Résolution du problème de Poisson en 2D

Dans la première partie nous avons vu des fonctions de bases de MPI pour échanger des données (*messages*) entre les processus. Nous appliquerons ces connaissances à la résolution d'un problème physique.

I. DESCRIPTION DU PROBLÈME

Nous considérons le problème de Poisson :

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) u(x,y) = f(x,y) \qquad \text{dans } \Omega = [0,1] \times [0,1]$$

$$f(x,y) = 2(x^2 - x + y^2 - y)$$

$$u(x,y) = 0 \qquad \text{sur } \partial\Omega$$

Nous chercherons par une méthode numérique les valeurs de la fonction $u(x,y)$ sur une grille de points. La solution exacte (supposée inconnue) de ce problème est donnée par

$$u_{\text{exact}}(x,y) = xy(x-1)(y-1)$$

A. Méthode numérique

Le domaine est discrétisé en x et y avec un pas de h_x et h_y suivant x et y (voir figure 1). Chaque fonction $f(x,y)$ sera donc représentée par un tableau $\mathbf{F}$ tel que $F_{ij} = f(x + i/h_x, y + j/h_y)$.

Pour résoudre le problème discret, on utilisera une méthode itérative. On va donc construire une séquence de tableaux $\mathbf{U}^n$ qui converge vers le résultat du problème discrétisé. L'itération de Jacobi qui construit $\mathbf{U}^{n+1}$ à partir de $\mathbf{U}^n$ est donnée par

$$U_{ij}^{n+1} = c_0 \left(c_1 \left(U_{i+1,j}^n + U_{i-1,j}^n \right) + c_2 \left(U_{i,j+1}^n + U_{i,j-1}^n \right) - F_{ij} \right)$$

avec

$$c_0 = \frac{1}{2} \frac{h_x^2 h_y^2}{h_x^2 + h_y^2} \qquad c_1 = \frac{1}{h_x^2} \qquad c_2 = \frac{1}{h_y^2}$$

Cette méthode repose sur un stencil à 5 points. Sur la figure 1, le point rouge est le point de $\mathbf{U}^{n+1}$ que l'on cherche à construire en fonction des quatre points de $\mathbf{U}^n$ aux positions voisines (en bleu).

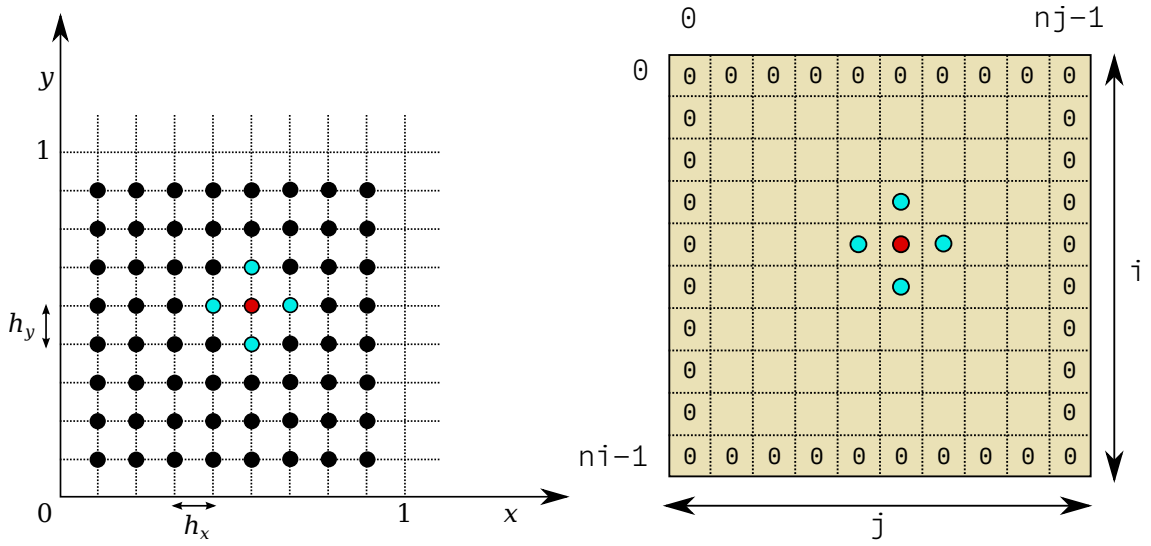


FIGURE 1: Discrétisation en x et y physique (à gauche) et représentation en mémoire (à droite).

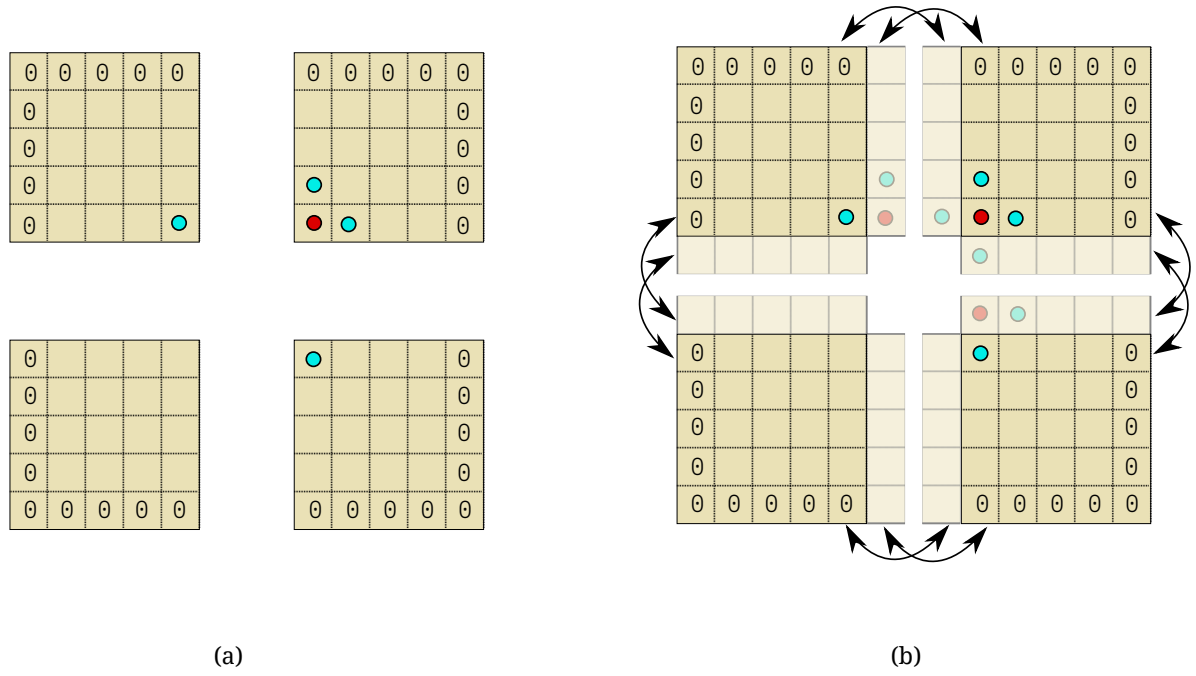
Algorithme séquentiel

```
while (!convergence) {
    for (i=1 ; i<ni-1 ; i++)
        for (j=1 ; j<nj-1 ; j++)
            U_new[i][j] = c0 * ( c1 * (U[i+1][j] + U[i-1][j])
                  + c2 * (U[i][j+1] + U[i][j-1]) - F[i][j] );
    convergence = test_convergence(U, U_new);
    echange (U, U_new);
}
```

Tant qu'un critère de convergence n'est pas satisfait, on calcule la nouvelle itération. Lorsque le nouveau tableau `U_new` est calculé, on peut remplacer les valeurs du tableau `U` par celles du tableau `U_new`. Pour éviter une copie de tableau, on préfèrera échanger les pointeurs correspondant à `U` et `U_new`.

B. Parallélisation

On peut voir qu'à une itération donnée, chacun des U_{ij}^{n+1} peut être calculé *indépendamment* des autres. Il sera donc possible de calculer en parallèle plusieurs valeurs U_{ij}^{n+1}.



(a) (b)

FIGURE 2: (a) Décomposition du domaine, (b) Échange des valeurs aux interfaces avec les sous-domaines voisins.

Notons que les quantités $U^n_{i+1,j}, U^n_{i-1,j}, U^n_{i,j+1}, U^n_{i,j-1}, F_{ij}$, requises à chaque itération pour calculer U^{n+1}_{ij}, ont toutes des indices voisins de `i` et `j` dans les tableaux. Ainsi, il est naturel de découper le domaine Ω en sous-domaines Ω_p, et de réaliser en parallèle le même travail sur chacun des sous-domaines. Lorsque le même programme tourne en parallèle sur des données différentes, on parle de stratégie SPMD (Single Program Multiple Data). Cette stratégie est parfaitement bien adaptée dans le cas présent où la quantité de calcul à réaliser sur chaque point est constante.

La principale difficulté que nous allons rencontrer est le fait que pour calculer les valeurs $\mathbf{U}^{n+1}$ sur les points des *interfaces* entre les sous-domaines, on aura besoin de valeurs de $\mathbf{U}^n$ qui se trouvent dans des sous-domaines voisins. Il sera donc nécessaire, à chaque itération, que chacun des sous-domaines échange les valeurs de $\mathbf{U}^n$ aux interfaces avec les voisins appropriés (figure 2).

Algorithme parallèle

Attention : `ni` et `nj` sont maintenant des dimensions locales des sous-domaines.

```
while (!convergence_globale) {
    communication_des_interfaces(U);
    for (i=1 ; i<ni-1 ; i++)
        for (j=1 ; j<nj-1 ; j++)
            U_new[i][j] = c0 * ( c1 * (U[i+1][j] + U[i-1][j])
                  + c2 * (U[i][j+1] + U[i][j-1]) - F[i][j] );
    calcul_de_U_new_aux_interfaces(U_new,U,F);
    convergence_locale = test_convergence(U, U_new);
    echange (U, U_new);
    convergence_globale = communication_convergence(convergence_locale);
}
```

II. EXERCICES

Les fichiers se trouvent dans le répertoire `exercices`.

Le but de ce TP est de réaliser le découpage du domaine Ω et d'implémenter les communications nécessaires aux interfaces entre les sous-domaines. Pour cela, nous utiliserons un petit programme modèle où chaque processus MPI crée un sous-domaine dont les valeurs sont égales à son rang MPI, et nous allons réaliser les échanges de messages nécessaires pour la communication des données aux bords.

Algorithme

1. `nj = 12` : Divisible par 1,2,3,4 et 6 donc bien adapté aux tests
2. `ni = nj / nb_process` : Découpage en groupes de lignes
3. Allocation d'un tableau `data[ni][nj]`
4. Initialisation des valeurs à `data[i][j] = rang`
5. Affichage du domaine
6. Barrière de synchronisation
7. Communication des valeurs aux bords
8. Affichage du domaine

A. MPI_Send / MPI_Recv à 2 processus

À 2 processus, le domaine est découpé en deux sous-domaines. Le processus de rang `0` va devoir envoyer sa dernière ligne au processus de rang `1`, et inversement le processus de rang `1` va devoir envoyer sa première ligne au processus de rang `0`. On crée une zone tampon `buffer` pour réaliser l'échange.

Modifier le programme `ex1.c` pour le faire fonctionner. Le résultat attendu est

```
Avant communication:
       0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0

       1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
       1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
       1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
       1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
       1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
       1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0

Apres communication:
       0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0

       0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
       1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
       1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
       1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
       1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
```

B. MPI_Send / MPI_Recv à N processus

À N processus, les processus de rang `0` et `N-1` vont se comporter comme précédemment, mais tous les autres processus de rang `k` vont réaliser deux échanges : la première ligne avec le processus de rang `k-1` et la dernière ligne avec le processus de rang `k+1`. On utilisera donc deux zones tampon `buffer_prev` et `buffer_next`. Copiez le programme précédent vers `ex2.c`, et modifiez-le pour obtenir, à 4 processus,

```
Avant communication:
       0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0

       1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
       1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
       1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0

       2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0
       2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0
       2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0

       3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0
       3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0
       3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0

Apres communication:
       0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0

       0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
       2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0

       1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0  1.0
       2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0
       3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0

       2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0  2.0
       3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0
       3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0  3.0
```

C. Introduction de `MPI_PROC_NULL`

Lorsqu'on donne au fonctions MPI une valeur égale à `MPI_PROC_NULL` comme rang pour le processus source ou destination, l'appel MPI n'a pas lieu :

```
MPI_Send(x , n, MPI_DOUBLE, MPI_PROC_NULL, tag, MPI_COMM_WORLD);
```

Cela permet de simplifier notre programme : si le processus de rang `0` envoie sa première ligne à `MPI_PROC_NULL` et si le processus de rang `N-1` envoie sa dernière ligne à `MPI_PROC_NULL`, tous les processus font strictement le même travail et la branch `if` peut être supprimée.

Copiez le programme précédent vers `ex3.c`, et simplifiez-le grace à l'utilisation de `MPI_PROC_NULL`.

D. MPI_Sendrecv

Chaque processus effectue un appel à `MPI_Send` et un appel à `MPI_Recv` pour échanger une ligne avec un voisin. Copiez le programme précédent vers `ex4.c`, et simplifiez-le en remplaçant ces appels par des appels à `MPI_Sendrecv`.

E. Introduction des zones fantômes

Dans la méthode numérique, nous n'actualisons que les points intérieurs du domaine. Le paradigme SPMD s'applique ici pour les sous-domaines : il faut donc transformer tous les points à calculer en points intérieurs d'un sous-domaine. Nous allons adjoindre à chaque sous-domaine des lignes et colonnes virtuelles ou fantômes qui seront chargées de stocker les informations issues des voisins (figure 2 (b)).

Modifiez le programme `ex5.c` pour obtenir :

```
Avant communication:
...... 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       0.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0  0.0
       0.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0  0.0
       0.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0  0.0
...... 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0

...... 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       0.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0  0.0
       0.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0  0.0
       0.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0  0.0
...... 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0

...... 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       0.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0  0.0
       0.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0  0.0
       0.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0  0.0
...... 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0

...... 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       0.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0  0.0
       0.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0  0.0
       0.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0  0.0
...... 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0

Apres communication:
...... 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
       0.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0  0.0
       0.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0  0.0
       0.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0  0.0
...... 0.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0  0.0

...... 0.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0 10.0  0.0
       0.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0  0.0
       0.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0  0.0
       0.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0  0.0
...... 0.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0  0.0

...... 0.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0 11.0  0.0
       0.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0  0.0
       0.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0  0.0
       0.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0  0.0
...... 0.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0  0.0

...... 0.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0 12.0  0.0
       0.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0  0.0
       0.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0  0.0
       0.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0 13.0  0.0
...... 0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
```

III. PROGRAMME POISSON

Dans cette section nous utiliserons le programme du répertoire `poisson`.

A. Compréhension du programme

La variable `nx` (ligne 46 du fichier `poisson.c`) définit le nombre de points en x. Pour simplifier, nous avons défini $n_y = n_x$. On reconnait l'échange de message des exercices précédents à la ligne 134 du fichier `poisson.c`.

Lorsque le programme s'exécute il réalise les itérations de Jacobi et s'arrête lorsque l'erreur atteint 10^{-10}. Il affiche l'erreur par rapport au résultat exact toutes les 100 itérations. Choisissons d'abord $n_x = 16$. Compilez le programme avec la commande `make` et lancez-le avec 1,2 et 4 processus. Après chaque lancement du programme, exécutez le script `plot.sh`. Ce script affiche d'abord le résultat exact, puis le résultat calculé sur chacun des processus.

B. Complexité

Maintenant, nous nous mettrons dans des conditions où le temps de calcul d'une itération est suffisamment long. Nous choisissons donc $n_x = \{5\ 000, 10\ 000, 15\ 000\}$. Dans ces conditions, le programme n'effectue que 10 itérations.

Étudiez le temps de calcul d'*une itération* en fonction de n_x. Est-ce que ce résultat correspond au résultat attendu ? Expliquez.

C. Efficacité parallèle

Calculez le speedup à 2, 4 processus pour $n_x = \{5\ 000, 10\ 000, 15\ 000\}$.

Recompilez le programme avec l'option `-O3`. Calculez le speedup à 2, 4 processus pour $n_x = \{5\ 000, 10\ 000, 15\ 000\}$. Que remarquez-vous ?

D. Optimisation

Supprimez le calcul de l'erreur. Que constatez vous sur le temps de restitution ? Pourquoi ? Comment pourrait-on accélérer le programme ?