Soutenance de Projet - AMS301

Problème stationnaire avec grille non structurée

Nour El Haddad & Adrien Sardi

22 Novembre 2023

Intervenants :
Axel Modave
Nicolas Kielbasiewicz

Plan

1. Présentation du problème
   - Problème continu
   - Problème discrétisé
   - Buts du projet
2. Implémentation des solveurs
   - Algorithme de **Jacobi**
   - Parallélisation du solveur **Jacobi**
   - Algorithme du **Gradient Conjugué**
   - Parallélisation du solveur **Gradient Conjugué**
3. Validation
   - Validation des codes séquentiels
   - Validation des codes parallèles
4. Comparaisons
   - Comparaison des convergences sur un benchmark
   - Comparaison des performances : scalabilité faible et scalabilité forte
   - Comparaison globale

Plan

1. Présentation du problème
   - Problème continu
   - Problème discrétisé
   - Buts du projet
2. Implémentation des solveurs
   - Algorithme de **Jacobi**
   - Parallélisation du solveur **Jacobi**
   - Algorithme du **Gradient Conjugué**
   - Parallélisation du solveur **Gradient Conjugué**
3. Validation
   - Validation des codes séquentiels
   - Validation des codes parallèles
4. Comparaisons
   - Comparaison des convergences sur un benchmark
   - Comparaison des performances : scalabilité faible et scalabilité forte
   - Comparaison globale

Présentation du problème

Problème continu :

- EDP dans un domaine Ω
- **Problème :**

Chercher $u \in H^1(\Omega)$ vérifiant :

$$\begin{cases} \alpha u - \Delta u = f & \text{dans } \Omega, \\ \dfrac{\partial u}{\partial n} = 0 & \text{sur } \partial\Omega, \end{cases}$$

$\longrightarrow$

$$\begin{array}{c} u \in H^1(\Omega) \quad : \\ \alpha \int_\Omega uv + \int_\Omega \nabla u \cdot \nabla v = \int_\Omega fv, \\ \forall v \in H^1(\Omega). \end{array}$$

Problème discrétisé :

- Approximation par les E.F. $\mathbb{P}^1$ (Méthode de Galerkin)
- **Système :**

$$[\alpha\mathbb{M} + \mathbb{K}]\, U = \mathbb{M}F$$

$\longrightarrow$

$$\begin{array}{rcl} M_{ij} & = & \int_\Omega w_i w_j, \\ K_{ij} & = & \int_\Omega \nabla w_i \cdot \nabla w_j, \\ f_i & = & f(M_i), \\ u_i & = & u_h(M_i). \end{array}$$

Présentation du problème : Les objectifs

1. Résolution de l'EDP par E.F.
2. Complétion/implémentation de deux solveurs itératifs : **Jacobi** et **Gradient Conjugué**
3. Validation du code et de la convergence
4. Validation de la parallélisation du code
5. Comparaison de la convergence et de la performance des solveurs

Plan

1. Présentation du problème
   - Problème continu
   - Problème discrétisé
   - Buts du projet
2. Implémentation des solveurs
   - Algorithme de **Jacobi**
   - Parallélisation du solveur **Jacobi**
   - Algorithme du **Gradient Conjugué**
   - Parallélisation du solveur **Gradient Conjugué**
3. Validation
   - Validation des codes séquentiels
   - Validation des codes parallèles
4. Comparaisons
   - Comparaison des convergences sur un benchmark
   - Comparaison des performances : scalabilité faible et scalabilité forte
   - Comparaison globale

Implémentation du solveur : **Jacobi**

- Code parallélisé déjà fourni
- On implémente et parallélise :
  1. La norme 2 du résidu pour le critère d'arrêt Jacobi
  2. La norme L^2 de l'erreur $||u_h - u_{ref}||_{L^2}$: $||v||_{L^2} \approx ||v_h^T \mathbb{M} v_h||$

Points d'attention

- Faire attention au passage par addresse
- Comprendre la différence entre la norme locale et la norme globale
- Rajouter une fonction `removeInterMPI`

Parallélisation du solveur **Jacobi**

Idée pour la parallélisation

2 points de vue :

1. **PDV vecteur :**
   Chaque processus p_i calcule $u|_{p_i}$ ainsi que les termes de bords
2. **PDV scalaire :**
   Chaque processus calcule $f(u) \in \mathbf{R}$ où $u \in \Omega$

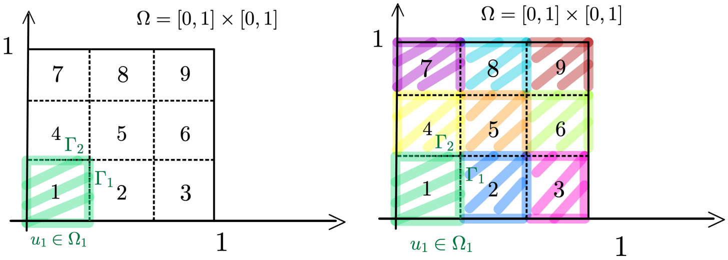

Figure 1 – Schéma qui illustre le fonctionnement du code parallèle

Parallélisation du solveur **Jacobi**

```
double norm_2_glo(ScaVector u, Mesh& mesh) {
    removeInterfMPI(u,mesh);
    double n_loc = 0; double n_glo; double size = u.size();
    for (int i = 0; i < size; ++i) {
        n_loc += u(i)*u(i);
    }

    MPI_Allreduce (&n_loc, &n_glo, 1, MPI_DOUBLE , MPI_SUM,
    MPI_COMM_WORLD);
    return sqrt(n_glo);
}
```

Listing 1 – Fonction norme globale

```
void removeInterfMPI(ScaVector& vec, Mesh& mesh)
{
  for(int nTask = 0; nTask < myRank+1; ++nTask){
    int numToExch = mesh.numNodesToExch(nTask);

    for(int nExch=0; nExch<numToExch; nExch++){
        vec(mesh.nodesToExch(nTask,nExch)) = 0;
    }}}
```

Listing 2 – Fonction removeInterMPI

Implémentation du solveur : **Gradient Conjugué**

- On écrit le système linéaire $Au = b$ sous la forme d'un problème de minimisation de la fonctionnelle :

$$\boxed{J(x) \;=\; \tfrac{1}{2}(Ax, x) - (b, x)}$$

- On résout alors le système itératif non stationnaire suivant :

$$\boxed{\begin{cases} x(0) \in \mathbf{R}^n \text{ donnée} \\ x^{(\ell+1)} = x^{(\ell)} + \alpha^{(\ell)} p^{(\ell)} \end{cases}}$$

où $\alpha^{(\ell)} = \frac{\left(p^{(\ell)}, p^{(\ell)}\right)}{\left(Ap^{(\ell)}, p^{(\ell)}\right)}$ et $p^{(\ell)} = -\nabla J(x^{(\ell)}) = b - Ax^{(\ell)}$

Code du solveur **Gradient Conjugué**

```
while (norm_r/norm_r_0 > tol && it < maxit){
    // 1. Update field
        Ap = A*p;
        exchangeAddInterfMPI(Ap, mesh);
        alpha_= prod_scal_glo(r,p,mesh)/prod_scal_glo(Ap,p,mesh);
        u += alpha_*p;

    // 2. Update residu and p
        r -= alpha_*Ap;
        Ar = A*r;
        exchangeAddInterfMPI(Ar, mesh);
        beta = -prod_scal_glo(Ar,p,mesh)/prod_scal_glo(Ap,p,mesh);
        p = r + beta*p;

    // 3. Update of norm_r
        norm_r = norm_2_glo(r,mesh);

    // 4. Affichage
        if(((it % (maxit/10)) == 0)){
            if(myRank == 0){
             cout << "["<< it <<"] residual:" << norm_r/norm_r_0 <<
 endl;}} it++;}
```

Listing 3 – Code pour le gradient conjugué

Parallélisation du solveur **Gradient Conjugué**

Points d'attention

L'algorithme fait intervenir des **produits scalaires** !

Solution

A chaque fois que l'on calcule un scalaire, on élimine les bords en communs selon la règle qu'on s'est fixés

Plan

1. Présentation du problème
   - Problème continu
   - Problème discrétisé
   - Buts du projet
2. Implémentation des solveurs
   - Algorithme de **Jacobi**
   - Parallélisation du solveur **Jacobi**
   - Algorithme du **Gradient Conjugué**
   - Parallélisation du solveur **Gradient Conjugué**
3. Validation
   - Validation des codes séquentiels
   - Validation des codes parallèles
4. Comparaisons
   - Comparaison des convergences sur un benchmark
   - Comparaison des performances : scalabilité faible et scalabilité forte
   - Comparaison globale

Validation des codes séquentiels

Objectif

Valider notre code en prenant $a, b, \alpha = 1$ avec la solution manufacturée :

$$u(x) = \cos(4\pi x)\cos(\pi y)$$

Lemme d'Aubin-Nitsche

$$||u - u_h||_{L^2} \leq Ch^2|u|_2$$

- On sait que notre erreur doit varier à la même vitesse que h^2
- $\log ||u - u_h||_{L^2}$ doit avoir une pente de 2 en fonction de $log(h)$
- $\log ||u - u_h||_{L^2}$ **doit avoir une pente de** -2 **en fonction de** $log(\frac{1}{h})$

Validation des codes séquentiels

Mission accomplie !

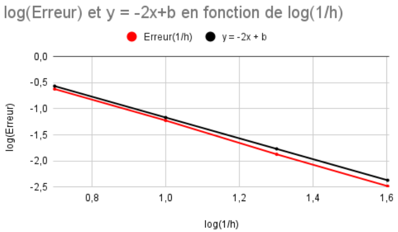

Figure 2 – Log erreur en fonction de log (1/h) Jacobi

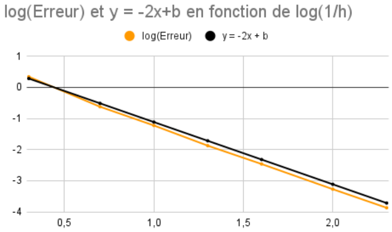

Figure 3 – Log erreur en fonction de log (1/h) Gradient Conjugué

- Régressions Linéaires :
  1. $y = -2.07x + 0.832R^2 = 1$ pour (2)
  2. $y = -2.08x + 0.885R^2 = 0.999$ pour (3)

Validation des codes parallèles

Validation

1. On obtient le même nombre d'itérations en séquentiel et en parallèle.
2. On obtient le même résidu final.

Validation des codes parallèles

Validation

1. On obtient le même nombre d'itérations en séquentiel et en parallèle.
2. On obtient le même résidu final.

Mission accomplie !

```
== gradient conjugate, nbTasks = 1
   [0] residual: 4.22068
   -> final iteration: 434 (prescribed max: 10000)
   -> final residual: 9.99672e-07 (prescribed tol: 1e-06)
   -> Erreur L2 : 0.000538397
== gradient conjugate, nbTasks = 4
   [0] residual: 4.22068
   -> final iteration: 434 (prescribed max: 10000)
   -> final residual: 9.99688e-07 (prescribed tol: 1e-06)
   -> Erreur L2 : 0.00053533
== gradient conjugate, nbTasks = 8
   [0] residual: 4.22068
   -> final iteration: 434 (prescribed max: 10000)
   -> final residual: 9.99688e-07 (prescribed tol: 1e-06)
   -> Erreur L2 : 0.00053533
```

Figure 4 – Même résidu et même nombre d'itération pour gradient conjugué

Plan

1. Présentation du problème
   - Problème continu
   - Problème discrétisé
   - Buts du projet
2. Implémentation des solveurs
   - Algorithme de **Jacobi**
   - Parallélisation du solveur **Jacobi**
   - Algorithme du **Gradient Conjugué**
   - Parallélisation du solveur **Gradient Conjugué**
3. Validation
   - Validation des codes séquentiels
   - Validation des codes parallèles
4. Comparaisons
   - Comparaison des convergences sur un benchmark
   - Comparaison des performances : scalabilité faible et scalabilité forte
   - Comparaison globale

Comparaison des convergences

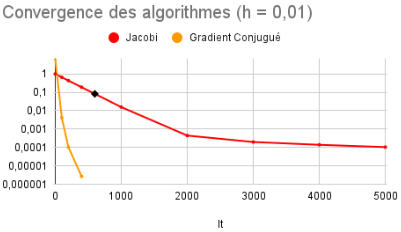

Figure 5 – Log du résidu en fonction du nombre d'itérations pour $h = 0.01$

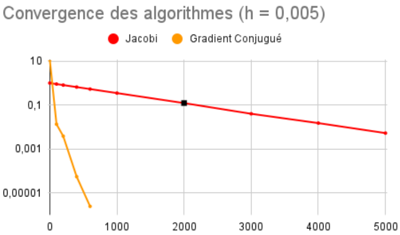

Figure 6 – Log du résidu en fonction du nombre d'itérations pour $h = 0.005$

Comparaison des convergences

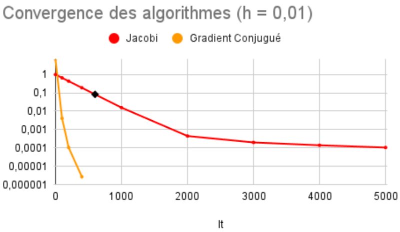

Figure 7 – Log du résidu en fonction du nombre d'itérations pour $h = 0.01$

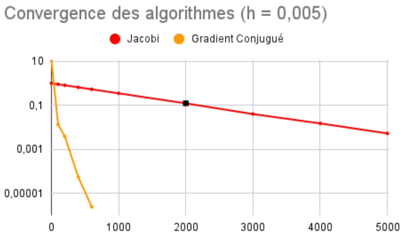

Figure 8 – Log du résidu en fonction du nombre d'itérations pour $h = 0.005$

Observations

- L'algorithme du Gradient conjugué converge beaucoup plus rapidement que Jacobi (attendu)
- Convergence d'autant plus rapide que le temps d'execution d'une boucle de l'algorithme du Gradient conjugué est plus long que celle de l'algorithme de Jacobi (calcul de produit scalaire)
- L'écart augmente lorsque h augmente

Comparaison des performances : Scalabilité forte

Scalabilité Forte

- Pour h fixé, le nombre de processeurs augmente de 1 à P et on calcule S pour chaque nombre de processeurs.
- $S = \frac{T_{\text{séquentiel}}}{T_{\text{parallèle}}} \in [0, P]$
- Calcul réalisé sur le Cluster Cholesky

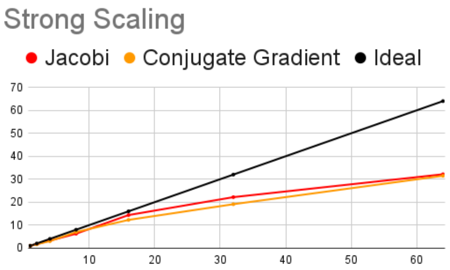

Figure 9 – Scalabilité forte pour Jacobi et Gradient Conjugué

Comparaison des performances : Scalabilité forte

Observations

- Une meilleure scalabilité forte pour Jacobi mais avec un écart relativement faible.

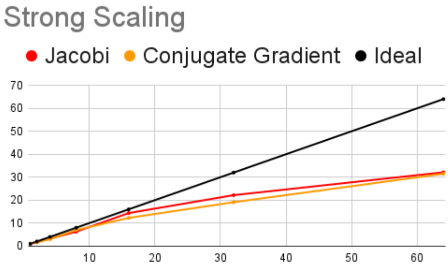

Figure 10 – Scalabilité forte pour Jacobi et Gradient Conjugué

Comparaison des performances : Scalabilité faible

Scalabilité Faible

- $E = \frac{S_{\text{actuel}}}{S_{\text{idéal}}} \in [0, 1]$
- $E = 1$ en théorie
- Calcul réalisé sur le Cluster Cholesky

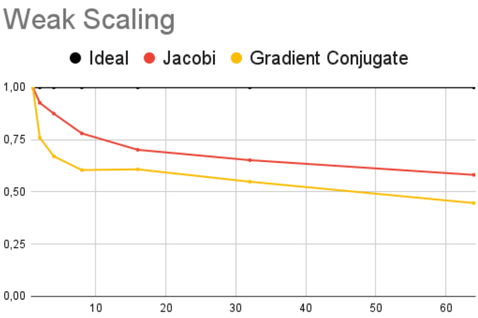

Figure 11 – Scalabilité faible pour Jacobi et Gradient Conjugué

Comparaison des performances : Scalabilité faible

Observations

- Une efficacité plus grande pour Jacobi.
- L'écart entre les deux est cette fois plus visible (de l'ordre de 15%).

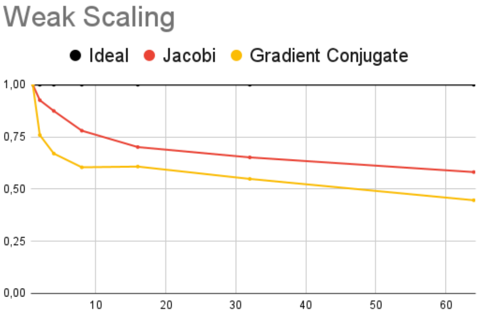

Figure 12 – Scalabilité faible pour Jacobi et Gradient Conjugué

Comparaison globale

En résumé

- Le **Gradient Conjugué** converge beaucoup plus rapidement que **Jacobi**.
- Les performances parallèles en terme de speedup et efficacité sont meilleures pour l'algorithme de **Jacobi**.

Conclusion

1. Implémentation et parallélisation d'un code en `C++`
2. Comparaison de la performance de nos algorithmes
3. Utilisation du mésocentre `Cholesky` de l'École Polytechnique

Merci pour votre attention !