut de *sequential consistency***

- Preuve : sequential consistency $\Rightarrow$ mutual exclusion
- Observation : ~~mutual exclusion~~
- Conséquence :

  - Mon laptop n'est pas séquentiellement consistent...
  - Le tien ne l'est pas non plus !

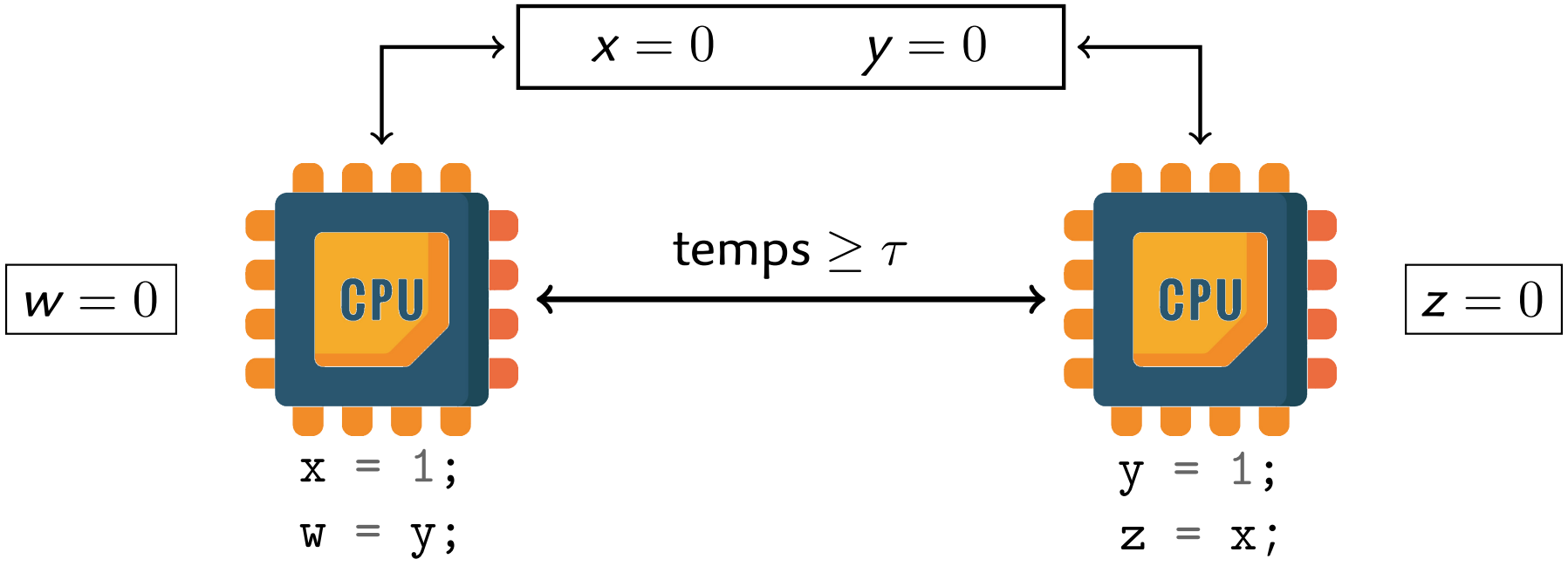
Les CPU ne sont pas séquentiellement consistents !

La *Sequential Consistency* est coûteuse



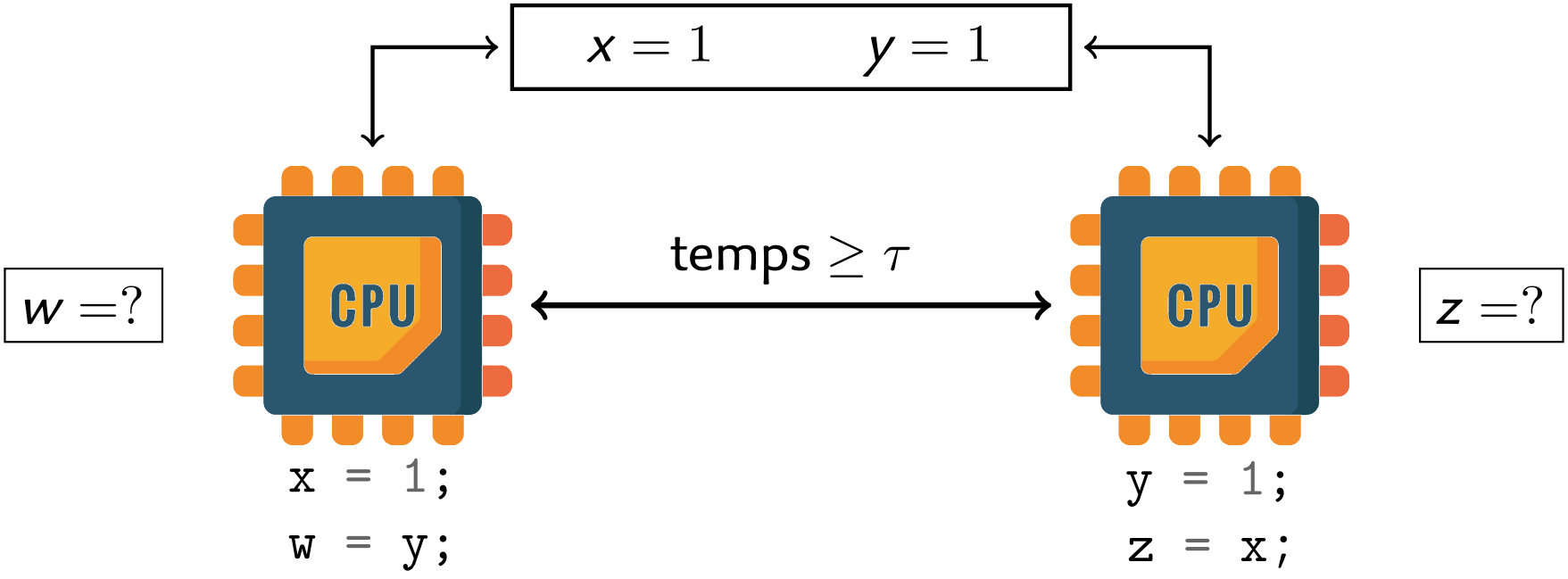
Les CPU ne sont pas séquentiellement consistents !

La *Sequential Consistency* est coûteuse



Les CPU ne sont pas séquentiellement consistents !

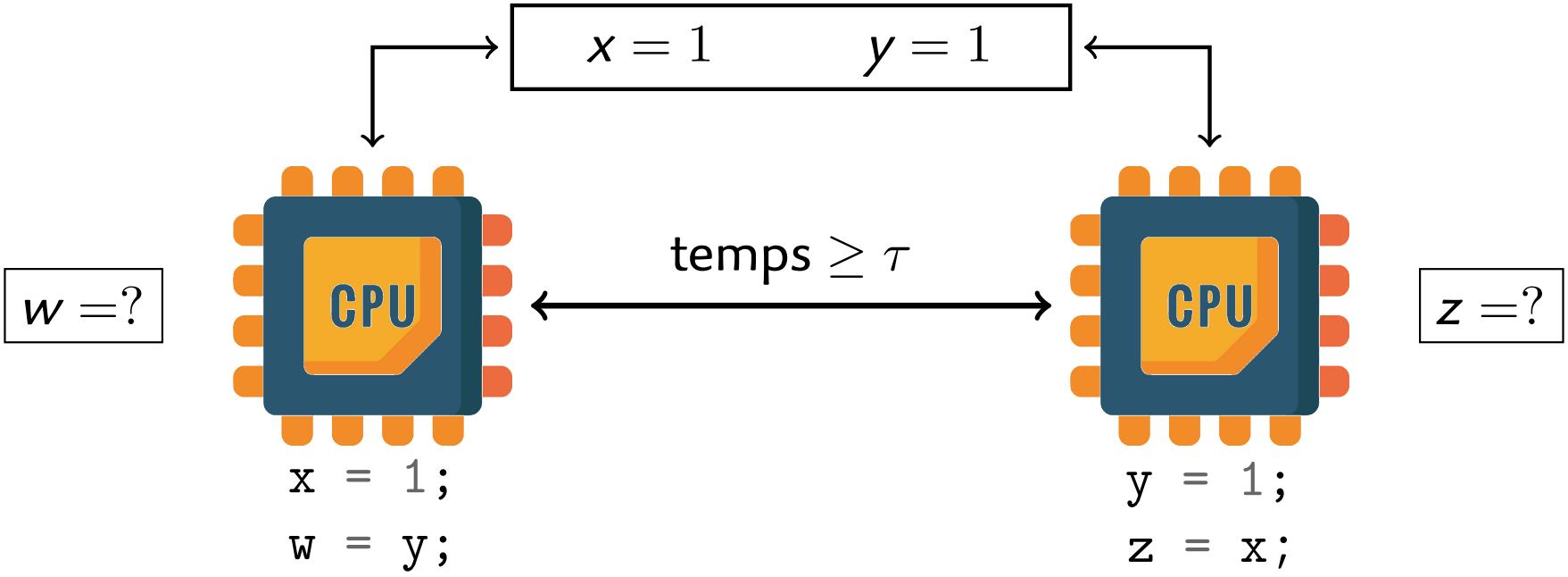
La *Sequential Consistency* est coûteuse

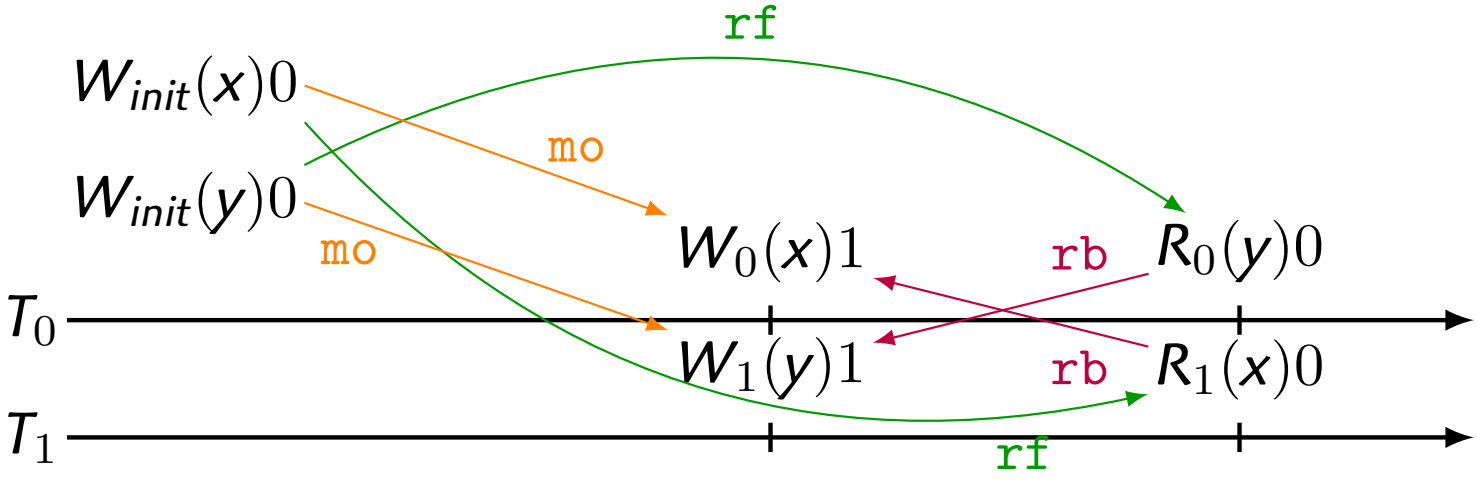


Sequential Consistency $\Rightarrow (w, z) \neq (0, 0)$

Les CPU ne sont pas séquentiellement consistents !

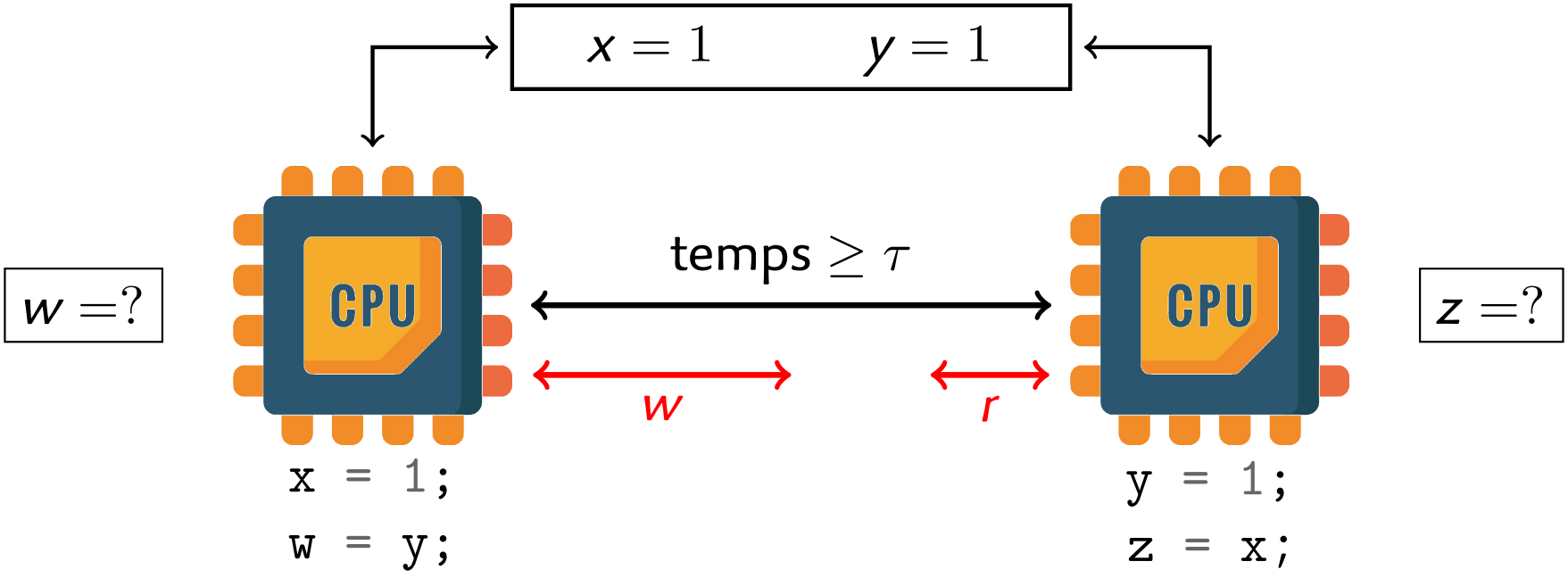
La *Sequential Consistency* est coûteuse





Les CPU ne sont pas séquentiellement consistents !

La *Sequential Consistency* est coûteuse



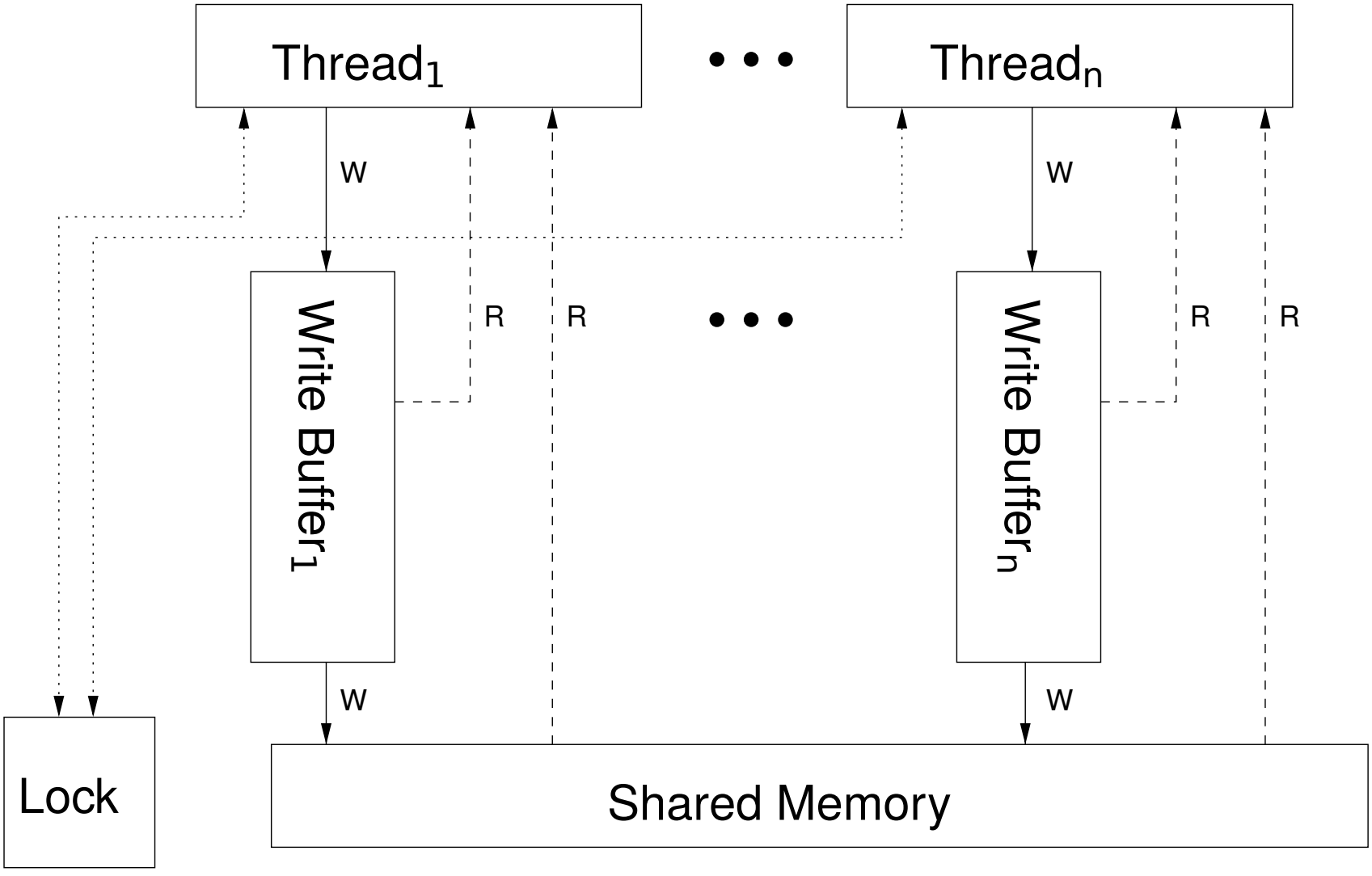
r : temps min. pour faire une lecture

w : temps min. pour faire une écriture

$$\textit{Sequential Consistency} \Longrightarrow r + w \geq \tau$$

L'un doit bien lire l'écriture de l'autre...

Le *Store Buffering*



(image : A Tutorial Introduction to the ARM and POWER Relaxed Memory Models)

Architectures avec *Total Store Ordering*

x86 et SPARC

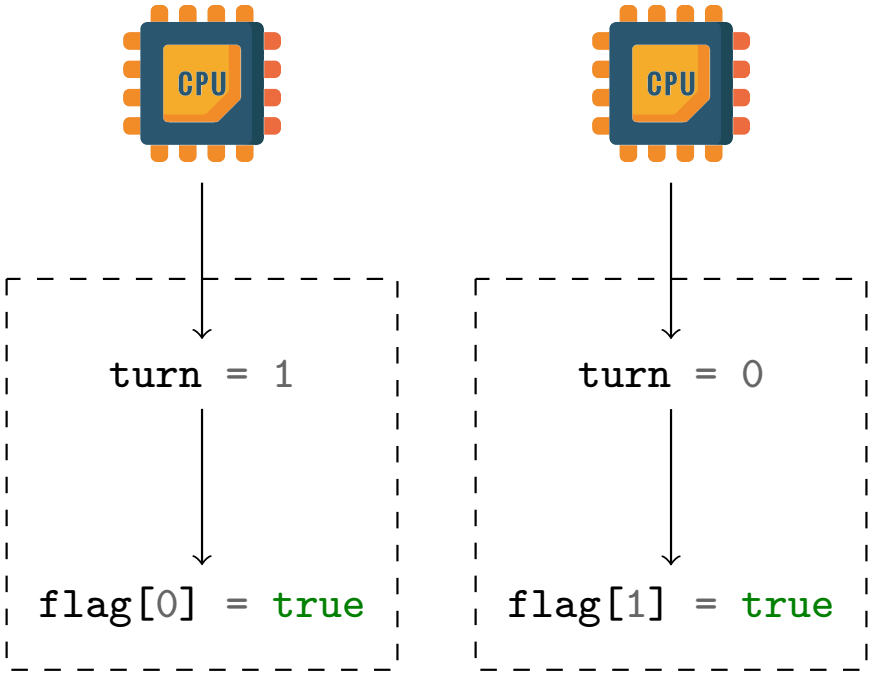
Chaque thread matériel possède un **Store Buffer**

- Mes écritures sont placées en attente dans mon *Store Buffer*
- C'est une file (on ne double pas !)
- Je *vois* mes écritures tout de suite
- Mon *Store Buffer* sera « vidangé » vers la mémoire… à terme
- À ce moment-là :
  - *Tous les autres threads* voient mes écritures.
  - Ils les voient dans le même ordre.

Échec du *Peterson Lock* en présence de *Store Buffering*

Le « *Peterson Lock* »

```
bool flag[2];
int victim;

void lock()
{
    int i = omp_get_thread_num();
    flag[i] = true;
    turn = 1 - i;
    while (turn != i && flag[1-i])
        {};
}
```

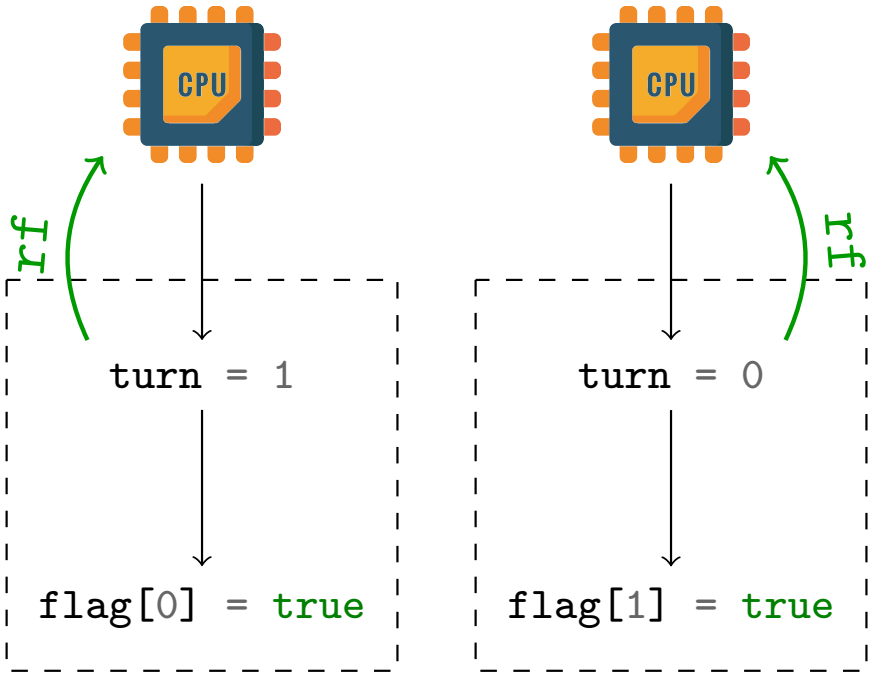


`flag[0] = false` `flag[1] = false`

Échec du *Peterson Lock* en présence de *Store Buffering*

Le « *Peterson Lock* »

```
bool flag[2];
int victim;

void lock()
{
    int i = omp_get_thread_num();
    flag[i] = true;
    turn = 1 - i;
    while (turn != i && flag[1-i])
        {};
}
```

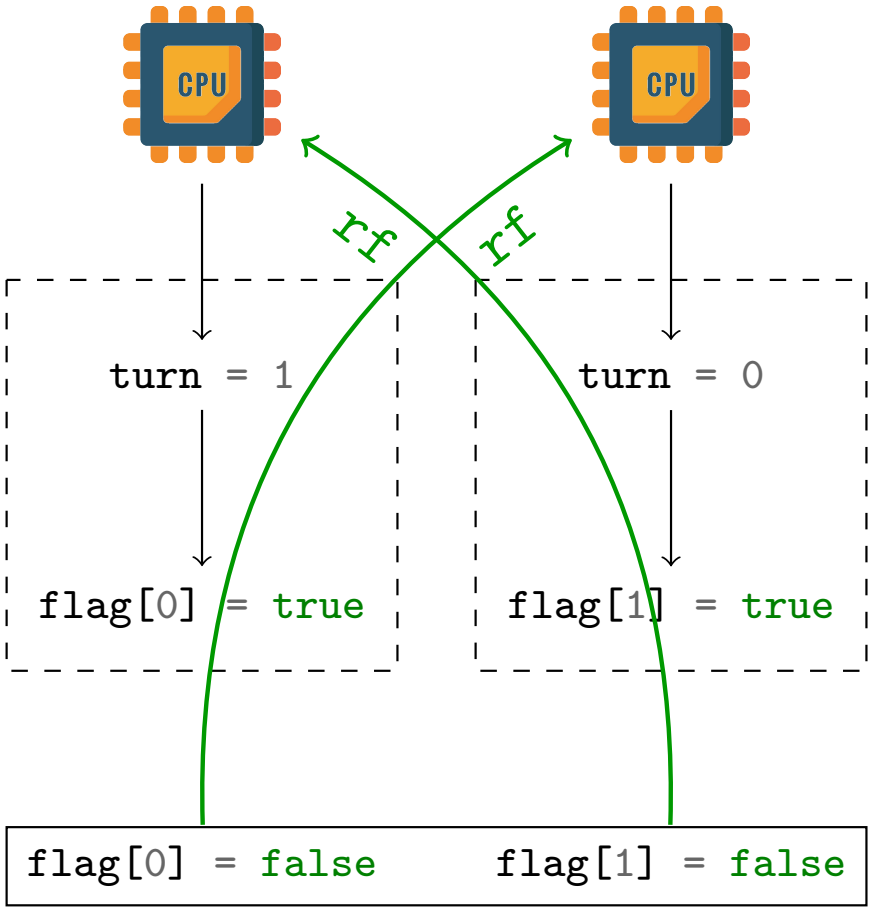


`flag[0] = false` `flag[1] = false`

Échec du *Peterson Lock* en présence de *Store Buffering*

Le « *Peterson Lock* »

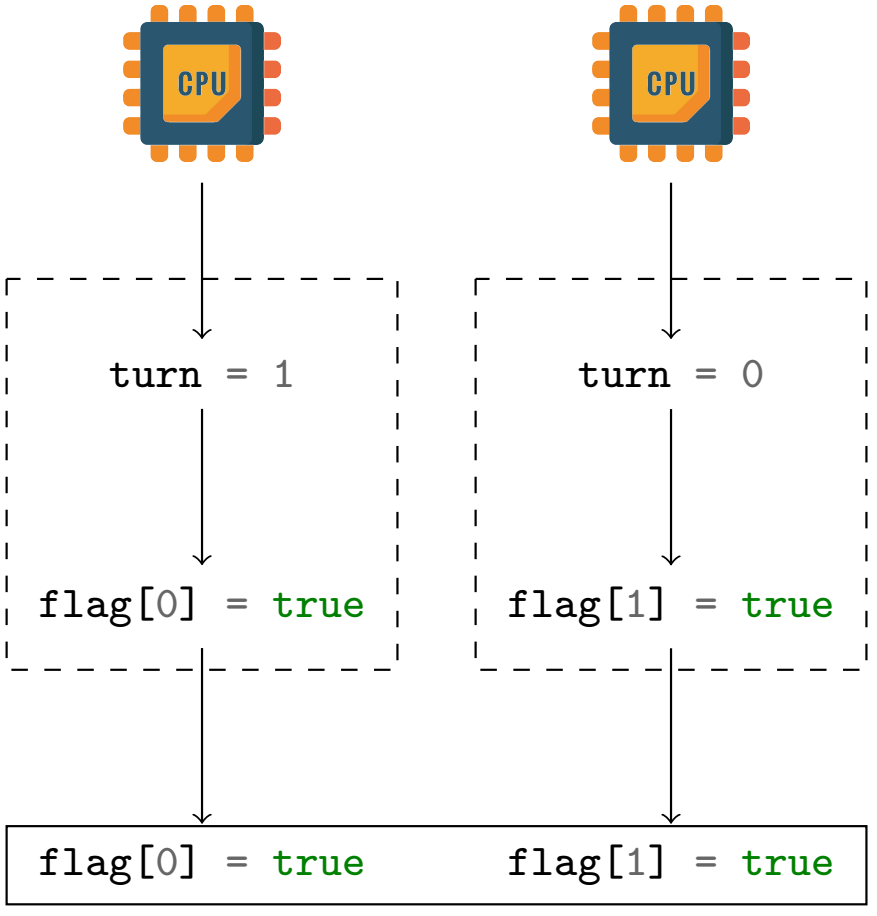
```
bool flag[2];
int victim;

void lock()
{
    int i = omp_get_thread_num();
    flag[i] = true;
    turn = 1 - i;
    while (turn != i && flag[1-i])
        {};
}
```



Échec du *Peterson Lock* en présence de *Store Buffering*

Le « *Peterson Lock* »

```
bool flag[2];
int victim;

void lock()
{
    int i = omp_get_thread_num();
    flag[i] = true;
    turn = 1 - i;
    while (turn != i && flag[1-i])
        {};
}
```



Le *Message Passing*

Producer

```
void produce(void *payload)
{
    msg = payload;
    flag = true;
}
```

Receiver

```
void * receive()
{
    while (!flag) {}; // wait
    return msg;
}
```

Le *Message Passing*

Producer

```
void produce(void *payload)
{
    msg = payload;
    flag = true;
}
```

Receiver

```
void * receive()
{
    while (!flag) {}; // wait
    return msg;
}
```

- Fonctionne en cas de *Total Store Ordering*
- x86, SPARC, ...

Le *Message Passing*

Producer

```
void produce(void *payload)
{
    msg = payload;
    flag = true;
}
```

Receiver

```
void * receive()
{
    while (!flag) {}; // wait
    return msg;
}
```

- Ne fonctionne pas sur ARM, POWER, ...
- Les threads ne *voient* pas les écritures dans le même ordre !

Le *Message Passing*

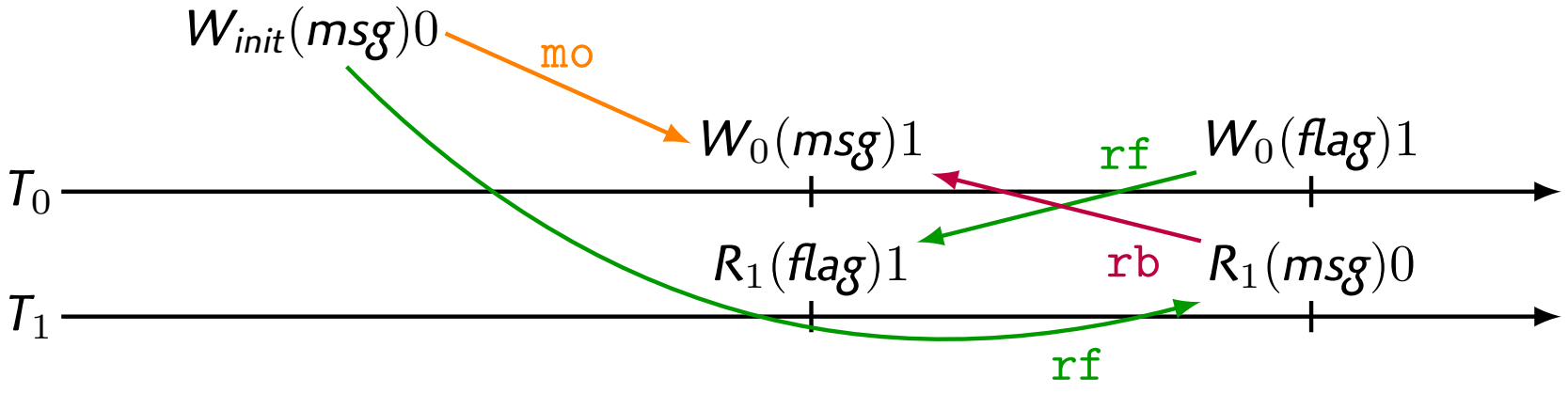
Producer

```
void produce(void *payload)
{
    msg = payload;
    flag = true;
}
```

Receiver

```
void * receive()
{
    while (!flag) {}; // wait
    return msg;
}
```

- Ne fonctionne pas sur ARM, POWER, ...
- Les threads ne *voient* pas les écritures dans le même ordre !



Modèle mémoire dans OpenMP

Bien sûr les threads ont accès à la mémoire partagée, mais...

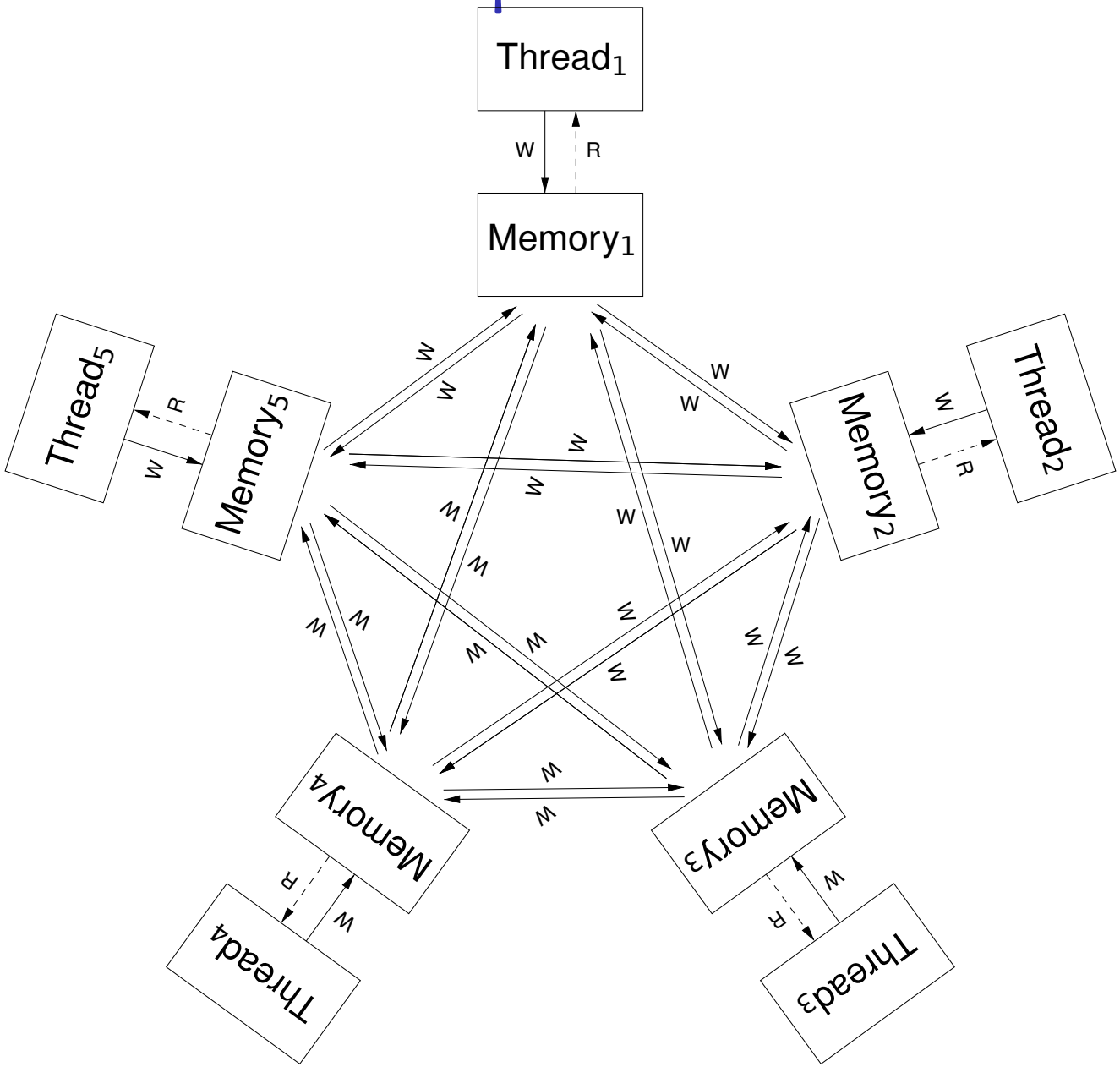
Un thread a une **vue temporaire privée** de la mémoire

- Pas forcément synchronisée en permanence.
- Une lecture **peut** provenir de la vue temporaire privée.
- Une écriture **peut** rester dans la vue temporaire privée.
- Synchronisation (implicite) lors de :
  - `#pragma omp barrier`
  - sortie de `#pragma omp for/sections/single`
  - entrée/sortie de `#pragma omp parallel/critical/atomic`
  - *Task Scheduling Points*
- Synchronisation (explicite) avec `#pragma omp flush`

Opérations atomiques (`#pragma omp atomic`)

- Obéissent à la *sequential consistency*

Représentation schématique



(image : A Tutorial Introduction to the ARM and POWER Relaxed Memory Models)

Tout le monde en passe inévitablement par là

- Bon, mais alors si on évite `i++`, ça va aller, non ?
- J'ai lu dans la doc de mon CPU que les lectures/écritures alignées étaient atomiques ; si on se tient à ça, ça va aller ?

Crise d'adolescence

Laissons tomber la règle d'or.

Guru switch ⟶

Safety

World of PAIN

Règle d'or de la programmation multithreads

Tous les accès potentiellement conflictuels* aux variables partagées doivent être protégés (`atomic`, `critical`, ...).

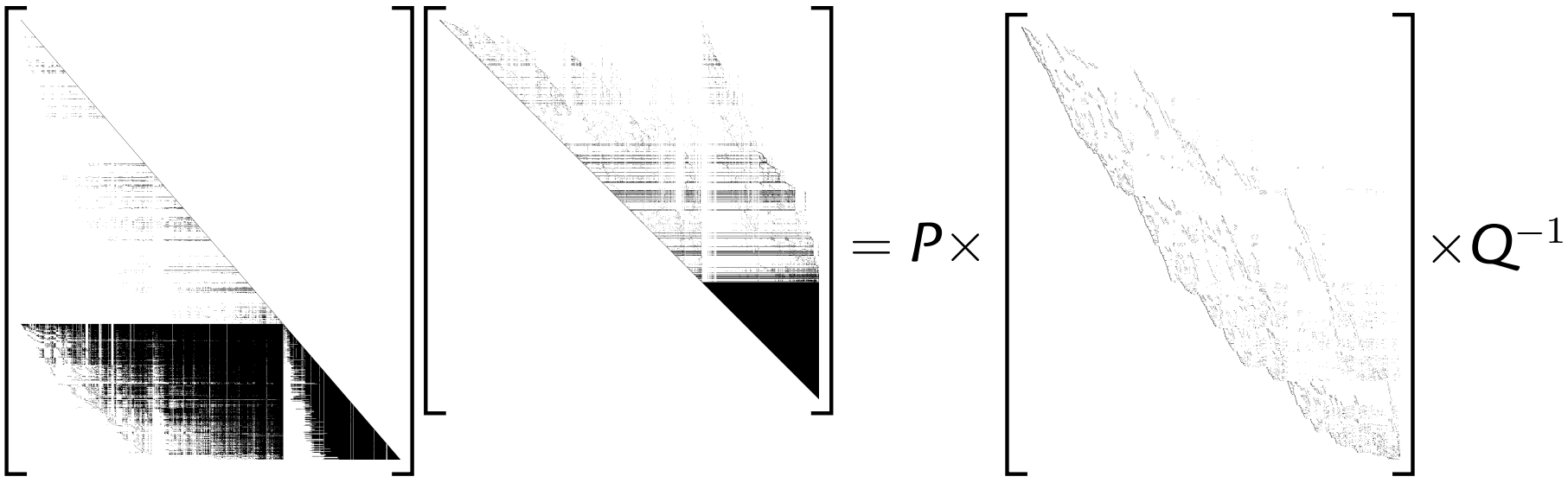
* au moins l'un d'entre eux est une écriture.

Idée générale n°2 : procéder par « phases »

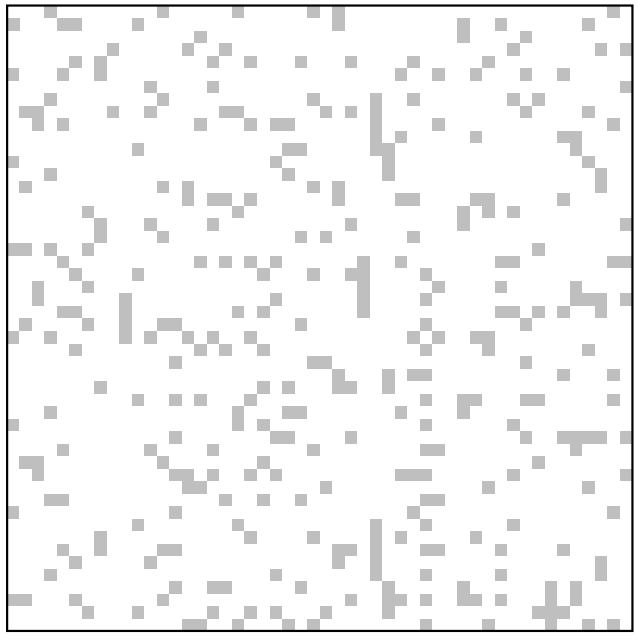
- Accepter une réduction du degré de parallélisme...

- ... Pour éliminer complètement les conflits

Exemple : factorisation LU creuse



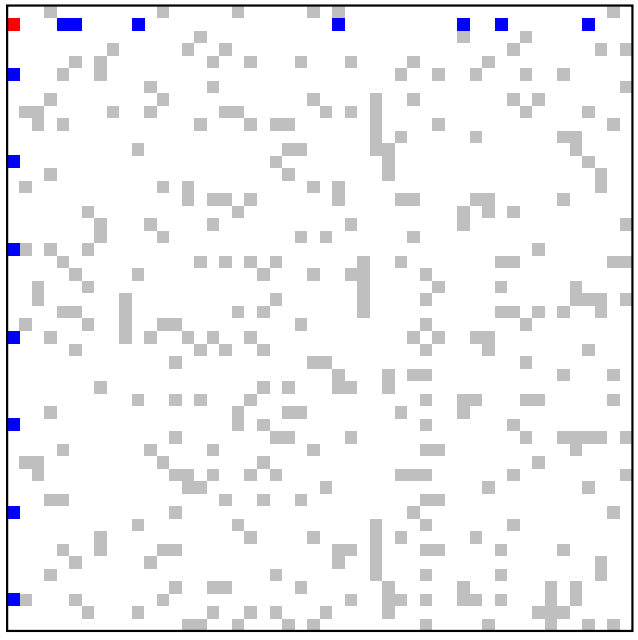
Exemple : factorisation LU creuse



Algorithme

1. [Début.] $j \leftarrow 1$
2. [Pivot.] Trouver i avec $M_{ij} \neq 0$
3. [Élimination.] Pour $M_{i'j} \neq 0$ avec $i' \neq i$, faire $M_{i'} \leftarrow (\ldots) \times M_i$.
4. [Boucle.] Incrémenter j. Si $j = n$, STOP. Sinon retourner en 2.

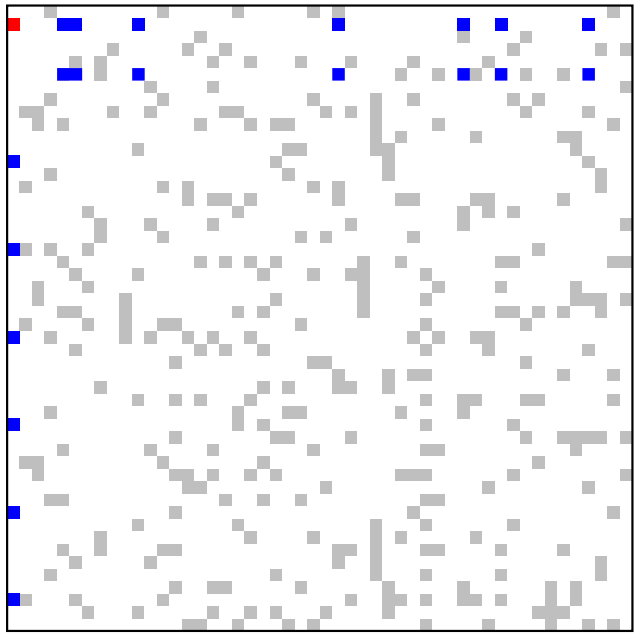
Exemple : factorisation LU creuse



Algorithme

1. [Début.] $j \leftarrow 1$
2. [Pivot.] Trouver i avec $M_{ij} \neq 0$
3. [Élimination.] Pour $M_{i'j} \neq 0$ avec $i' \neq i$, faire $M_{i'} \leftarrow (\ldots) \times M_i$.
4. [Boucle.] Incrémenter j. Si $j = n$, STOP. Sinon retourner en 2.

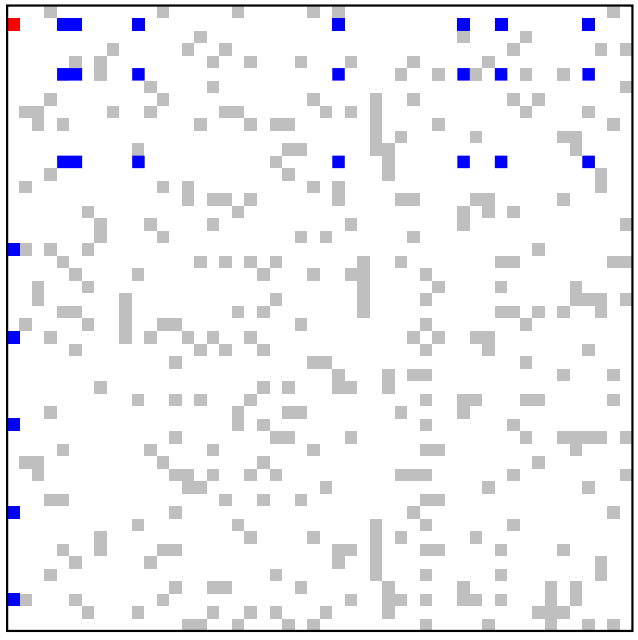
Exemple : factorisation LU creuse



Algorithme

1. [Début.] $j \leftarrow 1$
2. [Pivot.] Trouver i avec $M_{ij} \neq 0$
3. [Élimination.] Pour $M_{i'j} \neq 0$ avec $i' \neq i$, faire $M_{i'} \leftarrow (\ldots) \times M_i$.
4. [Boucle.] Incrémenter j. Si $j = n$, STOP. Sinon retourner en 2.

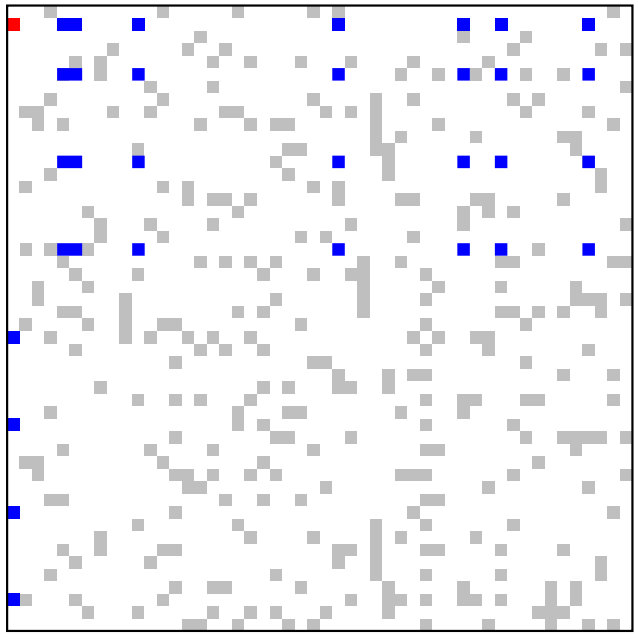
Exemple : factorisation LU creuse



Algorithme

1. [Début.] $j \leftarrow 1$
2. [Pivot.] Trouver i avec $M_{ij} \neq 0$
3. [Élimination.] Pour $M_{i'j} \neq 0$ avec $i' \neq i$, faire $M_{i'} \leftarrow (\ldots) \times M_i$.
4. [Boucle.] Incrémenter j. Si $j = n$, STOP. Sinon retourner en 2.

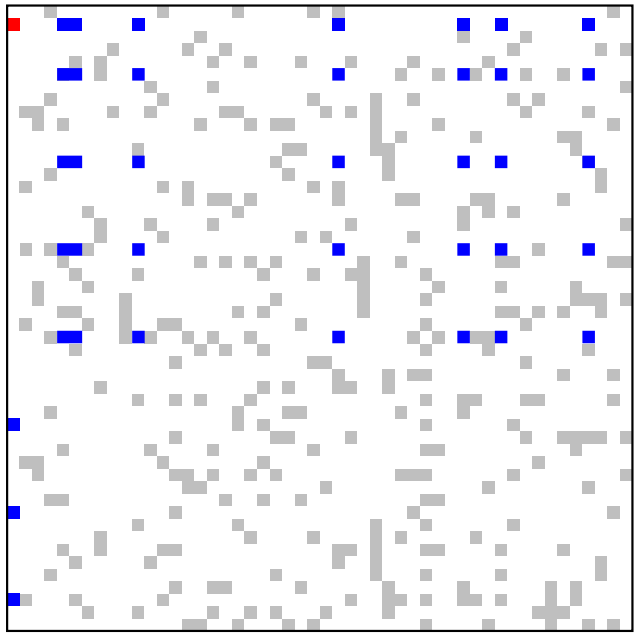
Exemple : factorisation LU creuse



Algorithme

1. [Début.] $j \leftarrow 1$
2. [Pivot.] Trouver i avec $M_{ij} \neq 0$
3. [Élimination.] Pour $M_{i'j} \neq 0$ avec $i' \neq i$, faire $M_{i'} \leftarrow (\ldots) \times M_i$.
4. [Boucle.] Incrémenter j. Si $j = n$, STOP. Sinon retourner en 2.

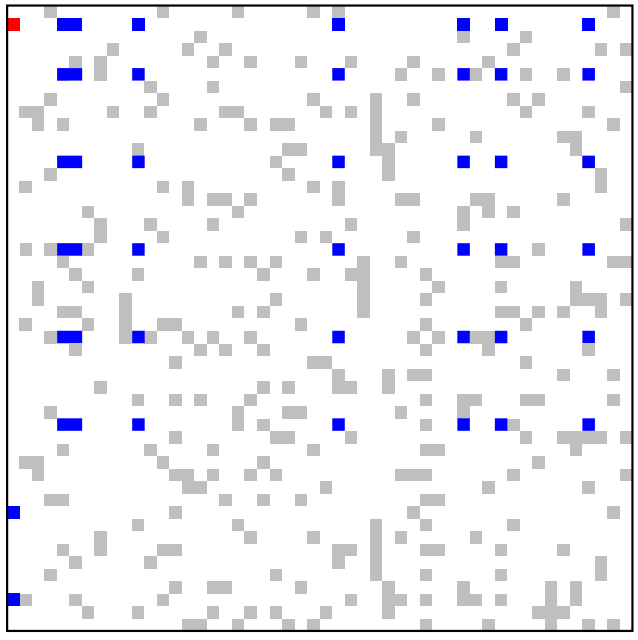
Exemple : factorisation LU creuse



Algorithme

1. [Début.] $j \leftarrow 1$
2. [Pivot.] Trouver i avec $M_{ij} \neq 0$
3. [Élimination.] Pour $M_{i'j} \neq 0$ avec $i' \neq i$, faire $M_{i'} \leftarrow (\ldots) \times M_i$.
4. [Boucle.] Incrémenter j. Si $j = n$, STOP. Sinon retourner en 2.

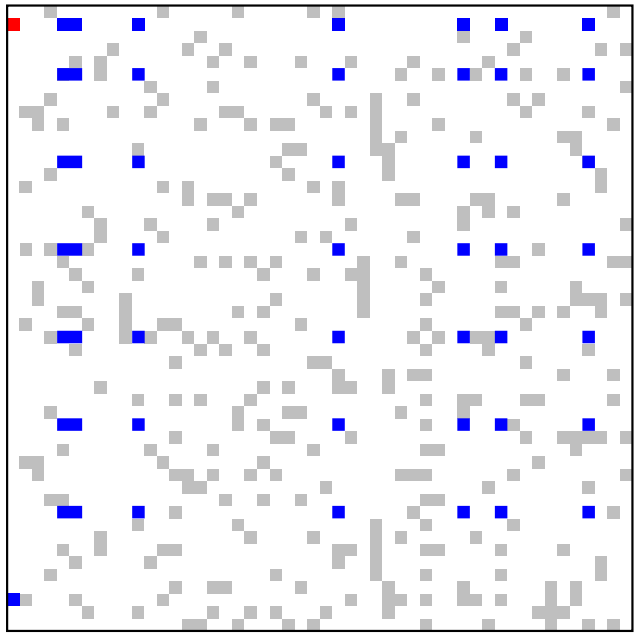
Exemple : factorisation LU creuse



Algorithme

1. [Début.] $j \leftarrow 1$
2. [Pivot.] Trouver i avec $M_{ij} \neq 0$
3. [Élimination.] Pour $M_{i'j} \neq 0$ avec $i' \neq i$, faire $M_{i'} \leftarrow (\ldots) \times M_i$.
4. [Boucle.] Incrémenter j. Si $j = n$, STOP. Sinon retourner en 2.

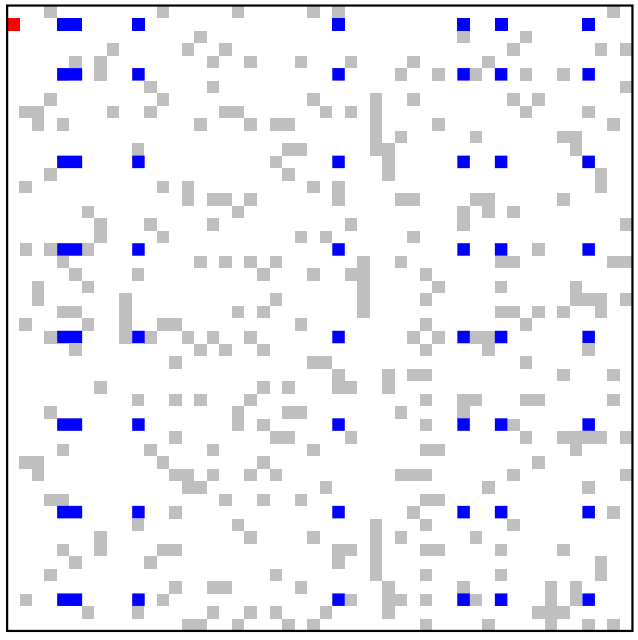
Exemple : factorisation LU creuse



Algorithme

1. [Début.] $j \leftarrow 1$
2. [Pivot.] Trouver i avec $M_{ij} \neq 0$
3. [Élimination.] Pour $M_{i'j} \neq 0$ avec $i' \neq i$, faire $M_{i'} \leftarrow (\ldots) \times M_i$.
4. [Boucle.] Incrémenter j. Si $j = n$, STOP. Sinon retourner en 2.

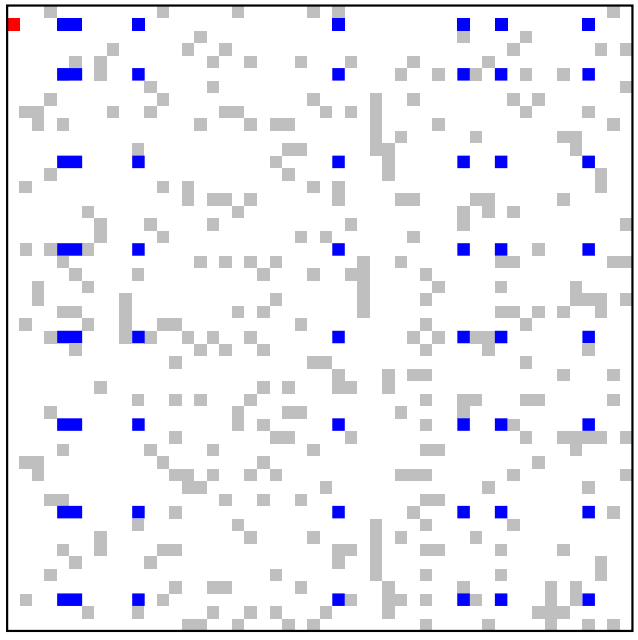
Exemple : factorisation LU creuse



Algorithme

1. [Début.] $j \leftarrow 1$
2. [Pivot.] Trouver i avec $M_{ij} \neq 0$
3. [Élimination.] Pour $M_{i'j} \neq 0$ avec $i' \neq i$, faire $M_{i'} \leftarrow (\ldots) \times M_i$.
4. [Boucle.] Incrémenter j. Si $j = n$, STOP. Sinon retourner en 2.

Exemple : factorisation LU creuse



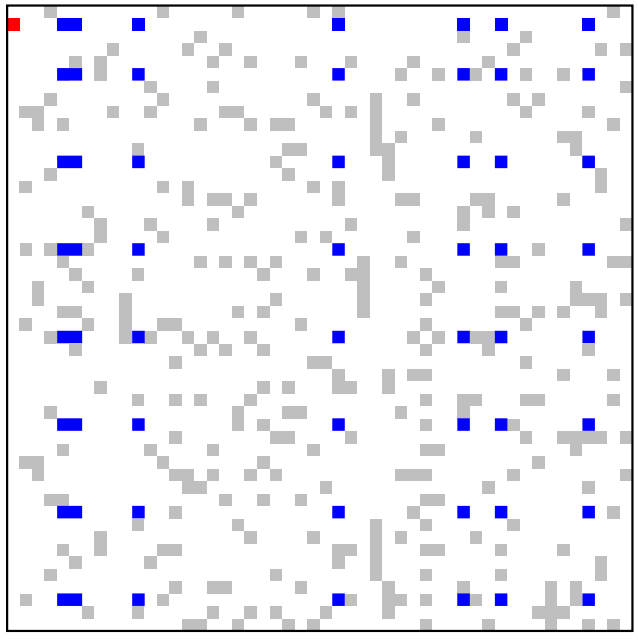
Plusieurs colonnes en parallèle ?

- Conflit d'accès aux lignes !

Algorithme

1. [Début.] $j \leftarrow 1$
2. [Pivot.] Trouver i avec $M_{ij} \neq 0$
3. [Élimination.] Pour $M_{i'j} \neq 0$ avec $i' \neq i$, faire $M_{i'} \leftarrow (\ldots) \times M_i$.
4. [Boucle.] Incrémenter j. Si $j = n$, STOP. Sinon retourner en 2.

Exemple : factorisation LU creuse



Plusieurs colonnes en parallèle ?

- Conflit d'accès aux lignes !

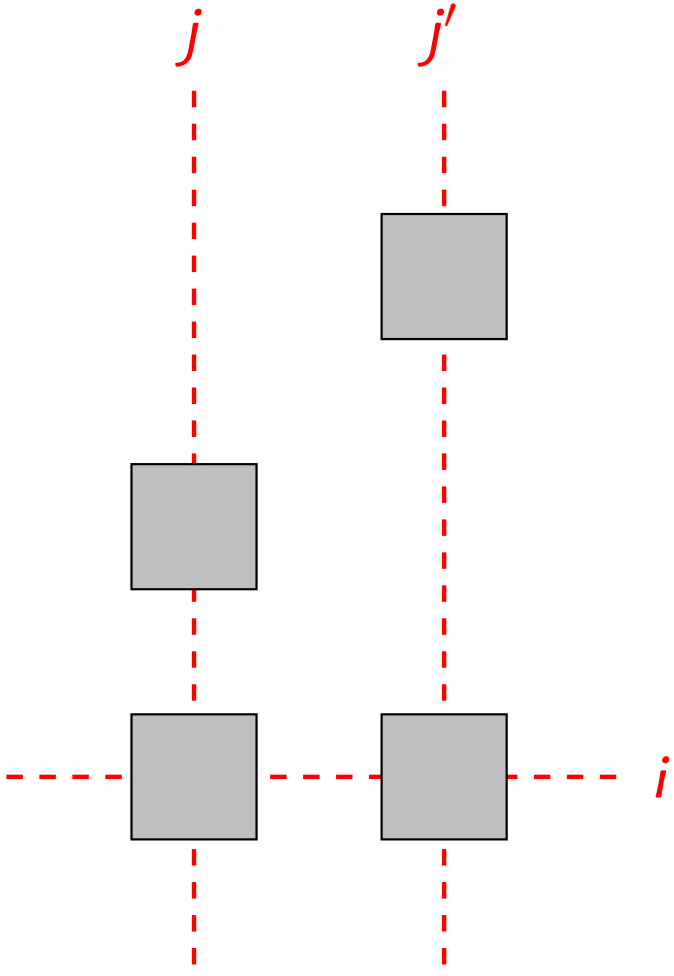
Solution

Identifier DES colonnes *indépendantes*.

Algorithme

1. [Début.] $j \leftarrow 1$
2. [Pivot.] Trouver i avec $M_{ij} \neq 0$
3. [Élimination.] Pour $M_{i'j} \neq 0$ avec $i' \neq i$, faire $M_{i'} \leftarrow (\ldots) \times M_i$.
4. [Boucle.] Incrémenter j. Si $j = n$, STOP. Sinon retourner en 2.

Exemple : factorisation LU creuse



Dépendances

- Colonnes j et j' liées par ligne i.

Graphe de dépendance G_{dep}

- Sommets V = ens. des colonnes.
- Arêtes :

$$E = \{j \leftrightarrow j' \ : \ \exists i.M_{ij} \neq 0 \wedge M_{ij'} \neq 0\}.$$

Colonnes indépendantes

$\rightarrow$ Ensemble indépendant dans G_{dep}.

Exemple : factorisation LU creuse

Nouvel algorithme :

- Tant que ce n'est pas fini :
  - Trouver un ensemble indépendant $\mathcal{I}$ dans G_{dep}.
    - (optimal = NP-dur. Ici : algorithme glouton séquentiel)
  - Éliminer toutes les colonnes de $\mathcal{I}$ en parallèle.
  - « Oublier » les colonnes éliminées

Pas de conflit !

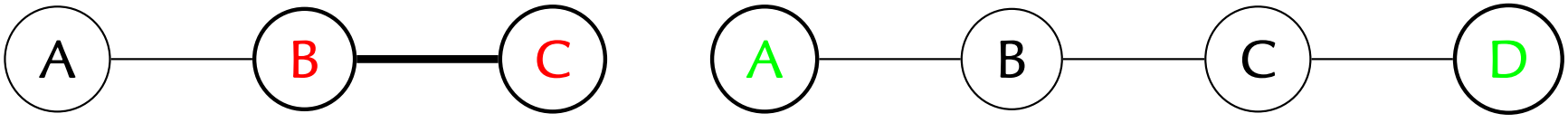
Portion séquentielle...

Exemple : factorisation LU creuse

Sous-exemple : algorithme glouton pour trouver un ensemble indépendant maximal

Algorithme (implantable en temps linéaire)

- $\mathcal{I} \leftarrow \emptyset$
- Tant que G n'est pas vide :
  - Choisir un sommet x quelconque.
  - Ajouter x à $\mathcal{I}$.
  - Retirer x et tous ses voisins de G.
- Renvoyer $\mathcal{I}$.



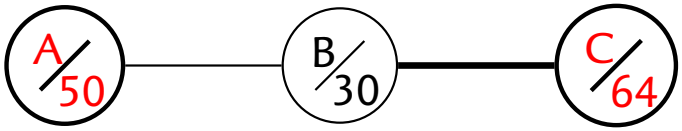
En parallèle : conflit avec ses voisins...

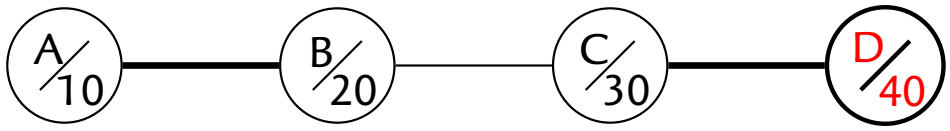
Exemple : factorisation LU creuse

Sous-exemple : algorithme glouton pour trouver un ensemble indépendant maximal

Algorithme modifié

- $\mathcal{I} \leftarrow \emptyset$
- Choisir une permutation aléatoire π (« *score* ») de $\{1, \ldots, n\}$.
- Tant que G n'est pas vide :
  - $X = \{u \in V \mid \pi[u] > \pi[v] \text{ pour tout } u \leftrightarrow v\}$
  - Ajouter X à $\mathcal{I}$.
  - Retirer X et tous les voisins des sommets de X de G.
- Renvoyer $\mathcal{I}$.





Exemple : factorisation LU creuse

Sous-exemple : algorithme glouton pour trouver un ensemble indépendant maximal

Algorithme modifié

- $\mathcal{I} \leftarrow \emptyset$
- Choisir une permutation aléatoire π (« *score* ») de $\{1, \ldots, n\}$.
- Tant que G n'est pas vide : ($\approx \log^2 n$ *iterations*)
  - $X = \{u \in V \mid \pi[u] > \pi[v]$ pour tout $u \leftrightarrow v\}$
  - Ajouter X à $\mathcal{I}$.
  - Retirer X et tous les voisins des sommets de X de G.
- Renvoyer $\mathcal{I}$.