Calcul parallèle et méthode de décomposition de domaine de type Schwarz sur maillage Cartésien régulier

Compt rendu de projet

Nicola Ciliberti, Julien Tenaud

Étudiants en 3ème année à l'ENSEIRB-MATMECA

Spécialité : Calcul haut performance

sous la direction de

Pr. H.Beaugendre

Professeur, ÉCOLE ENSEIRB-MATMECA

Année 2024/2025

1 Sommaire

Table des matières

1 Sommaire 1

2 Table des figures 2

3 Introduction 3

4 Partie I : Équilibre de la charge 3
- 4.1 La charge . . . 3
- 4.2 Les maillages . . . 3
- 4.3 Résultats de Speed-up . . . 4
- 4.4 Parallélisme du code . . . 5

5 Partie II: Developpement de un code de calcul paralléle 6
- 5.1 Algoritme de résolution . . . 7
- 5.2 Problème résolu sur chaque processus . . . 7
- 5.3 Stratégie de parallélisme . . . 7
  - 5.3.1 Distribution de la charge . . . 7
  - 5.3.2 Communication entre processus . . . 8
- 5.4 Test . . . 8
- 5.5 Speed Up . . . 9
  - 5.5.1 En fonction du nombre de processeur . . . 9
  - 5.5.2 En fonction du recouvrement . . . 10
  - 5.5.3 En fonction de la taille du maillage . . . 11

6 Conclusion 11

2 Table des figures

Liste des figures

1 Partition pour 10 sous-domaines 4
2 Comparaison des courbes de Speed-up pour le code éléments finis 6
3 Schéma de comunication avec le recouvrement 8
4 Erreur en fonction du temps avec un pas de temps de 0.1 et 100 itérations sur une grille de 50×50 avec un recouvrement de 4 lignes. 9
5 Speed Up en fonction du nombre de processeurs 10
6 Speed Up en fonction du nombre de lignes de recouvrement 10
7 Speed Up en fonction du nombre de point de maillage pour une condition de bord de Dirichlet 11

3 Introduction

Le but de ce projet est de développer un code de calcul parallèle pour résoudre un problème de diffusion (stationnaire et non-stationnaire) sur un domaine rectangulaire $[0, L_x] \times [0, Ly]$,

$$\begin{cases} \partial_t u(x,y,t) - Du(x,y,t) = f(x,y,t) \quad \text{sur } \bar{\Omega}, & \text{(1a)} \\ u|_{\Gamma_0} = g(x,y,t), & \text{(1b)} \\ u|_{\Gamma_1} = h(x,y,t). & \text{(1c)} \end{cases}$$

où Γ_0 représente les bords haut et bas et Γ_1 les bords droite et gauche.

Ce rapport est divisé en deux parties :

- **Dans la première**, nous explorerons le potentiel (et surtout les limites) du calcul parallèle à travers un processus appelé Speed-up, afin de mesurer ses avantages réels
- **Dans la seconde**, le calcul parallèle sera exploité pour développer un code permettant de résoudre le problème mentionné ci-dessus. Le schéma sera structuré pour être parallélisé, puis une analyse Speed-up sera effectuée pour observer son efficacité.

4 Partie I : Équilibre de la charge

Dans la première partie, l'objectif est de discuter de l'importance de l'équilibre de la charge lorsque l'on développe un code de calcul parallèle pour la résolution des EDP.

4.1 La charge

Le principe fondamental d'un code parallèle est de partager le calcul entre des processeurs différents. Le travail assigné à chaque processeur est appelé *charge*. Sous l'hypothèse que les processeurs ont tous la même puissance de calcul, la configuration la plus efficace est de distribuer la même charge pour chaque processeur. Cependant, ce n'est pas toujours possible. Supposons, par exemple, d'avoir un nombre impair de tâches $N_p + 1$ et un nombre pair de processeurs N_p à disposition. Il y aura évidemment un déséquilibre de charge, mais pour le minimiser, on peut assigner une tâche à chaque processeur et assigner le restant au premier processeur. Généralement, on peut partager s tâches à chaque processeur, tel que

$$s = \arg\max_{s \in \mathbb{N}} (sN_p \leq m)$$

et distribuer le reste aux premiers $m - sN_p$ processeurs disponibles.

4.2 Les maillages

Normalement, on ne considère pas la fonction comme une fonction continue, au contraire, on en prend en compte ses valeurs à certains points, appelés *nœuds*. Le choix du maillage est le premier enjeu de travailler avec le code parallèle. Normalement, on peut diviser les maillages en deux macrocatégories : maillages **structurés** et maillages **non structurés**.

Maillages structurés

Ils sont parmi les maillages les plus simples. Les nœuds sont distribués de façon régulière, c'est-à-dire comme une grille. Cette propriété permet d'identifier chaque nœud avec n (dimension du domaine) indices.

- **Avantages**
  - Bon contrôle de l'épaisseur des mailles (au voisinage des parois à courbure régulière).
  - Facilité pour mailler des géométries très allongées (contrôle aisé du nombre de nœuds dans une direction privilégiée).
- **Inconvénients**
  - Limitation d'emploi aux domaines descriptibles par un quadrilatère (2D) et un hexaèdre (3D).
  - Pas de possibilité de raffiner le maillage sans en augmenter la taille.

Maillages non structurés

Dans le cas d'un maillage bidimensionnel (2D), les éléments qui le constituent sont de type triangle ou encore quadrilatère. Pour une géométrie élémentaire donnée, il est possible de considérer différents nombres de nœuds par élément. La particularité majeure des maillages non structurés est qu'à partir d'une numérotation des nœuds et des éléments qui peut être aléatoire.

- **Avantages**
  - Création de maillages triangulaires ou tétraédriques dans des géométries quelconques et complexes.
  - Économie de points par rapport aux maillages structurés notamment dans les zones de raffinement.
  - Possibilités d'associer différentes topologies d'éléments.
- **Inconvénients**
  - Difficultés pour contrôler la densité des points dans une zone précise.
  - Contrainte de stockage (tableau de connectivités des éléments).
  - Plus longs à générer.

4.3 Résultats de Speed-up

Quand on parallélise le code, un aspect important à prendre en considération est la vitesse de calcul qu'on a gagnée. Idéalement, quand on partage le travail entre m processeurs, le temps que le code prend normalement sur un processeur ($T(1)$) devrait être divisé par le nombre de processeurs. Le rapport

$$S(p) = \frac{T(1)}{T(p)}$$

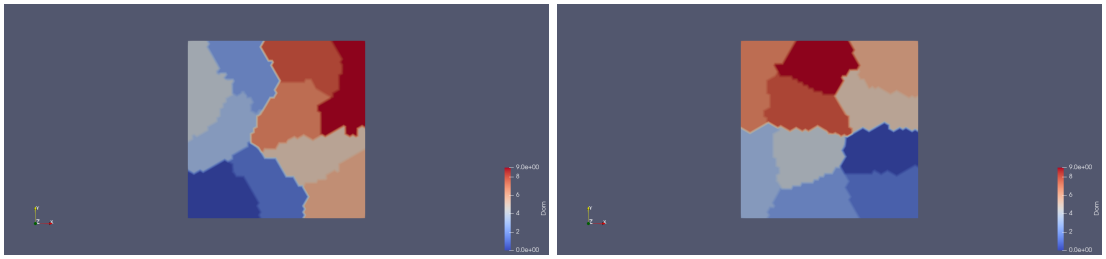
est appelé *speed-up*, avec $T(p)$ le temps passé avec p processeurs. Idéalement, ça doit être linéaire. En réalité, il y a une limite imposée par la *loi d'Amdahl*. Soit f la fraction du code parallélisable et p le nombre de processeurs :

$$S(p) \leq \frac{1}{(1-f) + \frac{f}{p}}$$

Pour tracer nos courbes, nous avons travaillé avec **METIS** et **SCOTCH**. Ces deux programmes ont comme objectif de générer et partager un domaine (dans notre cas carré) en n sous-domaines. Ils ne sont pas équivalents (en fait, les partitions sont toutes différentes), mais qualitativement, le résultat est le même. Pour tracer les courbes, nous avons considéré le problème du TP des éléments finis, autrement dit, résoudre le problème :

$$\begin{cases} \Delta u = 0 & \Omega \\ g(x,y) = x + y & \partial\Omega \end{cases}$$

En **Figure 1** un exemple de partition METIS et SCOTCH pour 10 sous-domaines. On se rend compte, que quelque soit le partitionneur, les domains ont un nombre de mailles à peu près équivalent. En revanche METIS et SCOTCH n'adopte pas la même stratégie de découpage.



(a) Patition METIS (b) Patition SCOTCH

Figure 1: Partition pour 10 sous-domaines

4.4 Parallélisme du code

Sur un domaine avec un nombre de sous-domaines progressif, équivalent au nombre de processeurs utilisés pour le test.

$$\#\text{ de procs} = (2, 3, 4, 7, 10, 15, 20, 24)$$

Le code va résoudre le problème individuellement sur chaque sous-domaine. Le problème se met sous la forme matricielle suivante :

$$Kv = b \text{ avec } K_{ij} = \int_{Triangle} a * \nabla\phi_j . \nabla\phi_i dx$$

Le système est stocké sous forme d'une matrix block pour chaque sous-domaine

$$K = \begin{pmatrix} A_1 & & & & B_1 \\ & A_2 & & & B_2 \\ & & \ddots & & \vdots \\ & & & A_p & B_p \\ C_1 & C_2 & \dots & C_p & A_s \end{pmatrix}, \quad v = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_p \\ v_s \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_p \\ b_s \end{pmatrix}$$

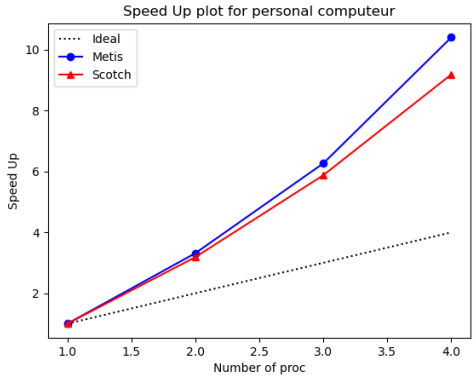
où les C_i sont les matrice transposée des B_i et les v_i sont les vecteurs inconnus avec les éléments dénotés $*_s$ appartenant au bord

Chaque processeur va lire sa portion de maillage et construire et construire son propore système avec les matrices A_p, B_p, A_s correspondant aux A_i, B_i, A_{si} et les vecteurs F_p et F_s correspondant aux vecteurs b_i et b_{si}. Une fonction permet d'effectuer le produit matric vecteur tout en échangant via des *MPI Send* et *MPI Recv* les données de bord entre processeurs. La résolution se fait de manière implicite avec un algorithme du Gradient conjugé qui appelle la fonction effectuant le produit matric vecteur.

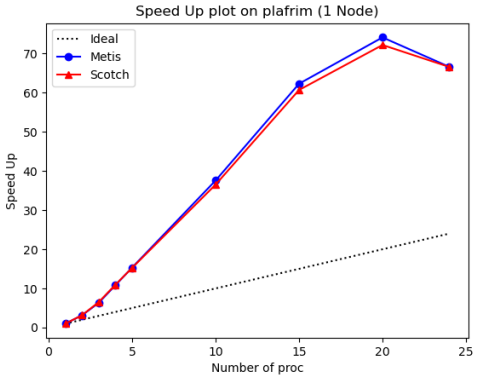
Pour le speed-up, nous avons mesuré le temps pour effectuer le calcul sur le domaine entier $T(1)$ et, pour chaque nombre de processeurs, le temps moyen sur chaque sous-domaine :

$$S(p) = \frac{T(1)}{\frac{1}{n}\sum_{i=1}^{p} T_i(p)}$$

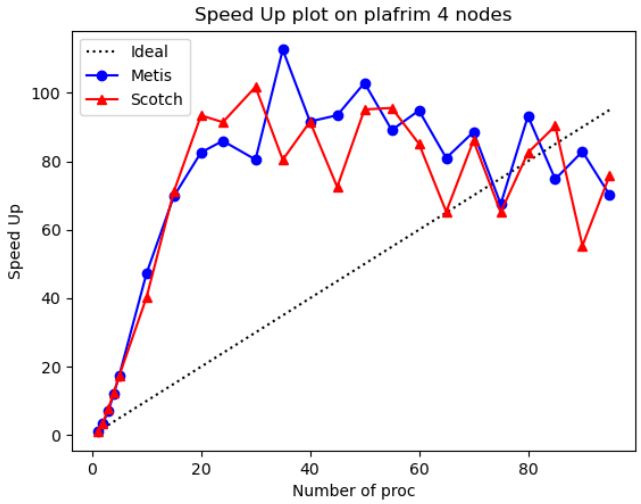
où $T_i(p)$ est le temps sur le sous-domaine i avec p processeurs. Les résultats sont montrés dans la **Figure 2**



(a) Speed Up réalisé sur un ordinateur personel avec une disponibilité de 4 processeurs



(b) Speed Up réalisé sur un noeud de Plafrim réservé de manière exclusive et comportant 24 processeurs



(c) Speed Up réalisé sur 4 noeud de Plafrim réservés de manière exclusive

Figure 2: Comparaison des courbes de Speed-up pour le code éléments finis

Concernant les figures 2b et 2c elles ont été obtenu sur la machine distante plafrim. Afin de compiler le code éléments finis nous avons du charger sur plafrim le module de compilation *gcc* ainsi que *mpicc*. L'obtention des partition SCOTCH et METIS avait préalablement été faite sur une machine personelle et copié sur l'espace plafrim ensuite.

D'après la Figure 2, que se soit avec un ordinateur personel comportant seulement 4 processeurs non exclusif ou avec plafrim, il y a un super speed up du code quand le nombre de processeur reste "petit". Ce super speed up peu s'expliqué de plusieurs manière. En parallèle, chaque processeur accède à un sous-domaine plus petit, ce qui peut réduire la latence mémoire et augmenter les taux de transfert mémoire. Aussi le préconditionement par bloc de la matrice K adaptés à la décomposition en sous-domaines permet d'accélérer la convergence du solveur itératif.
Arriver à un certain nombre de processeur le temps ne s'améliore plus, on a atteint un plateau imposé par la taille du système résolu. Une autre remarque possible est la présence de plusieurs noeuds qui peuvent ralentir le processus car les comunications entre noeuds sont plus longues.

5 Partie II: Developpement de un code de calcul paralléle

Le but de la deuxiéme partie du projet est de developper un code de calcùl paralléle pour resoudre un probléme de diffusiòn avec la decomposition du domaine de Schwartz-Robin. Le probléme est le suivant:

Soit un domaine $\Omega = [0, L_x] \times [0, L_y]$. *Trouver* u *tel que*

$$\begin{cases} \partial_t u - D\Delta u = f & \Omega \\ u = g & \Gamma_0 = [0, L_y] \times [0, L_x] \\ u = h & \Gamma_1 = [0, L_x] \times [0, L_y] \end{cases}$$

5.1 Algoritme de résolution

Le domaine de calcul est une grille 2D discrétisée en points avec des espacements h_x et h_y. Les dimensions globales du domaine sont N_x et N_y en termes de points de discrétisation.

Le domaine est divisé en P sous-domaines, où P représente le nombre de processus MPI disponibles. Chaque sous-domaine est constitué d'un certain nombre de lignes (ou "blocs de lignes"), ce qui signifie que la décomposition est réalisée dans la direction y. Chaque processus gère une région définie dans la direction y, tout en couvrant la totalité des N_x points dans la direction x.

5.2 Problème résolu sur chaque processus

Localément, chaque processus résout une l'équation discrétisée sur le sous-domaine assigné, en considérant les contributions suivantes:

- **Conditions internes:** Les points internes sont calculés sans interaction avec les autres processus. En particulier en résolvant le probléme:

$$\begin{cases} \partial_t u_K^{n+1} - D\Delta u_K^{n+1} = f \quad \text{sur } \Omega_K \\ u_K^{n+1}|_{\partial\Omega_K \bigcap \Gamma_0} = g \\ u_K^{n+1}|_{\partial\Omega_K \bigcap \Gamma_1} = h \\ a\frac{\partial u_K^{n+1}}{\partial n_K} + b u_K^{n+1} = a\frac{\partial u_{K'}^{n}}{\partial n_{K'}} + b u_{K'}^{n} \end{cases}$$

  Pour $K = 0, ..., P-1$ où P est le nombre de processeurs. Ici ce sont des condition aux limites de Robin entre processeur, mais en prenant $a = 0$ et $b = 1$ on retrouve des condition de Dirichlet.

- **Recouvrement avec les voisins:** Les points situés aux frontières du sous-domaine nécessitent des données des processus voisin, c'est-à-dire $U(i, j-k)$ ou $U(i, j+k)$ ($k \in \mathbb{N}$ depend du recouvrement) pour les échanges dans la direction y. Une déscription plus emple des communications entre processeur est faite par la suite.

5.3 Stratégie de parallélisme

La stratégie repose sur l'utilisation de MPI pour partager les calculs entre les processus :

5.3.1 Distribution de la charge

Afin de distribuer au mieux la charge selon la direction y (répartition en ligne) nous utilisons la formule suivante :

```
q = N/n_proc
r = N - q*n_proc

if (me < r) then
    ibeg = me * (q+1) + 1
    iend = (me + 1) * (q + 1)
else
    ibeg = 1 + r + me * q
    iend = jbeg + q - 1
end if
```

où N correspond ici à notre nombre de ligne total selon y, (Ny). Afin d'ajouter convenablement le recouvrement nous avons adopter la stratégie suivante :

```
jbeg = MAX(1, ibeg)
jend = MIN(n, iend + overlap)
```

où *overlap* correspond au nombre de lignes de recouvrement entre 2 processeur. Nous verrons par la suite que le recouvrement impacte le calcul et dépend des conditions aux limites choisies entre les processeurs

5.3.2 Communication entre processus

Les processus échangent les données nécessaires entre les sous-domaines adjacents (couches de recouvrement). En fonction des conditions aux bords, chaque processeur échange à ses voisins des données différentes.

- **Dirichlet**: Les conditions de Dirichlet entre processeur s'écrivent:

$$u_K^{n+1} = u_{K'}^n$$

On communique les lignes de bord de processus $me+1$ et $me-1$ (**Figure 3**). Il y a donc une seul ligne qui est comuniquée entre processeur voisin ceux qui fait en tout deux, sauf pour le processeur 0 et le dernier (qu'on appelera par la suite processeur np) qui ne possède qu'un voisin.

- **Neumann/Robin**: Avec ces conditions on à l'apparition d'une dérivé.

$$a\partial_{n_K} u_K^{n+1} + bu_K^{n+1} = a\partial_{n_{K'}} u_{K'}^n + bu_{K'}^n$$

On choisit d'utiliser un schéma centrer d'ordre 1 afin d'approcher cette dérivée. Il faut donc dans notre cas utiliser 3 lignes pour le calcul de la dérivée : celle de bord, la précédente et de manière fantôme la suivante. Pour éviter de communiquer 6 lignes on calcul avant l'envoi la partie nécessaire et on l'envoi sous forme d'une ligne. En somme, les processeurs communiqueront 2 lignes sauf les processeur 0 et np qui en communiqueront 1.

Rem: Il est nécessaire d'avoir un recouvrement supérieur ou égale à 2 (overlap ≥ 2) pour que les condition de Neumann ou Robin soit admissible avec une méthode de Schwarz additive et avec un schéma décentré d'ordre 1.

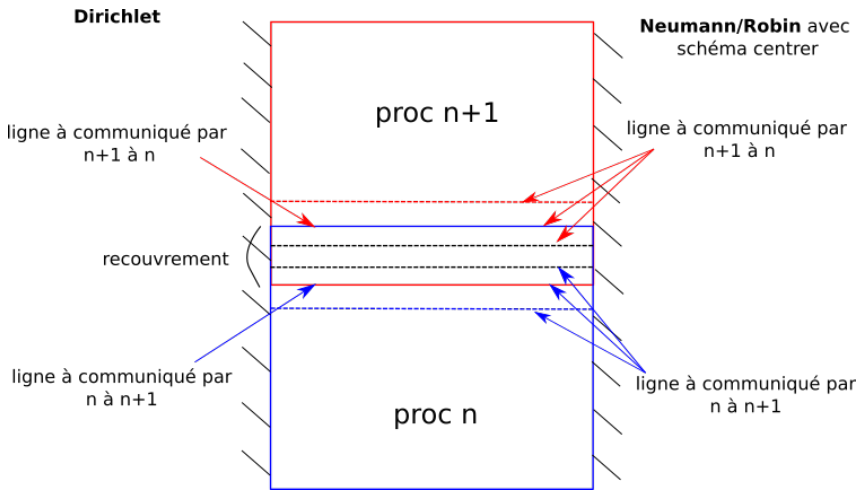


Figure 3: Schéma de comunication avec le recouvrement

Pour la stratégie de communication nous avons choisit de communiquer toutes les lignes correspondant aux frontières hautes des processeurs en excluant le processeur np et toutes les lignes correspondant aux frontières basse des processeur en excluant le processeur 0. Les fonctions `MPI_SEND` et `MPI_RECV` permettent l'envoi et la réception des données.

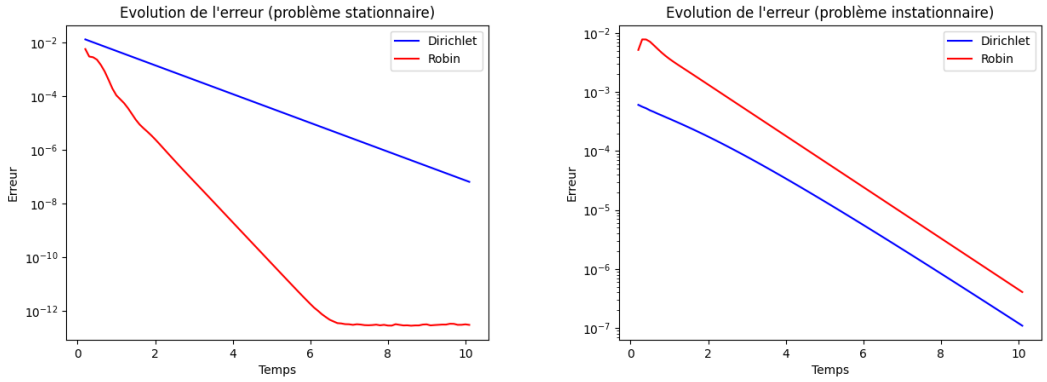
5.4 Test

Nous avons tous d'abord testé notre programme sur les trois cas présentés dans l'énoncé afin de valider notre code. Puis nous avons établie la solution manufacturer suivante :

- **Solution exacte :** $sin(\pi x)sin(\pi y)e^{-t}$
- **Condition initial :** $sin(\pi x)sin(\pi y)$
- **Terme source :** $(2\pi D - 1)sin(\pi x)sin(\pi y)e^{-t}$

- **Condition de bord :** Dirichlet homogène

Cette solution nous permet d'effectuer nos tests de speed up présentés dans la section suivante. Ici nous proposons d'afficher l'erreur en fonction du temps pour le problème stationnaire n°1 présenté dans l'énnoncé et pour le problème instationnaire manufacturé ci-dessus.



(a) Problème stationnaire

(b) Problème instationnaire

Figure 4: Erreur en fonction du temps avec un pas de temps de 0.1 et 100 itérations sur une grille de 50×50 avec un recouvrement de 4 lignes.

On se rend compte que pour le problème stationnaire la convergence avec des conditions de Robin est meilleure qu'avec des conditions de Dirichlet aux frontières interieurs. En revanche, pour le problème instationnaire les conditions se valent.

5.5 Speed Up

Dans cette partie, nous proposons de regarder différentes courbes de speed-up afin d'analyser l'influence de certains paramètres comme le recouvrement ou encore la taille du maillage. L'étude a été menée sur la solution manufacturée présentée dans la partie précédente

5.5.1 En fonction du nombre de processeur

Afin de faire une étude de speed up en fonction du nombre de processeurs nous avons choisi de faire tourner notre programme sur l'ordinateur distant, Plafrim. Pour cela il a fallut charger les module idoine pour pouvoir fair la compilation avec *mpif90*:

```
module load compiler/gcc
module add compiler/intel
module add mpi/intel
module add mpi/openmpi
```

En ce qui concerne le problème nous avons choisit de prendre une grille de 200×200 afin que la courbe de speed up ne sature pas trop rapidement. Nous avons pris 100 itérations avec un pas de temps $dt = 0.1$ et un recouvrement de 4 lignes. Dans cette expérience et par la suite les constantes a et b, dans les conditions de bord de Robin entre processeur, seront égale à 1.

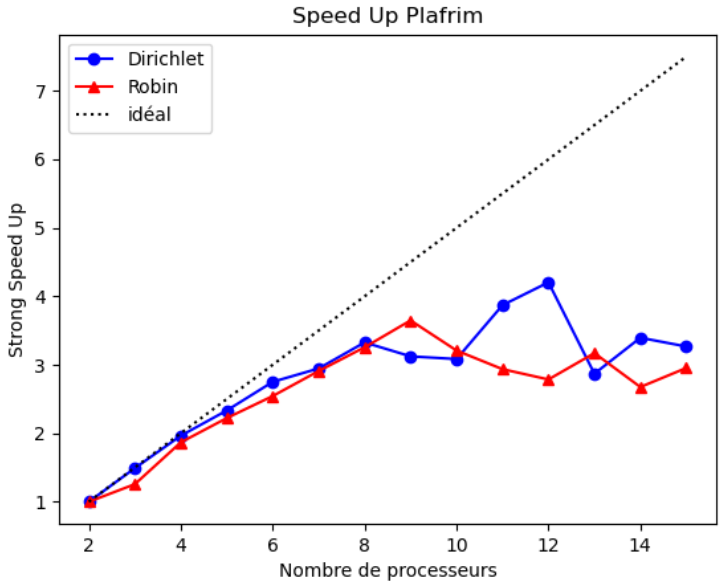


Figure 5: Speed Up en fonction du nombre de processeurs

La courbe de speed up est d'abord linéaire et suit le speed up idéal. Arrivé à 6 processeurs on observe un saturation du temps de calcul. En effet à partir d'un nombre de lignes trop faible par processeur le temps de gagné par rapport à l'augmentation du nombre de processeur ne compense plus le temps perdu par l'augmentation du nombre de communication. On à donc un plateau qui se crée, voir une diminution du temps de calcul.

5.5.2 En fonction du recouvrement

Nous avons choisi d'effectuer cette étude de speed up en utilisant notre machine personnelle. Celle-ci ne possède que 4 processeurs indépendants mais pour comprendre l'influence du recouvrement nous avons déduit qu'il n'était pas nécessaire d'avoir un grand nombre de processeurs. Pour tracer cette courbe nous avons pris un pas de temps $dt = 0.1$, une taille de maillage de $N = 100 \times 100$, un nombre d'itération de 50, un coefficient de diffusion de $D = 1.0$ et avons considéré des recouvrements allant de 2 lignes à 14 lignes (**Firgure 6**).

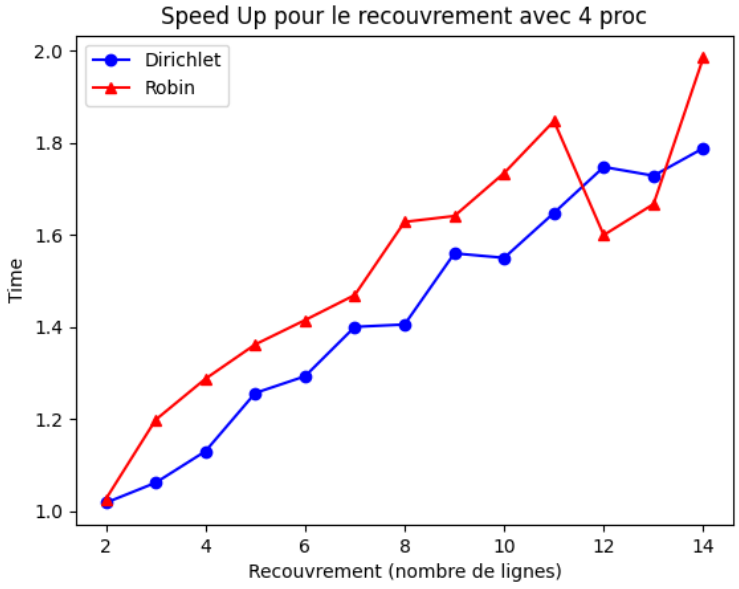


Figure 6: Speed Up en fonction du nombre de lignes de recouvrement

Quelque soit la condition de bord entre les processeurs, on observe sur la **Figure 6** que le temps minimal

d'exécution correspond à un recouvrement de 2 lignes. Aussi on remarque que le temps d'exécution tend à croître en fonction du recouvrement. En effet, plus le recouvrement est grand et plus le nombre de lignes calculées deux fois est, de même, grand. On perd donc en temps d'exécution. Sur cette figure certains temps de calcul pour des recouvrements plus grands sont plus faibles que les précédents. Cela est dû au fait que le calcul n'est pas exécuté sur des processeurs réservés exclusivement. Le temps dépend donc en partie d'une charge latente liée à l'exécution d'autres tâches sur l'ordinateur.

5.5.3 En fonction de la taille du maillage

De la même manière que précédemment cette étude s'est faite sur notre machine personnelle. Pour effectuer ce speed up, nous avons choisit un pas de temps de $dt = 0.1$, un nombre d'itération de 20, un coefficient de diffusion de $D = 1.0$, un recouvrement de 3 et avons considéré un domaines carré avec autant de points dans la direction x que y en faisant varier N_x de 10 à 300 points (**Figure 7**). Pour les condition au bord de Robin nous avons $a = 1.0$ et $b = 1.0$

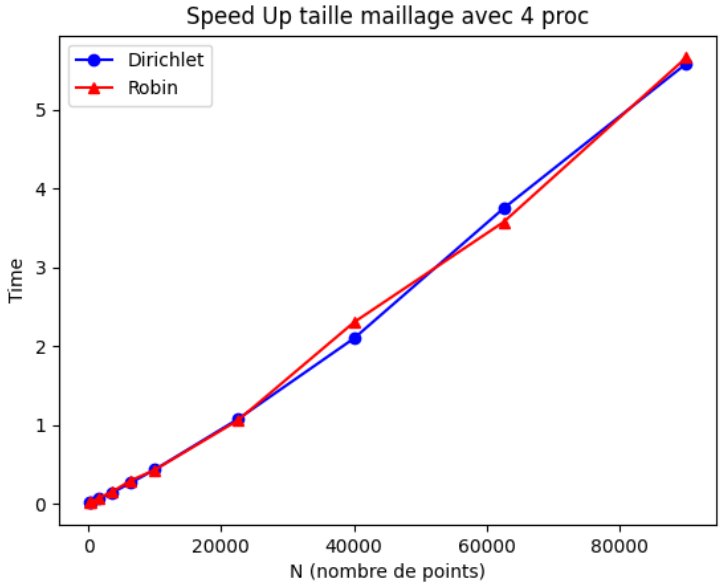


Figure 7: Speed Up en fonction du nombre de point de maillage pour une condition de bord de Dirichlet

Sur la (**Figure 7**) on remarque une évolution quasiment linéaire entre le temps d'exécution et le nombre de points de maillage. On se rend aussi compte que malgré la complexité de calcul accrue par l'implémentation de conditions de Robin, cela n'a pas un impact significatif sur le temps de calcul.

6 Conclusion

Le développement d'un code parallèle pour la résolution d'une équation de diffusion a permis d'illustrer les concepts fondamentaux du calcul parallèle, notamment la décomposition de domaine et les communications interprocessus. En particulier, nous avons pu constater les avantages du processus de parallélisation, qui est un outil très puissant mais également limité. Son utilisation doit être fait dans des cas adéquats aux risques de perdre en efficacité. Dans la stratégie de parallélisation et de résolution même de nombreuse étude peuvent être mener pour accélérer d'avantage le speed up du code. Pour des codes de recherche ou industrielles très lourds cela est indispensable pour pouvoir les faire tourner sans prendre un temps trop important et dépenser une énergie considérable.