POLYTECH LYON

Rapport du T.P. :
5A. Mathématiques appliquées et Modélisation 2017-2018 (option C.O. et B.D.P.)

Calcul haute performance :
MPI Méthodes numériques pour le C.H.P.

Groupe de travail : ABOUADALLAH Mohamed Anwar,

École Polytechnique Lyon
15 Boulevard Latarjet 69622 Villeurbanne

Table des matières

1 Introduction :

1.1 Parallélisme :

1.1.1 L'idée du calcul parallèle

Il existe deux types de programmations : Séquentiel et Parallèle. Dans le modèle séquentiel, un programme est exécuté par un unique processus, malheureusement, divers champs d'application tels la mécanique des fluides ou l'intelligence artificiel imposent une limitation de la puissance de calcul d'un processeur et de la capacité mémoire. D'où l'idée du calcul ou programmation parallèle qui consiste à associer plusieurs processeurs pour traiter un même problème.
En effet, le parallélisme présente plusieurs avantages par rapport au calcul séquentiel parmi eux on peut citer :

- ★ La réduction du temps de réponse Augmentation de la taille des problèmes traités Utilisation des ressources existantes.
- ★ La complexification des modèles de simulation.
- ★ Développement structuré chaque modèle simulé ne connaît que les champs de données et les solveurs qui lui sont nécessaires.
- ★ Une meilleure gestion de la mémoire : effort sur la localisation des données.

Durant cours de calcul haute performance, nous allons étudier trois logiciels très utilisés dans le calcul parallèle : MPI (Message Passing Interface), le openMP (Multithreading) et PetSci. Ainsi, durant le premier T.P., nous allons commencer par utiliser MPI et expliquer le fonctionnement du transfert des messages. Nous allons aussi étudier la question de l'efficacité de l'implémentation en Mpi.

1.2 Objective du T.P. :

Le projet suivant consiste à étudier l'efficacité [1] de l'implémentation M.P.I./C++ de deux algorithmes parallèles : Schwartz avec et sans recouvrement puis la mise en œuvre de l'accélération de Aitken pour accélérer la convergence de la méthode de Schwartz.
Premièrement nous allons commencer par mettre en œuvre une méthode de décomposition de type Schwartz pour résoudre un système linéaire de type $Ax = b$ où les dépendances de la matrice A n'ont pas une structure régulière. Par la suite nous allons étudier l'efficacité de cette méthode. Dans un second temps, nous allons mettre en œuvre la méthode de Schwartz avec recouvrement. Enfin, au cas où notre algorithme de Schwartz appliqué à un problème converge, nous allons accélérer cette convergence en ajoutant l'accélération d'Aitken.

1.3 Message passing interface :

1.3.1 Introduction :

M.P.I. (Message passing interface) est une norme conçue en 1993-94, qui définit une bibliothèque de fonctions, utilisable avec les langages C, C++ et Fortran. Elle permet d'exploiter des ordinateurs distants

1. **Efficacité :** $\frac{T_1}{pT_p}$

ou multiprocesseur par passage de messages. Elle fournit un moyen efficace de communication en parallèle entre une collection distribuée de machines. Cependant, pas toutes les implémentations MPI tirent parti de la mémoire partagée lorsqu'elle est disponible entre processeurs, le principe de base étant que deux processeurs qui partagent une mémoire commune peuvent communiquer les uns avec les autres plus rapidement grâce à l'utilisation du support partagé qu'à travers autres moyens de communication [**?**].

1.3.2 Comment se fait le transfert des messages :

a-Les six fonctions de bases de MPI : La norme M.P.I impose six fonctions de base :

- ★ *MPI_Init* Permet d'initialiser MPI.
- ★ *MPI_Comm_Size* retourne le nombre de processeurs
- ★ *MPI_Comm_Rank* retourne le numéro de processeur
- ★ *MPI_Send* Envoi un message
- ★ *MPI_Recv* Reçoit un message
- ★ *MPI_Finalize* Termine MPI

Ainsi, à partir de ses six fonction de base on pourrait ajuster les communications entre les processus dans le but de bien distribuer nos calculs ce qui nous permettrait de réduire le temps de calcul.

2 Partie I : Schwartz sans recouvrement

2.1 Partie théorique :

2.1.1 Introduction et notations :

L'algorithme de Schwartz sans recouvrement est une méthode de décomposition de domaine, c'est à dire une méthode qui permet de résoudre plus efficacement ce problème algébrique en le parallélisant.
Ainsi, l'objective de la première partie se résume à prendre $A = a_{ij} \in \mathbb{R}^{nxn}$ une matrice inversible creuse, on prend $b \in \mathbb{R}^n$ et on va essayer de résoudre de manière itérative sur p processeur le problème :

$$Ax = b$$

Pour cela on se donne :

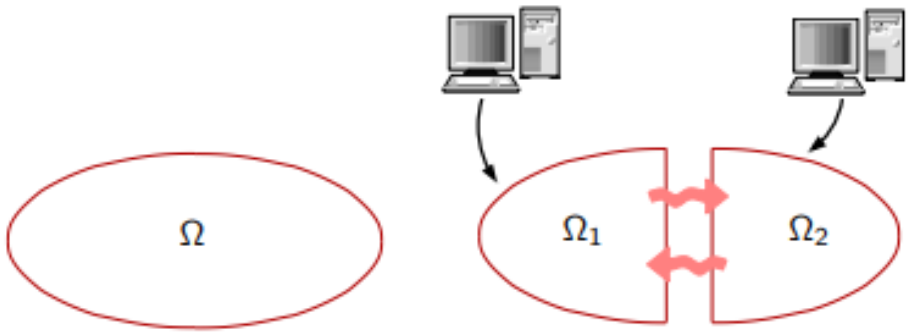
- ★ $\cup_{i=0}^{p-1} I_i = I$ une partition des processeurs tels que $\forall i \neq j, I_i \cap I_j = \emptyset$ avec I_i un sous domaine.
- ★ $A_{I_i} = A(I_i, 1:n)$ l'ensemble des lignes de A appartenant au sous domaine I_i.
- ★ $b_i = b(I_i, 1:n)$ l'ensemble des composantes de b appartenant au sous domaine I_i.
- ★ Pour chaque processus i, les matrices $A_{i,i} = A(I_i, I_i)$ la matrice de dépendances locales et $A_{i,e} = A(I_i, I_e)$ l'ensemble des ligne dans les autres processeurs (matrice contenant les dépendances non locales)
- ★ $x_{i,i} = (x_{i,i}, x_{i,e})$ avec les indices i,i correspondant aux indices I_i gérés par le processus i et i,e les indices correspondants à $I_{i,e}$ gérés par les autres processus.

En supposant que $A_{i,i}$ est inversible $\forall i$, l'itération $n+1$ de l'algorithme de Schwarz s'écrit :

$$\boxed{x_{i,i}^{n+1} = (A_{i,i}^{-1})(b_i - A_{i,e} x_{i,e}^n)}$$
$$\boxed{x_{i,e}^{n+1}(I_{i,e} \cap I_j) = x_{j;j}^{n+1}(I_{i;e} \cap I_j), \forall j \text{ tel que } I_{i,e} \cap I_j \neq \emptyset}$$

2.1.2 Explication :

Tout d'abord, nous allons commencer par parler de la décomposition du domaine. Soit γ le domaine ou se trouve un problème complexe. Si l'on souhaite résoudre notre problème de manière efficace et dans un temps acceptable alors nous pouvons le décomposer sur l'ensemble des processeurs dont on dispose : Ainsi si l'on dispose de deux processeurs on peut le décomposer de cette manière :



2

2. Image extraite de l'Introduction aux méthodes de Schwarz Véronique Martin.

Ce qui nous permettrait de réduire le temps de calcul considérablement car d'après la loi d'Amdahl :

$$T_p = sT_1 + \frac{(1-s)T_1}{p}$$

Maintenant, on passe à la méthode de Schwartz et pour faciliter son explication, nous allons utiliser une matrice $A \in \mathbb{R}^{2x2}$ par ce fait on prendra $b \in \mathbb{R}^2$ et nous suivront les mêmes notations qu'on utilisé durant notre implémentation [3].

$$\boxed{\begin{pmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \end{pmatrix}}$$

Dans cet exemple, notre matrice se composera en deux blocs et elle sera géré par deux processeurs (0 et 1) : Ainsi, comme on sait que $b_0 = a_{0,0}x_0 + a_{0,1}x_1$, alors on a besoin des valeurs gérées par le processeur 1. Chose qui nous contraint à faire communiquer les processeurs entre eux, d'où l'utilisation de M.P.I. dans ce T.P.
De plus comme A une matrice creuse et que nous utilisons p processeurs alors nouas aurons besoins de listes pour stocker les différents indices de A. Le reste des variables qu'on va mobiliser sera expliquer dans les différentes sous-sections qui vont suivre.

2.1.3 Algorithme :

Dans cette section, nous utilisons les mêmes notation présentées auparavant et nous allons rappeler qu'en supposant $A_{i,i}$ est inversible $\forall i$, l'itération $n+1$ de l'algorithme de Schwarz s'écrit :

$$\boxed{x_{i,i}^{n+1} = (A_{i,i}^{-1})(b_i - A_{i,e}x_{i,e}^n)}$$
$$\boxed{x_{i,e}^{n+1}(I_{i,e} \cap I_j) = x_{j;j}^{n+1}(I_{i;e} \cap I_j), \forall j \text{ tel que } I_{i,e} \cap I_j \neq \emptyset}$$

Ainsi notre algorithme sera du type itérative et devra converger. De plus, nous allons programmer une classe "sousdomaine" contenant une implémentation de cette itération en trois méthodes distinctes : Solve() pour faire notre solveur et donc notre résolution, Update() pour mettre à jour nos x_{ie} et enfin converge() pour tester la convergence de notre méthode.

2.2 Implémentation en MPI/C++

Durant cette partie nous allons mettre en œuvre une méthode de décomposition de domaine comme on l'avait expliqué auparavant. Notre but étant la résolution du système $Ax = b$ où les dépendances des données de la matrice A n'ont pas une structure régulière.
Ainsi, nous allons implémenter une classe "**sousdomaine**" respectant les charte de la classe "**ij**" mise à notre disposition.

3. indices commençant par 0.

2.2.1 Variables utilisées :

Tout d'abord nous allons présenter les différentes variables qu'on va utiliser celle explicité2.1.1 en plus de quelques variables qu'on va expliciter maintenant par l'intermédiaire de mon code commenté :

Listing 1 – variables utilisées

```
vector<double> xii; // vect sol local
vector<double> xiip; // ce vecteu permet de stoker les xii l'iteration
vector<double> xie; // vect cont les dependances non locales
vector<double> bi; // second membre local
map<int,std::list<int>> IR; //recuperer les valeurs dont le processeu
map<int,std::list<int>> IS;  //recuperer les valeurs dont le processe
map<int,int> GL; // passage global -->locale
map<int,int> GLii; // passage global --> locale pour les ii
map<int,int> GLie; // passage global --> locale pour les ii
int me; //no proc
int p; // nb de proc
int N, //taille de la matrice A
int Ninv;  //taille de la matrice Aincerse
```

2.2.2 Constructeur :

Pour cela, comme pour toute classe nous allons commencer par un constructeur [4] : Pour notre constructeur on n'aura besoin que des indices

Listing 2 – constructeur

```
sousdomaine(){
this->N=0;
this->Ninv=0;
// ouverture du fichier contenant la matrice
ifstream matriceA("Matrice_A.txt", ifstream::in);
matriceA >> this->N;
```

2.2.3 Lecture de la matrice :

Ensuite nous allons faire la lecture de notre matrice A. Pour cela on commence par gérer les communication entre les processeurs :

Listing 3 – parallélilisation via MPI

```
int ier;
ier = MPI_Comm_rank(MPI_COMM_WORLD, &me);
ier = MPI_Comm_size(MPI_COMM_WORLD, &p);
int iinit=me*N/p;
int ifinal=(me+1)*N/p-1;
```

4. Les librairies qu'on a utilisé ne seront pas exposée

```
6         int i, //indice des lignes
7         int j; //indice des colones
8         double a; //nos valeurs aij
```

Avec :

★ **MPI_Comm_rank(MPI_COMM_WORLD, &rank)** :Permet d'obtenir notre rang.

★ **MPI_Comm_size(MPI_COMM_WORLD, &p)** : Permet d'obtenir le nombre de taches.

Listing 4 – Lecture de la matrice

```
1         // lecture de la matrice A
2         // matrice inverse A
3         // vecteur b
4         if (matriceA){ // tester l'existance de A
5         while(matriceA.good()){
6         matriceA >> i >> j >> a;
7         if ((iinit<=i) && (i<=ifinal)){
8         if ((iinit<=j) && (j<=ifinal)){
9         Aii[ij(i,j)]=a;          // c bon marche quand on met le ij.h
10        Ii.push_back(j); // on rajoute l'indice j dans la liste Ii
11        } else {
12        // pour Aie on stock l'indice dans la liste Ie (comme pour les Aii)
13        Aie[ij(i,j)]=a;
14        Ie.push_back(j);
15        }
16        }
17        }
18        }     else { cout << "erreur dans la lecture" <<endl;}
```

Ensuite on effectue la lecture du vecteur b. En ce qui me concerne j'ai juste crée un vecteur b avec des valeurs aléatoire pour tester :

Listing 5 – Lecture de la matrice

```
1
2
3         ifstream fluxb("Vecteur_b.txt", ifstream::in);
4
5         double ib; // les indice de b
6         double bb; // les valeurs de b
7         if (fluxb){
8         while(!fluxb.good()){
9         fluxb >> ib >> bb;
10        if ((iinit<=ib) && (ib<=ifinal)){
11        bi.push_back(bb);
12        }
13        }
14        }else { cout << "erreur dans la lecture" <<endl;}
```

Ensuite, comme le contenu de nos I_i et I_e ne sera pas trié par ordre croissant et contiendra des doublons, nous allons trier et supprimé les doublons par l'intermédiaire de **sort()** et **unique()**

```
1        Ii.sort();
2        Ii.unique();
3        Ie.sort();
4        Ie.unique();
```

Remarque 1 *Jusqu'à l'instant on travail toujours avec la numérotation globale, maintenant on va convertir cette numérotation en locale.*

2.2.4 Passage de la numérotation locale à la globale :

Maintenant, on passe à la numérotation locale à partir de la globale : Pour cela on commencera par parcourir les liste Ii et Ie triées et nettoyer par l'intermédiaire d'un itérateur et un compteur. Le premier permet de faire la boucle for tandis que le second permet de prendre le numéro globale et le transformer en locale, ainsi si notre liste commence avec un 5 alors notre compteur fera le passage de 5 à 1 et on remet le compteur à zéro comme ça il n y aura pas de problème.

Listing 6 – Passage de la no locale à globale

```
1        int tailleTi = Ii.end();
2        for(it=Ii.begin(); it!=tailleIi; it++){
3        GLii[*it]=compt;
4        compt ++;
5        }
6        compt=0; // on reinitilie le compteur a 0 pour les Ie et on fait de m
7        int tailleTe = Ie.end();
8        for(it=Ie.begin(); it!=tailleIe; it++){
9        GLie[*it]=compt;
10       compt ++;
11       }
```

Ensuite on fera de même pour la matrice/map Aii, en effet on va parcourir ses éléments dans le but de remplir une nouvelle matrice/map locale.

Listing 7 – Passage de la no locale à globale pour Aii

```
1
2        int taillexii = xii.size();
3        invAii=new double[taillexii*taillexii];
4        for (int zi=0; z<taillexii*taillexii; z++){
5        invAii[z]=0;
6        }
7
8        ipiv = new int[taillexii];
9        rhs = new double[taillexii];
10
```

```
11
12          {map<ij , double> AiiLoc;
13          map<ij , double>::iterator iterA;
14          int nvij;
15          //
16          for (itA=Aii.begin(); iterA!=Aii.end(); iterA++){
17
18          AiiLoc[ij(GLii[itA->first.i],GLii[itA->first.j])]=itA->second;
19
20
21          nvij=GLii[itA->first.i]*taillexii+GLii[itA->first.j];
22
23
24          invAii[nvij]=itA->second;
25          }
26          Aii.insert(AiiLoc.begin(), AiiLoc.end());
27          }
```

Maintenant qu'on est passé à la numérotation locale, il est temps de passer à la méthode solve().

2.2.5 Methode solve() :

Cette méthode nous permettrait d'implémenter la résolution de A_{ii} via $x_{ii} = (b_i - A_{ie}x_{ie})$ Pour cela on va commencer par faire appel à degetrf, une subroutine de la librairie lapack (linear algebra package) qui permet d'effectuer

Listing 8 – Methode solve

```
1           // ici on va commencer a implementer les methodes solves et //update,
2
3           LAPACKE_dgetrf(LAPACK_ROW_MAJOR, taillexii , taillexii ,
4           &invAii[0], taillexii , ipiv);
5           //methode solve
6           void solve(){
7
8           vector<double> rst(bi);
9           for(map<ij ,double> :: iterator itA2=Aie.begin(); itA2 != Aie.end(); i
10          // On calcule bi - Aie*xie
11          rst[itA2->first.i]=rst[itA2->first.i]-itA2->second*xie[itA2->first.j];
12          }
13
14          {int compt=0; // on reinitia le compt a 0 pour et on met un itarateur
15          for (vector<double> :: iterator itres=rst.begin(); itres!=rst.end();
16          rhs[compt++]=*it;}
17          }
18
19          LAPACKE_dgetrs(LAPACK_ROW_MAJOR, 'N', xii.size(), 1, invAii, xii.size
20          for (int j=0; j<N/p;j++){
```

```
21          xii[j]=rhs[j]; //on remplit les xii avec notre reso
22          }
23          }
```

Explication détaillée de la méthode :

Tout d'abord on utilise la subroutine DGETRF qui effectue une factorisation LU de notre matrice A_{ii} dans $invA_{ii}$. Cette subroutine est appelée de la sorte : SUBROUTINE DGETRF(M, N, A, LDA, IPIV, INFO) mais comme dans ce t.p. on est en présence d'une matrice creuse on travail via les lignes, il est plus logique d'utiliser $LAPACK_ROW_MAJOR$.
Ensuite comme notre méthode ne retourne aucune valeur on va l'implémenter via le mot clés "void ", ensuite nous calculerons $b_i - A_{ie} * x_{ie}$ en bouclant sur l'itérateur itA2 qui parcourra notre map Aie.
Ensuite, on mettra les résultats calculés dans notre vecteur rhs puis on résoudra notre système par DGETRS, une autre subroutine appartenant à la bibliothèque lapack permettant de résoudre le système $AX = b$.

3 Résultats

Par manque de temps, les sections suivantes sont vides, je compte les terminer au début de la semaine prochaine.

4 Conclusion et bibliographie :