Compte rendu :
Calcul Haute Performance

Correctrice : Héloïse Beaugendre

10 mai 2017

Gabriel BALESTRE

Théo ROBUSCHI

Deuxième année, filière Mathématiques et Mécanique

École Nationale Supérieure, d'Électronique, Informatique, Télécommunications,
Mathématique et Mécanique de Bordeaux

Introduction

Au cours de nos projets nécessitant l'utilisation de code de calcul, nous avons pu remarquer que ceux-ci pouvaient nécessiter un temps de calcul important. Dès lors, la parallélisation des codes est un moyen de réduire le temps de calcul.

L'objectif de ce projet de Calcul Haute Performance est de paralléliser la méthode du gradient conjugué et d'étudier l'effet de la parallélisation sur l'efficacité et le speed-up lié à celle-ci. Pour la réalisation code code, nous avons travaillé en Fortran90 en utilisant la librairie MPI.

1 Analyse mathématique

Pour ce projet, nous avons à paralléliser le problème suivant, il s'agit de résoudre l'équation de la chaleur sur un domaine 2D.

$$\begin{cases} \partial_t u(x,y,t) - D\Delta u(x,y,t) = f(x,y,t) \\ u_{\Gamma 0} = g(x,y,t) \\ u_{\Gamma 1} = h(x,y,t) \end{cases} \tag{1}$$

Avec Γ0 et Γ1 les bords du domaine.
Afin de vérifier l'implémentation du code, différentes conditions aux bords et terme source ont été fournies à savoir :

- $f = 2(y - y^2 + x - x^2)$, $g = 0$, $h = 0$
  $u_{exa} = x(1-x)y(1-y)$
- $f = cos(y) + sin(x)$, $g = cos(y) + sin(x)$, $h = cos(y) + sin(x)$
  $u_{exa} = sin(x) + cos(y)$
- $f = e^{-(x - \frac{Lx}{2})^2} e^{-(y - \frac{Ly}{2})^2} cos(\frac{\pi}{2}t)$, $g = 0$, $h = 1$

Les deux premières conditions initiales ont une solution exacte connue. Cette solution exacte nous permettra de quantifier l'erreur entre la solution théorique et le calcul numérique.
Nous avons discrétisé ce problème en utilisant le schéma d'Euler implicite et des différences finies centrées pour discrétiser le Laplacien. Ce qui nous a conduit à l'écriture du schéma numérique.

$$\frac{u_{i,j}^{n+1} - u_{i,j}^{n}}{\Delta t} - D\left(\frac{u_{i+1,j}^{n+1} - 2u_{i,j}^{n+1} + u_{i-1,j}^{n+1}}{\Delta x^2} + \frac{u_{i,j+1}^{n+1} - 2u_{i,j}^{n+1} + u_{i,j-1}^{n+1}}{\Delta y^2}\right) = f(i,j,n+1)$$

Ainsi, nous avons pu construire le système matriciel suivant :

$$A = \begin{pmatrix} A_{N_y} & B_{N_y} & 0 & \cdots & 0 \\ B_{N_y} & A_{N_y} & B_{N_y} & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & B_{N_y} & A_{N_y} & B_{N_y} \\ 0 & \cdots & 0 & B_{N_y} & A_{N_y} \end{pmatrix}$$

La matrice A est composée de N_x blocs de taille $N_y \times N_y$ et est donc une matrice de taille $N_x \times N_y$. Avec A_{N_y} et B_{N_y} des matrices de taille $N_y \times N_y$ définies ainsi :

$$A_{N_y} = \begin{pmatrix} \alpha & \beta_y & 0 & \cdots & 0 \\ \beta_y & \alpha & \beta_y & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \beta_y & \alpha & \beta_y \\ 0 & \cdots & 0 & \beta_y & \alpha \end{pmatrix} \quad \text{et} \quad B_{N_y} = \begin{pmatrix} \beta_x & 0 & \cdots & \cdots & 0 \\ 0 & \beta_x & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \beta_x & 0 \\ 0 & \cdots & \cdots & 0 & \beta_x \end{pmatrix}$$

Les coefficients α, β_x et β_y sont définies par :

$$\alpha = \frac{1}{\Delta t} + \frac{2D}{\Delta x^2} + \frac{2D}{\Delta y^2}, \quad \beta_x = -\frac{D}{\Delta x^2}, \quad \beta_y = -\frac{D}{\Delta y^2}$$

Ainsi, avec une telle écriture de la matrice, nous pouvons observer que celle-ci est tri-diagonale par bloc avec le bloc A_{N_y} lui même tri-diagonal et le bloc B_{N_y} diagonal. De plus, nous pouvons voir que cette matrice est creuse il est donc simple de d'envisager les calculs à partir de cette matrice sans avoir à stocker les éléments de celle-ci.

2 Parallélisme

Dans cette partie, nous allons détailler les éléments important du code parallèle à savoir les types de communications et opérations utilisées pour la réalisation du code.

Pour effectuer le calcul du gradient conjugué en utilisant un code de calcul parallèle chaque processeur va devoir utiliser un certain nombre de variables différentes qui seront soit locales soit globales. Par exemple, nous pouvons citer comme variable globale la valeur des différents coefficients de la matrice précédemment exposée à savoir $\alpha, \beta_x, \beta_y, dt, D$. Cependant, certains éléments n'ont pas nécessairement besoin d'être connu par tous les processeurs comme par exemple la matrice entière A et le vecteur qui la multiplie. Il vient donc la nécessitée de réaliser des communications entre les processeurs car certains éléments, mais pas la totalité, des autres processeurs sont nécessaires aux calculs des produits scalaires de la méthode du gradient conjugué.

Dans notre code, les communications nécessaires entre les différents processeurs sont effectuées en début de boucle du gradient conjugué où l'on envoie et reçois aux processeurs

directement précédent et suivant. En effet, pour réaliser les différents calculs pour le processeur me, celui-ci à besoin d'éléments contenus dans les $me+1$ et $me-1$. Ainsi, me va recevoir de $me-1$ et $me+1$ un message de taille N_y et me va envoyer à $me-1$ et $me+1$ un message de taille N_y contenant la partie haute ou basse du vecteur nécessaire au calcul du produit matrice vecteur.

Nous avons par exemple utilisé la ligne de code suivante lorsqu'il s'agissait d'envoyer au $me+1$ les éléments du vecteur nécessaire pour le calcul du produit matrice vecteur :
CALL MPI_SEND($x(i_N - N_y + 1 : i_N)$*, ny, MPI_FLOAT,* $me+1$*,* $tag + 2 * me + 1$*, MPI_COMM_WORLD, statinfo).* Dans cette ligne, nous pouvons aussi voir que nous avons numéroté différemment chaque message afin de ne pas avoir de problème lors de l'appel de réception du message. La taille du message est bien de N_y ce qui correspond à la ligne directement supérieure de la matrice nécessaire au calcul du produit scalaire.

L'algorithme du gradient conjugué nécessite aussi de calculer à plusieurs reprises des coefficients qui proviennent du calcul d'un produit scalaire de deux vecteurs. Or dans le code parallèle seul des sections de vecteur de taille $(i_1 : i_N)$ sont connues par chaque processeur. Il a donc été nécessaire d'utiliser des instructions du type *MPI_ALLREDUCE* afin de réaliser un calcul et de la communiquer à tous les processeurs et ici les calculs réalisés sont des sommes de chacun des produits scalaires locaux.

Une fois la mise en place du parallélisme terminée, nous nous sommes attachés à la vérification des résultats ainsi qu'au calcul des efficacités et speed-up pour cette méthode du gradient conjugué.

3 Résultats et exploitation

Dans cette partie, nous allons détailler la méthode de validation du code ainsi que les différents résultats obtenus à propos de la parallélisation du code.

Dans un premier temps, nous avons codé la méthode du gradient conjugué en séquentiel afin de valider le code avant de le paralléliser. En effet, nous avons pu comparer les résultats de la solution numérique avec les résultats issus de la solution exacte. Dans notre cas, les erreurs obtenues sont de l'ordre de 10^{-6} ce qui est une erreur qui est difficilement réductible. Une fois ce travail sur le codage des conditions aux limites, de la solution exacte et de la méthode du gradient conjugué était terminé, nous avons pu paralléliser le code.

Il en est donc découlé une nouvelle phase de vérification des résultats du code parallèle avec les solutions exactes afin de vérifier que la parallélisation n'ait pas altéré nos résultats. A nouveau, nous avons trouvé des erreurs de l'ordre de 10^{-6} et donc comme nous l'avons exposé précédemment cette erreur étant acceptable, nous avons considéré le code fonctionnel. Ainsi, nous avons pu étudier les propriétés et les effets de la parallélisation du code.

Dans un second temps, nous avons étudié les effets de la parallélisation du code notamment en étudiant les Speed-up pour les différents cas tests et différentes taille de matrice. Les principaux résultats obtenus sont exposés dans les graphes et tables suivantes.

Par exemple, pour une matrice de taille 1000×1000, et en utilisant la première condition initiale, on obtient l'efficacité de la Figure 1. En utilisant un ordinateur comportant 8 processeurs, nous pouvons observer que l'efficacité du calcul décroît avec l'augmentation du nombre de processeurs. Cela est du aux communications entre les différentes processeurs qui retardent l'avancée globale du code. En effet, pour commencer une nouvelle itération du gradient conjugué il est nécessaire que les processeurs communiquent entre eux différents éléments comme expliqué dans la partie précédente. Dès lors, l'efficacité de notre code est limitée par ces communications est engendre un déclin de celle-ci comme nous pouvons le voir sur la Figure ci-dessous.

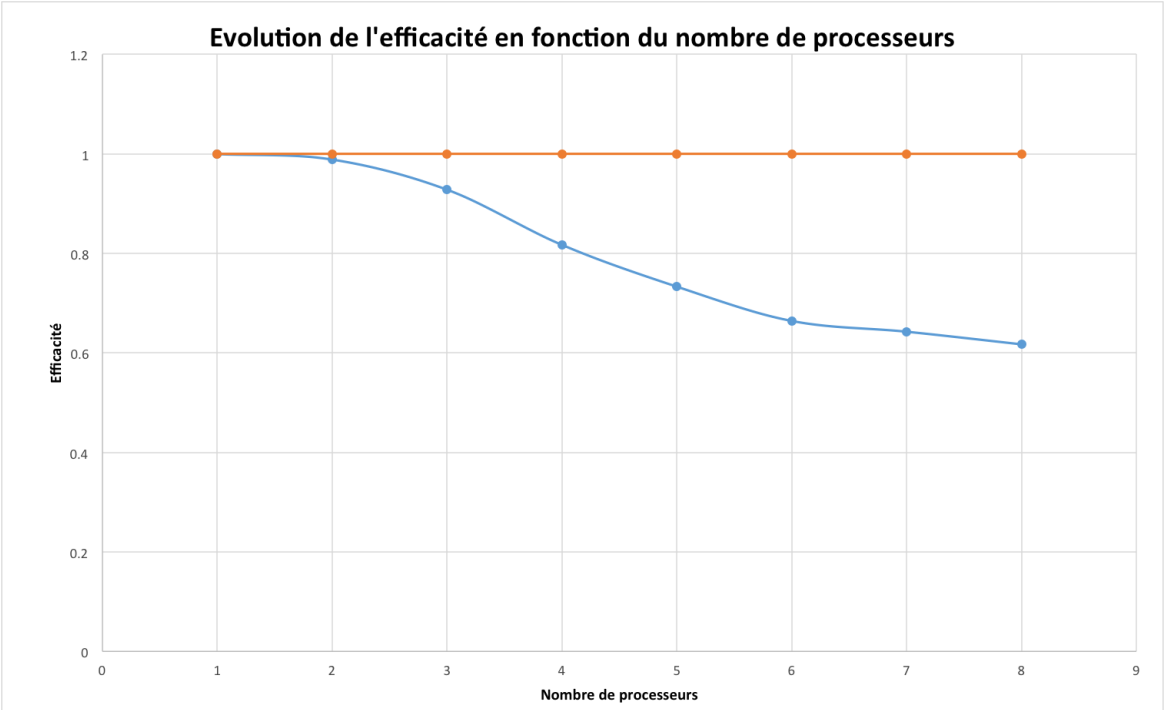

FIGURE 1 – Évolution de l'efficacité en fonction du nombre de processeurs.

D'après le cours, nous savons que lorsque que l'efficacité devient inférieure à 70% il n'est pas nécessaire d'augmenter le nombre de processeurs utilisés. Dans notre cas, nous avons étudié sur le nombre de processeurs disponibles afin de mettre en avant cette propriété issue du cours. En effet, nous pouvons voir sur ce graphe qu'au delà de 5 processeurs il n'est plus utile d'augmenter le nombre de processeurs utilisés pour le cas d'étude proposé dans le sujet.

Il a aussi été possible de déterminer le speed-up en fonction du nombre de processeurs, à nouveau nous avons utilisé tous les processeurs proposé par le matériel afin de conduire une étude approfondie du speed-up. Une nouvelle fois, nous comparons le speed-up théorique (en orange) par rapport à celui qui a été observé (en bleu).

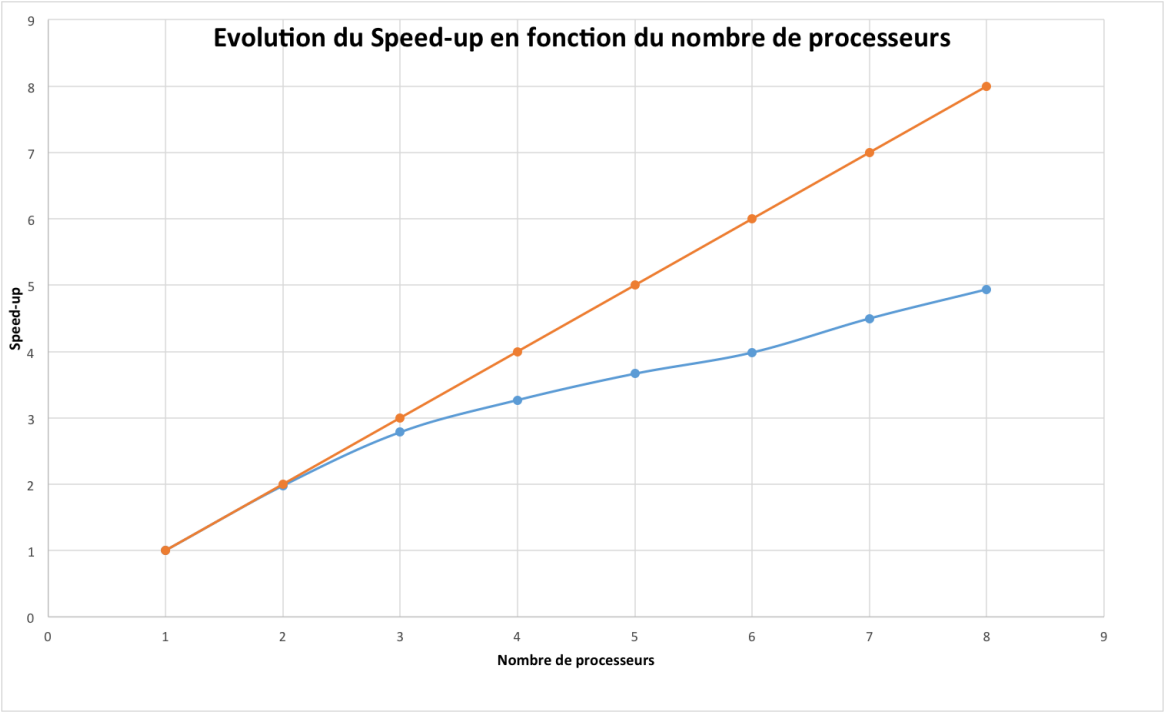

FIGURE 2 – Évolution du Speed-Up en fonction du nombre de processeurs.

Nous pouvons voir que le speed-up tend à ne pas dépasser une limite pour un nombre de processeurs important, c'est une fois de plus un moyen de conforter la proposition énoncée dans le cours. En effet, sur cette courbe, nous pouvons constater que le speed-up passe de 4 à 5 pour un nombre de processeurs passant de 5 à 8.

Nous avons ainsi à l'aide de ces deux graphiques pu caractériser l'effet de la parallélisation sur un code de calcul. Nous avons pu observer l'évolution de l'efficacité et du speed-up en fonction du nombre de processeurs dans des cas tests simples afin de mettre en évidence les spécifitées d'un tel code vis-à-vis d'un code séquentiel. Il n'est donc pas évident comme nous avions pu l'envisager d'utiliser un code parallèle, puisqu'il n'y a pas linéarité des propriétés en fonction du nombre de processeurs utilisé. Ainsi, lors de l'utilisation d'un code parallèle augmenter le nombre de processeurs utilisé n'est pas nécessairement un gage de l'augmentation de la performance d'un tel code.

Conclusion

Lors d'une étude sur un projet demandant un nombre important de calcul, il est intéressant de paralléliser le code pour diminuer le temps de calcul, cependant il demande plus de rigueur.

En effet, comme nous avons pu le constater précédemment, il n'est pas nécessaire d'augmenter le nombre de processeur si l'efficacité est inférieure à 70% et ce n'est pas non plus un gage de l'augmentation de la performance des calculs. Si nous demandons par exemple l'utilisation d'un nombre de processeurs plus important que le nombre de processeurs physiquement disponible dans la machine il y aura alors recouvrement des calculs sur un même processeur et donc une augmentation significative du temps de calcul. Donc, avant d'utiliser une code de calcul parallèle, il faut s'informer sur celui-ci, connaître ces courbes de speed-up et d'efficacité et connaître la machine sur laquelle nous travaillons.