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

Charles Bouillaguet
charles.bouillaguet@lip6.fr

2020-02-28

Quizz

Initialement, $x = y = 0$.

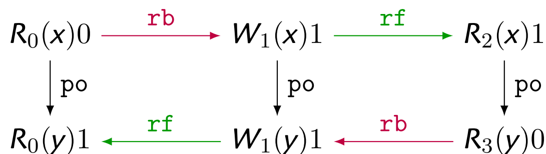
$$R_0(x)0 \xrightarrow{\texttt{po}} R_0(y)1 \qquad W_1(x)1 \xrightarrow{\texttt{po}} W_1(y)1 \qquad R_2(x)1 \xrightarrow{\texttt{po}} R_3(y)0$$

Possible ?

1. Non, c'est contradictoire
2. Non, car ce n'est pas séquentiellement consistant
3. Oui, sur les ARM et les POWER mais pas sur les x86 (TSO)
4. Oui, car c'est séquentiellement consistant

Quizz

Initialement, $x = y = 0$.

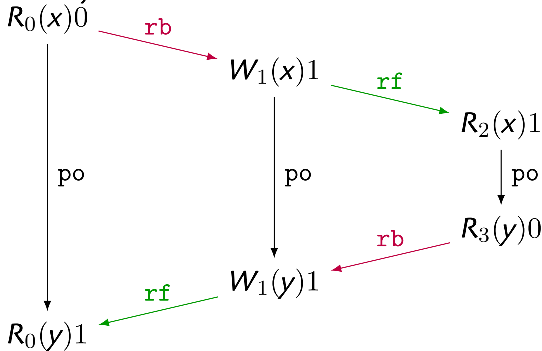


Possible ?

1. Non, c'est contradictoire
2. Non, car ce n'est pas séquentiellement consistant
3. Oui, sur les ARM et les POWER mais pas sur les x86 (TSO)
4. Oui, car c'est séquentiellement consistant

Quizz

Initialement, $x = y = 0$.



Possible ?

1. Non, c'est contradictoire
2. Non, car ce n'est pas séquentiellement consistant
3. Oui, sur les ARM et les POWER mais pas sur les x86 (TSO)
4. Oui, car c'est séquentiellement consistant

Quizz (plus dur)

Initialement, $x = y = 0$.

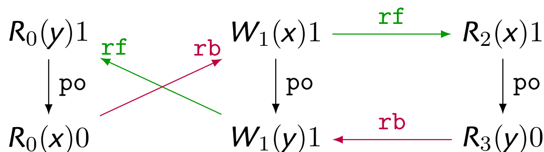
$$
\begin{array}{ccc}
R_0(y)1 & W_1(x)1 & R_2(x)1 \\
\downarrow \texttt{po} & \downarrow \texttt{po} & \downarrow \texttt{po} \\
R_0(x)0 & W_1(y)1 & R_3(y)0
\end{array}
$$

Possible ?

1. Non, c'est contradictoire
2. Non, car ce n'est pas séquentiellement consistant
3. Oui, sur les ARM et les POWER mais pas sur les x86 (TSO)
4. Oui, car c'est séquentiellement consistant

Quizz (plus dur)

Initialement, $x = y = 0$.

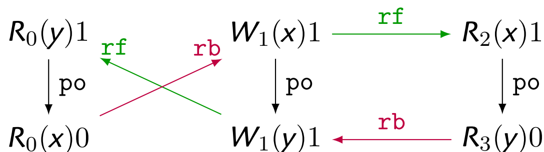


Possible ?

1. Non, c'est contradictoire
2. Non, car ce n'est pas séquentiellement consistant
3. Oui, sur les ARM et les POWER mais pas sur les x86 (TSO)
4. Oui, car c'est séquentiellement consistant

Quizz (plus dur)

Initialement, $x = y = 0$.



Possible ?

1. Non, c'est contradictoire
2. Non, car ce n'est pas séquentiellement consistant
3. Oui, sur les ARM et les POWER mais pas sur les x86 (TSO)
4. Oui, car c'est séquentiellement consistant

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 : **réorganiser**

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

- ... Pour éliminer complètement les conflits

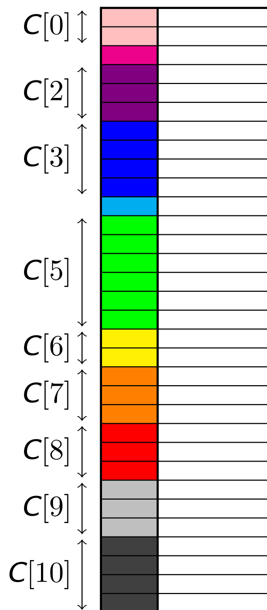
Exemple : Bucket Sort

```
// 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]++;
}
```

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

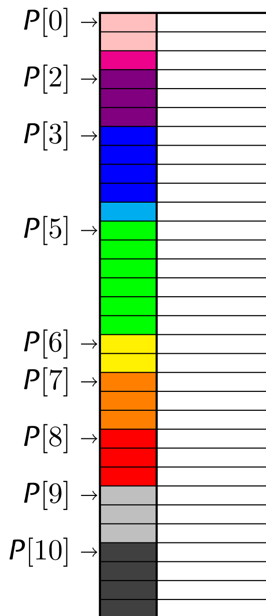
Exemple : Bucket Sort

```
// 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]++;
}
```



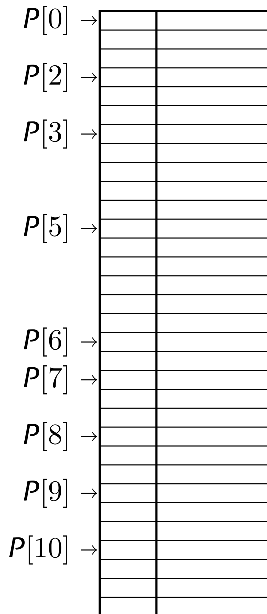
Exemple : Bucket Sort

```
// 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]++;
}
```



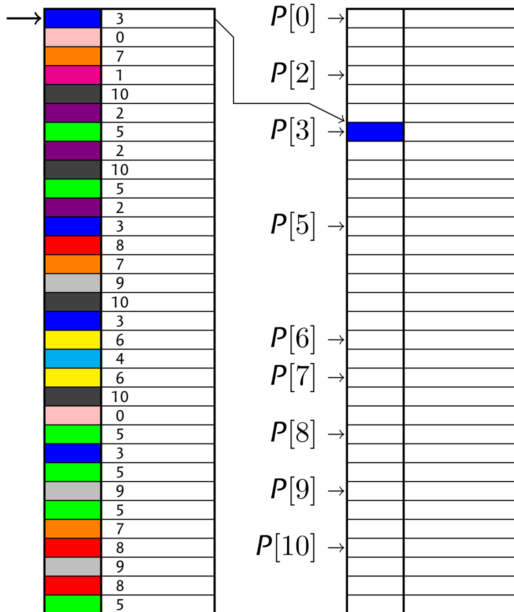
Exemple : Bucket Sort

```
// 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]++;
}
```



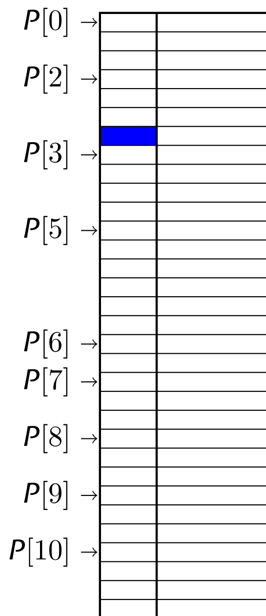
Exemple : Bucket Sort

```
// 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]++;
}
```



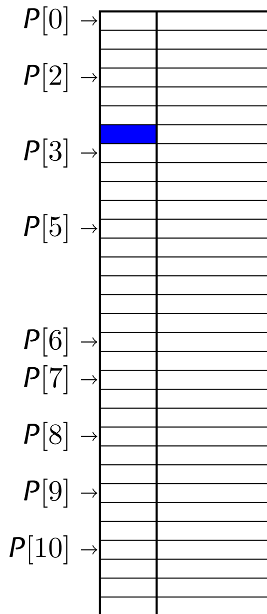
Exemple : Bucket Sort

```
// 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]++;
}
```



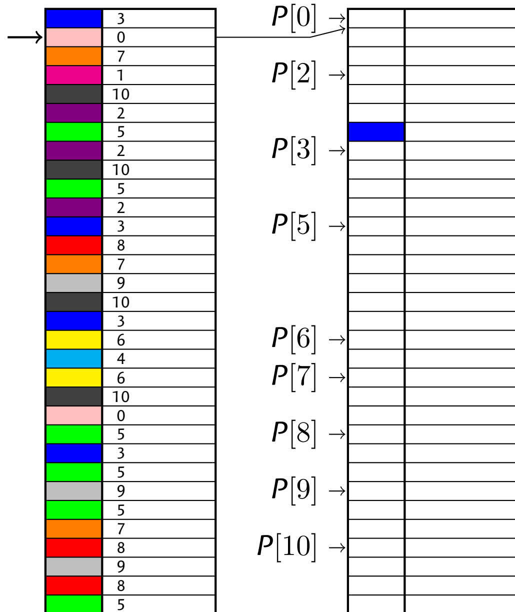
Exemple : Bucket Sort

```
// 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]++;
}
```



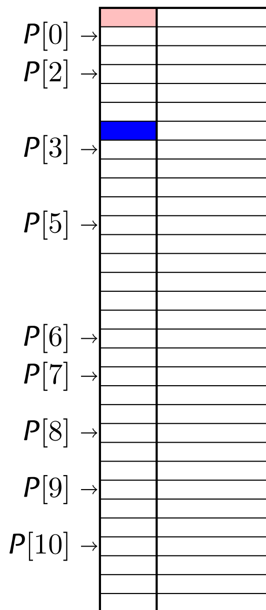
Exemple : Bucket Sort

```
// 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]++;
}
```



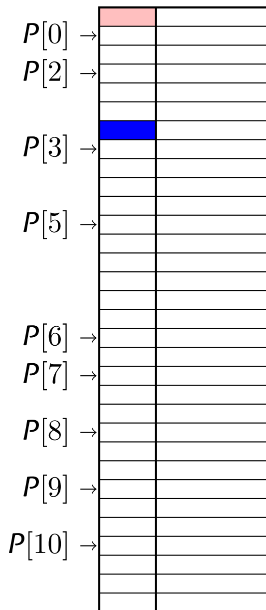
Exemple : Bucket Sort

```
// 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]++;
}
```



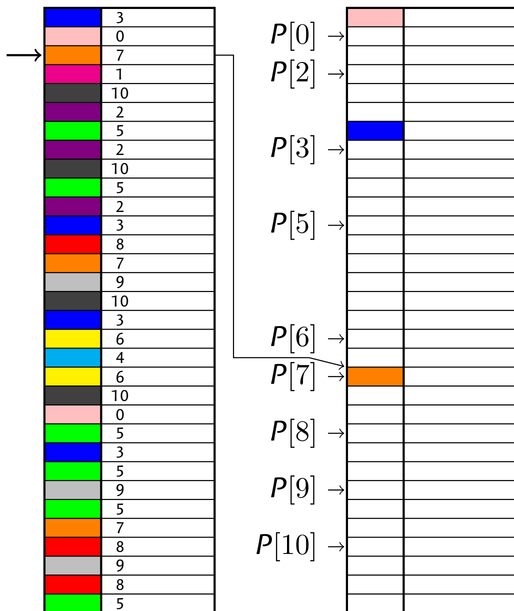
Exemple : Bucket Sort

```
// 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]++;
}
```



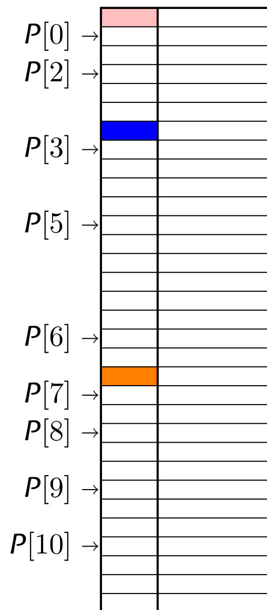
Exemple : Bucket Sort

```
// 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

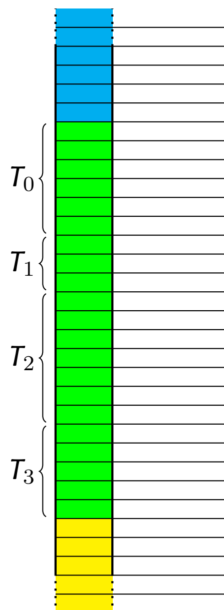
```
// Counting

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

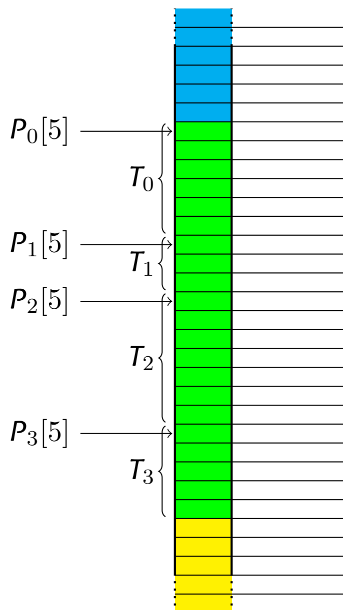
Parallélisation directe naïve

```
// 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];
}
```

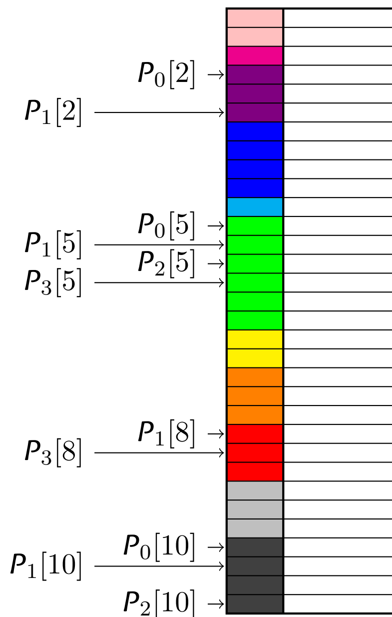
Exemple : Bucket Sort



Exemple : Bucket Sort



Exemple : Bucket Sort



Exemple : Bucket Sort

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

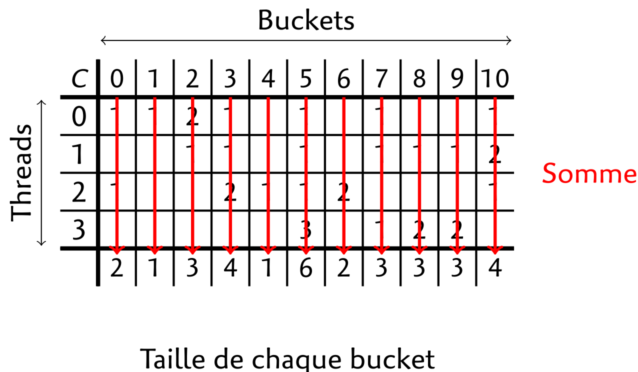
Buckets

Threads

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

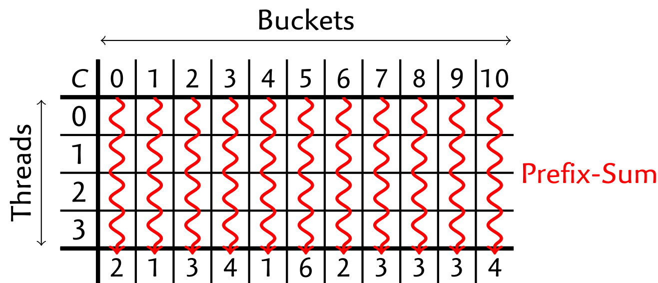
Exemple : Bucket Sort

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



Exemple : Bucket Sort

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



Taille de chaque bucket

Exemple : Bucket Sort

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

Buckets

Threads

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
	2	1	3	4	1	6	2	3	3	3	4

Taille de chaque bucket

Exemple : Bucket Sort

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

Buckets (→) / Threads (↓)

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
	2	1	3	4	1	6	2	3	3	3	4

Prefix-Sum

Exemple : Bucket Sort

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

Buckets (columns) / Threads (rows)

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
	0	2	3	6	10	11	17	19	22	25	28

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

Exemple : Bucket Sort

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

Buckets

Threads

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
	0	2	3	6	10	11	17	19	22	25	28

Somme

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

Exemple : Bucket Sort

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

Buckets (colonnes) / Threads (lignes)

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

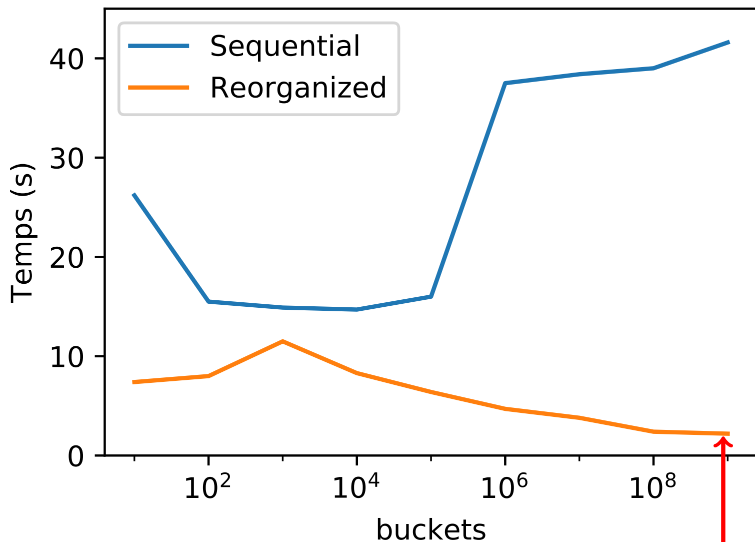
Indice du début de chaque bucket
(#Élements dans les buckets précédents)

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



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)

Transactions parallèles

lecture $A[i_1], A[i_2], \ldots \rightarrow$ **calcul** $\rightarrow$ **écriture** $A[k_1], A[k_2], \ldots$.

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 $\rightarrow$ Calcul $\rightarrow$ écriture $\rightarrow$ 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 $A[i_1], A[i_2], \ldots \rightarrow$ **calcul** $\rightarrow$ **écriture** $A[k_1], A[k_2], \ldots$.

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 : **analyser la fréquence des conflits**

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

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

Technique générale : le *versioning*

- Structure de donnée partagée, avec un numéro de version
- $v = 0$ au début (impair pendant les écritures).

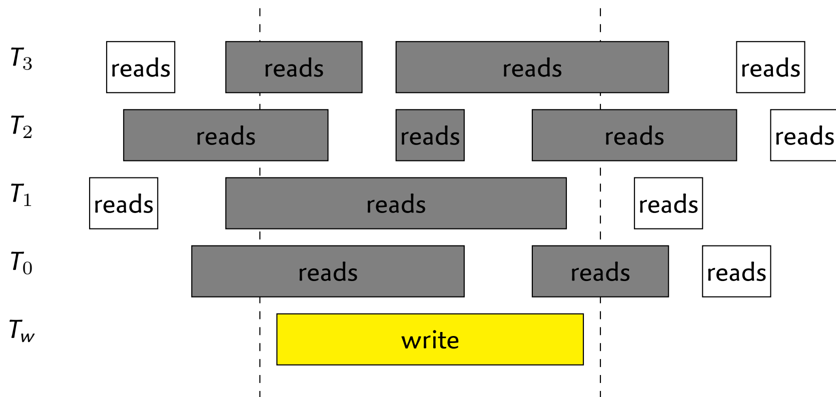
Écrivain

1. Entrer section critique ; incrémenter v
2. Effectuer les écritures
3. incrémenter v ; sortir section critique.

Lecteur

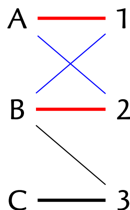
1. $v_{before} \leftarrow v$
2. Effectuer lectures
3. $v_{after} \leftarrow v$
4. Si v_{before} impair ou $v_{before} \neq v_{after}$, recommencer.

Technique générale : le *versioning*

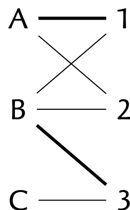


- Écrivains prioritaires sur les lecteurs

Exemple : plus grand couplage sans cycle alternant



Couplage avec **cycle alternant**
(polynomial)



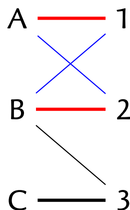
Couplage sans cycle alternant
(NP-dur)

Algorithme glouton

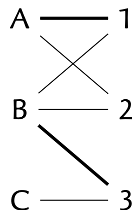
Pour tout sommet u (en parallèle) et toute arête $(u \leftrightarrow v)$:

- 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 **Versioning**

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]*

Concept de « transaction »

- 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, resources, ...)
- 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
```

1. Instruction du CPU (ou versions équivalents LL/SC)
2. La version *weak* peut avoir des faux négatifs

Et dans OpenMP ?

- Compare-and-swap est dans OpenMP 5.1
- Paru en novembre 2020

```
#pragma omp atomic compare
if (obj == expected)
        obj = desired;

#pragma omp atomic compare capture
if (obj == expected)
        obj = desired;
else
        v = obj;
```

- gcc 10.2 ne l'implante pas encore...

Transactions avec *Compare-And-Swap*

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

Idée générale : *Compare-And-Swap Loop*

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

Exemples avec *Compare-And-Swap*

Exemple : liste chainée

```
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);
    book ok = false
    while (!ok) {
        ok = atomic_compare_exchange_strong(H[i], EMPTY, item);
        i = (i + 1) % HASHTABLE_SIZE;
    }
}
```

Idée générale : algorithmes *lock-free*

Définition

Une fonction est **lock-free** si, lorsqu'elle est appelée par plusieurs threads à la fois, au moins l'une des invocations termine en un nombre fini d'étapes de calcul (quoi que fassent les autres, même s'ils bloquent).

Remarque : si une fonction acquiert un verrou / section critique, elle ne peut pas être *lock-free*.

Idée générale : algorithmes *lock-free*

Définition

Une fonction est **lock-free** si, lorsqu'elle est appelée par plusieurs threads à la fois, au moins l'une des invocations termine en un nombre fini d'étapes de calcul (quoi que fassent les autres, même s'ils bloquent).

Remarque : si une fonction acquiert un verrou / section critique, elle ne peut pas être *lock-free*.

Guru switch ⟶

Safety

World of PAIN

Exemple : ensemble d'entiers

```c
bool * A;
int N, min;

// A[i] indique si i est dans l'ensemble
// min pointe sur le plus petit élément

void setup()
{
   A = malloc((N + 1) * sizeof(*A));
   // astuce : sentinelle en position N
   for (int i = 0; i < N + 1; i++)
      A[i] = true;
   min = 0;
}
```

```c
bool remove(int i)
{
    bool x = A[i];
    A[i] = false;
    return x;
}

int extract_min()
{
   while(A[min] == false)
      min++;
   if (min < N)
      remove(min);
   return min;
}
```

Invariants

`A[N] == true` et `A[0:min] == false`.

Exemple : ensemble d’entiers lock-free

```
#include <stdbool.h>
#include <stdatomic.h>
#define CAS atomic_compare_exchange_weak

_Atomic bool * A;
_Atomic int min;
int N;
bool yes = true;

void setup()
{
    A = malloc((N + 1) * sizeof(*A));
    for (int i = 0; i < N + 1; i++)
        atomic_store(&A[i], true);
    atomic_store(&min, 0);
}

bool remove(int i)
{
    return atomic_exchange(&A[i], false);    // comme précédemment
}
```

Invariants

A[N] == true et A[0:min] == false.

```
int extract_min()
{
    int old_min = atomic_load(&min);
    int i = old_min;                    // cherche A[i] == true
    while(atomic_load(&A[i]) == false)
        i++;
```

Invariants

`A[N] == true` et `A[0:min] == false`.

```
int extract_min()
{
    int old_min = atomic_load(&min);
    int i = old_min;                    // cherche A[i] == true
    while(atomic_load(&A[i]) == false)
        i++;
    if (i == N) {                       // ensemble vide ?
        atomic_store(&min, N);
        return N;
    }
```

Invariants

`A[N] == true` et `A[0:min] == false`.

```
int extract_min()
{
    int old_min = atomic_load(&min);
    int i = old_min;                // cherche A[i] == true
    while(atomic_load(&A[i]) == false)
        i++;
    if (i == N) {                   // ensemble vide ?
        atomic_store(&min, N);
        return N;
     }
    atomic_store(&A[i], false);
```

Raté

- On a bien vu `A[i] == true` ...
- ... mais ça a pu changer entre-temps ...
- Si un autre `extract_min()` a eu lieu en même temps

Invariants

`A[N] == true` et `A[0:min] == false`.

```
int extract_min()
{
    int old_min = atomic_load(&min);
    int i = old_min;                    // cherche A[i] == true
    while(atomic_load(&A[i]) == false)
        i++;
    if (i == N) {                       // ensemble vide ?
        atomic_store(&min, N);
        return N;
     }
    bool ok = CAS(&A[i], &yes, false);
    if (!ok)
        return extract_min();           // A[i] == false ? On recommence
```

Invariants

`A[N] == true` et `A[0:min] == false`.

```
int extract_min()
{
    int old_min = atomic_load(&min);
    int i = old_min;                    // cherche A[i] == true
    while(atomic_load(&A[i]) == false)
        i++;
    if (i == N) {                       // ensemble vide ?
        atomic_store(&min, N);
        return N;
     }
    bool ok = CAS(&A[i], &yes, false);
    if (!ok)
        return extract_min();           // A[i] == false ? On recommence
    CAS(&min, &old_min, i);             // avance min (paresseusement)
    return i;
}
```

Invariants

A[N] == true et A[0:min] == false.

```
int extract_min()
{
    int old_min = atomic_load(&min);
    int i = old_min;                    // cherche A[i] == true
    while(atomic_load(&A[i]) == false)
        i++;
    if (i == N) {                       // ensemble vide ?
        atomic_store(&min, N);
        return N;
     }
    bool ok = CAS(&A[i], &yes, false);
    if (!ok)
        return extract_min();           // A[i] == false ? On recommence
    while(old_min < i) {                // avance min (consciencieusement)
        bool ok = CAS(&min, &old_min, i);
        if (ok)
            break;
        old_min = atomic_load(&min);
    }
    return i;
}
```

Est-ce correct ?

Est-ce correct ?

Question subsidiaire

Que signifie « être correct » ?

Est-ce correct ?

Question subsidiaire

Que signifie « être correct » ?

Définition

Une structure de donnée est **linéarisable** si tout se passe comme si les fonctions qui y accèdent prenaient effet *instantanément*, à un point quelconque entre leur invocation et leur terminaison.

Est-ce correct ?

Question subsidiaire

Que signifie « être correct » ?

Définition

Une structure de donnée est **linéarisable** si tout se passe comme si les fonctions qui y accèdent prenaient effet *instantanément*, à un point quelconque entre leur invocation et leur terminaison.

Correct $\approx$ linéarisable

Est-ce correct ?

Question subsidiaire

Que signifie « être correct » ?

Définition

Une structure de donnée est **linéarisable** si tout se passe comme si les fonctions qui y accèdent prenaient effet *instantanément*, à un point quelconque entre leur invocation et leur terminaison.

Correct $\approx$ linéarisable

Exercice : faites la preuve.