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 Motivation :

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'applications 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 avons commencer par utiliser MPI, ensuite nous avons déployer Petsci et enfin nous allons mobiliser OpenMp et expliquer le fonctionnement du transfert des messages. Nous allons aussi étudier la question de l'efficacité de l'implémentation en OpenMp.

1.2 Objective du T.P. :

Le projet suivant consiste à étudier l'efficacité [1] de l'implémentation OpenMp/C++ de deux algorithmes parallèles : Schwartz avec et sans recouvrement puis essayer si possible de mettre en œuvre l'accélération de Aitken à dessein d'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 OpenMp :

1.3.1 Introduction :

OpenMP (Open Multi-Processing) est une interface de programmation destinée au calcul parallèle sur des architectures à mémoires partagées, c'est à dire une même zone de mémoire vive accédée par plu-

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

sieurs processus. Cette API est prise en charge par de nombreuses plateformes, incluant les plateformes GNU/Linux, OS X et Windows et des langages de programmation C, C++ Python et Fortran.
Il se présente sous la forme d'un ensemble de directives, d'une bibliothèque logicielle et de variables d'environnement.

1.3.2 OpenMp Vs Mpi :

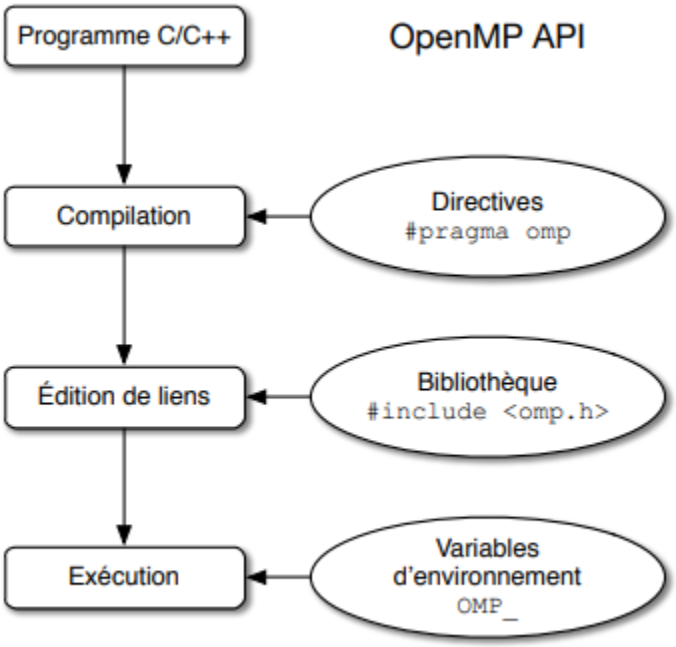
Comme dit auparavant nous avons utilisé Mpi durant notre premier projet qui est très similaire à celui là. Avant de commencer à expliquer les concepts théoriques de notre projet, nous nous permettrons d'expliquer les différences entre ces deux Api.
Voici un tableau qui pourrait résumer quelques unes d'entre elles :

OpenMp	MPI
OpenMP cible uniquement le système de mémoire partagée	MPI cible à la fois la mémoire distribuée et la mémoire partagée
Un seul parallélisme basé sur les taches.	Deux types d'approches possibles : les processus et les threads
Dépends de l'implémentation	Le temps système nécessaire à la création d'un processus est unique
Difficulté : Moyenne	Difficulté : Un peu difficile

2 Fonctionnement de OpenMp :

2.1 Principe :

OpenMp est basé sur une répartition des charges de calcul sur divers processus légers appelés threads réduisant ainsi considérablement le temps d'exécution du programme. Ainsi une tâche en séquentiel est exécutée par le thread maître : thread de rang 0.



2.2 Directives :

Ici nous allons présenter quelques fonctions de bases de OpenMP :

- **omp_get_num_threads() :** retourne le nombre total de threads utilisés
- **omp_set_num_threads(int) :** spécifie un nombre de thread dans une région parallèle
- **omp_get_max_threads() :** retourn un entier supérieur ou égale au nombre de threads disponibles.
- **omp_get_thread_num() :** retourne le numéro du thread courant
- **om_get_num_procs() :** retourne le nombre de threads disponibles sur la machine
- **omp_in_parallel() :** retourne F en séquentiel et T en parallèle
- **omp_get_wtime() :** retoune le temps écoulé d'un certain point

Ainsi, on peut appeler les fonction via les mots clés usuels de c++ exemples : **int omp_get_max_threads() où double omp_get_wtime( ) ;**

2.3 Utilisation :

Á la différence de Mpi et Petsci, OpenMp nécessite une manière différente pour définir une région parallèle :

```
1     # pragma omp parallel
2     {
3     ....
4     }
```

Ainsi, pour écrire un programme qui parralèlise "**Hello World**", il faudra faire tel cela :

```
1     #include <omp.h>
2     #include <stdio.h>
3     #include <stdlib.h>
4     int main (int argc, char *argv[]) {
5
6     int nthreads, tid;
7
8     #pragma omp parallel private(nthreads, tid)
9     {
10
11
12    tid = omp_get_thread_num();// numero du Thread
13    printf("Hello World de la part des threads = %d\n", tid);
14
15    if (tid == 0)
16    {
17    nthreads = omp_get_num_threads();
18    printf("Number of threads = %d\n", nthreads);
19    }
20
21    }
22
23    }
```

3 Partie I : Schwartz sans recouvrement

3.1 Partie théorique :

3.1.1 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 :

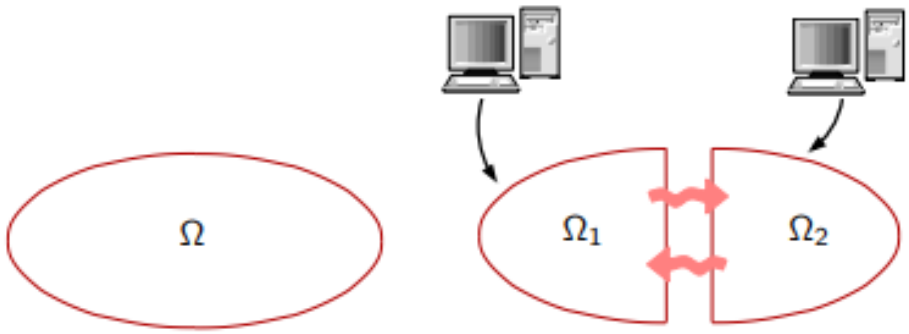
- ★ $\cup_{i=0}^{p-1} I_i = I$ une partition des processeurs tels que $\forall i \neq j, I_i \cap I_j = \emptyset$ avec I_i un sous domaine.
- ★ $A_{I_i} = A(I_i, 1:n)$ l'ensemble des lignes de A appartenant au sous domaine I_i.
- ★ $b_i = b(I_i, 1:n)$ l'ensemble des composantes de b appartenant au sous 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 :

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

3.1.2 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.

3.1.3 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.

4 Implémentation OpenMp/C++ :

4.1 Notre header :

Afin d'organiser notre projet dans l'esprit Cpp, on commencera par créer un simple header pour que notre code soit lisible. Cela permettrait de bien présenter nos méthodes et fonctions :

3. indices commençant par 0.

Listing 1 – variables utilisées

```
1   #ifndef SOUSDOMAINE_H_
2   #define SOUSDOMAINE_H_
3   #include<string>
4   #include<map>
5   #include "ij.h"
6   #include<complex>
7   #include <vector>
8   #include <list>
9   #include <mpi.h>
10  #include <mkl.h>
11  #include <cmath>
12  using namespace std;
13
14  class sousdomaine
15  {
16  public :
17  int n, p;
18  map <ij,double> Aie;
19  double **Aii;
20  double **bii;
21  double **xii;
22  double *xie;
23  double *xien;
24  int **ipiv;
25  double eps;
26  sousdomaine();
27  sousdomaine(int,int,string);
28  sousdomaine(const sousdomaine &);
29  sousdomaine &operator=(const sousdomaine &);
30  void read_matrice(string);
31  void Schwarz(double);
32  void compute();
33  void update();
34  void factorize();
35  void save();
36  virtual ~sousdomaine();
37  };
38  #endif /* SOUSDOMAINE_H_ */
```

4.2 Notre .cpp :

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.

4.2.1 Variables utilisées :

Tout d'abord nous allons présenter les différentes variables qu'on va utiliser celle explicité3.1.1 en plus de quelques variables qu'on va expliciter maintenant par l'intermédiaire de mon code commenté :

Listing 2 – variables utilisées

```
int n, p;
map <ij ,double> Aie;
double **Aii;
double **bii;
double **xii;
double *xie;
double *xien;
int **ipiv;
double eps;
```

4.2.2 Constructeur :

Pour cela, comme pour toute classe nous allons commencer par un constructeur[4] :

Listing 3 – constructeur par défaut

```
sousdomaine::sousdomaine()
{
}
```

Le constructeur par défaut sera vide. Ensuite nous allons construire un deuxième constructeur durant lequel nous allons utiliser le mot clés **this** qui constitue notre pointeur et qui pointera vers nos variables qu'on initialiser auparavant dans notre header.

Listing 4 – constructeur par copie

```
sousdomaine::sousdomaine(int n, int p, string name){
this->n = n;
this->p = p;
int m = n/p;
int info = 3;
this->Aii = new double*[p];
this->ipiv = new int*[p];
this->bii = new double*[p];
this->xii = new double*[p];
//this->xie = new double[n];
for (int i=0; i<p; i++){
```

4. Les librairies qu'on a utilisé ne seront ont été exposée dans la section du header.

```
12 Aii[i] = new double[m*m];
13 for (int j=0;j<m*m;j++){
14 Aii[i][j]=0.;
15 }
16 ipiv[i] = new int[m];
17 bii[i] = new double[m];
18 for (int j=0; j<m; j++){
19 bii[i][j] = 1.;
20 }
21 this->xii[i] = new double[m];
22
23 }
24 this-> read_matrice(name);
25 }
```

4.2.3 Factorisation LU :

Théorie :
Soit A une matrice carrée. On dit que A admet une décomposition LU s'il existe une matrice triangulaire inférieure formée de 1 sur la diagonale, notée L, et une matrice triangulaire supérieure.

Implémentation :

Ensuite nous allons implémenter la méthode de factorisation via les subroutines de LaPack

Listing 5 – parallélilisation via MPI

```
1 void sousdomaine::factorize(){
2 int m = this->n/this->p;
3
4 for (int i=0;i<p;i++){
5 int iii = LAPACKE_dgetrf(LAPACKE_ROW_MAJOR,m,m,&Aii[i][0],m,ipiv[i]);
6 }
7 }
```

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 $AX = b$. Cette subroutine servira énormément pour mettre à jour nos différents

4.2.4 Constructeur par copie :

Ici nous réinitialisons nos différente variables par le biais du pointeur **this**.

Listing 6 – Constructeur par copie :

```
1 sousdomaine::sousdomaine(const sousdomaine & autresousdom){
2 this->xii=autresousdom.xii;
3 this->xie=autresousdom.xie;
4 this->bii=autresousdom.bii;
5 this->Aii=autresousdom.Aii;
6 this->Aie=autresousdom.Aie;
7
8 }
```

4.2.5 Surcharge de l'opérateur = :

Pour la surcharge de l'opérateur =, nous allons faire de même mais avec quelques modifications.

Listing 7 – Surcharge de l'opérateur =

```
1 sousdomaine & sousdomaine::operator=(const sousdomaine & autresousdom){
2 this->xii=autresousdom.xii;
3 this->xie=autresousdom.xie;
4 this->bii=autresousdom.bii;
5 this->Aii=autresousdom.Aii;
6 this->Aie=autresousdom.Aie;
7
8 return *this;
9 }
```

4.2.6 Destructeur :

Listing 8 – Destructeur :

```
1 sousdomaine::~sousdomaine()
2 {
3 }
```

4.2.7 Les autres méthodes :

Premièrement la lecture de la matrice :

Listing 9 – Lecture de la matrice :

```
1 void sousdomaine::read_matrice(string name)
2 {
3   float i, j, val, iijj;
```

```
4     int n= 100;
5     map <ij , double >:: iterator it;
6     int pi, pj;
7     ifstream file(name.c_str(),ios::in);
8     while (file.good()){
9       file >> i >> j >> val;
10      i=int(i)-1;
11      j=int(j)-1;
12      pi = i*p/n;
13      pj = j*p/n;
14        if (pi == pj){
15          iijj = (i-pi*n/p)*n/p+(j-pj*n/p);
16          Aii[pi][iijj] = val;}
17         else {
18          Aie[pi][ij(i-pi*n/p,j)] = val;
19      }
20  }
21  }
```

Explication détaillée de la méthode :

On commence par intialiser i, j , val puis par initialiser la taille de matrice avec n=100.
Ensuite, nous créons notre map, nous ouvrons le fichier contenant notre matrice puis nous le parcourons pour extraire les valeurs avec lesquels on remplit notre matrice.

Deuxièmement, la méthode solve :

Listing 10 – Methode compute :

```
1
2  void sousdomaine::solve(){
3      int m=this->n/this->p;
4      #pragma omp for private(i,it)
5      for (int i=0;i<p;i++){
6          for (int j=0; j<this->n/this->p; j++){
7              xii[i][j] = bii[i][j];
8          }
9          for (map<ij,double>::iterator it=Aie[i].begin(); it!=Aie[i].end(); i++){
10             xii[i][it->first.i]=xii[i][it->first.i]-it->second*xie[it->first.j
11         }
12         LAPACKE_dgetrs(LAPACK_ROW_MAJOR, 'N',m,1,&Aii[0],m, ipiv[i],&xii[i][0],
13     }
14
15     // dgetrs(Aii[i], ipiv[i], xii[i]) recuperer dans xii notre solution
16 }
```

Explication détaillée de la méthode :

Dans cette méthode nous faisons la résolution de $x_{ie} = b_i - A_{ie}x_{ie}$
Pour cela, on commence par utiliser le mot clés pragma à dessein de paralléliser notre code en mettant un thread par sous domaine. Ensuite on commence notre calcul via une double boucle for qui nous permet de calculer $x_{ie} = b_i$.
Ensuite on calcul notre x_{ie} via la formule ci dessus et la bibliothèque mkl qui nous a permis de faire notre factorisation LU.

Troisièmement, update

Listing 11 – Methode compute :

```
1
2 void sousdomaine::update(){
3     for (int i=0;i<this->p;i++){
4         for (int j=0; j< this->n/this->p; j++){
5             xie[i*this->n/this->p+j] = xii[i][j];
6         }
7     }
8 }
```

Dans cette méthode nous faisons simplement la mise à jour de nos x_{ie} dans les différents sous domaines.

Quatrièmement, la méthode save :

Listing 12 – Methode save :

```
1
2 void sousdomaine::save(){
3    ofstream flux("res.txt",ofstream::out);
4    for (int i=0;i<this->n;i++){
5       flux << std::setprecision(20) << this->xie[i] << std::endl;
6       }
7       flux.close();
8       }
```

Dans cette méthode nous faisons simplement la sauvegarde dans un fichier txt .

4.2.8 Méthode Schwarz

Listing 13 – Methode Schwarz :

```
1 void sousdomaine::Schwarz(double normax){
2     double eps=1e-12;
3     double norm, normax=1.;
4     int it=0;
5     while (normax>eps){
6 #pragma omp for
7         for(int j=0;j<this->n;j++){
```

```
8                 this->xien[j] = this->xie[j];
9             }
10            this->compute();
11            this->update();
12 #pragma omp criticial
13            normax =0.;
14 #pragma omp for
15            for (int j=0;j<this->n;j++){
16                norm = fabs(this->xien[j]-this->xie[j]);
17                if (norm>normax){
18                    normax=norm;
19                }
20 #pragma omp sections{}
21                {
22                    it++;
23                    cout << it << ":" << normax << endl;
24                }
25            }
26        }
27 }
```

Tout d'abord, nous commençons par fixer notre tolérance à $\varepsilon = 10^{-12}$, ensuite nous initialisons notre iterateur à zéros et nos normes à 1.
Ensuite on utilise une boucle while pour définir notre critère de convergence $norm_{max} > \varepsilon$.
Nous utilisons pragma pour paralléliser notre boucle for qui permet de calculer la norme via $\|Xien(j) - Xie(j)\| \forall j = 1, ..., n$, puis on cherche la norme maximale.
Ensuite nous utilisons constructeur **Sections** qui attribue à chaque thread un bloc structuré différent puis on affiche notre itérateur et notre norme max.

5 Conclusion et bibliographie :

À dessein de conclure ce projet nous a permis d'effectuer cette analogie cours/réalité sur un domaine très prisé par les mathématiciens : Le calcul parallèle.
De plus, grâce à ce t.p. nous avons manipulé les différentes notions vu en cours afin de pouvoir faire cette compréhension de la théorie et passer à l'application des notions étudiées dans le contexte d'un projet un peu plus orienté ingénierie.