

Cours 6 : Théorie et pratique « avancées » pour la programmation multi-threads

Charles Bouillaguet
charles.bouillaguet@lip6.fr

2020-02-28

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.

La triste vérité...



- ▶ Barrière → attente
- ▶ Critical → séquentialisation
- ▶ Atomic → plus lent qu'un accès normal

Synchronisation → limite le passage à l'échelle.

⇒ rôle important de la localité des données.

#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];
```

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

```
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];
```

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

```
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];
```

$$T \geq 200s !!!$$

$$T = 0.46s (\times 12.9)$$

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

Histogramme (par ex. `numpy.histogram`)

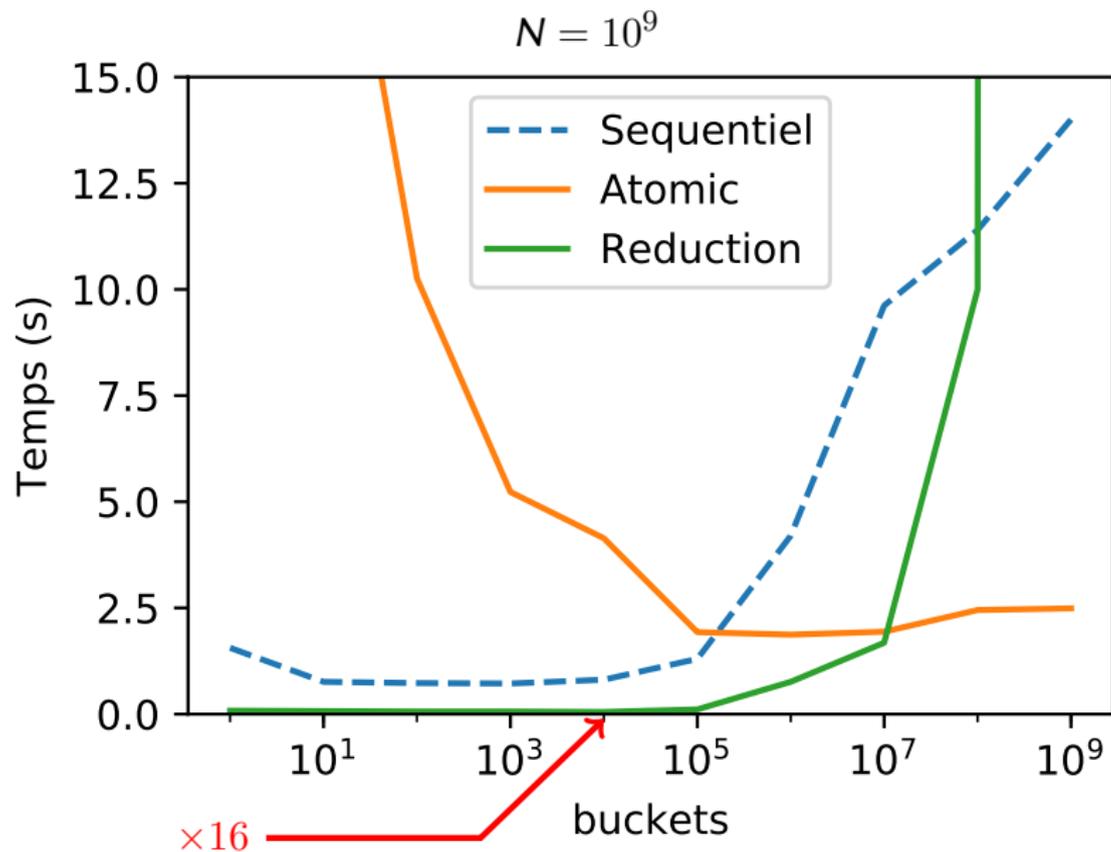
Plan A

```
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



Retour sur #pragma omp atomic

Garanties offertes

- ▶ #pragma omp atomic **garantit** que tout se passe comme si les opérations atomiques étaient exécutées séquentiellement
- ⇒ « **Sequential Consistency** »

Retour sur #pragma omp atomic

Garanties offertes

- ▶ #pragma omp atomic **garantit** que tout se passe comme si les opérations atomiques étaient exécutées séquentiellement
- ⇒ « **Sequential Consistency** »

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.
- ▶ compatible avec la cohérence de la mémoire.

Relations d'ordre entre accès à la mémoire

► Accès mémoire :

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

$R_j(x)b$: T_j 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.

► « *Memory coherence* » :

$w \xrightarrow{rf} r$: la lecture r renvoie la valeur écrite par w .

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

$r \xrightarrow{fr} w$: la lecture r a lieu avant l'écriture w .

Relations d'ordre entre accès à la mémoire

► Accès mémoire :

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

$R_j(x)b$: T_j lit la variable x et la valeur b .

► « Program Order » :

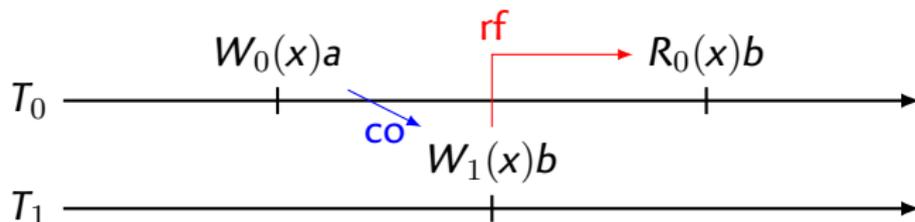
$x \xrightarrow{po} y$: le code demande qu'on fasse x d'abord et y après.

► « Memory coherence » :

$w \xrightarrow{rf} r$: la lecture r renvoie la valeur écrite par w .

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

$r \xrightarrow{fr} w$: la lecture r a lieu avant l'écriture w .



Relations d'ordre entre accès à la mémoire

► Accès mémoire :

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

$R_j(x)b : T_j$ 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.

► « Memory coherence » :

$w \xrightarrow{rf} r$: la lecture r renvoie la valeur écrite par w .

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

$r \xrightarrow{fr} w$: la lecture r a lieu avant l'écriture w .

Theorem

Sequential Consistency \iff pas de cycles avec $\xrightarrow{po} \cup \xrightarrow{rf} \cup \xrightarrow{co} \cup \xrightarrow{fr}$.

Application : preuve de correction du *Peterson Lock*

Le « *Peterson Lock* » : exclusion mutuelle pour 2 threads

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

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

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

▶ 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 :

$T_0 : W_0(\text{flag}[0])\text{true} \xrightarrow{po} W_0(\text{victim})0 \xrightarrow{po} R_0(\text{flag}[1])\text{false} \xrightarrow{po} R_0(\text{victim})1 \xrightarrow{po} CS_0$

$T_1 : W_1(\text{flag}[1])\text{true} \xrightarrow{po} W_1(\text{victim})1 \xrightarrow{po} R_1(\text{flag}[0])\text{false} \xrightarrow{po} R_1(\text{victim})0 \xrightarrow{po} CS_1$

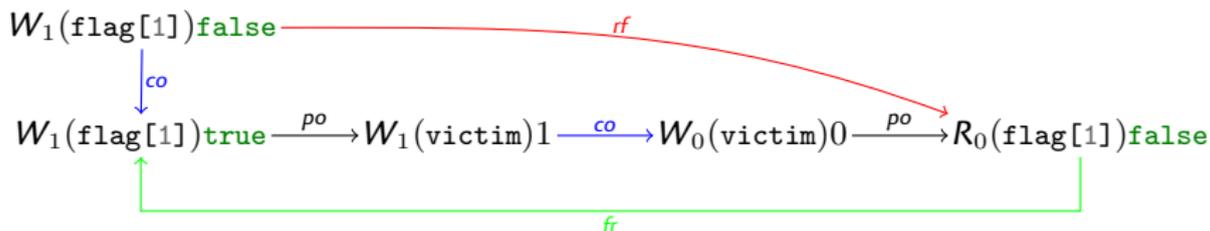
▶ Supposons que T_0 écrive `victim` en dernier :

$W_1(\text{victim})1 \xrightarrow{co} W_0(\text{victim})0.$

▶ T_0 sort de la boucle, et `victim = 0`, donc forcément :

$W_0(\text{victim})0 \xrightarrow{po} R_0(\text{flag}[1])\text{false}.$

▶ Si on met tout ceci bout-à-bout :



▶ Cycle \Rightarrow non-SC \Rightarrow Contradiction !

LIVE DEMO

Peterson Lock



- ▶ Mon laptop n'est pas séquentiellement consistant...

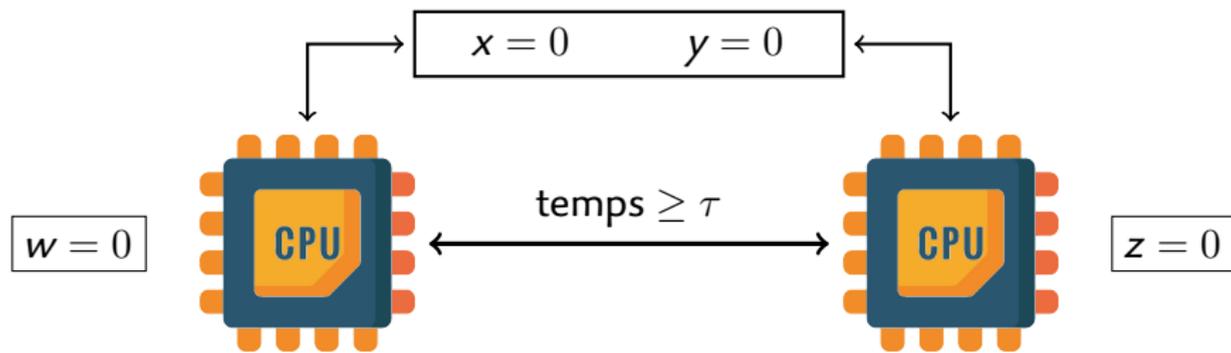
Peterson Lock



- ▶ Mon laptop n'est pas séquentiellement consistant...
- ▶ Les vôtres non plus !

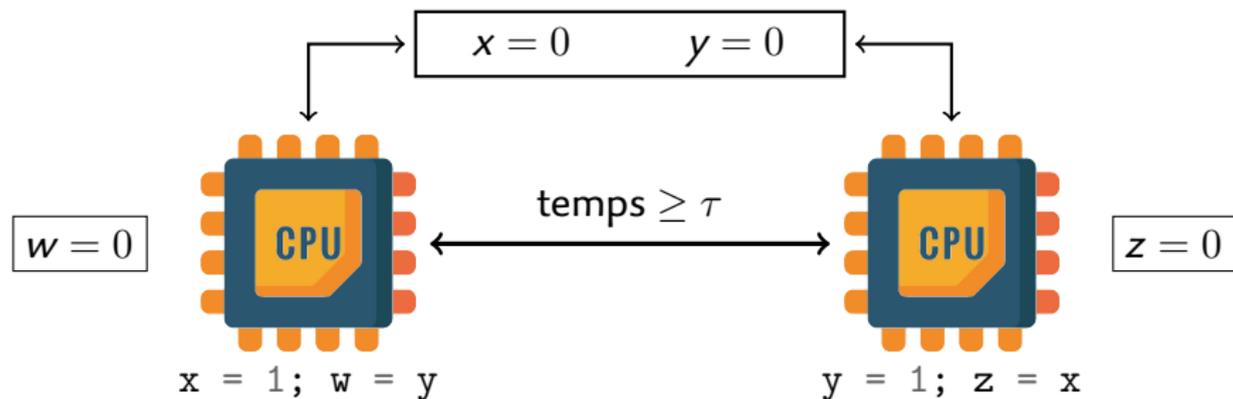
Les CPU ne sont pas séquentiellement cohérents !

La *Sequential Consistency* est coûteuse



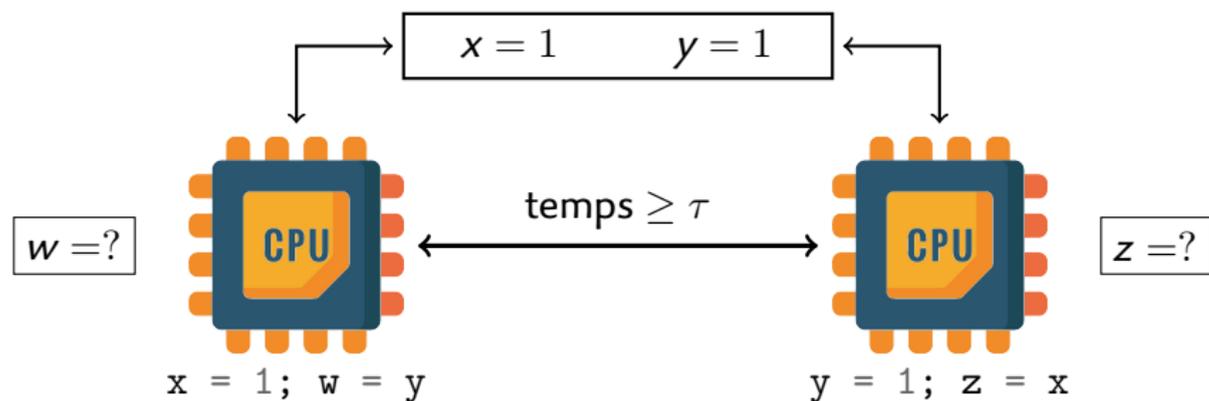
Les CPU ne sont pas séquentiellement consistents !

La *Sequential Consistency* est coûteuse



Les CPU ne sont pas séquentiellement cohérents !

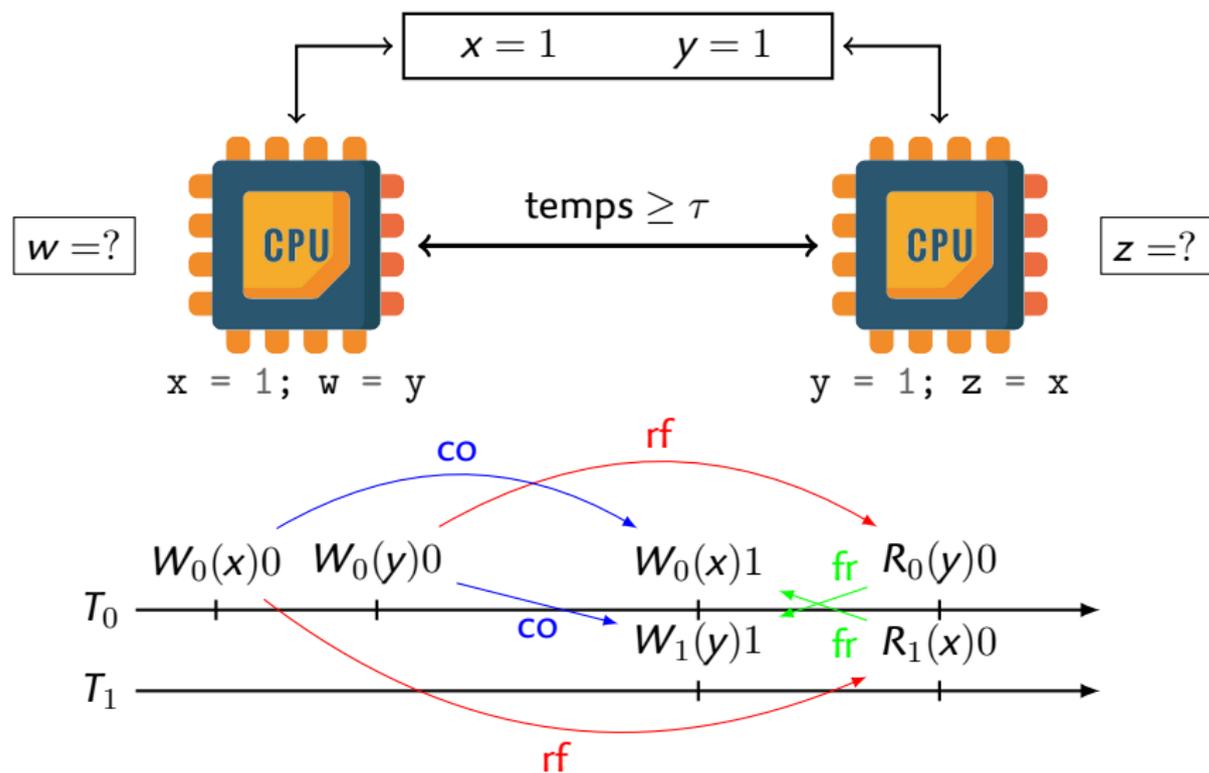
La *Sequential Consistency* est coûteuse



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

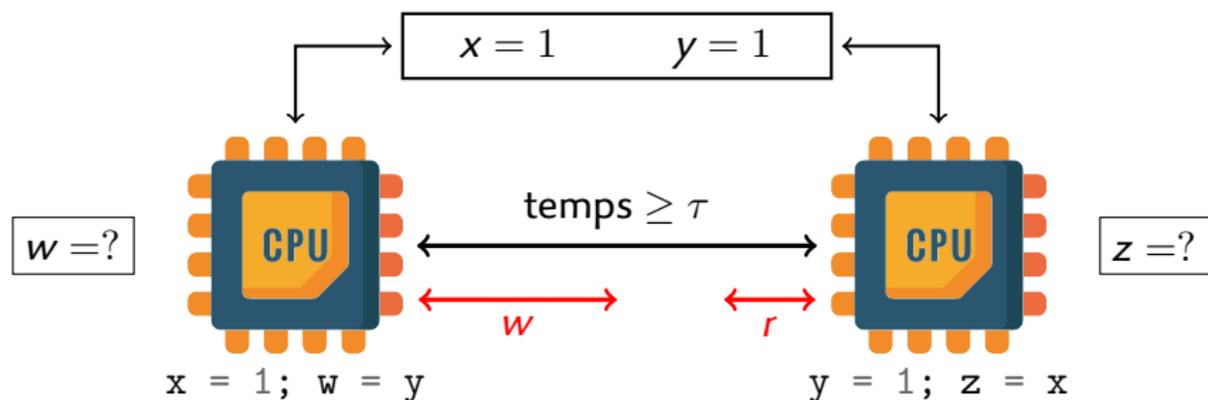
Les CPU ne sont pas séquentiellement consistents !

La *Sequential Consistency* est coûteuse



Les CPU ne sont pas séquentiellement consistents !

La *Sequential Consistency* est coûteuse

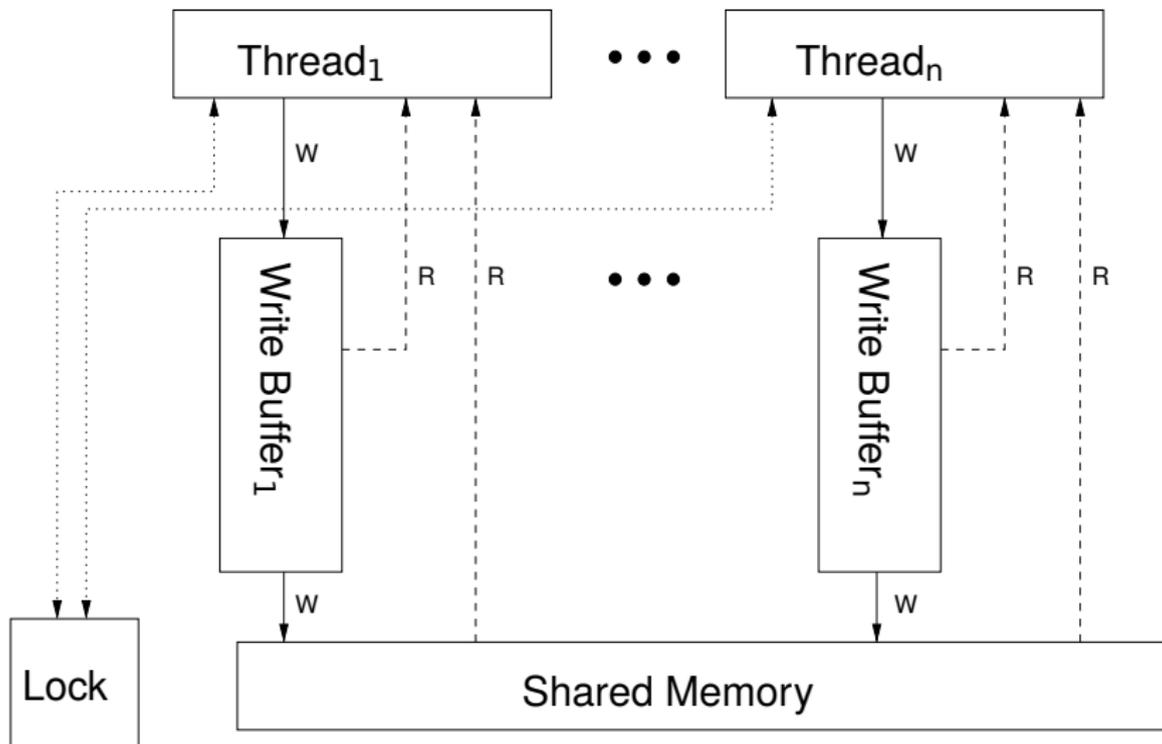


r : temps min. pour faire une lecture
 w : temps min. pour faire une écriture

Sequential Consistency $\implies r + w \geq \tau$

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, SPARC, etc.

Chaque thread **matériel** possède un **Store Buffer**

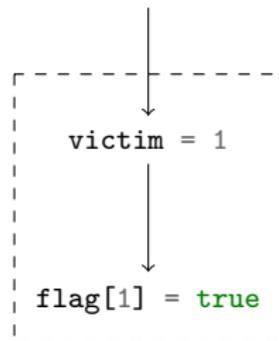
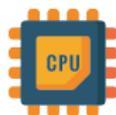
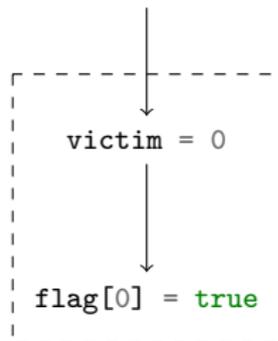
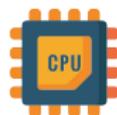
- ▶ Mes écritures sont placées en attente dans mon *Store Buffer*
- ▶ C'est une file (on ne double pas !)
- ▶ Je vois mes écritures tout de suite
- ▶ Mon *Store Buffer* sera « purgé » vers la mémoire... à terme
- ▶ À ce moment-là :
 - ▶ *Tous les autres threads* voient mes écritures.
 - ▶ Ils les voient 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;
    victim = i;
    while (flag[1-i] && victim == i)
        {};
}
```



flag[0] = false

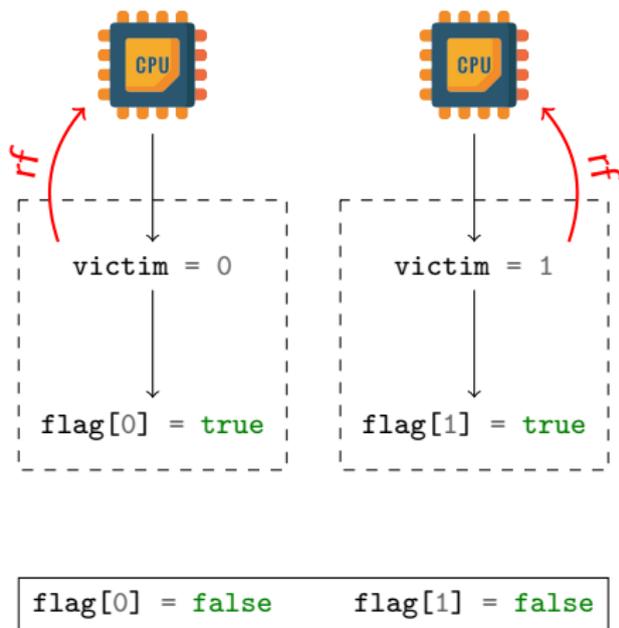
flag[1] = false

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;
    victim = i;
    while (flag[1-i] && victim == i)
        {};
}
```

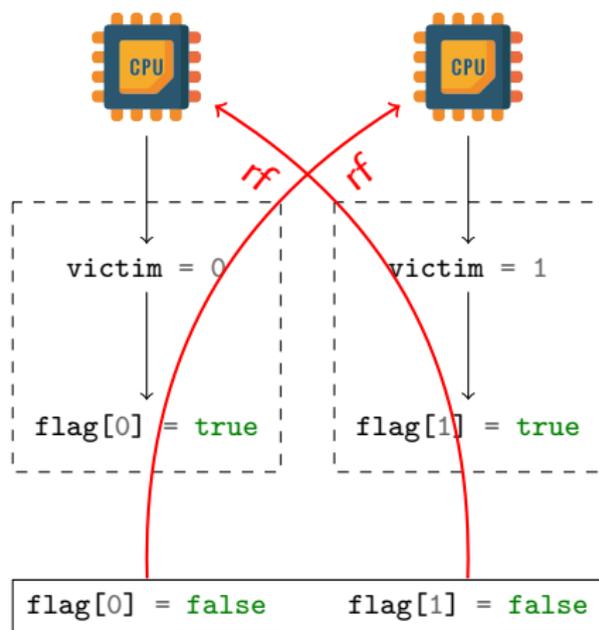


Échec du *Peterson Lock* en présence de *Store Buffering*

Le « *Peterson Lock* »

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

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

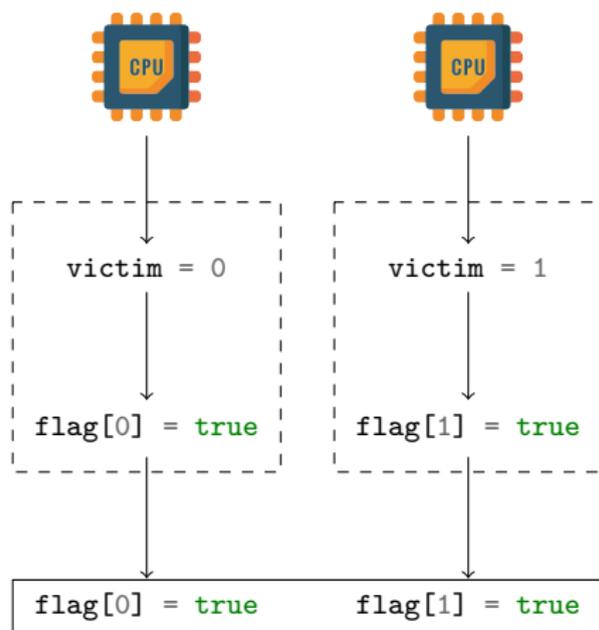


Échec du *Peterson Lock* en présence de *Store Buffering*

Le « *Peterson Lock* »

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

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



Le Message Passing

Sender

```
msg = data;  
flag = 1;
```

Receiver

```
while (flag == 0) {}; // wait  
recv = msg;
```

Le Message Passing

Sender

```
msg = data;  
flag = 1;
```

Receiver

```
while (flag == 0) {}; // wait  
recv = msg;
```



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

Le Message Passing

Sender

```
msg = data;  
flag = 1;
```

Receiver

```
while (flag == 0) {}; // wait  
recv = msg;
```



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

Le Message Passing

Sender

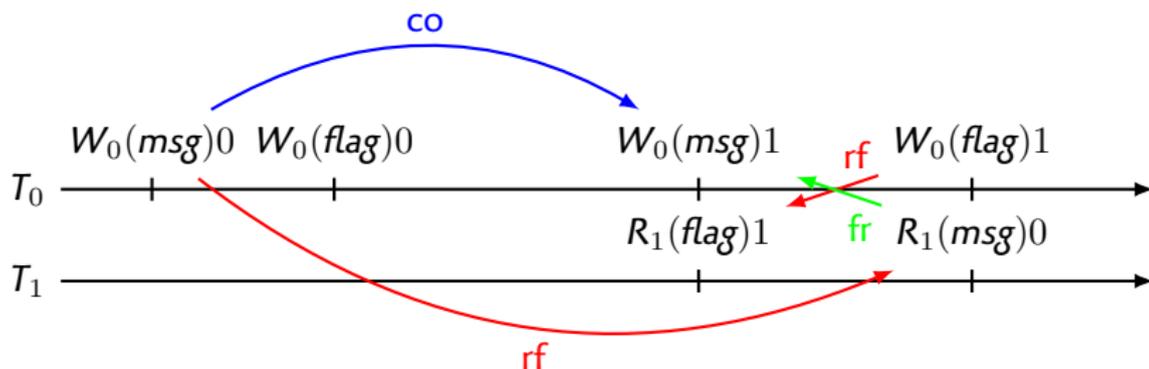
```
msg = data;  
flag = 1;
```

Receiver

```
while (flag == 0) {}; // wait  
recv = msg;
```



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



Comment peut-on s'en sortir?!?

Instructions CPU x86 de **barrière mémoire**

- mfence** (*Full Memory Fence*) tous les accès à la mémoire qui précèdent sont terminés (et globalement visibles) avant qu'un accès qui suit ne puisse avoir lieu.
Latence $\approx 35 - 40$ cycles.
- lfence** (*Load-Load Fence*) toutes les lectures qui précèdent sont terminées avant qu'une lecture qui suit ne puisse avoir lieu.
Latence $\approx 4 - 6$ cycles.
- sfence** (*Store-Store Fence*) toutes les écritures qui précèdent **sfence** sont terminées (et globalement visibles) avant qu'une écriture qui suit ne puisse avoir lieu.
Latence $\approx 5 - 7$ cycles.

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`

non-cohérence des CPUs, compilateur (variables dans registres)

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.

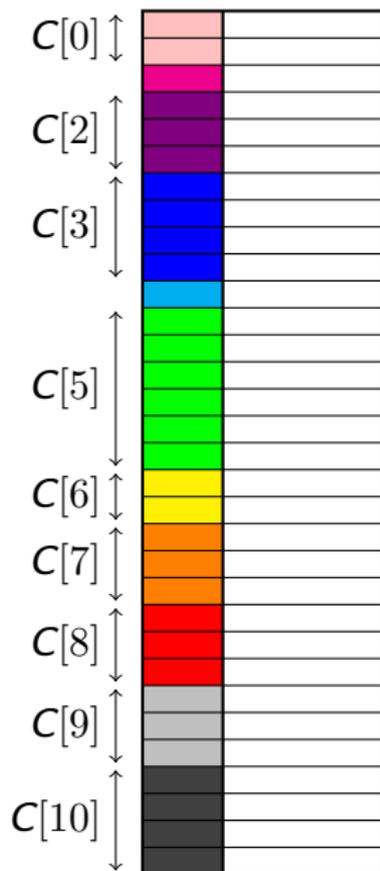
Exemple : Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



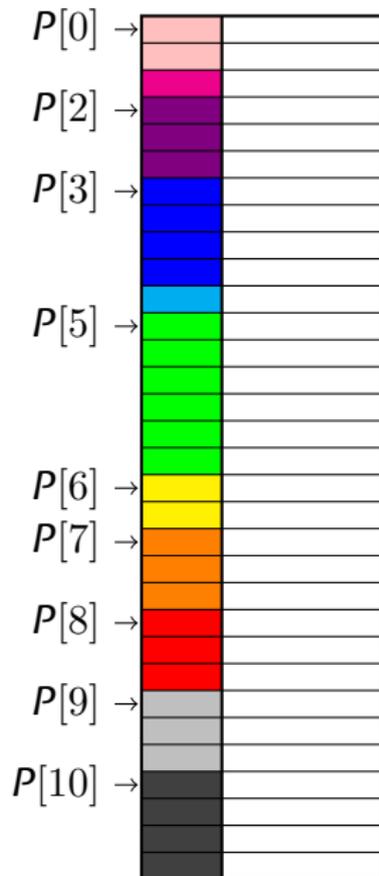
Exemple : Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



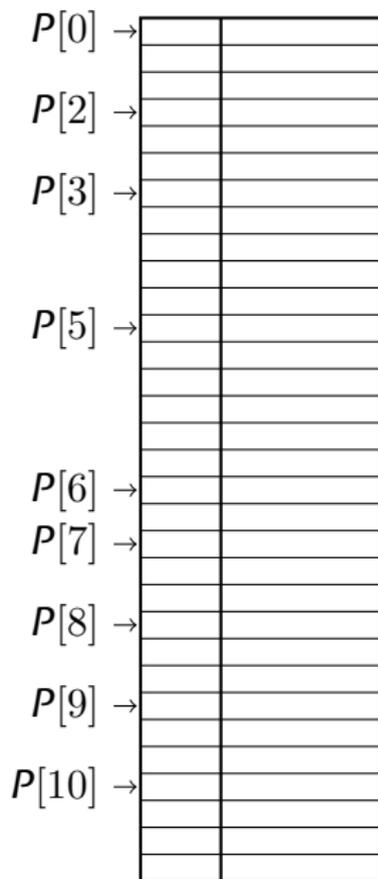
Exemple : Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}

// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



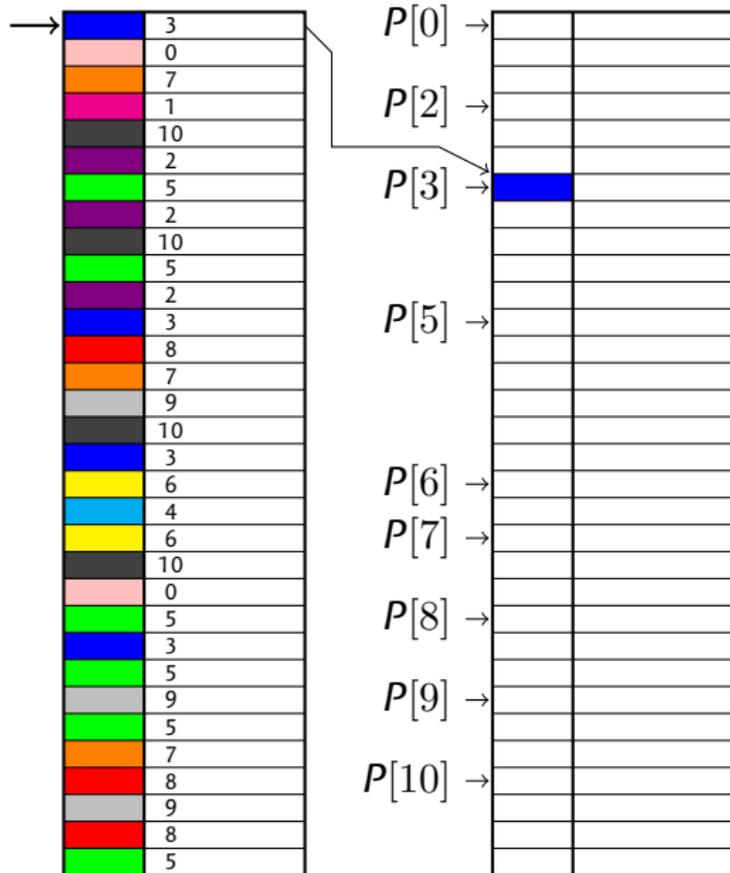
Exemple : Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}
```

```
// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}
```

```
// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}
```

```
// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



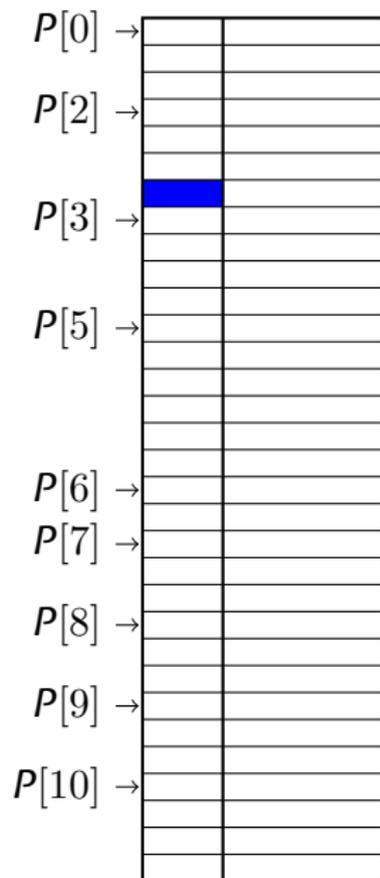
Exemple : Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}
```

```
// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}
```

```
// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}
```

```
// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



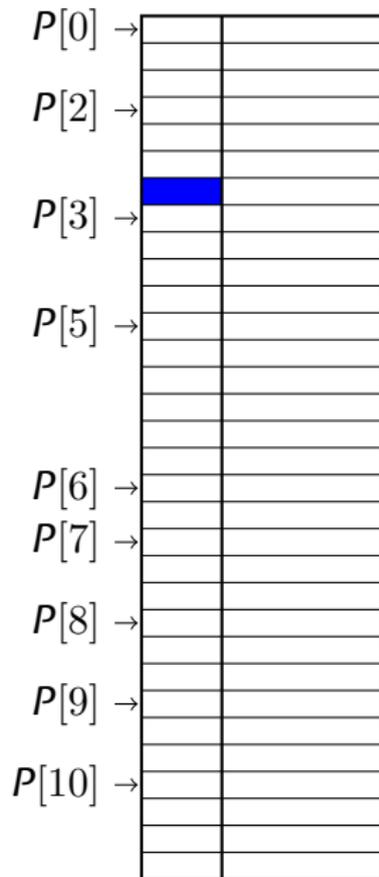
Exemple : Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}
```

```
// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}
```

```
// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}
```

```
// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



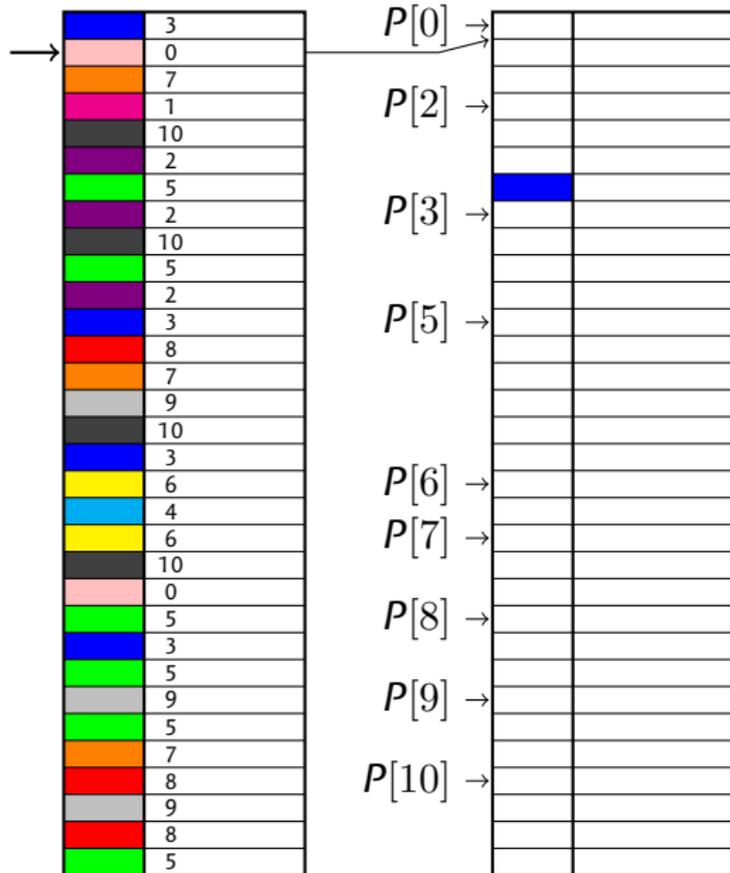
Exemple : Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}
```

```
// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}
```

```
// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}
```

```
// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



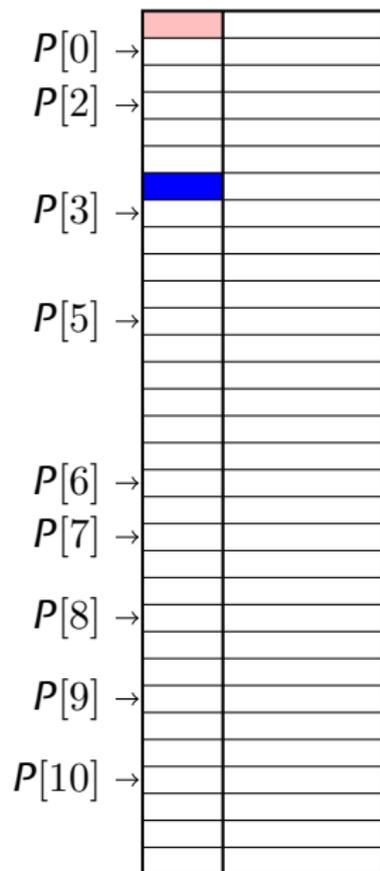
Exemple : Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}
```

```
// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}
```

```
// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}
```

```
// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



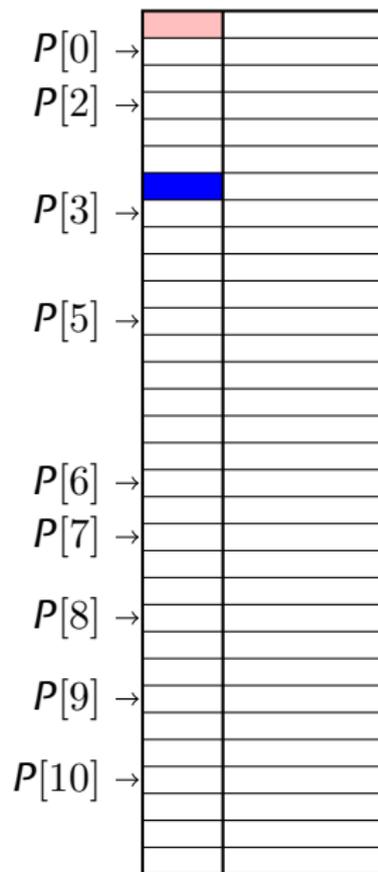
Exemple : Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}
```

```
// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}
```

```
// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}
```

```
// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



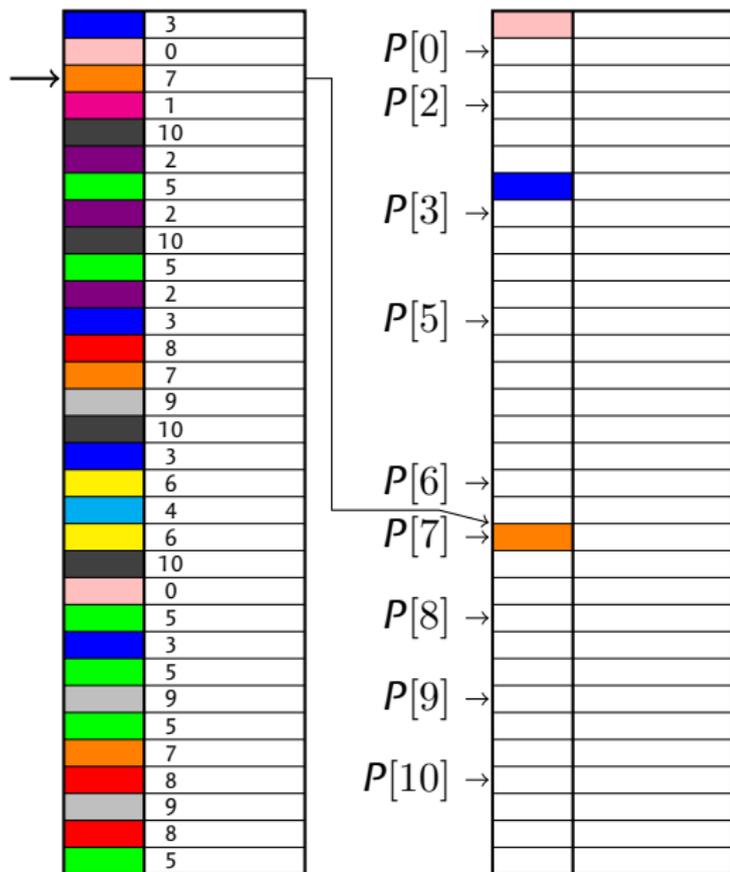
Exemple : Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}
```

```
// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}
```

```
// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}
```

```
// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



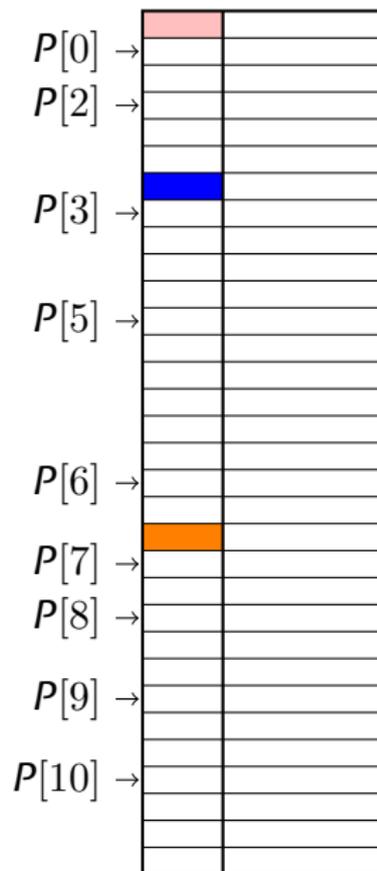
Exemple : Bucket Sort

```
// Initialization
for (int i = 0; i < M; i++) {
    C[i] = 0;
}
```

```
// Histogram
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}
```

```
// Prefix-sum
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}
```

```
// Dispatch
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    B[P[bucket]] = A[i];
    P[bucket]++;
}
```



Exemple : Bucket Sort

Parallélisation directe naïve

```
// Counting
```

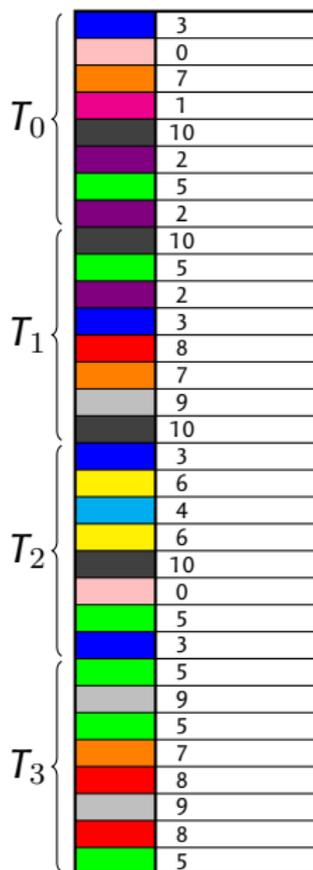
```
for (int i = 0; i < N; i++) {  
    int bucket = f(A[i]);  
    C[bucket]++;  
}
```

```
// Prefix-sum
```

```
int s = 0;  
for (int i = 0; i < M; i++) {  
    P[i] = s;  
    s += C[i];  
}
```

```
// Dispatch
```

```
for (int i = 0; i < N; i++) {  
    int bucket = f(A[i]);  
    int ptr;  
  
    ptr = P[bucket]++;  
    B[ptr] = A[i];  
}
```



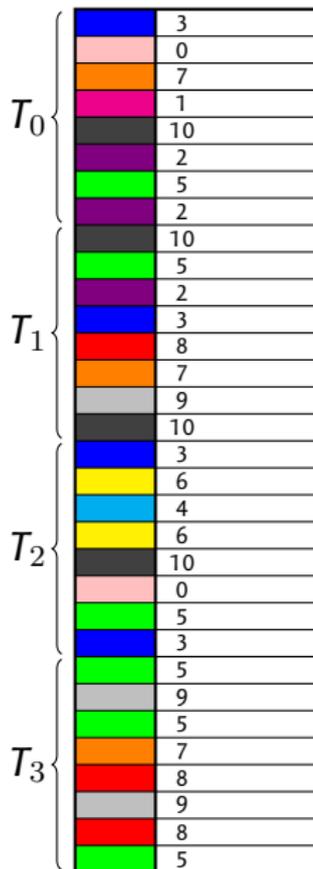
Exemple : Bucket Sort

Parallélisation directe naïve

```
// Counting
#pragma omp parallel for reduction(+:C[0:M])
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    C[bucket]++;
}

// Prefix-sum (sequential)
int s = 0;
for (int i = 0; i < M; i++) {
    P[i] = s;
    s += C[i];
}

// Dispatch
#pragma omp parallel for
for (int i = 0; i < N; i++) {
    int bucket = f(A[i]);
    int ptr;
    #pragma omp atomic capture
    ptr = P[bucket]++;
    B[ptr] = A[i];
}
```



Idée générale n°1 : réorganiser

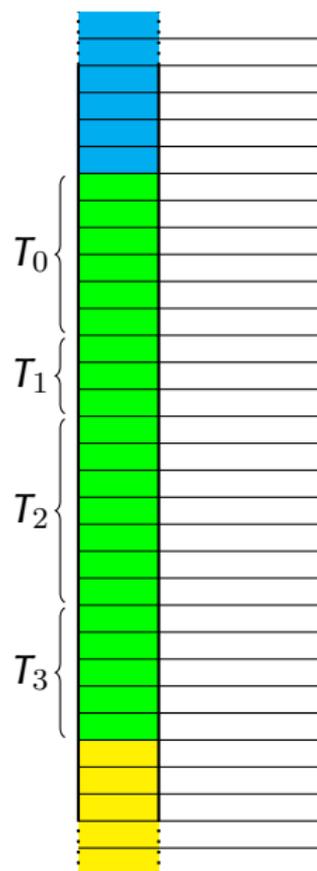
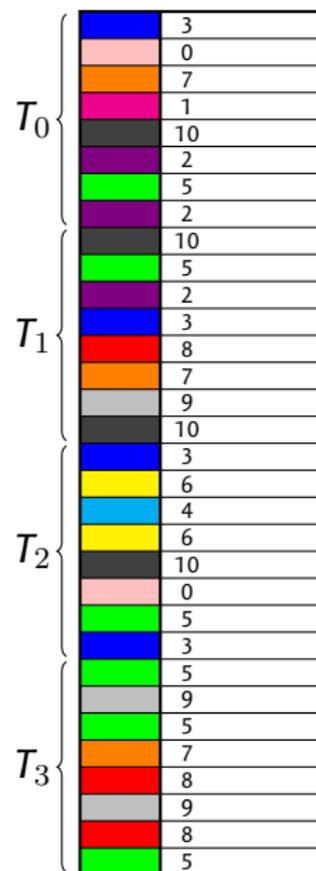


- ▶ Faire un (tout) petit peu de calculs en plus...

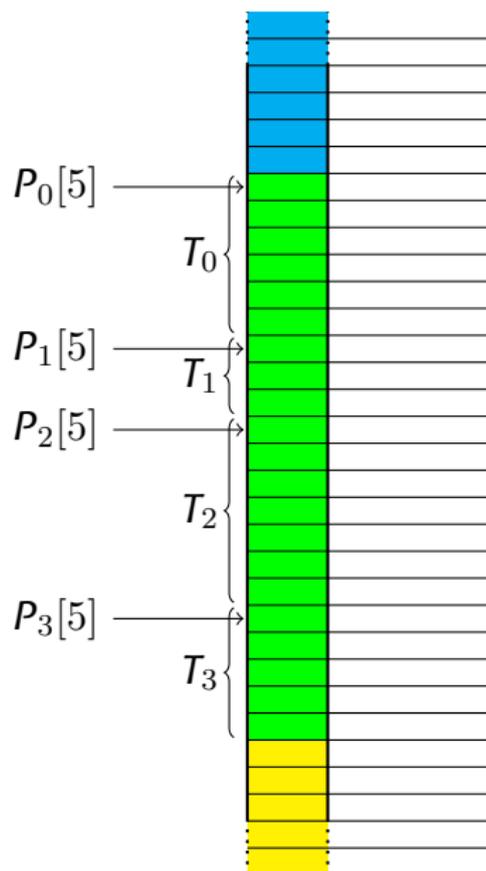
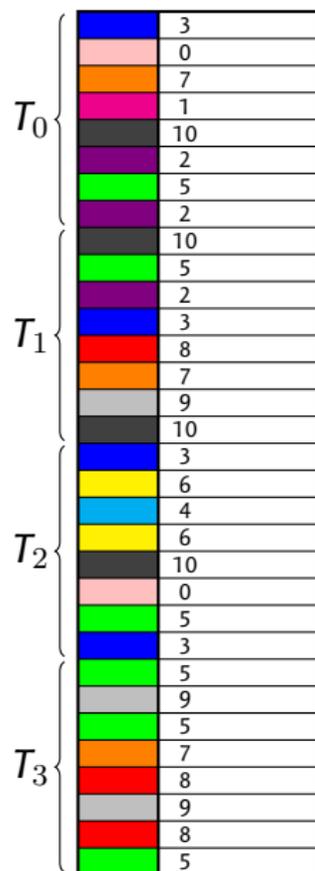


- ▶ ... Pour éliminer complètement les conflits

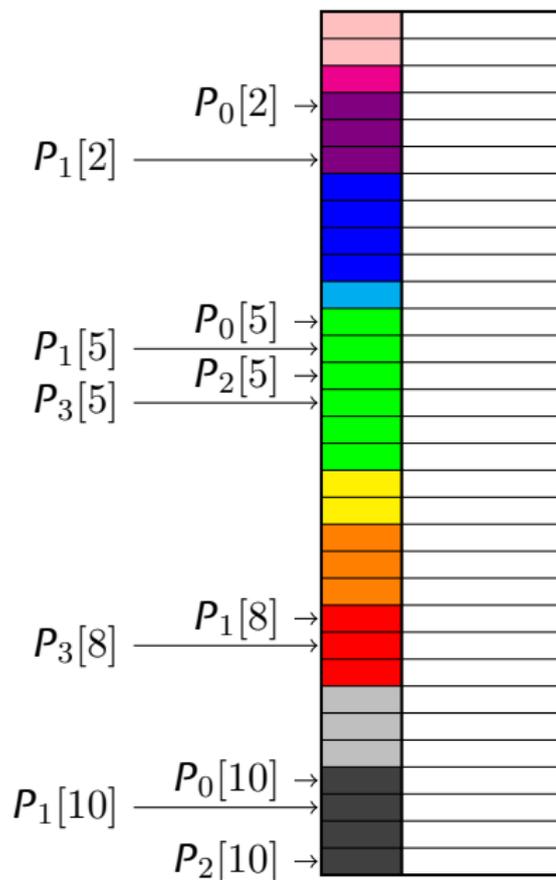
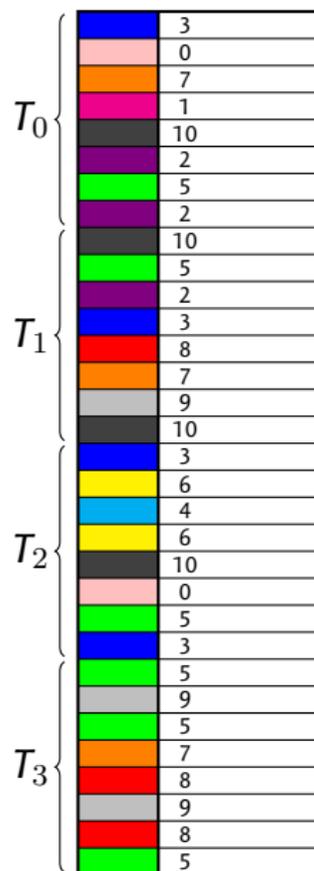
Exemple : Bucket Sort



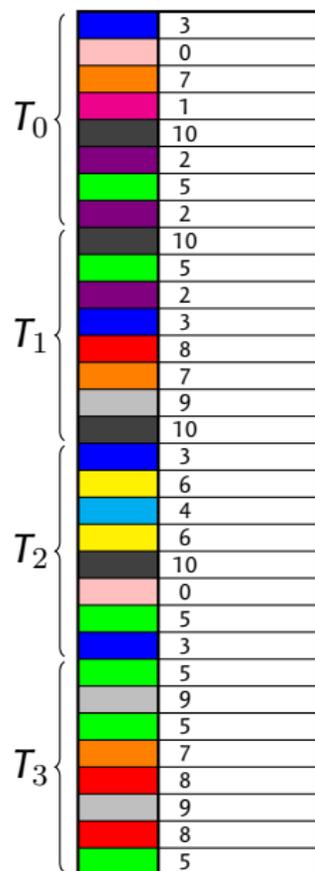
Exemple : Bucket Sort



Exemple : Bucket Sort



Exemple : Bucket Sort

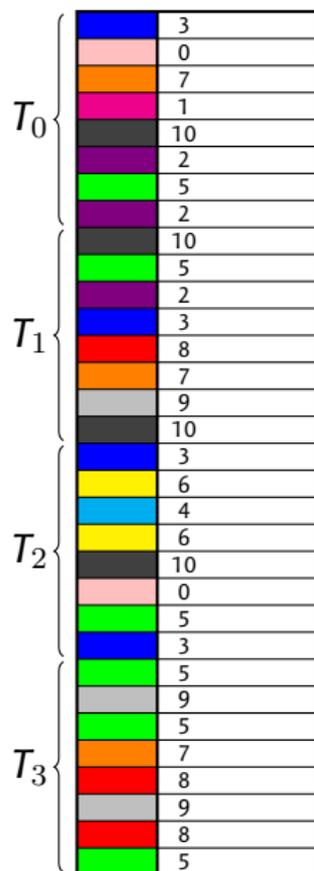


$C_i[x] = \# \text{Éléments de catégorie } x \text{ vus par le thread } i$

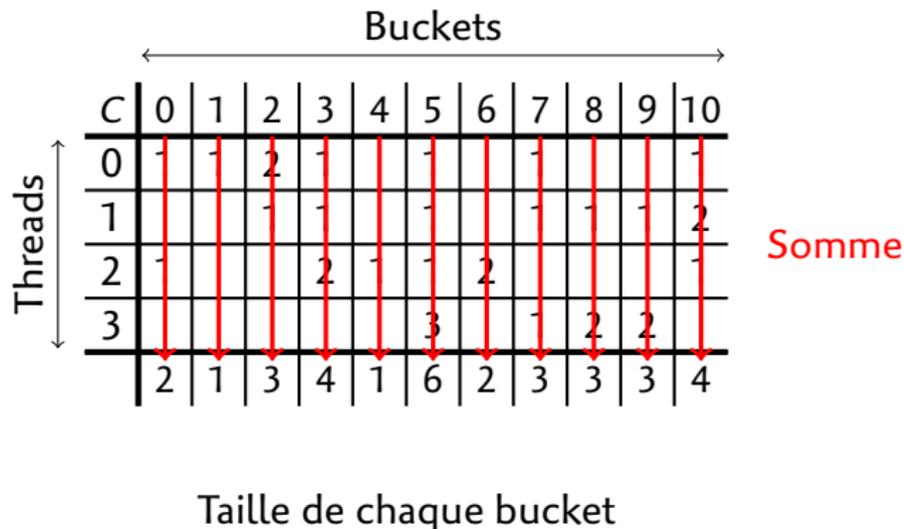
← Buckets →

	C	0	1	2	3	4	5	6	7	8	9	10
Threads	0	1	1	2	1		1		1			1
	1			1	1		1		1	1	1	2
	2	1			2	1	1	2				1
	3						3		1	2	2	

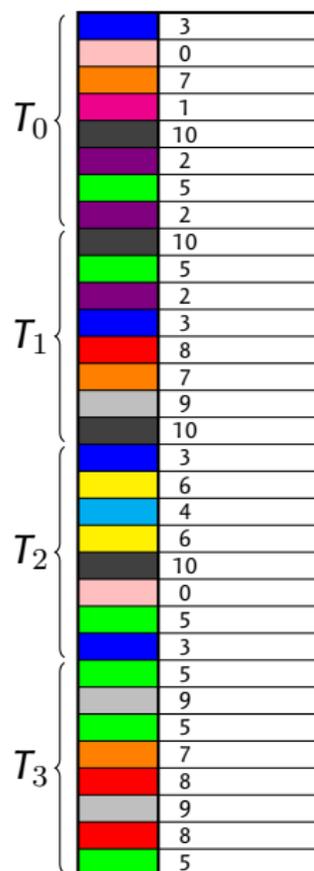
Exemple : Bucket Sort



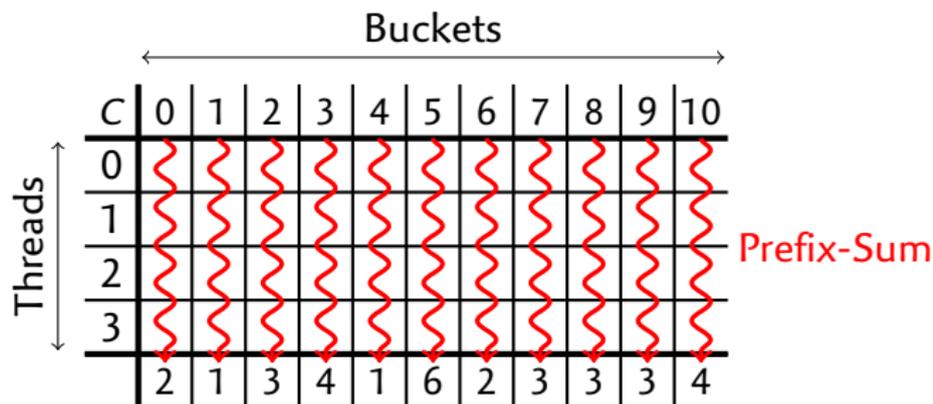
$C_i[x] =$ #Éléments de catégorie x
vus par le thread i



Exemple : Bucket Sort

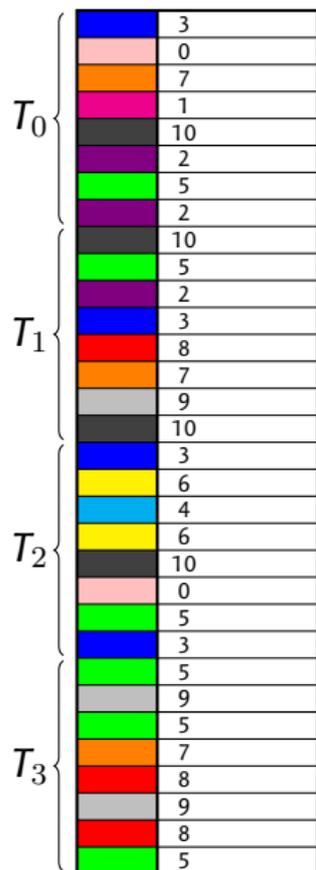


$C_i[x] =$ #Éléments de catégorie x
vus par le thread i



Taille de chaque bucket

Exemple : Bucket Sort



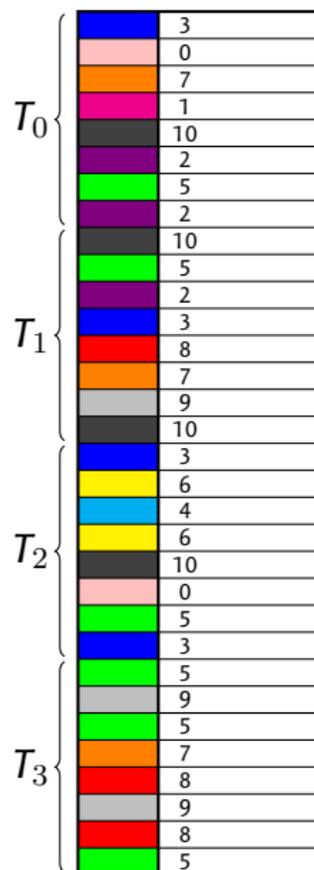
$C_i[x] =$ #Éléments de catégorie x
vus par les threads $< i$

← Buckets →

	C	0	1	2	3	4	5	6	7	8	9	10
Threads	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	2	1	0	1	0	1	0	0	1
	2	1	1	3	2	0	1	0	2	1	1	3
	3	2	1	3	4	1	3	2	2	1	1	4
		2	1	3	4	1	6	2	3	3	3	4

Taille de chaque bucket

Exemple : Bucket Sort



$C_i[x] =$ #Éléments de catégorie x
vus par les threads $< i$

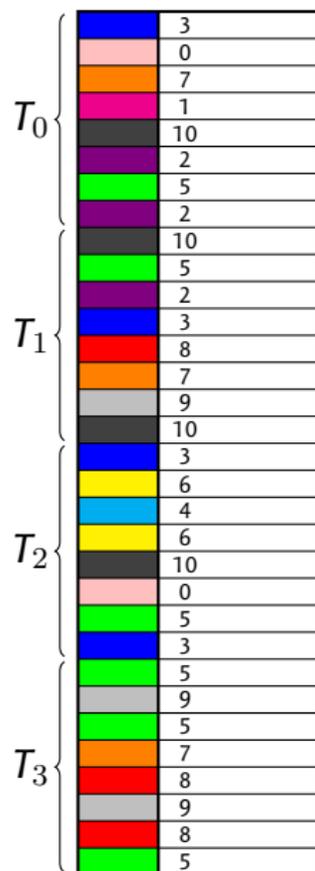
← Buckets →

C	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	2	1	0	1	0	1	0	0	1
2	1	1	3	2	0	1	0	2	1	1	3
3	2	1	3	4	1	3	2	2	1	1	4

Threads ↑

Prefix-Sum

Exemple : Bucket Sort



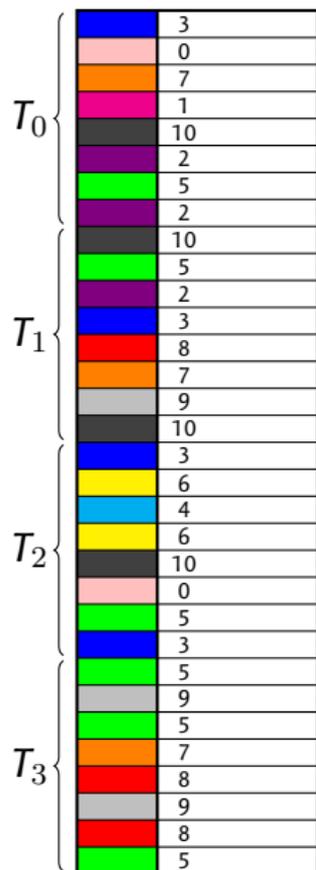
$C_i[x] =$ #Éléments de catégorie x
vus par les threads $< i$

← Buckets →

	C	0	1	2	3	4	5	6	7	8	9	10
Threads ↑	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	2	1	0	1	0	1	0	0	1
	2	1	1	3	2	0	1	0	2	1	1	3
	3	2	1	3	4	1	3	2	2	1	1	4
		0	2	3	6	10	11	17	19	22	25	28

Indice du début de chaque bucket
(#Éléments dans les buckets précédents)

Exemple : Bucket Sort



$C_i[x] =$ #Éléments de catégorie x
vus par les threads $< i$

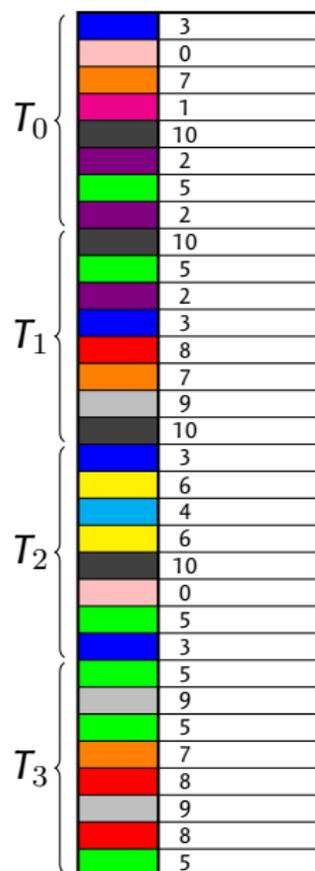
← Buckets →

	C	0	1	2	3	4	5	6	7	8	9	10
Threads ↑	0	0	0	0	0	0	0	0	0	0	0	0
	1	1	1	2	1	0	1	0	1	0	0	1
	2	1	1	3	2	0	1	0	2	1	1	3
	3	2	1	3	4	1	3	2	2	1	1	4
		0	2	3	6	10	11	17	19	22	25	28

Somme

Indice du début de chaque bucket
(#Éléments dans les buckets précédents)

Exemple : Bucket Sort



$C_i[x]$ = Indice du début du bucket x
pour le threads i

← Buckets →

	C	0	1	2	3	4	5	6	7	8	9	10
Threads ↑	0	0	2	3	6	10	11	17	19	22	25	28
	1	1	3	5	7	10	12	17	20	22	25	29
	2	1	3	6	8	10	12	17	21	23	26	31
	3	2	3	6	10	11	14	19	21	23	26	32
		0	2	3	6	10	11	17	19	22	25	28

Indice du début de chaque bucket
(#Éléments dans les buckets précédents)

Exemple : Bucket Sort

```
int C[T][M], S[M];

#pragma omp parallel
{
    int t = omp_get_thread_num();

    // Counting
    for (int i = 0; i < M; i++)
        C[t][i] = 0;
    #pragma omp for schedule(static)
    for (int i = 0; i < N; i++) {
        int bucket = f(A[i]);
        C[t][bucket]++;
    }

    // <<COMPUTE POINTERS>> ----->

    // Dispatch
    #pragma omp for schedule(static)
    for (int i = 0; i < N; i++) {
        int bucket = f(A[i]);
        int ptr = C[t][bucket]++;
        B[ptr] = A[i];
    }
}
```

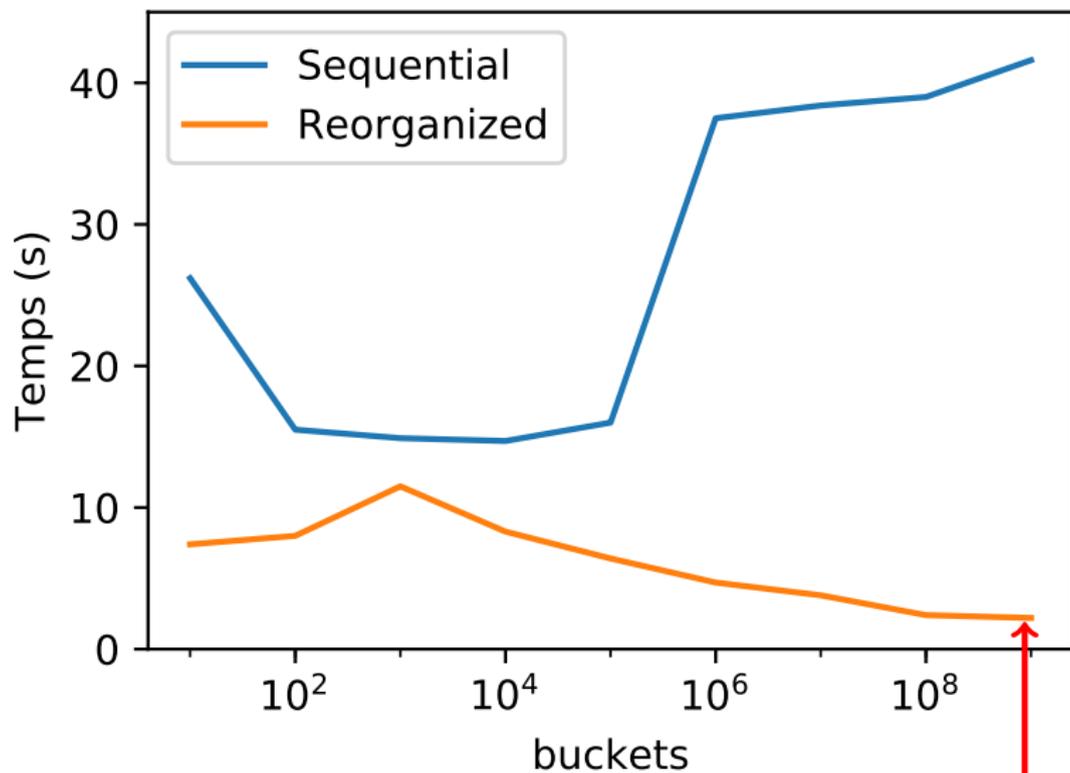
```
// sum (columns)
#pragma omp for
for (int i = 0; i < M; i++) {
    S[i] = 0;
    for (int j = 0; j < T; j++)
        S[i] += C[j][i];
}

// horizontal prefix-sum (sequential)
#pragma omp single
{
    int s = 0;
    for (int i = 0; i < M; i++) {
        int t = S[i];
        S[i] = s;
        s += t;
    }
}

// prefix-sum (columns)
#pragma omp for
for (int i = 0; i < M; i++) {
    int s = S[i];
    for (int j = 0; j < T; j++) {
        int t = C[j][i];
        C[j][i] = s;
        s += t;
    }
}
```

Exemple : Bucket Sort

$$N = 10^{10}$$



x19

Un tableau d'entiers à trier?

Pro Tip

Algorithme de tri parallèle efficace

- ▶ *Parallel Bucket Sort* sur les 8 bits de poids forts
- ▶ Pour $0 \leq i < 2^8$, faire (en parallèle) :
 - ▶ Trier le i -ème Bucket (avec un tri séquentiel normal)

Idée générale n°2 : procéder par « phases »



- ▶ Accepter une réduction du degré de parallélisme...

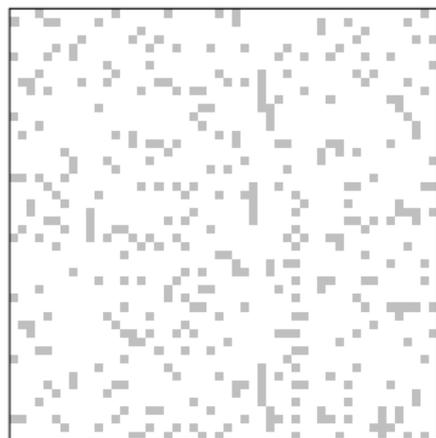


- ▶ ... Pour éliminer complètement les conflits

Exemple : factorisation LU creuse

$$\left[\begin{array}{c} \text{Sparse matrix} \\ \text{Fill-in pattern} \end{array} \right] = P \times \left[\begin{array}{c} \text{Sparse LU matrix} \\ \text{Fill-in pattern} \end{array} \right] \times Q^{-1}$$

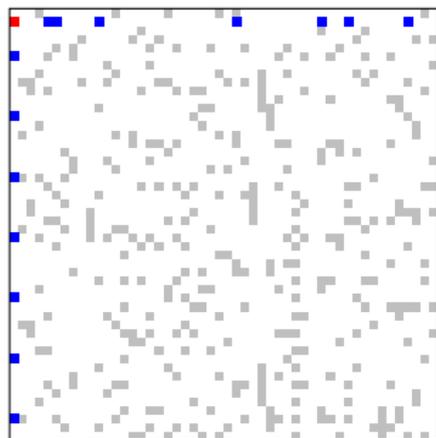
Exemple : factorisation LU creuse



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 $M_{i'} \leftarrow (\dots) \times M_i$.
4. [Boucle.] Incrémenter j . Si $j = n$, STOP. Sinon retourner en 2.

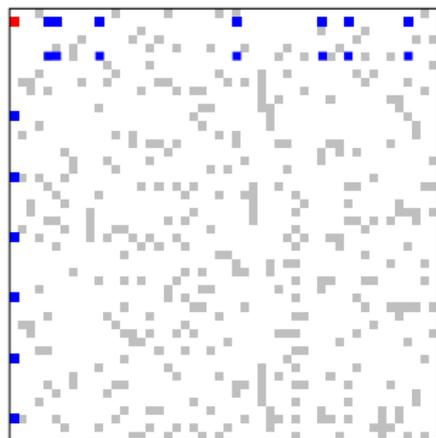
Exemple : factorisation LU creuse



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 $M_{i'} \leftarrow (\dots) \times M_i$.
4. [Boucle.] Incrémenter j . Si $j = n$, STOP. Sinon retourner en 2.

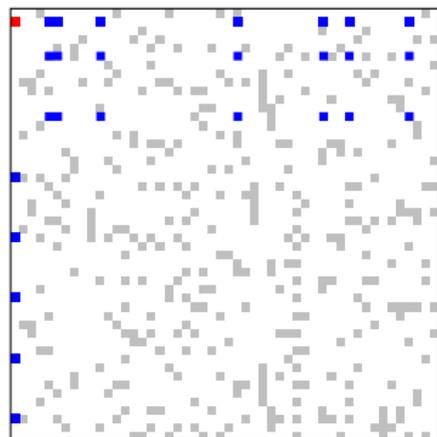
Exemple : factorisation LU creuse



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 $M_{i'} \leftarrow (\dots) \times M_i$.
4. [Boucle.] Incrémenter j . Si $j = n$, STOP. Sinon retourner en 2.

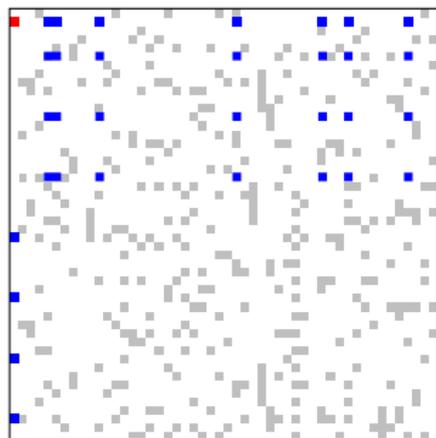
Exemple : factorisation LU creuse



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 $M_{i'} \leftarrow (\dots) \times M_i$.
4. [Boucle.] Incrémenter j . Si $j = n$, STOP. Sinon retourner en 2.

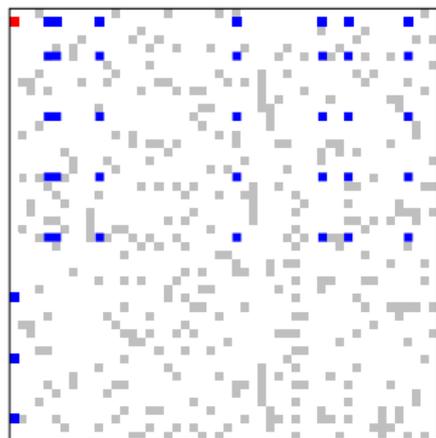
Exemple : factorisation LU creuse



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 $M_{i'} \leftarrow (\dots) \times M_i$.
4. [Boucle.] Incrémenter j . Si $j = n$, STOP. Sinon retourner en 2.

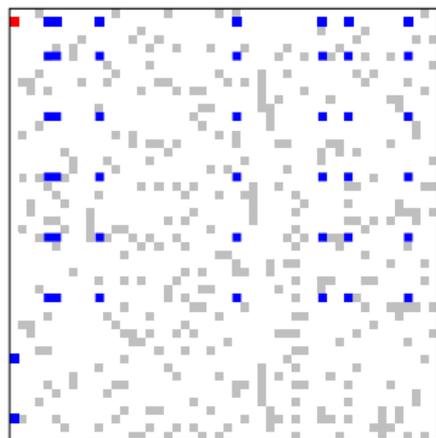
Exemple : factorisation LU creuse



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 $M_{i'} \leftarrow (\dots) \times M_i$.
4. [Boucle.] Incrémenter j . Si $j = n$, STOP. Sinon retourner en 2.

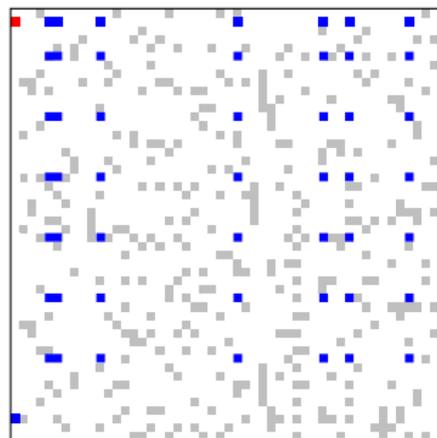
Exemple : factorisation LU creuse



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 $M_{i'} \leftarrow (\dots) \times M_i$.
4. [Boucle.] Incrémenter j . Si $j = n$, STOP. Sinon retourner en 2.

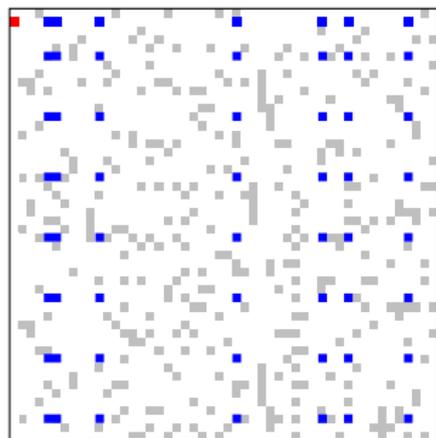
Exemple : factorisation LU creuse



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 $M_{i'} \leftarrow (\dots) \times M_i$.
4. [Boucle.] Incrémenter j . Si $j = n$, STOP. Sinon retourner en 2.

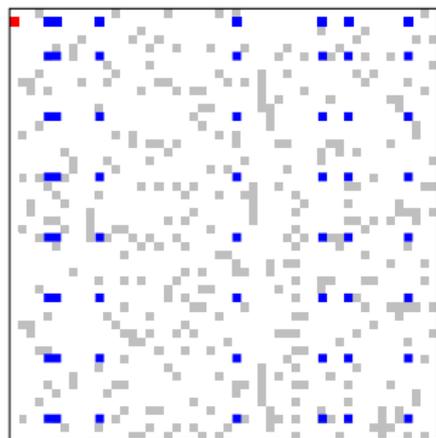
Exemple : factorisation LU creuse



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 $M_{i'} \leftarrow (\dots) \times M_i$.
4. [Boucle.] Incrémenter j . Si $j = n$, STOP. Sinon retourner en 2.

Exemple : factorisation LU creuse



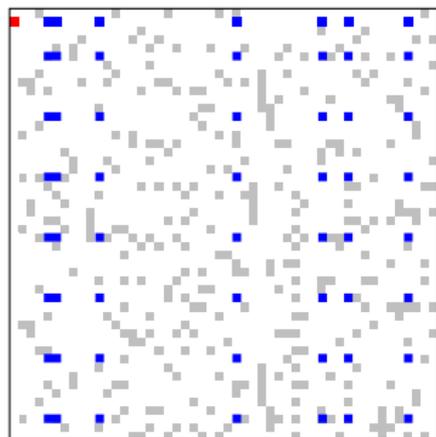
Plusieurs colonnes en parallèle?

► 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 $M_{i'} \leftarrow (\dots) \times M_i$.
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 $M_{i'} \leftarrow (\dots) \times M_i$.
4. [Boucle.] Incrémenter j . Si $j = n$, STOP. Sinon retourner en 2.

Exemple : factorisation LU creuse

Dépendences

- ▶ Colonnes j et j' liées par ligne i .

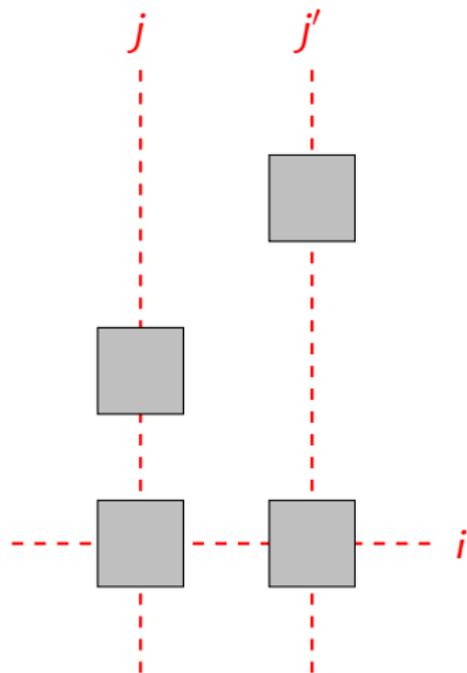
Graphe de dépendence G_{dep}

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

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

Colonnes indépendantes

→ 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 !



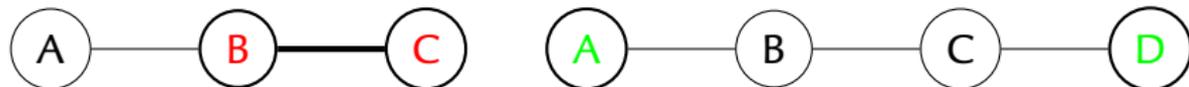
Portion séquentielle...

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} .



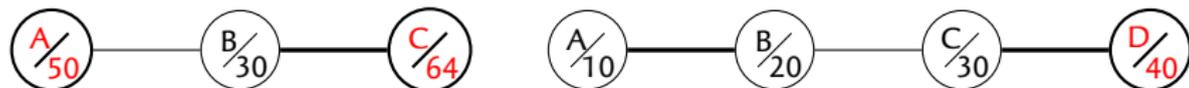
En parallèle : conflit avec ses voisins...

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 $\{1, \dots, n\}$.
- ▶ 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} .

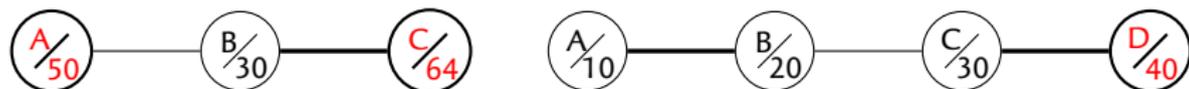


Exemple : factorisation LU creuse

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

Algorithme modifié

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



Transactions parallèles

lecture $A[i_1], A[i_2], \dots$ → **calcul** → **écriture** $A[k_1], A[k_2], \dots$

Obstacle à l'exécution « atomique » :

- ▶ Les données lues ont été modifiées avant l'écriture.
- ▶ Résultat du calcul « périmé ».

Approche pessimiste (« *Ask for Permission* »)

- ▶ « Verrouiller » les données lues.
- ▶ Lecture/Verrouillage → Calcul → écriture → déverrouillage
 - ▶ Bloque modification **potentielle** par un autre thread.
- ▶ Faire comme si le conflit **ALLAIT** avoir lieu.
- ▶ Surcoût inutile en l'absence de conflit.

Transactions parallèles

lecture $A[i_1], A[i_2], \dots$ → **calcul** → **écriture** $A[k_1], A[k_2], \dots$

Obstacle à l'exécution « atomique » :

- ▶ Les données lues ont été modifiées avant l'écriture.
- ▶ Résultat du calcul « périmé ».

Approche optimiste (« *Shoot First, Ask Questions Later* »)

- ▶ Lire (**sans précaution !!!**) → Calcul → **Commit** (atomique) :
 - ▶ Vérifier la fraîcheur des données lues,
 - ▶ Si OK, effectuer l'écriture ; sinon, tout recommencer.
- ▶ Faire comme si le conflit **N'ALLAIT PAS** avoir lieu.
- ▶ Travail perdu en cas de conflit.

Idée générale n°3 : analyser la fréquence des conflits

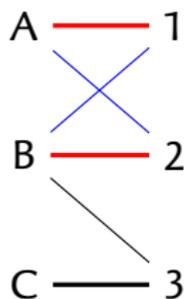


- ▶ Prendre le risque de gâcher un peu de calcul...

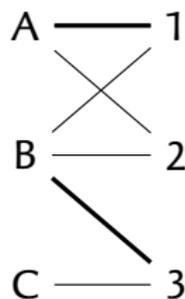


- ▶ ... Pour réduire le coût de la gestion des conflits

Exemple : plus grand couplage sans cycle alternant



Couplage avec **cycle alternant**
(polynomial)



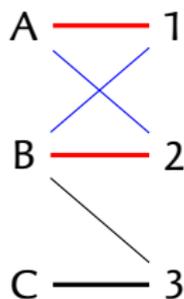
Couplage sans cycle alternant
(NP-dur)

Algorithme glouton

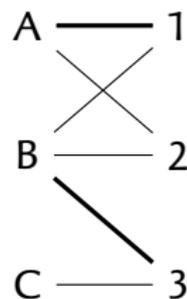
Pour tout sommet u (en parallèle) et toute arête ($u \leftrightarrow v$) :

- ▶ Parcours en largeur **alternant** depuis u ; atteint v ? \Rightarrow abort.
- ▶ Ajoute ($u \leftrightarrow v$) à \mathcal{C} . *[OK, pas de cycle]*

Exemple : plus grand couplage sans cycle alternant



Couplage avec **cycle alternant**
(polynomial)



Couplage sans cycle alternant
(NP-dur)

Algorithme glouton parallèle avec **Versionning**

Pour tout sommet u (en parallèle) et toute arête ($u \leftrightarrow v$) :

- ▶ $t \leftarrow |\mathcal{C}|$
- ▶ Parcours en largeur **alternant** depuis u ; atteint v ? \Rightarrow abort.
- ▶ **section critique** : si $t = |\mathcal{C}|$, ajoute ($u \leftrightarrow v$) à \mathcal{C} ; $ok \leftarrow 1$.
- ▶ Si $ok = 0$, recommencer. *[KO, couplage modifié]*

Retour sur les transactions

- ▶ Problèmes similaires dans les serveurs de bases de données.
- ▶ Nombreuses techniques de gestion des transactions.

CPU (très) modernes : transactional memory

```
#include <immintrin.h>
unsigned int status = _xbegin();
if (status == _XBEGIN_STARTED) {
    // Access shared data ...
    if (problem) // give up ?
        _xabort(0);
    // Access more shared data ...
    _xend();
    /* <----- Success !!! */
} else { /* <--- Failure */
    if (status & _XABORT_EXPLICIT)
        ...
    if (status & _XABORT_CONFLICT)
        ...
    if (status & _XABORT_CAPACITY)
        ...
}
```

- ▶ `_xbegin()` démarre une transaction
 - ▶ Renvoie `_XBEGIN_STARTED`
 - ▶ Purge le cache...
- ▶ `_xend()` tente le « commit ».
 - ▶ OK → l'exécution continue.
- ▶ `_xabort(cst)` force l'échec
- ▶ **En cas d'échec :**
 - ▶ Retourne après `_xbegin()`
 - ▶ Code erreur (conflit, ressources, ...)
- ▶ Toujours pas la panacée
 - ▶ Coût non-négligeable
 - ▶ Faux-positifs, ...
- ▶ Cf. aussi bibliothèque TinySTM

L'opération *Compare-And-Swap*

- ▶ OpenMP spécifie un *modèle mémoire* et des *opérations atomiques* séquentiellement consistantes.
- ▶ **C11** (ISO/IEC 9899 :2011) donne des équivalents.
- ▶ Avec un (gros) bonus : ***compare-and-swap atomique***.

```
#include <stdatomic.h>
```

```
bool atomic_compare_exchange_strong(volatile A* obj, C* expected, C desired);
```

```
bool atomic_compare_exchange_weak(volatile A *obj, C* expected, C desired);
```

Spécification — version *strong*

```
ok = (obj == expected); if (ok) obj = desired; return ok
```

Instruction du CPU (ou versions équivalents LL/SC)

Transactions avec *Compare-And-Swap*

On peut faire (presque) n'importe quoi avec *Compare-And-Swap* !

Idée générale : *Compare-And-Swap Loop*

1. [Begin.] $x_{ok} \leftarrow x$
2. [Work.] Calculer une mise à jour x_{new}
3. [Commit.] $ok = \text{atomic_compare_exchange_strong}(x, x_{ok}, x_{new})$
4. [Repeat.] Si pas ok, retourner en 1.

Exemples avec *Compare-And-Swap*

Exemple : liste chaînée

```
struct item_t {
    ...
    struct item_t *next;
}

void atomic_append(struct item_t *list, ...)
{
    struct item_t *new = malloc(sizeof(*new));
    ...
    bool ok = false;
    while (!ok) {
        new->next = list;
        ok = atomic_compare_exchange_strong(list, new->next, new);
    }
}
```

Exemples avec *Compare-And-Swap*

Exemple : table de hachage avec sondage linéaire

```
void insert(void *H, void *item)
{
    int i = hash_function(item);           // hash
    while (H[i] != EMPTY)                 // trouve une case vide
        i = (i + 1) % HASHTABLE_SIZE;
    H[i] = item;                          // insert
}
```

Version thread-safe

```
void ATOMIC_insert(void *H, void *item)
{
    int i = hash_function(item);
    bool ok = false
    while (!ok) {
        ok = atomic_compare_exchange_strong(H[i], EMPTY, item);
        i = (i + 1) % HASHTABLE_SIZE;
    }
}
```

Spécification C11

- ▶ Modèle mémoire très général/très précis.
- ▶ Opérations atomiques → *sequential consistency*
- ▶ Plus fin/moins coûteux → *Release-Acquire consistency*
- ▶ Bien compliqué, pas tout compris \rightsquigarrow RDV l'an prochain...