

# High Performance Computing

Parallélisation de la Méthode de Jacobi

Plessia Stanislas

Mars 2018

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Methode de Jacobi</b>	<b>3</b>
2.1	Problème . . . . .	3
2.2	Méthode de résolution . . . . .	3
2.3	Preuve et Convergence . . . . .	4
2.4	Vecteur erreur et résidu . . . . .	6
<b>3</b>	<b>Implémentation</b>	<b>7</b>
3.1	Architecture . . . . .	7
3.2	Structures de données . . . . .	7
3.3	Librairies . . . . .	8
3.4	Tests . . . . .	8
3.5	Données . . . . .	8
3.6	Compilation et Lancement . . . . .	9
3.7	Programme principal . . . . .	9
3.8	Erreurs . . . . .	10
<b>4</b>	<b>Résultats</b>	<b>11</b>

## 1 Introduction

Dans cet article, nous allons nous attaquer à la résolution de systèmes linéaires.

Ces systèmes sont de la forme suivante :

$$\begin{cases} ax + by + cz = s_1 \\ dx + ey + fz = s_2 \\ gx + hy + iz = s_3 \end{cases}$$

On les rencontre par exemple lors de modélisation par éléments finis de la solution d'une équation différentielle, ou lors de problème d'optimisation linéaire ou de marches aléatoires.

Un exemple concret serait celui des chaînes de Markov pour représenter un mouvement aléatoire où la question serait de trouver une solution stationnaire. Dans un problème tel que celui du voyageur de commerce en utilisant la méthode du recuit simulé, le problème peut se modéliser par une telle chaîne :

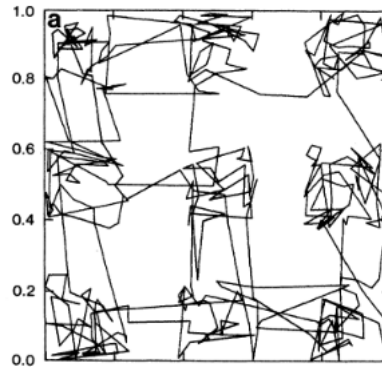


FIGURE 1 – Modélisation du problème du voyageur de commerce

## 2 Methode de Jacobi

### 2.1 Problème

Soit  $n \in \mathbb{N}$ ,  $A = (a_{i,j})_{i,j \in \llbracket 1,n \rrbracket^2}$  matrice carrée de taille  $n$ ,  $b = (b_i)_{i \in \llbracket 1,n \rrbracket}$  vecteur de taille  $n$ .

On cherche alors le vecteur  $x = (x_i)_{i \in \llbracket 1,n \rrbracket}$  tel que :

$$Ax = b \tag{1}$$

### 2.2 Méthode de résolution

Afin de résoudre ce problème, on va utiliser une méthode itérative appelée Méthode de Jacobi.

Tout d'abord, on va séparer la matrice  $A$  en deux sous matrices  $D$  et  $R$  de la manière suivante :

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix} = \begin{pmatrix} a_{1,1} & 0 & \cdots & 0 \\ 0 & a_{2,2} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{n,n} \end{pmatrix} + \begin{pmatrix} 0 & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & 0 & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & 0 \end{pmatrix}$$

Notons  $D$  la matrice diagonale, et  $R$  la matrice du reste. Comme la matrice  $D$  est diagonale (qu'on suppose à coefficients non nuls dans notre problème),  $D$  est trivialement inversible d'inverse :

$$D^{-1} = \begin{pmatrix} \frac{1}{a_{1,1}} & 0 & \cdots & 0 \\ 0 & \frac{1}{a_{2,2}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{a_{n,n}} \end{pmatrix}$$

On peut donc réécrire la formule (1) de la manière qui suit :

$$(D + R)x = b \quad (2)$$

$$\Leftrightarrow Dx = b - Rx \quad (3)$$

$$\Leftrightarrow x = D^{-1}(b - Rx) \quad (4)$$

La Méthode de Jacobi est alors une méthode itérative qui va chercher à trouver un point fixe à cette équation. On peut alors définir la suite  $x^{(k)}$ ,  $k \in \mathbb{N}$  telle que :

$$\begin{cases} x^{(0)} = \vec{0} \\ x^{(k+1)} = D^{-1}(b - Rx^{(k)}) \end{cases}$$

L'avantage flagrant de cette méthode de calcul, est que les éléments n'ont aucune dépendance verticale. Elle est donc très simple à paralléliser.

En effet, en écrivant la formule de récurrence pour un élément du vecteur  $x^{(k+1)}$ , on obtient :

$$\forall k \in \mathbb{N}, \forall i \in \llbracket 1, n \rrbracket, \quad x_i^{(k+1)} = \frac{1}{a_{i,i}} (b_i - \sum_{i \neq j} a_{i,j} x_j^{(k)}) \quad (5)$$

La seule contrainte devient alors que chaque processeur doit connaître toutes les composantes de  $x^{(k)}$  afin de pouvoir itérer, et il faut donc les communiquer, ce que nous ferons en utilisant la bibliothèque MPI.

### 2.3 Preuve et Convergence

D'après le théorème de la méthode du point fixe, on sait que la suite  $x_{n+1} = f(x_n)$  converge si la fonction  $f$  est contractante i.e  $k$ -lipshitzienne avec  $k < 1$

Notons  $C = -D^{-1}R$ ,  $d = D^{-1}b$  et

$$\begin{aligned}\mathcal{F} : \mathbb{R} &\longrightarrow \mathbb{R} \\ u &\longrightarrow Cu + d\end{aligned}$$

On veut trouver une condition nécessaire et suffisante pour que  $\mathcal{F}$  soit une application contractante. Or, on a :

Soit  $a, b$  deux vecteurs dans  $\mathbb{R}^n$ ,

$$\begin{aligned}\|\mathcal{F}(a) - \mathcal{F}(b)\| &= \|Ca - Cb\| \\ &\leq \|C\|_{\infty} \cdot \|a - b\|\end{aligned}$$

On doit donc avoir  $\|C\|_{\infty} < 1$  ce qui revient à dire que le rayon spectral de la matrice  $C$  noté  $\rho(C)$  doit être strictement inférieur à 1

On peut donc affirmer que :

$$\lim_{k \rightarrow \infty} x^{(k)} = x \Leftrightarrow \rho(C) = \rho(-D^{-1}R) < 1$$

Cherchons une condition simple suffisante pour affirmer que le rayon spectral de la matrice  $C$  soit inférieur à 1.

Soit  $\lambda$  valeur propre de  $C$  ie  $\forall y \in \mathbb{R}^n, Cy = \lambda y$

On a alors,  $\forall (i, j) \in \llbracket 1, n \rrbracket^2$

$$\begin{aligned}\|\lambda y\|_{\infty} &= \left| \sum_{j=1}^n c_{i,j} y_j \right| \\ \Leftrightarrow \lambda \|y\|_{\infty} &= \left| \sum_{j \neq i} \frac{a_{i,j}}{a_{i,i}} y_j \right| \\ \Leftrightarrow \lambda \|y\|_{\infty} &= \left| \frac{1}{a_{i,i}} \right| \cdot \left| \sum_{j \neq i} a_{i,j} y_j \right| \\ \Leftrightarrow \lambda \|y\|_{\infty} &\leq \left| \frac{1}{a_{i,i}} \right| \cdot \sum_{j \neq i} |a_{i,j} y_j| \\ \Leftrightarrow \lambda \|y\|_{\infty} &\leq \left| \frac{1}{a_{i,i}} \right| \cdot \sum_{j \neq i} |a_{i,j}| \cdot \|y\|_{\infty} \\ \Leftrightarrow \lambda &\leq \left| \frac{1}{a_{i,i}} \right| \cdot \sum_{j \neq i} |a_{i,j}|\end{aligned}$$

Une condition suffisante pour que  $\lambda < 1$  serait alors :

$$\forall (i, j) \in \llbracket 1, n \rrbracket^2 \quad |a_{i,i}| > \sum_{i \neq j} |a_{i,j}|$$

ie  $A$  est à diagonale strictement dominante

## 2.4 Vecteur erreur et résidu

Afin de mesurer la convergence de la méthode de Jacobi (et d'avoir une condition d'arrêt), on définit le vecteur erreur relative suivant :

$$e^{(k+1)} = \|x^{(k+2)} - x^{(k+1)}\|_\infty = \|C(x^{(k+1)} - x^{(k)})\|_\infty = \|C\|_\infty e^{(k)}$$

On a donc  $e^{(k)} = \|C\|_\infty^k e^{(0)}$

On a de plus le vecteur erreur :

$$\epsilon^{(k)} = \|x - x^{(k)}\|_\infty$$

$e^{(k)}$  est l'erreur relative de la méthode à l'itération  $k$ , mais comme on a pas connaissance du véritable vecteur  $x$ , on l'utilise comme référence pour le test d'arrêt.

En effet,

$$\begin{aligned} e^{(k)} &= \|x^{(k+1)} - x^{(k)}\|_\infty \\ &= \|Cx^{(k)} + d - x^{(k)}\|_\infty \\ &= \|Cx^{(k)} + x - Cx - x^{(k)}\|_\infty \\ &= \|x - x^{(k)} - C(x - x^{(k)})\|_\infty \\ &= \|(-C + I)(x - x^{(k)})\|_\infty \end{aligned}$$

On obtient alors par inégalité triangulaire :

$$e^{(k)} = \|I - C\|_\infty \epsilon^{(k)} \leq \epsilon^{(k)} + \|C\|_\infty \epsilon^{(k)}$$

Et comme on a vu précédemment la norme infinie de la matrice  $C$  est strictement inférieure à 1 on a donc :

$$e^{(k)} \leq 2\epsilon^{(k)}$$

On peut alors considérer que l'erreur relative est suffisamment proche de l'erreur réelle pour l'utiliser comme test d'arrêt.

## 3 Implémentation

### 3.1 Architecture

Le projet est divisé en plusieurs sous-dossiers

- `/lib` qui contient des bibliothèques pour manipuler les matrices et les vecteurs ainsi que les opérations basiques de manipulation de ces deux structures de données.
- `/src` qui contient le fichier *main* qui fait tourner le programme de l'algorithme de Jacobi.
- `/data` qui contient les fichiers des matrices et des vecteurs ainsi que les métadonnées contenant la taille des matrices considérées.
- `/bin` qui contient les binaires *main* et *test* afin de faire tourner le programme ou de tester les bibliothèques
- `/test` qui contient le code source du binaire de test dont l'objectif est de tester les fonctions des bibliothèques.

Le projet est de plus constitué d'un Makefile qui contient les procédures suivantes :

- *make all* (ou simplement *make*) pour compiler les bibliothèques, main et les tests
- *make test* pour compiler les tests
- *make clean* pour supprimer les fichiers objets

### 3.2 Structures de données

Dans ce projet, j'ai créé deux structures de données relativement similaires pour les vecteurs et les matrices. Chaque structure dispose d'un type complexe qui lui est associé pour des raisons d'ergonomie.

Ces structures sont contenues dans les fichiers `lib/matrix.h` et `lib/vector.h`.

La structure Matrice est composée de deux entiers non signés pour les dimensions et d'un tableau de *double* à 2 dimensions pour contenir les valeurs de la matrice.

```
typedef struct Matrix{  
    unsigned int col;  
    unsigned int rows;  
    double **matrix;  
5 }Matrix;
```

La structure Vecteur est très similaire mais n'a qu'une seule dimension :

```
typedef struct Vector{  
    unsigned int size;  
    double **vector;  
}Vector;
```

### 3.3 Bibliothèques

Ce projet est fourni avec 2 bibliothèques situées dans les fichiers C du dossier **/lib**.

Les deux bibliothèques ( Matrix et Vector ) sont analogues et possèdent des fonctions très similaires dont on remplacera le mot *structure* par la structure qui correspond (matrix/vector) :

- **build\_structure** qui alloue dans la heap la structure avec la taille souhaitée
- **display\_structure** qui affiche le contenu algébrique de la structure
- **randomize\_structure** qui génère un contenu aléatoire (pas utilisée dans le projet en soit)
- **free\_structure** qui désalloue la structure de la heap et la reset
- **read\_structure\_from\_file** qui utilise un fichier pour remplir la structure

La fonction **read\_structure\_from\_file** est la seule fonction non triviale (plusieurs sous structures de fonctionnements internes). Elle peut échouer (*cf.* partie 3.8 page 10)

```
function READ_STRUCTURE_FROM_FILE(structure, filename, first_line, size)
    Buffer of size 10000
    file ← open(filename)
    if file then
        while first_line not reached do
            buffer ← line
        end while
        build_structure(structure, size)
        if build failed then
            return error ENOMEM
        end if
        structure ← file values
        return 0
    else
        return error ENOENT
    end if
end function
```

### 3.4 Tests

Le fichier de test ne vérifie que le bon fonctionnement des bibliothèques en générant des matrices et vecteurs de manière aléatoire et en utilisant un fichier de test.

### 3.5 Données

Pour le problème, trois fichiers de données différents sont nécessaires :

- Un fichier de métadonnées (metadata.txt) qui contient les dimensions du problème
- Un fichier de matrice (matrix.txt) qui contient les données de la matrice  $A$  séparées par des espaces
- Un fichier de vecteur (vector.txt) qui contient le vecteur  $b$  du problème



### 3.6 Compilation et Lancement

Le Makefile n'est pas indispensable à la compilation du projet. Les bibliothèques doivent être compilées puis liées au main (cela peut se faire en utilisant seulement gcc car les bibliothèques n'utilisent pas MPI, mais mpicc est recommandé pour raison de compatibilité).

Le programme main est compilé en utilisant mpicc, le compilateur C de la bibliothèque MPI. Les différents flags de compilations sont (viennent de gcc) :

- `-Wall` qui active tout les messages de warning
- `-g` qui active les flags de debug
- `-O3` qui active toutes les optimisations de compilations

Pour lancer le programme :

```
mpirun -n n_proc bin/main [-v] [-h?]
```

### 3.7 Programme principal

Le programme principal est composé de `main.c` et `main.h`.

Celui ci à 3 fonctions :

- `main` qui initialise le programme et résoud le système linéaire
- `product_vector_matrix` qui effectue le produit final en mode verbose pour vérifier le résultat
- `usage` qui affiche l'usage de la Command Line Interface

Le programme peut recevoir les argument `"-v"` pour passer en mode verbose ou `"-h/-?"` pour afficher l'usage.

Les fichiers de données sont hardcodés dans le code source, mais cette partie est facilement modifiable pour pouvoir spécifier le chemin des fichiers en question.

Ce programme effectue la méthode de Jacobi en asynchrone utilisant :

- `MPI_Isend` qui envoi des données de manière asynchrone
- `MPI_Irecv` qui reçoit des données de manière asynchrone
- `MPI_Wait` qui permet d'attendre la réalisation d'un requête asynchrone (send ou recv)

Le principe de communication est le suivant :

Les processeurs disposent d'une variable `mtx` qui contient une sous-matrice et `vect` le sous-vecteur de  $b$  associée à la sous matrice suivant un découpage par lignes distribué de manière équilibré sur ceux-ci.

Par exemple, une matrice  $3 \times 3$  sur 2 processeurs serait découpées avec la distribution de lignes suivantes :

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}$$

Ensuite, chaque processeur dispose d'une variable `global_result` qui contient  $x^{(k)}$  à l'étape  $k$  et de `local_result` qui après itération contiendra les sous vecteurs de  $x^{(k+1)}$  à l'étape  $k$ .

Il y'a également un tableau `continue_iterate` qui pour chaque ligne de chaque processeur contient un booléen qui permet de décider si l'on continue d'itérer sur la ligne en question.

Au début de chaque itération, les processeurs envoient la valeur locale de `global_result` dans lequel on a stocké l'itération que l'on viens de terminer, et l'on complète cette variable en recevant la valeur locale des autres processeurs.

Les appels à la fonction `MPI_Irecv` sont suivi généralement d'un appel à `MPI_Wait` qui va attendre que la reception soit effective car elle est nécessaire à l'opération qui suit.

En revanche nous ne mettons pas de `MPI_Wait` après `MPI_Isend` car les données ne sont mutés qu'après réception de la données suivante, et la requête d'envoi sera donc terminée à ce moment.

Ensuite on applique la formule de récurrence sur ces nouvelles données que l'on stock dans `local_result`.

Pour le test d'arrêt, comme nous sommes en norme infinie, on peut calculer l'erreur relative de chaque composante  $e_i^{(k)} = x^{(k+1)}[i] - x^{(k)}[i]$

On va alors tester chaque ligne de chaque processeur et ne continuer l'itération de celles-ci que si la convergence est atteinte :

$$e_i^{(k)} \leq 1.10^{-6}$$

Pour cela, on modifie à l'intérieur de chaque processeur la tableau de booléens `continue_iterate`, puis une fois que le tableau ne contiens que des `false`, on envoi la valeur false au processeur root pour que celui ci fasse un masque binaire de tout les autres pour décider de stoper la boucle principale. La limite d'itérations fixée à 500 est arbitraire et pourrait certainement être abaissée.

### 3.8 Erreurs

L'architecture de code de ce programme suit les standard de développement basés sur le status d'une opération. En effet, dans ce programme beaucoup de fonctions sont susceptibles d'échouer (la manipulation de fichier, l'allocation mémoire, ...).

Dans cette optique, les fonctions des librairies utilisent des pointeurs sur les variables, qui sont alors mutées en place, et les fonctions renvoient un code stipulant la réussite ou l'échec de l'opération en utilisant les code d'erreurs standards de la librairie `errno.h`.

Les codes utilisés ici sont :

- Code Erreur 2, `ENOENT` : "No such file or directory"
- Code Erreur 12, `ENOMEM` : "Out of memory"
- Code Erreur 22, `EINVAL` : "Invalid argument"

## 4 Résultats