Quand Krylov rencontre Gram-Schmidt : trouver des vecteurs dans le noyau de grandes matrices creuses modulo p par l'algorithme de block-Lanczos

Charles Bouillaguet

v0.0 — le ... janvier 2022

1 introduction

Le but du projet est de paralléliser un programme séquentiel (que nous vous fournissons) qui effectue la résolution de systèmes linéaires du type $xM = 0 \bmod p$, où M est une matrice *creuse* de taille $N \times (N - k)$.

Ce problème est notamment une des étapes calculatoirement difficile des meilleurs algorithmes de factorisation des grands entiers (avec $p = 2$) et de calcul de logarithme discret dans $\mathbb{Z}_p$ (avec $p > 2$).

Ce programme implante l'algorithme de *Lanczos par bloc*. Inventée au milieu des années 1990, il s'agit d'une méthode *itérative* : le calcul se fait principalement en calculant des produits matrice-vecteur avec la matrice M. L'avantage de cette famille d'algorithmes, c'est qu'une fois que la matrice M tient en mémoire alors on peut lancer le calcul et la consommation mémoire ne va plus augmenter. *A contrario*, elle peut exploser dans les algorithmes d'élimination gaussienne creuse.

Ceci permet de résoudre de très grands systèmes linéaires creux, par exemple avec N de l'ordre de plusieurs millions, sur des ordinateurs personnels. Si ces systèmes étaient denses, il faudrait des centaines de Tera-octets ne serait-ce que pour stocker la matrice.

Une matrice creuse est principalement décrite par ses dimensions m et n ainsi que le nombre nz de ses coefficients qui sont non-nuls (*non-zero*). En effet, on évite soigneusement :

- De stocker le nombre zéro.
- De calculer $0 \times x$ et $0 + x$.

Ce document ne décrit pas précisément l'algorithme de Lanczos par bloc. Cet algorithme permet de résoudre $xA = 0 \bmod p$ lorsque A est une matrice *symétrique*. Comme celle de départ ne l'est pas forcément, on pose $A = MM^t$, qui l'est.

L'idée générale consiste à calculer une séquence $w_0, w_1, \ldots,$ de *blocs de vecteurs* A-orthogonaux, c'est-à-dire de matrices de taille $N \times n$ (pour de petites valeurs de n) telles que ${w_i}^t A w_j = 0 \bmod p$ lorsque $i \neq j$. Au bout d'un moment, w_i va fatalement appartenir à l'espace vectoriel engendré par les $w_0, \ldots, w_{i-1}$. Alors on a ${w_i}^t A w_i = 0 \bmod p$, donc ${w_i}^t M M^t w_i = 0 \bmod p$, et de là on obtient heuristiquement ${w_i}^t M = 0 \bmod p$.

L'algorithme peut aussi servir à résoudre $Mx = 0$, en transposant la matrice de départ.

2 Fonctionnement du projet

Vous devez télécharger le fichier :

`http://hpc.sfpn.net/static/project.py`

puis l'exécuter. Cela va télécharger le reste des fichiers nécessaire. Ce script vérifie automatiquement la présence de mises à jour et les télécharge automatiquement. Il pourrait bénéficier de nouvelles fonctionnalités pendant le semestre. Sans argument, il affiche la liste des commandes disponibles.

3 Description du code séquentiel

Le code séquentiel est constitué de trois fichiers : `lanczos_modp.c`, `mmio.c` et `mmio.h`. Un `Makefile` est également fourni. Les fichiers `mmio.*` sont issus d'une bibliothèque pour la lecture et l'écriture des matrices au format MatrixMarket [1] offerte par le NIST, une agence de standardisation du gouvernement américain. Ils ont été légèrement modifiés pour les besoins de ce projet.

Le « vrai » code est dans `lanczos_modp.c`. Il charge une matrice creuse M au format MatrixMarket depuis un fichier puis résout $xM = 0 \bmod p$ en se chronométrant.

Dans le fichier, il y a un en-tête puis la matrice est représentée sous la forme d'une *liste de triplets* (« *COOrdinate representation* ») : pour chaque entrée $A_{ij} \neq 0$, on stocke le triplet (i, j, A_{ij}).

Le code séquentiel fourni est une implantation directe et un peu naïve de l'algorithme. Du point de vue des performances, elle est très améliorable (même séquentiellement).

La taille du bloc (nommée n ci-dessus) est un paramètre de l'algorithme qui influe sur les performances.

4 Matrices de *benchmark*

Le programme fournit est capable de trouver un vecteur dans le noyau de n'importe quelle matrice à coefficients entier disponible dans la SuiteSparse Matrix Collection (`https://sparse.tamu.edu/`).

Par ailleur, le script de gestion du projet a une commande qui permet de générer des matrices de taille variable.

5 Questions « théoriques »

Voici un certain nombre de questions qui peuvent vous guider dans votre travail.

- Où se concentre l'essentiel du temps d'exécution ?
- Combien d'opérations arithmétiques sont nécessaires à l'exécution de l'algorithme, en fonction de N, n et nz ? (on peut aussi introduire la densité de la matrice, ou le nombre moyen d'éléments par ligne).
- Quelle est la valeur optimale de n ? Sous quelle(s) hypothèse(s) ?

1. `https://math.nist.gov/MatrixMarket/mmio-c.html`

— Dans le cas d'une version MPI avec distribution 1D de la matrice, quelle quantité de données doit entrer/sortir de chaque noeud lors de chaque itération ? En déduire une borne inférieure sur le temps d'exécution.
— Quel est le goulet d'étranglement ? Processeur ? Mémoire ? Réseau ?
— Mêmes questions avec une distribution 2D de la matrice.
— Quelles parties du calcul peuvent potentiellement conduire à un déséquilibrage de charge ?
— Quel moyen simple permettrait de l'éviter avec probabilité élevée ?
— Si $p < 2^{30}$, combien de fois peut-on effectuer l'opération $a+ = x * b$ *sans réduire modulo* p et sans que cela ne dépasse d'un entier 64 bits ? (MPI n'implante pas la réduction modulo p).
— Quelle est l'intensité arithmétique des différentes étapes du calcul ? Essayez de tracer un ou plusieurs diagramme(s) *roofline*.
— Quel intérêt flagrant y a-t-il à utiliser la combinaison MPI + OpenMP si la matrice est très grosse ?
— Si le calcul est interrompu, quelles données sont nécessaires pour redémarrer là où il s'est arrêté ?

6 Travail à effectuer

Il y a plusieurs « sous-tâches » que vous pouvez accomplir plus ou moins dans le désordre.

1. Exploration des performances du code séquentiel (*hots spots*, etc.).
2. Parallélisation avec MPI uniquement sur un *cluster*.
3. Parallélisation avec OpenMP uniquement sur une machine multi-coeurs.
4. Parallélisation avec MPI + OpenMP : on lance un seul processus MPI par noeud et on fait du multi-thread à l'intérieur.
5. Optimisation : le code séquentiel fourni peut être amélioré de plusieurs manières.
6. *Checkpointing* : il faut que si le calcul parallèle s'arrête (panne réseau, plantage, coupure electrique, ...) , on ne soit pas obligé de tout recommencer depuis le début. Pour cela, une solution consiste à sauvegarder périodiquement des *checkpoints*, et de pouvoir repartir du dernier checkpoint. Votre implantation parallèle devra sauvegarder un *checkpoint* chaque minute.
7. Localité des données : les performances en FLOPs du produit par une matrice creuse sont toujours mauvaises car les accès mémoires sont irréguliers. Ceci peut éventuellement être un peu amélioré.
8. Challenge : résolution de gros systèmes de benchmark.

Vous noterez par ailleurs les points suivants :

1. *Quoi que vous fassiez*, vous **devez** :
   (a) *Mesurer* les performance obtenues (notamment l'accélération atteinte par rapport au code séquentiel de départ). Se contenter de mesures sur de toutes petites matrices est une erreur *majeure*.
   (b) *Commenter* ces résultats : sont-ils bons ou pas ? S'ils sont mauvais, pourquoi ? En est-on réduit à émettre des hypothèses ou bien peut-on les confirmer par une expérience ? Pouvait-on prévoir les performances obtenues ?

2. En aucun cas la matrice ne doit être entièrement chargée depuis le système de fichiers par *tous* les processus à la fois : ceci saturerait le serveur de fichier.
3. Pour le checkpointing : il suffit de conserver le *dernier* checkpoint. Il faut cependant faire attention au fait que ça peut planter *pendant* son écriture (indice : dans les OS POSIX, l'appel système `rename` peut remplacer un fichier de manière atomique).

Il n'est pas obligatoire de tout faire pour avoir la moyenne. Cependant, nous attendons qu'à la fin du projet vous ayez réalisé :

1. Parallélisation MPI pur,
2. Parallélisation MPI + OpenMP,
3. Comparaison des performances entre les deux.

Pour viser la note maximale, vous mettrez en oeuvre le plus d'optimisations possibles.

7 Travail à remettre

Le projet est initialement à rendre en *deux fois* :
— Un premier rendu le 18 mars.
— Un deuxième rendu (final) le 23 mai.
Lors de chaque rendu, vous devrez remettre le code source, sous la forme d'une archive `tar.gz` compressée et nommée suivant le modèle `projet_HPC_nom1_nom2.tar.gz`. L'archive ne doit contenir aucun exécutable, et les différentes versions devront être localisées dans des répertoires différents. Chaque répertoire devra contenir un fichier `Makefile` : la commande `make` devra permettre de lancer la compilation.

Votre propre code doit compiler sans avertissements, même avec les options `-Wall -Wextra`. Nous savons que la compilation de `mmio.c` en déclenche. Ne cherchez pas à les corriger.

Un `Makefile` situé à la racine de votre projet devra permettre (avec la commande `make`) de lancer la compilation de chaque version.

Vous devrez écrire un rapport au format `PDF`(de 5 À 10 pages, nommé suivant le modèle `rapport_HPC_nom1_nom2.pdf`) présentant vos algorithmes, vos choix d'implantation (sans code source), vos résultats (notamment vos efficacités parallèles) et vos conclusions. Ce rapport ne doit pas contenir de capture d'écran.

8 Quelques précisions importantes

— Le projet est à réaliser par binôme.
— Vous **devez** respecter les conditions d'utilisation de Grid 5000. En particulier, les « grosses »expériences doivent être menées la nuit ou le week-end.
— Les rendus se feront sur Moodle.
— En cas d'imprévu ou de problème technique commun, n'hésitez pas à nous contacter pour que nous puissions vous proposer une solution ou une alternative.