

# Modèles et techniques en programmation parallèle hybride et multi-cœurs

## Introduction au parallélisme multithreads

Marc Tajchman

CEA - DEN/DM2S/STMF/LMES

10/08/2020

# Parallélisme multi-threads en mémoire partagée

- ▶ Le but du parallélisme multithreads est de découper l'ensemble des instructions en plusieurs parties et d'exécuter (le plus possible) simultanément ces différentes parties par des threads (exécutions) sur des cœurs différents.
- ▶ On appellera la version du code non parallélisé : "séquentiel".
- ▶ Dans le cas le plus simple, le nombre de threads est égal au nombre de cœurs. Mais on peut utiliser un nombre de threads différent du nombre de cœurs.
- ▶ En mémoire partagée signifie que différents groupes d'instructions travaillent sur des données contenues dans la même mémoire. Il faut donc faire attention que les modifications faites par certaines instructions ne perturbent pas les données utilisées par d'autres instructions.

Formellement :

*On veut exécuter en parallèle,  $N$  instructions*

$$I = \{I_i(u), i = 1..N\}$$

*où  $u$  est une structure de données commune.*

*On regroupe l'ensemble des instructions en  $n_G$  groupes*

$$G_k = \{I_{ij}(u), \quad i_j \in (1..N), \quad j = 1, \dots, N_k\} \quad k = 1..n_G$$

*de telle sorte que*

$$G_{k'} \cap G_{k''} = \emptyset \quad \text{si } k' \neq k''$$

$$\bigcup_{k=1}^{k=n_G} G_k = I$$

Autrement dit:

*Chaque instruction doit être exécutée une et une seule fois*

- ▶ Autant que possible, il faut pouvoir exécuter les différents groupes d'instructions de façon indépendante (c'est le système qui décidera de démarrer un groupe, en fonction des cœurs disponibles)
- ▶ A l'intérieur d'un groupe les instructions sont exécutées séquentiellement
- ▶ Il faut déterminer quelles sont les données partagées entre les groupes et celles qui sont privées (utilisées par un seul groupe).

Ce dernier point est souvent le plus complexe.

## Exemple 1:

on peut découper une boucle:

```
for (i=0; i<N; i++)  
    v[i] = f(a, u[i]);
```

en 3 parties (par exemple), avec  
 $0 = N_0 \leq N_1 \leq N_2 \leq N_3 = N$ :

```
for (i=N0; i<N1; i++)  
    v[i] = f(a, u[i]);
```

1<sup>er</sup> groupe d'instructions à exécuter par le 1<sup>er</sup> thread

```
for (i=N1; i<N2; i++)  
    v[i] = f(a, u[i]);
```

2<sup>ème</sup> groupe d'instructions à exécuter par le 2<sup>ème</sup> thread

```
for (i=N2; i<N3; i++)  
    v[i] = f(a, u[i]);
```

3<sup>ème</sup> groupe d'instructions à exécuter par le 3<sup>ème</sup> thread

Il faut examiner l'utilisation de toutes les données sinon on risque de tomber sur des erreurs difficiles à corriger.

Donnée	Comportement
N0, N1, N2, N3, a	Utilisées par plusieurs threads mais constantes dans l'algorithme, on peut les partager entre les threads
u	u est constant et donc peut être partagé
v	v varie dans l'algorithme, mais comme chaque thread modifie une partie différente du vecteur, v peut être partagé.
i	i prend des valeurs différentes dans les threads (i = N0 à N1-1 dans le premier thread, i=N1 à N2-1 dans le second thread, etc.), il faut donc utiliser des variables différentes qui représentent i dans les différents threads. i est dit "variable privée".

L'algorithme doit être modifié comme suit:

```
for (i0=N0; i0<N1; i0++)  
    v[i0] = f(a, u[i0]);
```

```
for (i1=N1; i1<N2; i1++)  
    v[i1] = f(a, u[i1]);
```

```
for (i2=N2; i2<N3; i2++)  
    v[i2] = f(a, u[i2]);
```

Voir l'implémentation en OpenMP dans l'[Exemple 1](#)

Dans le même exemple, on trouvera la façon habituelle (et beaucoup plus simple) de coder une boucle en OpenMP.





Exemple 2: si  $u$  et  $v$  sont des vecteurs de taille  $n > 4$ , on veut calculer la boucle for:

$$\begin{aligned} & \text{for}(i = 1; i < n - 1; i++) \\ & \quad v_i = (u_{i-1} + 2 * u_i + u_{i+1})/4; \end{aligned} \tag{1}$$

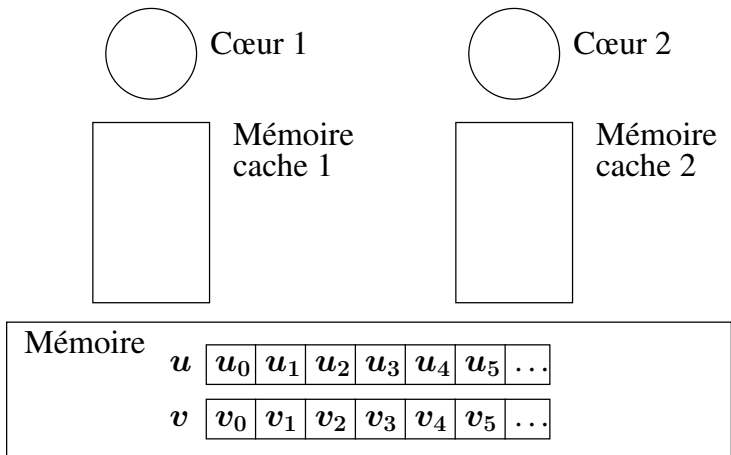
On va examiner en détail le calcul simultané sur 2 cœurs des 2 expressions

$$v_2 = (u_1 + 2u_2 + u_3)$$

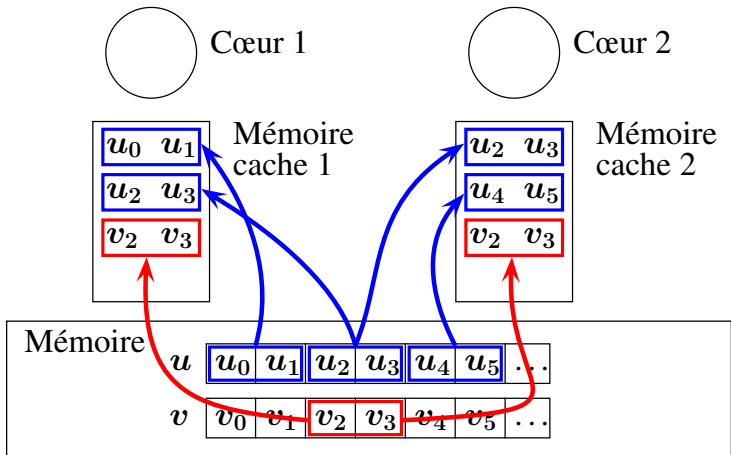
$$v_3 = (u_2 + 2u_3 + u_4)$$

On supposera que chaque cœur possède sa mémoire cache de taille 8 (4 lignes de cache de taille 2).

1. Avant d'exécuter les 2 instructions :

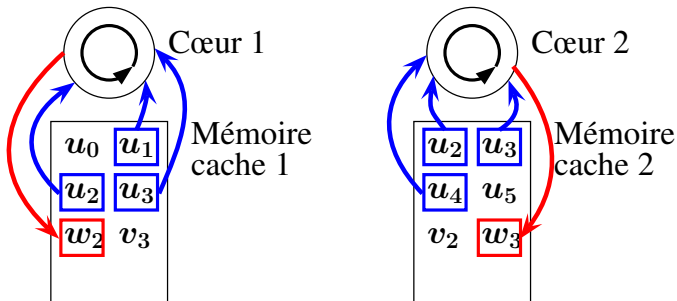


2. Les blocs qui contiennent les composantes utilisées sont copiés dans les mémoires cache :



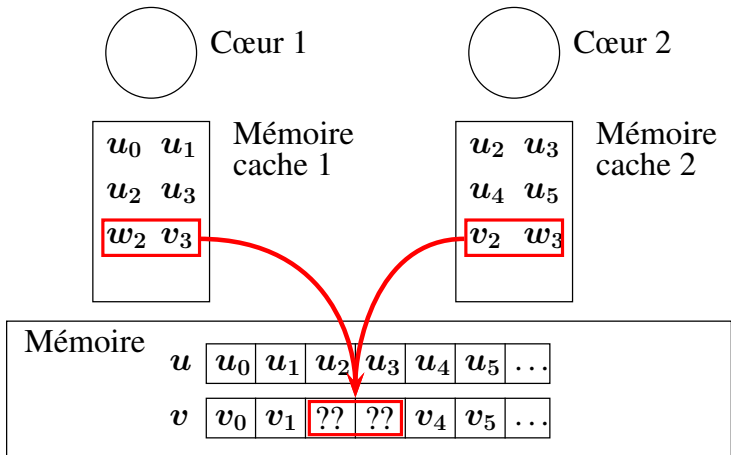
3. Les composantes de  $u$  sont recopiées dans les mémoires internes des processeurs et les résultats sont mis à la place de composantes de  $v$ :

$$w_2 = (u_1 + 2u_2 + u_3)/4 \quad w_3 = (u_2 + 2u_3 + u_4)/4$$



Mémoire							
$u$	$u_0$	$u_1$	$u_2$	$u_3$	$u_4$	$u_5$	$\dots$
$v$	$v_0$	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$	$\dots$

4. Les blocs qui contiennent les résultats sont copiés dans la mémoire centrale. **⚠ Le résultat est indéterminé.**

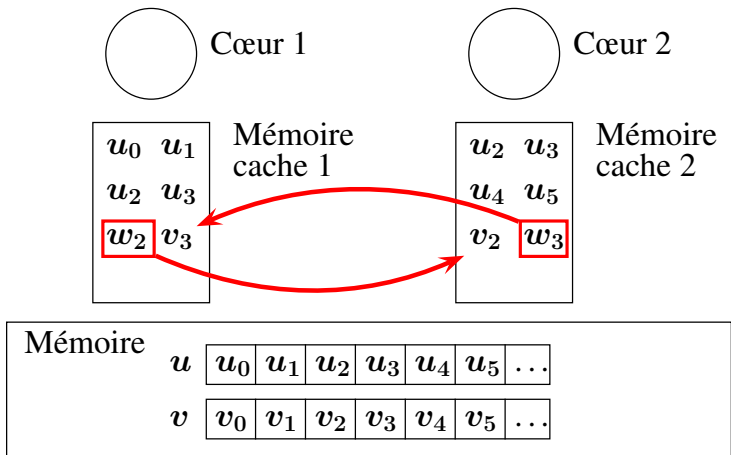


L'erreur vient du fait que 2 caches différents contiennent chacun un bloc (partiellement) différent qui seront recopiés dans la mémoire centrale au même endroit

Tous les ordinateurs dont les processeurs ont plusieurs cœurs, appliquent un algorithme pour vérifier et rétablir la **cohérence de cache** :

*Si deux lignes de cache dans deux mémoires cache correspondent au même emplacement dans la mémoire centrale, on garde les valeurs les plus récentes.*

3 bis. On rétablit la cohérence de cache :  $w_2$  dans le cache 1 est copié à la place de  $v_2$  dans le cache 2 et  $w_3$  dans le cache 2 est copié à la place de  $v_3$  dans le cache 1



Grace à l'étape de cohérence de cache les résultats seront corrects mais le maintien de la cohérence de cache prend du temps et diminue l'efficacité du parallélisme multithreads.

*On essaie de faire en sorte que les caches ne contiennent pas des copies des mêmes blocs mémoires (dans ce cas pas besoin de contrôler la cohérence des caches)*



D'où la règle supplémentaire :

Règle: les zones mémoire modifiées par deux threads différents au même instant doivent être distinctes (sinon les résultats peuvent être faux) et assez éloignées (pour diminuer le cout du maintien de la cohérence de cache)

En théorie, il faut une distance d'au moins la taille d'une ligne de cache dans le cache L1. En pratique, on

# Rappels sur OpenMP

Le principal (mais pas le seul) outil pour coder du parallélisme multithreads est OpenMP.

Pour l'exemple vu plus haut, on ajoute avant la boucle, le pragma

```
#pragma omp parallel for  
for (i=1; i<n-1; i++)  
    v[i] = (u[i-1] + 2*u[i] + u[i+1])/4;
```

## Avantages:

- ▶ un seul code source pour les versions parallèle ou séquentiel
- ▶ on peut choisir les boucles qu'on veut paralléliser ou non (parallélisation incrémentale)
- ▶ facile à coder

## Désavantages et difficultés:

- ▶ les performances sont parfois décevantes
- ▶ attention aux variables partagées par différentes itérations d'une boucle
- ▶ pas beaucoup de contrôle sur le placement des threads et sur le découpage en sous-boucles

Pour compiler du code utilisant OpenMP, on utilise une option de compilation (qui dépend du compilateur : `-fopenmp` pour `gcc/g++/gfortran`).

Pour exécuter un code utilisant plusieurs threads, il y a plusieurs possibilités:

- ▶ définir une variable d'environnement `OMP_NUM_THREADS`, par exemple :

```
OMP_NUM_THREADS=5 ./code.exe
```

- ▶ appeler dans le code source la fonction

```
omp_set_num_threads(5);
```

## Exemple OpenMP 1

## Source C++ séquentiel:

```
1  #include "calcul.hxx"
2  #include <cmath>
3  #include <iostream>
4  #include "timer.hxx"
5
6  #include <unistd.h>
7
8  double f(double a, double x)
9  {
10     usleep(100);
11     return sin(a*x);
12 }
13
14 void calcul_seq(std::vector<double> & v,
15                const std::vector<double> &
16                u)
17 {
18     size_t i, N = u.size();
19     double a = M_PI;
```

## Source C++ séquentiel:

```
1  #include "calcul.hxx"
2  #include "timer.hxx"
3  #include <iostream>
4  #include <thread>
5  #include <cmath>
6
7  // somme des composantes d'un vecteur sur un
   // intervalle
8  void somme(const std::vector<double> &v,
9           int i0, int i1, double &s)
10 {
11     int i;
12     s = 0;
13     for (i=i0; i<i1; i++)
14         s += v[i];
15 }
16
17 void somme_modifie(const std::vector<double>
```