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

Daniel Porumbel, CEDRIC-CNAM, Paris

Une surprise bien cachée dans un code de 7000 lignes

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 demande environ **23 secondes** qu'on va ramener à environ **0.2 secondes**. Et si on enlève le temps nécessaire à générer les matrices, le calcul effectif peut se faire en **0.03 secondes** !

```
n = 1000
k = 90
matrices = rand(n,n,k); #[:,:,i] = matrice n×n pour tout i∈[1..k]
poids    = rand(k);
#je calcule ci-dessous 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 suivante extraite d'un article récemment soumis sur la résolution des programmes SDP.[1] L'algorithme proposé utilise k matrices $A_1, A_2, \ldots A_k$ de taille $n \times n$; sans reposer sur une théorie révolutionnaire, il effectue toutefois plusieurs calculs bien plus élaborés qu'une somme de matrices.

> 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 record (the lower triangular part of) each matrix A_i as a column vector of size $\frac{n(n+1)}{2}$ and to collect the k vectors into a matrix $\widetilde{A} \in \mathbb{R}^{\frac{n(n+1)}{2} \times k}$. This $\widetilde{A}$ is calculated only once, before starting the iterations. Any (lower triangular part of a) sum of matrices $\sum_{i=1}^{k} A_i y_i$ is then computed as matrix product $\widetilde{A}\mathbf{y}$, capitalizing on the highly optimized matrix multiplication routines available in Matlab.

La solution évoquée plus haut réduit le temps de calcul de **23 à 0.2 secondes**, ou plus exactement **0.03** secondes si on ignore le temps nécessaire à générer les matrices ; voir vidéo démo ici : `https://youtu.be/ZdA_xZm9d2s` . Cela illustre 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 imprévisible. Je n'ai pas eu trop de surprises de cette ampleur en `C/C++` ; si on traduit le code plus haut en `C` ,

1. Semidefinite programming via `Projective Cutting-Planes` for dense (easily-feasible) instances, disponible à `https://cedric.cnam.fr/~porumbed/papers/sdpinter.pdf`

il sera même presque possible d'avoir une idée du code assembleur généré par le compilateur. Un inconvénient des nouveaux langages ou compilateurs plus complexes est qu'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 (et ne pas réinventer la roue). Cela conduit à la solution suivante qui prend **0.8 secondes** ou environ **0.5-0.6 secondes** si on ignore la génération des matrices.

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

Cette solution simple et naturelle *reste 20 fois moins rapide* que la précédente qu'on peut coder comme ci-après. La ligne 1 aplatit ou « vectorise » chaque matrice ; la ligne 2 calcule la somme comme un produit d'une matrice avec un vecteur ; la ligne 3 dévectorise le résultat.

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

Je ne sais pas exactement pourquoi ce calcul par multiplication de matrices est 20 fois plus rapide (0.03 vs 0.6 secondes) que celui qui utilise `sum(..)` ; `julia` fait peut-être appel à une bibliothèque BLAS ultra-optimisée écrite en `C` ou `Fortran`. Et ce calcul n'est qu'un exemple : mes 7000 lignes de code peuvent bien cacher d'autres « détails d'implémentation » de ce type.

La complexité des trois solutions évoquées doit être égale à la taille de l'entrée $O(kn^2)$. Ces exemples montrent que la complexité est devenue aujourd'hui une mesure assez relative du coût réel de calcul. 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 ? Personne ne lit plus la doc ; certains utilisent l'IA pour avoir une certaine idée de l'efficacité. 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 en `Julia` ou `Python` 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 la moindre opération. Beaucoup ne vont pas faire l'effort et on risque de perdre facilement une certaine notion de stabilité. [2]

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.

Conclusion

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/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 sont enclenchés derrière.

2. Je voudrais placer cette communication à l'interface entre le code et la théorie. Si la session « Sur les meilleures pratiques de programmation et leur lien avec la théorie » continue, je vais axer mon exposer sur le code `Julia`. Si je me retrouve dans une autre session (d'optimisation non-linéaire), je peux évoquer plutôt le pseudo-code théorique avec une discussion plus « haut niveau » vis à vis de `Julia`.