

Phase 2 du projet

MTH8211

Ulrich Baron-Fournier, Petru Lepreanu
Polytechnique Montréal
ulrich.baron-fournier@polymtl.ca, petru.lepreanu@polymtl.ca

Remise 2 LSRN : Une méthode parallèle pour les systèmes linéaires fortement rectangulaires

Lien du Github: https://github.com/Ulrizpascuit/Projet_MTH8211.git

Description de la problématique (révisée)

Le projet s'intéresse à la résolution de grands systèmes linéaires fortement rectangulaires, c'est-à-dire des problèmes aux moindres carrés linéaires où la matrice A est de taille $m \times n$ avec un écart extrême entre m et n (beaucoup plus d'équations que d'inconnues si $m \gg n$, cas surdéterminé, ou l'inverse $m \ll n$, cas sous-déterminé). Dans de telles situations, on cherche typiquement à calculer la solution de norme minimale du problème $\|Ax - b\|_2^2$, c'est-à-dire la solution de plus petite longueur qui satisfait au mieux $Ax \approx b$. Ces problèmes apparaissent dans de nombreuses applications scientifiques et d'ingénierie, et la demande en solveurs plus rapides et précis s'accroît avec la taille des données et des modèles. Les approches classiques pour les moindres carrés (telles que la résolution des équations normales $A^T Ax = A^T b$ ou les factorisations QR/SVD) deviennent coûteuses ou peu pratiques lorsque m et n sont très déséquilibrés, et peuvent en outre souffrir de problèmes de conditionnement numérique.

Les méthodes itératives comme LSQR réduisent les besoins en mémoire, mais leur vitesse de convergence dépend fortement du conditionnement de $A^T A$. Dans le cas de systèmes extrêmement rectangulaires, il est important de préconditionner le problème pour en améliorer le conditionnement avant d'appliquer un solveur itératif. C'est dans ce contexte qu'intervient LSRN, une méthode introduisant un élément aléatoire dans le processus de résolution. LSRN utilise un échantillonnage gaussien aléatoire (projection aléatoire) pour construire un préconditionneur qui rend le système préconditionné extrêmement bien conditionné avec une probabilité élevée. De plus, cette étape de préconditionnement est hautement parallélisable et profite de la structure creuse des matrices ou d'opérateurs linéaires rapides, ce qui la rend adaptée au calcul distribué moderne.

En somme, l'objectif est de résoudre rapidement et avec précision des systèmes très rectangulaires de grande taille – un défi que des méthodes aléatoires parallèles récentes comme LSRN cherchent à relever. Dans le cadre de ce projet, nous avons pour ambition de lire et comprendre en détail l'algorithme LSRN (décrit par Meng et al. en 2014), puis d'en fournir une implémentation efficace

en Julia et d'évaluer ses performances sur des problèmes de grande dimension. Le présent rapport (Phase 2) fait suite à une phase 1 où la problématique et les objectifs ont été définis, et il se concentre sur l'implémentation de LSRN (version séquentielle et parallèle) ainsi que sur les résultats préliminaires obtenus. Le code développé est disponible dans le dépôt GitHub du projet ([lien GitHub à insérer]).

Implémentation de la méthode LSRN en Julia

Version séquentielle (implantation de base)

La version séquentielle de LSRN a été codée comme une preuve de concept initiale, en suivant pas à pas l'algorithme décrit dans l'article de Meng et al. (2014). Pour un système $Ax \approx b$ de dimensions $m \times n$, l'algorithme s'exécute de la façon suivante :

Étape 1 : Projection aléatoire.

On génère une matrice aléatoire G de dimensions appropriées (de taille $l \times m$ si $m \geq n$, ou $l \times n$ dans le cas sous-déterminé) avec des entrées i.i.d. selon $\mathcal{N}(0, 1)$. Le paramètre l (nombre de lignes échantillonnées) est choisi un peu plus grand que le minimum (n ou m) afin de garantir avec une probabilité élevée que $\text{rang}(GA) = \text{rang}(A)$. Par exemple, nous utilisons typiquement $l = \lceil 2n \rceil$ dans le cas surdéterminé, conformément aux recommandations de l'article original. Ensuite, on calcule la matrice projetée $B = GA$, de taille $l \times n$.

Étape 2 : Préconditionneur via factorisation.

On factorise la matrice B pour en extraire un preconditionneur efficace. Dans notre implémentation séquentielle, nous employons une décomposition QR sur B . Plus précisément, on calcule la factorisation $B = QR$ (QR « économie », avec $Q \in \mathbb{R}^{l \times n}$ et $R \in \mathbb{R}^{n \times n}$ triangulaire supérieure). La matrice R obtenue sert de preconditionneur pour le système original. Intuitivement, R approxime (avec de bonnes garanties probabilistes) un facteur de Cholesky de $A^T A$ bien conditionné.

Étape 3 : Résolution itérative preconditionnée.

Une fois R calculé, nous devons résoudre le système $Ax \approx b$ en utilisant R comme preconditionneur à gauche. Concrètement, nous voulons résoudre $\|Ax - b\|_2$ avec $R^{-1}A$ au lieu de A (et $R^{-1}b$ au lieu de b) afin d'accélérer la convergence. Nous faisons appel à un solveur de Krylov adapté aux moindres carrés, en l'occurrence l'algorithme LSQR (fourni par la bibliothèque Krylov.jl).

L'implémentation séquentielle a été validée sur des cas tests de petite taille afin de s'assurer du bon fonctionnement de chaque étape. Par exemple, nous avons résolu un petit système dense aléatoire et comparé la solution obtenue avec celle donnée par un solveur direct (QR de Julia) pour vérifier que les solutions coïncident à la précision près. De même, sur un exemple creux aléatoire, nous avons confirmé que l'algorithme renvoie une résiduelle très faible.

Version parallèle (multi-threading en Julia)

La méthode LSRN se prête particulièrement bien à la parallélisation, en raison notamment de son étape de projection aléatoire qui peut exploiter des calculs matriciels de grande taille hautement parallélisables. Nous avons donc développé en parallèle une version multi-thread de l'implémentation Julia, dans le but de réduire les temps de calcul sur les grandes instances.

Génération et multiplication aléatoire en parallèle

La génération de la matrice G et le calcul de $B = GA$ sont répartis entre plusieurs threads Julia. Chaque thread multiplie une sous-matrice de G par A , puis les résultats sont combinés pour former B . Cette parallélisation par blocs est d'autant plus efficace que A est de grande taille.

Factorisation QR multi-threadée

Une fois la matrice B assemblée, nous utilisons les routines de factorisation QR optimisées de Julia (basées sur LAPACK/BLAS pour le dense, ou SPQR pour le creux). Ces bibliothèques exploitent le parallélisme interne pour accélérer le calcul de R .

Phase itérative parallèle

L'algorithme LSQR en lui-même est essentiellement séquentiel, mais chaque produit matrice-vecteur par A ou A^T bénéficie du parallélisme des opérations BLAS ou du multi-threading sur matrices creuses. Le gain de parallélisation est surtout visible pour les grandes matrices denses.

En résumé, la version parallèle de LSRN exploite le multi-threading lors des étapes les plus coûteuses (génération/projection aléatoire, multiplications matrice-vecteur), ce qui permet d'accélérer significativement la résolution sur des machines multi-cœurs. Cet aspect n'a pas encore été ajouté dans le code et donc aucune analyse et comparaison avec les algorithmes séquentiel n'ont été faites.

Résultats numériques préliminaires

Matrices denses aléatoires

Pour évaluer les performances, nous avons généré des matrices dense de taille $m = 10\,000$, $n = 1\,000$ et $m = 100\,000$, $n = 10\,000$ (sur-déterminée, $m \gg n$). Les entrées de A ont été tirées uniformément sur $[-1, 1]$ et le vecteur b également aléatoire. Nous comparons l'exécution de l'algorithme LSQR de base à l'algorithme LSQR en utilisant le préconditionnement de LSRN en version séquentielle. Les résultats sont résumés ci-dessous :

Système (dense)	Itération LSQR	Itérations LSRN_LSQR	Residu relatif LSQR	Residu relatif LSRN_LSQR
$A\ 10^4 \times 10^3$	1498	43	6.44×10^{-7}	1.78×10^{-7}
$A\ 10^5 \times 10^3$	1501	43	3.43×10^{-7}	1.78×10^{-7}

Dans le premier cas ($m = 10\,000$, $n = 1\,000$), LSQR trouve la solution en 1498 itérations, avec un résidu relatif $\|Ax - b\|/\|b\|$ d'environ 6.44×10^{-7} et LSRN_LSQR trouve la solution en 43 itérations avec un résidu relatif d'environ 1.78×10^{-7} (plus faible que LSQR). On voit donc que l'application de LSRN permet à LSQR de résoudre le problème en beaucoup moins d'itérations pour obtenir un résidu qui est même plus faible. Dans le deuxième cas, On trouve les mêmes genres de résultats, ce qui confirme le bon fonctionnement de l'algorithme LSRN!

De plus, un benchmark a été effectué pour la méthode LSQR et la méthode LSRN_LSQR afin de comparer les temps de calcul pour les même deux problèmes.

Système (dense)	Temps moyen LSQR	Temps moyen LSRN_LSQR
$A 10^4 \times 10^3$	2.604 s \pm 81.057 ms	900.802 ms \pm 34.475 ms
$A 10^5 \times 10^3$	34.445 s	4.110 s \pm 286.480 ms

On peut voir ici que le temps de calcul pour la méthode LSRN_LSQR est beaucoup plus faible que pour LSQR. Cela montre donc que le préconditionneur de LSRN permet vraiment de rendre le problème plus facile à résoudre en améliorant son conditionnement. On voit la puissance que peuvent avoir les préconditionneurs basés sur l'aléatoire.

Matrices creuses (sparse) aléatoires et réelles

Encore aucun test a été fait pour les matrices creuses. Le but serait le même que pour les matrices dense. Il faut comparer des solveurs de problèmes creux avant et après l'application du préconditionneur provenant de LSRN.

Subtilités et défis rencontrés en phase 2

Choix initial du solveur

Dans l'article, plusieurs solveurs ont été utilisés et comparé entre eux. **### Cas sous-déterminé** LSRN a été adapté pour $m < n$ en appliquant la projection aléatoire sur A^T . **### Utilisation d'exemples test multiples** Afin d'avoir plus de flexibilité sur les problèmes fortement rectangulaires, Une fonction générative de matrice a été **### Gestion du parallélisme** En premier lieu, nous avons essayé de produire un algorithme qui utilise la parallélisme avec l'astuce suivante:

```


nthreads = Threads.nthreads()
blocksize = ceil{Int, s / nthreads}








Threads.@threads for t = 1:nthreads
    i1 = (t-1)*blocksize + 1
    i2 = min(t*blocksize, s)
    if i1 <= i2
        B[i1:i2, :] .= G[i1:i2, :] * A
    end
end

```

En utilisant cet astuce pour utiliser le parallélisme, nous n'avons pas remarqué de gain de performance. Cela peut être dû à énormément de chose. Cependant, la librairie BLAS effectue du calcul multi-threadé de base. **Tests de robustesse**: Les cas mal conditionnés et de rang déficient sont bien gérés grâce au préconditionneur aléatoire.

Échéancier récapitulatif et prochaines étapes

Étape	Statut
Reproduction de l'algorithme LSRN de base pour système sur déterminé	Réalisée  -

Étape	Statut
Reproduction de l'algorithme LSRN de base pour système sous déterminé	Réalisée  -
Tests préliminaires sur matrices aléatoires.	Réalisée  -
Rédaction du rapport intermédiaire (phase 2).	Réalisée 
Tests à grande échelle, comparaison avec DGELSD, SuiteSparseQR, Blendenpik?	À faire 
Validation de la méthode parallèle	À faire 
Optimisations finales du code, documentation	À faire 
Rédaction du rapport final (phase 3)	À faire 

Conclusion provisoire

À l'issue de cette phase 2, nous disposons d'une implémentation de base fonctionnelle de LSRN en Julia, capable de résoudre des systèmes linéaires fortement rectangulaires de grande taille avec une haute précision. Les premiers résultats obtenus sont encourageants : ils confirment les performances attendues de LSRN en termes de nombre d'itération et de temps. Dans la prochaine phase, nous poursuivrons les tests à grande échelle et les comparaisons approfondies avec d'autres solveurs. Le projet permet d'illustrer concrètement l'apport des méthodes aléatoires pour l'algèbre linéaire numérique à grande échelle.
