

## Chapitre 7

# Généralités sur la concurrence et la synchronisation dans les systèmes parallèles

### 7.1 La « *strict consistency* »

Sur une seule machine mono-tâche (notre modèle de programmation le plus habituel, la « machine de Turing » ou de Von Neumann), on est habitué à ce que la mémoire nous offre une garantie très forte, nommée la « *strict consistency* », dont la définition est la suivante : *toute lecture d'une variable renvoie la dernière valeur écrite*. Nous avons l'habitude que la RAM d'un ordinateur, ou que son système de fichier fonctionne de cette manière-là.

Lorsqu'on a l'habitude d'écrire des programmes qui s'exécutent séquentiellement, sur une machine de type « architecture de Von Neumann », on développe un modèle mental dans lequel la mémoire nous offre une garantie très forte : *toute lecture d'une variable renvoie la dernière valeur écrite dedans*. Nous avons aussi l'habitude que les systèmes de fichiers offrent à peu près cette garantie.

Cette s'appelle la « *strict consistency* ». Les compilateurs et les processeurs garantissent que ce modèle est respecté par les programmes séquentiels, et la notion a du sens : dans *un* programme séquentiel, on peut parler de *la* dernière valeur écrite dans une variable.

 En réalité, compilateurs et processeurs prennent des libertés importantes avec la *strict consistency*, mais ils se débrouillent pour que ça n'affecte pas les programmes séquentiels, et la plupart des programmeurs n'en ont pas conscience. Mais ça peut affecter les programmes parallèles ! Plus de détails au chapitre 8.

Sur des machines parallèles, les choses deviennent plus compliquées. Pour commencer, sur une machine à plusieurs coeurs, que se passe-t-il si plusieurs *threads* décident d'écrire des valeurs différentes à la même adresse mémoire *en même temps* ?

**En parallèle, la *strict consistency* n'a pas de sens** Le problème dans les machines parallèles, c'est qu'il n'y a pas de notion raisonnable d'un *temps global*.

Premier exemple : imaginons une variable  $x$  stockée sur une machine  $S$  (comme Serveur). Dans le *rack* d'à côté, une machine  $C$  (comme Client) prend la décision de lire la valeur de  $x$ . Un message est donc envoyé de  $C$  vers  $S$  sur le réseau, et il est émis à l'instant  $t$ . Un tout petit peu plus tard, au temps  $t' = t + \varepsilon$ , un processus sur le serveur décide de modifier la valeur de  $x$ . La *strict consistency* exigerait que le client reçoive la vieille valeur de  $x$ , car dans l'absolu sa lecture a lieu *avant* la modification. Le problème, c'est que si le câble réseau qui sépare  $S$  et  $C$  mesure 1m de long, et que le décalage entre les deux opérations est de  $\varepsilon = 1$  nanoseconde, il faudrait que le signal envoyé au temps  $t$  dans le câble réseau aille trois fois plus vite que la lumière pour atteindre le serveur avant  $t'$ . Ceci semble exclu par les idées d'Einstein sur la relativité<sup>1</sup>. Il est donc inévitable, sur des machines parallèles, de « voir » des événements qui auront lieu... dans le futur !

Deuxième exemple (c'est le même problème, mais dans l'autre sens). Sur les tous derniers processeurs AMD EPYC 7742 « Rome », les coeurs peuvent être distants de 6cm (cf. fig 7.1). Cela signifie que, dans le meilleur des cas, l'information mettrait 0.2ns pour parcourir cette distance (à la vitesse de la lumière, ce qui est *très*

1. C'est en tout cas ce que l'auteur en a compris

## AMD EPYC Rome

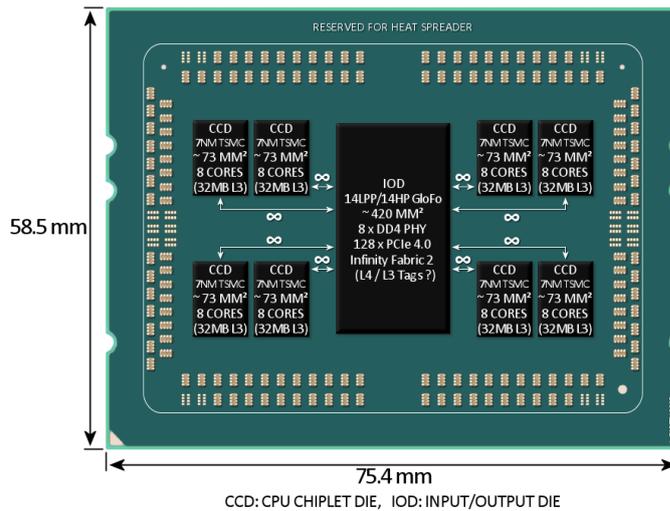


FIGURE 7.1 – Un processeur AMD EPYC Rome, taille réelle. Il y a 8 « chiplets » de 8 coeurs chacuns, physiquement séparés.

optimiste. En réalité ça doit prendre bien plus longtemps). À 3.4Ghz (la fréquence maximum de la puce), la durée d'un cycle d'horloge est de 0.29ns. Par conséquent, lorsqu'un des coeurs décide de modifier une variable  $x$ , les autres coeurs *ne peuvent pas le savoir avant plusieurs cycles* car l'information n'a tout simplement pas le temps de leur parvenir. Du coup, au moins pendant cet intervalle, les autres coeurs vont continuer à percevoir l'ancienne valeur de  $x$ . Ceci viole la *strict consistency* qui exigerait qu'ils perçoivent la dernière valeur écrite.

Dernier exemple : imaginons deux machines, toujours distantes d'un mètre. Les deux machines attendent toutes les deux qu'un certain événement se produise (par exemple, une intervention d'un utilisateur), et elles veulent déterminer sur laquelle l'événement se produit en premier. Pour cela, dès que l'événement a lieu sur une machine, elle envoie à l'autre un message sur le réseau disant « J'AI GAGNÉ ». Le problème, c'est que si les événements ont lieu dans un intervalle trop court, les messages risquent de se croiser... Et plus les machines sont éloignées physiquement, plus l'intervalle entre lequel il est impossible de faire la différence est grand.

## 7.2 La « Sequential Consistency »

### 7.2.1 Relations d'ordre sur les événements dans une machine à mémoire partagée

Lorsqu'un thread s'exécute, des *événements* ont lieu : lecture/écriture en mémoire, opération arithmétique, envoi/réception de message sur le réseau, etc. Indépendamment de leur nature, ces événements ont lieu dans l'ordre spécifié par le programme qui s'exécute (c'est le « *program order* »). Dans les machines parallèles, on peut au moins avoir une certitude<sup>2</sup> : les instructions programmes séquentiels sont exécutés dans l'ordre. On écrit  $e_1 \xrightarrow{po} e_2$  pour dire que l'évènement  $e_1$  a lieu *avant*  $e_2$  dans l'ordre dicté par le programme.

Il y a une deuxième chose dont on peut être sûr : un message qui transite sur le réseau est forcément émis avant d'être reçu. Ainsi, si  $e_1$  est l'évènement « envoi du message  $M$  » et  $e_2$  est l'évènement « réception du message  $M$  », alors on écrit  $e_1 \xrightarrow{net} e_2$ .

Discutons maintenant plus précisément de l'accès à une mémoire partagée. Les lectures et les écritures en mémoire effectuées par différents threads peuvent être concurrentes, mais les opérations qui concernent les mêmes variables ont lieu dans un certain ordre, qu'on va tâcher de caractériser.

On considère un ensemble de threads  $\mathcal{T} = \{T_1, \dots, T_m\}$ . Ils exécutent donc leurs instructions de manière séquentielle. On utilise la notation suivante, qui est bien commode, pour les opérations d'accès à la mémoire :

- On note  $W_i(x)a$  l'évènement : « le thread  $T_i$  écrit la valeur  $a$  dans la variable  $x$  ».
- On note  $R_i(x)b$  l'évènement : « le thread  $T_i$  écrit lit la variable  $x$  et obtient la valeur  $b$  ».

Lorsqu'une opération de lecture a lieu en mémoire, la valeur renvoyée est le résultat de l'exécution d'une certaine opération d'écriture. Si  $w = W(x)a$  est l'écriture qui stocke dans  $x$  la valeur lue par une lecture  $r = R(x)a$ , alors

2. enfin, certitude, on verra...

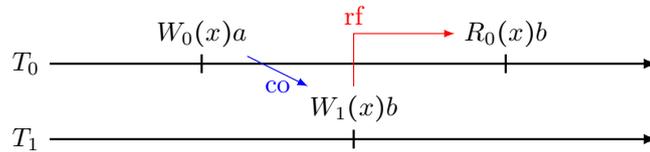
on dit que  $r$  « reads from »  $w$  et on écrit  $w \xrightarrow{rf} r$ . On peut affirmer que pour toute lecture  $r$ , il existe une unique écriture  $w$  telle que  $w \xrightarrow{rf} r$ . Si  $w \xrightarrow{rf} r$ , alors la lecture a lieu « après » l'écriture.

Une variable ne peut stocker qu'une seule valeur à la fois. Les écritures dans une même variable ont donc forcément lieu à la suite les unes des autres, dans un certain ordre (même si cet ordre n'est pas celui qu'on croit). On note  $w_1 \xrightarrow{co} w_2$  (« cohérence ») lorsque  $w_1$  et  $w_2$  sont des écritures dans la même variable  $x$  et que  $w_1$  a lieu « avant ». Si on se restreint aux événements d'écriture sur une seule variable, alors la relation  $\xrightarrow{co}$  est un ordre *total*.

Enfin, si  $w$  est une écriture sur une variable  $x$  qui a lieu *après* une lecture  $r$ , on note  $r \xrightarrow{fr} w$  (« from read »). Cela signifie que la lecture renvoie une valeur qui se trouvait dans  $x$  « avant » l'écriture, donc que la lecture a forcément lieu « après ». En fait  $r \xrightarrow{fr} w \stackrel{def}{=} \exists w'. w' \xrightarrow{rf} r \wedge w' \xrightarrow{co} w$ . Alternativement, on peut dire que  $\xrightarrow{fr} = (\xrightarrow{rf})^{-1}; \xrightarrow{co}$ .

Toutes ces relations sont transitives (c.a.d. que  $u \rightarrow v$  et  $v \rightarrow w$  impliquent  $u \rightarrow w$ , pour chacune des relations considérées).

En présence d'un programme séquentiel, les relations  $\xrightarrow{rf}$ ,  $\xrightarrow{co}$  et  $\xrightarrow{fr}$  coïncident avec  $\xrightarrow{po}$  : toutes les opérations qui ont lieu ont lieu dans l'ordre indiqué par le programme séquentiel exécuté. Mais ceci n'est plus vrai en présence de plusieurs threads. Par exemple :



On a bien  $W_0(x)a \xrightarrow{po} R_0(x)b$  pour  $T_0$ , et en présence d'un seul thread on devrait avoir  $W_0(x)a \xrightarrow{rf} R_0(x)a$ , mais à cause de l'écriture effectuée par  $T_1$  la valeur lue n'est plus la bonne, et donc on n'a *pas*  $W_0(x)a \xrightarrow{rf} R_0(x)b$ . Dans cet exemple précis on a

$$W_0(x)a \xrightarrow{co} W_1(x)b \xrightarrow{rf} R_0(x)b.$$

### 7.2.2 Définition de la *Sequential Consistency*

Lorsqu'on a l'habitude d'écrire des programmes séquentiels, on a un modèle mental dans lequel la mémoire nous offre une garantie très forte : *toute lecture d'une variable renvoie la dernière valeur écrite*. Nous avons aussi l'habitude que les systèmes de fichiers offrent à peu près cette garantie. C'est une sorte de *strict consistency*, et donc en parallèle ça va mal se passer. Que se passe-t-il si deux *harts* décident d'écrire des valeurs différentes à la même adresse mémoire en même temps ?

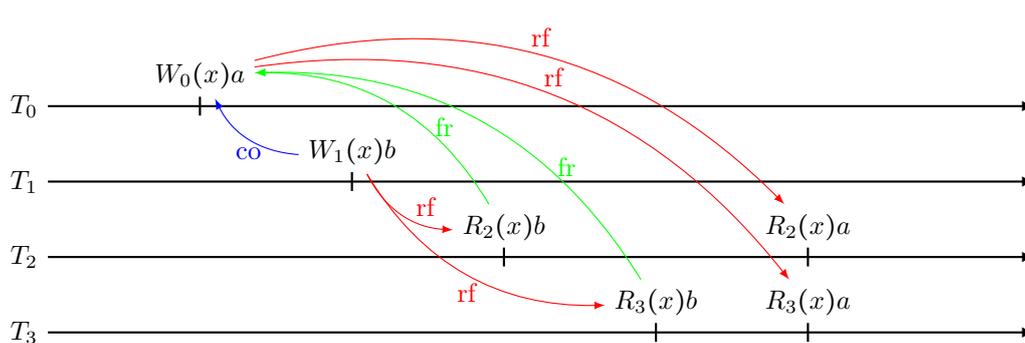
Forcément, un certain *non-déterminisme* va apparaître, et l'un des deux coeurs va « gagner la course ». On a vu au chapitre 4 qu'il est nécessaire, pour garantir la sémantique du code, de protéger les accès concurrents aux variables partagées par des sections critiques, des opérations atomiques ou encore des verrous. Ces mécanismes garantissent au programmeur que tout se passera comme si les accès aux variables partagées avaient lieu séquentiellement. Si plusieurs threads sont en compétition pour entrer dans une section critique, il vont y parvenir les uns à la suite des autres, mais *dans un certain ordre*, qui n'est pas spécifié et qui peut dépendre de l'OS, de l'environnement d'exécution.

Ceci est assez bien capturée par la notion que Leslie Lamport a appelé la *Sequential Consistency* [4]. Un système parallèle possède cette propriété si, bien que les accès à la mémoire des différents threads soient concurrents, tout se passe comme si les opérations de lecture/écriture dans la mémoire étaient exécutées de manière séquentielle.

En particulier, tous les threads observent les écritures vers la mémoire dans le même ordre, et donc ont la même « vision » de l'état de la mémoire. Les accès mémoire des threads peuvent s'entrelacer arbitrairement, mais tous les threads doivent percevoir le même ordre sur les opérations.

#### Exemple

| Considérons quatre threads qui font des opérations sur des variables partagées.



Ici, le comportement du système n'est manifestement pas régi par la *strict consistency* : comme  $W_0(x)a$  a lieu « avant »  $W_1(x)b$ , on ne devrait pas voir réapparaître la valeur  $a$  dans la variable  $x$  par la suite (or  $T_2$  et  $T_3$  observent ceci). Tout se passe pourtant comme si les opérations avaient eu l'une après l'autre dans l'ordre suivant :

$$W_1(x)b < R_2(x)b < R_3(x)b < W_0(x)a < R_2(x)a < R_3(x).$$

Ce que les threads observent pourrait être causé par un ordonnanceur (maléfique...) qui aurait retardé l'exécution de  $T_0$  pour que  $W_0(x)a$  tombe pile au mauvais moment, entre les deux lectures de  $T_3$ . Le système est néanmoins séquentiellement consistant car tout se passe comme si les accès mémoire avaient lieu séquentiellement dans un ordre raisonnable.

Essayons de rendre ça à peu près formel. Une *exécution*  $E_i$  d'un thread  $T_i$  décrit l'ensemble des événements de lecture/écriture de la mémoire (avec les valeurs associées) dont  $T_i$  a commandé l'exécution.

Un système parallèle possède la *sequential consistency* (il est « séquentiellement consistant ») si, pour chacune des exécutions possibles des threads auxquelles il peut aboutir, on peut construire un *historique*, c'est-à-dire une séquence totalement ordonnée  $H$  qui contient une et une seule fois chaque événements de l'ensemble des exécutions  $E_1, \dots, E_n$ . De manière équivalente, cela revient à dire qu'il y a une relation d'ordre totale  $<$  sur les événements. Cette relation/cet historique (c'est la même chose) doit respecter deux contraintes :

1. Compatibilité avec le *program order* : si  $e_1 \xrightarrow{po} e_2$ , alors  $e_1 < e_2$  dans l'historique.
2. Compatibilité avec *reads from* : toute lecture d'une variable  $x$  doit renvoyer la valeur écrite par l'évènement d'écriture sur  $x$  le plus récent dans l'historique.

Une définition alternative de la *sequential consistency* consiste à dire que la relation  $(\xrightarrow{po} \cup \xrightarrow{rf} \cup \xrightarrow{co} \cup \xrightarrow{fr})$  doit être acyclique.

### 7.2.3 La *sequential consistency* est désirable

L'environnement d'exécution d'OpenMP garantit que les accès à des variables partagés dans des sections `omp atomic` sont séquentiellement consistants, de même que les accès mémoire qui ont lieu dans des sections `omp critical`.

Même si elle tolère un certain non-déterminisme, la *sequential consistency* est une caractéristique souhaitable, et elle permet de faire des raisonnements assez simples sur la correction des programmes multi-threads. Par exemple, considérons un mécanisme d'exclusion mutuelle pour deux threads, le *Peterson lock*.

```

bool flag[2];
int victim;

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

void unlock()
{

```

```

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

```

**Théorème 1.** *Si la machine est séquentiellement consistante, alors le Peterson lock garantit l'exclusion mutuelle entre deux threads.*

### Démonstration

Raisonnons par l'absurde, et supposons que ce ne soit pas le cas. Cela signifie que les deux threads  $T_0$  et  $T_1$  peuvent parvenir à exécuter la fonction `lock()`, à en sortir tous les deux et à se retrouver tous les deux dans la section critique (CS) simultanément. Chaque thread exécute donc la séquence d'opérations :

$$\begin{aligned}
T_0 &: W_0(\text{flag}[0])\text{true} \xrightarrow{po} W_0(\text{victim})0 \xrightarrow{po} R_0(\text{flag}[1])? \xrightarrow{po} R_0(\text{victim})? \xrightarrow{po} CS_0 \\
T_1 &: W_1(\text{flag}[1])\text{true} \xrightarrow{po} W_1(\text{victim})1 \xrightarrow{po} R_1(\text{flag}[0])? \xrightarrow{po} R_1(\text{victim})? \xrightarrow{po} CS_1
\end{aligned}$$

Supposons, sans perte de généralité, qu'au moment où les deux threads se trouvent dans la section critique, le thread 0 était le dernier à écrire dans `victim` :

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

Ceci implique que  $T_0$  a observé `victim == 0` dans la condition de la boucle (il n'y a plus d'autre écriture dans `victim` et  $T_0$  doit lire la dernière valeur écrite). Puisque  $T_0$  est entré dans la section critique, c'est forcément qu'il a également observé `flag[1] == false`. Par conséquent, on a :

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

En combinant tout ce qu'on sait, on trouve que :

$$W_1(\text{flag}[1])\text{true} \xrightarrow{po} W_1(\text{victim})1 \xrightarrow{co} W_0(\text{victim})0 \xrightarrow{po} R_0(\text{flag}[1])\text{false}.$$

En particulier, on a  $W_1(\text{flag}[1])\text{true} \xrightarrow{*} R_0(\text{flag}[1])\text{false}$ , et il n'y a pas d'autre écriture dans `flag[1]`.

On a donc  $W_1(\text{flag}[1])\text{true} \xrightarrow{rf} R_0(\text{flag}[1])\text{false}$ , une incohérence.

Par conséquent, la machine n'est pas séquentiellement consistante. CQFD. □

Le *Peterson lock* est exempt de *deadlock* : les deux threads ne peuvent pas rester tous les deux bloqués dans la boucle `while()` car il faudrait que `victim` vaille simultanément 0 et 1. En fait, c'est encore mieux : *Peterson lock* est *starvation-free* : chaque thread qui tente d'acquiescer le verrou fini par y parvenir en temps fini lorsque l'autre le libère.

## 7.3 Modèle mémoire de OpenMP

Tant qu'on se tient à la règle de base de la programmation avec OpenMP (utiliser `atomic` + `critical` pour tous les accès potentiellement conflictuels), la *sequential consistency* est garantie et on ne devrait pas avoir de problème.

Ceci dit, si on veut aller plus loin, par exemple parce qu'on a des problèmes de performance liés au coût des synchronisations, il faut être conscient qu'OpenMP n'offre que des garanties assez faibles sur ce qui se passe lors des *autres* accès à la mémoire partagée.

Ces garanties sont formalisées dans le *modèle mémoire* offert par OpenMP. Ce modèle mémoire est dit « relaxé » car il ne garantit pas que les différents threads aient une vue cohérente du contenu de la mémoire partagée. En particulier, il ne garantit *pas du tout* la *sequential consistency* pour les accès mémoire non-protégés par `atomic` ou `critical`.

### 7.3.1 Description

- Tous les threads ont accès à la mémoire partagée.
- Cependant, chaque thread peut avoir, de manière *privée*, une *vue temporaire* de la mémoire. La nature précise de cette vue temporaire privée ne fait pas partie de la description de OpenMP, mais peut être matérialisée par des données stockées dans des registres du processeur, dans des caches, etc.
- La vue temporaire de la mémoire d'un thread n'est pas nécessairement synchronisée en permanence avec le contenu de la mémoire partagée.

- Une valeur écrite dans une variable par un thread peut rester dans sa vue temporaire pendant un certain temps, jusqu'à ce qu'elle soit finalement « poussée » vers la mémoire. Du coup, la modification n'est pas forcément visible instantanément par les autres threads.
- Une valeur lue depuis une variable partagée peut provenir de la vue temporaire du thread et être « périmée » (une valeur plus récente a été écrite en mémoire, mais la vue temporaire du thread n'a pas été rafraîchie).
- Les vues privées temporaires des threads sont synchronisées avec la mémoire à certains moments : lors de `omp barrier` ; lors de la sortie de toutes les constructions de partage de travail (`omp for`, `omp sections` et `omp single`) ; lors de l'entrée et de la sortie de `omp parallel`, `omp critical`, `omp atomic` ; lors de chaque point d'ordonnancement/annulation des tâches. Dans cette liste, les exceptions notables sont l'entrée des constructions de partage de travail ainsi que l'entrée et la sortie de `omp master`.
- On peut forcer la synchronisation avec `omp flush`.

On pourrait faire une analogie avec les systèmes de contrôle de version tels que `git` ou `subversion` avec lesquels les programmeurs sont familiers : chaque développeur possède sa « copie privée » du code et se synchronise avec une copie centrale de temps en temps.

### 7.3.2 Justification de la « faiblesse » du modèle mémoire

Ce modèle mémoire est dans le fond adapté pour des raisons de performance et d'architecture matérielle. En effet, les compilateurs d'une part et le matériel d'autre part se permettent un certain nombre de libertés dans la façon dont le code écrit par les programmeurs est traduit et exécuté. Ces libertés ne sont pas perceptibles dans la programmation séquentielle, mais elles le deviennent en multi-thread. Le modèle mémoire `OpenMP` incorpore ces libertés.

**Les compilateurs** Les compilateurs, surtout lorsqu'ils tentent d'optimiser le code produit, gardent au maximum les valeurs des variables dans les registres du processeur. Mais voilà : ceci constitue une « vue temporaire privée » : pendant que la valeur d'une variable  $x$  est stocké dans le registre `%rax` du CPU, elle n'est pas « synchronisée » avec sa valeur en mémoire.

Ceci est illustré par le petit bout de code suivant. Le thread  $T_0$  fait  $2 \cdot 10^6$  iterations tandis que  $T_1$  n'en fait que la moitié, et va donc sortir de la boucle en premier, puis modifier  $n$ . Mais ceci ne sera pas perçu par  $T_0$  qui reste avec sa « vue privée temporaire ». Quand  $T_0$  sort de sa section parallèle, sa vue privée temporaire est synchronisée avec la mémoire partagée, donc il écrit sa valeur de  $n$  en mémoire.

```
double n = 0, m = 0xDEADBEEF;
#pragma omp parallel num_threads(2)
{
    int i = omp_get_thread_num();
    for (int j = 0; j < 1000000 * (1 + (i == 0)); j++) {
        if (i == 0)
            n += 1.0;
        else
            m += 1.0;
    }
    if (i == 1)
        n = m;
}
printf("n = %.0f, m = %.0f\n", n, m);
```

On obtient :

`n = 2000000, m = 3736928559`



Un petit examen du code assembleur produit par le compilateur confirme ceci. On voit clairement que  $n$  et  $m$  sont copiées dans des registres au début de la boucle, puis restaurées en mémoire à la fin de la boucle.

```
parallel_section:
    pushq   %rbx                ; préambule
    movq   %rdi, %rbx           ;
    call   omp_get_thread_num@PLT ; i = omp_get_thread_num()
    movsd  8(%rbx), %xmm0       ; lire la valeur de m en mémoire, stocker dans %xmm0
    movsd  (%rbx), %xmm1        ; lire la valeur de n en mémoire, stocker dans %xmm1
    testl  %eax, %eax           ; if (i == 0) ...
```

```

je      T_0                ; ... goto T_0

T_1:
movsd   .LC0(%rip), %xmm2  ; %xmm2 = 1.0
movl    $1000000, %edx     ; j = 1000000

loop1:
addsd   %xmm2, %xmm0      ; m += 1.0
subl    $1, %edx          ; j--
jne     .L3               ; if (j != 0) goto loop1
movsd   %xmm0, 8(%rbx)    ; écrire la valeur de m en mémoire depuis %xmm0
cmpl   $1, %eax          ; if (i == 0) ...
jne     .L1               ; ... goto end1
movsd   %xmm0, (%rbx)    ; écrire la valeur de n en mémoire depuis %xmm1

end1:
popq    %rbx              ; fin de la section parallèle
ret

T_0:
movsd   .LC0(%rip), %xmm2  ; %xmm2 = 1.0
movl    $2000000, %eax     ; j = 2000000

loop0:
addsd   %xmm2, %xmm1      ; n += 1.0;
subl    $1, %eax          ; j--
jne     loop0             ; if (j != 0) goto loop1
movsd   %xmm1, (%rbx)    ; écrire la valeur de n en mémoire depuis %xmm1
popq    %rbx              ; fin de la section parallèle
ret

```

**Le matériel** Il y en outre des raisons architecturales. Presque toutes les architectures matérielles des microprocesseurs actuels<sup>3</sup> sont de type *load-store* : les opérations arithmétiques ne peuvent se faire qu'entre les registres du processeur, et pas de/vers la mémoire. Il faut donc, dans l'ordre : 1) charger les données dans des registres (*load*) ; 2) faire les calculs ; 3) écrire le résultat en mémoire (*store*). Sur un tel processeur, le thread a une « vue privée temporaire » du contenu de la mémoire à partir de la fin de l'étape 1 (la valeur est chargée dans un registre ; même si elle change en mémoire pendant l'étape 2 ou 3, l'ancienne valeur reste dans le registre du processeur).

Voyons un autre exemple<sup>4</sup> :

```

int x[2] = {};
#pragma omp parallel for
for (int i = 0; i < 1000000000; i++) {
    int j = i & 0x00000001; /* récupère le bit de poids faible de i */
    x[j]++;
}
for (int i = 0; i < 2; i++)
    printf("x[%2d] : %10u\n", i, x[i]);

```

Comme il y a autant d'entiers pairs et impairs dans l'intervalle  $[0; 10^9]$ , on s'attend à voir apparaître la valeur  $5 \cdot 10^8$  dans les deux cases de *x*. C'est bien le cas si on exécute ce code en désactivant OpenMP lors de la compilation. Résultat avec OpenMP (sur un laptop à deux coeurs x86-64) :

```

x[ 0] : 262301234
x[ 1] : 262261007

```

Le programme ne produit *jamais* les bonnes valeurs. Le problème, c'est que l'instruction `x[j]++`; du langage C n'est pas *atomique*. Au contraire, elle se décompose en trois phases :

1. Charger  $x[j]$  depuis la mémoire vers le CPU.
2. Incrémenter la valeur.
3. Écrire le résultat dans la mémoire.

Voici (vraisemblablement) ce qui se passe lors de l'exécution du test :

3. en tout cas ceux de la famille RISC : les ARM, POWER, SPARC, MIPS, RISC-V, etc. et même les x86-64 si on gratte un peu sous le jeu d'instruction qui date de la fin des années 1970...

4. DISCLAIMER : Le programme de test n'est pas aussi simple que voulu. Je voulais initialement faire ce test avec `for (int i = 0; i < 1000000000; i++) x++`; mais le compilateur gcc 8.3.0 est suffisamment évolué pour remplacer automatiquement ceci par `x = 1000000000`;

1. Le thread 0 lit la valeur de  $x[j]$ .
2. Le thread 1 lit la valeur de  $x[j]$  (c'est la même).
3. Le thread 0 écrit dans  $x[j]$  une nouvelle valeur.
4. Le thread 1 écrit dans  $x[j]$  une nouvelle valeur, et écrase la valeur écrite à l'étape précédente.

Ceci explique qu'on voit apparaître des valeurs plus faibles que celles attendues (puisque l'écriture du deuxième threads « écrase » l'incrémentatation effectuée par le premier).

Ce problème doit être contourné par des moyens logiciels, car sur les architectures *load-store*, la procédure en trois étapes décrite ci-dessus est inévitable.



Un examen du code assembleur généré par gcc sur un Raspberry Pi, donc un ARM, le confirme. On voit bien apparaître la séquence `ldr` (*LoaD into Register*) ; `add` ; `str` (*STore from Register*).

```

; r0 contient 10^9
; r2 contient i
; r3 contient j
; x se trouve à l'adresse sp
; r1 contient l'adresse de x[j]
loop:
    and    r3, r2, #1          ; j = i & 0x00000001
    add    r1, sp, #8          ; [calcul de l'adresse de x[j]
    add    r3, r1, r3, lsl #2  ; r3 = adresse de x[j]
    add    r2, r2, #1          ; i++
    ldr    r1, [r3, #-8]       ; tmp = x[j]
    cmp    r2, r0              ; si i < 10^9...
    add    r1, r1, #1          ; tmp = tmp + tmp + 1
    str    r1, [r3, #-8]       ; x[j] = tmp
    bne   loop                 ; ... alors goto loop

```

De toute façon, même sur les processeurs qui ne sont pas explicitement conçus sur l'architecture *load-store*, comme les x86-64, le problème a lieu aussi (et d'ailleurs le test le démontre).



La situation est même encore pire, car le test démontre en fait que l'exécution d'instructions *individuelles* du processeur n'est même pas atomique. En effet, *une seule* instruction permet d'incrémenter l'entier situé en mémoire à l'adresse donnée (donc, en théorie pas de besoin de `load/add/store`). Le compilateur gcc l'utilise dans le test, et il génère le code assembleur suivant :

```

; %edx contient i
; le tableau x est à l'adresse %rsi
; %ecx contient j
; %ecx contient tmp
; %eax contient la borne de la boucle
loop:
    movl   %edx, %ecx
    addl   $1, %edx           ; i++
    andl   $1, %ecx           ; j = i & 0x00000001
    addl   $1, (%rsi,%rcx,4)  ; x[j]++      (x[j] est à l'adresse %rsi + 4 * %rcx)
    cmpl  %edx, %eax         ; si i < 10^9...
    jne   loop                ; ... alors goto loop

```

L'instruction `addl` ajoute son premier opérande au deuxième, c'est-à-dire pour celle qui nous intéresse la constante 1 au contenu de la case mémoire `(%rsi,%rcx,4)` qui contient  $x[j]$ . Eh bien ce que le test montre, c'est que cette instruction *ne s'exécute pas de manière atomique*.

## 7.4 Les processeurs modernes ne sont pas *Sequentially Consistent*

Une des raisons fondamentales pour lesquelles le modèle mémoire d'OpenMP ne garantit pas la *sequential consistency* c'est... que les processeurs ne la garantissent pas.

Le problème avec la *sequential consistency* c'est que ça a un coût. Maintenir la même vue de la mémoire pour tous les coeurs d'un processeur serait trop coûteux et trop difficile à mettre en oeuvre. Le problème principal réside dans la nécessité de devoir rendre simultanément et instantanément visible à tous les threads chaque écriture. Ceci interdit que les opérations d'accès à la mémoire puissent se dérouler de façon « locale » à chaque coeur, sans interaction avec le reste des processeurs. Aussi, lire et écrire en mémoire nécessiterait forcément des communications et une synchronisation coûteuse pour maintenir la *sequential consistency*.

Sans trop rentrer dans les détails (on y reviendra abondamment au chapitre 9), les processeurs ont des *caches*. Les coeurs ont généralement des caches « privés » dont le contenu n'est pas visible par les autres coeurs. Lorsqu'un thread écrit une valeur dans une variable  $x$ , en réalité seul son cache privé est mis à jour, du moins dans un

premier temps. À un moment indéterminé du futur, lorsqu'il faudra faire de la place dans le cache pour d'autres données plus récentes, l'adresse contenant  $x$  sera évincée du cache et (peut-être) écrite dans la mémoire. Pendant toute une période, le coeur qui écrit  $x$  a une vision de la mémoire différente des autres, dans laquelle  $x$  a la nouvelle valeur. Or ceci est précisément interdit par la *sequential consistency*.



Ceci est vrai si le cache est *write-back*. Mais de toute façon, même s'il est *write-through*, il va y avoir un délai non-négligeable de propagation de l'écriture jusqu'à la RAM.

Du coup, les processeurs modernes n'offrent pas la *sequential consistency*.

### 7.4.1 La *sequential consistency* est trop coûteuse

Voici un argument formel qui démontre que dans un système séquentiellement consistant, les opérations d'accès à la mémoire ne peuvent pas être rapides. Considérons un système à deux processeurs, et notons  $\tau$  la *latence* des communications entre eux (c'est le temps minimal nécessaire pour transférer des informations entre eux). Si  $d$  désigne la distance qui les sépare, alors il semble raisonnable de supposer que  $\tau \geq d/c$ , où  $c$  désigne la vitesse de la lumière. Par ailleurs, on a aussi que  $\tau \geq \rho l$ , où  $\rho$  désigne le temps nécessaire pour franchir un transistor et  $l$  désigne le nombre de transistors sur le parcours. Cette deuxième borne est a priori plus forte que la première.

Le résultat suivant figure dans [6].

**Théorème 2.** *Notons  $r$  et  $w$  les temps nécessaire pour effectuer des opérations de lecture et d'écriture dans le meilleur des cas sur chacun des deux processeurs. Si le système composé des deux processeurs est séquentiellement consistant, alors  $r + w \geq \tau$ .*

#### Démonstration

Considérons deux threads qui accèdent à quatre variables partagées  $x, y, w$  et  $z$ , initialement égales à zéro, et qui s'exécutent sur chacun des deux processeurs. Les deux threads exécutent les programmes suivants :

$$T_0 : W_0(x)1 \xrightarrow{po} R_0(y)w$$

$$T_1 : W_1(y)1 \xrightarrow{po} R_0(x)z$$

Comme la machine est séquentiellement consistante, il est impossible d'avoir  $(w, z) = (0, 0)$  à la fin de l'exécution du programme. En effet, pour obtenir  $w = 0$ , il faut que  $R_0(y)0 \xrightarrow{fr} W_1(y)1$ . Mais alors comme les deux threads effectuent leurs opérations dans l'ordre, on devrait avoir

$$W_0(x)1 \xrightarrow{po} R_0(y)0 \xrightarrow{fr} W_1(y)1 \xrightarrow{po} R_0(x)z,$$

On a donc  $W_0(x)1 \xrightarrow{rf} R_0(x)z$ , donc  $z = 1$ .

Maintenant, supposons que les deux threads exécutent leurs opérations de manière synchrone et que  $r + w < \tau$ . Aucun des deux processeurs ne peut alors « percevoir » l'écriture que l'autre a effectué, car il ne s'est pas écoulé assez de temps pour que l'information lui parvienne. La seule issue possible est alors  $(w, z) = (0, 0)$ , une contradiction. Ceci démontre que  $r + w \geq \tau$ .  $\square$

### 7.4.2 Validation expérimentale du défaut de *Sequential Consistency*

Il est possible de démontrer expérimentalement qu'un ordinateur moderne multi-coeurs n'est pas séquentiellement consistant. Essayons de tester le *Peterson lock* (tous les tests ont lieu sur un laptop normal) :

```
int x = 0;
#pragma omp parallel for num_threads(2)
for (int i = 0; i < 10000000; i++) {
    lock();
    x++;
    unlock();
}
printf("x : %10u\n", x);
```

**Premier essai** Les deux threads restent coincés indéfiniment dans la fonction `lock()`, ce qui semble contredire l'affirmation que le *Peterson lock* est *deadlock-free*.

C'est la faute du compilateur : il suppose que les variables `flag` et `victim` ne sont pas modifiées de manière extérieure. Du coup l'optimiseur retire le test `victim == i` de la condition de la boucle, car il « sait » qu'il est vrai. La boucle devient alors `while(true) {}` et ça bloque le programme.



Voici le code assembleur, avec la boucle infinie bien visible en bas.

```
lock:
    pushq    %rbx
    call    omp_get_thread_num@PLT
    movslq  %eax, %rdx          ; %rdx == i
    leaq    flag(%rip), %rax    ; %rax == adresse de flag
    movq    %rdx, %rbx         ; %rbx == i
    movb    $1, (%rax,%rdx)     ; flag[i] = true
    movl    %edx, victim(%rip) ; victim = i
    movl    $1, %edx           ;
    subl    %ebx, %edx         ;
    movslq  %edx, %rdx         ; j = i - 1
    cmpb    $0, (%rax,%rdx)    ; if flag[j] = 0...
    je     critical           ; ... then goto CS ...

loop:
    jmp     loop              ; ... else goto loop ...
```

Pour empêcher le compilateur d'effectuer cette optimisation, on a deux solutions : désactiver toutes les optimisations (compiler avec `-O0`), ou bien déclarer que les variables `flag` et `victim` peuvent être modifiées extérieurement, donc qu'elles sont `volatile`.

**Deuxième essai** On modifie le code en ajoutant :

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

Ceci permet au programme de s'exécuter et le résultat est :

x : 9999827

Cette fois, l'acquisition du verrou termine en temps fini, mais il semble que l'exclusion mutuelle ne soit pas garantie : on retrouve en apparence un phénomène de non-atomicité des incréments.

**Troisième essai** Il est facile de vérifier explicitement si les deux threads se trouvent ou pas en même temps dans la section critique. Modifions les deux fonctions de la façon suivante :

```
volatile int inside_cs = -1;

void lock()
{
    int i = omp_get_thread_num();
    int j = 1 - i;
    flag[i] = true;           // I'm interested
    victim = i;              // you go first
    while (flag[j] && victim == i) {}; // wait
    assert(inside_cs == -1); // critical section must be empty
    inside_cs = i;          // now I'm in the critical section
}

void unlock()
{
    int i = omp_get_thread_num();
    assert(inside_cs == i); // I was in the critical section, not the other one
    inside_cs = -1;        // now the critical section is empty
    flag[i] = false;      // I'm not interested
}
```

Et voici le résultat, prévisible :

lock: Assertion `inside\_cs == -1' failed.

Nous avons prouvé mathématiquement que le *Peterson lock* garantit l'exclusion mutuelle sur les machines séquentiellement consistantes; l'exclusion mutuelle n'est manifestement pas garantie; conclusion : un laptop x86-64 n'offre pas la *sequential consistency*.

### 7.4.3 Une des causes : le « *Store Buffering* »

On peut pointer le problème plus précisément. Le programme suivant démontre explicitement un défaut de *sequential consistency*. En fait, c'est le même programme utilisé dans la preuve du théorème 2. Il compte le nombre de fois où le processeur exhibe un comportement incompatible avec la *sequential consistency*. Deux threads partagent deux variables. Ils en écrivent une puis lisent l'autre. La question est de savoir s'il peuvent lire tous les deux les valeurs initiales des deux variables (si la machine était séquentiellement consistante ce serait impossible, comme on l'a vu).

```
int relaxed = 0;
int x, y, w, z;

#pragma omp parallel num_threads(2)
{
    int i = omp_get_thread_num();
    for (int j = 0; j < 10000000; j++) {
        #pragma omp master
        x = y = 0;

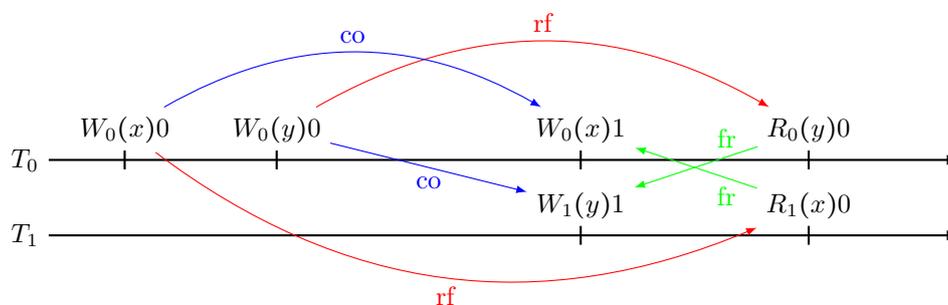
        #pragma omp barrier
        if (i == 0) {
            x = 1;
            w = y;
        } else {
            y = 1;
            z = x;
        }
        #pragma omp barrier

        #pragma omp master
        if (z == 0 && w == 0)
            relaxed++;
    }
}
printf("relaxed behavior = %d\n", relaxed);
```

Résultat de l'exécution (laptop x86-64) :

Relaxed behavior = 1561

Quand on observe le comportement « relâché », ce qui se passe est décrit par le diagramme suivant :



Il y a violation de SC. On le voit à cause du cycle  $W_0(x)1 \xrightarrow{po} R_0(y)0 \xrightarrow{fr} W_1(y)1 \xrightarrow{po} R_1(x)0 \xrightarrow{fr} W_0(x)1$ .



Un examen du code assembleur produit par gcc 8.3.0 démontre que le compilateur s'est permis d'entrelacer les lectures et les écritures.

```
T_0:
    movq    $0, 8(%rbx)    ; x = 0
```

```

    movq    $0, (%rbx)           ; y = 0
    call   GOMP_barrier@PLT
    movq   8(%rbx), %rax        ; %rax = y
    movq   $1, (%rbx)          ; x = 1
    movq   %rax, 16(%rbx)       ; w = %rax
    orq    24(%rbx), %rax       ; if (w != 0 || z != 0) then ...
    jne    skip                 ; ... goto skip
    addl   $1, 32(%rbx)         ; n++

skip:
    subl   $1, %ebp             ; i--
    jne    T_0                  ; if (i != 0) goto T_0

T_1:
    call   GOMP_barrier@PLT
    movq   (%rbx), %rax         ; %rax = x
    movq   $1, 8(%rbx)         ; y = 1
    movq   %rax, 24(%rbx)       ; z = %rax
    subl   $1, %ebp             ; i--
    jne    T_1                  ; if (i != 0) goto T1

```

On peut l'empêcher en écrivant :

```

x = 1;
asm volatile("" ::: "memory"); // ne fait rien mais prétend modifier arbitrairement la mémoire
w = y;
...
y = 1;
asm volatile("" ::: "memory"); // ne fait rien mais prétend modifier arbitrairement la mémoire
z = x;

```

Et on obtient alors le code assembleur attendu :

```

T_0:
...
    movq   $1, (%rbx)           ; x = 1
    movq   8(%rbx), %rax        ; %rax = y
    movq   %rax, 16(%rbx)       ; w = y
...
T_1:
...
    movq   $1, 8(%rbx)         ; y = 1
    movq   (%rbx), %rax         ; %rax = x
    movq   %rax, 24(%rbx)       ; z = x
...

```

Résultat :

Relaxed behavior = 1296

C'est bien le processeur qui est coupable, et pas le compilateur qui fait n'importe quoi.

En fait, ce comportement est « normal ». Les manuels des processeurs x86 le documentent et expliquent que chaque thread matériel possède une *file* d'écritures vers la mémoire (le « *Store Buffer* »). L'intérêt, c'est qu'un thread matériel n'a pas besoin d'attendre que l'écriture soit complète pour commencer à exécuter les instructions suivantes. Les écritures qui sont dans la file sont effectuées dans l'ordre.

Pour lire le contenu de la mémoire à une adresse  $x$ , un thread matériel examine d'abord sa file d'écriture : si une écriture en attente vers l'adresse  $x$ , alors la lecture renvoie la dernière valeur qui devrait être écrite en  $x$ . Sinon, on va lire la mémoire à l'adresse  $x$  (cf. fig. 7.2).

Voici comment le *Store Buffering* explique que le processeur ne soit pas séquentiellement consistant, comme mis en évidence par l'expérience précédente : le thread  $T_0$  fait  $x = 1$ , c'est-à-dire qu'il enfile l'écriture de  $x$  dans sa file d'écriture. Il lit ensuite la valeur de  $y$  en mémoire (qui est alors 0,  $T_1$  est en retard). Puis, plus tard, les écritures en attente sont finalement exécutées, donc les autres thread percevront  $x = 1$ ... mais entre temps  $T_1$ , qui s'est réveillé, a eu le temps de lire  $x = 0$  en mémoire.

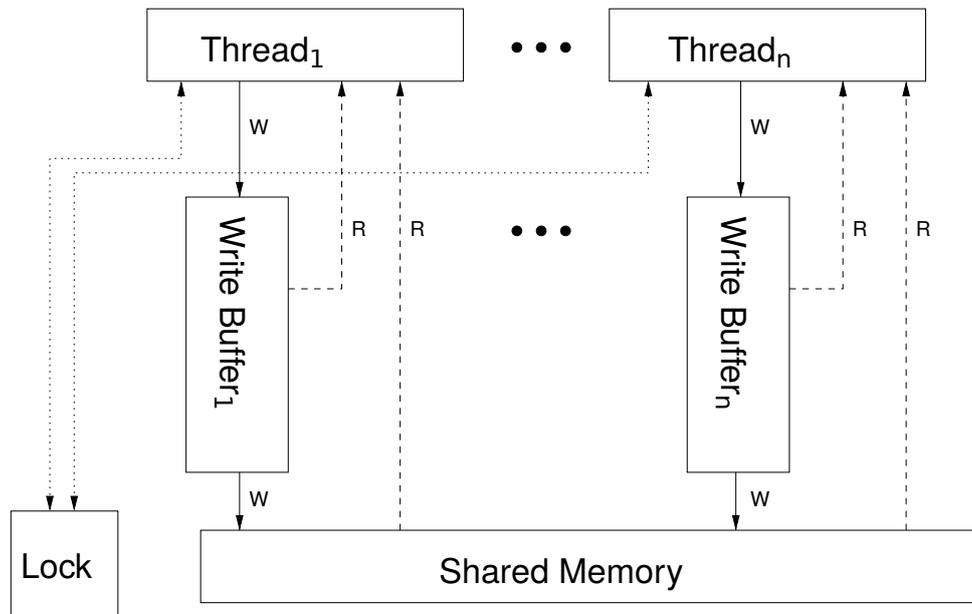


FIGURE 7.2 – Le fonctionnement des architectures TSO (image : A Tutorial Introduction to the ARM and POWER Relaxed Memory Models)

Les machines qui implémentent cette architecture possèdent la propriété de *Total Store Ordering* (TSO). Dans les processeurs TSO, un thread perçoit ses propres écritures *avant* qu'elles ne soient visibles par les autres threads. Par contre, les écritures d'un thread sont perçues par tous les *autres* threads simultanément. Ce n'est pas aussi bien que la *sequential consistency* mais ce n'est pas si mal. Les processeurs SPARC ont aussi cette caractéristique.

#### 7.4.4 Peut-on synchroniser des threads ? Le « Message Passing »

Essayons de synchroniser et de faire communiquer deux threads sans utiliser de mécanisme spécial. Le thread  $T_0$  effectue un calcul ; une fois que le résultat est prêt,  $T_0$  informe  $T_1$  qui récupère la valeur en question. Pour cela, un *flag* qui vaut initialement 0 passe à 1 lorsque les données sont prêtes.  $T_1$  attend activement que ce *flag* passe à 1 puis récupère les données.

Le test est répété plusieurs fois, avec des flags et des données différentes, prises dans un ordre pseudo-aléatoire.

```

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main()
{
    int relaxed = 0;
    int ntest = 1000000;
    volatile int *flag = malloc(ntest * sizeof(*flag));
    int *data = malloc(ntest * sizeof(*data));
    int *shuffle = malloc(ntest * sizeof(*shuffle));

    // setup
    for (int i = 0; i < ntest; i++) {
        flag[i] = data[i] = 0;
        // random-looking permutation of 0, ..., ntest-1
        shuffle[i] = (3 * 7 * i + 0xCAFE) % ntest;
    }

    #pragma omp parallel num_threads(2)
    {
        int i = omp_get_thread_num();
        for (int j = 0; j < ntest; j++) {

```

```

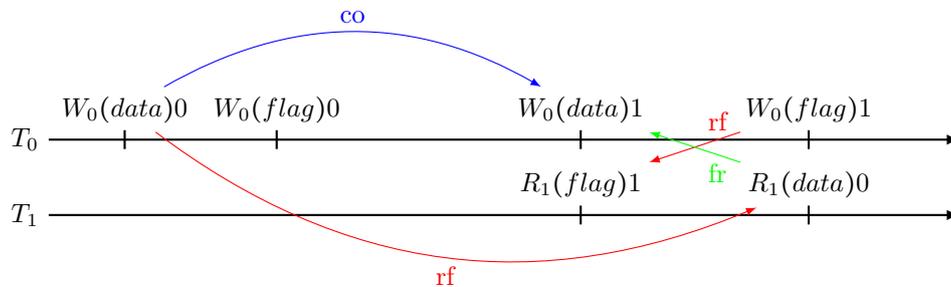
int k = shuffle[j];
if (i == 0) {
    data[k] = 1;           // write data
    flag[k] = 1;         // flag "data is ready"
} else {
    while (flag[k] == 0) {} // busy-wait for flag
    if (data[k] == 0)      // read data
        relaxed++;
}
}
printf("Relaxed behavior = %d\n", relaxed);
}

```

La bonne nouvelle, c'est que ceci se comporte correctement sur les architectures TSO. En effet, comme les écritures sont mis dans une *file* et qu'elles ne sont pas réordonnées, alors  $T_1$  ne peut pas percevoir `flag[k] == 1` sans percevoir aussi `data[k] == 1`. La mauvaise nouvelle, c'est que toutes les architectures ne sont pas TSO : les ARM et les POWER ne le sont pas ! Et ça ne loupe pas : sur un Raspberry Pi 3B+ on obtient :

Relaxed behavior = 964

Voici ce qu'on observe :



Le défaut de SC est démontré par le cycle  $W_0(data)1 \xrightarrow{po} W_0(flag)1 \xrightarrow{rf} R_1(flag)1 \xrightarrow{po} R_1(data)0 \xrightarrow{fr} W_0(data)1$ .

Comment expliquer ceci ? Les écritures effectuées par  $T_0$  ont lieu à des adresses différentes, et donc le processeur pourrait les réordonner et les effectuer dans le désordre ; et/ou elles pourraient être « propagées » à  $T_1$  dans le désordre ; et/ou les lectures effectuées par  $T_1$  pourraient être réordonnées et réalisées dans l'ordre opposé à celui spécifié par le programme. Quoi qu'il en soit, on observe qu'il est donc possible de percevoir certaines écritures tout en ne percevant pas d'autres écritures qui étaient censées avoir lieu *avant*.

### 7.4.5 Les threads peuvent percevoir des ordres différents sur les écritures

Mais en fait, les choses peuvent être encore pire. Alors que c'est impossible dans architectures TSO, sur les ARM et les POWER, des threads peuvent percevoir les écritures dans des ordres différents les uns des autres. Le test suivant le met en évidence. Il consiste essentiellement à faire deux fois en parallèle le test qui montre le *Store Buffering*.

```

#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

inline static void fast_barrier(int tid, int master, int volatile *b)
{
    if (tid == master)
        *b = 1 ;
    else
        while (*b == 0) {}
}

```

```

int main()
{
    int relaxed = 0;
    int ntest = 1000000;
    volatile int *x = malloc(ntest * sizeof(*x));
    volatile int *y = malloc(ntest * sizeof(*x));
    volatile int a, b, c, d;
    int *shuffle = malloc(ntest * sizeof(*shuffle));
    int *barrier = malloc(ntest * sizeof(*barrier));

    // setup
    for (int i = 0; i < ntest; i++) {
        barrier[i] = x[i] = y[i] = 0;
        // random-looking permutation of 0, ..., ntest-1
        shuffle[i] = (3 * 7 * i + 0xCAFE) % ntest;
    }

    #pragma omp parallel num_threads(4)
    {
        int i = omp_get_thread_num();
        for (int j = 0; j < ntest; j++) {
            int k = shuffle[j];

            // not a real barrier, but omp is too slow for this...
            fast_barrier(i, j % 4, &barrier[k]);

            if (i == 0) {
                x[k] = 1;
            } else if (i == 1) {
                a = x[k];
                b = y[k];
            } else if (i == 2) {
                y[k] = 1;
            } else {
                c = y[k];
                d = x[k];
            }

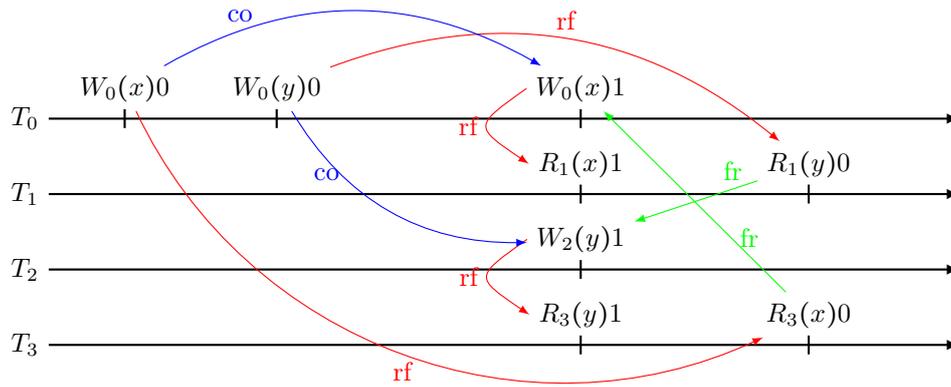
            #pragma omp barrier

            #pragma omp master
            if (a == 1 && b == 0 && c == 1 && d == 0)
                relaxed++;
        }
    }
    printf("Relaxed behavior = %d\n", relaxed);
}

```

Toujours sur un Raspberry Pi 3B+, on parvient à obtenir  $\text{relaxed} > 0$ . Ceci signifie que  $T_1$  perçoit l'écriture sur  $x$  réalisée par  $T_0$  mais pas l'écriture sur  $y$  de  $T_2$ . Du côté du thread  $T_3$ , c'est l'inverse : il perçoit l'écriture sur  $y$  de  $T_2$ , mais pas l'écriture sur  $x$  de  $T_0$ ...

Voici ce qu'on observe :



### 7.4.6 Memory Fences

Pour permettre que l'ordre puisse régner dans ce chaos (et permettre par exemple le fonctionnement de mécanismes d'exclusions mutuelles), les processeurs possèdent des instructions de synchronisation. Par exemple, les processeurs x86-64 en ont trois :

**mfence** (*Full Memory Fence*) Garantit que tous les accès à la mémoire qui précèdent l'instruction **mfence** dans l'ordre imposé par le programme sont terminées (et globalement visibles) avant qu'un accès mémoire qui suit **mfence** (dans l'ordre imposé par le programme) ne puisse avoir lieu. Latence  $\approx 35 - 40$  cycles.

**lfence** (*Load-Load Fence*) Garantit que toutes les lectures de la mémoire qui précèdent **lfence** sont terminées avant qu'une lecture qui suit **lfence** ne puisse avoir lieu. En particulier, une lecture avant ne peut pas renvoyer une valeur « plus ancienne » qu'une lecture après. Latence  $\approx 4 - 6$  cycles.

**sfence** (*Store-Store Fence*) Garantit que toutes les écritures dans la mémoire qui précèdent **sfence** sont terminées (et globalement visibles par les autres threads) avant qu'une écriture qui suit **sfence** ne puisse avoir lieu. Latence  $\approx 5 - 7$  cycles.

Les autres architectures ont des instructions comparables, avec encore plus de variantes : il y a des versions *Load-Store*, *Store-Load*, les processeurs POWER ont aussi une « Store-Load Light fence » qui garantit que les écritures qui précèdent sont bien visibles lorsque les lectures suivantes *du thread courant* auront lieu, mais les écritures antérieures peuvent se propager aux autres threads plus tard...

Insérer des *Full Memory Fences* entre tous les accès mémoire restaure la *sequential consistency*, mais ralentit considérablement l'exécution. Savoir où il faut en mettre exactement, et de laquelle on peut se contenter (la lente, complète, ou les rapides, partielles?) est un art délicat. Mieux vaut laisser le compilateur ou OpenMP se débrouiller (cf. section 7.5).