



## TD1 - OS202

### PROGRAMMATION PARALLÈLE

André COSTA WERNECK  
Leonardo GRECO PICOLI

#### Produit matrice-matrice

##### Exercise 1

En exécutant le fichier **TestProduitMatrix.exe** pour les différentes dimensions, on obtient les résultats suivants:

Dimension	MFlops	Temps CPU
1023 x 1023	2013.25	1.06355 secondes
1024 x 1024	815.027	2.63486 secondes
1025 x 1025	2004.96	1.07423 secondes
2048 x 2048	391.841	43.844 secondes
2060 x 2060	1486.43	11.7622 secondes

On peut voir que quand les dimensions des matrices ont des valeurs carrées, les temps d'exécution sont beaucoup plus larges par rapport aux autres dimensions, même si les dimensions sont plus grandes.

Cela est dû à l'effet de la "localité temporelle et spatiale".

Lorsque nous effectuons une multiplication matricielle, nous accédons élément par élément aux matrices A et B de différentes manières. L'accès à chaque élément entraîne la sauvegarde d'un morceau de mémoire dans les lignes du cache.

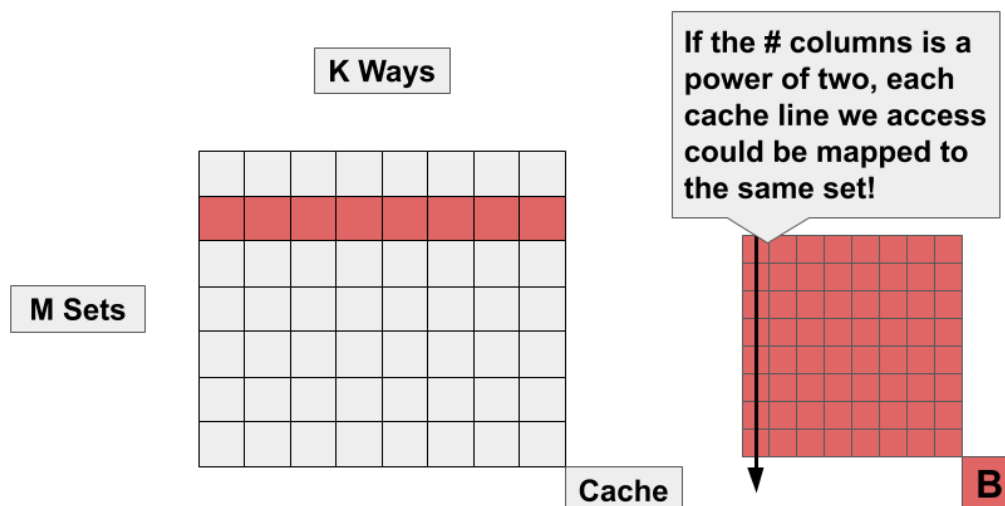
Par exemple, lorsque nous accédons au premier élément  $a_{11}$  de la première matrice A, les éléments consécutifs en mémoire ( $a_{12}$ ,  $a_{13}$ ,  $a_{14}$ , etc.) sont déjà enregistrés dans les lignes de cache pour garantir la "localité spatiale", de sorte que lorsque nous accédons à l'élément de processus ( $a_{12}$ ) de la matrice A, nous profitons de ce qui a été stocké dans le cache.

Dans la multiplication matricielle classique (naïf), les boucles sont imbriquées de sorte qu'à chaque interaction, on accède aux éléments voisins en mémoire dans la matrice A ( $a_{11}$ ,  $a_{12}$ ,  $a_{13}$ , etc.).

Cependant, ce n'est pas le cas pour la matrice B, car les éléments sont accédés de la manière suivante : ( $b_{11}$ ,  $b_{21}$ ,  $b_{31}$ ,  $b_{41}$ , etc.), qui sont des éléments non voisins en mémoire. Cela signifie que pour chaque accès que nous faisons à notre matrice B est séparé par  $2^N$  éléments.

Un modèle d'accès avec ce type d'intervalle peut entraîner de nombreux manques de conflit, car chaque ligne de cache sera mappée sur un petit sous-ensemble de jeux de caches (ou même un seul jeu de caches).

L'image ci-dessous est utilisée comme exemple pour aider à la compréhension de l'explication.

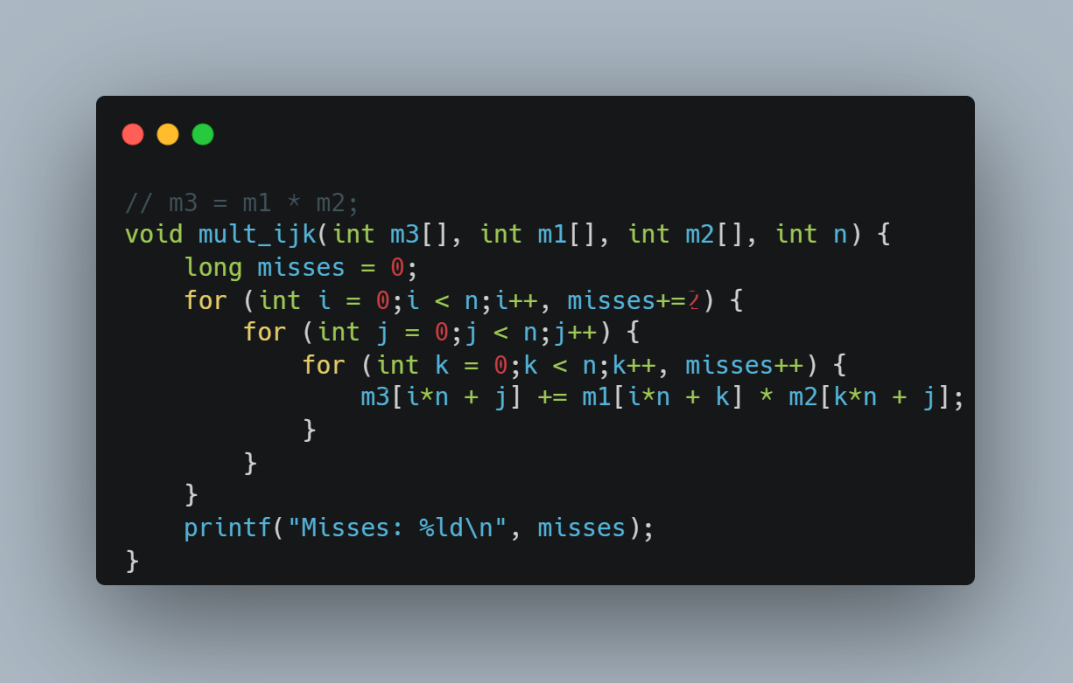


Reference (<https://coffeebeforearch.github.io/2020/06/23/mmul.html>)

Chaque ensemble a un nombre limité de voies (généralement 4 ou 8). Bien que la taille de notre cache (en octets) puisse être relativement grande par rapport à la taille de nos matrices, nous ne pouvons pas utiliser efficacement cette capacité si chaque ligne de cache se bat pour les mêmes 4 ou 8 voies dans un seul ensemble.

## Exercise 2

En exécutant le code test.c ci-dessous, il est devenu évident que, même de manière contre-intuitive, permuter les lignes des boucles en i,j,k fait vraiment une grande différence pour le temps d'exécution du programme. Ainsi, après avoir étudié le fonctionnement de la mémoire cache, nous avons analysé le code et réalisé sur papier une multiplication d'une petite matrice pour comprendre ce qui se passe dans ce cas.



```
// m3 = m1 * m2;
void mult_ijk(int m3[], int m1[], int m2[], int n) {
    long misses = 0;
    for (int i = 0; i < n; i++, misses+=2) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++, misses++) {
                m3[i*n + j] += m1[i*n + k] * m2[k*n + j];
            }
        }
    }
    printf("Misses: %ld\n", misses);
}
```

Ceci étant fait, nous pouvons remarquer que la grande différence est due au fait que le cache est une mémoire associative qui va chercher les données dans la mémoire RAM en blocs de taille spécifique et, en outre, il les enregistre également en suivant une orientation spécifique dans la mémoire et en respectant la politique de voisinage, c'est-à-dire que le bloc reçoit les données requises et également les données les plus proches de lui jusqu'à un certain point. Dans le cas de notre programme C, par exemple, pour effectuer la multiplication des matrices m1 et m2, lorsque nous sauvegardons un élément de RAM dans le Cache, nous sauvegardons aussi d'autres éléments de la même ligne des deux matrices. Cependant, dans un produit matriciel, la multiplication se fait entre une ligne d'une des matrices et une colonne de l'autre. Ainsi, pour la matrice dont la ligne sera multipliée, le programme est efficace, mais pour la matrice dont la colonne sera multipliée, le programme est beaucoup plus lent, car les données sauvegardées dans le cache ne sont que les lignes de la matrice. Par conséquent, chaque multiplication de colonne représentera plusieurs caches miss, car chaque fois que nous changeons une ligne (une colonne est constituée de N lignes), nous devons chercher à nouveau des données dans la RAM que nous n'utilisons pas, ce qui représente un cache miss important, comme dit précédemment.

On en conclut donc que le meilleur cas est celui dans lequel la multiplication est effectuée avec le moins de changements de lignes possible dans les matrices. Il est représenté dans le code ci-dessous :

```

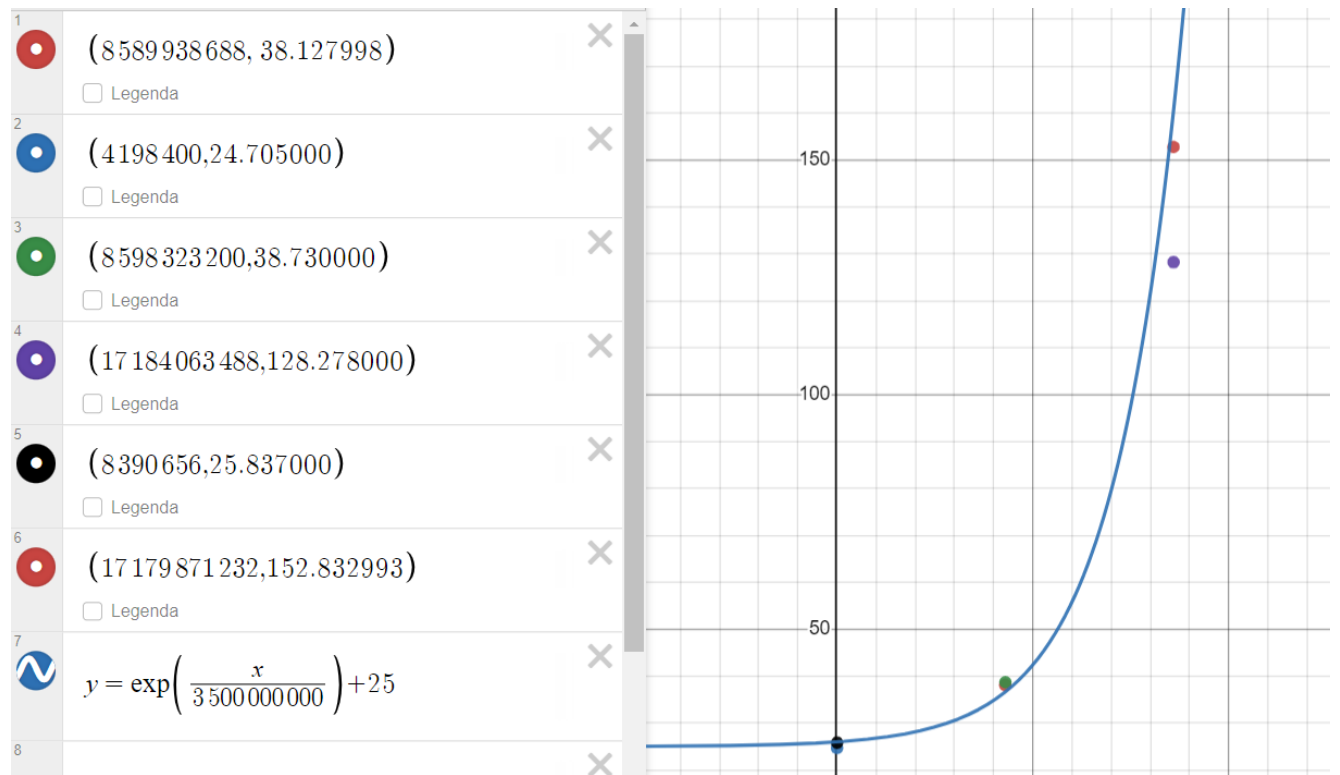
void mult_ikj(int m3[], int m1[], int m2[], int n) {
    long misses = 0;
    for (int i = 0; i < n; i++, misses+=2) {
        for (int k = 0; k < n; k++, misses++) {
            for (int j = 0; j < n; j++) {
                m3[i*n + j] += m1[i*n + k] * m2[k*n + j];
            }
        }
    }
    printf("Misses: %ld\n", misses);
}

```

Afin d'évaluer quantitativement l'effet de la permutation des boucles, nous avons créé la métrique *misses*, représentée simplement par une variable qui était incrémentée à chaque changement de ligne dans le calcul. Même s'il s'agit d'une mesure approximative, les résultats obtenus illustrent très bien ce qui a été observé. Pour matrices de dimensions 2048x2048, ils suivent ci-dessous :

	IJK	IKJ	JKI	JKI	KIJ	KJI
Misses	8.5899E9	4.1984E6	8.5983E9	17.1841E9	8.3906E6	17.179E9
Diff (s)	38,1	24,7	38,7	128,3	25,8	152,8

La variable *diff* représente le temps d'exécution et, pour observer encore mieux la relation entre le nombre *misses* et le temps d'exécution, le graphique ci-dessous a été dessiné :



Ainsi, il est devenu évident que la relation entre les variables est exponentielle, ce qui est très logique lorsqu'on analyse la complexité des algorithmes avec les différentes permutations des boucles *for*.

### Exercise 3

Threads	Temps (seconds)	MFlops
1	52.1839	329.218
2	39.6779	432.983
4	27.8734	616.353
8	23.3562	735.56
16	23.721	724.247

Pour N qui ne sont pas puissances de 2 (par exemple 2049), on voit que les effets de cache se préserver encore.

Threads	Temps (seconds)	MFlops
16	12.9714	1326.38

On peut noter que les threads aident à paralléliser la multiplication matricielle, comme prévu, puisque l'algorithme est parallélisable pour chaque boucle.

### Exercise 4

Nous savons que la plus grande efficacité se produit pour le cas JKI (dans le code C++ utilisé dans le TP). De plus, si nous utilisons la parallélisation avec 8 threads (meilleur temps obtenu en analysant l'exercice 3), nous obtenons le meilleur temps possible pour la multiplication de matrices 2048x2048:

Threads	Temps (seconds)	MFlops
8	1.37376	12505.8

### Exercise 5

Après avoir mis en œuvre le code, en faisant varier la taille de chaque bloc, le résultat suivant a été obtenu:

Taille bloque	8	16	32	64	128	256
Temps(s)	14.25	10.18	9.09	8.05	8.61	7.32

## Exercice 6

Le meilleur temps pour la division "**bloc-bloc**" c'est pour la taille de 256 qui c'est égal à **7.32s**

Le meilleur temps pour la division "**scalaire**", en utilisant la forme JKI c'est égal à **6.76s**

Le produit scalaire est un peu plus efficace, ça peut être interprété par le fonctionnement de la cache.

## Exercice 7

Après avoir mis en œuvre le code, en faisant varier la taille de chaque bloc, le résultat suivant a été obtenu (pour le cas avec OpenMP, parallélisé):

Taille bloque	8	16	32	64	128	256	512
Temps(s)	3.95	2.41	2.08	1.88	1.92	1.79	2.39

Le meilleur temps pour la division "**bloc-bloc parallèle**" c'est pour la taille de 256 qui c'est égal à **1.79s**

Le meilleur temps pour la division "**scalaire parallèle**", en utilisant la forme JKI c'est égal à **1.40s**

Le produit scalaire est un peu plus efficace, ça peut être interprété par le fonctionnement de la cache.

## Exercice 8

En faisant la comparaison de la meilleur version de notre benchmark JKI parallélisé en 8 threads avec la méthode blas, on obtient les résultats ci-dessous:

Taille matrice (N x N)	Temps Blas	Temps JKI scalaire (8 threads)
512	0.08s	0.02s
1024	0.64s	0.19s
1536	2.82s	0.69s
2048	6.34s	1.98s
3000	19.70s	5.19s

La meilleure version c'est pour la méthode parallélisé JKI.

## Exercise 2.1 - Code dans le git - fichier jeton.py

Résultats:

```
Je suis le processus 0 sur 6 processus
Je m'execute sur l'ordinateur DESKTOP-LIQ0T0F
Je suis le processus 1 sur 6 processus
Je m'execute sur l'ordinateur DESKTOP-LIQ0T0F
Je suis le processus 3 sur 6 processus
Je m'execute sur l'ordinateur DESKTOP-LIQ0T0F
Je suis le processus 4 sur 6 processus
Je m'execute sur l'ordinateur DESKTOP-LIQ0T0F
Je suis le processus 5 sur 6 processus
Je m'execute sur l'ordinateur DESKTOP-LIQ0T0F
Je suis le processus 2 sur 6 processus
Je m'execute sur l'ordinateur DESKTOP-LIQ0T0F
Last Received = 6
```



## Exercise 2.2 Pi

Number of nodes MPI	Temps	PI
1	25.435975 s	3.14166198
2	15.143204 s	3.1416434
3	11.032454 s	3.141635812
4	9.511863 s	3.141587392
5	7.998892 s	3.141639
6	7.673688 s	3.14176996
7	6.775163 s	3.141832256
8	6.576483 s	3.141801824
10	7.272626 s	3.141263416
12	7.449974 s	3.14146168
16	6.915771 s	3.141470944
20	6.673100 s	3.141502776

On peut noter qu'après avoir atteint le nombre de processus 8, le temps optimal est obtenu, et qu'ensuite (pour des valeurs supérieures à 8) le temps obtenu atteint une valeur seuil, qui est proche de 7 secondes pour 1E9 interactions.

Les résultats sont prédits en fonction des paramètres de l'ordinateur utilisé, comme on peut le voir ci-dessous :

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
Address sizes:          39 bits physical, 48 bits virtual
CPU(s):                 8
On-line CPU(s) list:    0-7
Thread(s) per core:     2
Core(s) per socket:     4
Socket(s):              1
Vendor ID:              GenuineIntel
CPU family:             6
Model:                  140
Model name:              11th Gen Intel(R) Core(TM) i7-11390H @ 3.40GHz
Stepping:                2
CPU MHz:                2918.401
BogoMIPS:                5836.80
```