Projet Julia

Comparaison et écosystème Julia

ROCHAS Martin et CHEVAUX Alice
UE Optimisation Python Julia
M2 SSD 2023 - 2024

Sommaire

Introduction

Ce travail se divise en deux parties,

la première partie est imposée et consiste à discuter des / comparer les langages R, Python et Julia (nous avons ajouté C en bonus) sur un algorithme a priori gourmand en calculs. Nous nous interrogerons aussi sur l'indépendance des langages entre eux et sur les apparentes faiblesses actuelles de Julia.

Dans la seconde partie, plus libre, nous avons choisi d'élaborer un package Julia destiné à l'analyse de survie indépendant de toute librairie statistique existante et avec le minimum de dépendance. Nous terminons ce travail par une comparaison / vérification de notre package Julia avec son équivalent connu en R *survival*.

Figures imposées

Multiplication de matrices carrées

Nous avons choisi la multiplication de matrice carrées pour cette partie. C'est un problème important en algorithmique dont l'efficacité est utile à de nombreux domaines pratiques comme l'ingénierie, l'imagerie et la statistique (réseaux de neurones).

L'algorithme "naïf" qui découle de la définition possède une complexité en temps en $O(n^3)$ avec n la dimension des matrices carrées multipliées. C'est cet algorithme que nous avons implémenté en R, Python, C et Julia.

Pour rappel, soient $A = (a_{ij})_{1 \leq i,j \leq n}$ et $B = (b_{ij})_{1 \leq i,j \leq n}$ deux matrices carrées de taille n. On définit C le produit de A par B comme suit,

$$C = \left(\sum_{k=1}^{n} a_{ik} b_{kj} \right)_{1 \leq i,j \leq n}.$$

Algorithme naïf : pseudo-code

Voici à quoi correspond l'algorithme issu de la définition en pseudo code.

```
Fonction AxB(A, B):

   Créer une nouvelle matrice C de dimensions n x n

   Pour i de 1 à n faire:
       Pour j de 1 à n faire:
           temp = 0
           Pour k de 1 à taille de A faire:
               temp[i, j] = temp[i, j] + A[i, k] * B[k, j]
          C[i, j] = C[i, j] + temp

   Retourner C
```

Implémentation

Les scripts

Tous les scripts,

- R_script.r

- Python_script.py
- C_script.c (et le .exe compilé)
- Julia_script.jl

se trouvent sur le dépôt : https://github.com/MartiRoc/M2_SSD_Julia_projet_Alice-Martin. Ils sont tous organisés / commentés de la même manière. L'implémentation naïve de la multiplication de matrices carrées est identique dans les 4 langages (avec une petite subtilité en C mais cela revient au même lors du calcul du temps d'exécution), c'est la fonction *AxB*.

Dans ces scripts nous calculons la moyenne et l'écart-type du temps d'exécution de *AxB* sur des matrices carrées aléatoires (coefficients $\in \mathcal{U}(0,1)$). Le nombre de produits effectués est encodé par la constante N en début de script et la taille des matrices multipliées par n.

Algorithmes optimisés

Nous calculons aussi la moyenne et l'écart-type du temps d'exécution de *AxB2* qui utilise les outils construits dans les langages ou des librairies pour multiplier les matrices :

- opérateur %*% en R,
- fonction *numpy.dot()* en Python (librairie *numpy*),
- opérateur * sur les Array{} en Julia.

Résultats

Algorithme naïf

Sur $N = 100$ produits de matrices carrées aléatoires de taille $n = 300$ on obtient les moyennes de temps d'exécution suivantes pour $AxB(\ )$,

- R : 1.5838s ($\hat{\sigma} = 0.0460$),
- Python : 2.0545s ($\hat{\sigma} = 0.0859$),
- Julia : 0.0660s ($\hat{\sigma} = 0.0012$),
- C : 0.0793s.

On remarque deux choses importantes, Julia est loin devant Python et R en terme de performance et le temps d'exécution en Julia est proche de celui du C. Leur point commun est que ce sont tous deux des langages compilés ! Notons que dans le temps d'exécution est pris en compte le temps d'allocation de l'espace en mémoire pour la matrice résultat (qu'on déclare dans AxB), qui possède 900 000 coefficients.

Algorithme avec outils

Sur $N = 1000$ produits de matrices carrées aléatoires de taille $n = 300$ on obtient les moyennes de temps d'exécution suivantes pour $AxB2(\ )$,

- R : 0.0097s ($\hat{\sigma} = 0.0095$),
- Python : 0.0041s ($\hat{\sigma} = 0.0010$),
- Julia : 0.0009s ($\hat{\sigma} = 0.0020$),

On note des gains de performances significatifs ! Notons qu'il y a deux explications selon nous à ces gains, la première est algorithmique et résulte du fait que les opérateurs / fonctions utilisées sont basés sur un algorithme de multiplication de matrices plus performant, pour en citer un : l'algorithme de Volker Strassen (1969) possède une complexité en temps en $O(n^{2.807})$. La deuxième raison fait l'objet de la partie suivante.

Dépendances inter-langages : résultats “avec outils”

Nous pensons que l'opérateur %*% de R est basé à 100% sur du C et que la librairie numpy de Python repose principalement sur du C et du Fortran. Tandis que nous pensons qu'en Julia l'opérateur de multiplication d'Array{} est construit 100% en Julia.

Limites de Julia

Dans les figures imposées, il est demandé de citer si possible un exemple d'utilisation de base où nous pensons que Julia est moins bien que R ou Python. Nous n'en avons pas trouvé. A la limite nous pourrions reprocher à Julia de ne pas être aussi riche que R et Python en librairies spécialisées (en statistique par exemple), mais Julia est en plein développement avec une communauté active et grandissante, donc cela est amené à changer avec le temps.

Figures libres

Package SurvieAM.jl

Pour cette partie nous avons élaboré un package en Julia destiné à l'analyse de survie (AM pour Alice & Martin évidemment). Sa présentation et son installation sont expliquées ici,

https://github.com/MartiRoc/SurvieAM.jl

Ce package possède trois fonctions principales,

- KM : qui calcule l'estimateur de kaplan-Meier à partir de données de survie,
- KM_curve : qui trace n'importe quelle sortie de KM et,
- Log_Rank : qui opère le test du même nom.

Son installation importe automatiquement les données de survie fictives suivantes sous forme de DataFrame dans la variable *df_test*:

duree	statut	traitement	fonction	duree	statut	traitement	fonction
8	1	1	A	220	1	1	N
8	1	1	N	365	0	1	N
13	1	2	A	632	1	2	N
18	1	2	A	700	1	2	N
23	1	2	A	852	0	1	N
52	1	1	A	1296	1	2	N
63	1	1	A	1296	0	1	N
63	1	1	A	1328	0	1	N
70	1	2	N	1460	0	1	N
76	1	2	N	1976	0	1	N
180	1	2	N	1990	0	2	N
195	1	2	N	2240	0	2	N
210	1	2	N				

N : fonction rénale normale, A : fonction rénale anormale

Comparaison avec *survival* de R

Nous donnons ici des résultats d'analyse de survie sur les données ci-dessus, obtenus avec R et le package *survival*, ainsi que les commandes pour faire les mêmes analyses avec notre package en Julia. Nous prétendons obtenir la même chose.

A noter que la documentation des fonctions (détail des entrées et sorties) se trouve dans la présentation du package ici https://github.com/MartiRoc/SurvieAM.jl. Et le code extensivement commenté est disponible ici https://github.com/MartiRoc/M2_SSD_Julia_projet_Alice-Martin, dans Script_auto-suffisant_package.jl.

Estimateur K-M pour le groupe à la fonction rénale normale

```
renal_N <- df_test[df_test$fonction == "N",] # extraction du groupe
survfit(Surv(duree, statut) ~ 1, data = renal_N)$surv
```

```
##  [1] 0.9444444 0.8888889 0.8333333 0.7777778 0.7222222 0.6666667 0.6111111
##  [8] 0.6111111 0.5500000 0.4888889 0.4888889 0.4190476 0.4190476 0.4190476
## [15] 0.4190476 0.4190476 0.4190476
```

En julia,

```
res = KM(df_test.duree, df_test.statut, df_test.fonction)
res.b
```

Test du Log-rank

Entre les deux groupes de fonction rénale

```
survdiff(Surv(duree, statut) ~ fonction, data = df_test)
```

```
## Call:
## survdiff(formula = Surv(duree, statut) ~ fonction, data = df_test)
##
##             N Observed Expected (O-E)^2/E (O-E)^2/V
## fonction=A  7        7      1.6     18.24        24
## fonction=N 18       10     15.4      1.89        24
##
##  Chisq= 24  on 1 degrees of freedom, p= 1e-06
```

En julia,

```
Log_Rank(df_test.duree, df_test.statut, df_test.fonction)
```

Entre les deux groupes de traitement

```
survdiff(Surv(duree, statut) ~ traitement, data = df_test)
```

```
## Call:
## survdiff(formula = Surv(duree, statut) ~ traitement, data = df_test)
##
##                N Observed Expected (O-E)^2/E (O-E)^2/V
## traitement=1 12        6     8.34     0.655      1.31
## traitement=2 13       11     8.66     0.631      1.31
##
##  Chisq= 1.3  on 1 degrees of freedom, p= 0.3
```

En julia,

```
Log_Rank(df_test.duree, df_test.statut, df_test.traitement)
```

Courbes de survie estimées pour les deux groupes de traitements

Avec notre fonction *KM_curve* en Julia,

```
res = KM(df_test.duree, df_test.statut, df_test.traitement)
KM_curve(res)
```

on obtient,