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.

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.

Exemple 1: si `u` et `v` sont des vecteurs de taille `n > 4`, on veut calculer la boucle `for`:

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

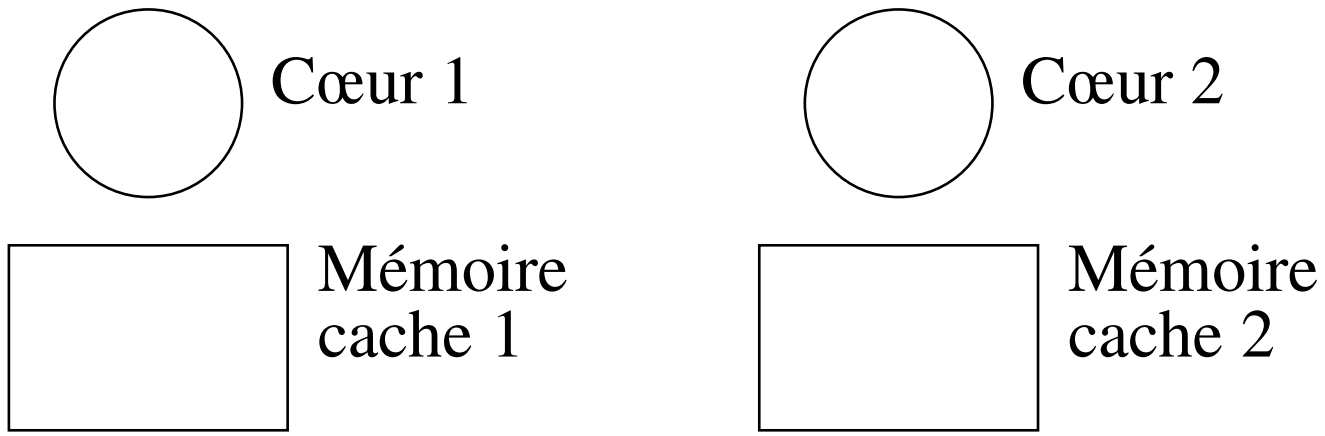
Ces instructions (pour différentes valeurs de i) utilisent des composantes de `u` différentes, mais (presque) toutes les composantes de u sont utilisées par plusieurs instructions de la boucle `for`.

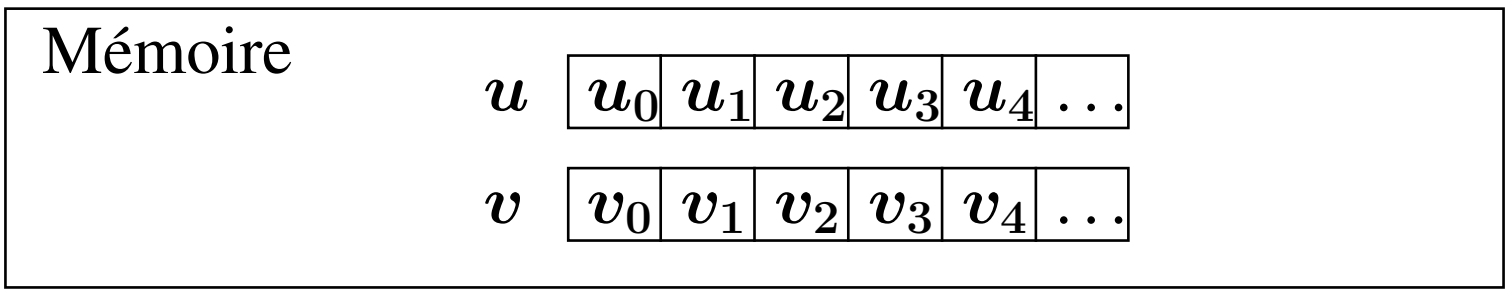
Les composantes de u ne sont pas modifiées par les instructions.

Les composantes de v sont modifiées (mais chaque instruction calcule une composante différente).

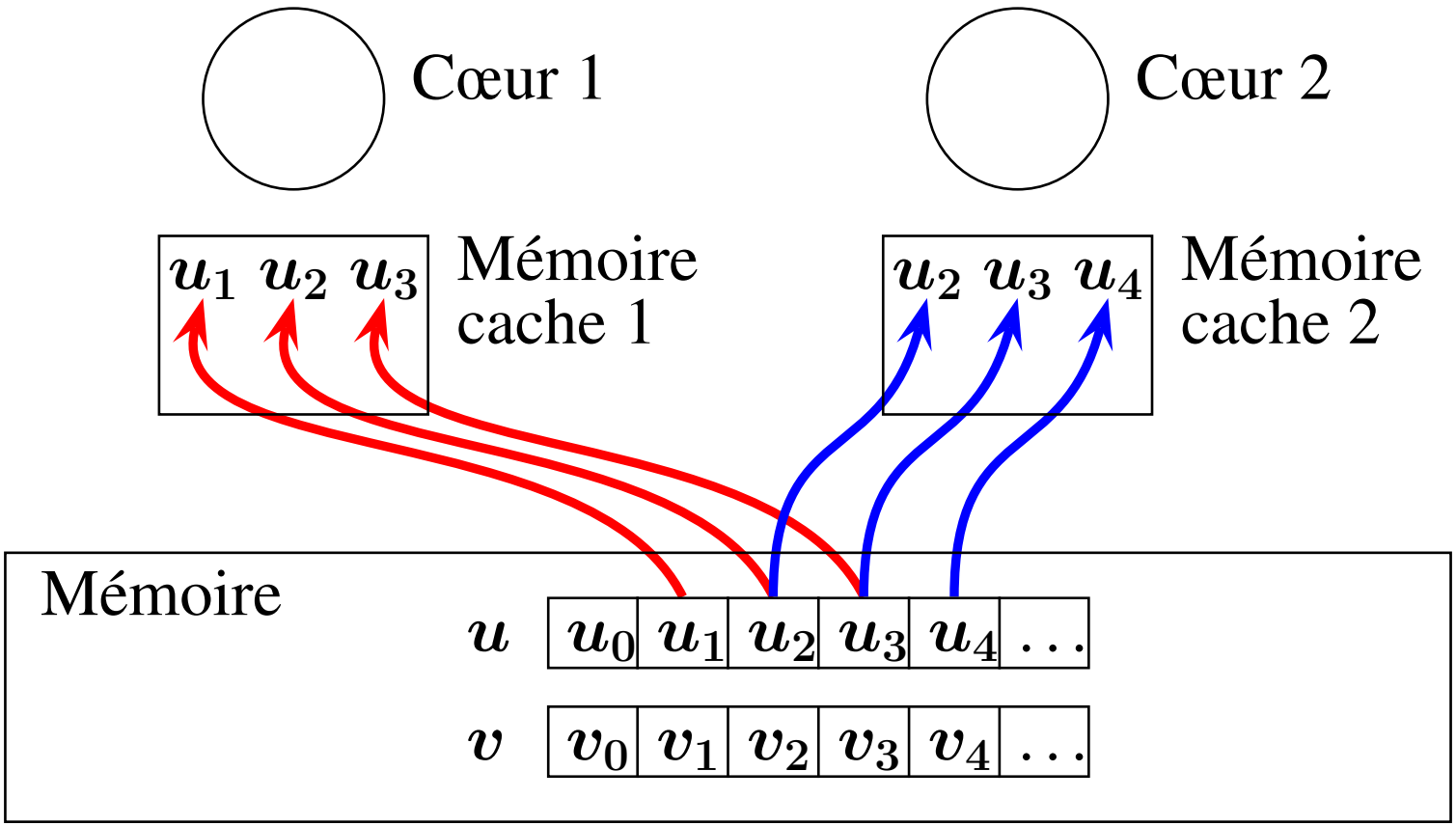
On remarque que le calcul de v_i ne dépend pas de celui de v_j ($j \neq i$) et donc ces 2 calculs peuvent se faire en même temps par 2 exécutions différentes qui utilisent les mêmes vecteurs u et v.

1. Avant d'exécuter les 2 instructions (i=2 et i=3) :

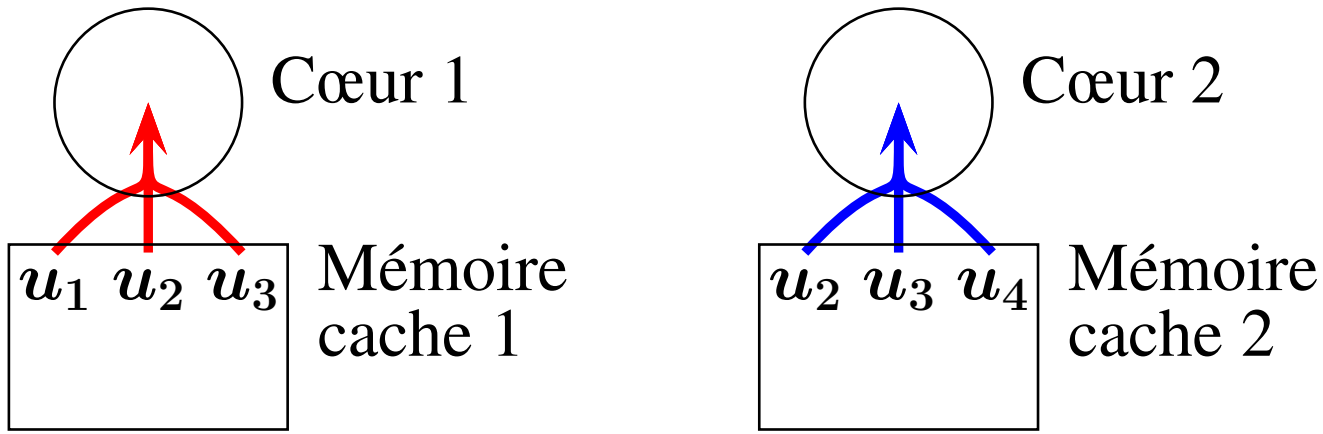


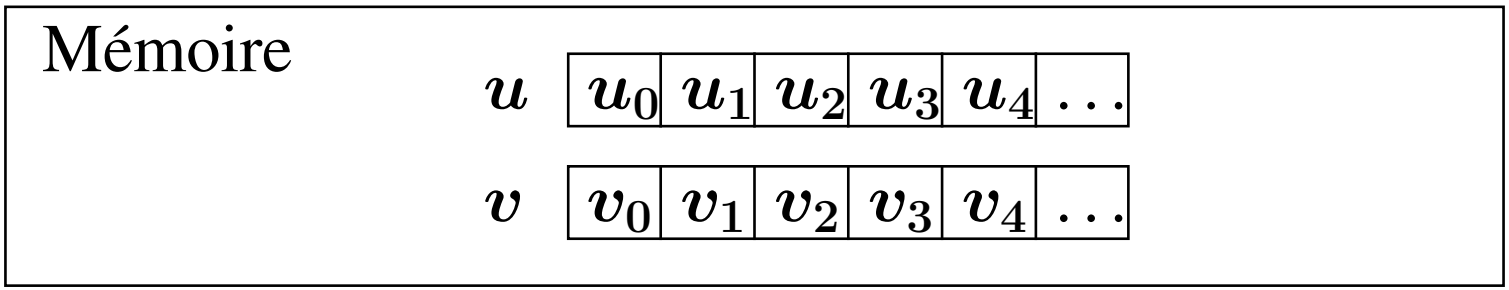


2. Les composantes de u sont recopiées dans les mémoires cache :



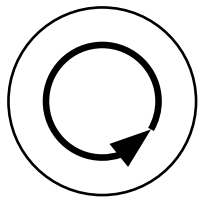
3. Les composantes de u sont recopiées dans les mémoires internes des processeurs:



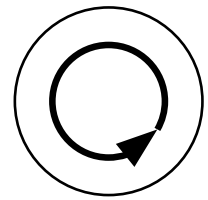


4. Le calcul est effectué dans les cœurs (2 instructions en //) :

$v_2 = (u_1 + 2u_2 + u_3)/4$ $v_3 = (u_2 + 2u_3 + u_4)/4$



Cœur 1



Cœur 2

u_1 u_2 u_3

Mémoire cache 1

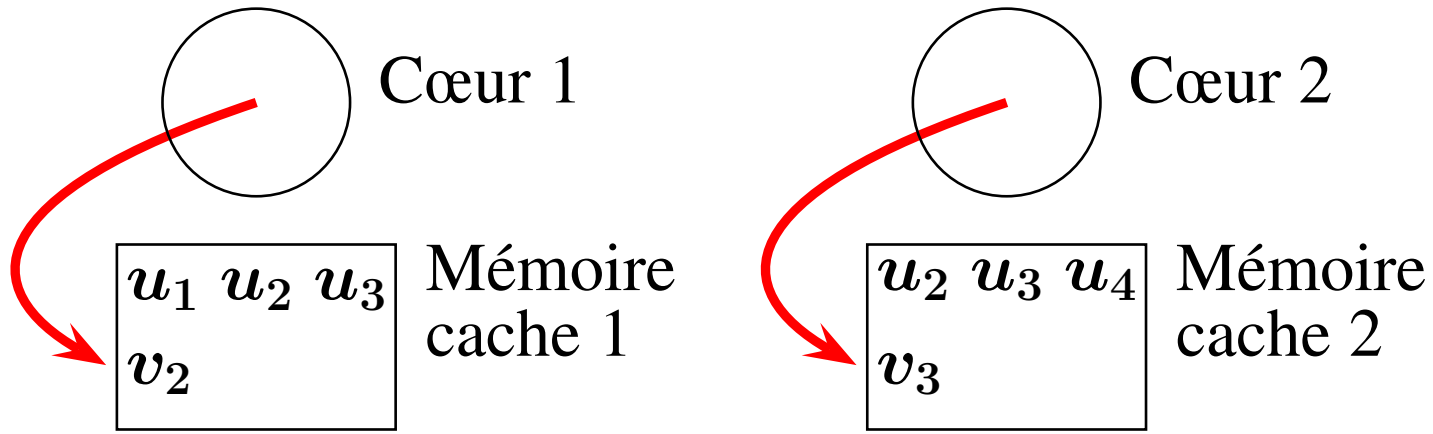
u_2 u_3 u_4

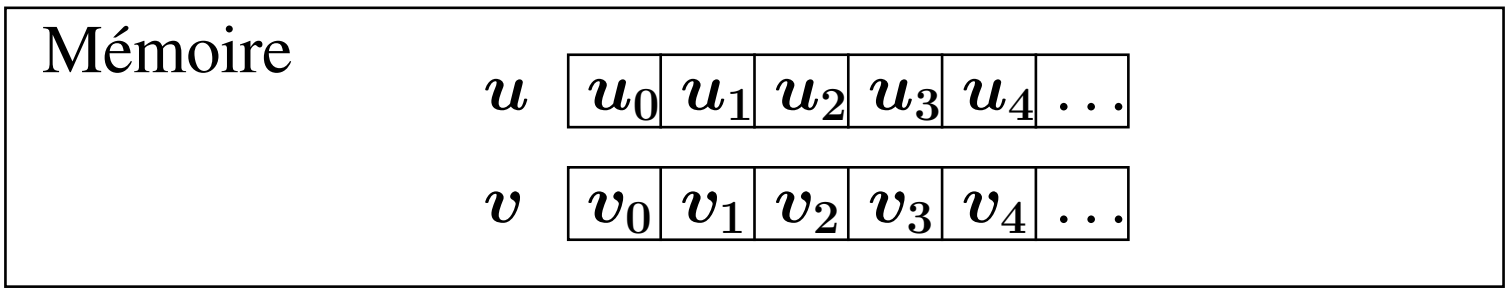
Mémoire cache 2

Mémoire

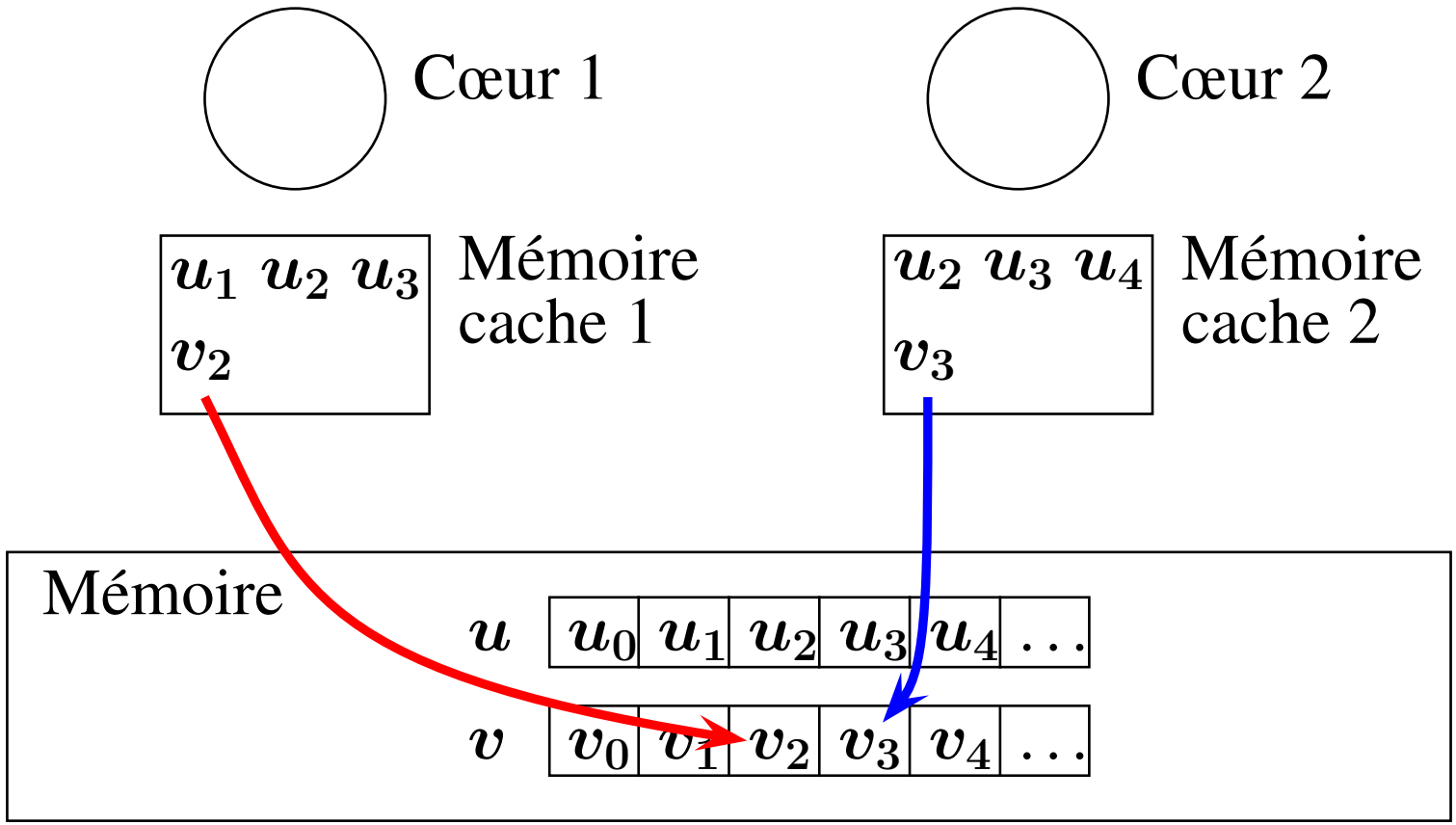
u	u_0	u_1	u_2	u_3	u_4	$\ldots$
v	v_0	v_1	v_2	v_3	v_4	$\ldots$

5. Le résultat est recopié dans la mémoire cache :





6. Le résultat est recopié dans la mémoire principale :



Le plus simple est de découper la boucle séquentielle :

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

en K sous-boucles :

```
for (i=p[k]; i<p[k+1]; i++)      // k = 0...K-1
    v[i] = (u[i-1] + 2*u[i] + u[i+1])/4;
```

avec `p[0] = 1` et `p[K] = n-1`

L'exécution de toutes les sous-boucles est identique à l'exécution de la boucle complète. Toutes les sous-boucles doivent si possible être exécutées en même temps.

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

On ajoute une ligne avant la boucle (pragma)

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

- la ligne qui commence par `#pragma` est appelée pragma
- en fonction des options de compilation, cette ligne est ignorée ou pas
- le compilateur va découper lui-même la boucle en sous-boucles en répartissant les sous-boucles entre les threads

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
3  void calcul(std::vector<double> & v, const
       std::vector<double> & u)
4  {
5    size_t i, n = u.size();
6
7    v[0] = u[0];
8
9
10   for (i = 1; i<n-1; i++)
11     v[i] = (u[i-1]+2*u[i]+u[i+1])/4;
12   v[n-1] = u[n-1];
13 }
```

Source C++ séquentiel:

```
1  #include "calcul.hxx"
2
3  void calcul(std::vector<double> & v, const
       std::vector<double> & u)
4  {
5    size_t i, n = u.size();
6
7    v[0] = u[0];
8
9    #pragma omp parallel for
10   for (i = 1; i<n-1; i++)
11     v[i] = (u[i-1]+2*u[i]+u[i+1])/4;
12   v[n-1] = u[n-1];
13 }
```