

Modèles et techniques en programmation parallèle hybride et multi-cœurs

Quelques notions d'optimisation séquentielle

Marc Tajchman

CEA - DEN/DM2S/STMF/LMES

10/08/2020

Deux “règles d’or” :

- ▶ Avant/après optimisation, avant/après parallélisation, il faut s’assurer que le code sur lequel on travaille, donne des résultats corrects.
- ▶ Avant de paralléliser un code et/ou avant d’optimiser le parallélisme, il faut optimiser la partie séquentielle d’un code.

On parle en général de version Debug et Release d'un code :

- ▶ version Debug : avec les options de compilation (-g -O0 par exemple avec gcc/g++) qui désactivent les optimisations faites par le compilateur, qui ajoutent dans le code des points de repère pour exécuter un code instruction par instruction, afficher la valeur de variables en cours d'exécution, etc.
- ▶ version Release : avec les options de compilation (-O2 ou -O3 par exemple avec gcc/g++) qui permettent au compilateur d'optimiser lui-même le code.

- Pendant l'écriture, les tests et la correction d'un code, on utilise en général une version Debug.
- A la fin, on fournit aux utilisateurs une version Release.
- Bien vérifier que la version Release donne les mêmes résultats que la version Debug (on a parfois des surprises).

Pour évaluer l'optimisation d'un code, on utilise des outils de mesure de performance. Il existe de nombreux moyens de mesurer le temps d'exécution de code ou de parties de code :

- ▶ **Commande unix `time`** : mesure globale (temps ressenti par l'utilisateur)
- ▶ **Fonctions définies par le langage de programmation** et utilisables depuis l'intérieur du code :
 - ▶ `second(...)` (fortran),
 - ▶ `gettimeofday(...)` (C/C++),
 - ▶ `std::clock()` (C++),
 - ▶ `time.time()` (python)
 - ▶ `tic/toc` (matlab),
 - ▶ ...

Permet de mesurer le temps d'exécution d'un groupe d'instructions.

Penser à vérifier dans la documentation quelle est la précision des mesures.

- **Librairies**, par exemple MPI, OpenMP, PAPI

On ajoute dans le code des appels à des fonctions de la librairie.

- MPI et OpenMP proposent des fonctions pour mesurer le temps de calcul : `MPI_Wtime()`, `omp_get_wtime()`.

- PAPI

(https://icl.cs.utk.edu/projects/papi/wiki/Main_Page)

est une librairie qui donne des informations très précises.

Permet de consulter des compteurs système très bas niveau (par exemple : nombre d'opérations, utilisation des caches, registres, etc.)

► Outils externes de “profilage”

Ajoutent automatiquement des points de mesure dans le code (gprof), s'interposent entre le code et le système pour récupérer des informations (valgrind, perf, vtune (intel), etc.)

Permet de connaître des informations intermédiaires : nombre d'appels et temps moyen d'exécution de fonctions par exemple (gprof, valgrind).

Certains sont plus précis et descendent au niveau de l'instruction (perf, vtune)

En général, il faut utiliser les outils de profilage avec une version “debug” (même s'ils peuvent fonctionner en version “release”, ils donneront alors beaucoup moins d'informations).

Programmation séquentielle efficace

Pour obtenir un code efficace (en temps d'exécution), il faut:

- ▶ **utiliser les algorithmes les plus efficaces possible (pas couvert par ce cours)**
- ▶ **organiser le placement des données (améliorer la localité spatiale)**
- ▶ **organiser la séquence d'instructions (améliorer la localité temporelle)**
- ▶ **écrire les instructions pour qu'elles soient les plus rapides possibles (éviter de calculer plusieurs fois la même expression, éviter si possible les tests (instructions 'if'), "faciliter le travail du compilateur")**

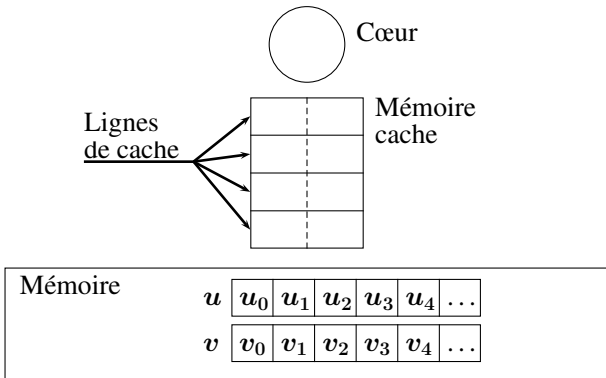
Une bonne utilisation de la mémoire est très importante pour l'optimisation de code.

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

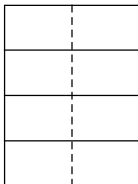
$$\begin{aligned} & \text{for}(i = 1; i < n - 1; i++) \\ & \quad v_i = (u_{i-1} + 2 * u_i + u_{i+1})/4; \end{aligned} \tag{1}$$

Supposons un système idéalisé par:

- ▶ un processeur,
- ▶ une mémoire cache de taille 8 (4 lignes cache de taille 2),
- ▶ la mémoire centrale

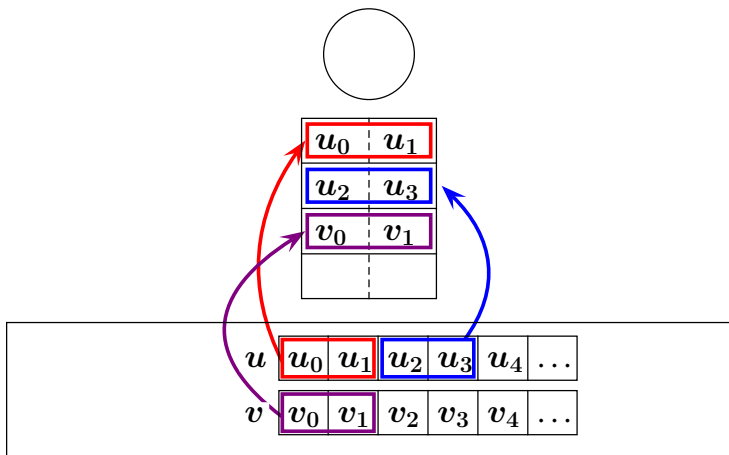


1. Avant d'exécuter $v_1 = (u_0 + 2 * u_1 + u_2)/4$:

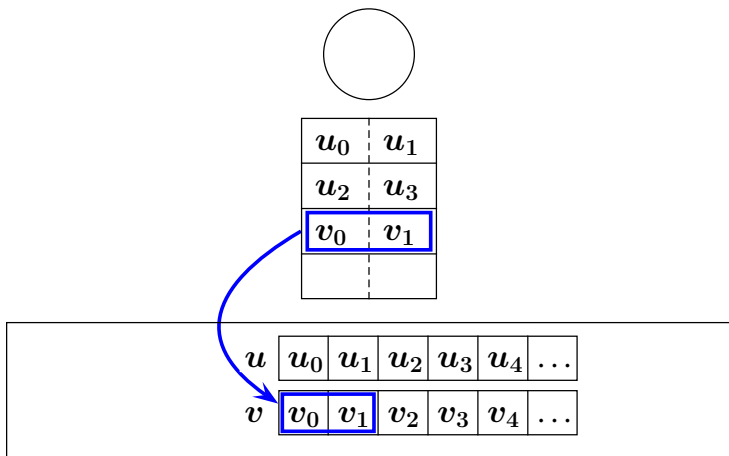


u	u_0	u_1	u_2	u_3	u_4	\dots
v	v_0	v_1	v_2	v_3	v_4	\dots

2. Les blocs contenant u_0 , u_1 , u_2 et v_1 (3 blocs) sont copiés dans la mémoire cache :



4. Le bloc contenant le résultat est recopié dans la mémoire principale :



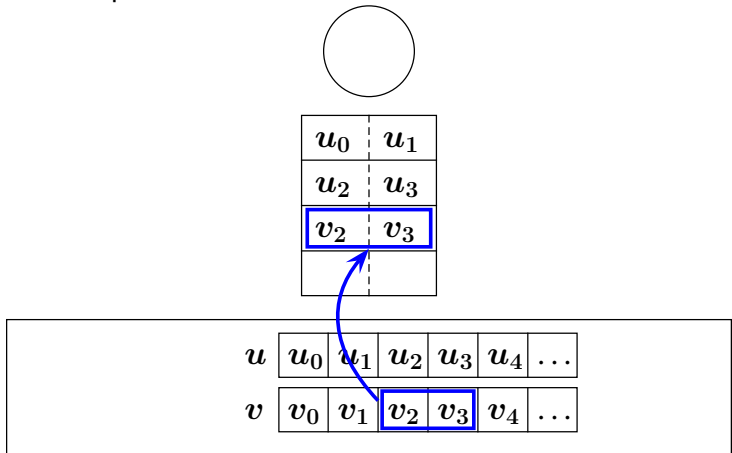
5. Le calcul de l'instruction suivante $v_2 = (u_1 + 2 * u_2 + u_3) / 4$ peut commencer:



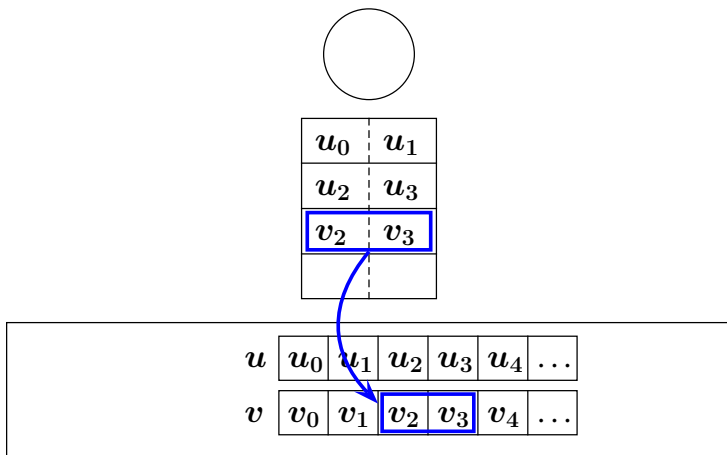
u_0	u_1
u_2	u_3
v_0	v_1

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

6. Les composantes de u nécessaires **sont déjà dans la mémoire cache**, seul le bloc contenant la composante v_2 doit être copié dans la mémoire cache :



6. Le bloc contenant le résultat est recopié dans la mémoire principale :



En résumé :

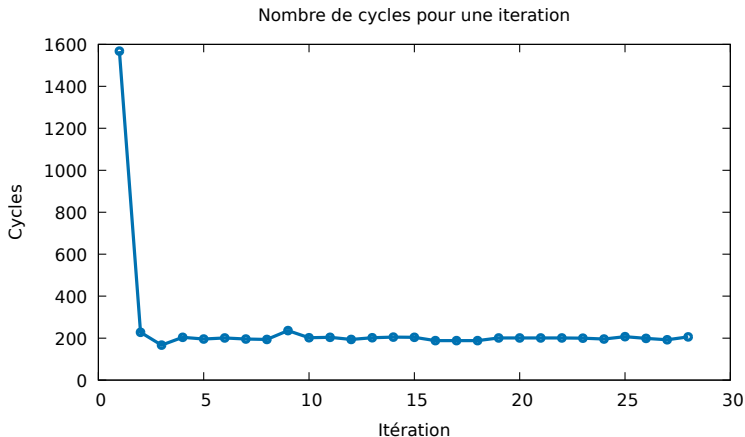
- ▶ La première instruction $v_1 = (u_0 + 2 * u_1 + u_2)/4$ utilise
 - ▶ 4 transferts (lents) mémoire principale - mémoire cache
 - ▶ 4 transferts (rapides) mémoire cache - cœur
- ▶ La deuxième instruction $v_2 = (u_1 + 2 * u_2 + u_3)/4$ utilise
 - ▶ 2 transferts (lents) mémoire principale - mémoire cache
 - ▶ 4 transferts (rapides) mémoire cache - cœur

Attention : cet exemple est (très) simplifié : en général la mémoire cache est de taille plus grande que dans l'exemple et, de plus, il y a plusieurs types de mémoire cache dans un ordinateur

Examiner et tester l'exemple *Exemple1*

- ▶ Sans mémoire cache, toutes les itérations prendraient (à peu près) le même temps.
- ▶ La première instruction prend plus de temps parce qu'elle remplit la mémoire cache.
- ▶ Beaucoup d'instructions suivantes n'auront pas besoin de chercher des valeurs dans la mémoire centrale.

La mémoire cache améliore beaucoup le temps de calcul de presque toutes les itérations



Un cycle est l'unité de temps pour le processeur. Le processeur qui a été utilisé ici, a une fréquence de 3 GHz. Un cycle prend à peu près $0.3 \cdot 10^{-9}$ secondes.

Sur le graphique, une itération (sauf la première) prend $0.6 \cdot 10^{-7}$ secondes.

On voit donc qu'il est important de bien utiliser la mémoire cache.

Deux règles sont déduites de ce comportement : optimiser les localités spatiale et temporelle.

Localité spatiale

Règle: autant que possible, utiliser des zones mémoires proches les unes des autres dans une séquence d'instructions

Le transfert entre mémoire centrale et mémoire cache se fait par bloc de données.

Donc, si une donnée est à côté d'une donnée qui vient d'être utilisée (et donc transférée en mémoire cache), un nouveau transfert sera peut-être inutile.

Voir l'[Exemple2](#)

Localité temporelle

Règle: autant que possible, pour une zone mémoire, les instructions qui l'utilisent doivent s'exécuter de façon rapprochée dans le temps

La mémoire cache étant de petite taille, le gestionnaire mémoire, s'il a besoin de place, effacera dans la mémoire cache les données les plus anciennes.

Si une donnée est utilisée par plusieurs instructions proches dans le temps, elle sera maintenue plus longtemps en mémoire cache (et donc nécessitera moins de transferts)

Voir l'[Exemple3](#)

Utilisation du // interne au processeur

Règles:

- ▶ essayer de rassembler plusieurs instructions simples en une seule (quand cela a un sens),
utiliser au maximum la présence de plusieurs unités de calcul (additionneurs, multiplicateurs, etc) dans un cœur
- ▶ essayer d'éviter les tests
simplifier le travail du processeur (enchaînement des instructions le plus déterministe possible).

Exemple: remplaceur

```
for (i=0; i<n; i++) {  
    u[i] = u0[i];  
    if (terme1) u[i] += a*v[i];  
    if (terme2) u[i] += b*w[i];  
}
```

par:

```
if (terme1) aa = a; else aa = 0.0;  
if (terme2) bb = b; else bb = 0.0;  
for (i=0; i<n; i++) {  
    u[i] = u0[i] + aa*v[i] + bb*w[i];  
}
```