Descente de gradient stochastique - parallélisation

Paul Graffant, Julien Foënet

4 février 2020

Introduction

Ce projet a pour but de déterminer les différences de performances entre plusieurs langages, de manière parallèle et non parallèle, d'un algorithme classique d'optimisation : la descente de gradient stochastique. Pour cela nous avons choisit le modèle simple de la régression linéaire, car cela nous permet de facilement déterminer la justesse de nos résultats et ainsi de pouvoir nous concentrer sur la comparaison de performance de nos algorithmes.

Nous allons tout d'abord présenter brièvement le modèle ainsi que l'expression de la descente de gradient que nous avons implémenté, puis nous allons dans une deuxième partie comparer les performances de trois langages sur l'algorithme codé de manière séquentielle : Python, Cython et C++. Enfin dans une dernière partie nous comparerons les performances de notre algorithme en code C++ de manière parallèle et séquentielle. Tout ceci a pour but de bien se rendre compte des ordres de grandeurs et des gains réalisés en terme de temps de calcul sur un algorithme utilisé très fréquemment dans les modèles de machine learning, et qui tourne parfois derrière des méthodes que nous appliquons grâce à des librairies très optimisés telles que scikit-learn.

I) Le modèle

Notre modèle est une régression linéaire multiple :

$$Y = X\beta + \epsilon$$

avec $X \in \mathbb{R}^{n \times p}$, $Y \in \mathbb{R}^n$, $\beta \in \mathbb{R}^p$ et $\epsilon \sim \mathcal{N}(0, 4)$, dans laquelle nous simulons les données X_{ij} par une loi uniforme sur $(0, 10)$, et les coefficients de la régression β_j par une loi normale $\mathcal{N}(0, 9)$.

La fonction à minimiser pour obtenir les coefficients β_j est donc :

$$f(\beta) = \|Y - X\beta\|^2 = \frac{1}{n}\sum_{i=1}^{n}(Y_i - X_i\beta)^2$$

Et la descente de gradient stochastique que nous avons implémenté pour résoudre ce problème s'écrit :

$$\beta_{t+1} = \beta_t - \eta \, \partial f(\beta_t)$$

où $\partial f(\beta_t) = -2X_i(Y_i - X_i\beta)$

II) Algorithme séquentiel

Dans un premier temps nous avons décidé de nous concentrer sur la descente de gradient séquentielle afin de bien différentier les performances de chacun des langages utilisés et ainsi mieux appréhender par la suite les différences de performances réalisées avec la parallélisation.

Nous avons comparé trois langages différents : python, cython et C++. Pour chacun des trois langages nous avons réalisé une descente de gradient stochastique afin de trouver les paramètres de notre régression avec un nombre de feature $p = 8$, en faisant varier le nombre de données n entre 10000 et 50000000, et nous avons ainsi pu récupérer et mettre sous forme de tableau les différents temps de performance de nos algorithmes.

Pour la version python du code, nous n'avons pas utilisé la librairie numpy pour calculer les produits scalaires $< X_i, \beta >$ et $< X_i, y_i - X_i\beta >$ au sein de notre boucle, car comme nos boucles en cython et C++ ne contiennent pas de méthodes issues de librairies optimisées pour calculer cela, nous avons trouvé préférable de mettre les trois algorithmes au même niveau afin de bien pouvoir comparer leurs performances.

Nous avons du jouer avec le paramètre η de nos algorithmes afin d'obtenir toujours si possible de très bon résultats en terme de distance quadratique entre les vrais coefficients et ceux que l'on estime. En effet nous nous sommes rendus compte que lorsque l'on bouge le nombre d'observation, il arrive que nos coefficients explosent voir même qu'ils produisent des "nan" (car trop volumineux), il faut alors baisser la valeur de η afin d'éviter ce phénomène.

Il est également à noter que les valeurs des $(\beta - \hat{\beta})^2$ ne sont pas les même avec C++ qu'avec python et cython, car la randomisation n'est pas la même et car le paramètre η ne décroit pas de la même manière (en python et en cython on le divise par un nombre dans un np.logspace, alors qu'en C++ on le multiplie par un nombre très proche de 1). Voici nos résultats :

n	p	eta	Temps de performance			Ecart quadratique coefs		
			Python	Cython	C++	Python	Cython	C++
10000	8	0,004	0,52s	0.32s	0,034s	6,7	6,7	1,33
50000	8	0,004	1,44s	0,67s	0,14s	0,032	0,032	0,037
100000	8	0,0025	2,63s	1,14s	0.28s	0,0416	0,0416	0,095
200000	8	0,0025	5.23s	2,21s,	0,57s	0,00007	0,00007	0,041
500000	8	0,001	12,60s	4,89s	1.44s	0,00031	0,00031	0,0069
1000000	8	0,0005	24,71s	9,32s	2,89s	0,000014	0,000014	0,012
2000000	8	0,0005	47,61s	18,47s	5,78s	0,000069	0,000069	0,0025
5000000	8	0,0002	1m58,70s	45,11s	15,34s	0,00000184	0,00000184	0,00067
10000000	8	0,00008	3m59,79s	1m35,37s	30,35s	0,00000022	0,00000022	0,000019
50000000	8	0,00005	21m46,90s	8m57,21s	2m35,43s	0,0000025	0,0000025	0,0015

FIGURE 1 – Temps de performance et écarts quadratiques des coefficients pour python, cython, C++

En mettant cela sous la forme d'un graphique avec le nombre de donnée en abscisse et le temps de calcul en ordonnée, nous obtenons la figure suivante :

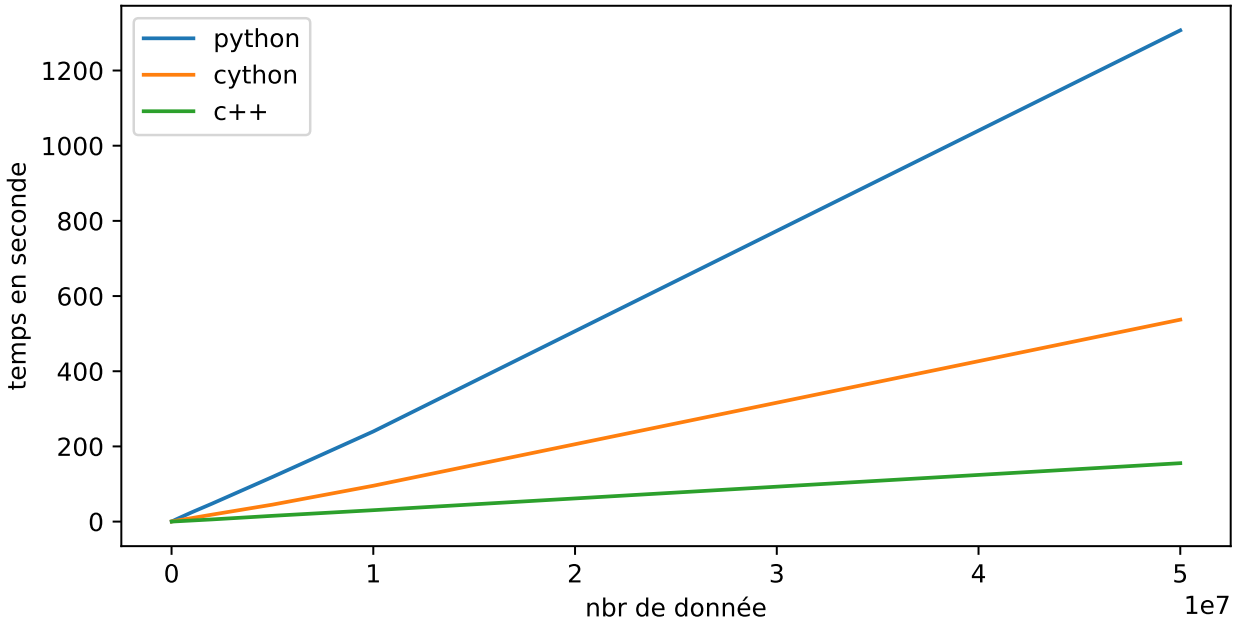

FIGURE 2 – Temps de calcul en fonction du nombre de données

On remarque ainsi que, comme on pouvait s'y attendre, le temps de calcul en C++ est considérablement réduit par rapport au temps de calcul en cython, qui lui-même est bien plus faible que celui en python classique.

III) Parallélisation

Pour paralléliser la descente du gradient nous avons utilisé l'api OpenMp en C++. OpenMp permet de paralléliser le code en faisant du "multi threading" qui est une parallélisation en mémoire partagée. Dans ce paradigme de programmation, les threads coopèrent et se coordonnent entre eux à l'aide de directives de compilation et partagent l'information à l'aide d'un espace mémoire commun le "shared". OpenMp utilise le modèle "fork-join" pour l'exécution parallèle. Le programme commence avec un unique thread (le master thread) qui exécute la partie séquentielle du programme. Lorsqu'une directive de parallélisation est rencontrée, le master thread crée une équipe de threads pour paralléliser la tâche. A la fin de la région parallèle les threads se synchronisent et s'arrêtent ne laissant que le master thread.

i Algorithme

Afin de paralléliser la descente du gradient stochastique nous avons employé la stratégie suivante :
Nous avons divisé le jeu de données entre les threads afin que chacun puisse calculer le *gradient* et mettre à jour les coefficients du paramètre *beta*. Les gradients calculés sont stockés dans une matrice située dans le "shared", elle est donc accessible par tous les threads en lecture et en écriture. A la fin de la partie parallèle les coefficients de *beta* sont calculés en faisant la moyenne des coefficients trouvés par chaque thread.

Dans la partie parallèle de l'algorithme on notera que les variables *beta-t* et *gradient-t* sont en *firstprivate*, ceci indique que chaque thread reçoit une copie privée des variables avec les valeurs initialisées dans la partie séquentielle. Ainsi chaque thread peut modifier ces variables de manière indépendante et hermétique. La variable *beta-hist* qui est

la matrice accueillant les *beta-t* des threads est en *shared*. Ainsi elle est dans la mémoire partagée et peut être accédée (en lecture/écriture) par tous les threads dans la région parallèle.

```
#pragma omp parallel for private(id) firstprivate(beta_t, y_hat_t, gradient_t, eta, p, tab_index)\
shared(beta_hist)  num_threads(num_threads) schedule(static)
for (int j=0; j<n; j++){
   eta = eta * 0.999955;
   float* X_j = X[tab_index[j]];
   float y_j = y[tab_index[j]];
   y_hat_t = 0;
   for (int k=0; k<p; k++){
      y_hat_t += X_j[k] * beta_t[k];
   }
   for (int k=0; k<p; k++){
      gradient_t[k] = 0;
      gradient_t[k] = (X_j[k] * (y_j - y_hat_t)) * (-2.0);
      beta_t[k] = beta_t[k] - eta * gradient_t[k];
      beta_hist[k][j] = beta_t[k];
      }

}

return average_para(beta_hist, n, p, num_threads);
```

FIGURE 3 – Partie parallèle de l'algorithme de descente du gradient stochastique

Nous avons utilisé la politique de partage du travail *static* qui répartie de manière équitable les itérations de la boucle *for* entre les threads. Cette politique de partage du travail est la mieux adaptée lorsque les charges de travail de chaque itération sont équilibrées. Nous avons comparé avec la distribution du travail *dynamic* qui assigne aux threads les itérations au fur et à mesure que chaque threads termine son travail (mieux adpaté au charges de travail déséquilibrées) sans observer de changement dans les performances.

On notera également que nous avons organisé la matrice *beta-hist* de façon à avoir en ligne les coefficients betas et en colonne les individus sur lesquels ils sont calculés. Ainsi lorsque nous faisons la moyenne $\beta_k^{final} = \frac{1}{n}\sum_{i=1}^{n}\beta_{ki}$ nous itérons d'abord sur les lignes puis sur les colonnes (fig 2).

```
float* res = new float[p];
for (int i=0; i<p; i++){
   res[i] = 0;
   for (int j=0; j<n; j++){
      res[i] += arr[i][j];
   }
   res[i] = res[i] / n;
```

FIGURE 4 – Calcul du paramètres *beta* final en tant que moyenne des itérations intermédiaires

En respectant cet ordre d'itération on optimise les performances au niveau du cache. En effet, naturellement en C++ le processeur va "prefetch" la ligne de la matrice sur laquelle on travail et la monter en mémoire dans le cache L1 et L2. Si nous avions choisis de stocker les betas en colonnes et les individus en lignes alors le temps gagné par cette fonctionnalité aurait été perdu.
(Un sujet subsiste du fait que la variable utilisée est un pointeur vers une matrice (https ://ubuntuforums.org/archive/index.php/t-1223537.html forum traitant de la question). Cependant, en expérimentant les deux méthodes d'implémentation nous avons constaté une amélioration des performances).

ii Comparaison des performances entre SGD séquentielle et parallèle

Dans un premier temps, nous allons fixer le nombre de variables (p) à 8, le nombre de threads à 4 et faire varier le nombre de données. L'échelle du nombre de données étant en logarithme voici la correspondance en millions [1M, 5M, 10M, 15M, 20M, 50M]. Le nombre de threads est à 4 car la machine sur laquelle nous avons fait les tests possède 4 coeurs. De plus afin d'atteindre la convergence il est nécessaire de faire des ajustements du *eta*.

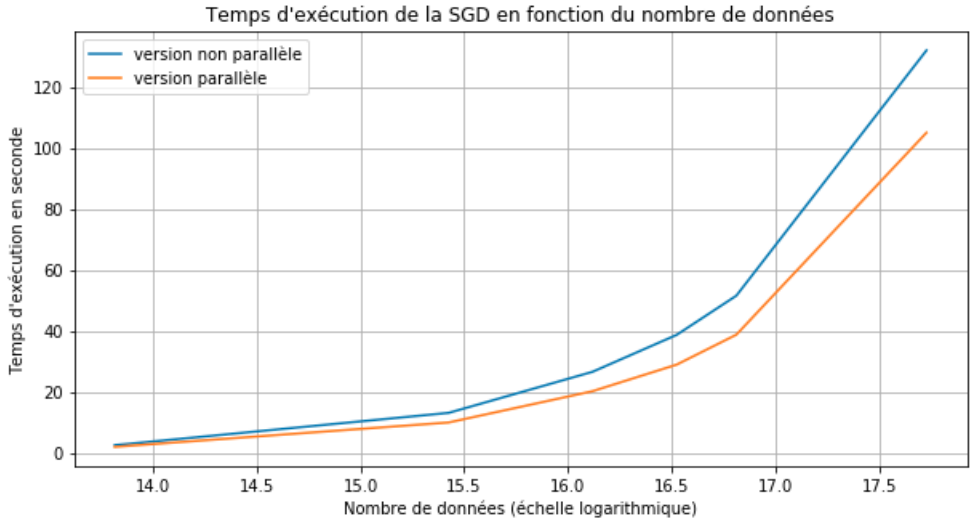

FIGURE 5 – Temps d'exécution de la SGD en fonction du nombre de données

La figure 5 confirme que l'implémentation parallèle de l'algorithme réduit le temps d'exécution.

Maintenant nous allons fixer le nombre de données à 5 millions, le nombre de threads à 4 et faire varier le nombre de variables de la régression.

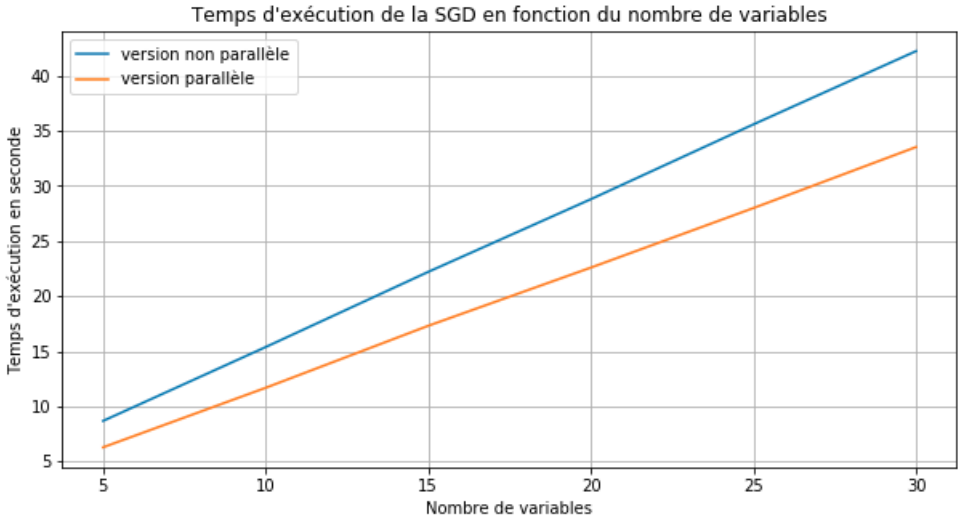

FIGURE 6 – Temps d'exécution de la SGD en fonction du nombre de variables

Ici aussi on constate (figure 6) que la version parallèle est plus rapide que la version séquentielle.

Pour finir nous allons étudier l'évolution du temps d'exécution en fonction du nombre de threads utilisés. On fixe le nombre de données à 5 millions et le nombre de variables à 8.

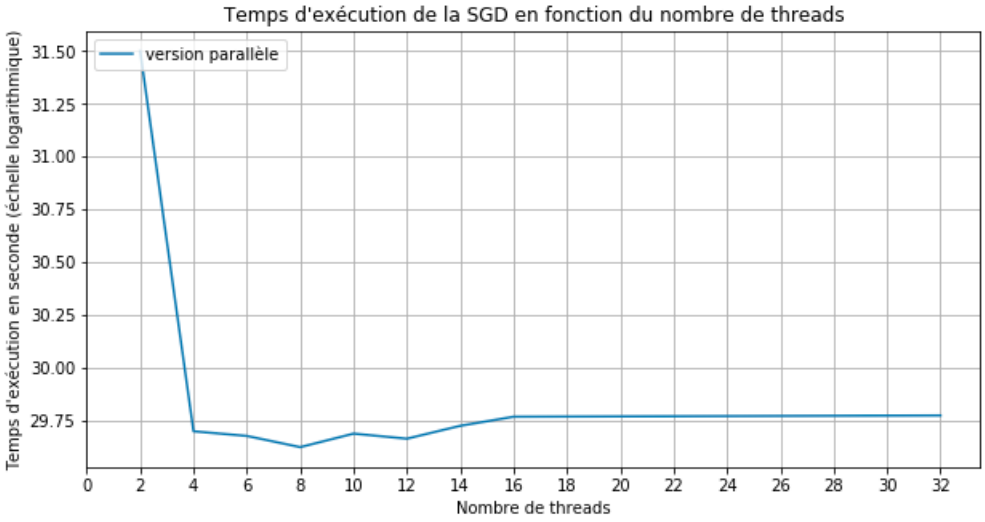

FIGURE 7 – Calcul du paramètres *beta* final en tant que moyenne des itérations intermédiares

On voit sur la figure 7 qu'après avoir atteint le nombre de coeurs de la machine (4), le temps d'exécution arrête de diminuer. En effet, une fois le nombre de coeurs atteint, les threads en "plus" s'exécutent de manière concurrente et non plus parallèle. On ne gagne donc plus de performance en distribuant les tâches.

Conclusion

Pour conclure ce projet nous a permis de mettre en application les enseignements du cours *d'éléments logiciels pour le traitement des données massives*. Nous avons pu pratiquer le cython, l'api OpenMp et nous sensibiliser à l'optimisation de cache.

Une amélioration possible de notre projet serait de supprimer la matrice stockant les résultats intermédiaires dans la partie parallèle du code. En effet cette matrice occupe de l'espace en mémoire et peut causer des problèmes quand on a des centaines de millions de données. Une idée serait de faire des étapes de synchronisation intermédiaires des threads afin de libérer une partie de l'espace mémoire. Cependant cette technique aurait un coût en temps car elle force les threads à s'attendre plusieurs fois pendant l'exécution du programme.