

# ► Programmation parallèle

Cycle 2020-2021

ECE – Ing5 – Systèmes Embarqués

Etienne Hamelin

[ehamelin@inseec-edu.com](mailto:ehamelin@inseec-edu.com)

Alexandre Berne

[aberne@inseec-edu.com](mailto:aberne@inseec-edu.com)

## ► Planning

- Introduction & concepts
  - TP: Pthreads
- Cohérence mémoire
  - TP: OpenMP
- **Consistance mémoire, équilibrage et placement des données**
  - **TP: Synchronisations**
- Présentation du projet
  - Projet

## ► Le récap' de la séance 2

- Associativité:
  - *direct mapped, n-set associative, fully associative*
- Politiques de remplacement:
  - LRU, FIFO, random, ...
- Politiques d'écriture:
  - *write hit (write-through / write-back), write miss (write-allocate / write-non-allocate)*
- Les sources de conflits (4C):
  - *Compulsory miss, conflict miss, capacity miss, coherence miss*
- TP : OpenMP

## ► Temps d'accès moyen

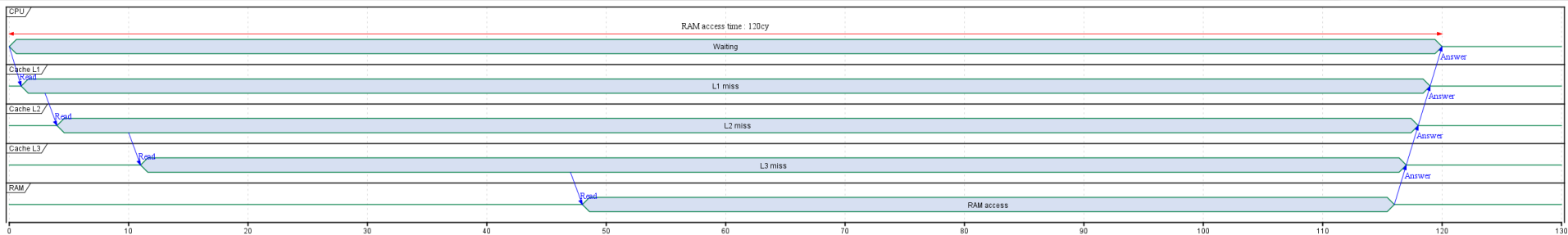
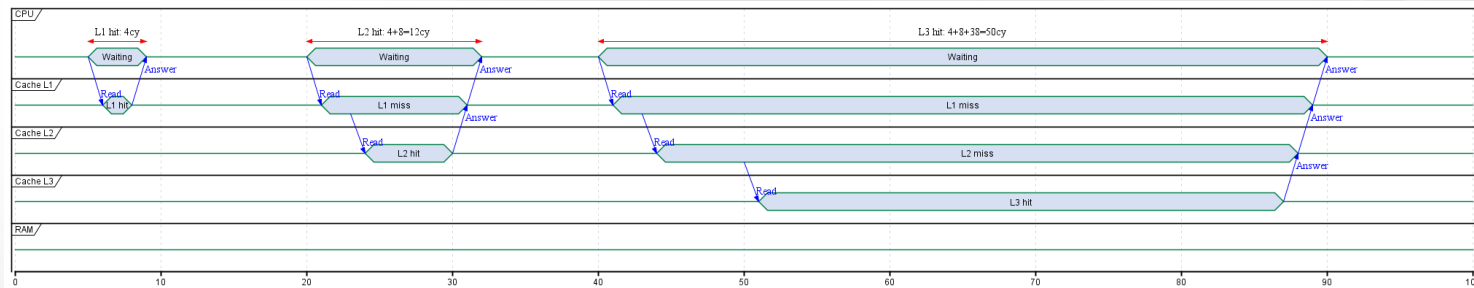
- Taux de probabilités:
  - De succès: *hit rate*  $hr$ ,
  - D'échec: *miss rate*  $mr = 1 - hr$
- Temps d'accès:
  - En cas de succès: *hit time*  $L_{hit}$
  - En cas d'échec: *hit time* + *miss penalty* ;  $L_{miss} = L_{hit} + MP$
- Temps d'accès moyen (*average memory access time*)
  - $AMAT = hr \cdot L_{hit} + mr \cdot L_{miss} = L_{hit} + mr \cdot MP$

## ► Multi-niveau

- Cache  $n$  miss = accès au cache  $n + 1 \Rightarrow MP_{Ln} = AMAT_{L(n+1)}$ 
$$AMAT = L_{hit(L1)} + mr_{L1} \cdot MP_{L1} = L_1 + mr_1 \cdot (L_2 + mr_2 \cdot MP_2)$$

## ► Exemple

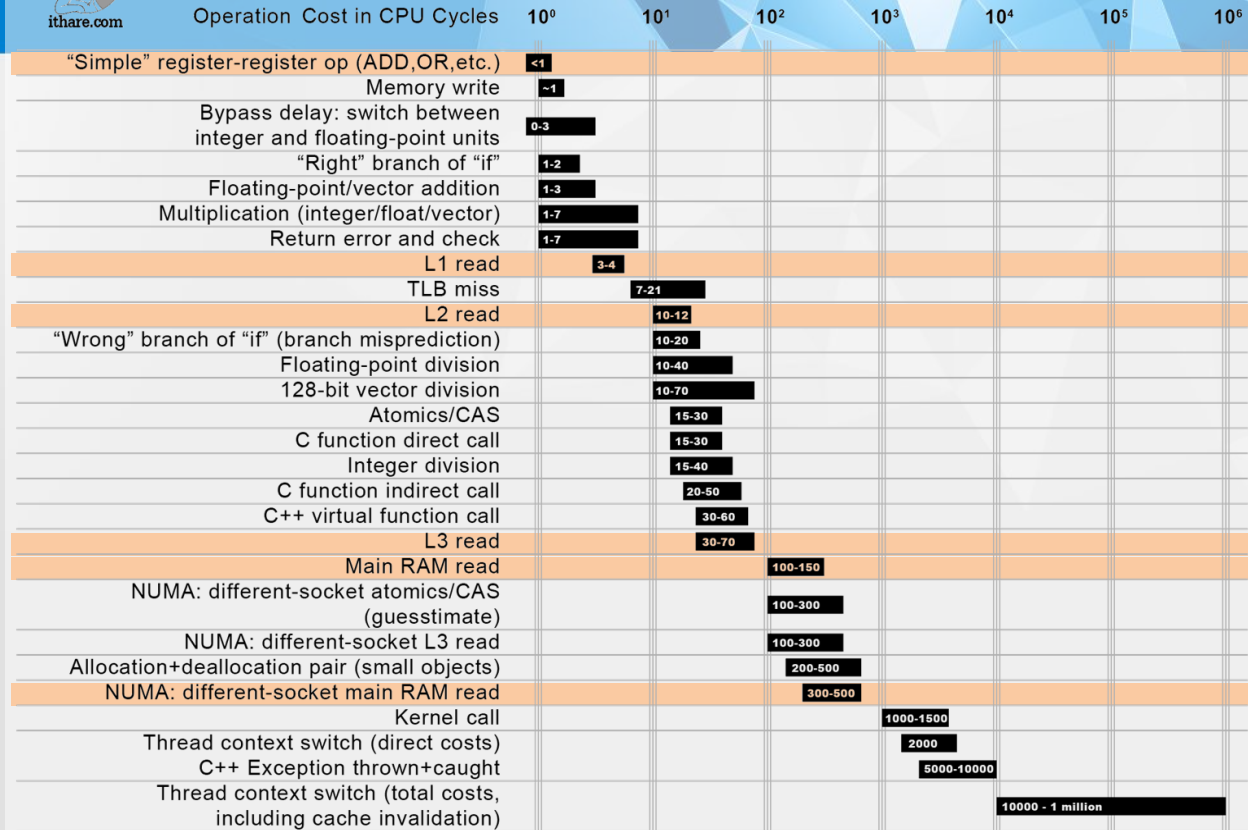
- 50% des données dans le cache L1 ( $L_1 = 4cy$ ), 25% dans L2 ( $L_2 = 8cy$ ), 12,5% L3 ( $L_3 = 38cy$ ), 12,5% RAM ( $L_{RAM} = 70cy$ ).
- $AMAT = 26$  cycles => La moitié des accès ne prennent que 4 cycles, et pourtant la moyenne est quasiment 7 fois plus !



► Note: le temps d'attente n'est pas toujours perdu (exécution out-of-order, spéculative)



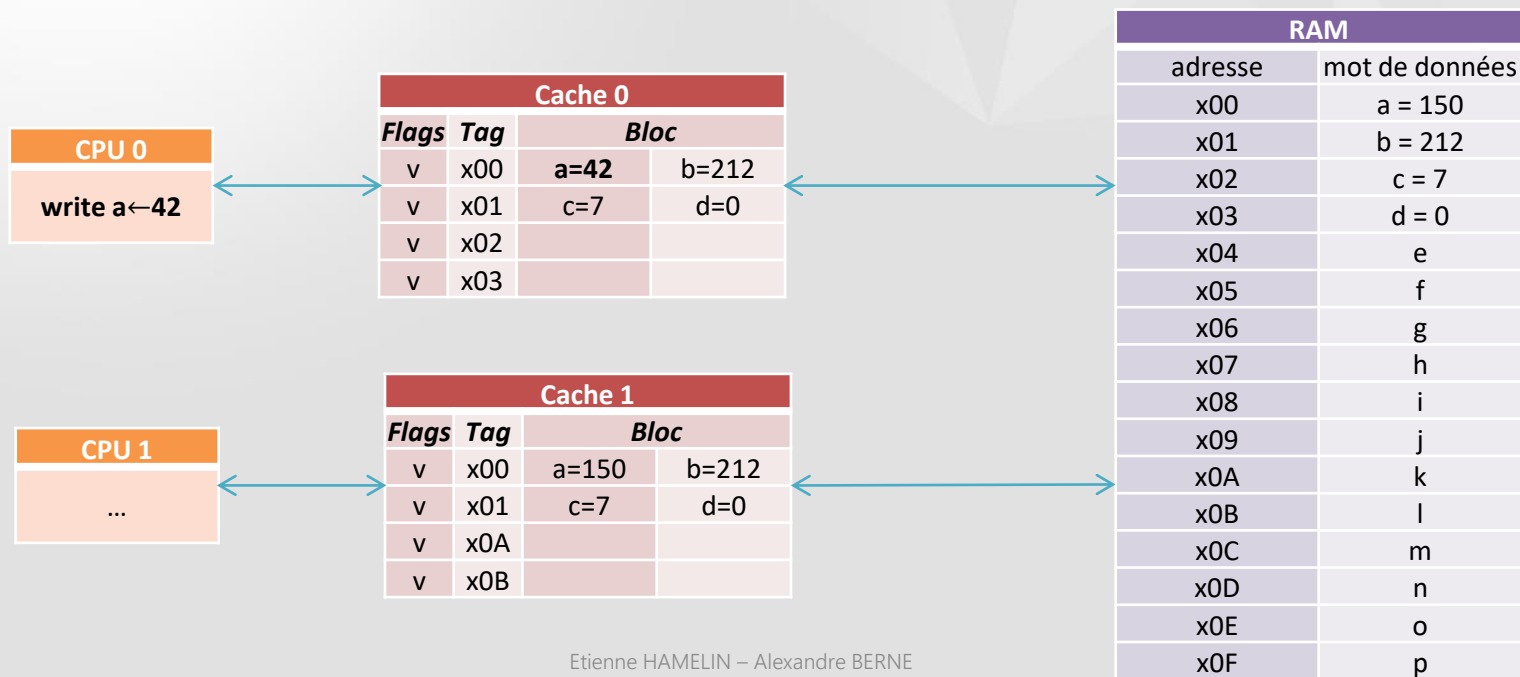
## Not all CPU operations are created equal



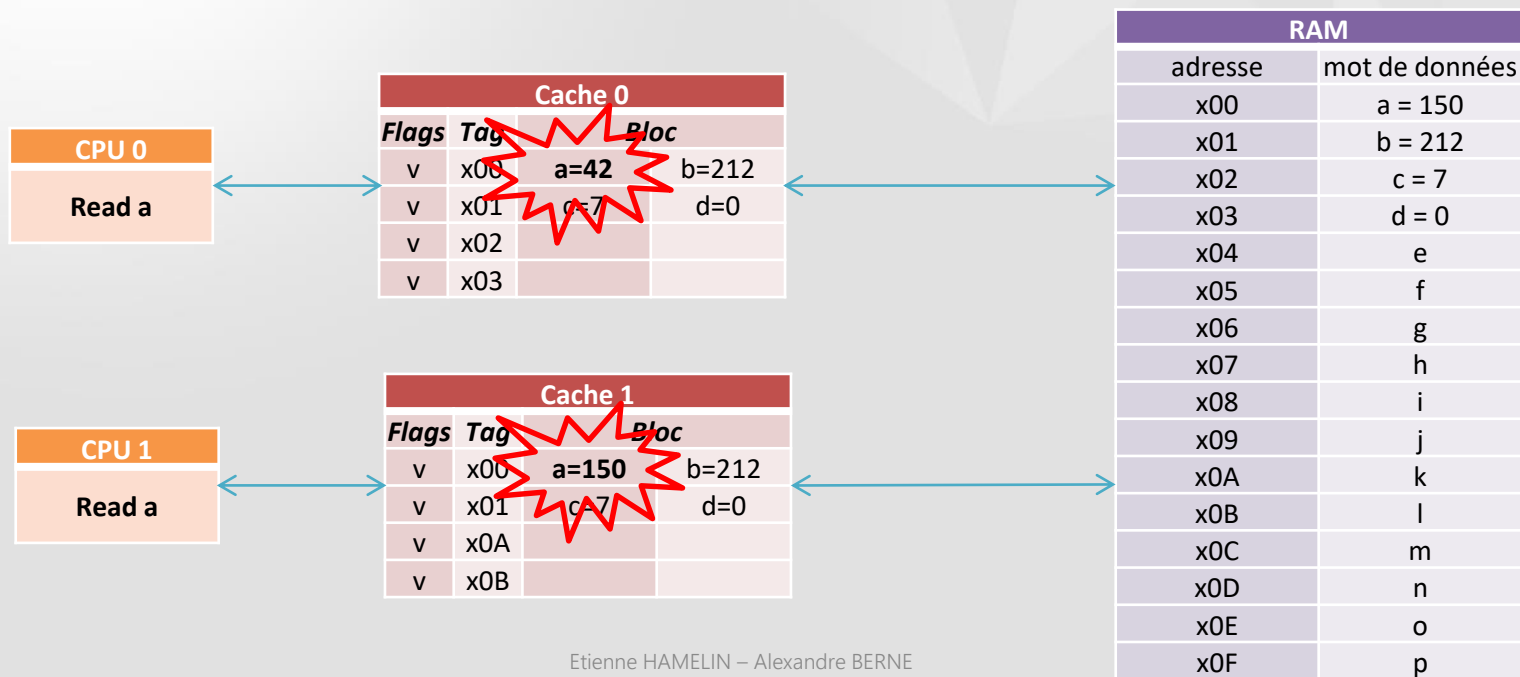
Distance which light travels  
while the operation is performed



## Le cas multicoeur maintenant!



## ► Problème de cohérence

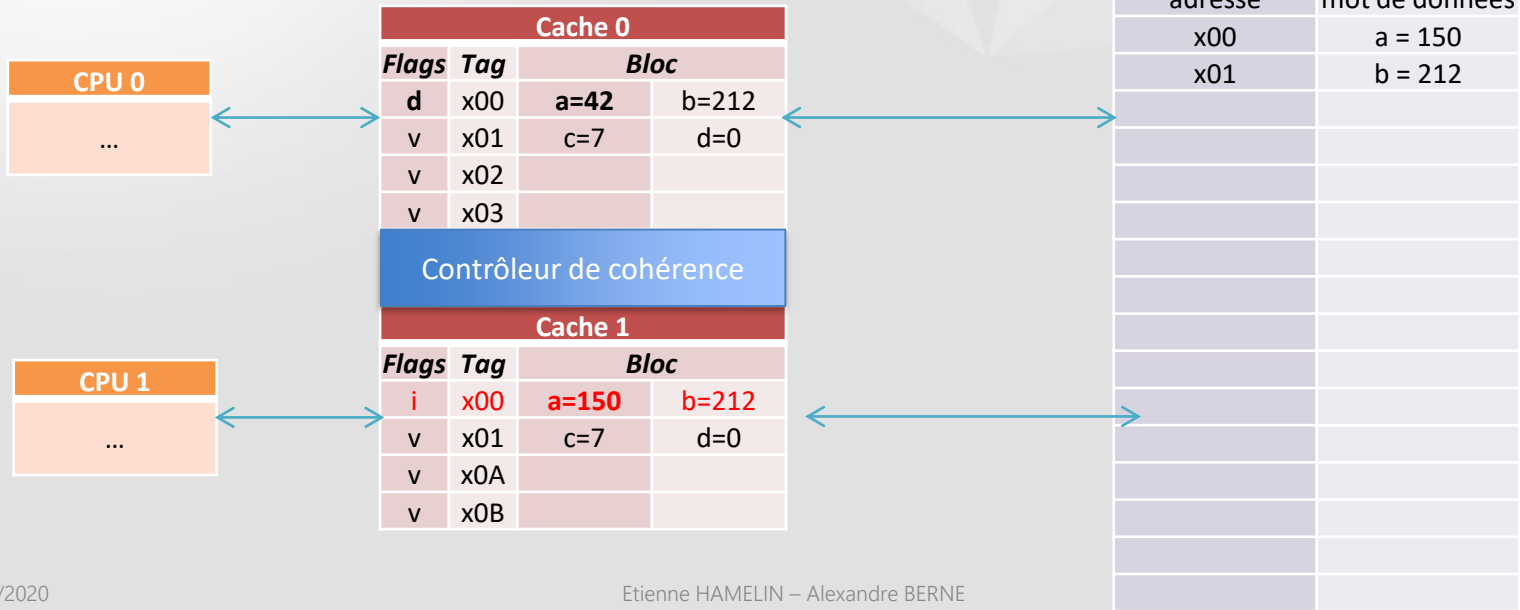




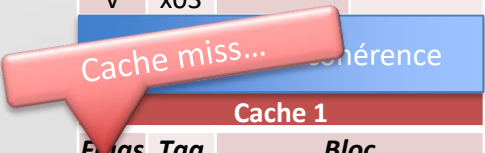
## Protocole *write-invalidate*

Ligne Cache0::tag x00 modifiée → flag *dirty*

Contrôleur de cohérence propage l'info, **invalide** la liche tag x00 de Cache1.



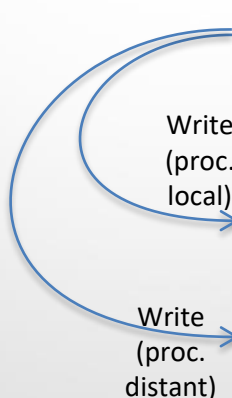
## Fonctionnement d'un cache



## Fonctionnement d'un cache

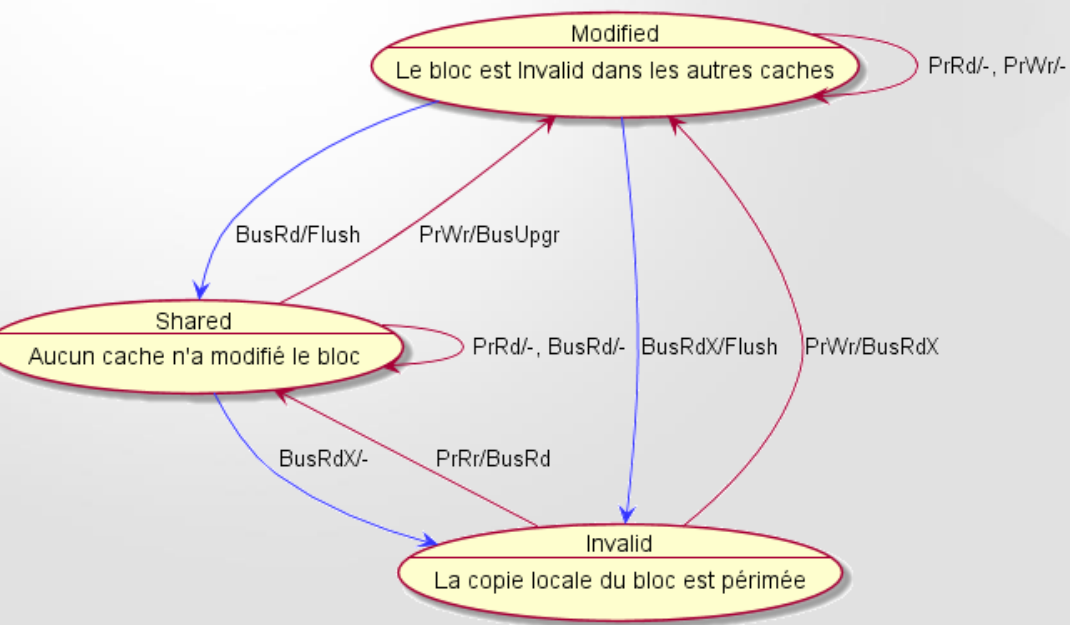


## ► Protocole MSI : un bloc de cache peut être dans l'état

- 
- *Shared* (v)
    - le bloc est cohérent avec la RAM;
    - d'autres caches peuvent éventuellement avoir une copie
  - *Modified* (d)
    - le bloc a été modifié par le processeur local
    - la RAM et les autres caches sont périmés
  - *Invalid* (i)
    - le bloc a été modifié dans un autre cache; la copie locale est périmée.

## ► Variantes

- MESI: *Modified, Exclusive, Shared, Invalid* ()
- MOESI: *Modified, Owned, Exclusive, Shared, Invalid*



## Note:

PrRd: lecture par le processeur propriétaire du cache

PrWr: écriture par le processeur

BUSRd: demande sur le bus de lecture par un processeur

BUSRdX: demande sur le bus de lecture exclusive pour écriture par un processeur

BUSUpgr: demande de mise à jour sur le bus

Flush: écriture du bloc sur le bus

Cache local Autres caches	M	S	I
M	✗	✗	✓
S	✗	✓	✓
I	✓	✓	✓

## ► Familles de protocole

- *Write update / write broadcast*
  - Contrôleur de cohérence partage les écritures de blocs à tous les caches
- *Write invalidate*
  - Contrôleur de cohérence invalide les autres lignes de cache
  - Le bloc sera rechargé en cache plus tard

## ► Implémentations

- *Snooping-based*
  - Chaque cache espionne les requêtes RAM (ou shared cache) des autres processeurs sur leurs cache
- *Directory-based*
  - Une table de méta-données (côté interface RAM/shared cache) connaît l'état de partage des lignes de cache

## ► Cohérence: plusieurs proc. voient 1 bloc

- Propagation des écritures: un Write sur un bloc doit devenir visible à tous les processeurs (*après un délai variable*)
- Sérialisation des écritures: les Write sur un même bloc doivent être vus par tous les processeurs *dans le même ordre*

## ► Consistance

- Lectures & écritures concurrentes prennent un temps variable : comment plusieurs processeurs voient les Read/Write sur *plusieurs blocs*

## ► Modèles de consistance

- Idéalement: *consistance stricte*. Nécessite que les Read/Write soient propagés instantanément  $\Rightarrow$  trop coûteux !
- *Consistance séquentielle* :
  - les Writes sont vus dans *le même ordre*,
  - et cet ordre est compatible avec la séquence d'instruction de chaque processeur



## ► Exemple: consistance mémoire

Partant de l'état  $a = b = x = y = 0$ ,

Quelles couples  $(x, y)$  peut produire ce code ?

Thread0 / Processeur 0	Thread1 / Processeur 1
(1) $a = 1;$ (2) $x = b;$	(3) $b = 1;$ (4) $y = a;$

► ça dépend de l'ordre d'exécution des instructions...

Si on respecte  $(1) < (2)$  et  $(3) < (4)$ : on ne peut pas avoir  $(x, y) = (0, 0)$ .

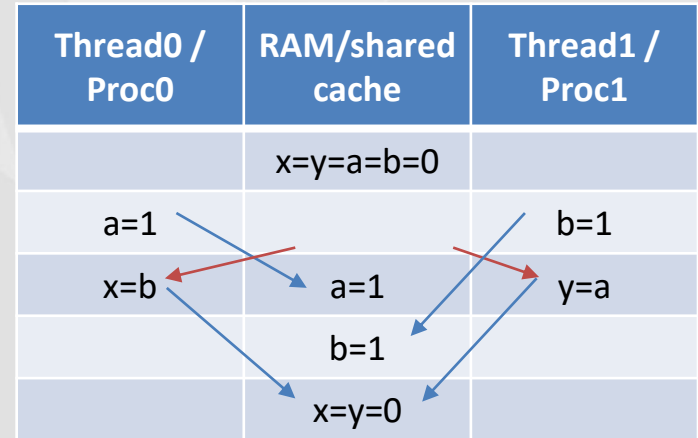
Thread0 / Proc0	Thread1 / Proc1
(1) $a = 1;$ (2) $x = b;$	(3) $b = 1;$ (4) $y = a;$

Ordre	Résultat (x,y)
(1); (2); (3); (4)	(0,1)
(1); (3); (2); (4)	(1,1)
(1); (3); (4); (2)	(1,1)
(3); (1); (2); (4)	(1,1)
(3); (1); (4); (2)	(1,1)
(3); (4); (1); (2)	(1,0)

# Séance 3

## Inconsistance mémoire

```
(x = 0, y = 0) :      141 times (0.001%)
(x = 0, y = 1) : 9999032 times (99.990%)
(x = 1, y = 0) :      787 times (0.008%)
(x = 1, y = 1) :       40 times (0.000%)
```



## ▶ Modèle de consistance faible x86

- *"loads may be reordered with older stores to different locations"*

## ▶ Barrière mémoire *memory fence*

- *Full-fence*: tous les accès Read/Write avant la barrière doivent terminer avant d'exécuter un accès Read/Write après la barrière.
- En C: `asm volatile ("mfence" ::: "memory");` en pratique: caché derrière les primitives de synchronisation `_sync_bool_compare_and_swap`, `mutex_*` etc.
- En C++: `atomic<bool> flag;`



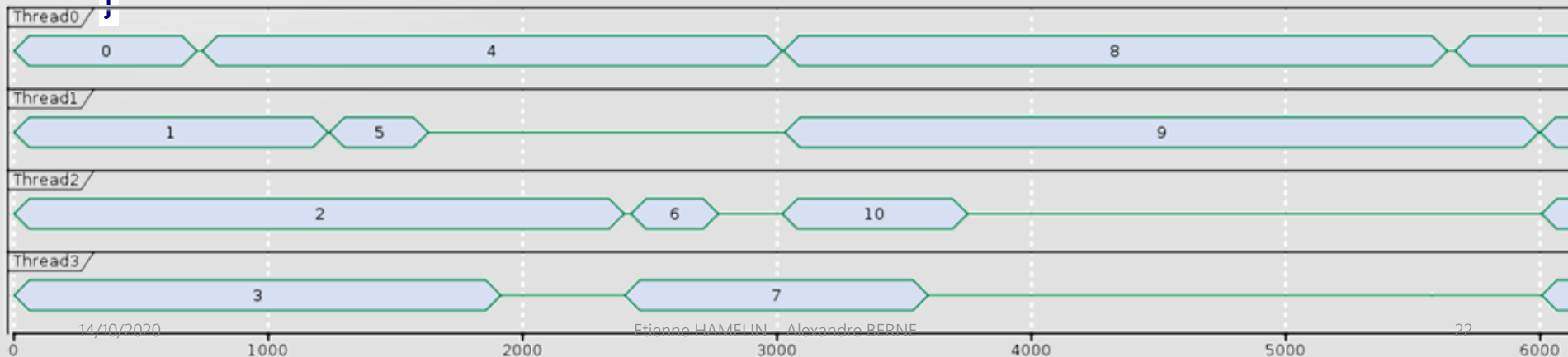
▶ Séance 3

# **Equilibrer la charge**

## Exemple : fractales

- Chaque image  $\rightarrow$  1 job.
- Ordonnancement statique: job  $j \rightarrow$  thread  $j \bmod N$

```
for (job = 0; job < 16; job++) {  
    pthread_join(threads[job % N]);  
    pthread_create(threads[job % N], NULL, compute, &data[job]);  
}
```



## ► Découpage de l'image

- Pixel blanc : jusqu'à 255 itérations
- Pixel noir : très peu d'itérations
- Bandes centrales

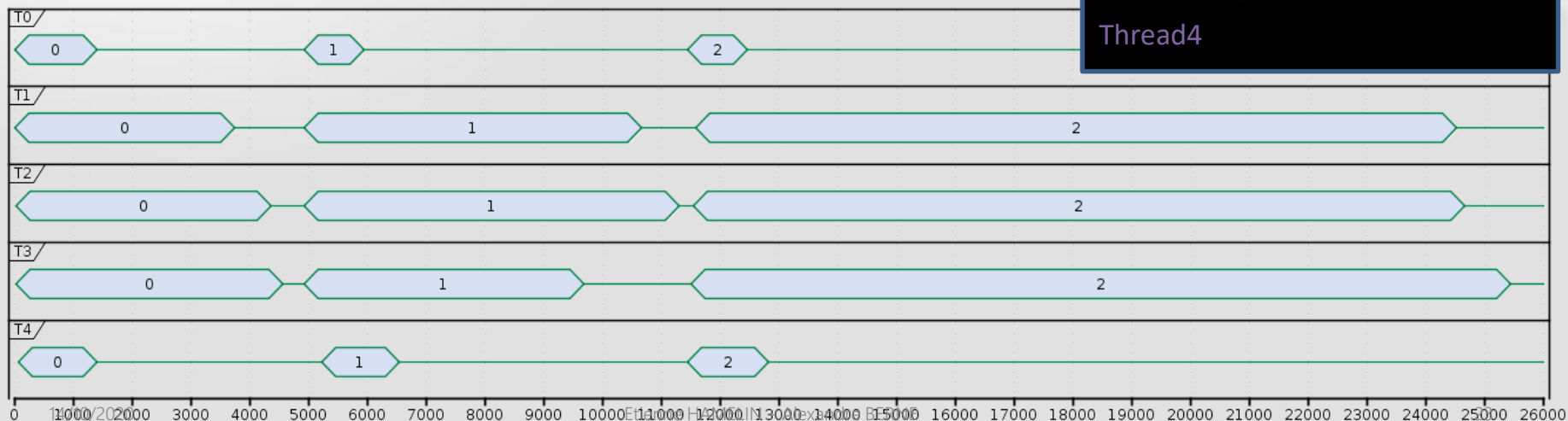
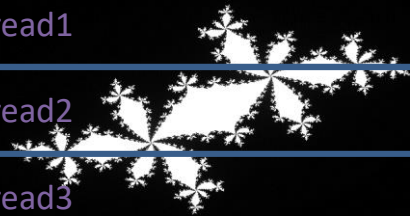
Thread0

Thread1

Thread2

Thread3

Thread4



## ► Ordonnancement dynamique

- Chaque thread:

```
do {  
    job = fetch_job();  
    process(job);  
} while (job != null);
```

Mais **attention**: fonction `fetch_job` réentrante... difficile à programmer correctement!



## ► Load balancing

- Minimiser le temps de calcul -> maximiser l'occupation des processeurs
- Répartition
  - Statique: allocation job->thread définie à la compilation
    - Exemples: « chacun son tour » *round-robin*, aléatoire, ...
    - OpenMP: `schedule(static, chunk_size)`
  - Dynamique: allocation job->thread déterminée au runtime
    - Centralisé (thread maître) ou décentralisé (+ synchronisations!)
    - OpenMP: `schedule(dynamic, chunk_size)` (centralisé)



▶ Séance 3

# Répartir les données

## ► Rappel: layout mémoire

- La pile (*stack*) : variables locales des fonctions ; chaque thread a son stack
- Le tas (*heap*) : espace alloué dynamiquement (e.g. *malloc*) ; 1 tas par processus
- Tableaux et structures sont stockés linéairement

```
char *c="Hello";
int32_t i[2][2]={ {00,01}, {10, 11} };
double d[2]; // !variable non initialisée!
```

Adresse	Octet0	Octet1	Octet2	Octet3	Octet4	Octet5	Octet6	Octet7
0x7ffc8eb6cfd8								
0x7ffc8eb6cfd0	H	e	l	l	o	0x00	?	?
0x7ffc8eb6cfc8	i[1][0] = 10				i[1][1] = 11			
0x7ffc8eb6cfc0	i[0][0] = 00				i[0][1] = 01			
0x7ffc8eb6cfb8	d[1] = ?							
0x7ffc8eb6cfb0	d[0] = ?							

Etienne HAMELIN – Aérospatiale ERNE

## ► Dans chaque thread / sur chaque cœur

- Limiter les *capacity* & *conflict-misses*

- améliorer la localité spatiale :

- structures de données,  
tableaux

- améliorer la localité temporelle

- structure des boucles  
pre-fetch

- **Exemple: recherche par clé dans un tableau**
  - Pattern: nombreuses lectures des clés; peu de lecture des données.

- **Lectures disjointes**

```
struct {  
    int key;  
    char data[128];  
} a_table[N];  
  
// recherche par clé k  
for (i = 0; i < N; i++) {  
    if (a_table[i].key == k) {  
        do_something(a_table[i].data);  
    }  
}
```

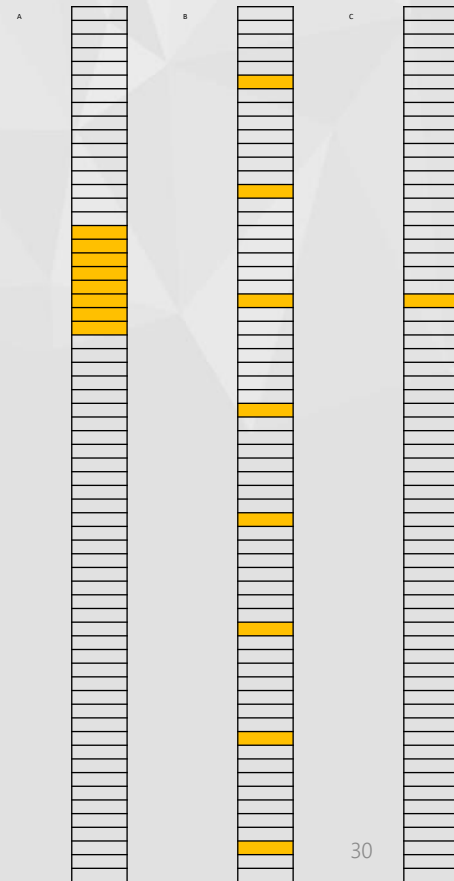
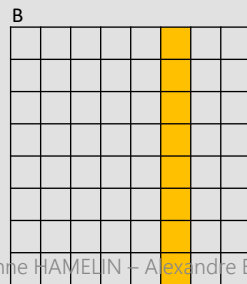
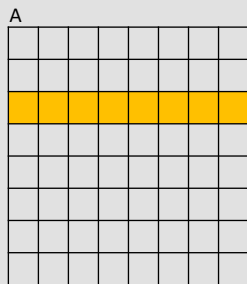
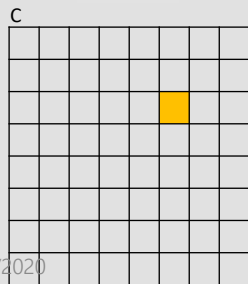
- **Lectures contiguës**

```
int a_key[N];  
char a_data[N][128];  
  
// recherche par clé k  
for (i = 0; i < N; i++) {  
    if (a_key[i] == k) {  
        do_something(a_data[i]);  
    }  
}
```

## Exemple: multiplication de matrices

$$c_{i,j} = \sum_{k=0 \dots N} a_{i,k} \cdot b_{k,j}$$

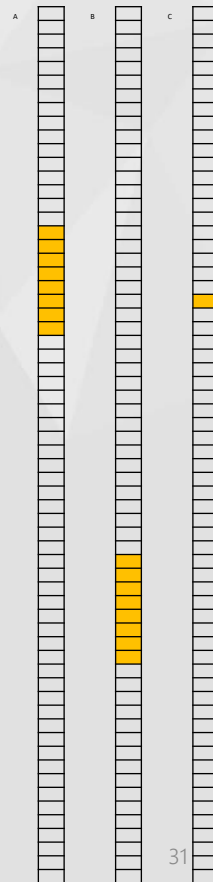
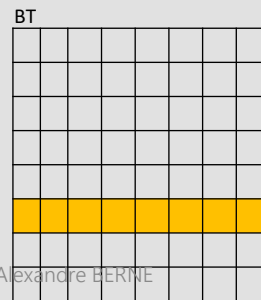
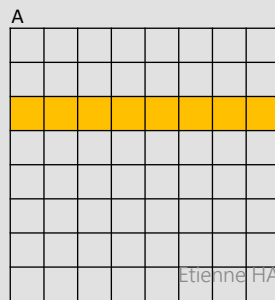
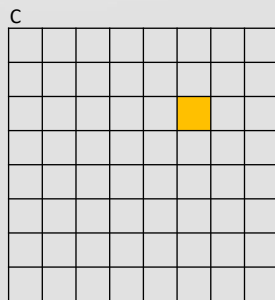
```
for (i = 0; i < N; i++) {  
  for (j = 0; j < N; j++) {  
    C[i][j] = 0;  
    for (k = 0; k < N; k++) {  
      C[i][j] = C[i][j] + A[i][k] * B[k][j];  
    }  
  }  
}
```



### Exemple: multiplication de matrices

```
for (i = 0; i < N; i++) {  
  for (j = 0; j < N; j++) {  
    BT[i][j] = B[j][i];} // transposition
```

```
for (i = 0; i < N; i++) {  
  for (j = 0; j < N; j++) {  
    C[i][j] = 0;  
    for (k = 0; k < N; k++) {  
      C[i][j] = C[i][j] + A[i][k] * BT[j][k];}}
```



## ► Pré-chargement (*prefetch*)

– gcc:

```
__builtin_prefetch(address);
```

```
for(i = 0; i < N; i++) {  
    __builtin_prefetch(&A[i+s]);  
    __builtin_prefetch(&B[i+s]);  
    A[i] += k * B[i];  
}
```

fetch A[56], B[56];  
A[53] += k\*B[53];

fetch A[57], B[57];  
A[54] += k\*B[54];

fetch A[58], B[58];  
A[55] += k\*B[55];

fetch A[59], B[59];  
A[56] += k\*B[56];

fetch A[60], B[60];  
A[57] += k\*B[57];



## ► Partitionner les données

- Données partagés: à **éviter** autant que possible !  
Mais souvent nécessaire...
- Risques:
  - Variables partagées (*true sharing*)
    - écritures concurrentes: *race-conditions* **!danger ahead!**
    - écritures & lectures concurrentes: invalidation de cache, ordre non garanti
  - Outils :
    - Synchronisations: mutex/semaphores, #pragma omp critical / atomic, ... : coûteux
    - Structures *lock-free* + barrières mémoires (au bon endroit) : +performant, **seulement si** vous savez ce que vous faites!
    - Minimiser les synchros: e.g. travail sur copie locale + réduction
  - Moins grave: faux partage (*false sharing*)
    - nombreux cache miss (coherency)
- outil: séparer les données (*padding*)

## ► Méthodes de synchronisations

- Mutex: ne peut être verrouillé que par 1 thread à la fois
- Semaphore: compteur de jetons disponibles ( $> 0$ ) ou de threads en attente ( $< 0$ ).
- Verrou tournant (*spinlock*): attente active
  - si vous savez que l'attente sera courte ou très rare
- Avec l'aide du matériel: *lock-free programming*
  - Test-and-set, Compare-and-swap, etc.
  - Barrières mémoire

## ► Il faut connaître les risques

- Famine, interblocage, inversion de priorité...

### ► Rappel: false sharing

- Défauts de cache lorsque les threads manipulent des variables distinctes, situées *dans le même bloc*

```
void main() {  
    int x, y; // dans le même bloc mémoire
```

CPU0	CPU1
<pre>void thread0() {     for (...) {         x += f(...);     } }</pre>	<pre>void thread1() {     while(...) {         y = g(...);     } }</pre>

- Chaque fois que CPU0 (thread0) écrit x, il invalide le bloc → le prochain accès de thread1 fera un cache miss. Idem quand Thread1 écrit y.
- Le bloc fait du « ping-pong » entre CPU0 et CPU1. Cache miss **inutiles**, car aucune variable réellement partagée.

## ► Outils

### – Padding

```
struct {  
    int data;  
    char __padding[64 - sizeof(int)];  
} a_shared_data[N];
```

### – Alignement

syntaxe gcc: `int x __attribute__((aligned (64)))`;