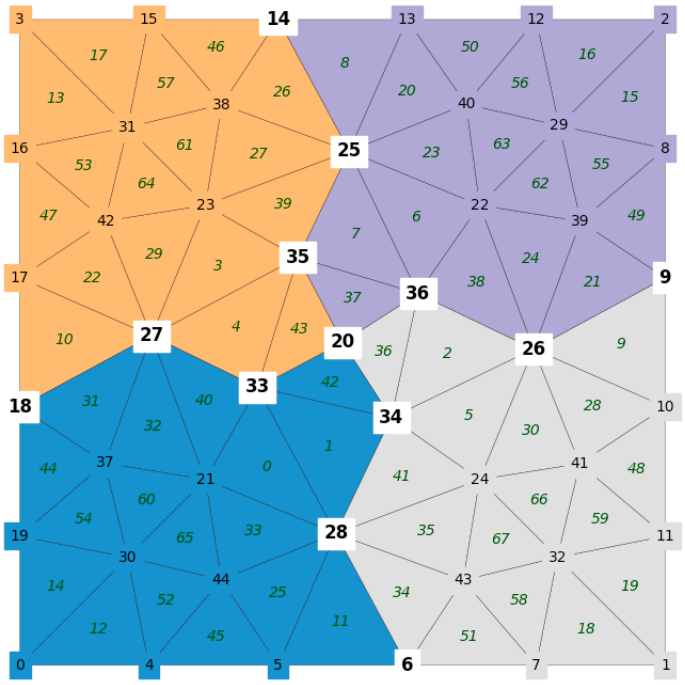
SORBONNE UNIVERSITÉ
CRÉATEURS DE FUTURS
DEPUIS 1257

Sorbonne Université

Calcul parallèle

Projet Laplacien



Kevin Ancourt - Alexis Squarcioni

Master 2 IMPE-Meca
2019-2020

Table des matières

1 Introduction

L'objectif du projet est de paralléliser un code donné dans le but de résoudre un problème numérique. Dans un premier temps, ce projet consistera à partitionner un maillage et ensuite, à paralléliser un code numérique afin de pouvoir utiliser le gradient conjugué.

2 Mise en place du problème locale

Tout d'abord, avant de rentrer dans le vif du sujet, il nous faut mentionner que lors de ce projet nous utiliserons la méthode dite **par élément**. Celle-ci s'avèrera utile lors de la construction du maillage, cependant quelques difficultés apparaîtront dans la suite du projet.

Le code nous est fourni avec un maillage global d'un carré, plus ou moins serré. La première étape de notre projet est de *partitionner* ce maillage afin de créer plusieurs maillages locaux. Le défi ici est de redéfinir une numérotation, une **numérotation locale**. Voici un exemple ci-dessous :

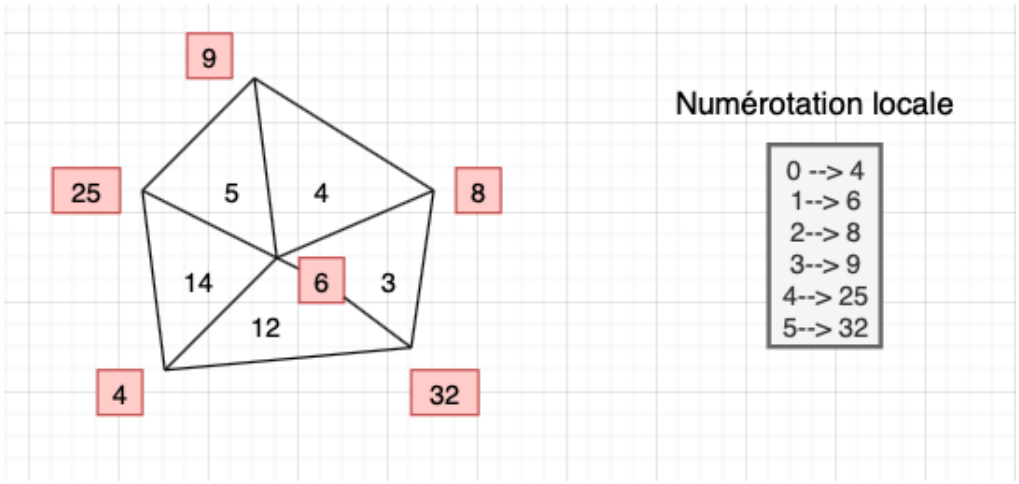


FIGURE 1 – Exemple de numérotation locale

Ancienne numérotation	Sommets correspondants		
3	32	8	6
12	4	6	32
5	25	9	6

Nouvelle numérotation	Sommets correspondants		
0	5	2	1
1	0	1	5
2	4	3	1

La figure 1 est un exemple simple de ce que nous devons faire sur notre carré. Le premier tableau correspond à la numérotation du dessin de gauche de la figure 1, c'est à dire la numérotation globale, celle qui prend en compte la géométrie dans sa totalité. Ensuite ,on met donc en place, à

droite de la figure 1, une numérotation locale, on prend simplement nos sommets et on les trie de manière croissante. Cette numérotation locale est expliquée dans le second tableau.

Il y a précisément deux étapes à faire avant de pouvoir paralléliser le code. Il faudra créer localement (de la manière dont nous l'avons expliqué juste au-dessus) dans un premier temps les domaines au nombre de processus demandés. Plus concrètement, nous devons parvenir à construire *elt2verts* localement et nous appellerons cette nouvelle liste *tri_global*. La seconde étape consiste à créer *coords* localement, qui se nommera par la suite *Coords*.

Construction de *list_sommet*

La création de cette liste est importante, c'est à travers elle que nous allons pouvoir construire *tri_global* et *Coords*. Pour la définir on se sert principalement de *elt2verts* de la manière explicitée ci-dessous, en se servant simplement de *etsDomains_Proc* qui rassemble les éléments d'un certain domaine.

```
for e in etsDomains_Proc:
  tri_global.append(list(elt2verts[e])) #elt2vert version local
  list_sommet.append(elt2verts[e][0])
  list_sommet.append(elt2verts[e][1])
  list_sommet.append(elt2verts[e][2])
```

Listing 1 – Construction de tri_global

Par la même occasion on construit aussi *tri_global* mais pour l'instant la numérotation est fausse.

Construction de *tri_global*

Le code déjà fourni nous permet de ressortir, sous forme de liste, l'ensemble de triangles (avec leur 3 numéros de sommets) d'un certain domaine en numérotation globale. La première chose à faire est de construire une liste qui correspond aux numéros des sommets dans le domaine, cette liste s'appelle *list_sommet* telle que nous l'avons construite précédemment. Il ne faut pas oublier non plus de trier cette liste de manière croissante afin de numéroter correctement tri_global. On peut alors facilement mettre en place notre numérotation locale du domaine voulu en indexant simplement par rapport à *list_sommet*. Voici le code de sa construction ci-dessous :

```
for i in range(len(tri_global)):
  for j in range(3):
    tri_global[i][j]= list_sommet.index(tri_global[i][j])
```

Listing 2 – Construction de tri_global

Construction de *Coords*

Coords est encore plus simple à définir, on se sert simplement de la fonction *coords* déja établis tout en nous servant de *list_sommet*.

```
for i in range (len(list\_sommet)):
  Coords.append((list(coords[list\_sommet[i]])))
```

Listing 3 – Construction de Coords

3 Construction de l'interface

Une fois *tri_global* et *Coords* construit il est très facile de paralléliser le code car ce sont ces deux notions qui permettent d'établir le code donné. En effet, il nous suffit de reprendre le code déjà établi mais en remplaçant *elt2verts* par *tri_global* et *coords* par *Coords*. Afin de ne pas se mélanger dans le code, nous avons mis le mot *_new* sur les commandes qui parallélisent le code. Cependant, il faut faire attention, la construction du second membre b n'est pas triviale, il faut que le vecteur local soit une restriction du vecteur global. Pour ce faire, nous faisons des échanges à l'interface afin de faire en sorte que nos vecteurs b locaux soient bien une restriction du vecteur global et que les valeurs dans les domaines voisins en chaque sommet de l'interface soit égales entre elles.

Remarque : Il est important de convertir tri_global et Coords en array afin que les processus que l'on fait dessus puissent bien se dérouler.

```

for i in bord:
    list_b=[]
    for j in bord[i]:
        list_b.append(b[j])
    comm.send(list_b,dest=i)

for i in bord:
    value_b=comm.recv(source =i)
    for elt_recv in bord[i]:
        b[elt_recv] += value_b[bord[i].index(elt_recv)]
```

Listing 4 – Echange à l'interface pour le vecteur second membre

le but est de reconstituer notre vecteur b et pour cela nous devons rajouter les éléments du bord, c'est à dire les additionner aux endroits exacts où ces éléments de bords sont situés. La première étape est de créer une liste qui rassemble directement les éléments de bord afin de les envoyer à l'aide de la commande *comm.send*. La tache compliquée est de remplir notre vecteur local b aux bons endroits. Pour ce faire on va boucler sur notre liste *bord* afin de remplir notre vecteur b en additionnant celui-ci avec les valeurs reçues, c'est à dire les valeurs dans *value_b*. Le défi ici est, non pas seulement d'ajouter les valeurs de *value_b* dans b mais de les ajouter aux endroits exacts. On doit construire notre vecteur b morceau par morceau. On trouve l'endroit exact où additionner en ressortant la position à partir de notre liste *bord*

4 Produit scalaire et produit matrice-vecteur

Afin de résoudre notre problème numérique, le gradient conjugué doit être quelque peut modifié. La structure de son code reste inchangée mais deux de ses arguments doivent être construit de manière parallèle, le produit scalaire et le produit matrice-vecteur.

```
1
2  def prodMatVect(mat,vect,bord,list_sommet):
3      y=np.zeros(vect.shape[0])
4
5      for i in range(len(list_sommet)):
6          for j in range(len(list_sommet)):
7              y[i] +=mat[i,j] * vect[j]
8
9      for i in bord:
10         voisin=[]
11         for j in bord[i]:
12             voisin.append(y[j])
13         comm.send(voisin,dest=i)
14
15     for i in bord:
16         value_voisin=comm.recv(source =i)
17         for elt_recv in bord[i]:
18             y[elt_recv] += value_voisin[bord[i].index(elt_recv)]
19     return y
```

Listing 5 – La fonction produit scalaire

Le produit matrice-vecteur en parallèle commence par faire un produit matrice-vecteur classique. C'est ce qui est fait jusqu'à la ligne 7. Ensuite il faut bien entendu paralléliser le calcul, c'est-à-dire prendre en compte les éléments (sommets) qui partagent plusieurs domaines à la fois. Pour cela, l'utilisation de la même méthode que pour assembler le second membre est utilisée ici. À l'aide de *comm.send* on envoie les voisins repérés et ensuite on utilise *comm.recv* afin d'additionner les contributions des différents éléments du bords au bon endroit dans notre vecteur de sortie.

```
1
2  def prodScal(x,y,sommet_interne,bord):
3      scal=0.
4      list_bord=[]
5      for i in bord:
6          for j in bord[i]:
7              if j not in list_bord:
8                  list_bord.append(j)
9
10     for i in sommet_interne:
11         scal += x[i] * y[i]
12
13     for j in list_bord:
14
15         cout=1
16         for k in bord:
17             for l in bord[k]:
18                 if l ==j:
19                     cout +=1
20
21         scal += (1./cout)*(x[j] * y[j])
22
23     scalglobal = comm.allreduce(scal, MPI.SUM)
24     return scalglobal
```

Listing 6 – La fonction produit matrice vecteur

Ici dans un premier temps on créer une liste des éléments du bord, *list_ bord*, tout en évitant les doublons et en gardant l'ancienne liste *bord*. Ensuite le produit scalaire est fait de manière classique concernant les éléments intérieurs. Pour les éléments sur le bord on met en place un produit scalaire ordinaire mais tout en comptant pour chaque noeuds de l'interface le nombre de domaines auquel il appartient. L'assemblage est ici délicat il faut paralléliser morceau par morceau. Le produit scalaire parallèle est fait de manière local mais la sortie doit être globale. Ainsi à la fin, à l'aide de de la commande *comm.allreduce*, on somme tout les produits scalaires calculés par chacun de nos processeur et on les revoit à tout le monde.

5 Résolution numérique

Après avoir paralléliser le code, nous souhaitons savoir si nous obtenons les bons résultats. Pour ce faire nous allons tout d'abord afficher la solution globale que nous obtenons :

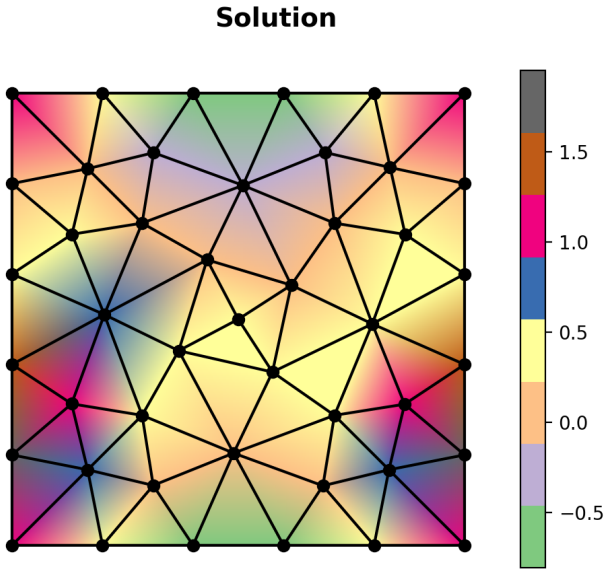


FIGURE 2 – Solution globale sans parallélisation

Afin de vérifier nos résultats il faut comparer plusieurs choses, tout d'abord nous vérifions les réponses uniquement de manière visuelle, puis nous allons analyser le résidu que l'on obtient à l'aide du gradient conjugué, enfin nous calculons l'erreur relative entre la solution globale et la solution obtenue après parallélisation.

Nous avons lancé le programme parallélisé sur 4 processeurs, nous obtenons les solutions suivantes sur chaque sous-domaines :

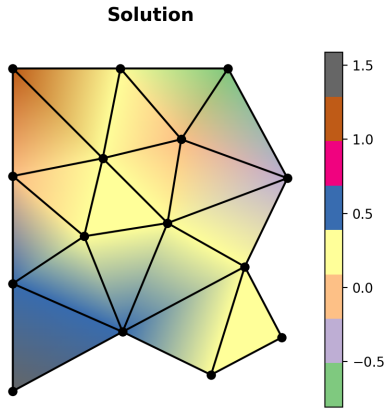


FIGURE 3 – Solution locale sur le processus 1

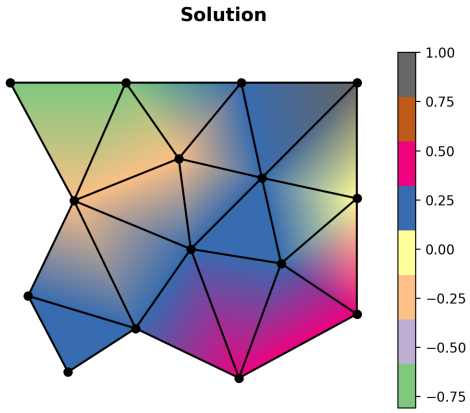


FIGURE 4 – Solution locale sur le processus 2

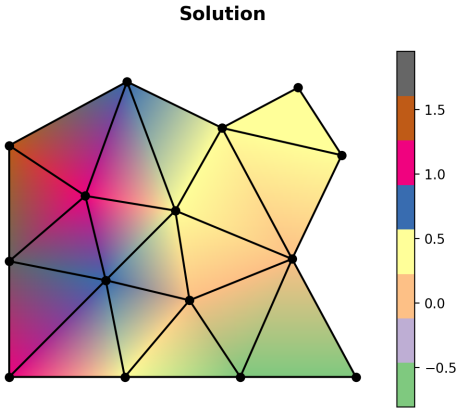


FIGURE 5 – Solution locale sur le processus 0

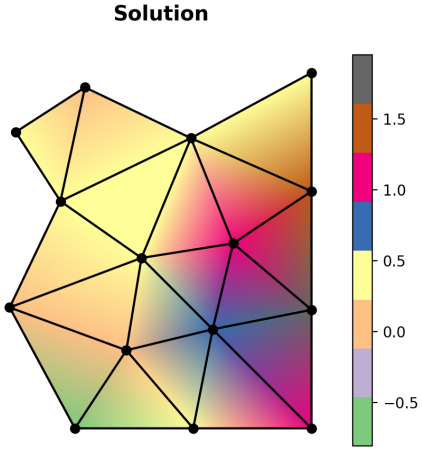


FIGURE 6 – Solution locale sur le processus 3

Si on compare visuellement en prenant en compte l'échelle de chaque graphique la solution semble cohérente par rapport à la solution globale. Nous avons ensuite vérifier le résidu obtenue par le gradient conjugué, on obtient pour le programme non paralléliser

$$||r_{final}|| = 6.923755311046541e-15,\ nombre\_itrations : 26$$

Pour le programme paralléliser, nous obtenons :

$$||r_{final}|| = 4.681496092242066e-15,\ nombre\_itrations : 29$$

Les résidus sont identiques pour chaque processus ce qui est cohérent, de plus c'est égale à l'erreur machine.

Enfin, nous avons calculer l'erreur relative pour notre solution :

$$err_{relative} = \left\| \frac{solution_{refaite} - solution_{globale}}{solution_{globale}} \right\|$$

On obtient donc toujours sur 4 processus, l'erreur relative suivante :

$err_{relative} = 9.154634012193224e - 15$

On a donc une solution égale à la solution globale.

6 Conclusion

Les objectifs du projet sont remplis, nous sommes parvenus à paralléliser un code de calcul dans le but d'une résolution numérique par le gradient conjugué. Afin de comparer nos résultats avec les résultats fournis par le code sans parallélisation nous étudions l'erreur relative. Celle-ci est égale à l'erreur machine peu importe le nombre de processus, les deux solutions sont donc les mêmes, la parallélisation est faite avec succès. Ce projet nous a permis d'acquérir de solides bases en calcul parallèle à travers l'utilisation et la compréhension des principales commandes de MPI. Ces commandes nous ont d'ailleurs servis à mettre en oeuvre des méthodes mathématiques telles que les méthodes de Krylov. Nous avons aussi appris à redéfinir des méthodes mathématiques classiques que nous connaissions déjà, à travers MPI, comme le produit scalaire ainsi que le produit matrice-vecteur.