Avant-propos

Ce document fait l'objet d'un complément de cours & travaux dirigés TD pour le module « Méthodes numériques Appliquées » enseigné pour Master I enérgetiques. L'idée de base est :

- de permettre aux étudiants de Master I de compléter leurs notions dans les méthodes numériques appliquées en consultant la première désignée par A, de ce document tout en intégrant le concept du calcul parallèle orienté vers le MPI qu'on présente dans la partie B (seconde).
- de présenter une initiation au calcul parallèle par MPI (message passing interface) deuxième partie par désignée par A en se focalisant sur les notions déjà acquises en cours exposée dans la section A (chapitre 03).

Deux raisons nous ont poussé à considérer ce travail et à le proposer comme complément de cours à nos étudiants.

- En premier lieu, en consultant ces notions développées dans ce document (chapitre 02) et en travaillant sur les différentes applications (chapitre 03), l'étudiant maîtrisant la programmation linéaire du module « Méthodes Numériques Appliquées », intégrera cette nouvelle tache avec beaucoup de maitrise.

Ce document est basé sur une présentation du calcul MPI dans un but d'enrichir ce module que j'enseigne au sein de notre « Département de Génie Mécanique »; et que je désire l'inscrire comme un complément dans le calcul scientifique et permettre à notre Université une contribution dans ce domaine.

- Par ailleurs, l'implémentations MPI Open source désigne le deuxième argument :

Avec les notions acquises en programmation linéaire par l'utilisation du Fortran 90 et l'accessibilité à la librairie MPI par la distribution Linux, la librairie MPI difficilement accessible dans la version Windows (coût élevé de la licence d'exploitation environ 1000 $), ce type de calcul devient à porter de main pour chaque étudiant par le biais du Gfortran (open source).

Table des matières

Introduction générale

Introduction générale

La simulation numérique est devenue aujourd'hui incontournable pour la totalité des domaines scientifiques à savoir les secteurs technologiques et industrielles. Un large de spectre de travaux scientifiques et d'investigations expérimentales emploient des méthodes basées sur le concepte de modélisation voire la simulation numérique pour valider les résultats; il n'existe plus un projet industriel d'envergure voire un travail de recherche où la simulation n'a pas été utilisée lors de la réalisation.

Les chercheurs dans différentes disciplines peuvent ainsi recourir à des investgations numériques de différents phénomènes pour lesquels il aurait été difficile et coûteux de disposer d'un travail expérimental, comme dans le cas de la nanotechnologie ou la combustion avec de nouveaux conceptes et procédés (cleaner combustion) cas de la cinétique chimique nécéssitant des calculs développés .

Ainsi, avec les nouveaux défis scientifiques (problèmes d'énergie, les nanotechnologies, les maladies infectieuses, la climatologie et les big datas, les problèmes d'émissions ... etc) qui ne cessent de s'inviter dans notre quotidien, ces nouvelles méthodes et procédés de calculs deviennent vitales et nécessaires. Cela, nous pousse à développer ces paradigmes et à étudier ces process pour une meilleure gestion. Comme dans le cas de l'utilisation à la recherche de gros calculateurs et à modifier tout nos méthodes d'approche dans le calcul scientifique comme le cas des calculs paralléles.

Le besoin en puissance de calcul de ces simulations est considérable; plus la puissance de calcul disponible est importante, plus les modèles établis peuvent être précis, ou plus la taille des objets simulés peut être augmentée. Cette course à la puissance, demande des innovations matérielles continuelles et des process de calcul évolués afin de concevoir des supercalculateurs de plus en plus performants.

De nos jours, cette puissance est produite à travers une recherche de parallélisme toujours plus massif. Cependant, un logiciel de simulation est généralement un logiciel de grande taille, sur lequel de nombreuses contributions ont été établies et différents travaux entretenus. Bâtir un tel logiciel peut nécessiter de gros moyens et de longues années. Il est dès lors très difficile pour les programmeurs de repenser leurs applications pour accompagner le développement des architectures des supercalculateurs.

Introduction générale

Il est cependant clair que la notion de parallélisme – c'est-à-dire l'exécution simultanée de plusieurs instructions – est centrale lorsque l'on parle de calcul intensif. Les supercalculateurs modernes sont tous bâtis sous la forme de grappe de nœuds de calcul (grouper plusieurs ordinateurs en réseau ou cluster), et un gros effort a été fourni par les développeurs pour adapter leurs applications de simulation à ce type d'architectures parallèles. Mais la tendance actuelle est de créer de plus en plus de parallélisme à tous les niveaux de ces grappes de calcul. Ainsi, autrefois épargné par cette tendance, le processeur principal de la machine lui même devient parallèle. Actuellement il est courant d'avoir des processeurs de huit à dix cœurs possédant des instructions vectorielles pour chaque cœur. Un parallélisme de plus en plus massif est donc nécessaire pour exploiter les processeurs modernes.… A cet effet, le calcul MPI devient une solution à ces types de problèmes.

Nous avons jugé utile d'organiser notre document en 02 grandes sections:

La premiére section concerne le cours des méthodes numériques appliquées enseigné au Département de Génie Mécanique.

Le premier chapitre traite des phénomènes de transport ;

Le chapitre 02 enseigne l'introduction de la Méthode des Différences finies ;

Le chapitre 03 expose la résolutions des problèmes paraboliques par l'étude de l'équation de la chaleur ;

La seconde section se focalise principalement autour du calcul MPI de l'équation de la chaleur qu'on a déjà étudié dans la partie A par une programmation séquentielle (linéaire).

Le 4 ^{er} chapitre traite des généralités du calcul parallèle par le MPI.

Le chapitre 05 résume les notions utilisées dans ce type de calcul, se veut comme un cours.

Le chapitre 06 rassemble des applications concernant le calcul MPI et s'organise comme des travaux dirigés (TD).

Le chapitre 07 permet d'identifier la terminologie utilisée sous forme de fonctions.

Chapitre 01 :
Méthodes d’étude et description mathématique des phénomènes de transport

1.1- Le phénomène de transport

les phénomènes de transport sont défins comme des processus irréversibles de nature statistique issus du mouvement aléatoire continu de molécules, ce phénomène est principalement observé dans le domines des fluides.

Chaque aspect de ces types de phénomènes de transport repose essentielement sur les conceptes des lois de conservation.

*- Ces lois de conservation, qui dans le contexte des phénomènes de transport sont formulées sous forme d'équations de continuité, décrivant comment la quantité étudiée (variable) doit être conservée.

*- Les équations de quantité de mouvement décrivent comment la quantité en question répond à divers stimuli à travers le transport. Des exemples connus comme la loi de Fourier de conduction thermique et les équations de Navier – Stokes, qui décrivent, respectivement, la réponse du flux thermique aux gradients de température et la relation entre le flux de fluide et les forces appliquées au fluide.

*- Ces équations démontrent également la relation entre les phénomènes de transport et la thermodynamique, ce lien qui explique pourquoi ces phénomènes de transport sont irréversibles. Presque tous ces phénomènes physiques impliquent en fin de compte des systèmes recherchant leur état d'énergie.

À l'approche de certain état, ils tendent à atteindre un véritable équilibre thermodynamique, au point qu'il n'y aurait plus de forces motrices dans le système en question et le transport cesse. Les différents aspects d'un tel équilibre sont directement liés à un transport spécifique: le transfert de chaleur est la tentative du système à atteindre l'équilibre thermique avec son environnement, tout comme le transport de masse et d'enérgie déplacent le système vers l'équilibre chimique et mécanique.

Des exemples de processus de transport comprennent la conduction thermique (transfert d'énergie), l'écoulement de fluide, la diffusion moléculaire (transfert de masse), le rayonnement.

Le transport de masse, d'énergie et de quantité de mouvement peut être affecté par la présence de sources externes:

- La vitesse de refroidissement d'un solide qui dissipe la chaleur dépend de l'application ou non d'une source de chaleur.

- La force gravitationnelle agissant sur une goutte empeche la résistance ou la traînée conférée par l'air environnant.

En ingénierie, physique et chimique, l'étude des phénomènes de transport concerne l'échange de masse, d'énergie, de charge, de moment et de moment angulaire entre les systèmes observés et étudiés.

Bien qu'il s'inspire de domaines aussi divers que la mécanique des milieux continus et la thermodynamique, il met fortement l'accent sur les points communs entre les sujets traités. La masse, la quantité de mouvement et le transport de chaleur partagent tous un cadre mathématique très similaire, et les parallèles entre eux sont exploités dans l'étude des phénomènes de transport pour établir des connexions mathématiques profondes qui fournissent souvent des outils très utiles dans l'analyse d'un domaine qui sont directement dérivés des autres.

Les phénomènes de transport sont présents dans toutes les disciplines de l'ingénierie. Certains des exemples les plus courants d'analyse des transports en ingénierie sont vus dans les domaines du génie des procédés, chimique, biologique et mécanique, mais le sujet (phénomène de transport) est une composante fondamentale du programme d'études dans toutes les disciplines impliquées de quelque manière que ce soit avec la mécanique des fluides, transfert de chaleur et transfert de masse. Il est maintenant considéré comme faisant partie de la discipline de l'ingénierie autant que la thermodynamique, la mécanique et l'électromagnétisme.

Il existe des similitudes notables dans les équations de quantité de mouvement, d'énergie et de transfert de masse qui peuvent toutes être transportées par diffusion, comme l'illustrent les exemples suivants:

- Masse: la diffusion et la dissipation des odeurs dans l'air est un exemple de diffusion de masse.
- Énergie: la conduction de la chaleur dans un matériau solide est un exemple de diffusion de chaleur.
- quantité de mouvement : la traînée subie par une goutte lorsqu'elle tombe dans l'atmosphère est un exemple de diffusion d'impulsion (la goutte de pluie perd son élan par rapport à l'air environnant en raison de contraintes visqueuses et décélère).

Les équations de transfert moléculaire de la loi de Newton pour la mécanique des fluides, de la loi de Fourier pour la chaleur et de la loi de Fick pour la masse sont très similaires. On peut passer d'un coefficient de transport à un autre afin de comparer les trois phénomènes de transport différents.

Ces phénomène de transport qu'on a évoqué sont tous décrit par un modèle mathématique qu'on essaiera de le détailler par la suite reconu par l'équation de diffusion, dans notre cas c'est l'équation de la chaleur . Cette équation va faire l'objet d'étude.

1.2- Le modèle mathématique

Un modèle mathématique est la mise en équation d'un phénomène dans le but de représenter fidèlement le comportement réel du phénomène. Des relations reliant les variables d'entrées aux variables de sorties sont établies. Le modèle est construit selon le but à atteindre par exemple pour analyser le mouvement de la terre autour du soleil, la terre et le soleil sont assimilés à des points matériels avec la loi de comportement correspondante tandis que si on veut étudier le mouvement de la terre par rapport à son axe, le modèle mathématique représente la terre par une sphère avec la loi de comportement donnant le mouvement de rotation de la terre par rapport à son axe.

Le succès du modèle dépend de sa facilité d'utilisation et de la précision des résultats prédis par le modèle.

Le modèle mathématique n'est pas spécifique aux sciences de l'ingénieur seulement, mais se retrouve dans d'autres domaines comme les sciences naturelles, les sciences sociales, les sciences économiques, etc . . .

1.3 - La formulation mathématique

Le module mathématique est formulé par des équations aux dérivés partielles et des conditions aux limites qui garantissent l'unicité de la solution, donc le fonctionnement du système physique. Nous nous intéressons particulièrement aux différentes types d'équations du seconde ordre, à deux variable indépendantes x et y, de la physique mathématique écrites sous la forme générale :

$$A\frac{\partial^2\phi}{\partial x^2} + B\frac{\partial^2\phi}{\partial x\partial y} + C\frac{\partial^2\phi}{\partial y^2} + D\frac{\partial\phi}{\partial x} + E\frac{\partial\phi}{\partial y} + F\phi = G(x,y) \qquad (1.1a)$$

Où $\phi = \phi(X, y)$ est la fonction recherchée, dépendante de x et de y c'est la fonction qui donne le comportement du modèle.

A, B, … et F sont les coefficients de l'équation aux dérivées partielles.

Ils sont fonction de x et y et peuvent être des constantes.

L'équation (1.1a) peut être réécrite sous la forme

$$A \frac{\partial^2 \phi}{\partial x^2} + B \frac{\partial^2 \phi}{\partial x \partial y} + C \frac{\partial^2 \phi}{\partial y^2} f\left(x, y, \phi, \frac{\partial \phi}{\partial x}, \frac{\partial \phi}{\partial y}\right) \quad (1.1b)$$

Selon le signe du déterminant $B^2 - 4AC$ on adopte le classent suivant

Si $B^2 - 4AC < 0$ l'équation est dite elliptique,

Si $B^2 - 4AC > 0$ l'équation est dite hyperbolique,

Si $B^2 - 4AC = 0$ l'équation est dite parabolique,

Exemple :

1-Equation de la place 2D

Soit l'équation de Laplace donné par :

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0$$

On a :

A = 1, B = 0, C = 1, d'où

$B^2 - 4AC = -4 < 0$,

l'équation de Laplace est donc elliptique.

2- Equation de conduction

soit l'équation de conduction instationnaire de la chaleur :

$$\frac{\partial^2 T}{\partial x^2} = \frac{\partial T}{\partial t}$$

On a :

A = 0, B = 0, C = 1

d'où $B^2 - 4AC = 0$,

l'équation est donc parabolique.

3- Equation de vibration

Equation de vibration transversale ou équation des ondes :

$$\frac{\partial^2 y}{\partial t^2} = c^2 \frac{\partial^2 y}{\partial x^2}$$

L'équation peut s'écrire

$$c^2 \frac{\partial^2 y}{\partial x^2} - \frac{\partial^2 y}{\partial t^2} = 0$$

on a :

A = c^2 , B = 0, C = - 1, d'où

D'où $B^2 - 4AC = 4/c^2 > 0$,

et l'équation est hyperbolique.

1.4- Classification des problèmes aux limites (PVL)

Les problèmes aux limites sont régis par des équations aux dérivées partielles accompagnées de conditions aux limites spécifiques. Selon le type d'équation on obtient le problème aux limites correspondant

- Si l'équation est elliptique, le problème est elliptique et on a un problème d'équilibre ou de valeurs aux limites (PVL)

- Si l'équation est parabolique, le problème est parabolique et on a un problème de valeurs initiales (PVI)

- Si l'équation est hyperbolique, le problème est hyperbolique et on a un problème de valeurs propres (PVP)

1.4.1 -Les problèmes de valeurs aux limites

Les équations aux dérivées partielles peuvent être par exemple, les équations bidimensionnelles de Laplace $\nabla^2\phi = 0$ ou de Poisson $\nabla^2\phi = f(x,y)$. On prescrit des conditions aux limites de types Dirichlet[1], Neumann[2] ou Cauchy[3] pour ce type de problèmes (Fig. 1.1).

La condition de Dirichlet est une condition sur ϕ. Celle-ci est imposée sur une partie (C_1) de la frontière.

La condition de Cauchy consiste à imposer ϕ et $\frac{\partial \phi}{\partial n}$ sur la frontière (C_2). On aura une condition de flux, sur cette frontière. Si $\alpha = 0$, la condition de flux est dite de Neumann, si $\alpha \neq 0$ la condition est dite de Cauchy.

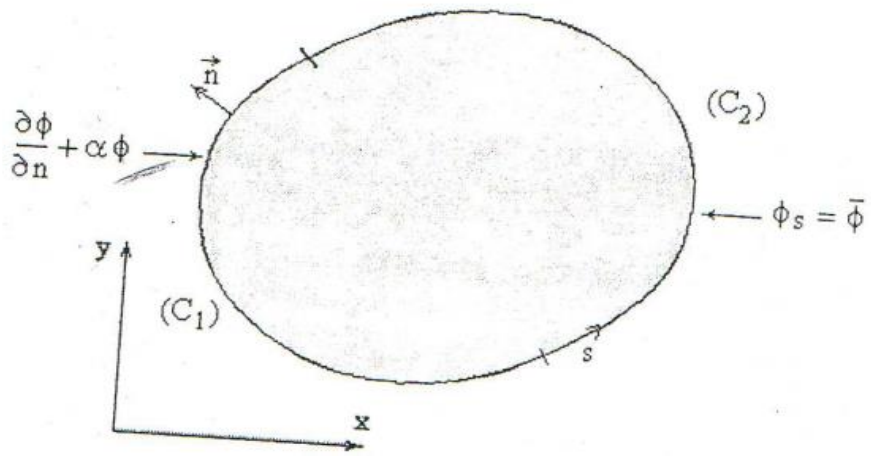


Fig. 1.1 : Problème de valeurs aux limites.

Quand ϕ est une fonction imposée connue sur toute la frontière du domaine (Ω) le problème est souvent désigné comme un problème de Dirichlet.

On trouve l'équation de Laplace dans de nombreuses applications comme la conduction thermique en régime stationnaire, le potentiel gravitationnel et électrostatique, ect …

Exemple

a -/ Conduction stationnaire de la chaleur dans une plaque rectangulaire (condition de Dirichlet)

Conduction stationnaire de la chaleur dans une plaque rectangulaire soumise des températures imposées sur les 4 cotés est un problème de valeurs aux limites, problème consiste à déterminer, par exemple, la température en tout point du domaine.

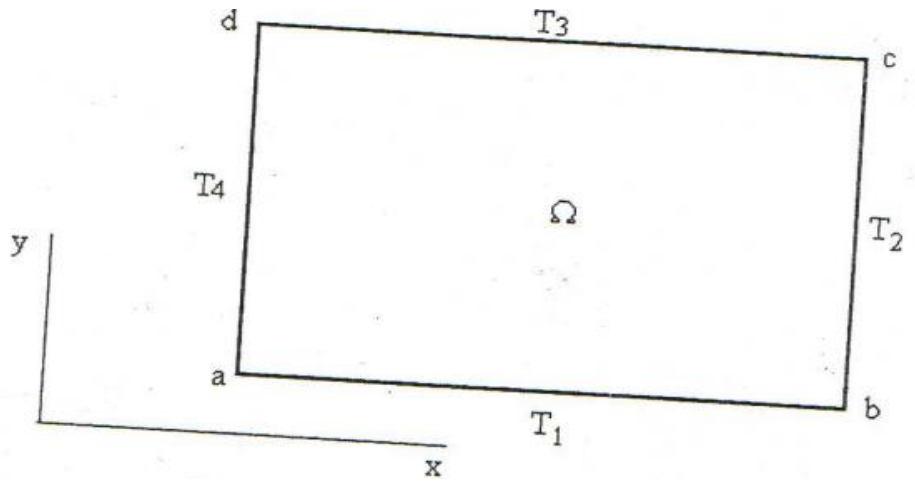


Fig. 1.2 problème de Dirichlet

La température est régie par l'équation de Laplace

$$\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}=0 \text{ sur } (\Omega)$$

Et les conditions aux limites

$$T(x,0)=T_1, T(L,y)=T_2, T(x,l)=T_3, \text{ et } T(0,y)=T_4$$

b/- Conduction de la chaleur dans une plaque rectangulaire

avec conditions de Dirichlet et Cauchy (Fig. 1.3)

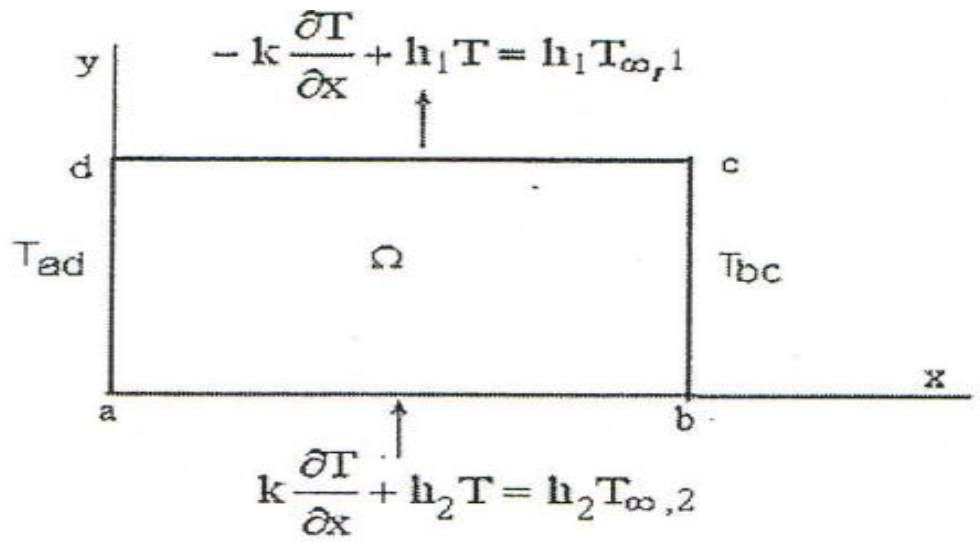


Fig. 1.3 : problème de valeurs aux limites avec condition de Dirichlet et Cauchy, le problème est régi par l'équation de Laplace et les conditions aux limites de types Dirichlet

$$T(L,y)=T_{bc}\ 0\le y\le l$$

$$T(0,y)=T_{ad}\ 0\le y\le l$$

Et les conditions de Cauchy données par le transfert thermique par convection aux frontières ab et cd respectivement :

Flux thermique entrant : (ab)

$$k\frac{\partial T}{\partial x}+h_1T=h_1T_{\infty,1}\ y=0\ ,\ 0\le x\le L$$

Flux thermique sortant : (dc)

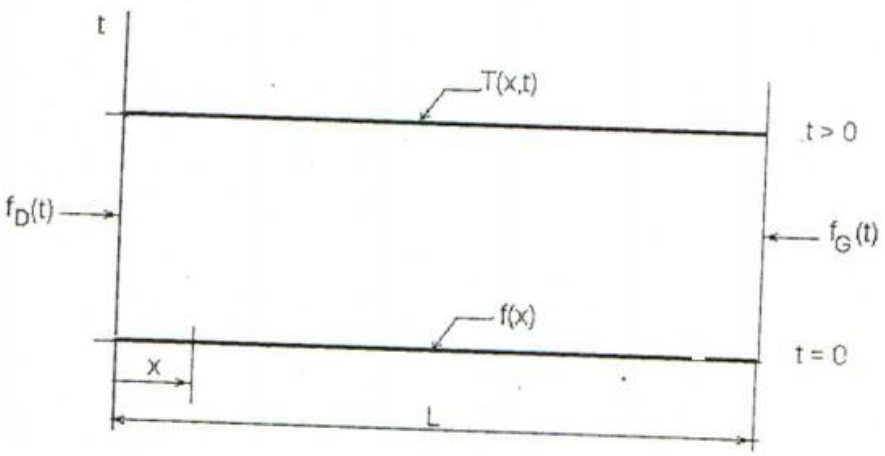
$$-k\frac{\partial T}{\partial x}+h_2T=h_2T_{\infty,2}\ y=l\ ,\ 0\le x\le L$$

1.4.2 le problème de valeurs initiales

Les problèmes régis par des équations paraboliques sont des problèmes de valeurs initiales ou instationnaires. On peut prescrire des conditions aux limites de types Dirichlet ou de Cauchy et des conditions initiales.

Exemple : propagation de la chaleur dans un mur

Le problème de transfert thermique transitoire dans un mur est l'exemple type de ce problème (Fig. 1.4). soit par exemple un mur d'épaisseur L se trouvant à la température initiale donnée par la relation $F(x)$. Aux deux extrémités et à un instant $t>0$ on impose des conditions aux limites de Cauchy.



L'équation gouvernante est l'équation parabolique :

$$\alpha \frac{\partial^2 T}{\partial x^2} = \frac{\partial T}{\partial t} \quad 0 \le y \le L, t > 0$$

Les conditions aux frontières sont données par :

$$T(0,t) = T_G \quad t > 0$$

$$T(L,t) = T_D \quad t > 0$$

Ou par des conditions de flux :

$$a_1 T(0,t) + b_1 \frac{\partial T(0,t)}{\partial x} = f_G(t) t > 0$$

$$a_2 T(L,t) + b_2 \frac{\partial T(L,t)}{\partial x} = f_D(t) t > 0$$

La condition initiale s'écrit

$$T(x,0) = f(x)$$

1.4.3- le problème de valeurs propres

Les problèmes régis par des équations hyperboliques sont des problèmes de valeurs propres ou de vibration. On prescrit aux frontières des conditions aux limites de Neumann ou de Dirichlet et des conditions initiales.

Exemple : vibration d'une corde élastique

Une corde parfaitement élastique de longueur L, de masse linéique ρ, est soumise aux deux extrémité à une force T constante (Fig. 1.5). la déformation élastique transversale $y = y(x,t)$ de la corde obéit à l'équation hyperbolique :

$\frac{\partial^2 y}{\partial t^2} = c^2 \frac{\partial^2 y}{\partial x^2}$ est la vitesse de la propagation de l'onde $c = \sqrt{\frac{T}{\rho}}$

Chapitre 02 :

Introduction à la Méthode des Différences finies

2.1 Introduction

La méthode des différences finies est une méthode numérique de résolution des équations différentielles ordinaires ou des équations aux dérivées partielles. Sa formulation est basée sur l'approximation locale au voisinage d'un point donné des fonctions dérivées apparaissant dans les équations différentielles. Les fonctions dérivées sont approchées par des fonctions polynomiales données par le développent en série de Taylor [1], [2].

Dans ce chapitre on utilise la méthode des différences finies pour résoudre des problèmes aux limites régis par les équations aux dérivées partielles linéaires du second type exposée dans le chapitre 01 (1.1).

2.2 Le développement de Taylor

a) Développement en série de Taylor

On montre que si une fonction $f(x)$ est analytique, indéfiniment dérivable au voisinage d'un point $x = x_0$ (c'est-à-dire dans un intervalle ouvert contenant le point x_0 , $0 < |x - x_0| < R$,

alors cette fonction peut être approchée par une fonction polynomiale écrite sous la forme de série convergence qu'on appelle série de Taylor

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x-x_0)^2}{2!}f''(x_0) + \cdots \frac{(x-x_0)^n}{n!}f^{(n)}(x_0) + \cdots (2.1)$$

Le second membre de l'équation (2.1) est le développement en série de la fonction f au voisinage du point $x = x_0$.

L'équation (2.1) s'écrit encore

$$f(x) = \sum_{n=1}^{\infty} \frac{(x - x_0)^n}{n!} f^{(n)}(x_0)$$

b) Développement limité en série de Taylor

En fait, dans l'équation (2.1) on ne peut tenir compte que d'un nombre fini de termes : on effectue une troncature de la série. On a donc un développement à termes finis c'est le développement limité de Taylor (appelée aussi formule de Taylor) de la fonction f auteur du point $x = x_0$

$$f(x) = f(x_0) + (x - x_0)f'(x_0) + \frac{(x-x_0)^2}{2!}f''(x_0) + \cdots \frac{(x-x_0)^n}{n!}f^{(n)}(x_0) + R_n (2.2)$$

Le dernier terme de l'équation (2.2) est appelé reste ou erreur de troncation, est donné par la formule de Lagrange

$$R_n = \frac{(x-x_0)^{n-1}}{(n+1)!}f^{(n+1)}(\xi) x - x_0 \leq \xi \leq x + x_0 (2.3)$$

Cette erreur est de l'ordre de grandeur de $(x - x_0)^{n+1}$et est notée par O$(x - x_0)^{n+1}$. Puisque la fonction est infiniment dérivable $f^{(n+1)}(\xi)$existe et est bornée. Puisque $\lim_{n\to\infty} \frac{(x-x_0)^{n-1}}{(n+1)!} = 0$, la série converge. Ecrite sous la forme (2.3) la formule de Taylor est utilisée pour approcher les fonctions par des fonctions polynomiales.

En posant$h = \Delta x = x - x_0$la formule de Taylor (2.3) devient :

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2!}f''(x_0) + \cdots + \frac{h^n}{n!}f^{(n)}(x_0) + O(h^{n+1}) (2.4)$$

On peut interpréter graphiquement l'erreur de troncation. Par exemple, si on tient compte uniquement du terme contenant la dérivée première, l'équation (2.4) s'écrit :

$$f(x_0 + h) = f(x_0) + hf'(x_0) \pm erreur (2.5)$$

Le signe $\pm$ est introduit pour tenir compte de l'allure (concave ou convexe) de la courbe donnée par$f(x)$. Soit à interpréter l'erreur dans le cas, d'une fonction $f(x)$ convexe.

Soit C le point de rencontre de la tangente T à la courbe f au point x_0 et la verticale au point

$x_0 + h$. On a :

$$M'C = M'A + AC = AC - AM'$$

Où :

$$AM' = f(x_0 + h)$$

Et

$$AC = AB + BC$$

Sachant que

$$BC = MBtan\alpha = htan\alpha$$

Et

AB = $f(x_0)$

$$tan\alpha = f'(x_0)$$

2.3 Enoncé de la méthode

La méthode des différences finies est une méthode d'approximation des équations différentielles ou des équations aux dérivées partielles. La méthode consiste à :

- Discrétiser la fonction $f(x_0)$ aux différents points M_i de coordonnées x_i espacés d'un pas $h = \Delta x$ selon la direction x (Fig. 2.2)
- Reformuler les dérivés partielles qui apparaissent dans l'équation aux dérivées partielles. Les fonctions dérivées sont approchées en utilisant le développement limite de Taylor de la fonction f au voisinage des pointsx_i.

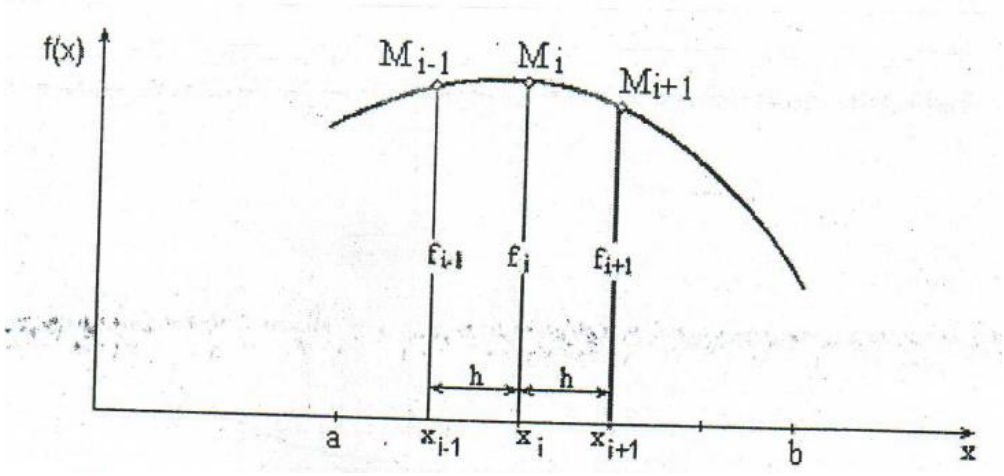


Fig 2.2 Discrétisation du domaine d'analyse de la fonction f

Pour la discrétisation, on adopte les notations suivantes :

$x \leftrightarrow x_i$

$x_i = x_1 + (i-1)h$

$f_i = f(x_i)$

$f_{i+1} = f(x_i + h)$ …etc.

2.3.1 Expression des dérivées premières

a) Différences finies en avant

La fonction f est connue aux points x_i (points pivots) du domaine d'analyse. A l'aide de la formule de Taylor (2.4) on développe la fonction f jusqu'à l'ordre 2

$$f(x_i+h) = f(x_i) + hf'(x_i) + \frac{h^2}{2!} f^{(2)}(\xi) \quad (2.6)$$

ξ Abscisse d'un point se trouvant dans le voisinage de x_i avec $x_i \leq \xi \leq h + x_i$

En résolvant (2.6) pour $f'(x_i)$, on a :

$f'(x_i) = \frac{f(x_i+h)-f(x_i)}{h} + O(h)(2.7)$

Avec O(h) l'erreur de troncation

$O(h) = \frac{h}{2!} f^{(2)}(\xi) x_i \leq \xi \leq h + x_i(2.8)$

L'erreur est de l'ordre de grandeur du pas h (de l'ordre du degré le plus petit du pas h).

La formule de la dérivée première s'écrit en notation indicielle,

$f' = \frac{f_{i+1}-f_i}{h} + O(h)(2.9)$

L'équation (2.7) représente l'expression de la dérivée première de la fonction f écrite par un schéma de différences finies en avant ou différences progressive. L'appellation différence en avant se justifie par le fait que la différence $f_{i+l} - f_i$ est calculée aux points x_{i+l} et x_i, le point x_{i+l} se trouvant en avant du point pivot x_i

La formulation analytique de la dérivée est donnée par $f'(x_i) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$

L'utilisation de la méthode des différences finies s'explique par le fait que le pas h ne peut être égale à 0, les dérivées sont calculées à l'aide des relations (2.7) ou (2.8) en utilisant une quantité finie de h. l'erreur commise est alors O(h)

b) Différences finies en arrière

En changent h en –h dans l'équation (2.6) on obtient :

$$f(x_i - h) = f(x_i) - hf'(x_i) + \frac{h^2}{2!} f^{(2)}(\xi) x_i - h < \xi < x_i(2.10)$$

En résolvant pour la dérivée première :

$$f'(x_i) = \frac{f(x_i)-f(x_i-h)}{h} + O(h)(2.11)$$

Avec l'erreur de troncation

$O(h) = \frac{h^2}{2!} f^{(2)}(\xi) x_i - h \leq \xi \leq x_i(2.12)$

L'erreur est de même ordre de grandeur que celle obtenue pour le schéma de différences en avant

L'équation (2.11) s'écrit en notation indicielle

$f' = \frac{f_i - f_{i-1}}{h} + O(h)$(2.13)

L'équation (2.11) représente l'expression de la dérivée première de la fonction f écrite par un schéma de différences finies en arrière ou différence finies régressives.

L'appellation de différences en arrière se justifie par le fait que la différence $f_i - f_{i-1}$ est calculée aux points x_i et $x_i - l$, le point $x_i - l$ se trouvant en arrière du point pivot x_i.

c) Différences centrées

L'élimination de $f(x_i)$ des équations

$x_i + h_f) = f(x_i) + hf^i(x_i) + \frac{h^2}{2!}f^{(2)}(x_i) + \frac{h^3}{3!}f^{(3)}(x_i) + \frac{h^4}{4!}f^{(4)}(\xi)$(2.14)

$x_i - h_f) = f(x_i) - hf^i(x_i) + \frac{h^2}{2!}f^{(2)}(x_i) - \frac{h^3}{3!}f^{(3)}(x_i) + \frac{h^4}{4!}f^{(4)}(\xi)$ (2.15)

Nous permet de trouver la dérivée première par un schéma de différences centrées

$f'(x_i) = \frac{f(x_i+h) - f(x_i-h)}{2h} + O(h^2)$(2.16)

$O(h^2) = \frac{h^2}{6}f_i^{(3)}(\xi) x_i - h \leq \xi \leq x_i$+1 (2.17)

Qui s'écrit en notation indicielle

$f_i^{(1)} = \frac{f_{i+1} - f_{i-1}}{2h} + O(h^2)$(2.18)

L'erreur est de l'ordre deh^2. La dérivée première centrée est plus précise que dans le cas de différences en avant ou en arrière.

l'appellation différences centrées se justifie par le fait que la différence $f_{i+1} - f_{i-1}$ est calculée aux points x_{i+1} et x_{i-1}, le point pivot se trouvant au centre de x_{i+1} et x_{i-1}

Exemple 2.1

En considérant un pas □x = 0.1, Calculer par un schéma de différences finies (a) en avant, (b) en arrière (c) et centré la dérivée première de la fonction $f(x) = x^2$ au point $x = 2$. Calculer l'erreur de troncation pour chaque cas.

Réponse

$x_i = 2$, □x = 0.1 et $f(x_i) = x_i^2$

$$x_{i+1} = x_i + \Delta x = 2.1$$

$$x_{i-1} = x_i - \Delta x = 1.9$$

$$f_i = f(2) = 4$$

$$f_{i+1} = f(2.1) = 4.41$$

$$f_{i-1} = f(1.9) = 3.61$$

$$f''(x) = 2$$

(a)Différence premières en avant

$$f_i' = \frac{f_{i+1} - f_i}{\Delta x} = 4{,}1$$

(b) Différences en arrière

$$f_i' = \frac{f_i - f_{i-1}}{\Delta x} = 3.9$$

$$O(h) = \frac{\Delta x}{2} f''(\xi) = 0.1$$

(a) Différences centrées

$$f_i' = \frac{f_{i+1} - f_{i-1}}{2\Delta x} = 4$$

$$O(h^2) = \frac{\Delta x^2}{6} f^3(\xi) = 0$$

2.3.2 Expression des dérivées secondes

a) Différences finies en avant

On écrit le développement de $f(x_1 + h)$ et $f(x_1 + 2h)$:

$$f(x_1+h)= f(x_i)+ hf'(x_i)+ \frac{h^2}{2!}f''(x_i)+ \frac{h^3}{3!}f^{(3)}(\xi) \quad \textbf{(2.19)}$$

$$f(x_1+2h)= f(x_i)+ 2hf'(x_i)+2h^2f''(x_i)+ \frac{8}{3!}h^3f^{(3)}(\xi) \quad \textbf{(2.20)}$$

Eliminant $f(x_i)$ entre les deux équations, on obtient :

$f''(x_i) = \frac{f(x_i)-2f(x_i+h)+f(x_i+2h)}{h^2} + O(h)$(2.21)

$O(h) = -hf_i^{(3)}(\xi)$(2.22)

 point du voisinage de x_i, on prend le voisinage qui contient les autres $x_i \leq \xi \leq x_i$+2h en

En notation indicielle, on a

$f_i^{(2)} = \frac{f_i-2f_{i+1}+f_{i+2}}{h^2} + O(h^2)$(2.23)(2.23)

L'équation (2.23) représente l'expression de la dérivé seconde écrite par un schéma de différences finies en avant ou progressive.

b) Différences finies en arrière

On considère le développement de $f(x_i + h)$ et $f(x_i - 2h)$:

$$\underset{f}{x_i + h}) = f(x_i) + hf^i(x_i) + \frac{h^2}{2!}f''(x_i) + \frac{h^3}{3!}f^{(3)}(\xi) \quad (2.24)$$

$$\underset{f}{x_i - 2h}) = f(x_i) \text{ -2 } hf^i(x_i) + 2h^2f''(x_i) - \frac{4}{3}h^3f^{(3)}(\xi) \quad (2.25)$$

Eliminant $f(x_i)$ entre les deux équations, on obtient le schéma de différences finies en arrière de la dérivée seconde :

$f''(x_i) = \frac{f(x_i-2h)-2f(x_i+h)+f(x_i)}{h^2} + O(h)$(2.26)

$O(h) = hf_i^{(3)}(\xi)x_i - 2h \leq \xi \leq x_i$(2.27)

$f_i^{(2)} = \frac{f_{i-2}-2f_{i-1}+f_i}{h^2} + O(h^2)$(2.28)

Qui est un schéma de différences finies en arrière de la dérivée seconde

c) Différences finies centrées

On considère le développement de $f(x_i + h)$ et $f(x_i - h)$:

$f(x_i+ h)=f(x_i)+ hf'(x_i)+ \frac{h^2}{2!}f''(x_i)+ \frac{h^3}{3!}f^{(3)}(x_i)+ \frac{h^4}{4!}f^{(4)}(\xi)$ **(2.29)**

$f(x_i - h)=f(x_i)- hf'(x_i)+ \frac{h^2}{2!}f''(x_i)+ \frac{h^3}{3!}f^{(3)}(x_i)+ \frac{h^4}{4!}f^{(4)}(\xi)$ **(2.30)**

Eliminant $f(x)$ entre les deux équations, on a :

$f''(x_i) = \frac{f(x_i-h)-2f(x_i)+f(x_i+h)}{h^2} + O(h^2)$(2.31)

$O(h^2) = \frac{1}{12}h^2f^{(4)}(\xi)x_i - h \leq \xi \leq x_i$+h (2.32)

$f_i^{(2)} = \frac{f_{i-1}-2f_i+f_{i+1}}{h^2} + O(h^2)$(2.33)

Qui est un schéma de différences finies centrées de la dérivée seconde.

2.4 Procédure de résolution des problèmes aux limites

La résolution d'un problème aux limites par la méthode des différences finies se fait selon les principales étapes suivantes :

a) construire le maillage ou grille du domaine Ω

b) transformer l'équation aux dérivées partielles et l'exprimer sous forme de molécule ou *schéma numérique* de différences finies.

Dans notre cas , c'est l'équation de la chaleur

c) Ecrire l'équation de différences finies aux points du maillage. Cette partie sera traitée dans le chapitre 03.

d) Obtenir le système d'équations algébriques discrètes $[K]\{\phi\} = \{\phi_c\}$

$\{\phi_c\}$Est le vecteur connu donné par les conditions aux limites non homogène, $[K]$ est la matrice des coefficients et $\{\phi\}$ est le vecteur solution recherché en tout point du maillage.

e) trouver la solution $\{\phi\}$ en résolvant le système d'équations $[K]\{\phi\} = \{\phi_c\}$

Chapitre3 :

Résolutions des problèmes paraboliques équation de la chaleur

Chapitre3 :Résolutions des problèmes paraboliques équation de la chaleur

3.1 Formulation

Pour un problème 1 D, la formulation est donnée par l'équation parabolique,

$$\frac{\partial \phi}{\partial t} = \alpha \frac{\partial^2 \phi}{\partial x^2} \qquad \square \in \Omega_{x,t} \qquad (3.1)$$

Les conditions aux deux extrémités de gauche et de droite

$$\phi(0,t) = \phi_G t > 0, \qquad (3.2)$$

$$\phi(L,t) = \phi_D t > 0, \qquad (3.3)$$

Et la condition initiale :

$$\phi(x,0) = f(x) \qquad (3.4)$$

Avec $\Omega_{x,t} = \Omega_x ¤ \Omega_1$, domaine spatio-temporel où $\Omega_x = \{X / 0 < X < L\}$ et $\Omega_x = \{t / t > 0\}$,

α est le coefficient de diffusion (ou de propagation de chaleur dans le matériau du milieu Ω_x). ϕ_D , ϕ_G , $f(x)$ sont des valeurs spécifiées (connues).

Le problème consiste à déterminer :

$\phi(x,t)$ en point x et à un instant donnée t dans le domaine $\Omega_{x,t}$.

3.2 Le maillage

Le maillage du domaine $\Omega_{x,t}$ est construit à l'aide des relations connues m = L / □x + 1 et n = D / □y+1, D est la durée du phénomène physique transitoire.

L'équation aux dérivées partielles est discrétisée aux points pivots du maillage, les conditions aux limites (3.3), (3.4)sont spécifiées aux frontières gauche et droite du domaine géométrique Ω_x comme le montre la figure 3.1.

Chapitre3 :Résolutions des problèmes paraboliques équation de la chaleur

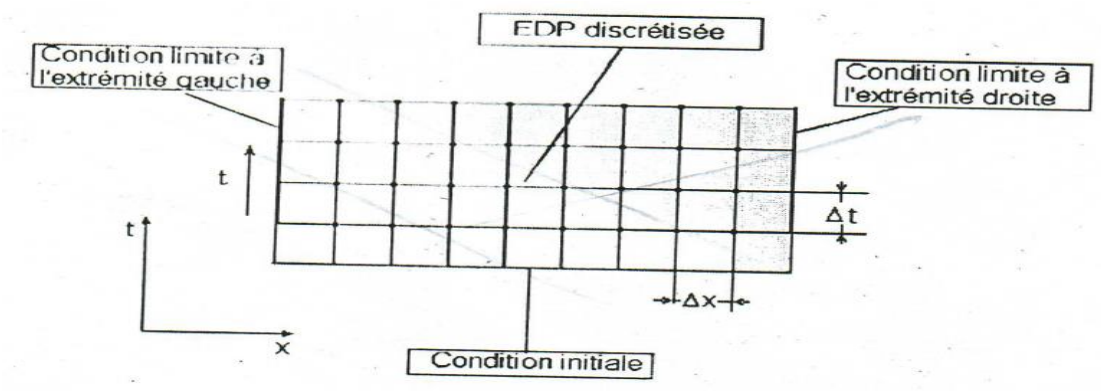


Fig. 3.1 : le problème parabolique et sa discrétisation

Différentes méthode sont utilisées pour la résolution du problème parabolique. On peut citer principalement la méthode explicite, la méthode implicite et la méthode de Crank-Nicolas.

3.3 La méthode explicite (schéma FTCS)

On exprime la dérivée première temporelle à l'aide des différences finies en avant (la différence centrée pourtant plus précise n'est pas considérée ici car elle conduit à un schéma instable).

$$\frac{\partial \phi}{\partial t} = \frac{\phi_{i;j+1} - \phi_{i;j}}{\Delta t} + O(\Delta t)$$

où $0(\Delta t) = -\frac{\Delta t}{2}\phi^{2}(\xi) \qquad t_j < \xi < t_j + \Delta t$

La dérivée seconde spatiale est une différence centrale

Chapitre3 :Résolutions des problèmes paraboliques équation de la chaleur

$$\frac{\partial^2 \phi}{\partial x^2} = \frac{\phi_{i-1,j} - 2\phi_{i,j} + \phi_{i+1,j}}{(\Delta x)^2} + 0(\Delta x^2)$$

L'équation (3.4) s'écrit alors :

$$\frac{\phi_{i,j+1} - \phi_{i,j}}{\Delta t} = \alpha \frac{\phi_{i-1,j} - 2\phi_{i,j} + \phi_{i+1,j}}{(\Delta x)^2} + O\left[\Delta t, (\Delta x)^2\right] \quad (3.5)$$

En tronquant l'erreur et en réarrangeant, l'équation (2.73) s'écrit:

$$\phi_{i,j+1} = r\phi_{i-1,j} + (1-2r)\phi_{i,j} + r\phi_{i+1,j} \quad (3.6)$$

où

$$r = \frac{\alpha \Delta t}{(\Delta x)^2} \quad (3.7)$$

On a utilisé un schéma temporal de différences en avant et un schéma spatial centre pour formuler l'équation parabolique. Ce type de schéma est appelé FTCS (de l'anglais 'Forward Time Centered Space'). C'est un schéma explicite de résolution de l'équation (3.1). Comme le montre l'équation (3.6), l'appellation explicite de la méthode se justifie par le fait que la solution recherchée $Ø_{i,j+1}$ à l'étape de temps actuel $(j+1)\Delta t$ est explicitement déterminée à partir des solutions $Ø_{i-1,j}$, $Ø_{i,j}$ et $Ø_{i+1,j}$ connues à l'étape de temps précédent $j\Delta t$.

Chapitre3 :Résolutions des problèmes paraboliques équation de la chaleur

3.4 Le schéma numérique

Le schéma numérique de l'équation explicite peut se mettre sous la forme moléculaire

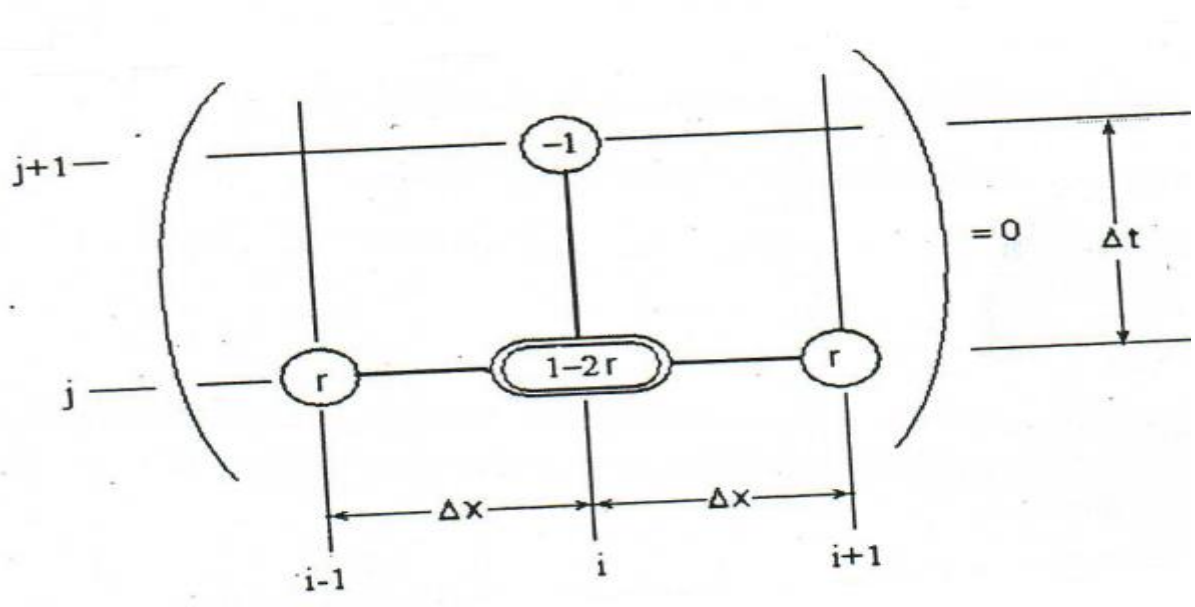


Fig. 2.32 Représentation moléculaire de l'équation explicite

Chapitre 04:
Généralités sur le calcul MPI

Chapitre 4 :Généralités sur le calcul MPI

Avant d'aborder le calcul MPI proprement dit, il est primordial d'introduire de nouvelles définitions afin de se familiariser avec de nouveaux paradigmes voire nouveaux concepts de calcul en se basant sur les notions déjà connues ou étudiées en programmation séquentielle (linéaire) vers ces nouvelles notions (le calcul parallèle).

4.1/- Problèmes numériques et calculs scientifiques- Discrétisation

De façon générale, l'analyse numérique est un domaine des mathématiques très répandu dans les autres domaines technologiques comme l'énergétique et qui produit des programmes souvent compliqués à mettre en œuvre et nécessitant de bonnes connaissances de programmation.
Grâce à une bonne connaissance de l'informatique et de la programmation, il est possible d'obtenir des programmes rapides et peu coûteux en mémoire, par le concept de calcul parallèle.
C'est pour cette raison que, bien souvent, l'analyse numérique mène à des collaborations entre mathématiciens et informaticiens. L'analyse numérique consiste à développer des méthodes numériques permettant d'obtenir des solutions exactes ou approchées à des problèmes mathématiques. Les solutions approchées résultent, dans la plupart des cas, de méthodes de discrétisation, comme dans le cas des équations différentielles.
Comme dans la plupart des méthodes de résolution numérique des EDP (Équations aux dérivées partielles), un ordinateur n'étant pas capable de résoudre une équation sur des variables continues, et les simulations d'EDP sont très souvent transformées en problèmes discrétisés. (Passage du continu au discret- Discrétisation).

4.2/- Aspect architecturaux hardwares des calculateurs

Le modèle utilisant le calcul parallèle permet de résoudre d'importants problèmes de calcul nécessitant des temps d'exécution très longs en environnement « classique ».

Il est utile de distinguer les concepts suivants :

- réseau : association physique ou logique d'unités informatiques qui collaborent et partagent des données et qui dépendent d'une unité centrale de contrôle;
- cluster : regroupement d'unités informatiques qui coopèrent et forment une seule unité informatique virtuelle sur les plans fonctionnel;
- grille :

Une grille de calcul permet de faire du calcul distribué : elle exploite la puissance de calcul (processeurs, mémoires…) de milliers d'ordinateurs afin de donner l'illusion d'un ordinateur virtuel très puissant. Une grille est en effet une infrastructure, c'est-à-dire des équipements techniques d'ordres matériel et logiciel.

Une grille se compose de ressources informatiques dans un but de permettre une gestion globale et de dépasser les limitations d'un ordinateur pour :

- augmenter la disponibilité ;
- permettre une répartition de la charge ;

- faciliter la gestion des ressources (processeur, mémoire vive, disques durs, bande passante réseau).

4.3/- Relation entre langage de programmation et calcul MPI

Fortran (*mathematical FORmula TRANslating system*) est un langage de programmation généraliste dont le domaine de prédilection est le calcul scientifique et le calcul numérique. Il est utilisé aussi bien sur ordinateur personnel que sur les superordinateurs(bibliothèque LINPACK).

Avec les améliorations des bibliothèques scientifiques écrites en Fortran, pour exploiter les nouvelles possibilités des calculateurs (vectorisation, coprocesseurs, parallélisme) ont maintenu l'usage de ce langage qui ne cesse d'évoluer.

Parmi les fonctionnalités ajoutées ces dernières décennies, on citera le calcul sur les tableaux (qui peuvent comporter jusqu'à quinze dimensions), la programmation modulaire, la programmation générique (Fortran 90), le calcul haute performance (Fortran 95), la programmation orientée objet et l'interopérabilité avec les bibliothèques du langage C (Fortran 2003), la programmation concurrente et le calcul parallèle à l'aide des cotableaux (Fortran 2008), des équipes, des évènements et des sous-routines collectives (Fortran 2018), en plus des interfaces de la bibliothèque Message Passing Interface.

Remarque :

La prochaine norme désignée par Fortran 202x, est attendue pour 2023.

4.4/-Paradigmes et modèles de parallélisation

Avec l'apparition des architectures parallèles sont apparus les premiers problèmes de programmation parallèle. En effet, un programme séquentiel en lui même, et sans l'aide particulière du système d'exploitation ou de tout autre système externe de répartition de charge sur les cœurs ou les processeurs, n'exploite pas directement les ressources d'une machine parallèle. Or, la conception d'un programme parallèle peut s'avérer très complexe et très dépendante des architectures utilisées.

Avec l'apparition des machines parallèles sont donc apparus également des paradigmes de programmation parallèle, offrant un ensemble d'approches générales pour envisager un programme parallèle, puis des modèles de programmation parallèle, permettant de concevoir de façon plus précise des programmes sur ces machines.

Les paradigmes de programmation parallèle représentent donc un niveau d'abstraction plus bas et moins précis que les modèles de programmation parallèle [3]

Les modèles de programmation parallèle, même si certains sont naturellement induits par le matériel, peuvent être implémentés pour différentes architectures parallèles. On distingue donc les modèles de programmation des implémentations qui y sont associées pour un modèle d'exécution donné. Par exemple un modèle de programmation parallèle initialement pensé pour des architectures à mémoire distribuée pourrait être implémenté sur une architecture DSM (Distributed Shared Memory) qui permet de construire un espace mémoire partagé pour tous les processeurs, bien que cet espace mémoire soit physiquement distribué.

Chapitre 4 :Généralités sur le calcul MPI

En 1972, Michael J. Flynn définit une classification des architectures des ordinateurs [4]. Quatre classes étaient alors répertoriées et sont représentées comme suit:
- La première classe, nommée Single Instruction, Single Data (SISD) représente les machines séquentielles n'exploitant aucun parallélisme.
- La deuxième classe, Singe Instruction, Multiple Data (SIMD), représente les machines pouvant appliquer une unique instruction à un ensemble de données. Cette classe concerne donc typiquement les architectures vectorielles ou GPU.
- La troisième classe, Multiple Instruction, Single Data (MISD), représente les machines permettant d'appliquer plusieurs instructions à la suite sur une même donnée d'entrée. Cette classe concerne typiquement les programmes de type pipeline ou les systèmes de tolérance aux pannes cherchant à comparer deux résultats issus d'une même donnée.
- La quatrième classe, Multiple Instruction, Multiple Data (MIMD), représente les machines multiprocesseurs qui peuvent exécuter simultanément des instructions différentes sur des données différentes.

Cette classification est toujours utilisée dans le parallélisme actuel, mais sous une autre forme. En effet, les différentes classes ne sont plus représentatives d'architectures matérielles particulières, la plupart des machines répondant à l'ensemble de ces classes.
En revanche, les classes de Flynn représentent désormais des paradigmes de programmation Table 2.1 – Taxinomie de Flynn parallèle, très souvent associés aux paradigmes de parallélisation de tâches et de données.

4.5/- Parallélisme de tâches- Parallélisme de données.

Ce paradigme de programmation cherche à diviser un programme en un ensemble de tâches qui peuvent être dépendantes, mais aussi indépendantes. Dans ce cas c'est l'exécution du programme qui cherche à être parallélisée. Les paradigmes de parallélisation MISD et MIMD peuvent être associés au parallélisme de tâches. Dans le premier cas, des opérations successives sont appliquées sur un jeu de données d'entrée, on appelle communément ce type de calcul un pipeline.
L'instruction i + 1 ne peut alors être exécutée qu'une fois l'instruction i terminée. Toutefois, sur des données d'entrées suffisamment nombreuses, le parallélisme peut apparaître en quinconce. En effet étant donné une donnée d'entrée [x 1 , . . . , x n], une fois x 1 calculé pour l'instruction i, il est possible de calculer simultanément x 2 pour l'instruction i et x 1 pour l'instruction i + 1. Le paradigme MIMD, quant à lui, est rencontré plus fréquemment et offre plus de possibilités de parallélisassions.
Dans ce cas, on cherchera à identifier des tâches travaillant sur des données différentes, ce qui rend les tâches indépendantes les unes des autres. Toutefois, le développeur devra se charger de synchroniser les différentes tâches ensemble afin de garantir la cohérence des résultats. Nous pouvons enfin noter le paradigme MPMD (Multiple Program, Multiple Data), qui étend le concept MIMD à des programmes. Ainsi, chaque processeur peut appliquer un ou plusieurs programmes qui lui sont propres à des données éventuellement différentes des autres processeurs de façon indépendante. Les synchronisations nécessaires au bon fonctionnement du programme parallèle sont alors à la charge de l'utilisateur.

Dans ce paradigme, le parallélisme se focalise sur la façon dont les données sont distribuées sur les différents processeurs. L'ensemble des processeurs effectuent alors le même jeu d'instructions sur des données d'entrée qui leurs sont propres.
Dans ce type de parallélisme les tâches effectuées par le programme sont peu modifiées. Il faut toutefois réfléchir et concevoir les communications, les échanges ou les synchronisations nécessaires entre les processeurs pour que le programme parallèle soit correct et donne le même résultat qu'en séquentiel. Les paradigmes SIMD et SPMD (Simple Program, Multiple Data) sont associés au parallélisme de données. Ils représentent le même type de parallélisation, toutefois SIMD est associé aux architectures vectorielles et GPU, où la notion d'instruction est clairement définie et synchrone. L'approche SPMD est plus vaste et moins tournée vers la solution matérielle. Elle peut s'appliquer à des architectures à mémoire partagée comme distribuée. Ce paradigme considère une exécution indépendante d'un programme sur chaque processeur, et sur des données différentes, et met à la charge du programmeur les synchronisations nécessaires à la cohérence du calcul général. Ce type de parallélisation est l'un des plus utilisés, notamment pour les architectures à mémoire distribuée [5], [6].
À travers ce document, nous allons nous focaliser sur le parallélisme basé sur les mémoires partagées. Dans les architectures à mémoire partagée, les processus peuvent interagir par l'écriture et la lecture dans des espaces mémoire partagés et communs. Ces architectures permettent donc des interactions entre les processus par la simple utilisation de la mémoire de la machine, mais font intervenir des problèmes de concurrence d'accès aux données ainsi que de cohérence ou d'intégrité des données [7], [8].

4.6/-Historique des Modèles de programmation parallèle par MPI

Depuis la première version en juin 1994, les modèles de programmation parallèle par MPI n'ont pas cessé d'évoluer [09], [10], [11]:

- Version 1.0 :
En juin 1994, le forum MPI (Message Passing Interface Forum), avec la participation d'une quarantaine d'organisations, abouti à la définition d'un ensemble de sous-programmes concernant la bibliothèque d'échanges de messages MPI

- Version 1.1 :
En Juin 1995, avec implémentation seulement des changements mineurs par rapport à la précédente.

- Version 1.2 :
En 1997, avec des changements mineurs pour une meilleure cohérence des dénominations de certains sous-programmes.

- Version 1.3 :
En Septembre 2008, avec des clarifications dans MPI 1.2, en fonction des clarifications elles-mêmes apportées par MPI-2.1

- ***Version 2.0 :*** apparue en juillet 1997, cette version apportait des compléments essentiels volontairement non intégrés dans MPI 1.0 (gestion dynamique de processus, copies mémoire à mémoire, entrées-sorties parallèles, etc.)

- ***Version 2.1 :***
En Juin 2008, avec seulement des clarifications dans MPI 2.0 mais aucun changement.

- ***Version 2.2 :***
En Septembre 2009, avec seulement de « petites » additions.

- ***MPI 3.0 / Version 3.0 :***
En Septembre 2012, changements et ajouts importants par rapport à la version 2.2 ; principaux changements :

- Communications collectives non bloquantes ;
- Révision de l'implémentation des copies mémoire à mémoire ;
- Fortran (2003-2008) bindings ;
- Suppression de l'interface C++ ;
- Interfaçage d'outils externes (pour le débogage et les mesures de performance).

Chapitre 05 :
Notions et méthodes dans le calcul MPI

Chapitre 5 :**Notions et méthodes dans le calcul MPI**

Dans ce chapitre, nous allons nous intéressés au developpement du calcul MPI dans son aspect conception en introduisant les définitions générales de la programmation par échange de messages et la mise en pratique des différents programmes et algorithmes.
l'élaboration de ces différents programmes nécéssitent une connaissance approfondis du langage Fortran 90, comme la programmation modulaire, les dérivées types, pointeurs et les tableaux dynamiques.

5.1/-Programmation par échange de messages

5.1.1- Notion de programmation séquentiel

Le programme est exécuté par un et un seul processus; toutes les variables et constantes du programme sont allouées dans la mémoire allouée au processus; un processus s'exécute sur un processeur physique de la machine.

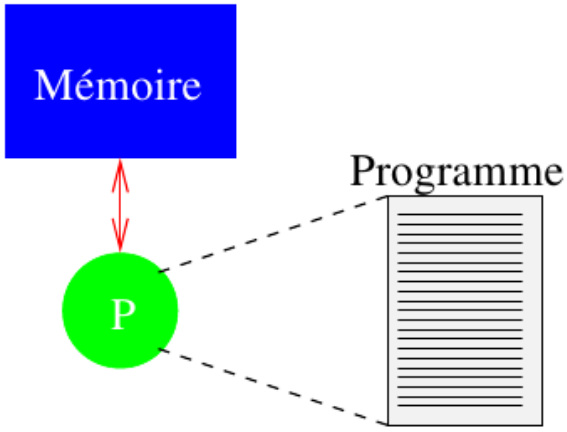


Figure 5.1 – Schéma de programmation séquentielle

5.1.2- Modéle de programme

Le programme est écrit dans un langage classique (Fortran, C, C++, etc.); toutes les variables du programme sont privées et résident dans la mémoire locale allouée à chaque processus; chaque processus exécute éventuellement des parties différentes d'un programme; une donnée est échangée entre deux ou plusieurs processus via un appel, dans le programme, à des sous-programmes particuliers.

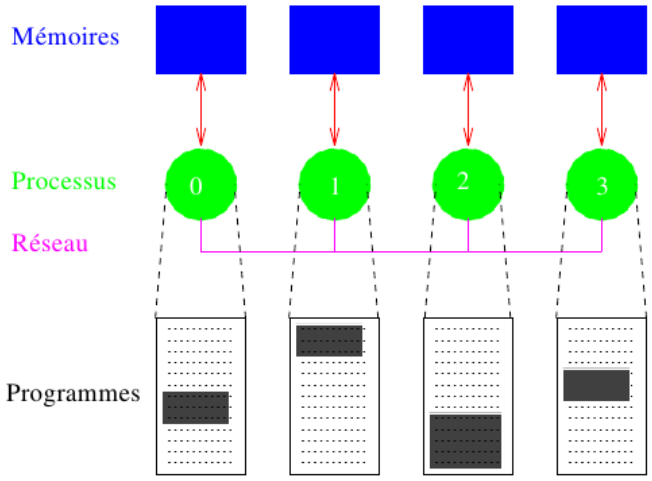


Figure 5.2 – Programmation par échange de messages

5.1.3 - Concept de l'échange de messages

Si un message contenu dans le systéme et qu'on désire envoyer du processus initial à un processus autre, celui-ci doit ensuite le recevoir

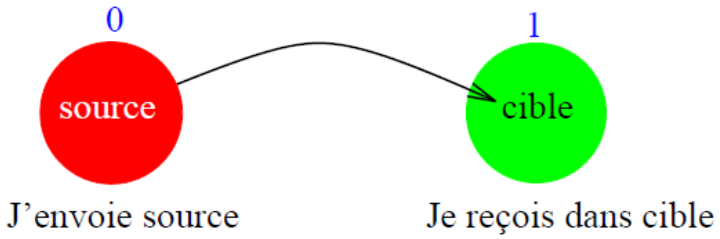


Figure 5.3 – schéma d'un échange de message

5.1.4 - Organisation d'un message

Un message s'organise par paquets de données en passant du processus initial au(x) processus final(s)

En plus des données (variables scalaires, variables complexes, tableaux, etc.) à envoyer, un message doit se composer des informations suivantes :

- l'identificateur du processus émetteur ;
- le type de la donnée ;
- sa longueur ;
- l'identificateur du processus récepteur.

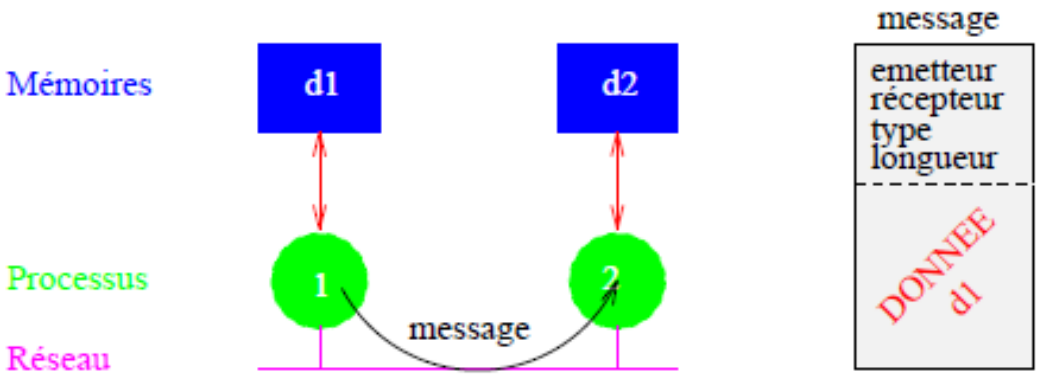


Figure 5.4 – Organisation d'un message

5.1.5- Environnement du calcul MPI

Les messages échangés lors du calcul sont interprétés et gérés par un environnement qui peut être comparé à la téléphonie, à la télécopie, au courrier postal, à la messagerie électronique, etc.
Le message est envoyé à une adresse déterminée.
Le processus qui recoit le (s) message(s) doit pouvoir classer et interpréter les messages qui lui ont été adressés.
L'environnement étudié est MPI (Message Passing Interface).
Une application utilisant le MPI est un ensemble de processus autonomes exécutant chacun leur propre code et communiquant via des appels à des sous-programmes qu'on retrouve dans la bibliothèque MPI.

a- Architectures des calculateurs intensifs

La plupart des supercalculateurs sont des machines à mémoire distribuée. Ils sont composés d'un ensemble de nœuds, à l'intérieur d'un nœud la mémoire est partagée.

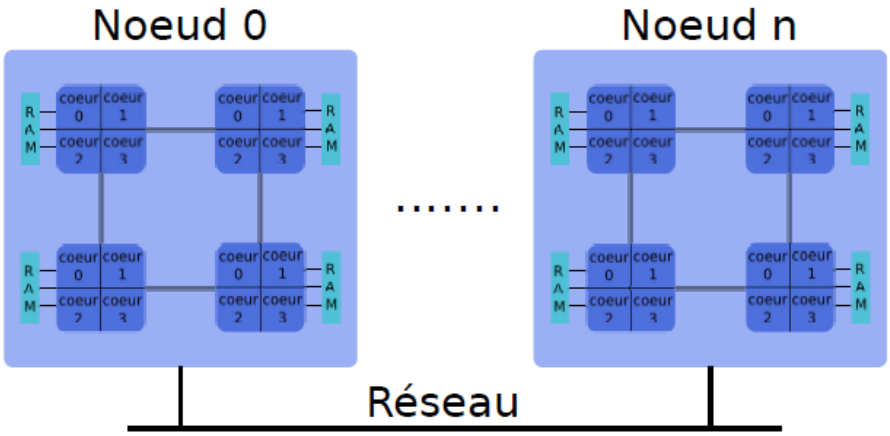


Figure 5.5 – Architecture des calculateurs intensifs

Chapitre 5 :**Notions et méthodes dans le calcul MPI**

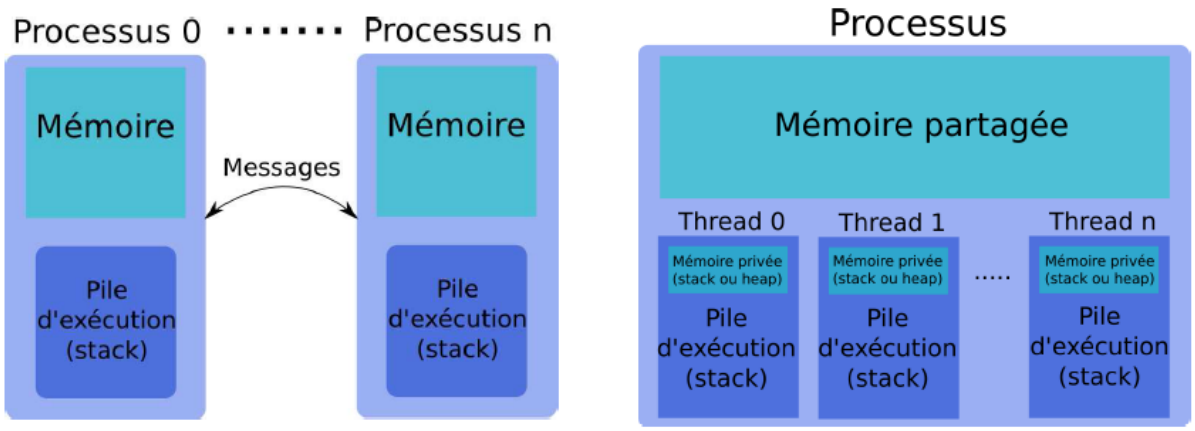


Figure 5.6 – Schéma de calcul par MPI

b- Décomposition de domaine

Un schéma que l'on rencontre très souvent dans le calcul MPI est la décomposition de domaine. Chaque processus possède une partie du domaine global, et effectue principalement des échanges avec ses processus voisins.

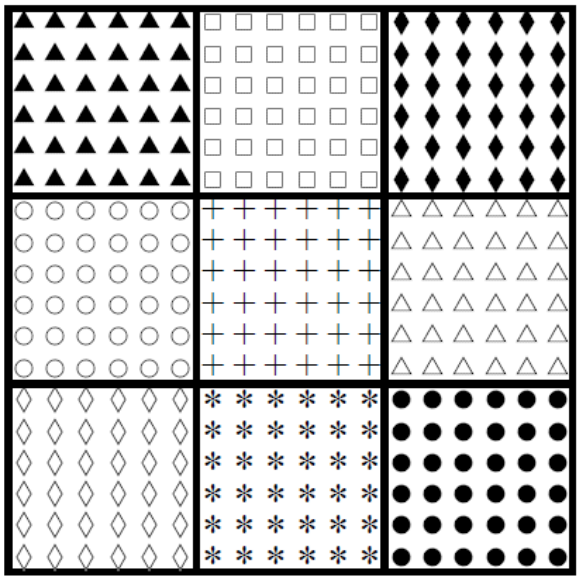


Figure 5.7– Découpage en sous-domaines

c – Analyses de données

Toute unité de programme appelant des sous-programmes MPI doit inclure un fichier d'en-têtes. En Fortran, il faut utiliser le module MPI introduit dans MPI-2.
Dans MPI-1, il s'agissait du fichier mpif.H.

Chapitre 5 :**Notions et méthodes dans le calcul MPI**

Le sous-programme MPI_INIT permettent d'initialiser l'environnement nécessaire :

```
integer, intent(out) :: code
call MPI_INIT (code)
```

Réciproquement, le sous-programme **MPI_FINALIZE** désactive cet environnement :

```
integer, intent(out) :: code
call MPI_FINALIZE (code)
```

Communicateurs
Toutes les opérations effectuées par MPI traitent principalement des communicateurs.
Le communicateur par défaut est MPI_COMM_WORLD qui comprend tous les processus actifs.

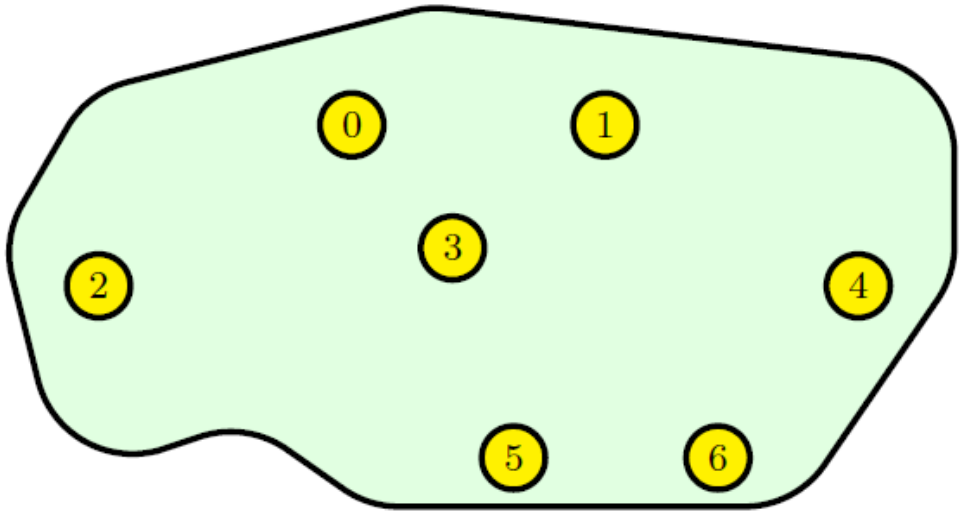


Figure 5.8 – Communicateur MPI_COMM_WORLD

Arrêt d'un programme
Parfois un programme se trouve dans une situation où il doit s'arrêter sans attendre la fin normale.
C'est typiquement le cas si un des processus ne peut pas allouer la mémoire nécessaire à son calcul.
Dans ce cas il faut utiliser le sous-programme MPI_ABORT et non l'instruction Fortran stop.

```
integer, intent(in) :: comm, erreur
integer, intent(out) :: code
call MPI_ABORT (comm, erreur, code)
```

comm : tous les processus appartenant à ce communicateur seront stoppés
erreur : numéro d'erreur retourné à l'environnement UNIX.
Il n'est pas nécessaire de tester la valeur de code après des appels aux routines MPI.

Chapitre 5 :**Notions et méthodes dans le calcul MPI**

Par défaut, lorsque MPI rencontre un problème, le programme s'arrête comme lors d'un appel à MPI_ABORT().

Rang et nombre de processus
À tout instant, on peut connaître le nombre de processus gérés par un communicateur donné par le sous-programme MPI_COMM_SIZE :

```
integer, intent(out) :: nb_procs,code
call MPI_COMM_SIZE ( MPI_COMM_WORLD , nb_procs,code)
```

De même, le sous-programme MPI_COMM_RANK permet d'obtenir le rang d'un processus (i.e. son numéro d'instance, qui est un nombre compris entre 0 et la valeur renvoyée par– 1) :

```
integer, intent(out) :: rang,code
call MPI_COMM_RANK ( MPI_COMM_WORLD ,rang,code)
```

Exemple :
- Les communications point-à-point permettent à deux processus à l'intérieur d'un même communicateur d'échanger une donnée (scalaire, tableau ou type dérivé).
Les fonctions correspondantes sont MPI_Send, MPI_Recv et MPI_Sendrecv.
un premier programme décrivant l'échange par message entre différents processus lors d'un calcul MPI est éxprimé par l'organigramme ci-dessous :

Organigramme :
- Introduire le Nom du programme « qui_je_suis »
- Appel des procedures mise en jeu:
    * initialisation de l'environnement MPI,
    * initialisation du nombre et du rang de processus
- Affichage du résultats
- Arrêt du calcul

Listing du programme :

```
program qui_je_suis
use mpi
implicit none
integer :: nb_procs,rang,code
call MPI_INIT (code)
call MPI_COMM_SIZE ( MPI_COMM_WORLD ,nb_procs,code)
call MPI_COMM_RANK ( MPI_COMM_WORLD ,rang,code)
print *,'Je suis le processus ',rang,' parmi ',nb_procs
call MPI_FINALIZE (code)
end program qui_je_suis
```

Resultat après execution
> mpiexec -n 7 qui_je_suis

Je suis le processus 3 parmi 7
Je suis le processus 0 parmi 7
Je suis le processus 4parmi 7
Je suis le processus 1 parmi 7
Je suis le processus 5 parmi 7
Je suis le processus 2 parmi 7
Je suis le processus 6parmi 7

5.2 – Utilisation de la Communications point à point

5.2.1 – Notions de bases

Un des modes de communication beaucoup utilisée en MPI est la communication point à point. Elle se fait entre deux processus, l'un appelé processus émetteur et l'autre processus récepteur (ou destinataire).

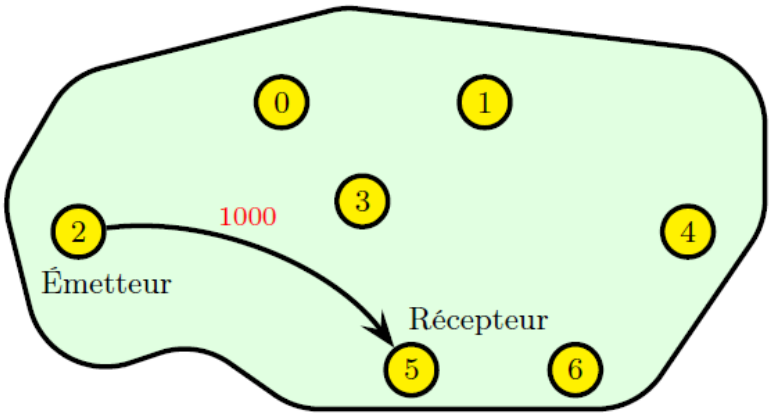


Figure 5.9 – Communication point à point

L'émetteur et le récepteur sont identifiés par leur rang dans le communicateur. l'enveloppe d'un message est constitué :
du rang du processus émetteur ;
du rang du processus récepteur ;
de l'étiquette (tag) du message ;
du communicateur qui définit le groupe de processus et le contexte de communication.
Les données échangées sont typées (entiers, réels…etc ou types dérivés personnelles).
Il existe dans chaque cas plusieurs modes de transfert, faisant appel à des protocoles différents.

5.2.2 – Opérations d'envoi et de réception bloquantes

Opération d'envoi MPI_SEND
<type et attribut>:: message
integer :: longueur, type

```
integer :: rang_dest, etiquette, comm, code
call  MPI_SEND (message,longueur,type,rang_dest,etiquette,comm,code)
```

Envoi, à partir de l'adresse message, d'un message de taille longueur, de type type, étiqueté étiquette, au processus rang_dest dans le communicateur comm.

Remarque :
Cette opération est bloquante : l'exécution reste bloquée jusqu'à ce que le contenu de message puisse être réécrit sans risque d'écraser la valeur qui devait être envoyée.

Opération de réception MPI_RECV

```
type et attribut>:: message
    integer :: longueur, type
    integer :: rang_source, etiquette, comm, code
    integer, dimension( MPI_STATUS_SIZE ) :: statut
    call MPI_RECV (message,longueur,type,rang_source,etiquette,comm,statut,code)
```

Réception, à partir de l'adresse message, d'un message de taille longueur, de type, étiqueté etiquette, du processus rang_source.

Remarques :
statut reçoit des informations sur la communication : rang_source, étiquette, code, ...
L'appel MPI_RECV ne pourra fonctionner avec une opération MPI_SEND que si ces deux appels ont la même enveloppe (rang_source, rang_dest, etiquette, comm).
Cette opération est bloquante : l'exécution reste bloquée jusqu'à ce que le contenu de message correspond au message reçu.

Exemple :
Un deuxiéme cas de programmation par calcul MPI est présenté par cet exemple ci-dessous par l'utilisation des communications points à points (ping-pong) entre différents processus.
Le principe de base est l'envoi de messages entre 02 ou plusieurs processus.

Organigramme :
- Introduire le Nom du programme « qui_je_suis »
- Appel des procedures mise en jeu:
    * initialisation de l'environnement MPI,
    * initialisation du nombre et du rang de processus
- Appel du communicateur envoi par MPI_SEND
- test du contenu des messages mise nen jeu
- Appel du communicateur recevoir par MPI_RECV
- Affichage du résultats
- Arrêt du calcul

listing du programme :

```
program point_a_point
use mpi
implicit none
integer, dimension( MPI_STATUS_SIZE ) :: statut
```

```
integer, parameter
:: etiquette=100
integer
:: rang,valeur,code
call MPI_INIT (code)
call MPI_COMM_RANK ( MPI_COMM_WORLD ,rang,code)
if (rang == 2) then
valeur=1000
call MPI_SEND (valeur,1, MPI_INTEGER ,5,etiquette, MPI_COMM_WORLD ,code)
elseif (rang == 5) then
call MPI_RECV (valeur,1, MPI_INTEGER ,2,etiquette, MPI_COMM_WORLD ,statut,code)
print *,'Moi, processus 5, j''ai reçu ',valeur,' du processus 2.'
end if
call MPI_FINALIZE (code)
end program point_a_point
```

Le résultat probable peut etre de la forme :

```
> mpiexec -n 7 point_a_point
Moi, processus 5 , j'ai reçu 1000 du processus 2
```

Autres possibilités

D'autres possibilités peuvent etre introduites selon la nature et le contenu du message envoyé par le processus envoi et la réception de ce message par le processus reception.
Opérations d'envoi et de réception simultanés

```
<type et attribut>:: message_emis, message_recu
        integer :: longueur_message_emis, longueur_message_recu
        integer :: type_message_emis, type_message_recu
        integer :: rang_source, rang_dest, etiq_source, etiq_dest, comm, code
        integer, dimension(                    ) :: statut
        call                (message_emis,longueur_message_emis,type_message_emis,
        rang_dest,etiq_source,
message_recu, longueur_message_recu, type_message_recu, rang_source, etiq_dest, comm, statut,
code)
```

Envoi, à partir de l'adresse message_emis, d'un message de taille longueur_message_emis, de type type_message_emis, étiquetté etiq_source, au processus rang_dest dans le communicateur comm;
réception, à partir de l'adresse message_recu, d'un message de taille longueur_message_recu, de type type_message_recu, étiquetté etiq_dest, du processus rang_source dans le communicateur comm.

Opérations d'envoi et de réception simultanées

l'organisation se fait selon l'organigramme suivant :
- Envoi, à partir de l'adresse message émis, d'un message de taille longueur_message_emis, de type type_message_emis, étiquetté etiq_source,
- au processus rang_dest dans le communicateur comm ;
sinon,

- réception, à partir de l'adresse message_recu, d'un message de taille longueur_message_recu, de type type_message_recu, étiquetté etiq_dest,
- du processus rang_source dans le communicateur comm.

On pourra l'organiser selon :

```
<type et attribut>:: message_emis, message_recu
integer :: longueur_message_emis, longueur_message_recu
integer :: type_message_emis, type_message_recu
integer :: rang_source, rang_dest, etiq_source, etiq_dest, comm, code
integer, dimension( MPI_STATUS_SIZE ) :: statut
call MPI_SENDRECV (message_emis,longueur_message_emis,type_message_emis,
rang_dest,etiq_source,
message_recu, longueur_message_recu, type_message_recu,
rang_source, etiq_dest, comm, statut, code)
```

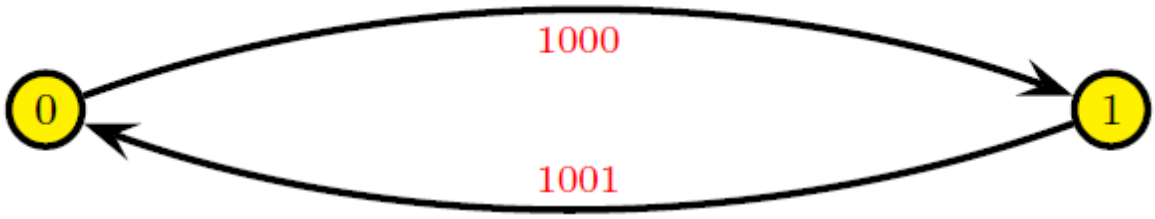


Figure 5.10 – Communication sendrecv entre les processus 0 et 1

```
program sendrecv
use mpi
implicit none
integer
integer,parameter
```

Autres possibilités

```
:: rang,valeur,num_proc,code
:: etiquette=110
call MPI_INIT (code)
call MPI_COMM_RANK ( MPI_COMM_WORLD ,rang,code)
! On suppose avoir exactement 2 processus
num_proc=mod(rang+1,2)
call MPI_SENDRECV (rang+1000,1, MPI_INTEGER ,num_proc,etiquette,valeur,1, MPI_INTEGER , &
num_proc,etiquette, MPI_COMM_WORLD , MPI_STATUS_IGNORE ,code)
print *,'Moi, processus',rang,', j''ai reçu',valeur,'du processus',num_proc
call MPI_FINALIZE (code)
end program sendrecv
```

```
> mpiexec -n 2 sendrecv
Moi, processus 1 , j'ai reçu 1000 du processus 0
Moi, processus 0 , j'ai reçu 1001 du processus 1
```

Exemple :
Un autre cas de programmation par calcul MPI qu'on nomme « joker« est présenté lors de cet exemple ci-dessous par :
- l'utilisation des communications points à points (ping-pong) entre différents processus.
- l'utilisation des fonctions de base dans l'environnement MPI comme les adresse, les dérivées types et autres.

Le principe de base est l'envoi de messages entre 02 ou plusieurs processus toutes en indiquant le contenu du message et son adresse de provenance d'un processus connu.

Organigramme :

- Introduire le Nom du programme « qui_je_suis »
- Appel des procedures mise en jeu:
  - * initialisation de l'environnement MPI,
  - * initialisation du nombre et du rang de processus
- Appel du communicateur envoi par MPI_SEND
- test du contenu des messages mise nen jeu
- Appel du communicateur recevoir par MPI_RECV
- Affichage du résultats

listing du programme :

```
program joker
use mpi
implicit none
integer, parameter :: m=4,etiquette=11
integer, dimension(m,m) :: A
integer :: nb_procs,rang,code,i
integer, dimension( MPI_STATUS_SIZE ):: statut
call MPI_INIT (code)
call MPI_COMM_SIZE ( MPI_COMM_WORLD ,nb_procs,code)
call MPI_COMM_RANK ( MPI_COMM_WORLD ,rang,code)
A(:,:) = 0
if (rang == 0) then
! Initialisation de la matrice A sur le processus 0
A(:,:) = reshape((/ (i,i=1,m*m) /), (/ m,m /))
! Envoi de 3 éléments de la matrice A au processus 1
call MPI_SEND (A(1,1),3, MPI_INTEGER ,1,etiquette, MPI_COMM_WORLD ,code)
else
! On reçoit le message
call MPI_RECV (A(1,2),3, MPI_INTEGER , MPI_ANY_SOURCE , MPI_ANY_TAG , &
MPI_COMM_WORLD ,statut,code)
print *,'Moi processus ',rang ,'je recois 3 elements du processus ', &
statut( MPI_SOURCE ), "avec l'etiquette", statut( MPI_TAG ), &
" les elements sont ", A(1:3,2)
```

```
end if
call MPI_FINALIZE (code)
end program joker
```

le résultat est :

```
> mpiexec -n 2 joker
Moi processus 1 je recois 3 elements du processus 3
avec l'etiquette 11 les elements sont 1  2  3
```

5.3 – Types de données dérivés

5.3.1 – Introduction

Dans les communications, les données échangées sont typées :

MPI_INTEGER, MPI_REAL, MPI_COMPLEX, etc.

Nous pouvons créer des structures de données plus complexes à l'aide de sous-programmes tels que :

MPI_TYPE_CONTIGUOUS(), MPI_TYPE_VECTOR(),
MPI_TYPE_INDEXED() ou MPI_TYPE_CREATE_STRUCT().

Les types dérivés permettent notamment l'échange de données non contiguës ou non homogènes en mémoire et de limiter le nombre d'appels aux sous-programmes de communications :

Types contigus
MPI_TYPE_CONTIGUOUS()

crée une structure de données à partir d'un ensemble homogène de type préexistant de données contiguës en mémoire :

```
call MPI_TYPE_CONTIGUOUS (5, MPI_REAL ,nouveau_type,code)
integer, intent(in) :: nombre, ancien_type
integer, intent(out) :: nouveau_type,code
call MPI_TYPE_CONTIGUOUS (nombre,ancien_type,nouveau_type,code)
```

Types avec un pas constant
MPI_TYPE_VECTOR() crée une structure de données à partir d'un ensemble homogène de type préexistant de données distantes d'un pas constant en mémoire. Le pas est donné en nombre d'éléments.

```
call MPI_TYPE_VECTOR (6,1,5, MPI_REAL ,nouveau_type,code)
```

- MPI_TYPE_CREATE_HVECTOR() crée une structure de données à partir d'un ensemble homogène de type préexistant de données distantes d'un pas constant en mémoire. Le pas est donné en nombre d'octets.

- Cette instruction est utile lorsque le type générique n'est plus un type de base (MPI_INTEGER, MPI_REAL,...) mais un type plus complexe construit à l'aide des sous-programmes MPI, parce qu'alors le pas ne peut plus être exprimé en nombre d'éléments du type générique.

Exemples sur Types de données dérivés
Les Types de données dérivés est une programmation moderniser d'une variables ou données sous une forme evolué.
Dans le calcul numérique qui est notre cas (équation de la chaleur) nous utilisons principalement les données liées à la matrice (vecteur statique ou tableau dynamique).
On peut toujours citer un cas sur les types de données dérivés :

Listing du programme:

```
program colonne
use mpi
implicit none
integer, parameter
integer, parameter
real, dimension(nb_lignes,nb_colonnes)
integer, dimension( MPI_STATUS_SIZE )
integer :: nb_lignes=5,nb_colonnes=6
… ! suite du programme
call MPI_INIT (code)
call MPI_COMM_RANK ( MPI_COMM_WORLD ,rang,code)
! Initialisation de la matrice sur chaque processus
a(:,:) = real(rang)
! Définition du type type_colonne
call MPI_TYPE_CONTIGUOUS (nb_lignes, MPI_REAL ,type_colonne,code)
! Validation du type type_colonne
call MPI_TYPE_COMMIT (type_colonne,code)
```

5.3 –Distribution d'un tableau sur plusieurs processus

Le calcul dedistribution d'un tableau par MPI est central dans ce type de probléme (résolution des équations différentielles partielles).
Le sous-programme MPI_TYPE_CREATE_DARRAY() permet de générer un tableau sur un ensemble de processus suivant une distribution par blocs ou cyclique.

```
integer,intent (in) :: nb_procs,rang,nb_dims
integer,dimension(nb_dims),intent(in) :: profil_tab,mode_distribution
integer,dimension(nb_dims),intent(in) :: profil_sous_tab,distribution_procs
integer,intent(in) :: ordre,ancien_type
integer,intent(out) :: nouveau_type,code
call MPI_TYPE_CREATE_DARRAY (nb_procs,rang,nb_dims,profil_tab,mode_distribution,
profil_sous_tab,distribution_procs,ordre,ancien_type, nouveau_type,code)
```

Listing du programme:

```
program darray_bloc
use mpi
implicit none
integer,parameter
:: nb_lignes=4,nb_colonnes=6, &
nb_dims=2,etiquette1=1000,etiquette2=1001
integer
:: nb_procs,code,rang,i,type_bloc
integer,dimension(nb_lignes,nb_colonnes) :: tab
integer,dimension(nb_dims)
:: profil_tab,mode_distribution, &
profil_sous_tab,distribution_procs
integer,dimension( MPI_STATUS_SIZE )
:: statut
call MPI_INIT (code)
call MPI_COMM_SIZE ( MPI_COMM_WORLD ,nb_procs,code)
call MPI_COMM_RANK ( MPI_COMM_WORLD ,rang,code)
! Initialisation du tableau tab sur chaque processus
tab(:,:)=reshape((/(i*(rang+1),i=1,nb_lignes*nb_colonnes)/),(/nb_lignes,nb_colonnes/))
! Profil du tableau tab
profil_tab(:) = shape(tab)
! Mode de distribution
mode_distribution(:) = (/ MPI_DISTRIBUTE_BLOCK , MPI_DISTRIBUTE_BLOCK /)
! Profil d'un bloc
profil_sous_tab(:) = (/ 2,3 /)
! Nombre de processus dans chaque dimension
distribution_procs(:) = (/ 2,2 /)
! Création du type dérivé type_bloc
call MPI_TYPE_CREATE_DARRAY (nb_procs,rang,nb_dims,profil_tab,mode_distribution, &
profil_sous_tab, distribution_procs, MPI_ORDER_FORTRAN , &
MPI_INTEGER ,type_bloc,code)
call MPI_TYPE_COMMIT (type_bloc,code)
select case(rang)
case(0)
! Le processus 0 envoie son tableau tab au processus 1
call MPI_SEND (tab,1,type_bloc,1,etiquette1, MPI_COMM_WORLD ,code)
case(1)
! Le processus 1 reçoit son tableau tab du processus 0
call MPI_RECV (tab,1,type_bloc,0,etiquette1, MPI_COMM_WORLD ,statut,code)
case(2)
! Le processus 2 envoie son tableau tab au processus 3
call MPI_SEND (tab,1,type_bloc,3,etiquette2, MPI_COMM_WORLD ,code)
case(3)
! Le processus 3 reçoit son tableau tab du processus 2
call MPI_RECV (tab,1,type_bloc,2,etiquette2, MPI_COMM_WORLD ,statut,code)
```

```
end select
! Affichage du tableau tab sur chaque processus
.....
call MPI_TYPE_FREE (type_bloc,code)
call MPI_FINALIZE (code)
end program darray_bloc
```

Chapitre 6 :
Applications et traitement de cas en calcul MPI

Chapitre 6 :Applications et traitement de cas en le calcul MPI

Avant de s'interesser à la résolution de l'équation de la chaleur par MPI qui est notre aboutissement principal, il est primordial d'aller de manière progressive pour la compréhension et la réalisation de programme de calcul numérique dédié au MPI.
en passant par la résoultion par la méthode séquentielle qu'on a présenté lors des chapitre 01,02 et 03 (partie A) ;
nous élaborons dans ce chapitre des programmes MPI pour divers problémes en mettant en pratique les notions de base du calcul MPI présentées dans les chapitres 04, 05 et 06 (partie B).
Ce chapitre 06 étudie une série de problèmes et d'applications très connus dans le domaine de la programmation classique linéaire.
Les programmes ont été élaborés dans le cadre de ce type de calcul. Nous avons introduits les concepts étudiés dans le chapitre 02.

Remarque :
Ces applications ont été testées dans l'environnement Linux sur une machine multiprocesseurs (Intel Core i5).

6.1- Applications sur le calcul MPI

Application 01

Gestion de l'environnement de MPI :
Dans cette application, le but est de faire afficher un message par chacun des processus, mais différent selon qu'ils sont de rang pair ou impair.
Dans cette application, il serait question de se familiariser avec la biliothéque MPI ainsi que l'environnement de calcul.
Nous abordons de façon sommaire l'organisation d'un programme MPI, toutes en utilisant des fonctions interne propre (voire chapitre 07) comme : call MPI_INIT; call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code) ; call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_processus,code); call MPI_FINALIZE(code)
Soit par exemple pour les processus de rang pair un message du genre :
Je suis le processus pair de rang M
Et pour les processus de rang impair un message du genre :
Je suis le processus impair de rang N

Remarque :
La fonction intrinsèque Fortran à utiliser pour tester la parité est mod :
mod(nombre1,nombre2)

Solution

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!
!!Traitement de message différent par les différents processeurs du calculateur
!!
!! pairs et les processus impairs
!!
!!
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

program pair_impair
USE MPI
implicit none

integer :: rang,nb_processus,code

call MPI_INIT(code)

call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
call MPI_COMM_SIZE(MPI_COMM_WORLD,nb_processus,code)

if (mod(rang,2) == 0) then
print *,'je suis le processus pair   ',rang
else
print *,'je suis le processus impair ',rang
end if

call MPI_FINALIZE(code)

end program pair_impair
```

Application 02 :

Utilisation des communications points à points - Ping-pong entre 02 processus

Ce programme permet de rentrer au cœur du MPI par la notion d'envoi de messages entre les différents processeurs qu'on nomme communication point à point ; toutes en gardant les notions acquises dans l'application 01.

Communications point à point :

Réalisation d'un ping-pong entre deux processus.

1- Dans le premier sous-exercice, on fera uniquement un ping (envoi d'un message (balle) du processus 0 au processus 1)

2- Dans le deuxième sous-exercice, on enchaînera le pong après le ping (le processus 1 renvoyant le message reçu du processus 0)

3- Dans le troisième sous-exercice, on effectuera une répétition du ping-pong en Faisant varier à chaque fois la taille du message à échanger

Soit :

1-Envoyer un message contenant 1000 nombres réels du processus 0 vers le processus 1 (il s'agit alors seulement d'un ping)

2-Faire une version ping-pong où le processus 1 renvoie le message reçu au processus 0 et mesurer le temps de communication à l'aide de la fonction MPI_WTIME()

3-Faire une version où l'on fait varier la taille du message dans une boucle et mesurer les temps de communication respectifs ainsi que les débits.

Chapitre 6 :Applications et traitement de cas en le calcul MPI

Remarques :
La génération de nombres réels pseudo-aléatoires uniformément répartis dans l'intervalle [0., 1.[se fait en Fortran par un appel au sous-programme random_number :

```
call random_number(variable)
```

- variable pouvant être un scalaire ou un tableau

Les mesures de temps peuvent s'effectuer de la façon suivante :

```
...................................................
temps_debut=MPI_WTIME ()
...................................................
temps_fin=MPI_WTIME ()
print ('("... en",f8.6," secondes.")'),temps_fin-temps_debut
...................................................
```

listing du programme

1-

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! -*- Mode: F90 -*- !!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Communications point à point :
!! envoi d'un message du processus 0 au processus 1
!!
!!
!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

program ping_pong_1

```
USE MPI
implicit none

integer, dimension(MPI_STATUS_SIZE)  :: statut
integer, parameter                   :: nb_valeurs=1000,etiquette=99
integer                         :: rang,code
integer, parameter                   :: dp = kind(1.d0)
real(kind=dp), dimension(nb_valeurs) :: valeurs

call MPI_INIT(code)

call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

if (rang == 0) then
call random_number(valeurs)
call MPI_SEND(valeurs,nb_valeurs,MPI_DOUBLE_PRECISION,1,etiquette, &
MPI_COMM_WORLD,code)
elseif (rang == 1) then
call MPI_RECV(valeurs,nb_valeurs,MPI_DOUBLE_PRECISION,0,etiquette, &
MPI_COMM_WORLD,statut,code)
```

```
print ('("Moi, processus 1, j"ai recu ",i4," valeurs (derniere = ", &
& f4.2,") du processus 0.")'), nb_valeurs,valeurs(nb_valeurs)
end if

call MPI_FINALIZE(code)

end program ping_pong_1
```

2-

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! -*- Mode: F90 -*- !!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!  : Communications point à point : ping-pong
!!
!!
!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

program ping_pong_2

```
USE MPI
implicit none

integer, dimension(MPI_STATUS_SIZE)  :: statut
integer, parameter                   :: nb_valeurs=1000,etiquette=99
integer                              :: rang,code
integer, parameter                   :: dp = kind(1.d0)
real(kind=dp)                        :: temps_debut,temps_fin
real(kind=dp), dimension(nb_valeurs) :: valeurs

call MPI_INIT(code)

call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

if (rang == 0) then
call random_number(valeurs)
temps_debut=MPI_WTIME()
call MPI_SEND(valeurs,nb_valeurs,MPI_DOUBLE_PRECISION,1,etiquette, &
MPI_COMM_WORLD,code)
call MPI_RECV(valeurs,nb_valeurs,MPI_DOUBLE_PRECISION,1,etiquette, &
MPI_COMM_WORLD,statut,code)
temps_fin=MPI_WTIME()
print ('("Moi, processus 0, j"ai envoye et recu ",i5, &
&" valeurs (derniere = ",f4.2,") du processus 1", &
&" en ",f8.6," secondes.")'), &
nb_valeurs,valeurs(nb_valeurs),temps_fin-temps_debut
elseif (rang == 1) then
call MPI_RECV(valeurs,nb_valeurs,MPI_DOUBLE_PRECISION,0,etiquette, &
MPI_COMM_WORLD,statut,code)
call MPI_SEND(valeurs,nb_valeurs,MPI_DOUBLE_PRECISION,0,etiquette, &
MPI_COMM_WORLD,code)
```

```
end if

call MPI_FINALIZE(code)

end program ping_pong2
```

3-

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! -*- Mode: F90 -*- !!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!  Communications point à point :
!!        ping-pong pour des tailles variables de messages
!!
!!
!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

program ping_pong_3
USE MPI
implicit none

integer, dimension(MPI_STATUS_SIZE)          :: statut
integer, parameter                           :: nb_valeurs_max=7000000, &
nb_tests=9,etiquette=99
integer, dimension(nb_tests)                 :: nb_valeurs
integer                                      :: rang,code,i
integer, parameter                           :: dp = kind(1.d0)
real(kind=dp), dimension(0:nb_valeurs_max-1) :: valeurs
real(kind=dp)                                :: temps_debut,temps_fin

call MPI_INIT(code)

nb_valeurs = (/ 0,1,10,100,1000,10000,100000,1000000,7000000 /)

call MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

do i=1,nb_tests
if (rang == 0) then
call random_number(valeurs)
temps_debut=MPI_WTIME()
call MPI_SEND(valeurs,nb_valeurs(i),MPI_DOUBLE_PRECISION,1,etiquette, &
MPI_COMM_WORLD,code)
call MPI_RECV(valeurs,nb_valeurs(i),MPI_DOUBLE_PRECISION,1,etiquette, &
MPI_COMM_WORLD,statut,code)
temps_fin=MPI_WTIME()
call affichage
elseif (rang == 1) then
call MPI_RECV(valeurs,nb_valeurs(i),MPI_DOUBLE_PRECISION,0,etiquette, &
MPI_COMM_WORLD,statut,code)
call MPI_SEND(valeurs,nb_valeurs(i),MPI_DOUBLE_PRECISION,0,etiquette, &
```

```
MPI_COMM_WORLD,code)
end if
end do

call MPI_FINALIZE(code)

contains
subroutine affichage

if (nb_valeurs(i)/=0) then
print ('("Moi, processus 0, j"ai envoye et recu ",i8, &
&" valeurs (derniere = ",f4.2,") du processus 1", &
&" en ",f8.6," secondes, soit avec un debit de ",f7.2, &
&" Mo/s.")'), &
nb_valeurs(i),valeurs(nb_valeurs(i)-1),temps_fin-temps_debut, &
real(2*nb_valeurs(i)*8)/1000000./(temps_fin-temps_debut)
else
print ('("Moi, processus 0, j"ai envoye et recu ",i8, &
&" valeurs en ",f8.6," secondes, soit avec un debit de ",f7.2, &
&" Mo/s.")'), &
nb_valeurs(i),temps_fin-temps_debut, &
real(2*nb_valeurs(i)*8)/1000000./(temps_fin-temps_debut)
end if
end subroutine affichage

end program ping_pong3
```

Application 03
Calcul de transposée d'une matrice

le calcul de transposée d'une matrice donnée est un probléme connu en méthodes numériques (algèbre des matrices).
À travers cette application, nous abordons la discrétisation de façon implicites...
Dans cet exercice, on se propose de se familiariser avec les types dérivés.
On se donne une matrice A de 5 lignes et 4 colonnes sur le processus 0.
Il s'agit pour le processus 0 d'envoyer au processus 1 cette matrice mais d'en faire automatiquement la transposition de la matrice au cours de l'envoi.

- Pour ce faire, on va devoir se construire deux types dérivés :
- un type type_ligne
- un type type_transpose

listing du programme

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! -*- Mode: F90 -*- !!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! transpose.f90  --- Utilisation d'un type derive (type_transpose)
!!                   pour transposer une matrice.
```

```
!!
!!
!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

PROGRAM transpose
USE MPI
IMPLICIT NONE
INTEGER, PARAMETER                  :: nb_lignes=5,nb_colonnes=4,&
etiquette=1000
INTEGER                             :: code,rang,type_ligne,&
type_transpose,taille_reel,i,j
REAL, DIMENSION(nb_lignes,nb_colonnes) :: A
REAL, DIMENSION(nb_colonnes,nb_lignes) :: AT
INTEGER(kind=MPI_ADDRESS_KIND)         :: pas
INTEGER, DIMENSION(MPI_STATUS_SIZE)    :: statut

!Initialisation de MPI
CALL MPI_INIT(code)

!-- Savoir qui je suis
CALL MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

!-- Initialisation de la matrice AT
AT(:,:) = 0.

!Connaitre la taille du type de base MPI_REAL
CALL MPI_TYPE_SIZE(MPI_REAL,taille_reel,code)

!Construction du type derive type_ligne qui correspond
!a une ligne de la matrice A
CALL MPI_TYPE_VECTOR(nb_colonnes,1,nb_lignes,MPI_REAL,type_ligne,code)

!Construction du type derive type_transpose pour transposer la
!matrice A composee de nb_lignes et de nb_colonnes
pas=taille_reel
CALL MPI_TYPE_CREATE_HVECTOR(nb_lignes,1,pas,type_ligne,type_transpose,code)

!Validation du type cree type_transpose
CALL MPI_TYPE_COMMIT(type_transpose,code)

IF (rang == 0) THEN
!Initialisation de la matrice A sur le processus 0
A(:,:) = RESHAPE( (/ (i,i=1,nb_lignes*nb_colonnes) /), &
(/ nb_lignes,nb_colonnes /) )

PRINT *,'Matrice A'
DO i=1,nb_lignes
```

```
PRINT *,A(i,:)
END DO

!Envoi de la matrice A au processus 1 avec le type type_transpose
CALL MPI_SEND(A,1,type_transpose,1,etiquette,MPI_COMM_WORLD,code)

ELSE
!Reception pour le processus 1 dans la matrice AT
CALL MPI_RECV(AT,nb_colonnes*nb_lignes,MPI_REAL,0,etiquette,&
MPI_COMM_WORLD,statut,code)

PRINT *,'Matrice transposee AT'
DO i=1,nb_colonnes
PRINT *,AT(i,:)
END DO

END IF

!Sortie de MPI
CALL MPI_FINALIZE(code)

END PROGRAM transpose

!!!!!!!!!!!!!!!!!!!!!!!!!!!! !!!!!!!!!!!!!!!!!!!!!!!!!!!
!! transpose.f90  --- Utilisation d'un type derive (type_transpose)
!!   calcul pour transposer une matrice.
!!
!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

PROGRAM transpose
USE MPI
IMPLICIT NONE
INTEGER, PARAMETER                  :: nb_lignes=5,nb_colonnes=4,&
etiquette=1000
INTEGER                             :: code,rang,type_ligne,&
type_transpose,taille_reel,i,j
REAL, DIMENSION(nb_lignes,nb_colonnes) :: A
REAL, DIMENSION(nb_colonnes,nb_lignes) :: AT
INTEGER(kind=MPI_ADDRESS_KIND)      :: nouvelle_taille,nouvelle_borne_inf=0
INTEGER, DIMENSION(MPI_STATUS_SIZE)    :: statut

!Initialisation de MPI
CALL MPI_INIT(code)

!-- Savoir qui je suis
CALL MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
```

```
!-- Initialisation de la matrice AT
AT(:,:) = 0.

!Connaitre la taille du type de base MPI_REAL
CALL MPI_TYPE_SIZE(MPI_REAL,taille_reel,code)

!Construction du type derive type_ligne qui correspond
!a une ligne de la matrice A
CALL MPI_TYPE_VECTOR(nb_colonnes,1,nb_lignes,MPI_REAL,type_ligne,code)

!Construction du type derive type_transpose pour transposer la
!matrice A composee de nb_lignes et de nb_colonnes
nouvelle_taille=taille_reel
CALL MPI_TYPE_CREATE_RESIZED(type_ligne,nouvelle_borne_inf,nouvelle_taille,&
type_transpose,code)

!Validation du type cree type_transpose
CALL MPI_TYPE_COMMIT(type_transpose,code)

IF (rang == 0) THEN
!Initialisation de la matrice A sur le processus 0
A(:,:) = RESHAPE( (/ (i,i=1,nb_lignes*nb_colonnes) /), &
(/ nb_lignes,nb_colonnes /) )

PRINT *,'Matrice A'
DO i=1,nb_lignes
PRINT *,A(i,:)
END DO

!Envoi de la matrice A au processus 1 avec le type type_transpose
CALL MPI_SEND(A,nb_lignes,type_transpose,1,etiquette,MPI_COMM_WORLD,code)

ELSE
!Reception pour le processus 1 dans la matrice AT
CALL MPI_RECV(AT,nb_colonnes*nb_lignes,MPI_REAL,0,etiquette,&
MPI_COMM_WORLD,statut,code)

PRINT *,'Matrice transposee AT'
DO i=1,nb_colonnes
PRINT *,AT(i,:)
END DO

END IF

!Sortie de MPI
CALL MPI_FINALIZE(code)

END PROGRAM transpose
```

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! !!!!!!!!!!!!!!!!!!!!!!!!!!!!
!! Utilisation d'un type derive (type_transpose)
!!        pour transposer une matrice.
!!
!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

PROGRAM transpose
USE MPI
IMPLICIT NONE
INTEGER, PARAMETER                :: nb_lignes=5,nb_colonnes=4,&
etiquette=1000
INTEGER                           :: code,rang,type_ligne,&
type_transpose,taille_reel,i,j
REAL, DIMENSION(nb_lignes,nb_colonnes) :: A
REAL, DIMENSION(nb_colonnes,nb_lignes) :: AT
INTEGER(kind=MPI_ADDRESS_KIND)        :: pas
INTEGER, DIMENSION(MPI_STATUS_SIZE)    :: statut

!Initialisation de MPI
CALL MPI_INIT(code)

!-- Savoir qui je suis
CALL MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)
!-- Initialisation de la matrice AT
AT(:,:) = 0.

!Connaitre la taille du type de base MPI_REAL
CALL MPI_TYPE_SIZE(MPI_REAL,taille_reel,code)

!Construction du type derive type_ligne qui correspond
!a une ligne de la matrice A
CALL MPI_TYPE_VECTOR(nb_colonnes,1,nb_lignes,MPI_REAL,type_ligne,code)

!Construction du type derive type_transpose pour transposer la
!matrice A composee de nb_lignes et de nb_colonnes
pas=taille_reel
CALL MPI_TYPE_CREATE_HVECTOR(nb_lignes,1,pas,type_ligne,type_transpose,code)

!Validation du type cree type_transpose
CALL MPI_TYPE_COMMIT(type_transpose,code)

IF (rang == 0) THEN
!Initialisation de la matrice A sur le processus 0
A(:,:) = RESHAPE( (/ (i,i=1,nb_lignes*nb_colonnes) /), &
(/ nb_lignes,nb_colonnes /) )
```

```
PRINT *,'Matrice A'
DO i=1,nb_lignes
PRINT *,A(i,:)
END DO

!Envoi de la matrice A au processus 1 avec le type type_transpose
CALL MPI_SEND(A,1,type_transpose,1,etiquette,MPI_COMM_WORLD,code)

ELSE
!Reception pour le processus 1 dans la matrice AT
CALL MPI_RECV(AT,nb_colonnes*nb_lignes,MPI_REAL,0,etiquette,&
MPI_COMM_WORLD,statut,code)

PRINT *,'Matrice transposee AT'
DO i=1,nb_colonnes
PRINT *,AT(i,:)
END DO

END IF

!Sortie de MPI
CALL MPI_FINALIZE(code)

END PROGRAM transpose
```

Application 04
Équation de Poisson

On considère l'équation de Poisson (classique)dans le domaine de calcul [0, 1]x[0, 1].
On va résoudre cette équation avec une méthode de décomposition de domaine.
l'ensemble de cette méthode a été éxposé dans les chapitres précédents.
L'équation est discrétisée sur le domaine par la méthode des différences finies.
Le système obtenu est résolu avec un solveur Jacobi.
Le principe de base est qu'on procéde au découpage du domaine global en sous domaines.
Il faut suivre les étapes suivantes :
initialiser l'environnement MPI ;
créer la topologie cartésienne 2D ;
déterminer les indices de tableau pour chaque sous-domaine ;
déterminer les 4 processus voisins d'un processus traitant un sous-domaine donné ;
créer deux types dérivés type_ligne et type_colonne ;
échanger les valeurs aux interfaces avec les autres sous-domaines ;
calculer l'erreur globale.
Lorsque l'erreur globale sera inférieure à une valeur donnée (précision machine par exemple), alors on considérera qu'on a atteint la solution.
Reformer la matrice u globale (identique à celle obtenue avec la version monoprocesseur) dans un fichier donnees.dat.

Chapitre 6 :Applications et traitement de cas en le calcul MPI

listing du programme

```
!!!!! module_calcul.f90
!!subroutine initialisation
!!subroutine calcul
!!subroutine sortie_resultats
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

MODULE calcul_poisson
USE TYPE_PARAMS
IMPLICIT NONE

!Coefficients
REAL(kind=dp), DIMENSION(1:3)                       :: coef
!Second membre
REAL(kind=dp), ALLOCATABLE, DIMENSION(:, :)           :: f

CONTAINS

SUBROUTINE initialisation(u, u_nouveau, u_exact)
!**************************
!Initialisation des valeurs
!**************************
!Solution u et u_nouveau a l'iteration n et n+1
REAL(kind=dp), ALLOCATABLE, DIMENSION(:, :), INTENT(out) :: u, u_nouveau
!Solution exacte
REAL(kind=dp), ALLOCATABLE, DIMENSION(:, :), INTENT(out) :: u_exact

!Compteurs
INTEGER                         :: i, j
!Coordonnees globales suivant x et y
REAL(kind=dp)                   :: x, y
!Pas en x et en y
REAL(kind=dp)                   :: hx, hy

!Allocation dynamique des tableaux u, u_nouveau, u_exact, f
ALLOCATE(u(sx-1:ex+1, sy-1:ey+1), &
u_nouveau(sx-1:ex+1, sy-1:ey+1))
ALLOCATE(f(sx-1:ex+1, sy-1:ey+1), &
u_exact(sx-1:ex+1, sy-1:ey+1))

!Initialisation des matrices
u(sx-1:ex+1, sy-1:ey+1)        = 0.
u_nouveau(sx-1:ex+1, sy-1:ey+1) = 0.
f(sx-1:ex+1, sy-1:ey+1)        = 0.
u_exact(sx-1:ex+1, sy-1:ey+1)   = 0.

!Pas
```

```
hx = 1./REAL(ntx+1)
hy = 1./REAL(nty+1)

!Calcul des coefficients
coef(1) = (0.5*hx*hx*hy*hy)/(hx*hx+hy*hy)
coef(2) = 1./(hx*hx)
coef(3) = 1./(hy*hy)

!Initialisation du second membre et calcul de la solution exacte
DO i=sx, ex
DO j=sy, ey
x = i*hx
y = j*hy
f(i, j) = 2*(x*x-x+y*y-y)
u_exact(i, j) = x*y*(x-1)*(y-1)
END DO
END DO

END SUBROUTINE initialisation

SUBROUTINE calcul(u, u_nouveau, x1, x2, y1, y2)
!*****************
! Calcul de la solution u_nouveau a l'iteration n+1
!*****************
!Solution u  a l'iteration n
REAL(kind=dp), ALLOCATABLE, DIMENSION(:, :), INTENT(in)   :: u

!Solution u_nouveau a l'iteration n+1
REAL(kind=dp), ALLOCATABLE, DIMENSION(:, :), INTENT(inout):: u_nouveau

! Borne du calcul
INTEGER, INTENT(in), OPTIONAL :: x1, x2, y1, y2

!Compteur
INTEGER                       :: i, j, ix1, ix2, iy1, iy2

IF (present(y2)) THEN
ix1 = x1
ix2 = x2
iy1 = y1
iy2 = y2
ELSE
ix1 = sx
ix2 = ex
iy1 = sy
iy2 = ey
END IF
DO j=iy1, iy2
```

```
DO i=ix1, ix2
u_nouveau(i, j) = coef(1) * (coef(2)*(u(i+1, j)+u(i-1, j)) &
+ coef(3)*(u(i, j+1)+u(i, j-1)) - f(i, j))
END DO
END DO

END SUBROUTINE calcul


SUBROUTINE sortie_resultats(u, u_exact)
!**************************
!Affichage
!**************************
!Solution u a l'iteration n
REAL(kind=dp), ALLOCATABLE, DIMENSION(:, :), INTENT(in) :: u
!Solution exacte
REAL(kind=dp), ALLOCATABLE, DIMENSION(:, :), INTENT(in) :: u_exact

INTEGER :: j

PRINT *, 'Solution exacte u_exact ', 'Solution calculee u'
DO j=sy, ey
PRINT 10, u_exact(1, j), u(1, j)
10     FORMAT('u_exact=  ', E12.5, ' u =  ', E12.5)
END DO

END SUBROUTINE  sortie_resultats

END MODULE  calcul_poisson

!module_parallel_mpi.f90
!!!!
!!subroutine initialisation_mpi
!!subroutine creation_topologie
!!subroutine domaine
!!subroutine voisinage
!!subroutine type_derive
!!subroutine communication
!!function erreur_globale
!!subroutine ecrire_mpi
!!subroutine finalisation_mpi
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

MODULE parallel
USE TYPE_PARAMS
USE MPI
IMPLICIT NONE
```

```
!Rang du sous-domaine local
INTEGER                         :: rang
!Nombre de processus
INTEGER                         :: nb_procs
!Communicateur de la topologie cartesienne
INTEGER                         :: comm2d
!Nombre de dimensions de la grille
INTEGER, PARAMETER              :: ndims = 2
!Nombre de processus dans chaque dimension definissant la topologie
INTEGER, DIMENSION(ndims)       :: dims
!Periodicite de  la topologie
LOGICAL, DIMENSION(ndims)       :: periods
! Coordonnees du domaine locale dans la topologie cartesienne
INTEGER, DIMENSION(ndims)       :: coords
! Tableau definissant les voisins de chaque processus
INTEGER, PARAMETER              :: NB_VOISINS = 4
INTEGER, PARAMETER              :: N=1, E=2, S=3, W=4
INTEGER, DIMENSION(NB_VOISINS)  :: voisin
! Types derives
INTEGER                         :: type_ligne, type_colonne
!Constantes MPI
INTEGER                         :: code

CONTAINS

SUBROUTINE initialisation_mpi
!************
!Initialisation pour chaque processus de son rang et du
!nombre total de processus nb_procs
!************

!Initialisation de MPI

!Savoir quel processus je suis

!Connaitre le nombre total de processus

END SUBROUTINE initialisation_mpi

SUBROUTINE creation_topologie
!************
!Creation de la topologie cartesienne
!************

!Constantes MPI
LOGICAL, PARAMETER              :: reorganisation = .FALSE.
```

```
! Lecture du nombre de points ntx en x et nty en y
OPEN(10, FILE='poisson.data', STATUS='OLD')
READ(10, *) ntx
READ(10, *) nty
CLOSE(10)

!Connaitre le nombre de processus selon x et le nombre de processus
!selon y en fonction du nombre total de processus

!Creation de la grille de processus 2D sans periodicite

IF (rang == 0) THEN
WRITE (*,'(A)') '-----------------------------------------'
WRITE (*,'(A,i4,A)') 'Execution code poisson avec ', nb_procs, ' processus MPI'
WRITE (*,'(A,i4,A,i4)') 'Taille du domaine : ntx=', ntx, ' nty=', nty
WRITE (*,'(A,i4,A,i4,A)') 'Dimension de la topologie : ', &
dims(1), ' suivant x, ', dims(2), ' suivant y'
WRITE (*,'(A)') '-----------------------------------------'
END IF

END SUBROUTINE creation_topologie

SUBROUTINE domaine
!************
!Calcul des coordonnées globales limites du sous domaine local
!************

! Connaitre mes coordonnees dans la topologie

!Calcul pour chaque processus de ses indices de debut et de fin suivant x
sx = (coords(1)*ntx)/dims(1)+1
ex = ((coords(1)+1)*ntx)/dims(1)

!Calcul pour chaque processus de ses indices de debut et de fin suivant y
sy = (coords(2)*nty)/dims(2)+1
ey = ((coords(2)+1)*nty)/dims(2)

WRITE (*,'(A,i4,A,i4,A,i4,A,i4,A,i4,A)') 'Rang dans la topologie : ', rang, &
' Indice des tableaux :', sx, ' a', ex, ' suivant x, ', &
sy, ' a', ey, ' suivant y'

END SUBROUTINE domaine

SUBROUTINE voisinage
!************
!Calcul des processus voisins pour chaque processus
!************
```

```
!Recherche des voisins Nord et Sud

!Recherche des voisins Ouest et Est

WRITE (*,'(A,i4,A,i4,A,i4,A,i4,A,i4)') "Processus ", rang, " a pour voisin : N", voisin(N), " E", voisin(E), &
" S", voisin(S), " W", voisin(W)

END SUBROUTINE voisinage

SUBROUTINE type_derive
!************
!Creation des types derives type_ligne et type_colonne
!************

!Creation du type type_ligne pour echanger les points
!au nord et au sud

!Creation du type type_colonne pour echanger
!les points  a l'ouest et a l'est

END SUBROUTINE type_derive

SUBROUTINE communication(u)
!************
!Echange des points aux interfaces
!************

REAL(kind=dp), ALLOCATABLE, DIMENSION(:, :), INTENT(inout) :: u

!Constantes MPI
INTEGER, PARAMETER                :: etiquette=100
INTEGER, DIMENSION(MPI_STATUS_SIZE)  :: statut

!Envoi au voisin N et reception du voisin S

!Envoi au voisin S et reception du voisin N

!Envoi au voisin W et reception du voisin E

!Envoi au voisin E et reception du voisin W

END SUBROUTINE communication

FUNCTION erreur_globale(u, u_nouveau)
!************
!Calcul de l'erreur globale (maximum des erreurs locales)
```

```
!************
REAL(kind=dp), ALLOCATABLE, DIMENSION(:, :), INTENT(in) :: u, u_nouveau

REAL(kind=dp)             :: erreur_globale, erreur_locale

erreur_locale = MAXVAL(ABS(u(sx:ex, sy:ey) - u_nouveau(sx:ex, sy:ey)))

!Calcul de l'erreur sur tous les sous-domaines

END FUNCTION erreur_globale

SUBROUTINE ecrire_mpi(u)
!*********************
! Ecriture du tableau u a l'interieur d'un domaine pour chaque processus
! dans le fichier donnees.dat
!*********************
REAL(kind=dp), ALLOCATABLE, DIMENSION(:, :), INTENT(inout) :: u

!Constantes MPI
INTEGER, DIMENSION(MPI_STATUS_SIZE)   :: statut
INTEGER                               :: descripteur
INTEGER(kind = MPI_OFFSET_KIND)       :: deplacement_initial
INTEGER, PARAMETER                    :: rang_tableau=2
INTEGER, DIMENSION(rang_tableau)      :: profil_tab, profil_sous_tab, coord_debut
INTEGER, DIMENSION(rang_tableau)      :: profil_tab_vue, profil_sous_tab_vue,
coord_debut_vue
INTEGER                               :: type_sous_tab, type_sous_tab_vue

!Ouverture du fichier "donnees.dat" en écriture

!Test pour verifier que le fichier a bien pu etre ouvert
IF (code /= MPI_SUCCESS) THEN
PRINT *, 'Erreur ouverture fichier'
CALL MPI_ABORT(comm2d, 2, code)
END IF

!Définition de la vue sur le fichier a partir du debut

!Creation du type derive type_sous_tab correspondant a la matrice u
!sans les cellules fantomes

!Ecriture du tableau u par tous les processus avec la vue

! Fermeture du fichier

END SUBROUTINE ecrire_mpi
```

```
SUBROUTINE finalisation_mpi
!************
!Desactivation de l'environnement MPI
!************

! Desactivation de MPI

END SUBROUTINE finalisation_mpi

END MODULE parallel

!*********************************************************************
!   poisson.f90 - Résolution de l'équation de Poisson par une méthode aux différences finies
!   en utilisant un solveur Jacobi sur le domaine [0,1]x[0,1].
!
!   Delta u = f(x,y)= 2*(x*x-x+y*y -y)
!   u sur les bords vaut 0
!   La solution exacte est u = x*y*(x-1)*(y-1)
!
!   La valeur de u est donnée par la formule:
!    coef(1) = (0.5*hx*hx*hy*hy)/(hx*hx+hy*hy)
!    coef(2) = 1./(hx*hx)
!    coef(3) = 1./(hy*hy)
!
!    u(i,j)(n+1)= coef(1) * ( coef(2)*(u(i+1,j)+u(i-1,j)) &
!              + coef(3)*(u(i,j+1)+u(i,j-1)) - f(i,j))
!
!   Dans cette version, on se donne le nombre de points interieurs
!   total suivant x (ntx) et le nombre de points interieurs total
!   suivant y (nty).
!
!   hx represente le pas suivant x, hy le pas suivant y.
!    hx = 1./(ntx+1)
!    hy = 1./(nty+1)
!
!   Pour chaque processus :
!   1) décomposer le domaine
!   2) connaître ses 4 voisins
!   3) échanger les points aux interfaces
!   4) calculer
!   5) recomposer la matrice u dans un fichier de sortie donnees.dat
!
!*********************************************************************

PROGRAM poisson
USE TYPE_PARAMS
USE PARALLEL
USE CALCUL_POISSON
```

```
IMPLICIT NONE

!Solution u et u_nouveau à l'itération n et n+1
REAL(kind=dp), ALLOCATABLE, DIMENSION(:, :) :: u, u_nouveau
!Solution exacte
REAL(kind=dp), ALLOCATABLE, DIMENSION(:, :) :: u_exact
!Nombre iterations en temps
INTEGER                     :: it
!Convergence
REAL(kind=dp)                  :: diffnorm, dwork
!Mesure du temps
REAL(kind=dp)                  :: t1, t2
!Test de convergence
LOGICAL                     :: convergence

!********************************************************************************
!Initialisation de MPI
CALL initialisation_mpi

!Creation de la topologie cartesienne 2D
CALL creation_topologie

!Determinination des indices de chaque sous domaine
CALL domaine

!Initialisation du second membre, u, u_nouveau et u_exact
CALL initialisation(u, u_nouveau, u_exact)

!Recherche de ses 4 voisins pour chaque processus
CALL voisinage

!Creation des types derives type_ligne et type_colonne
CALL type_derive

!Schema iteratif en temps
it = 0
convergence = .FALSE.

!Mesure du temps passe dans la boucle en temps (en secondes)
t1 = MPI_WTIME()

DO WHILE ((.NOT. convergence) .AND. (it < it_max))

it = it +1

u(sx:ex, sy:ey) = u_nouveau(sx:ex, sy:ey)
```

```
!Echange des points aux interfaces pour u a l'iteration n
CALL communication(u)

!Calcul de u a l'iteration n+1
CALL calcul(u,  u_nouveau)

!Calcul de l'erreur globale
diffnorm = erreur_globale(u, u_nouveau)

!Arret du programme si on a atteint la precision machine obtenu
!par la fonction F90 EPSILON
convergence = (diffnorm < eps)

!Affichage pour le processus 0 de la difference
IF ((rang == 0) .AND. (MOD(it, 100) == 0)) THEN
PRINT *, 'Iteration ', it, ' erreur_globale = ', diffnorm
END IF

END DO

!Mesure du temps a la sortie de la boucle
t2 = MPI_WTIME()

IF (rang == 0) THEN
!Affichage du temps de convergence par le processus 0
PRINT *, 'Convergence apres ', it, ' iterations en  ', t2 - t1, ' secs '

!Comparaison de la solution calculee et de la solution exacte
!sur le processus 0
CALL sortie_resultats(u, u_exact)
END IF

!Ecriture des resultats u(sx:ex, sy:ey)
!pour chaque processus
CALL ecrire_mpi(u)

!Desactivation de MPI
CALL finalisation_mpi

END PROGRAM poisson

PROGRAM lire_tab

USE mpi

IMPLICIT NONE

INTEGER, DIMENSION(MPI_STATUS_SIZE)           :: statut
```

```
INTEGER                              :: rang, code, descripteur, ntx, nty
integer, parameter                   :: dp = kind(1.d0)
REAL(kind=dp), ALLOCATABLE, DIMENSION(:, :)    :: u_lu
INTEGER(KIND=MPI_OFFSET_KIND)                  :: taille_fichier
INTEGER                              :: taille_reel

CALL MPI_INIT(code)

CALL MPI_COMM_RANK(MPI_COMM_WORLD, rang, code)

OPEN(11, FILE='poisson.data', STATUS='OLD')
READ(11, *) ntx
READ(11, *) nty
CLOSE(11)

ALLOCATE(u_lu(ntx, nty))
u_lu(:, :) = 0.d0

!Ouverture du fichier "donnees.dat" en écriture
CALL MPI_FILE_OPEN(MPI_COMM_WORLD, "donnees.dat", &
MPI_MODE_RDONLY, &
MPI_INFO_NULL, descripteur, code)

!Test pour savoir si ouverture du fichier est correcte
IF (code /= MPI_SUCCESS) THEN
PRINT *, 'ATTENTION erreur lors ouverture du fichier'
CALL MPI_ABORT(MPI_COMM_WORLD, 2, code)
CALL MPI_FINALIZE(code)
END IF

CALL MPI_FILE_GET_SIZE(descripteur, taille_fichier, code)
CALL MPI_TYPE_SIZE(MPI_DOUBLE_PRECISION, taille_reel, code)
if (taille_fichier /= ntx*nty*taille_reel) then
print *, " ATTENTION Donnees.dat n'a pas la bonne longueur (",taille_fichier,",",&
ntx*nty*taille_reel,")"
write(11,*) 0
else
CALL MPI_FILE_READ(descripteur, u_lu, SIZE(u_lu), &
MPI_DOUBLE_PRECISION, statut, code)
WRITE(11, 101)  u_lu
101 FORMAT (E19.12)
end if

CALL MPI_FILE_CLOSE(descripteur, code)

CALL MPI_FINALIZE(code)

END PROGRAM lire_tab
```

6.2- La résolution de l'équation de la chaleur par le calcul MPI

Pour résoudre l'équation de la chaleur par MPI, nous utilisons la méthode de décomposition de domaine.
Étant donné un nombre p de processeurs, nous divisons l'intervalle [A, B] que nous désirons calculer en sous-intervalles égaux par le nombre p.

Remarque :
ce traitement de cas englobe tout les principes déjà cités au préalable.

listing du programme

```
program main

!**********************************************************************
!
!!  program principal " main"
!  équation de la chaleur
!
!
use mpi

integer id
integer ierr
integer p
double precision wtime


call MPI_Init ( ierr )

call MPI_Comm_rank ( MPI_COMM_WORLD, id, ierr )

call MPI_Comm_size ( MPI_COMM_WORLD, p, ierr )

if ( id == 0 ) then
write ( *, '(a)' ) ''
write ( *, '(a)' ) 'HEAT_MPI:'
write ( *, '(a)' ) '  FORTRAN90/MPI version.'
write ( *, '(a)' ) '  Solve the 1D time-dependent heat equation.'
end if
!
!  Record the starting time.
!
if ( id == 0 ) then
```

```
wtime = MPI_Wtime ( )
end if

call update ( id, p )
!
!  Record the final time.
!
if ( id == 0 ) then
wtime = MPI_Wtime ( ) - wtime
write ( *, '(a)' ) ''
write ( *, '(a,g14.6)' ) '  Wall clock elapsed seconds = ', wtime
end if
!
!  Terminate MPI.
!
call MPI_Finalize ( ierr )
!
!  Terminate.
!
if ( id == 0 ) then
write ( *, '(a)' ) ''
write ( *, '(a)' ) 'HEAT_MPI:'
write ( *, '(a)' ) '  Normal end of execution.'
end if

stop
end
subroutine update ( id, p )

!**********************************************************************
!
! initialisation  des données du problème
!
!
!
!  Parameters:
!
!
!    Input, integer P, nombre de processeur
!
use mpi

integer, parameter :: n = 12

double precision boundary_condition
double precision cfl
double precision h(0:n+1)
integer h_file
```

```
double precision h_new(0:n+1)
integer i
integer id
double precision initial_condition

integer j
integer j_max
integer j_min
double precision k
integer p
double precision rhs
integer status(MPI_STATUS_SIZE)
integer tag
double precision time
double precision time_delta
double precision time_max
double precision time_min
double precision time_new
double precision x(0:n+1)
double precision x_delta
integer x_file
double precision x_max
double precision x_min

h_file = 11
j_max = 100
j_min = 0
k = 0.002
x_file = 12
time_max = 10.0
time_min = 0.0
x_max = 1.0
x_min = 0.0
!
!
!
if ( id == 0 ) then
write ( *, '(a)' ) "
write ( *, '(a)' ) '  Compute an approximate solution to the time dependent'
write ( *, '(a)' ) '  one dimensional heat equation:'
write ( *, '(a)' ) "
write ( *, '(a)' ) '    dH/dt - K * d2H/dx2 = f(x,t)'
write ( *, '(a)' ) "
write ( *, '(a,g14.6,a,g14.6)' ) &
'  for ', x_min, ' = x_min < x < x_max = ', x_max
write ( *, '(a)' ) "
write ( *, '(a,g14.6,a,g14.6)' ) &
'  and ', time_min, ' = time_min < t <= t_max = ', time_max
```

```
write ( *, '(a)' ) "
write ( *, '(a)' ) '  Boundary conditions are specified at x_min and x_max.'
write ( *, '(a)' ) '  Initial conditions are specified at time_min.'
write ( *, '(a)' ) "
write ( *, '(a)' ) '  The finite difference method is used to discretize the'
write ( *, '(a)' ) '  differential equation.'
write ( *, '(a)' ) "
write ( *, '(a,i8,a)' ) '  This uses ', p * n, ' equally spaced points in X'
write ( *, '(a,i8,a)' ) '  and ', j_max, ' equally spaced points in time.'
write ( *, '(a)' ) "
write ( *, '(a,i8,a)' ) '  Parallel execution is done using ', p, ' processors.'
write ( *, '(a)' ) '  Domain decomposition is used.'
write ( *, '(a,i8,a)' ) '  Each processor works on ', n, ' nodes,'
write ( *, '(a)' ) '  and shares some information with its immediate neighbors.'
end if
!
!  Set the X coordinates of the N nodes.
!  We don't actually need ghost values of X but we'll throw them in
!  as X(0) and X(N+1).
!
do i = 0, n + 1
x(i) = ( dble (         id * n + i - 1 ) * x_max   &
+ dble ( p * n - id * n - i     ) * x_min ) &
/ dble ( p * n             - 1 )
end do
!
!  In single processor mode, write out the X coordinates for display.
!
if ( p == 1 ) then

open ( unit = x_file, file = 'x_data.txt', status = 'unknown' )

write ( x_file, '(11f14.6)' ) x(1:n)

close ( unit = x_file )

end if
!
!  Set the values of H at the initial time.
!
time = time_min

h(0) = 0.0
do i = 1, n
h(i) = initial_condition ( x(i), time )
end do
h(n+1) = 0.0
```

```
time_delta = ( time_max - time_min ) / dble ( j_max - j_min )
x_delta = ( x_max - x_min ) / dble ( p * n - 1 )
!
!  Check the CFL condition, have processor 0 print out its value,

!
cfl = k * time_delta / x_delta / x_delta

if ( id == 0 ) then
write ( *, '(a)' ) ''
write ( *, '(a)' ) 'UPDATE'
write ( *, '(a,g14.6)' ) '  CFL stability criterion value = ', cfl
end if

if ( 0.5 <= cfl ) then
if ( id == 0 ) then
write ( *, '(a)' ) ''
write ( *, '(a)' ) 'UPDATE - Warning!'
write ( *, '(a)' ) '  Computation cancelled!'
write ( *, '(a)' ) '  CFL condition failed.'
write ( *, '(a,g14.6)' ) '  0.5 <= K * dT / dX / dX = ', cfl
end if
return
end if
!
!  In single processor mode, write out the values of H.
!
if ( p == 1 ) then

open ( unit = h_file, file = 'h_data.txt', status = 'unknown' )

write ( h_file, '(11f14.6)' ) h(1:n)

end if
!
!  Compute the values of H at the next time, based on current data.
!
do j = 1, j_max

time_new = ( dble (         j - j_min ) * time_max   &
+ dble ( j_max - j         ) * time_min ) &
/ dble ( j_max     - j_min )

!
!  The odd processors send H(1) to ID-1 and the even ones receive it as H(N+1),
!  then
!  the even processors send H(1) to ID-1 and the odd ones receive it as H(N+1).
!
```

```
!  Processor 0 does NOT send.
!  Processor P-1 does NOT receive.
!
tag = 1

if ( 0 < id .and. mod ( id, 2 ) == 1 ) then
call MPI_Send ( h(1), 1, MPI_DOUBLE_PRECISION, id-1, tag, &
MPI_COMM_WORLD, ierr )
else if ( id < p - 1 .and. mod ( id, 2 ) == 0 ) then
call MPI_Recv ( h(n+1), 1,  MPI_DOUBLE_PRECISION, id+1, tag, &
MPI_COMM_WORLD, status, ierr )
end if

if ( 0 < id .and. mod ( id, 2 ) == 0 ) then
call MPI_Send ( h(1), 1, MPI_DOUBLE_PRECISION, id-1, tag, &
MPI_COMM_WORLD, ierr )
else if ( id < p - 1 .and. mod ( id, 2 ) == 1 ) then
call MPI_Recv ( h(n+1), 1,  MPI_DOUBLE_PRECISION, id+1, tag, &
MPI_COMM_WORLD, status, ierr )
end if
!
!  The odd processors send H(N) to ID+1 and the even ones receive it as H(0),
!  then
!  the even processors send H(N) to ID+1 and the odd ones receive it as H(0),
!
!  Processor P-1 does NOT send.
!  Processor 0 does NOT receive.
!
tag = 2

if ( id < p - 1 .and. mod ( id, 2 ) == 1 ) then
call MPI_Send ( h(n), 1, MPI_DOUBLE_PRECISION, id+1, tag, &
MPI_COMM_WORLD, ierr )
else if ( 0 < id .and. mod ( id, 2 ) == 0 ) then
call MPI_Recv ( h(0), 1, MPI_DOUBLE_PRECISION, id-1, tag, &
MPI_COMM_WORLD, status, ierr )
end if
!
!  Update the temperature based on the four point stencil.
!
do i = 1, n
h_new(i) = h(i) &
+ ( time_delta * k / x_delta / x_delta ) &
* ( h(i-1) - 2.0 * h(i) + h(i+1) ) &
+ time_delta * rhs ( x(i), time )
end do
!
!  H at the extreme left and right boundaries was incorrectly computed
```

```
!
if ( 0 == id ) then
h_new(1) = boundary_condition ( x(1), time_new )
end if

if ( id == p - 1 ) then
h_new(n) = boundary_condition ( x(n), time_new )
end if
!
!  Update time and temperature.
!
time = time_new

do i = 1, n
h(i) = h_new(i)
end do
!
!  In single processor mode, add current solution data to output file.
!
if ( p == 1 ) then
write ( h_file, '(11f14.6)' ) h(1:n)
end if

end do

if ( p == 1 ) then
close ( unit = h_file )
end if

return
end
function boundary_condition ( x, time )

!**********************************************************************
!
!
! estimation des conditions aux limites de l'EDP
!
!  Parameters:
!
!    Input, double precision X, TIME, the position and time.
!
!    Output, double precision, the value of the boundary condition.
!
implicit none

double precision boundary_condition
```

```
double precision time
double precision x
!
!  Left condition:
!
if ( x < 0.5 ) then
boundary_condition = 100.0 + 10.0 * sin ( time )
else
boundary_condition = 75.0
end if

return
end
function initial_condition ( x, time )

!**********************************************************************
!
!! INITIAL_CONDITION , the differential equation.
! condition initial
!
!  Parameters:
!
!    Input, double precision X, TIME, the position and time.
!
!    Output, double precision INITIAL_CONDITION.
!
implicit none

double precision initial_condition
double precision time
double precision x

initial_condition = 95.0

return
end
function rhs ( x, time )

!**********************************************************************
!
!
! paramètre à droite de l'équation RHS
!
!  Parameters:
!
!    Input, double precision X, TIME, the position and time.
!
!    Output, double precision RHS, the right hand side.
```

```
!
double precision rhs
double precision time
double precision x

rhs = 0.0

return
end
```

Chapitre 07 :

Fonctions du calcul MPI en Fortran 90

Chapitre 07 : Fonctions du calcul MPI en Fortran 90

Dans cette partie, on se propose de lister des fonctions qu'on utilise dans l'environnement MPI, c'est une bibliothéque de fonctions interne les plus utilisées.

1 Environnement

1.1 Initialisation de l'environnement MPI

```
integer :: code
call MPI_INIT (<OUT> code)
```

1.2 Rang du processus

```
integer :: comm, rang, code
call MPI_COMM_RANK (<IN> comm, <OUT> rang, <OUT> code)
```

1.3 Nombre de processus

```
integer :: comm, nb_procs, code
call MPI_COMM_SIZE (<IN> comm, <OUT> nb_procs, <OUT> code)
```

1.4 Désactivation de l'environnement MPI

```
integer :: code
call MPI_FINALIZE (<OUT> code)
```

1.5 Arret d'un programme MPI

```
integer :: comm, error, code
call MPI_ABORT (<IN> comm, <IN> error, <OUT> code)
```

1.6 Prise de temps

```
real(kind=8) :: temps
temps= MPI_WTIME ()
call MPI_ISEND (
```

2. Communications point à point

2.1Envoi de message

```
<type et attribut>:: message
integer :: longueur, type, rang_dest
integer :: etiquette, comm, code
call MPI_SEND (
```

2.2 Envoi non bloquant de message

<type et attribut>:: message
integer :: longueur, type, rang_dest
integer :: etiquette, comm, requete, code
call MPI_ISEND (
<IN> message,
<IN> longueur,
<IN> type,
<IN> rang_dest,
<IN> etiquette,
<IN> comm,
<OUT> requete,
<OUT> code)

2.3 Reception de message

<type et attribut>:: message
integer :: longueur, type, rang_source
integer :: etiquette, comm, code
integer, dimension(MPI_STATUS_SIZE) :: statut
call MPI_RECV (
<OUT> message,
<IN> longueur,
<IN> type,
<IN> rang_source,
<IN> etiquette,
<IN> comm,
<OUT> statut,
<OUT> code)

2.4 Reception non bloquant de message

<type et attribut>:: message
integer :: longueur, type, rang_source
integer :: etiquette, comm, requete, code
call MPI_IRECV (
<OUT> message,
<IN> longueur,
<IN> type,
<IN> rang_source,
<IN> etiquette,
<IN> comm,
<OUT> requete,
<OUT> code)

Chapitre 07 : Fonctions du calcul MPI en Fortran 90

2.5 Envoi et réception de message

```
<type et attribut>:: message_emis, message_recu
integer :: longueur_message_emis, type_message_emis
integer :: etiquette_message_emis, etiquette_message_recu
integer :: longueur_message_recu, type_message_recu
integer :: rang_source, rang_dest, comm, code
integer, dimension( MPI_STATUS_SIZE ) :: statut
call MPI_SENDRECV (
<IN> message_emis,
<IN> longueur_message_emis,
<IN> type_message_emis,
<IN> rang_dest,
<IN> etiquette_message_emis,
<OUT> message_recu,
<IN> longueur_message_recu,
<IN> type_message_recu,
<IN> rang_source,
<IN> etiquette_message_recu,
<IN> comm,
<OUT> statut,
<OUT> code)
```

3 Types dérivés

3.1 Types contigus

```
integer :: nb_elements, ancien_type, nouveau_type, code
call MPI_TYPE_CONTIGUOUS (
<IN> nb_elements,
<IN> ancien_type,
<OUT> nouveau_type,
<OUT> code)
```

3.2 Types avec un pas constant

```
integer :: nb_elements, longueur_bloc
integer :: pas, ancien_type, nouveau_type, code
call MPI_TYPE_VECTOR (
<IN> nombre_bloc,
<IN> nbr_elt_par_bloc,
<IN> pas,
<IN> type_elt,
<OUT>
```

nouveau_type,
<OUT> code)
.....
integer :: nb_elements,longueur_bloc
integer(MPI_ADDRESS_KIND) :: pas
integer :: ancien_type, nouveau_type, code
call MPI_TYPE_CREATE_HVECTOR (
<IN> nombre_bloc,
<IN> nbr_elt_par_bloc,
<IN> pas,
<IN> type_elt,
<OUT> nouveau_type,
<OUT> code)

3.3 Types à pas variable

integer :: nb_elements, code
integer, dimension(nb_elements) :: longueur_bloc, pas
integer :: ancien_type, nouveau_type
call MPI_TYPE_INDEXED (
<IN> nb_elements,
<IN> longueur_bloc,
<IN> pas,
<IN> ancien_type,
<OUT> nouveau_type,
<OUT> code)

3.4 Types sous-tableau

integer :: nb_dims, adresse_debut, ordre
integer :: ancien_type, nouveau_type, code
integer, dimension(nb_dims) :: profil_tab, profil_sous_tab
integer, dimension(nb_dims) :: adresse_debut
call MPI_TYPE_CREATE_SUBARRAY (
<IN> nb_dims,
<IN> profil_tab,
<IN> profil_sous_tab,
<IN> adresse_debut,
<IN> ordre,
<IN> ancien_type,
<OUT> nouveau_type,
<OUT> code)

Conclusion

Conclusion

Le domaine des méthodes numériques est une discipline à l'interface des mathématiques et de l'informatique. Son développement est étroitement lié à celui des outils informatiques ; notament avec les derniers developpements dans le calcul scientifique cas du calcul paralléle en particulier le « Message Passing Interface - MPI ». Ces méthodes s'intéressent à la mise en pratique de solutions permettant de résoudre, par des calculs purement numériques, des problèmes mathématique liés généralement aux simulation numériques et à l'ingénierie. Plus formellement, elles regroupent l'application des algorithmes permettant de résoudre numériquement par discrétisation les problèmes de mathématiques cas de l'équation de la chaleur.

Ce document est une initiation au calcul parallèle par « Message Passing Interface » à travers une série d'applications. Le choix des ces applications a été fait de manière progressive en allant des notions les plus simples (réalisation des communications points à points entre 02 processus) vers de problémes pratiques (résolution de l'équation de la chaleur) afin de permettre au lecteur plus d'accessibilité dans ce domaine de calcul.

Nous avons rédiger ce travail en dédiant la section A aux cours des méthodes numériques appliquées enseignés au sein de notre Département (Génie-mécanique) suivi d'une deuxième section B rassemblant les notions générales et principaux outils du calcul MPI.
Toutes en gardant un aspect objectif sur la rédaction de ce document, le but inscrit est de permettre une ouverture sur un champ nouveau voir de nouvelles perspectives dans le domaine des méthodes numériques appliquées (calcul scientifique) et contribuer ainsi à l'enrichissement scientifique de notre établissement.

Références bibliographiques

[1] S. Godunov et Ryaben'kii
Introduction to the theory of difference schemes, Fizmatgiz, 1962.

[2] E. Godlewski et P. A. Raviart.
Numerical Approximation of Hyperbolic Systems of Conservation Laws, Appl. Math. Sci. 118, Springer-Verlag, New York, 1996.

[3] Message Passing Interface Forum, MPI :
A Message-Passing Interface Standard, Version 2.2, High Performance Computing Center Stuttgart (HLRS), 2009 https ://fs.hlrs.de/projects/par/mpi/mpi22/

[4] William Gropp, Ewing Lusk et Anthony Skjellum :
Using MPI : Portable Parallel Programming with the Message Passing Interface, second edition, MIT Press, 1999

[5] William Gropp, Ewing Lusk et Rajeev Thakur :
Using MPI-2, MIT Press, 1999

[6] Peter S. Pacheco :
Parallel Programming with MPI, Morgan Kaufman Ed., 1997

[7] Ian Foster, Designing and Building Parallel Programs :
http ://www.mcs.anl.gov/itf/dbpp/

[8] Lawrence Livermore Nationnal Laboratory :
https ://computing.llnl.gov/tutorials/mpi

[9] MPICH2 :
http ://www.mcs.anl.gov/research/projects/mpich2/
Open MPI : http ://www.open-mpi.org/

[10] http://aramis.obspm.fr/~semelin/enseignement.html : ce cours
http://wwwunix.mcs.anl.gov/dbpp/index.html : livre en ligne sur le parallélisme

[11] http://www.mpiforum.org/docs/mpi11html/mpireport.html : Manuel de référence (html)