Résolution de systèmes linéaires creux par la méthode du gradient conjugué

Charles Bouillaguet & Lilia Ziane Khodja

11 mars 2020

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 $Ax = b$, où A est une matrice *creuse réelle symétrique définie positive*. Dans toute la suite, n désigne la taille du système.

Ce programme utilise la *méthode du gradient conjugué*. Il s'agit d'une (ancienne) méthode *itérative* : le calcul se fait principalement en calculant des produits matrice-vecteur avec la matrice A. L'avantage de cette famille d'algorithmes, c'est qu'une fois que la matrice A 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 A !

Une matrice (symétrique) creuse est principalement décrite par sa taille n et 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 $0 + x$.

De plus, dans le cas des matrices symétriques, il est possible de ne stocker que le triangle sous la diagonale, par exemple, et de reconstituer le triangle supérieur par transposition. La plupart des formats de stockage de matrices creuses utilisent cette astuce.

Ce document ne décrit pas précisément la méthode du gradient conjugué. L'idée générale consiste à remarquer que si A est une matrice symétrique définie positive, alors la forme quadratique $\mathbb{R}^n \to \mathbb{R}$ donnée par $f(x) = \frac{1}{2}x^tAx - x^tb$ admet un unique minimum, et ce minimum est la solution de $Ax = b$. En effet, en « dérivant » on trouve que $\text{grad}f = Ax - b$. Du coup, partant d'un point arbitraire de $\mathbb{R}^n$, il suffit de « suivre la pente » (indiquée par $\text{grad}f$ pour atteindre le minimum de f. La méthode garantit de plus qu'à chaque étape, on « avance » dans une direction orthogonale à toutes celles utilisées précédemment (par rapport au produit scalaire défini par $(x, y) \mapsto x^tAy$).

Vous pouvez consulter un certain nombre de pages Wikipédia sur le sujet qui sont assez bonnes (notamment : « Conjugate gradient method », « Preconditioner », « Sparse matrix », ...).

2 Description du code séquentiel

Le code séquentiel est constitué de trois fichiers : `cg.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. Le « vrai » code est dans `cg.c`. Il charge une matrice creuse A au format MatrixMarket depuis un fichier, génère un vecteur b puis résout $Ax = b$ en se chronométrant. Par défaut il affiche b sur la sortie standard quand il a fini.

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}). Ce format est très pratique pour les entrées/sorties, mais moins pour les calculs. Du coup, le programme convertit la matrice au format CSR (*Compressed Sparse row*) : les entrées y sont groupées par lignes. Ceci permet d'accéder facilement à tous les coefficients qui sont sur une ligne donnée. La fonction suivante charge la matrice creuse depuis le (descripteur de) fichier f (qui doit être ouvert), puis effectue la conversion en représentation CSR :

```
struct csr_matrix_t * load_mm(FILE * f)}
```

Deux fonctions sont fournies pour accéder à la matrice stockée dans ce format :

```
void extract_diagonal(const struct csr_matrix_t *A, double *d);
void sp_gemv(const struct csr_matrix_t *A, const double *x, double *y);
```

La première extrait les entrées sur la diagonale de la matrice A et les écrit dans le vecteur d ($d_i \leftarrow A_{ii}$). La seconde calcule le produit matrice-vecteur ($y \leftarrow Ax$). La fonction principale qui implante le gradient conjugué est :

```
void cg_solve(const struct csr_matrix_t *A, const double *b, double *x,
              const double epsilon, double *scratch);
```

Elle n'accède pas directement à la la structure de donnée qui contient la matrice, et se sert uniquement des deux fonctions ci-dessus. Du coup elle pourrait marcher avec n'importe quelle structure de données de matrice creuse. Son code suit quasiment à la lettre le pseudo-code donné sur la page Wikipédia (anglais) consacrée au gradient conjugué (avec préconditionneur) [2]. La seule déviation est qu'ici nous choisissons arbitrairement $x_0 = 0$.

Enfin, quatre fonctions auxiliaires sont également fournies :

```
double wtime();
double PRF(int i, unsigned long long seed);
double dot(const int n, const double *x, const double *y);
double norm(const int n, const double *x);
```

`wtime` renvoie (avec précision) le nombre de secondes écoulées depuis un certain point du passé. `PRF` est une *fonction pseudo-aléatoire* qui prend en entrée une graine sur 64 bits (`seed`) ainsi qu'un entier `i` et renvoie un `double` pseudo-aléatoire dans l'intervalle $]-1;1[$. Ceci permet de générer facilement un vecteur b pseudo-aléatoire (et donc de rendre les calculs reproductibles) à partir d'une graine en posant $b_i \leftarrow$ `PRF(i, seed)`. Enfin, `dot` calcule le produit scalaire de deux vecteurs de taille n et `norm` calcule la norme euclidienne d'un vecteur de taille n.

1. `https://math.nist.gov/MatrixMarket/mmio-c.html`
2. `https://en.wikipedia.org/wiki/Conjugate_gradient_method#The_preconditioned_conjugate_gradient_method`

Matrice	n	nz	$\approx T_{seq}$ (s)	MByte
bcsstk13	2003	83 883	0	2
cfd1	70 656	1 825 580	8	39
cfd2	123 440	3 085 406	54	65
parabolic_fem	525 825	3 674 625	29	88
tmt_sym	726 713	5 080 961	125	122
hood	220 542	10 768 436	145	222
ecology2	999 999	4 995 991	158	128
G3_circuit	1 585 478	7 660 826	172	198
thermal2	1 228 045	8 580 313	257	206
nd24k	72 000	28 715 634	620	576
Serena	1 391 349	64 131 971	171	1330
Flan_1565	1 564 794	114 165 372	2530	2390
Queen_4147	4 147 110	329 499 284	15505	6700

FIGURE 1 – Matrices de *benchmark*.

Pour accélérer la convergence, le code fourni utilise le préconditionneur de Jacobi, qui est l'un des plus simples. C'est-à-dire qu'on forme la matrice D qui contient uniquement la diagonale de A, puis qu'on utilise D comme préconditionneur (dans le pseudo-code de Wikipédia, ça donne $M = D$). Ceci améliore sensiblement la vitesse convergence vers la solution.

3 Systèmes à résoudre

Sur le papier, l'algorithme de gradient conjugué peut résoudre n'importe quel système $Ax = b$ lorsque A est symétique définie positive en moins de n itérations (lorsque A est de taille n). Mais en pratique la situation est plus compliquée. Plus précisément, ce serait vrai avec si on utilisait des flottants de précision infinie. Concrètement, on utilise des `double`, et il y a des matrices mal conditionnées pour lesquelles l'algorithme ne converge pas. La vitesse de convergence peut par ailleurs dépendre du membre droit b et/ou du choix initial de la valeur de x. Il y a des cas où l'algorithme finit par converger après *bien plus* de n itérations... Par exemple, la matrice `bcsstk36` est de taille 23 052 mais la convergence n'a lieu qu'après plus de 300 000 itérations...

Des matrices de tailles et de caractéristiques variables ont été sélectionnées pour vous (mais bien sûr, rien ne vous interdit d'en tester d'autres !), et nous avons vérifié que la convergence avait bien lieu. Elles sont listées dans la fig. 1.

Toutes ces matrices proviennent d'une collection publique de matrices creuses [3] qui servent à rudoyer des solveurs et/ou à tester d'autres programmes. Elles ont presque toutes été produites par des chercheurs ou des ingénieurs qui voulaient résoudre les systèmes linéaires correspondants pour mener leurs travaux.

Ces matrices ont un *nom*. Vous pouvez récupérer un certain nombre d'informations à leur sujet sur le site de la collection. Il vous faudra 7.3Go pour les stocker toutes de façon non-compressée (et 2.2Go pour les stocker *gzipées*). Ceci est potentiellement un problème si vous avez des quotas limités, et des solutions seront proposées sur le site de l'UE quand elles seront mises en place.

Elles sont disponibles en ligne sur le site de la collection officielle, et elles sont répliquées sur le serveur de benchmark dont il est question ci-dessous. La matrice `foobar` est disponible à

3. `https://sparse.tamu.edu/`

l'adresse :

`http://hpc.fil.cool/matrix/foobar.mtx.gz`

4 Serveur de *Benchmark*

Comme l'objectif de ce projet consiste à paralléliser un code séquentiel fourni, l'équipe pédagogique vous propose un moyen fiable et objectif d'évaluer les performances obtenues par vos programmes parallèles : le serveur de *benchmark*.

Il est disponible à l'adresse : `http://hpc.fil.cool`.

Un script python fourni (`runner.py`) se connecte au serveur, et vous envoie une graine pseudo-aléatoire qui sert à générer le membre droit b. Le script lance votre programme parallèle puis se reconnecte au serveur lorsque le calcul est fini. Le serveur peut donc mesurer combien de temps vous mettez à résoudre le système $Ax = b$. Si vous parvenez à calculer correctement la solution (le vecteur x), alors le serveur vous délivre un *reçu* qui démontre que vous avez bien résolu le système pour la matrice A en un temps donné. En cas de litige, vous pourrez toujours exhiber vos reçus.

Pour chaque calcul, il faut éditer le script `runner.py` pour y indiquer le nom de la matrice, le matériel utilisé, etc.

5 Travail à effectuer

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

1. Parallélisation avec MPI uniquement sur un *cluster*.
2. Parallélisation avec MPI + OpenMP : on lance un seul processus MPI par noeud et on fait du multi-thread à l'intérieur.
3. *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.
4. Optimisation : le code séquentiel fourni peut être amélioré de plusieurs manières (indice : dans `cg_solve`, on peut éliminer un des vecteurs).
5. Localité des données : les performances en FLOPs du produit matrice-vecteur creux sont toujours mauvaises car les accès mémoires sont irréguliers. Essayez d'améliorer un peu la localité des accès à x.
6. Vectorisation : le compilateur vectorise parfois automatiquement, mais il risque d'échouer sur la fonction `sp_gemv`. Essayez de la vectoriser.
7. Challenge : tentez votre chance sur la dernière matrice du tableau. Ceci peut vous amener à vouloir répartir entre plusieurs machines son chargement depuis le fichier (ou bien à lancer le calcul depuis un « gros » noeud).

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) sur *plusieurs* matrices.
   (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 ?
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).

Vous n'êtes pas obligé de tout faire pour avoir la moyenne. Cependant le *minimum* attendu est le suivant :

1. Pour le premier rendu : parallélisation MPI pur, parallélisation MPI + OpenMP, comparaison des performances entre les deux.
2. Pour le deuxième rendu : idem, mais en plus le *checkpointing* est obligatoire.

Pour viser la note maximale, vous mettrez en oeuvre toutes les optimisations possibles.

6 Travail à remettre

Pour chacune des deux parties, 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 demandées 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. L'archive doit aussi contenir les reçus que le serveur de *benchmark* vous aura délivrés.

Des soutenances auront lieu lors de la séance de TD/TP après le premier rendu. Prévoir 10 minutes par binôme, plus cinq minutes de questions.

À la fin du projet, lors du 2ème rendu, 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 pour les deux parties. L'analyse du comportement de vos programmes sera particulièrement appréciée. Les machines n'étant pas strictement identiques d'une salle à l'autre, on précisera dans le rapport la salle utilisée pour les tests de performance.

7 Quelques précisions importantes

— Le projet est à réaliser par binôme.
— Vous **devez** lire le document intitulé « Projet HPC : conditions d'utilisation d'OpenMPI ». Vous veillerez notamment à n'utiliser qu'une salle à la fois pour vos tests, et vous n'oublierez pas de tuer **tous** vos processus sur **toutes** les machines utilisées à la fin de vos tests.

— Attention à ne pas lancer des calculs sur des machines qui sont déjà occupées par un autre binôme : les mesures de performance ne voudraient plus rien dire. On peut examiner la situation des machines sur lesquelles on s'apprête à lancer un calcul, par exemple en lançant une copie du programme `w` sur chacune d'entre elles avec `mpiexec`. Il montre notamment le *load average* et indique qui est connecté.
— Le code de la première partie, accompagné des slides de votre soutenance au format `PDF`, est à remettre au plus tard le vendredi 17 avril 2020 à 23h59. Les soutenances de présentation de la première partie auront lieu lors de la séance de TDTP de la semaine suivante.
— Le code de la seconde partie et le rapport final (au format PDF) sont à remettre au plus tard le dimanche 17 mai 2020 à 23h59.
— SFPN : les remises se feront par email à
  — `charles.bouillaguet@univ-lille.fr`
  — `lzianekhodja@aneo.fr`
  — `gmoreau@aneo.fr`
— MAIN4 : les remises se feront par email à `charles.bouillaguet@univ-lille.fr`.
— 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.