Sur quelques surprises du temps de calcul (dans un algorithme d'optimisation SDP)

Daniel Porumbel 2026

Le code `Julia` ci-après calcule une somme pondérée de matrices d'une façon un peu *vielle école* avec des *boucles fabrication maison*. Sur ma machine, cela prend **23 secondes** qu'on va ramener *à **0.5 secondes** et ensuite à **0.03 secondes!***

```
n = 1000
k = 90
matrices = rand(n,n,k); #[:,:,i] = matrice n×n
                        #      pour tout i∈[1..k]
poids    = rand(k);

#Ci-dessous des boucles maison pour calculer
#sum(poids[i]*matrices[:,:,i] for i in 1:k)
somme_pond = zeros(n, n)
for i in 1:n
    for j in 1:n
        s = 0.0
        for p in 1:k
            s += poids[p] * matrices[i, j, p]
        end
        somme_pond[i, j] = s
    end
end
```

Cet exposé est motivé par la citation ci-après, extraite d'un article sur la résolution des programmes SDP (*Semidefinite programming via Projective Cutting-Planes*). Un programme SDP utilise k matrices $A_1, A_2, \ldots, A_k$ de taille n×n. Sans reposer sur une théorie révolutionnaire, mon algorithme doit effectuer toutefois plusieurs calculs bien plus élaborés qu'une somme de matrices. Une telle somme, même pondérée, peut échapper facilement à l'attention de beaucoup dans le contexte général.

> Lines 6-7 involve computing two weighted-sums of the form $\sum_{i=1}^{k} A_i y_i$. This only requires multiplying some matrices with a scalar and summing them up, which may seem too simple to attract attention. But in the first implementation, we were surprised to notice that the calculation of these two sums was the primary computational bottleneck for $n \geq 1000$ and $k \geq 500$; it was slower than all the proposed projection algorithms. Calculating such sums in the standard schoolbook manner can be way too slow in `Matlab` (and also in Julia according to preliminary experiments). It is better to .....

Ces différences d'efficacité illustrent un paradoxe: s'il est facile de coder en `Julia`, il est aussi trop facile d'écrire un code dont le temps de calcul est assez imprévisible. Je n'ai pas eu trop de surprises de cette ampleur en `C/C++`; si on traduit le code plus haut en `C`, il sera même presque possible d'avoir une idée du code assembleur généré par le compilateur. Voici un inconvénient des nouveaux langages ou compilateurs hautement optimisé et bien (trop?) complexes :

Il est difficile de savoir ce qui se passe lorsque un bloc de code est exécuté.

Les concepteurs de `Julia` ont élaboré une (longue) liste de *performance tips* dont un des premiers dit: s'il existe une méthode toute faite pour exécuter une tâche, il faut l'utiliser au

lieu de réinventer la roue. Cela conduit à la solution suivante qui prend environ **0.5-0.6 secondes** (ou 0.8 si inclut la génération des matrices).

```
sum(poids[i] * matrices[:,:,i] for i in 1:k)
```

Cette dernière solution me paraît simple et naturelle, mais elle reste 20 moins rapide que la solution suivante qui prend **0.03 secondes.**

```
#struct 3D n×n×k → struct 2D n²×k
mat2d    = reshape(matrices,:,k)
#prod matrice 2D*vect via BLAS en C?
somme1d = mat2d*poids
#struct 1D n² → struct 2D n×n
somme2d = reshape(somme1d,n,n)
```

- Premier `reshape(...)` vectorise chacune des k matrices
- Le produit `mat2d*poids` réalise le calcul souhaité
- Dernier `reshape(...)` effectue une dévectorisation

`Julia` utilise peut-être ici une bibliothèque BLAS ultra-optimisée écrite en `C` ou `Fortran`. Cela réalise le calcul souhaité 20 fois plus vite que `sum(...);` pour ceux qui ne veulent pas se spécialiser dans ces bibliothèques, ce niveau d'accélération restera une surprise/mystère. Nota bene : *un code de 7000 lignes peut cacher d'autres surprise de ce type ; qui aura le temps de les étudier pendant des heures et des heures ?*

J’ai fait toutefois une petite étude d’efficacité de ces boucles fabrication maison vis à vis des méthodes à base de vectorisation = opérations sur `arrays` entiers. Déjà, l’existence de ce besoin monte qu’il n’est pas si facile de programmer en *`Julia, Python, Matlab :`* on peut facilement se casser les dents, écrire un code *naïf 1000 fois trop peu rapide.*

Je ne peux pas focaliser trop sur ce `for i for j` : on risque de bloquer sur l'arbre qui cache la forêt

L’optimisation est une science à l’interface avec la programmation. Beaucoup de nos communications ne s’attardent pas sur le code qui fait tourner nos algorithmes. Ces questions d’efficacité de boucles sont peut-être mieux connues dans d’autres communautés. Dans la nomenclature thématique du CNU27 vous trouverez *une rubrique (76) différente de la notre : Langages, compilation, génération de code, interprétation.*

Côté optim, ces surprises de vitesse de calcul sont un symptôme d’un phénomène plus général qui peut trop facilement nous échapper

Exemple : A-t-on encore besoin de passer des heures et des heures à étudier la théorie de complexité en RO? Des fois, peut-être oui. Mais des fois, les garanties de performances qu’elles nous offre risquent d’être trop loin de nos réalités. La complexité des trois solutions évoquées doit être égale à la taille de l'entrée $O(kn^2)$ *:* une mesure trop relative du coût réel de calcul.

On s’est déjà habitué à vivre sans certitudes de performance. Si un langage offre trois méthodes pour réaliser un calcul, comment choisir la plus rapide et être sur de ne pas plomber la performance pour une raison un peu bête? L’IA peut donner quelques indices. Mais si on varie les valeurs de n et de k, on risque de changer le classement d'efficacité des trois méthodes.

Je ne vois pas comment trouver la solution la plus rapide sans tester toutes les alternatives possibles. Si cette étape devient nécessaire pour trop d'opérations, coder avec les nouveaux langages réputés faciles devient plus fastidieux que prévu.

Ce type de langages nous mettent à disposition beaucoup de routines hautement optimisées, écrites d'une manière complexe pour les accélérer au maximum. Beaucoup sont *très habitués* à

les utiliser comme briques de base sans trop connaitre *leur fonctionnement interne ou leur performance qui varie selon les scénarios d'utilisation*. Nous avons quitté depuis longtemps la logique de programmation « what you see is what you get ».

Je pense qu'on aura de plus en plus besoin d'outils de profilage pour mesurer la performance de beaucoup d'opérations. Comme on ne va pas trop faire l'effort, on risque de perdre une certaine notion de stabilité.

Si dix personnes implémentent aujourd'hui deux algorithmes *A* et *B*, cinq pourront dire que la méthode *A* est plus rapide et cinq pourront dire l'inverse -- à chaque fois pour une autre raison. Cela vient du fait que chaque programmeur va déléguer beaucoup de tâches à d'autres routines ou structures de données, faute de temps de les coder soi-même.

Les théoriques pourront argumenter qu'ils ont toujours averti que la programmation était une science trop approximative, sans garanties de performance. Peut-être; mais résoudre le problème comme ça revient à jeter le papier sur lequel il a été écrit. Car il y 20 ans quand le marché était dominé par le *C* et le *C++*, on pouvait compter un peu sur certaines certitudes empiriques de performance; les gens faisaient moins souvent appel à des routines extérieures pour coder le moindre calcul.

Aujourd'hui on utilise ces routines de plus en plus sans connaître leur fonctionnement, à l'image de quelqu'un qui appuie sur la pédale d'accélération sans connaître le rapport de vitesse engagé. On ne sait pas trop quels mécanismes seront enclenchés derrière.