Un environnement interactif pour la programmation des Méthodes de Décomposition de Domaine

THÈSE

présentée à la Faculté des sciences, pour obtenir

le grade de Docteur ès sciences, par

Noureddine El Mansouri

UNIVERSITÉ DE NEUCHÂTEL
Institut d'Informatique et d'Intelligence Artificielle
rue Émile-Argand 11
2007 Neuchâtel, Suisse

IMPRIMATUR POUR LA THÈSE

Un environnement interactif pour la programmation des méthodes de décomposition de domaine

de M. Noureddine El Mansouri

UNIVERSITÉ DE NEUCHÂTEL

FACULTÉ DES SCIENCES

La Faculté des sciences de l'Université de Neuchâtel sur le rapport des membres du jury,

MM. H.-H. Naegeli (directeur de thèse), P.-J. Erard, E. Zuur et M. Deville (EPFL)

autorise l'impression de la présente thèse.

Neuchâtel, 12 juin 1998

Le doyen:

F. Stoeckli

F. Stoeckli

Remerciements

Je tiens à remercier toutes le personnes qui, par leur participation et leurs encouragements, m'ont permis de mener à bonne fin ce travail de thèse.

En premier lieu, le Professeur Hans-Heinrich NÄGELI, mon directeur de thèse, pour son soutien, ses orientations et ses critiques constructives durant toutes ces années de recherche.

Le Docteur Saîd OUALIBOUCH pour son assistance , son aide et nos discussions qui m'ont permis d'élargir mes connaissances dans le domaine de l'analyse numérique.

Le Docteur Edouard ZUUR, directeur de la société time-steps, pour ses conseils et pour avoir accepter d'être membre du jury.

Le Professeur Pierre-Jean ERARD, directeur de l'Institut d'Informatique, pour son soutien, ses encouragements et pour avoir accepter d'être membre du jury.

Le Professeur Michel DEVILLE, directeur du Laboratoire de Mécanique des Fluides (EPFL), pour ses critiques constructives et pour avoir accepter d'être membre du jury.

Merci également à mes collègues et amis de l'Institut d'Informatique qui, par leur présence, ont donné à ces années passées à Neuchâtel beaucoup de gaieté et de joie dans le travail.

Merci à mon épouse Susy qui a su me donner son aide, ses encouragements et son soutien pour l'aboutissement de cette thèse.

Résumé

Les méthodes de décomposition de domaine (MDD) sont de plus en plus utilisées pour résoudre numériquement les équations aux dérivées partielles. Ces MDD sont devenues indispensables pour trois raisons:

1. physique: les sous-problèmes correspondant à des sous-domaines peuvent être découplés et résolus indépendamment;
2. algorithmique: elles sont des *pré-conditionneurs* (*i*.e., des accélérateurs) des algorithmes itératifs classiques;
3. parallélisme: elles peuvent être implantées naturellement sur des machines parallèles.

Sur le plan de l'implantation, les MDD possèdent, malheureusement, plusieurs variantes. L'efficacité de ces méthodes dépend à la fois de la structure physique du domaine et de la nature des EDP à résoudre.

Grâce à la notion de problème auxiliaire et aux différentes manières de les coordonner, nous avons défini un cadre simple et unifié qui permet de spécifier une MDD. La simplicité de ce cadre offre aux modélisateurs et aux numériciens un environnement de programmation interactif et flexible, qui non seulement leur permet de spécifier les MDD les plus courantes, mais aussi d'en développer de nouvelles.

Mots clés: équations aux dérivées partielles, simulation numérique, méthodes de décomposition de domaine, pré-conditionnement, outils pour la programmation numérique.

À mon fils Idris

Table des matières

Liste des algorithmes

Liste des figures

Introduction

Avec le développement des ordinateurs, la simulation numérique est devenue un complément indispensable à l'étude des phénomènes physiques.

Que ce soit pour des raisons économiques (utiliser des programmes au lieu de maquettes), de prévisions (météorologie, propagation du feu, etc.) ou écologiques (impact de la pollution, etc.), on voit apparaître, de plus en plus des programmes qui simulent le comportement de ces systèmes.

Pour aboutir à un programme de simulation, on a besoin de ces differentes compétences:

1. La compétence du **modélisateur**. Il est un "expert" dans l'étude de ces phénomènes. Il peut être un physicien, un hydrogéologue, un géologue, etc. Il est, dans notre contexte, tout sauf un numéricien ou un informaticien. Après avoir observé un phénomène, le modélisateur décrit le comportement (ou lois du comportement) de ce phénomène. Nous retenons dans notre dissertation la modélisation sous forme d'équations aux dérivées partielles.

2. La compétence du **numéricien**. Il possède des connaissances en sciences mathématiques et peut élaborer des méthodes ou des algorithmes numériques qui permettent la résolution des équations aux dérivées partielles. A priori, nous n'attendons pas de lui qu'il ait des connaissances dans le domaine de la programmation.

3. La compétence de l'**informaticien**. Il est responsable de la programmation des méthodes numériques que le numéricien met à sa disposition. Peu importe le type de machine ou le langage utilisé, il doit traduire ces méthodes en des programmes qui produisent des résultats. Le jugement de la concordance de ces résultats avec la réalité (basé sur l'observation) est du ressort du modélisateur.

Une relation de client à serveur s'établit entre ces trois acteurs: l'informaticien facilite le travail du numéricien et le numéricien celui du modélisateur.

En l'absence de dialogue, un sentiment de frustration est vite né chez l'informaticien, car lorsque les résultats ne correspondent pas aux attentes du modélisateur, l'informaticien ne sait pas où chercher pour corriger l'erreur. Est-ce dans le programme? Ou dans la méthode numérique? Le modèle est-il mal formulé? Telles sont les questions qu'il peut se poser.

Cette frustration entraîne le rejet de son travail par le modélisateur. Ces deux acteurs se séparent avec un constat d'échec. Par conséquent, nous remarquons que la plupart des programmes de simulation sont écrits par des numériciens, ou parfois par des modélisateurs. Quant à l'informaticien, il se contente d'écrire des utilitaires qui ne sont pas forcement destinés à la simulation numérique. Il suffit de voir la multitude des librairies de communications, de l'algèbre linéaire, etc.

Aujourd'hui, nous constatons – de plus en plus – une volonté évidente d'unir les efforts des numériciens et des informaticiens pour aboutir à des environnements de programmations qui satisfont les attentes des modélisateurs. Cette volonté se mesure par les nombreuses conférences nationales et internationales qui se tiennent chaque année.

Dans le cas des problèmes modélisés sous la forme d'équations aux dérivées partielles, notre objectif est de contribuer à cette unification des efforts en nous intéressant à la programmation des méthodes numériques et plus précisément, aux méthodes de décomposition de domaine.

Nous avons choisi les méthodes de décomposition de domaine car les applications concrètes utilisent un nombre important de données et donc le passage à la programmation parallèle est devenu indispensable. En plus, ces méthodes sont utiles pour les raisons suivantes:

1. **Physique:** elles permettent de décomposer le problème en des sous-problèmes qu'on peut découpler et résoudre indépendamment.

2. **Algorithmique:** ces méthodes jouent le rôle d'accélérateurs des algorithmes itératifs classiques.

3. **Parallélisme:** les sous-problèmes obtenus sont facilement adaptables à la programmation parallèle.

Dans cette thèse, nous montrons l'intérêt de ces méthodes et soulevons la difficulté quant à leur programmation. Notre but est de montrer qu'un environnement de développement *interactif* est utile à la programmation de ces méthodes, et de proposer un concept simple qui sera la base de la mise en place d'un tel environnement.

Sachant que l'utilisateur final de cet environnement est le modélisateur, nos objectifs principaux seront la *simplicité*, l'*adaptation* et l'*interaction*.

Pour cela, le rapport est structuré de la manière suivante. Dans le premier chapitre, nous donnons un aperçu des étapes par lesquelles il faut passer pour aboutir à une programmation de simulation numérique. Dans le deuxième, nous montrons les méthodes de décomposition de domaine les plus connues afin d'accentuer leurs diversités. Dans le troisième, nous proposons notre concept pour l'unification de ces méthodes et montrons qu'elles obéissent à ce concept. Dans le quatrième chapitre, suivant ce concept, nous donnerons une vue d'ensemble sur l'environnement que nous avons programmé et testé.

Chapitre 1

La simulation numérique

1.1 Introduction

Nous entendons par simulation numérique l'étude des comportements des systèmes physiques à l'aide d'ordinateurs.

D'une manière générale, le comportement d'un système physique discret est représenté par des équations algébriques discrètes pouvant être résolues par des algorithmes numériques. Par contre dans le cas d'un système physique continu, où le comportement est décrit à l'aide d'équations aux dérivées partielles (EDP), il est nécessaire de discrétiser ces équations; c'est-à-dire de les remplacer par des équations algébriques, et ensuite d'utiliser des méthodes numériques pour les résoudre.

On a principalement recours à deux types de discrétisation. La première est temporelle, où il s'agit de remplacer les EDP dépendantes du temps en une série d'EDP stationnaires (*i.e.*, indépendantes du temps). La deuxième discrétisation est spatiale. Il s'agit de remplacer des EDP stationnaires par des équations algébriques en ne faisant intervenir que les valeurs des fonctions à un certain nombre de points de l'espace physique traité.

Parmi les méthodes de discrétisation spatiale, on trouve la méthode des différences finies et la méthode des éléments finis (MEF).

Dans ce qui suit, nous nous intéresserons à la résolution des EDP stationnaires discrétisées à l'aide des MEF.

La résolution des EDP stationnaires à l'aide des MEF consiste à passer par les quatre étapes suivantes:

Formulation forte: modéliser ou décrire le comportement à l'aide des EDP en utilisant, en général, des résultats de physique.

Formulation faible: transformer ces EDP en des équations faisant appel à des intégrales.

Discrétisation: approcher les solutions sur des éléments finis et formuler les équations algébriques matricielles.

Résolution numérique: élaborer une méthode numérique pour résoudre le système d'équations algébriques obtenu.

Par souci de clarté pour les chapitres suivants, nous allons étudier de manière introductive ces quatre étapes. Il existe de nombreuses publications traitant en profondeur ces quatre

étapes, par exemple, le livre de Ciarlet (1978) pour la théorie sur l'utilisation des éléments finis, le livre de Dhatt et Touzot (1984) pour l'implantation, et pour terminer le livre de Pironneau (1988) pour la résolution des équations de la dynamique des fluides à l'aide des MEF.

Ce chapitre sera plus addressé à un informaticien qu'à un numéricien. Il montrera les connaissances de base, que nous pensons utiles, pour la compréhension de la mise en place des programmes de simulation numérique à l'aide des MEF.

1.2 Méthodes des éléments finis

Par rapport à un domaine physique ouvert $\Omega \subset I\!R^d$, $d = 1$, 2 ou 3, les EDP d'un système physique font intervenir une ou plusieurs variables ou fonctions u qui expriment les grandeurs considérées (température, vitesse, etc.). Ces fonctions sont approchées par d'autres fonctions u_h plus simples à calculer. Dans le cas de la méthode des éléments finis, il s'agit de:

1. Subdiviser le domaine Ω en un certain nombre de petites parties géométriques simples telles que des segments, des triangles, des cubes etc. Cette subdivision, appelée discrétisation, fait intervenir des points (voir la figure 1.1) qu'on désigne par noeuds. Les parties géométriques sont appelées éléments finis.

2. Définir des fonctions $u_h^e = u_{h|\mathcal{K}_e}$, restrictions de u_h sur l'élément fini $\mathcal{K}_e$, de telle façon que:

   (a) l'approximation de u_h^e ne fasse intervenir que les noeuds situés sur l'élément $\mathcal{K}_e$,

   (b) les fonctions u_h^e soient construites de manière à être continues sur $\mathcal{K}_e$.

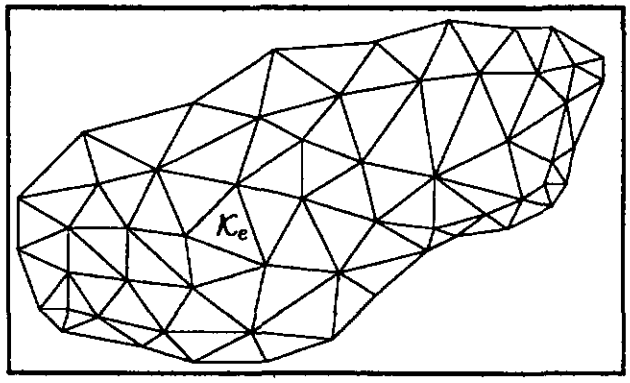

Figure 1.1 : discrétisation d'un domaine.

Nous nous intéressons aux fonctions u_h qui s'écrivent sous la forme:

$$u_h(x) = \sum_{i=1}^{N} u_i \varphi_i(x), \qquad \forall x \in \Omega$$

où N est le nombre de noeuds et $(\varphi_i)_{i\in 1,\ldots,N}$ sont des fonctions polynomiales par morceaux qui satisfassent:

$$\forall i, j \in \{1, \ldots, N\}, \qquad \varphi_i(x_j) = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j \end{cases}$$

Dans ce cas, la fonction recherchée est supposée être polynomiale par morceaux sur tout le domaine Ω.

1.2.1 Notations et définitions

Puisque la solution recherchée est une fonction, il est essentiel de préciser dans quel *espace de fonctions* cette solution doit être trouvée. Nous serons amenés à utiliser deux principaux espaces:

1. $L^2(\Omega)$ est l'ensemble des fonctions u de "carré intégrable", ou "carré sommable":

$$\int_\Omega |u|^2 \in \mathbb{R}^+ \qquad (i.e., \text{ est fini}) \tag{1.2.1}$$

2. $H^1(\Omega)$ est l'ensemble des fonctions dont les dérivées premières sont dans $L^2(\Omega)$.

De plus, nous définissons:

- $\Gamma = \partial\Omega$ est la frontière du domaine Ω.
- $H_0^1(\Omega) = \{u \in H^1(\Omega) |\ u = 0 \text{ sur } \Gamma\}$.
- $\mathbf{n}$ est la dérivée normale (unitaire) sortante du domaine Ω (à ne pas confondre avec les noeuds n_i.).
- $span\{\varphi_i\}$ est l'espace fonctionnel engendré par les fonctions φ_i: l'ensemble des fonctions qui s'écrivent sous forme d'une combinaison linéaire des φ_i.

Par souci de simplicité, nous ne tiendrons compte que des fonctions indépendantes du temps et scalaires (définies de Ω dans $\mathbb{R}$). L'écriture des intégrales se fera sans le "dx", comme dans la formule (1.2.1).

Durant cette thèse, nous allons nous restreindre aux problèmes de la forme:
trouver $u \in H_0^1(\Omega)$ *tel que*

$$-\Delta u = f \tag{1.2.2}$$

où Δ est l'opérateur de Laplace défini par $\Delta u = \sum_{i=1}^{d} \frac{\partial^2 u}{\partial x_i^2}$.

Tous les résultats obtenus sont valables pour les problèmes elliptiques et linéaires de de la forme:

$$-\sum_{i,j=1}^{d} \frac{\partial}{\partial x_i}(\alpha_{i,j} \frac{\partial u}{\partial x_j}) + \alpha_0 u = f \tag{1.2.3}$$

où $\alpha_{i,j}(x) = \alpha_{j,i}(x)$ sont des fonctions strictement positives et bornées sur Ω, et $\alpha_0(x)$ est une fonction positive ou nulle.

1.2.2 Formulation faible

La formulation sous forme d'intégrales consiste à transformer le problème (1.2.2) en: *trouver* $u \in H_0^1(\Omega)$ *telle que*

$$\int_\Omega v(-\Delta u - f) = 0 \tag{1.2.4}$$

où v sont des fonctions, appelées fonctions de test, bien choisies.

Nous allons nous intéresser à la méthode de Galerkin qui prend comme fonctions de test les fonctions appartenant à l'espace où la solution devrait être trouvée, c'est-à-dire $H_0^1(\Omega)$.

La formulation intégrale du problème (1.2.4) devient alors:
trouver $u \in H_0^1(\Omega)$ *tel que*:

$$-\int_\Omega v\,\Delta u = \int_\Omega f\,v, \qquad \forall v \in H_0^1(\Omega) \tag{1.2.5}$$

Ce problème est aussi allégé en utilisant l'intégration par parties (formule de Green):

$$\int_\Omega v\,\Delta u = -\int_\Omega \nabla u \cdot \nabla v + \int_\Gamma v\,\frac{\partial u}{\partial \mathbf{n}} \tag{1.2.6}$$

(∇ est le vecteur gradient; $\nabla u = \left(\frac{\partial u}{\partial x_i}\right)$, $i = 1, 2, 3$.)

En tenant compte du fait que v est nulle sur Γ, nous obtenons la formulation faible (dite formulation *variationnelle*) suivante:

$$\int_\Omega \nabla u \cdot \nabla v = \int_\Omega f\,v, \qquad \forall v \in H_0^1(\Omega) \tag{1.2.7}$$

Cette formulation nous permet non seulement de remplacer les dérivées secondes par les dérivées premières, mais aussi d'insérer à l'intérieur de la formulation les conditions aux limites. Dans notre cas, $u = 0$ sur Γ (voir la section 1.2.4 pour d'autres conditions.). L'autre atout de cette formulation est qu'après la discrétisation de l'intégrale à gauche (voir la section 1.2.3), nous obtiendrons une matrice A qui est à la fois symétrique et *définie positive* (SDP) (*i.e.*, $z \cdot (Az) > 0$ pour tout vecteur z non nul). Ces caractéristiques SDP de la matrice nous permettent d'avoir un grand choix d'algorithmes numériques pour la résolution du système d'équations obtenu après la discrétisation (Hageman et Young, 1981).

1.2.3 Discrétisation spatiale

La discrétisation spatiale, parfois appelée *triangulation*[1], d'un domaine consiste à construire les éléments finis $\mathcal{K}_e$ qui composent ce domaine. L'ensemble de ces éléments finis, $\mathcal{T}_h$, est appelé le *maillage* du domaine Ω. La figure 1.1 montre le cas où les éléments finis sont des triangles.

Le maillage est généralement caractérisé par le paramètre h défini grossièrement comme la moyenne des valeurs $h_e = \dfrac{\text{Diamètre}(\mathcal{K}_e)}{\text{Diamètre}(\Omega)}$ où Diamètre(G) est le diamètre du plus petit cercle

[1]Nous maintiendrons cette terminologie même si le domaine n'est pas dans $\mathbb{R}^2$ et même si les éléments ne sont pas des triangles

(ou sphère dans le cas tridimensionnel) contenant G. Pour étudier une méthode, généralement, on prend des domaines tels que Diamètre$(\Omega) = 1$. Dans ce cas, on a $h \leq 1$.

Soit N_h le nombre de noeuds obtenu par ce maillage. Nous désignerons les noeuds par n_i, $i = 1, \ldots, N_h$.

Soient (φ_i) les fonctions vérifiant: $\forall i, j = 1, \ldots, N_h$,

$$\begin{cases} \varphi_i(x) \text{ sont des polynômes par morceaux,} \\ \varphi_i(n_j) = \begin{cases} 1 & \text{si} \quad i = j, \\ 0 & \text{sinon} \end{cases} \end{cases}$$

La figure 1.2 montre le cas où le domaine $\Omega =]0, 1[$ est une barre élastique discrétisée par des

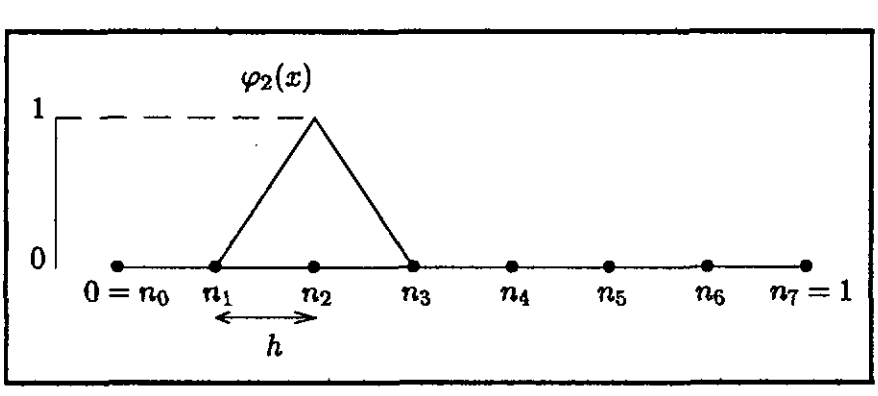

Figure 1.2 : exemple de fonctions de la base.

segments $[n_i, n_{i+1}]$, $i = 0, \ldots, 6$. Si tous les segments ont la même longueur h et les fonctions φ_i sont des polynômes par morceaux de degré 1, alors nous avons pour tout i:

$$\varphi_i(x) = \begin{cases} (x - n_{i-1})/(n_i - n_{i-1}) = (x - n_{i-1})/h & \text{si} \quad x \in [n_{i-1}, n_i],\ i > 0 \\ (x - n_{i+1})/(n_i - n_{i+1}) = (n_{i+1} - x)/h & \text{si} \quad x \in [n_i, n_{i+1}],\ i < 7 \\ 0 & \text{sinon} \end{cases}$$

La solution approchée, u_h, est dans l'espace fonctionnel de dimension finie $V^h = span\{\varphi_i\} \cap H_0^1(\Omega)$. Dans le cas de la barre, nous avons $V^h = span\{\varphi_i\}$, $i = 1, \ldots, 6$.

Cette fonction est solution de l'équation sous forme variationnelle (Ciarlet, 1978):

$$\int_\Omega \nabla u_h \cdot \nabla v_h = \int_\Omega f\, v_h, \qquad v_h \in V^h \tag{1.2.8}$$

Généralement, nous écrivons cette équation sous la forme: $\mathrm{a}(u_h, v_h) = \mathrm{f}(v_h)$, où

$$\mathrm{a}(u_h, v_h) = \int_\Omega \nabla u_h \cdot \nabla v_h \text{ et } \mathrm{f}(v_h) = \int_\Omega f\, v_h$$

En prenant $\mathrm{u} = (\mathrm{u}_1, \ldots, \mathrm{u}_{N_h}) \in I\!R^{N_h}$, les composantes de u_h dans V^h, alors ce vecteur est la solution du système d'équations algébriques:

$$\mathrm{Au} = \mathrm{F} \tag{1.2.9}$$

où $A = (a_{ij})$, une matrice réelle de dimension $N_h \times N_h$ et $F = (F_i)$, un vecteur de dimension N_h tels que $\forall i, j = 1, \ldots, N_h$:

$$\begin{cases} a_{ij} = a(\varphi_i, \varphi_j) = \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_j \\ F_i = f(\varphi_i) = \int_\Omega f\varphi_i \end{cases}$$

En effet, en utilisant φ_i comme fonction de test dans l'équation (1.2.8), nous avons:

$$\begin{aligned} a(u_h, \varphi_i) &= f(\varphi_i) \\ \int_\Omega \nabla\varphi_i \cdot \nabla u_h &= \int_\Omega f\varphi_i \\ \sum_{j=1}^{N_h} u_j \int_\Omega \nabla\varphi_i \cdot \nabla\varphi_j &= F_i \\ \sum_{j=1}^{N_h} a_{i,j} u_j &= F_i \end{aligned}$$

En répétant l'équation pour tous les $i = 1, \ldots, N_h$, nous avons l'équation matricielle (1.2.9).

La matrice A est SDP et **creuse** (*i.e.*, contient beaucoup de zéros).

Finalement, en notant

$$\begin{cases} a^e_{i,j} = a^e(\varphi_i, \varphi_j) = \int_{\mathcal{K}_e} \nabla\varphi_i \cdot \nabla\varphi_j \\ F^e_i = f^e(\varphi_i) = \int_{\mathcal{K}_e} f\varphi_i \end{cases}$$

et en tenant compte de l'égalité $\int_\Omega = \sum_{\mathcal{K}_e \in \mathcal{T}_h} \int_{\mathcal{K}_e}$ nous avons: $\forall i, j = 1, \ldots, N_h$

$$a_{i,j} = \sum_{\mathcal{K}_e \in \mathcal{T}_h} a^e_{i,j}; \qquad F_i = \sum_{\mathcal{K}_e \in \mathcal{T}_h} F^e_i$$

Dans la pratique, on peut calculer ces intégrales relativement à chaque élément fini et ensuite additionner la contribution de chacun de ces éléments à la matrice A et au vecteur F pour obtenir le système d'équations final. Cette technique, appelée *assemblage*, est une procédure automatique (pré-programmée) du fait que généralement tous les éléments finis sont géométriquement de même type. Il suffit de calculer une seule fois a^r et F^r par rapport à un élément de référence $\mathcal{K}_r$. En utilisant un changement de variables adéquat, nous obtenons les contributions des autres éléments (voir le livre de Dhatt et Touzot, 1984, chapitre 1). Les méthodes d'intégration numériques, en particulier la méthode de Gauss, sont utilisées pour le calcul de ces intégrales.

Pour résumer, un programme utilisant la MEF est constituée de quatre étapes:

1. discrétisation (ou triangulation) du domaine en des éléments finis $\mathcal{K}_e$;
2. calcul des contributions A^e et F^e de chaque élément respectivement à la matrice A et au vecteur F;
3. assemblage de $A = \sum_e A^e$ et de $F = \sum_e F^e$;
4. résolution du système d'équations $Au = F$.

1.2.4 Traitement des conditions aux limites

Il existe principalement deux types purs de conditions aux limites (CL). La première est désignée par CL de type Dirichlet et la deuxième par CL de type Neumann:

1. **CL de type Dirichlet:** on fixe la valeur de u sur la frontière. Si $u = g$ sur Γ, alors on peut écrire:

$$u_h = u_h^D + u_h^{\neg D}$$

avec

$$u_h^{\neg D} = \sum_{n_j \notin \Gamma} u_j \varphi_j \text{ et } u_h^D = \sum_{n_j \in \Gamma} g(n_j)\varphi_j$$

En utilisant l'équation variationnelle, on a:

$$\int_\Omega \nabla v \cdot \nabla u_h^{\neg D} = \int_\Omega f\, v - \int_\Omega \nabla v \cdot \nabla u_h^D$$

Finalement, pour former le système d'équations, il suffit de remplacer v par les fonctions de base φ_i, n_i n'étant pas sur la frontière Γ. Ainsi, en séparant chaque u en deux vecteurs u_D et $u_{\neg D}$, on a le système d'équations suivant:

$$A_{(\neg D,\neg D)} u_{\neg D} = F_{\neg D} - A_{(\neg D,D)} u_D \tag{1.2.10}$$

où

- $A_{(\neg D,\neg D)}$ est la matrice carrée dont les indices des lignes et des colonnes correspondent aux noeuds qui n'appartiennent pas à la frontière.
- $A_{(\neg D,D)}$ est la matrice dont les indices des colonnes (respectivement des lignes) correspondent aux noeuds qui sont (respectivement ne sont pas) sur la frontière.
- $u_D = G_D = (g(n_i))$ où $n_i \in \Gamma$

Le terme $-A_{(\neg D,D)} u_D$ peut être interprété comme une nouvelle force qui s'ajoute à f.

2. **CL de type Neumann:** on fixe la valeur de la dérivée de u par rapport à la normale extérieure sur la frontière. Cette CL s'écrit sous la forme de:

$$\frac{\partial u}{\partial \mathbf{n}} = g \qquad \text{sur } \Gamma$$

En revenant à l'intégration par parties (1.2.6) (*i.e.*, formule de Green), on a:

$$\begin{aligned} \int_\Omega \nabla u \cdot \nabla v &= \int_\Omega f\, v + \int_\Gamma v \frac{\partial u}{\partial \mathbf{n}} \\ &= \int_\Omega f\, v + \int_\Gamma v\, g \end{aligned}$$

Le système d'équations devient alors:

$$Au = F + G_N \tag{1.2.11}$$

où le vecteur $G_N = (G_i)$ est défini par

$$G_i = \begin{cases} \int_\Gamma g\,\varphi_i & \text{si} \quad n_i \in \Gamma \\ 0 & \text{si} \quad n_i \notin \Gamma \end{cases}$$

Dans ce cas, la matrice A fait intervenir les indices de tous les noeuds, puisqu'il n'y a pas de condition de type Dirichlet. La matrice A est alors singulière. En effet, avec ces CL, on a fixé les valeurs des dérivées. Or toutes les fonctions à une constante additive satisfont les mêmes CL de type Neumann: il en existe alors une infinité. Pour remédier à cette lacune, on peut fixer une valeur de la fonction à chercher (par exemple nulle en un noeud) et résoudre le système d'équations pour obtenir une solution du problème.

Finalement, on peut aussi avoir des CL mixtes. Si $\Gamma = \Gamma_D \cup \Gamma_N$, $\Gamma_D \cap \Gamma_N = \emptyset$, avec une CL de type Dirichlet sur Γ_D et une autre de type Neumann sur Γ_N.

Dans la pratique, la règle générale est que le système d'équations final doit être de la forme:

$$A_{(\neg D,\neg D)} u_{\neg D} = F_{\neg D} - A_{(\neg D,D)} u_D + G_N$$

Seuls les indices des noeuds où la valeur de u n'est pas fixée sont pris en compte dans la matrice du système.

1.3 Algorithmes numériques

1.3.1 Les algorithmes itératifs linéaires

Étant donné qu'on est en présence de systèmes d'équations avec des milliers d'inconnues, il n'est pas souhaitable d'utiliser des méthodes directes telles que l'inversion de la matrice A ou sa factorisation sous forme de LU car ceci demandera un temps d'exécution assez important. Par contre, nous utilisons les méthodes itératives linéaires (MIL) qui, pour résoudre l'équation $Au = F$, partent d'une valeur initiale u^0 et, à chaque itération, produisent u^{k+1} qui dépend de u^k. Bien entendu, u^{k+1} doit s'approcher de la solution $\bar{u}$ du système.

Toutes les MIL fonctionnent selon le principe de la méthode de Newton, avec quelques variantes selon les caractéristiques de la matrice A. Le but est qu'à partir d'une valeur u^k, on aimerait trouver un "déplacement" w de telle façon que "idéalement" $u^{k+1} = u^k - w$ soit la solution du système. Dans ce cas, nous aurions:

$$\begin{aligned} Au^{k+1} &= F \\ A(u^k - w) &= F \end{aligned}$$

est donc de

$$\text{trouver } w \text{ tel que: } Aw = r^k \tag{1.3.1}$$

où $r^k = Au^k - F$. En essayant de résoudre l'équation (1.3.1), nous n'avons pas avancé. Par contre, si on choisit une matrice S adéquate, si possible proche de A, et facile à inverser, on peut alors remplacer cette équation par $Sw = Au^k - F$. L'algorithme itératif sera alors:

Algorithme 1.1 (Algorithme itératif linéaire matriciel)

(1) u^0 étant donné (sinon prendre $u^0 = 0$);
(2) **iterer**
(3) $r^k = Au^k - F$;
(4) **résoudre** $Sw^k = r^k$;
(5) $u^{k+1} = u^k - w^k$;
(6) **fin d'itérations**

Le choix de la matrice S, appelée **matrice pente**, fait la distinction entre les différentes méthodes proposées dans la littérature (Hageman et Young, 1981; Golub et van Loan, 1989).

Pour mieux situer le rôle de la matrice pente S, la figure 1.3 montre les étapes d'un algorithme itératif qui tente de résoudre l'équation Au = F dans le cas unidimensionnel.

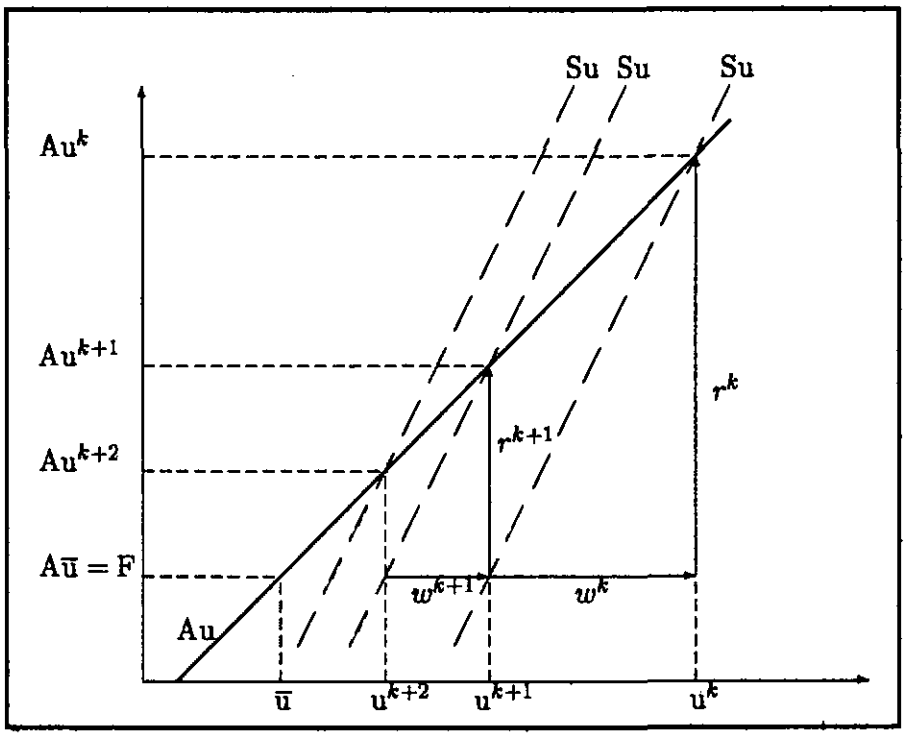

Figure 1.3 : rôle de la matrice pente dans le cas unidimensionnel.

Nous allons introduire des notations dont nous aurons besoin par la suite. Par rapport à une matrice A réelle de dimension N, nous notons:

- $\lambda_i(A)$, $i = 1, \ldots, N$, les valeurs propres de A avec $\lambda_{\min}(A)$ et $\lambda_{\max}(A)$ respectivement sa plus petite et sa plus grande valeur propre,
- $\kappa(A) = \dfrac{\max\{|\lambda_i(A)|,\ i = 1, \ldots, N\}}{\min\{|\lambda_i(A)|,\ i = 1, \ldots, N\}}$ le **nombre de condition**, parfois appelé **la condition**, de la matrice A. Pour raccourcir, nous dirons le "conditionnement" de la matrice.
- $\rho(A) = \max\{|\lambda_i(A)|,\ i = 1, \ldots, N\}$ le **rayon spectral** de la matrice A.

Si nous revenons à l'algorithme itératif ci-dessus, nous constatons que la mise à jour de u^{k+1} obéit à l'équation de récurrence:

$$u^{k+1} = (I - S^{-1}A)u^k - S^{-1}F \tag{1.3.2}$$

et que la solution $\bar{u}$ est un *point fixe* de cette équation (*i.e.*, si $u^k = \bar{u}$ alors $\forall p \geq k$, $u^p = \bar{u}$.). On peut montrer que l'algorithme itératif converge (atteint $\bar{u}$) si et seulement si nous avons $\rho(I - S^{-1}A) < 1$.

D'un autre côté, si on veut voir le rapprochement de u^{k+1} par rapport à $\bar{u}$, nous avons:

$$\begin{aligned} u^{k+1} - \bar{u} &= u^k - \bar{u} - S^{-1}(Au^k - F) \\ &= u^k - \bar{u} - S^{-1}A(u^k - \bar{u}) \\ u^{k+1} - \bar{u} &= (I - S^{-1}A)(u^k - \bar{u}) \end{aligned}$$

En d'autres termes, la vitesse de convergence dépend aussi de $\rho(I - S^{-1}A)$. Plus cette valeur est petite ($S \approx A$), plus rapidement u^k s'approche de la solution. Nous pouvons alors conclure que le choix d'une MIL, donc de la matrice S, doit tenir compte des trois points suivants:

1. **accessibilité:** l'équation $Sw^k = r^k$ doit être facile à résoudre;
2. **convergence:** il faut que $\rho(I - S^{-1}A) < 1$;
3. **rapidité:** $\rho(I - S^{-1}A)$ doit être aussi petit que possible. Nous verrons plus loin que dans le cas des méthodes des gradients, la vitesse de convergence est exprimée en fonction de $\kappa(S^{-1}A)$ qui doit être cette fois-ci proche de 1.

À titre d'exemple et dans le cadre des EDP, les méthodes les plus connues sont:

- Richardson: $S = \lambda_{max}(A)I$ où I est la matrice identité;
- Jacobi: $S = D$ où D est la diagonale de la matrice A;
- Gauss-Seidel: $S = D + L$ où L est la partie inférieure de la matrice A.

Par extension, ces techniques peuvent également être appliquées non pas à des éléments mais à des sous-matrices appelées blocs. On peut définir la diagonale D sous forme de blocs:

$$D = \begin{pmatrix} A_{1,1} & 0 & \dots & 0 \\ 0 & A_{2,2} & \dots & 0 \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & \dots & A_{J,J} \end{pmatrix} \text{ et } D^{-1} = \begin{pmatrix} A_{1,1}^{-1} & 0 & \dots & 0 \\ 0 & A_{2,2}^{-1} & \dots & 0 \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & \dots & A_{J,J}^{-1} \end{pmatrix}$$

si les sous-matrices $A_{i,i}$ ne sont pas singulières.

La mise à jour de u^{k+1} dans l'algorithme 1.1 (instr. 5) peut être remplacée par $u^{k+1} = u^k - \alpha_k w^k$ où α_k est un facteur d'amortissement qui peut jouer à la fois le rôle de stabilisateur et d'accélérateur de convergence de l'algorithme itératif (Hageman et Young, 1981).

1.3.2 L'insuffisance des MIL séquentielles

Dans le cas où la matrice A est SDP, on peut utiliser l'algorithme du gradient conjugué (GC)(Ashby *et al.*, 1990). Cet algorithme est devenu l'algorithme itératif de référence dans de nombreuses publications. En maintenant cette tradition, nous allons le prendre comme exemple pour étudier la portée, dans la pratique, de ces algorithmes.

Théoriquement, le GC fait au plus N_h (dimension de la matrice) itérations pour converger vers $\bar{\mathrm{u}}$. En réalité, on espère arrêter les itérations lorsque l'erreur $e^k = \mathrm{u}^k - \bar{\mathrm{u}}$ est améliorée en norme, de l'ordre de ε (*i.e.*, $\|e^k\| \leq \varepsilon\|e^0\|$). Dans la pratique, et comme nous n'avons pas la solution $\bar{\mathrm{u}}$, alors on remplace ce test d'arrêt par $\|r^k\| \leq \varepsilon\|r^0\|$ qui exprime la réduction du résidu: $r^k = \mathrm{A}e^k = \mathrm{A}\mathrm{u}^k - \mathrm{F}$.

Pour pouvoir faire une estimation du nombre d'itérations et du nombre d'opérations du GC, nous allons remplacer la norme $\|.\|$ par $\|.\|_\mathrm{A}$ avec $\forall \mathrm{v}, \|\mathrm{v}\|_\mathrm{A} = \mathrm{v} \cdot (\mathrm{A}\mathrm{v})$.

En utilisant l'algorithme du GC, nous avons (voir Quarteroni et Valli, 1994, chapitre 2): $\forall k > 0$,

$$\|e^k\|_\mathrm{A} \leq 2\left(\frac{\sqrt{\kappa(\mathrm{A})}-1}{\sqrt{\kappa(\mathrm{A})}+1}\right)^k \|e^0\|_\mathrm{A}$$

Si $k = \mathcal{I}(h)$ est le nombre d'itérations qu'il faut faire, en moyenne, pour avoir $\|e^k\|_\mathrm{A} \leq \varepsilon\|e^0\|_\mathrm{A}$, alors on a:

$$\mathcal{I}(h) \backsimeq \frac{\ln\left(\frac{\varepsilon}{2}\right)}{\ln(\mathcal{R}(h))}$$

où $\mathcal{R}(h) = (\sqrt{\kappa(\mathrm{A})}-1)/(\sqrt{\kappa(\mathrm{A})}+1)$.

Dans le cas d'un maillage régulier, nous savons que $\kappa(\mathrm{A}) \backsimeq h^{-2}$ (Fried, 1971). Dans la pratique, nous avons $h \ll 1$. Pour avoir un aperçu, nous prenons comme exemple un carré de 1×1 km^2 avec un noeud tous les mètres. Ici nous avons $h^{-2} = 10^6$ qui est le nombre de noeuds.

Dans ce cas nous avons:

$$\begin{aligned}
\ln\mathcal{R}(h) &= \ln(\sqrt{\kappa(\mathrm{A})}-1) - \ln(\sqrt{\kappa(\mathrm{A})}+1) \\
&= \ln(1-\kappa(\mathrm{A})^{-\frac{1}{2}}) - \ln(1+\kappa(\mathrm{A})^{-\frac{1}{2}}) \\
&\backsimeq -2\kappa(\mathrm{A})^{-\frac{1}{2}} \qquad (\ln(1+x) \backsimeq x,\ x \to 0)
\end{aligned}$$

Le nombre d'itérations est estimé à:

$$\mathcal{I}(h) \backsimeq \frac{\ln\left(\frac{2}{\varepsilon}\right)}{2}\sqrt{\kappa(\mathrm{A})} = \frac{\ln\left(\frac{2}{\varepsilon}\right)}{2h} \tag{1.3.3}$$

En prenant l'exemple du carré, si $\varepsilon = 10^{-4}$, comme $\ln(2 \times 10^4)/2 \backsimeq 5$, nous avons approximativement besoin de $\mathcal{I}(h) \backsimeq 5h^{-1} = 5 \times 10^3$ itérations.

En tenant compte que A est creuse, nous supposons qu'à chaque itération le GC fait environ h^{-2} opérations (h^{-2} étant la dimension de A).

Le nombre total d'opérations:

$$\mathcal{OP}(h) = \mathcal{I}(h) \times h^{-2} \backsimeq \frac{\ln\left(\frac{2}{\varepsilon}\right)}{2}h^{-3}$$

Dans notre exemple, $\mathcal{OP}(h) \backsimeq 5h^{-3} = 5 \times 10^9$. Même avec les machines les plus rapides (par exemple 1 GigaFlops c'est-à-dire 10^9 opérations par seconde), on a besoin d'au moins

5 secondes. En plus, dans le cas de la dynamique des fluides, on résout deux ou trois fois le même système. Dans ce cas, nous avons vite atteint les dizaines de secondes. Continuons dans nos estimations et supposons que nous ayons un domaine de 2×2 km^2 avec le même maillage (un noeud tous les mètres), alors au lieu de h nous aurons $h/2$ et donc

$$\mathcal{I}(h/2) = 2\mathcal{I}(h), \qquad \mathcal{OP}(h/2) = 8\mathcal{OP}(h) \tag{1.3.4}$$

Autrement, le nombre d'itérations est doublé et le nombre d'opérations est octuplé! Même avec cette approximation généreuse, nous sommes loin de pouvoir simuler ou, dans le cas de la météorologie prévoir, le comportement des systèmes dynamiques concrets où la largeur des domaines que nous étudions se mesurent par des centaines, voire des milliers, de kilomètres.

1.4 Conclusions

La section précédente nous montre à quel point le temps d'exécution des algorithmes numériques – dans les application concrète – est long. Pour réduire ce temps d'exécution, la collaboration entre informaticiens et numériciens joue un rôle très important. Chacun apporte sa contribution de la façon suivante :

- La contribution de l'informaticien se caractérise par la réduction du temps effectué à chaque itération. Pour cela, il a recourt à la parallélisation des algorithmes itératifs en faisant intervenir des machines à plusieurs processeurs ou des machines reliées par un réseau. Dans ce cas, nous parlerons de la parallélisation d'un algorithme itératif (voir la section 4.5) où il s'agit de répartitions des données.

- La contribution du numéricien se caractérise par la réduction du nombre d'itérations. Pour cela, le système $Au = F$ est remplacé par $C^{-1}Au = C^{-1}F$, où $\kappa(C^{-1}A) < \kappa(A)$. Ici, nous parlerons d'accélération d'un algorithme itératif (voir chapitre suivant). Contrairement à la parallélisation où il s'agit de garder le même système d'équations algébriques, dans l'accélération on cherche à remplacer la matrice de ce système afin d'obtenir une matrice avec un conditionnement meilleur.

Chapitre 2

Les méthodes de décomposition de domaine

2.1 Introduction

Bien que la parallélisation des algorithmes itératifs soit en mesure de réduire le temps d'exécution, dans les applications concrètes ce temps reste encore long. Des investissements dans le domaine d'analyse numérique ont été faits pour réduire ce temps et plus précisément le nombre d'itérations des algorithmes itératifs.

Dans le cas du gradient conjugué, nous avons vu au chapitre précédent que le nombre d'itérations dépend du conditionnement $\kappa(A)$ de la matrice A. De cette constatation, on voit apparaître des techniques appelées "pré-conditionnement" d'un système d'équations, qui consistent à remplacer le système de départ $Au = F$ par le système $(C^{-1}A)u = C^{-1}F$. La matrice C est choisie de telle manière qu'elle soit "simple" à inverser et que le matrice produit $C^{-1}A$ ait un conditionnement plus petit que celui de la matrice A. Dans ce cas, nous dirons que la matrice C est un **pré-conditionneur**, l'élaboration de cette matrice est un "pré-conditionnement" et enfin, l'algorithme itératif utilisé est un algorithme "pré-conditionné".

Dans le cadre des EDP, il existe deux niveaux d'intervention pour appliquer un tel pré-conditionneur:

Niveau matriciel. Après avoir discrétisé le problème et obtenu la matrice A, on cherche à former une matrice C à partir des valeurs de A. La factorisation LU incomplète et sa version modifiée sont les techniques les plus utilisées.

Niveau du problème. Ici, on essaye de transformer le problème de telle manière que la matrice obtenue après discrétisation ait un conditionnement meilleur. Ces techniques sont appelées – communément – par Méthodes de Décomposition de Domaine (MDD).

Dans la majorité des MDD, le problème aux EDP est transformé en un certain nombre de sous-problèmes. Ces sous-problèmes sont résolus au cours de chaque itération. Une MDD montre comment utiliser les solutions de ces sous-problèmes pour diminuer le nombre d'itérations d'un algorithme. Une MDD est plus intéressante si elle joue à la fois le rôle d'accélérateur et celui de "paralléliseur". En effet, si les sous-problèmes sont indépendants et s'ils peuvent être résolus en parallèle, on peut attribuer chaque sous-problème à un processeur.

Dans ce qui suit, nous allons distinguer trois types de MDD:

1. les méthodes avec recouvrement: il s'agit de subdiviser le domaine Ω en un certain nombre de sous-domaines et de restreindre le problème global à ces sous-domaines;

2. les méthodes à plusieurs niveaux: le maillage est transformé pour obtenir des sous-problèmes avec un maillage plus grossier;

3. les méthodes de réduction aux frontières: le problème de départ dans un domaine de dimension d est transformé en un problème de dimension $d-1$.

Nous allons donner une vue d'ensemble des méthodes les plus connues et non pas une étude théorique poussée. Notre but est de souligner à la fois leur diversité et la particularité des unes par rapport aux autres. Pour chacune de ces techniques, nous allons montrer comment le problème est transformé et dans le cas où il y aura plusieurs sous-problèmes, montrer comment les utiliser au cours d'un algorithme itératif. Pour chacune de ces MDD, notre démarche consistera à décrire la méthode et à exhiber la matrice C^{-1} en comparant les itérations de cette méthode à la formule générale de récurrence:

$$u^{k+1} = u^k - \alpha_k C^{-1}(Au^k - F) = u^k - \alpha_k C^{-1} A(u^k - \overline{u})$$

Pour écrire ce chapitre, nous nous sommes basés sur les articles de Dryja et Widlund (1990), de Xu (1992) et de Le Tallec (1994) qui constituent, à notre avis, une base pour une étude approfondie des MDD.

Sauf mention explicite, nous nous intéresserons aux problèmes de la forme:

$$\text{(2.1.1)} \qquad \begin{cases} -\Delta u = f & \text{dans } \Omega \\ u = 0 & \text{sur } \Gamma = \partial\Omega \end{cases}$$

Tous les résultats obtenus restent valables pour tout problème dont l'opérateur est symétrique, elliptique et linéaire (voir l'équation (1.2.3)).

2.2 Méthodes avec recouvrement

Nous supposons que le domaine Ω est subdivisé en deux sous-domaines Ω_1 et Ω_2 disjoints admettant une frontière commune $\Gamma_{1,2} = \overline{\Omega_1} \cap \overline{\Omega_2}$. À partir du sous-domaine Ω_1 (respectivement Ω_2) on forme un sous-domaine Ω_1^o (respectivement Ω_2^o) en tirant la frontière $\Gamma_{1,2}$ vers Ω_2 (respectivement Ω_1) de telle manière que les deux sous-domaines Ω_1^o et Ω_2^o se recouvrent et qu'ils aient au moins un élément fini en commun. Finalement, les frontières $\partial\Omega_1^o$ et $\partial\Omega_2^o$ ne doivent pas couper un élément fini.

2.2.1 Variante multiplicative

Cette méthode tend à résoudre le problème d'origine sur chacun des sous-domaines Ω_1^o et Ω_2^o. Chaque solution d'un sous-problème sert à ajuster les valeurs aux frontières de l'autre sous-problème.

Soit $u_j = u_{|\Omega_j^o}$ (respectivement f_j) la restriction de la fonction u (respectivement f) au sous-domaine Ω_j^o. La méthode est définie comme suit:
supposons avoir une valeur u^0 au départ, à l'itération $n \geq 1$, on résout successivement les

problèmes suivants:

$$\begin{cases} -\Delta u_1^{k+1} = f_1 & \text{dans } \Omega_1^\circ \\ u_1^{k+1} = u_2^k & \text{sur } \partial\Omega_1^\circ \\ u_1^{k+1} = 0 & \text{sur } \partial\Omega \end{cases} \quad \text{et} \quad \begin{cases} -\Delta u_2^{k+1} = f_2 & \text{dans } \Omega_2^\circ \\ u_2^{k+1} = u_1^{k+1} & \text{sur } \partial\Omega_2^\circ \\ u_2^{k+1} = 0 & \text{sur } \partial\Omega \end{cases}$$

Nous remarquons que les conditions aux limites aux frontières sont prises à partir des valeurs

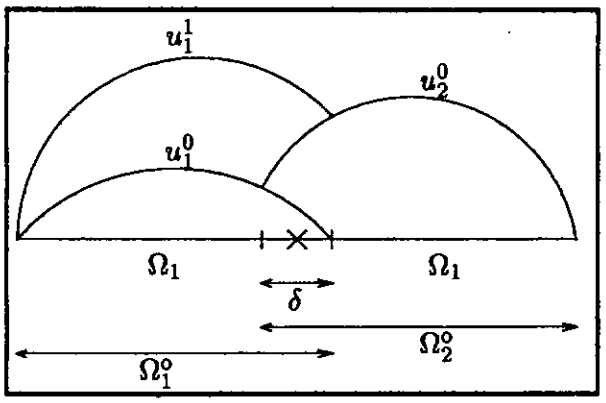

Figure 2.1 : méthode avec recouvrement: variante multiplicative.

de la solution obtenue dans l'autre sous-domaine (voir figure 2.1). C'est un "va-et-vient" entre les sous-problèmes.

Cette méthode est désignée par la méthode de Schwarz.

Pour montrer le rôle de cette décomposition par rapport à un algorithme itératif, on définit:

$$u^{k+\frac{1}{2}} = \begin{cases} u_1^{k+1} & \text{sur} \quad \Omega_1^\circ \\ u_2^k & \text{sur} \quad \Omega \setminus \Omega_1^\circ \end{cases} \qquad u^{k+1} = \begin{cases} u_1^{k+1} & \text{sur} \quad \Omega \setminus \Omega_2^\circ \\ u_2^{k+1} & \text{sur} \quad \Omega_2^\circ \end{cases}$$

En se basant sur les travaux de Lions (1988), on a au niveau matriciel:

$$\begin{cases} \mathrm{A}_1(\mathrm{u}^{k+\frac{1}{2}} - \mathrm{u}^k) &= \mathrm{R}_1(\mathrm{F} - \mathrm{A}\mathrm{u}^k) = \mathrm{R}_1\mathrm{A}(\overline{\mathrm{u}} - \mathrm{u}^k) \\ \mathrm{A}_2(\mathrm{u}^{k+1} - \mathrm{u}^{k+\frac{1}{2}}) &= \mathrm{R}_2(\mathrm{F} - \mathrm{A}\mathrm{u}^{k+\frac{1}{2}}) = \mathrm{R}_2\mathrm{A}(\overline{\mathrm{u}} - \mathrm{u}^{k+\frac{1}{2}}) \end{cases}$$

Pour $j = 1, 2$, A_j est la matrice obtenue après discrétisation du sous-problème appliqué au sous-domaine Ω_j° et R_j est la matrice de restriction permettant de sélectionner les indices des noeuds de Ω qui sont dans Ω_j°.

En combinant ces deux équations, on a l'équation de récurrence suivante:

$$\mathrm{u}^{k+1} = (\mathrm{I} - \mathrm{R}_2^T\mathrm{A}_2^{-1}\mathrm{R}_2)(\mathrm{I} - \mathrm{R}_1^T\mathrm{A}_1^{-1}\mathrm{R}_1)(\overline{\mathrm{u}} - \mathrm{u}^k)$$

Le pré-conditionneur obtenu par cette méthode est, dans le cas général,

$$\mathrm{C}^{-1} = (\mathrm{I} - \prod_{j=J}^{1}(\mathrm{I} - \mathrm{R}_j^T\mathrm{A}_j^{-1}R_j\mathrm{A}))\mathrm{A}^{-1}$$

où J est le nombre de sous-domaines utilisés.

Remarques pratiques:

- Il n'est pas nécessaire de calculer explicitement la matrice C, mais il faut suivre le procédé suivant:
  1. recevoir les valeurs aux frontières;
  2. résoudre le sous-problème local;
  3. transmettre les valeurs aux frontières de l'autre sous-domaine.
- Dans le cas où on a plusieurs sous-domaines, les sous-problèmes correspondants aux sous-domaines qui n'ont pas de frontières communes peuvent être résolus en parallèle. Ceci vient du fait que la solution d'un sous-problème ne dépend que des valeurs des solutions des sous-domaines adjacents.

À cause de la forme multiplicative du pré-conditionneur, cette méthode est désignée par la variante multiplicative de Schwarz.

2.2.2 Variante additive

La variante additive, proposée dans Matsokin et Nepomnyaschikh (1985) et approfondie dans Dryja et Widlund (1990, 1992), permet de résoudre tous les problèmes en parallèle sans attendre les solutions obtenues dans les sous-domaines voisins. Il suffit de prendre aux frontières les valeurs obtenues à l'itération précédente. Autrement dit, à chaque itération et dans chaque sous-domaine Ω_j^o, on résout le problème (voir figure 2.2):

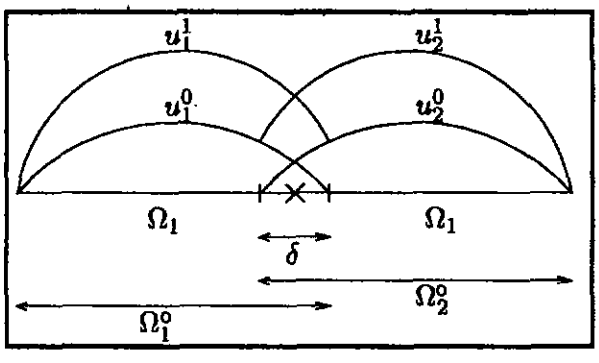

Figure 2.2 : méthode avec recouvrement: variante additive.

$$\text{(2.2.1)} \qquad \begin{cases} -\Delta u_j^{k+1} = f & \text{dans } \Omega_j^o \\ u_j^{k+1} = u_j^k & \text{sur } \partial\Omega_j^o \\ u_j^{k+1} = 0 & \text{sur } \partial\Omega_j^o \cap \partial\Omega \end{cases}$$

En posant

$$\text{(2.2.2)} \qquad \tilde{u}^k = u_1^k + u_2^k$$

on a dans la formulation matricielle:

$$\text{(2.2.3)} \qquad \tilde{u}^{k+1} = \tilde{u}^k - (R_1^T A_1^{-1} R_1 + R_2^T A_2^{-1} R_1) A(\tilde{u}^k - \bar{u})$$

La suite $\tilde{u}^k$ converge vers $\bar{u}$, solution du système $Au = F$. Dans ce cas, on a une méthode itérative avec un pré-conditionneur:

$$C^{-1} = \sum_{j=1}^{J} R_j^T A_j^{-1} R_j$$

Jusqu'ici on n'a pas une étude approfondie sur la comparaison, dans le sens de bon accélérateur, entre la variante additive et la variante multiplicative de Schwarz. Cependant, la variante additive possède deux caractéristiques intéressantes: la matrice $C^{-1}A$ est une SDP et les sous-problèmes peuvent être résolus en parallèle.

2.2.3 Méthodes proximales

Dans cette approche (voir Oualibouch et El Mansouri, 1997), on remplace les sous-problèmes par:

$$\begin{cases} \frac{1}{\lambda} u_j^{k+1} - \Delta u_j^{k+1} = f_j & \text{dans } \Omega_j^o \\ u_j^{k+1} = u_j^k & \text{sur } \partial\Omega_j^o \\ u_j^{k+1} = 0 & \text{sur } \partial\Omega_j^o \cap \partial\Omega \end{cases}$$

où λ est un réel strictement positif.

Si φ_i sont les fonctions de bases obtenues après discrétisation par des éléments finis, la matrice du système algébrique de ce sous-problème est définie par $\frac{1}{\lambda}M_j + A_j$, où M_j est la matrice de "masse" dont les éléments valent:

$$\int_{\Omega_j^o} \varphi_m \varphi_n$$

De la même manière que la variante additive de Schwarz, si $\tilde{u}^k = u_1^k + u_2^k$ alors l'algorithme effectué par cette décomposition est équivalent à:

$$\tilde{u}^{k+1} = \tilde{u}^k - (R_1^T(\frac{1}{\lambda}M_1 + A_1)^{-1}R_1 + R_2^T(\frac{1}{\lambda}M_2 + A_2)^{-1}R_1)A(\tilde{u}^k - \bar{u})$$

Dans le cas où on a J sous-domaines, le pré-conditionneur devient:

$$C_\lambda^{-1} = \sum_{j=1}^{J} R_j^T(\frac{1}{\lambda}M_j + A_j)^{-1}R_j$$

Nous pouvons remarquer que la variante additive de Schwarz est un cas particulier de la méthode proximale avec un paramètre λ choisi très grand: $\lambda \to \infty$.

2.2.4 Introduction d'un sous-domaine grossier

Généralement le conditionnement de $C^{-1}A$ se détériore quand le nombre de sous-domaines augmente. Pour remédier à cela, un nouveau "sous-domaine" appelé *sous-domaine grossier* est introduit. Ce domaine grossier est, en général, un domaine dont les sous-domaines jouent le rôle d'éléments finis. Dans l'exemple de la figure 2.3, les noeuds du sous-domaine grossier sont indiqués par $\otimes$. En introduisant ce nouveau problème, il est démontré que le conditionnement ne dépend que du rapport H/h, la racine carrée du nombre de noeuds par sous-domaine

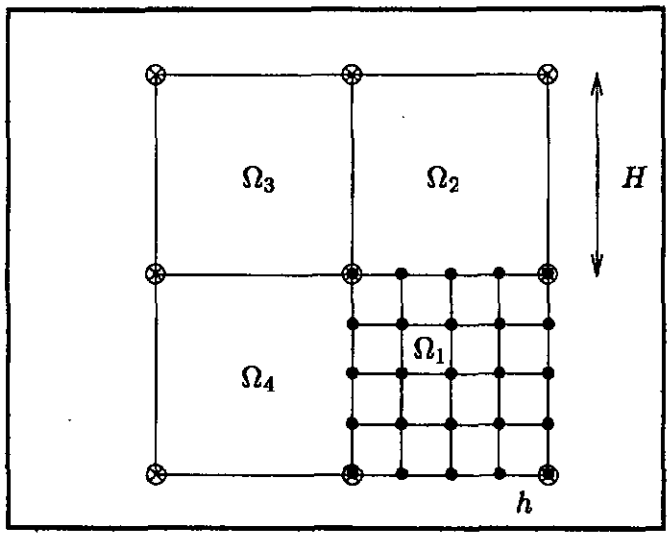

Figure 2.3 : exemple de subdivision en quatre sous-domaines.

(Bramble *et al.*, 1989; Smith, 1992; Mandel, 1993; Dryja et Widlund, 1994). Par conséquent, en maintenant le nombre de noeuds par sous-domaine constant, le nombre d'itérations reste invariant même si le nombre de noeuds ou le nombre de sous-domaines augmente. Cette constatation est vérifiée dans la pratique.

2.3 Méthodes à plusieurs niveaux

Les méthodes à plusieurs niveaux ont la particularité d'utiliser différents maillages appliqués au même domaine Ω. Chacun de ces maillages définit un nouvel espace fonctionnel dans lequel le problème d'origine sera résolu.

Dans le cas d'un maillage régulier, les maillages sont formés de la manière suivante. En partant du premier maillage $\mathcal{T}_h$, nous grossissons celui-ci, par exemple deux fois, en enlevant un noeud sur deux pour obtenir le maillage $\mathcal{T}_{2h}$ et ainsi de suite.

Dans le cas général, on procède de la façon suivante: supposons qu'on ait l niveaux de maillages; si nous prenons $h_l = h$ (*i.e.*, $\mathcal{T}_{h_l}$ est le niveau le plus fin), on prend pour $m = 1, 2, \ldots, l$, h_m de telle manière que:

$$\begin{cases} h_{m+1} < h_m, \\ h_m \text{ soit proportionnel à } \beta^{2m}; \end{cases}$$

avec $\beta < 1$ un réel quelconque qui ne dépend pas de m. Dans les exemples que nous allons montrer, nous avons $\beta = \sqrt{2}$. Nous notons $\mathcal{T}_{h_m}$ le maillage obtenu au niveau m et V_m^h l'espace fonctionnel correspondant à ce niveau.

2.3.1 Variante multigrille

Les méthodes itératives classiques peuvent être inefficaces même si la matrice A est proche de la matrice identité (Quarteroni et Valli, 1994, page 65). Le fait de joindre un nouveau problème appliqué sur le maillage grossier accélère sensiblement tout algorithme itératif appliqué au problème relatif au maillage d'origine. La figure 2.4 montre les fonctions de bases φ_i, dans le cas de la barre avec trois niveaux.

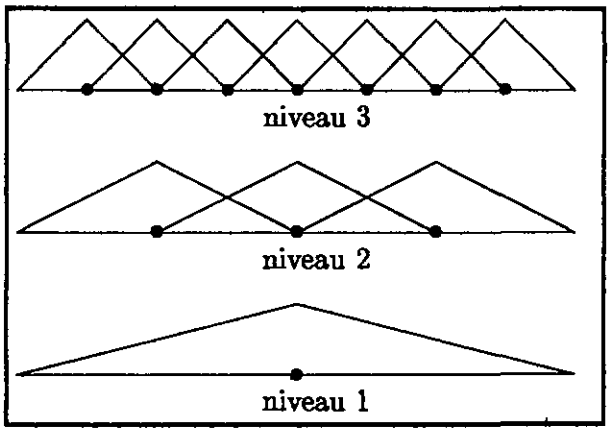

Figure 2.4 : génération des espaces par la méthode multigrille.

Soit A_m la matrice obtenue après discrétisation du problème au niveau m.

La méthode multigrille consiste à utiliser la matrice A_m comme un pré-conditionneur à un algorithme itératif appliqué au problème du niveau plus fin (*i.e.*, $A_{m+1}u_{m+1} = F_{m+1}$). Malheureusement, ce pré-conditionnement ne peut pas être utilisé à chaque itération, puisque la matrice A_m ne permet de corriger que certaines composantes du vecteur u_{m+1}. Il y a "perte d'informations" au cours du passage de V_{m+1}^h vers V_m^h. Pour remédier à cette lacune, la règle générale consiste à utiliser ce pré-conditlonnement toutes les $n_1 + p \times n_2$ itérations où n_1, n_2 et p sont des entiers vérifiant: $n_1 \geq 0$, $n_2 > 0$ et $p \geq 0$.

Dans le cas où nous avons deux niveaux m et $m-1$, et que l'algorithme itératif appliqué au niveau m est de la forme (voir section 1.3):

$$u_m \leftarrow u_m - S^{-1}(A_m u_m - F_m)$$

alors la méthode multigrille au niveau m est programmée comme suit:

Algorithme 2.1 (Multigrille à deux niveaux)

R_m^{m-1} *est la matrice de passage de* V_m^h *vers* V_{m-1}^h *et* $R_m^{m-1^T}$ *sa transposée.*

(1) **itérer** n_1 fois: $u_m \leftarrow u_m - S^{-1}(A_m u_m - F_m)$;
(2) **iterer**
(3) **calculer** le résidu: $r_m = A_m u_m - F_m$;
(4) **transférer** vers le niveau grossier: $r_{m-1} = R_m^{m-1} r_m$;
(5) **résoudre** dans le niveau grossier: $A_{m-1} w_{m-1} = r_{m-1}$;
(6) **revenir** au niveau plus fin: $w_m = R_m^{m-1^T} w_{m-1}$;
(7) **corriger**: $u_m \leftarrow u_m - w_m$
(8) **itérer** n_2 fois: $u_m \leftarrow u_m - S^{-1}(A_m u_m - F_m)$;
(9) **fin d'itérations**

Les n_1 (respectivement n_2) itérations sont appelées "pré-lissage" (respectivement "post-lissage").

Dans le cas général, on peut faire en sorte que la résolution du problème grossier (instruction (5)) se fasse en utilisant un algorithme itératif auquel on applique la méthode multigrille. La méthode multigrille à plusieurs niveaux peut être vue comme un pré-conditionnement récursif: chaque matrice A_{m-1} est utilisée comme pré-conditionneur de l'algorithme itératif appliqué au niveau m (Bramble, 1993; Xu, 1989).

2.3.2 Les bases hiérarchiques

Grâce aux travaux de Yserentant (1986) et plus tard de Bramble *et al.* (1990), des méthodes à plusieurs niveaux qui se prêtent bien à la parallélisation ont été proposées. Ces méthodes sont généralement désignées par "méthodes à bases hiérarchiques". Dans ces

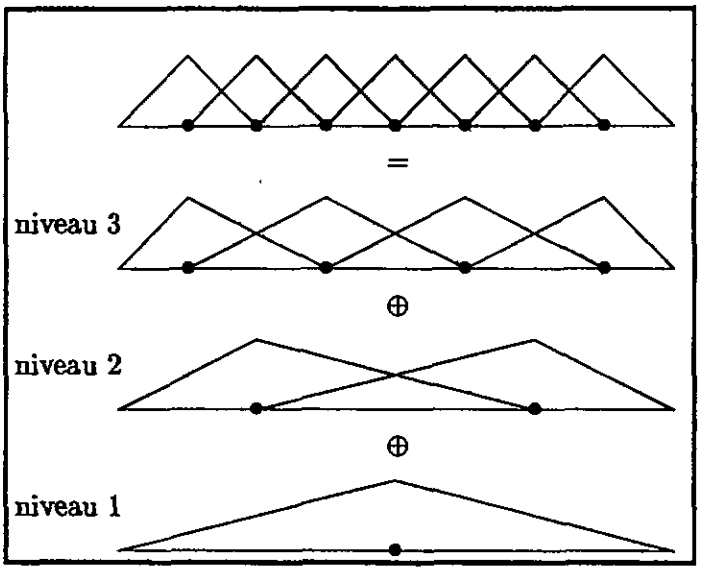

Figure 2.5 : génération des espaces par la méthode des bases hiérarchiques.

méthodes, on garde la même idée de grossissement du maillage, sauf qu'à chaque niveau m, on élimine du maillage $\mathcal{T}_{h_m}$ tous les noeuds qui se trouvent dans le maillage $\mathcal{T}_{h_{m-1}}$. De cette manière un noeud n'appartient qu'à un et un seul maillage (voir figure 2.5 dans le cas de la barre).

On forme alors des espaces fonctionnels $\widehat{V}_m^h = V_m^h \setminus V_{m-1}^h$ et de ce fait nous avons:

$$V^h = \widehat{V}_1^h \oplus \widehat{V}_2^h \oplus \cdots \oplus \widehat{V}_l^h \tag{2.3.1}$$

où $\oplus$ désigne la somme directe: toute fonction $u_h \in V^h$ est écrite d'une façon unique comme la somme des fonctions $u_m \in \widehat{V}_m^h$, $m = 1, \ldots, l$.

Le pré-conditionneur de cette approche ressemble à la variante additive de la méthode de Schwarz:

$$\mathrm{C}^{-1} = \sum_{m=1}^{l} \mathrm{R}_m{}^T \widehat{\mathrm{A}}_m^{-1} \mathrm{R}_m \tag{2.3.2}$$

où $\widehat{\mathrm{A}}_m$ est la matrice obtenue au niveau m.

On peut choisir d'autres formes de grossissement de maillage (Griebel et Thurner, 1993) dont les espaces fonctionnels générés V_m^h à chaque niveau m n'obéissent pas à la somme directe de l'équation (2.3.1).

2.4 Méthodes de réductions aux frontières

L'idée sous-jacente de ces méthodes est de ramener le problème de départ défini sur un domaine de dimension d à un autre problème défini sur un domaine dont la dimension est

$d-1$. Dans les méthodes sans recouvrement, il s'agit des frontières internes (*i.e.*, entre les sous-domaines) tandis que dans la technique d'enveloppement du domaine, c'est toute la frontière externe qui sera utilisée. Par la suite, un algorithme itératif appliqué au problème obtenu pourrait être pré-conditionné.

Nous tenons à signaler que les transformations d'un problème à celui sur les frontières conduisent à des formulations plus ou moins compliquées. Nous allons essayer de les introduire d'une façon grossière dans le but seulement de montrer l'intérêt de la démarche. Sans cela, il nous sera difficile de faire la relation entre le pré-conditionnement d'un système et ces méthodes.

2.4.1 Méthodes sans recouvrement

Il s'agit dans cette méthode de subdiviser le domaine Ω en un certain nombre de sous-domaines Ω_j et de résoudre le problème de départ indépendamment et en parallèle sur chacun des sous-domaines. L'objectif est de s'assurer que les valeurs de ces solutions locales sont à la fois cohérentes et correspondent aux restrictions de la solution du problème global.

Pour montrer l'approche, nous supposons que le domaine Ω est subdivisé en deux sous-domaines Ω_1 et Ω_2 qui ne se recouvrent pas et nous avons:

$\overline{\Omega}$: la fermeture du domaine Ω (*i.e.*, le domaine plus la frontière externe.);
$\Gamma_{1,2} = \overline{\Omega}_1 \cap \overline{\Omega}_2 \neq \emptyset$;
$\Gamma = \partial\Omega$;
$\Gamma_j = \Gamma \cap \overline{\Omega_j}$, $j = 1, 2$;
$\mathbf{n}_j$: le vecteur unitaire normal sortant du sous-domaine Ω_j.

Pour $j = 1$ ou 2, nous désignons par $f_j = f_{|\Omega_j}$ (respectivement $u_j = u_{|\Omega_j}$) la restriction de la fonction f (respectivement u) au sous-domaine Ω_j.

À cause de l'unicité de la solution, on sait que trouver $u \in H^1(\Omega)$ solution du problème (2.1.1) revient à trouver les fonctions u_1 et u_2 telles que, pour $j = 1, 2$, on a:

$$\left\{ \begin{array}{rcll} -\Delta u_j & = & f_j & \text{dans } \Omega_j \\ u_j & = & 0 & \text{sur } \Gamma_j \end{array} \right. \tag{2.4.1}$$

et qui respectent, sur la frontière $\Gamma_{1,2}$, les contraintes suivantes:

$$\left\{ \begin{array}{rcl} u_1 & = & u_2 \\ \dfrac{\partial u_1}{\partial \mathbf{n}_1} + \dfrac{\partial u_2}{\partial \mathbf{n}_2} & = & 0 \end{array} \right. \tag{2.4.2}$$

La première contrainte reflète la continuité de la fonction u et la deuxième, la continuité de ses dérivées normales.

Nous introduisons, pour toute fonction λ définie sur $\Gamma_{1,2}$, la fonction $\tilde{\lambda} \in H_0^1(\Omega)$ solution du problème de Dirichlet:

$$\left\{ \begin{array}{rcll} -\Delta\tilde{\lambda} & = & 0 & \text{dans } \Omega \\ \tilde{\lambda} & = & \lambda & \text{sur } \Gamma_{1,2} \\ \tilde{\lambda} & = & 0 & \text{sur } \Gamma \end{array} \right.$$

et nous définissons l'opérateur $\mathcal{D}$ par:

$$\mathcal{D}(\lambda) = \tilde{\lambda} \tag{2.4.3}$$

L'idée maîtresse de la méthode sans recouvrement consiste dans un premier temps, à se donner une fonction quelconque λ^0 sur la frontière $\Gamma_{1,2}$ (par exemple $\lambda^0 = 0$) et à trouver la fonction $u^{\lambda^0} = \mathcal{D}(\lambda^0)$. La solution $u^{\lambda^0} = (u_1^{\lambda^0}, u_2^{\lambda^0})$ respecte la première contrainte (continuité de u), mais pas forcement la deuxième. Il s'agit alors de maintenir cette première contrainte satisfaite tout en corrigeant la deuxième contrainte (continuité des dérivées normales). En d'autres termes, on doit trouver une fonction $\overline{\lambda}$ telle que $u^{\overline{\lambda}} = \mathcal{D}(\overline{\lambda})$ soit la correction qu'il faut apporter à la solution initiale u^{λ^0} pour que la fonction $\overline{u} = u^{\lambda^0} + u^{\overline{\lambda}}$ respecte la deuxième contrainte.

Ceci revient à résoudre le problème:
trouver $\overline{\lambda}$ *tel que*,

$$\frac{\partial \overline{u}}{\partial \mathbf{n}_1} + \frac{\partial \overline{u}}{\partial \mathbf{n}_2} = \frac{\partial u^{\overline{\lambda}}}{\partial \mathbf{n}_1} + \frac{\partial u^{\overline{\lambda}}}{\partial \mathbf{n}_2} + \frac{\partial u^{\lambda^0}}{\partial \mathbf{n}_1} + \frac{\partial u^{\lambda^0}}{\partial \mathbf{n}_2} = 0 \tag{2.4.4}$$

autrement dit, résoudre le problème

$$\mathcal{A}\lambda = \mathcal{F} \tag{2.4.5}$$

où $\mathcal{A}\lambda = \dfrac{\partial u^{\lambda}}{\partial \mathbf{n}_1} + \dfrac{\partial u^{\lambda}}{\partial \mathbf{n}_2}$ et $\mathcal{F} = -\left(\dfrac{\partial u^{\lambda^0}}{\partial \mathbf{n}_1} + \dfrac{\partial u^{\lambda^0}}{\partial \mathbf{n}_2}\right) = -\mathcal{A}\lambda^0$.

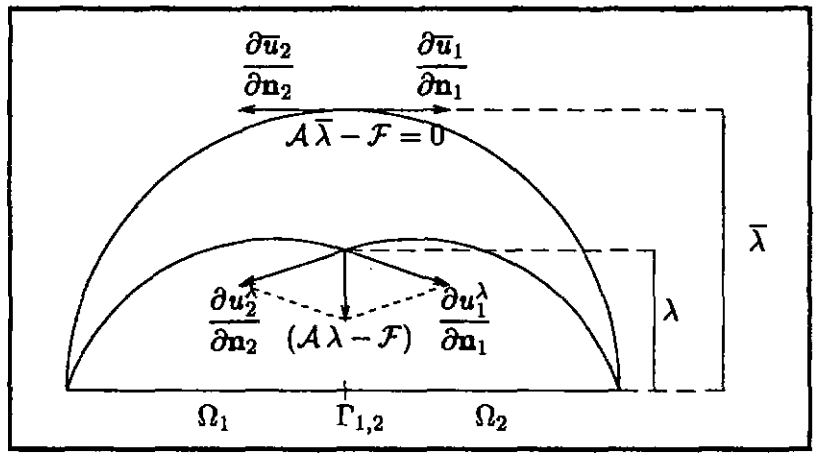

Figure 2.6 : exemple de la méthode sans recouvrement.

La figure 2.6 montre que, dans le cas d'une barre, $\mathcal{A}\lambda - \mathcal{F}$ représente la cassure de la solution $u^{\lambda} + u^{\lambda^0}$. L'objectif sera d'annuler (*i.e.*, enlever) cette cassure.

Au niveau fonctionnel, la formule générale de la mise à jour de λ est définie par:

$$\lambda^{k+1} = \lambda^k - \alpha_k(\mathcal{A}\lambda^k - \mathcal{F}) \tag{2.4.6}$$

où α_k est réel strictement positif.

Nous allons décrire les méthodes les plus utilisées pour la résolution du problème aux frontières (2.4.5):

Approche directe

Pour trouver la fonction λ^{k+1} de l'équation (2.4.6), on a recours à la formulation variationnelle de ce problème:

trouver λ^{k+1} *tel que, pour tout* μ, *nous avons:*

$$\int_{\Gamma_{1,2}} \mu\lambda^{k+1} = \int_{\Gamma_{1,2}} \mu\lambda^k - \alpha_k \int_{\Gamma_{1,2}} \mu(\mathcal{A}\lambda^k - \mathcal{F})$$

En posant $u^k = u^{\lambda^k} + u^{\lambda^0}$ et en utilisant la formule de Green, nous avons:

$$\begin{aligned}\int_{\Gamma_{1,2}} \mu(\mathcal{A}\lambda^k - \mathcal{F}) &= \int_{\Gamma_{1,2}} \mu(\frac{\partial u^k}{\partial \mathbf{n}_1} + \frac{\partial u^k}{\partial \mathbf{n}_2}) \\ &= \int_\Omega \nabla u^\mu \cdot \nabla u^k - \int_\Omega f\, u^\mu\end{aligned}$$

Donc si E est la matrice définie par $(\int_{\Gamma_{1,2}} \varphi_i\varphi_j)$ alors nous avons au niveau matriciel:

$$\begin{aligned}\mathrm{E}\lambda^{k+1} &= \mathrm{E}\lambda^k - \alpha_k(\mathrm{A}u^k - \mathrm{F}) \\ \text{ou} & \\ \lambda^{k+1} &= \lambda^k - \alpha_k \mathrm{E}^{-1}(\mathrm{A}u^k - \mathrm{F})\end{aligned}$$

Par conséquent,

$$\mathrm{u}^{k+1} = \mathrm{u}^k - \alpha_k \mathrm{D}\mathrm{E}^{-1}(\mathrm{A}u^k - \mathrm{F}) \tag{2.4.7}$$

où D est la matrice de l'opérateur $\mathcal{D}$ défini dans (2.4.3).

Approche avec enveloppement de la frontière

En utilisant la formule de Green (1.2.6), on peut constater que pour tout μ nous avons:

$$\int_{\Gamma_{1,2}} \mu\,\mathcal{A}\lambda = \int_{\Gamma_{1,2}} \mu(\frac{\partial u^\lambda}{\partial \mathbf{n}_1} + \frac{\partial u^\lambda}{\partial \mathbf{n}_2}) = \int_W \nabla u_W^\mu \cdot \nabla u^\lambda$$

où W est un sous-domaine de Ω contenant la frontière $\Gamma_{1,2}$ (voir figure 2.7) et $u_W^\mu \in H_0^1(W)$

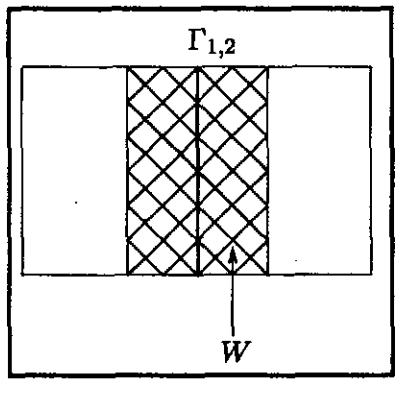

Figure 2.7 : enveloppement de la frontière par W.

solution de l'équation:

$$\begin{cases} -\Delta u = 0 & \text{dans} \quad W \\ u = \mu & \text{sur} \quad \Gamma_{1,2} \\ u = 0 & \text{sur} \quad \partial W \end{cases}$$

Nous pouvons alors remplacer E par la matrice A_W de ce problème. Ce qui revient à:

$$u^{k+1} = u^k - \alpha_k D A_W^{-1}(Au^k - F)$$

Cette équation de récurrence est un cas particulier de la variante multiplicative de Schwarz. En effet, avec A_W^{-1} nous résolvons un problème sur W et avec D nous résolvons le même problème sur Ω_1 et Ω_2. Ces trois sous-domaines forment une subdivision avec recouvrement du domaine Ω.

Pré-conditionnement du problème aux frontières

- L'opérateur $\mathcal{A}$ s'écrit comme la somme de deux opérateurs $\mathcal{A}_j$, pour tout j, définis par:

$$\mathcal{A}_j \lambda = \frac{\partial u_j^\lambda}{\partial n_j}$$

Cet opérateur admet un inverse $\mathcal{A}_j^{-1}$ défini par:

$$\mathcal{A}_j^{-1} \mu = \widehat{u}_j^\mu | \Gamma_{1,2}$$

la restriction aux frontières de la fonction $\widehat{u}_j^\mu$, solution du problème:

$$(2.4.8) \qquad \begin{cases} -\Delta \widehat{u}_j^\mu = 0 & \text{dans} \quad \Omega_j \\ \dfrac{\partial \widehat{u}_j^\mu}{\partial n_j} = \mu & \text{sur} \quad \Gamma_{1,2} \\ \widehat{u}_j^\mu = 0 & \text{sur} \quad \Gamma_j \end{cases}$$

Le système défini par l'équation $\mathcal{A}\lambda = \mathcal{F}$, peut alors être pré-conditionné par la matrice de l'opérateur:

$$C^{-1} = \sum_j \mathcal{A}_j^{-1}$$

Ceci revient à résoudre en plus des problèmes locaux, un deuxième problème local (2.4.8) de type Neumann si $\Gamma_j = \emptyset$ et mixte si $\Gamma_j \neq \emptyset$. Ce pré-conditionneur appelé Neumann-Neumann a été introduit par de Roeck et Le Tallec (1991). D'autres pré-conditionneurs ont été traités dans Bjørstad et Widlund (1986); Börgers (1989); Marini et Quarteroni (1989).

Si B_j est la matrice obtenue après discrétisation du problème (2.4.8), nous avons:

$$u^{k+1} = u^k - \alpha_k (\sum_j B_j^{-1}) D E^{-1} (Au^k - F)$$

- Dans le cas où nous avons plus de deux sous-domaines, il s'agit d'appliquer la variante additive de Schwarz (section 2.2.2) pour résoudre le problème aux frontières. Dans ce cas, les frontières sont subdivisées en des sous-domaines avec recouvrement. Cette approche a été introduite par Bramble *et al.* (1989). Nous trouvons aussi une variante de Smith (1992) qui diffère par le type des sous-problèmes à résoudre dans chaque sous-domaine de la frontière.

Conclusions

Ce que nous pouvons retenir de ces démarches:

- la subdivision qui est sans recouvrement;
- l'aspect naturellement parallèle de la démarche;
- la satisfaction des contraintes aux frontières revient à résoudre un système (linéaire) sur les frontières. Ce système peut être pré-conditionné.

Dans notre démarche, nous avons maintenu la contrainte $u_1 = u_2$ tout en essayant d'annuler la somme des dérivées normales. La méthode "duale" qui maintient la somme des dérivées normales nulle tout en essayant d'annuler $u_1 - u_2$ sur les frontières pourrait aussi être utilisée (voir Dinh *et al.*, 1984, pour plus de détails).

2.4.2 Enveloppement de domaine

Il est connu qu'un algorithme itératif converge plus vite si le domaine est régulier (*i.e.*, rectangle, carré, cube etc.). En plus si le maillage est aussi simple et régulier, nous pouvons trouver des algorithmes directs ou itératifs qui ont la caractéristique d'être rapides et simples à programmer. Dans la collection d'articles de Schumann (1978), on peut trouver différentes études et applications concernant ces algorithmes.

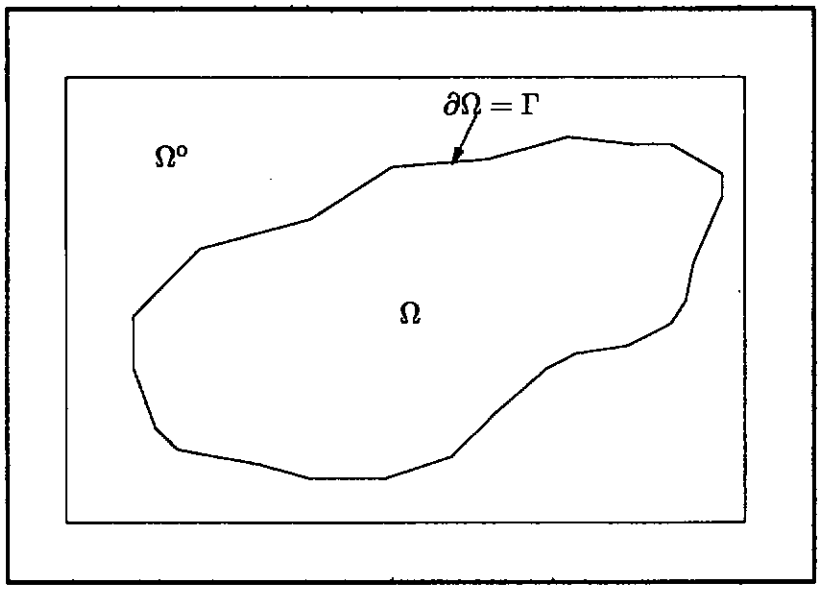

Figure 2.8 : enveloppement d'un domaine en deux dimensions.

La méthode d'enveloppement du domaine (Börgers, 1990; Dinh *et al.*, 1992; Nepomnyaschikh, 1992) tire profit de la rapidité de ces algorithmes et tente de résoudre le problème des EDP, non pas dans le domaine Ω, mais dans un domaine régulier Ω° plus grand qui contient Ω (voir la figure 2.8). Pour qu'une solution $\tilde{u}$ dans Ω° soit un simple prolongement de la solution $\overline{u}$ du problème de départ, il faut que $\tilde{u}$ satisfasse les contraintes imposées sur $\partial\Omega$.

Soient Ω° un domaine régulier tel que $\overline{\Omega} \subset \Omega^\circ$ et $\tilde{f}$ sur $L^2(\Omega^\circ)$ un prolongement quelconque de f (nous pouvons prendre par exemple $\tilde{f}$ nulle sur $\Omega^\circ \setminus \Omega$).

Dans cette méthode le problème de départ défini par:

$$\begin{cases} -\Delta u &= f & \text{dans } \Omega \\ u &= g & \text{sur } \partial\Omega \end{cases} \tag{2.4.9}$$

est transformé en:
trouver $\tilde{u} \in H^1(\Omega^\circ)$ *tel que*

$$\begin{cases} -\Delta \tilde{u} &= \tilde{f} & \text{dans } \Omega^\circ \\ \tilde{u} &= 0 & \text{sur } \partial\Omega^\circ \end{cases} \tag{2.4.10}$$

Le but est de faire en sorte que $\tilde{u} = g$ sur $\partial\Omega$. Dans ce cas, et à cause de l'unicité, la solution du problème (2.4.9) sera la restriction de $\tilde{u}$ à Ω;

Pour bien situer l'approche, nous allons nous baser sur ce que nous avons vu dans la méthode de décomposition sans recouvrement. Nous avons vu comment à partir d'une solution initiale, nous avons pu corriger la cassure aux bords des frontières pour trouver la solution du problème de départ. Dans cette section, nous allons faire le chemin inverse: nous supposons avoir la fonction $\tilde{u}^0$ solution du problème (2.4.10). En partant de cette solution, nous allons "tordre" cette fonction en exerçant une "force" λ sur $\partial\Omega$. Le but est d'obtenir une fonction $\tilde{u}$ qui satisfasse la contrainte $\tilde{u} = g$ sur $\partial\Omega$ (voir figure 2.9).

Exercer une force λ en $\partial\Omega$ sur la solution de $\tilde{u}^0$ revient à trouver $\tilde{u}^\lambda$ dans $H_0^1(\Omega^\circ)$, solution du problème variationnel:

$$\int_{\Omega^\circ} \nabla\tilde{u}^\lambda \cdot \nabla v = \int_{\partial\Omega} \lambda v, \; \forall v \in H_0^1(\Omega^\circ) \tag{2.4.11}$$

et ensuite d'ajouter $\tilde{u}^\lambda$ à $\tilde{u}^0$.

Pour arriver à la solution du problème (2.4.9), nous devons trouver la force adéquate $\overline{\lambda}$ telle que la fonction $\tilde{u}^{\overline{\lambda}} + \tilde{u}^0$ ait la valeur g sur $\partial\Omega$. Ce qui revient à trouver $\overline{\lambda}$ pour que:

$$\tilde{u}^{\overline{\lambda}} = g - \tilde{u}^0 \qquad \text{sur } \partial\Omega \tag{2.4.12}$$

Ceci nous suggère de définir l'opérateur $\mathcal{A}$ comme suit:

$$\mathcal{A}\lambda = \tilde{u}^\lambda_{|\partial\Omega} \tag{2.4.13}$$

la restriction de $\tilde{u}^\lambda$ sur $\partial\Omega$, où $\tilde{u}^\lambda$ est la solution du problème (2.4.11) (voir l'exemple de la barre dans la figure 2.9).

Le problème aux frontières est formulé de la manière suivante:
trouver $\overline{\lambda}$, *solution du problème*

$$\mathcal{A}\lambda = \mathcal{F} \tag{2.4.14}$$

où $\mathcal{F} = g - \tilde{u}^0_{|\partial\Omega}$

Ce problème est un problème "dual" par rapport au problème de la réduction aux frontières internes de deux sous-domaines. En effet, Ω° peut être considéré comme un domaine subdivisé en deux sous-domaines sans recouvrement: $\Omega_1^\circ = \Omega$ et $\Omega_2^\circ = \Omega^\circ \setminus \overline{\Omega}$. De cette constatation, nous pouvons appliquer par analogie tous les types d'approches de pré-conditionnement vus dans la section précédente.

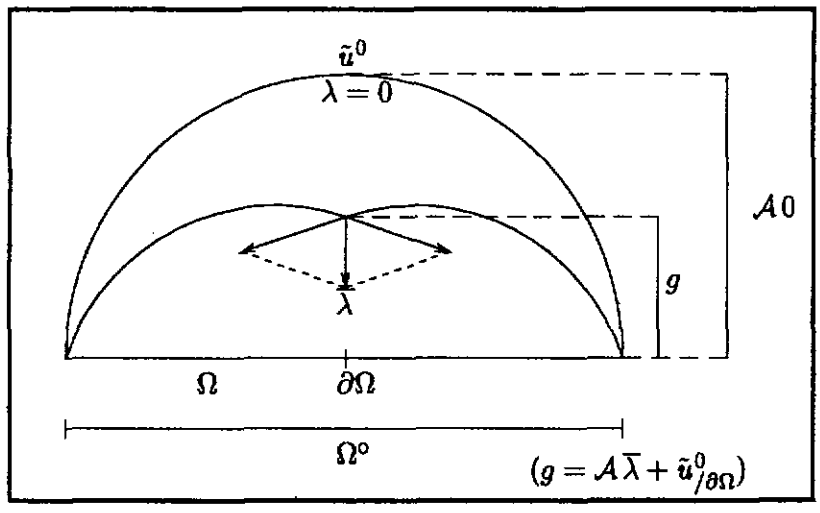

Figure 2.9 : exemple du prolongement du problème en une dimension.

2.5 Conclusions

L'intérêt des MDD a été largement prouvé par la multitude des expériences faites au cours de ces dernières années.

Volontairement, nous avons omis de parler de l'implantation des MDD. Ceci dans le but de soulever l'intérêt "théorique" de ces méthodes.

Dans la pratique et pour des applications concrètes, généralement, on s'intéresse plus à la parallélisation des algorithmes itératifs qu'à l'introduction des pré-conditionneurs. Car si la parallélisation "optimale" est définie comme étant une répartition des données optimale (*i.e.*, équilibrage des charges et minimisation du temps du communication entre processeurs), l'utilisation optimale des méthodes de décomposition n'est connue que pour des domaines simples (Shao, 1993).

Comme pour toute méthode, une utilisation maladroite d'une MDD peut entraîner un ralentissement plutôt qu'une accélération.

On sait que ces MDD joueront un rôle important dans les applications à calculs massivement parallèles où le gain en nombre d'itérations est plus que souhaitable. Malheureusement, lorsqu'on veut utiliser ces MDD dans un environnement parallèle, on est – tout de suite – confronté à des questions comme:

- Quelle est la relation entre le nombre de sous-problèmes et le nombre de processeurs pour une utilisation parallèle optimale?
- Parmi toutes les MDD, quelle est la méthode la plus adéquate pour un problème donné?
- Quelle est la subdivision (respectivement le grossissement) optimale d'un domaine (respectivement d'un maillage) donné?
- etc.

À notre avis, pour pouvoir répondre à ces questions aujourd'hui, il nous faut faire un travail d'ingénierie: expérimenter et ensuite modéliser l'utilisation optimale des MDD. Pour cela, il faut penser à mettre en place un environnement qui jouera le rôle d'un "laboratoire d'expériences".

Dans les chapitres suivants, nous allons montrer notre contribution à ces objectifs.

Chapitre 3

Proposition d'unification des méthodes de décomposition de domaine

3.1 Introduction

Les MDD jouent un rôle important dans la simulation numérique, lorsqu'on est en présence d'applications qui requièrent un nombre important de données. Ce cas est coustaté dans toutes les applications concrètes de la simulation en dynamique des fluides et dans la résolution des EDP en général. Mettre en place un programme de simulation fait appel à plusieurs techniques telles que la construction du maillage, les algorithmes numériques, la gestion de la mémoire, la répartition des données, les algorithmes de communication entre processeurs, etc. Par rapport à chacune de ces disciplines, on voit apparaître des librairies informatiques qui tendent à épargner au modélisateur des détails de la programmation. Cela lui permet de se concentrer uniquement sur la résolution des problèmes en testant des méthodes connues ou nouvelles. À titre d'exemple, nous avons LAPACK, CLAPACK et SCALAPACK pour les opérations algébriques, les librairies PVM et MPI pour les communications, METIS pour le découpage de domaine et bien d'autres librairies que nous pouvons trouver dans le domaine public[1] .

Malheureusement, à notre connaissance, les librairies qui permettent d'exploiter la puissance des MDD sont presque inexistantes. Cela est dû principalement au fait que le choix de l'utilisation d'une MDD dépend fortement du type de problème que l'on veut résoudre. En effet, le choix d'une MDD doit tenir compte à la fois de la structure géométrique du domaine, du type d'éléments finis, du type des EDP, etc.

On peut voir ces MDD comme des techniques d'accélération au niveau matriciel. Il suffit donc d'insérer la contribution d'une matrice de pré-conditionnement dans les procédures qui résolvent les systèmes d'équations algébriques. Des librairies, comme Block95 suivent cette approche. Elles permettent d'implanter des pré-conditionneurs de type Jacobi ou de type Gauss-Seidel par blocs. Autrement, fournir une librairie qui permette d'exploiter les différentes MDD est un procédé très difficile sinon impossible. Il suffit d'imaginer, au regard du chapitre précédent, le nombre de paramètres dont il faut tenir compte.

[1] NETLIB: http://www.netlib.org/

Pour pouvoir exploiter les MDD, il serait intéressant de laisser le modélisateur subdiviser, grossir ou raffiner le maillage et redéfinir de nouveaux problèmes relativement à chaque région du domaine. C'est lui seul, par ses expériences et même son intuition, qui saura comment adapter puis exploiter les MDD pour résoudre un problème de simulation.

À notre avis, la programmation des MDD nous a montré la faiblesse de l'approche qui consiste à produire des librairies informatiques accompagnées de longs manuels d'utilisation. Les MDD ne peuvent pas être préprogrammées, mais doivent être adaptées à un problème précis.

Les MDD nous fournissent un outil puissant: "accélération par transformation du problème". On doit trouver un moyen de définir ces transformations au niveau de la spécification des problèmes et non au niveau matriciel, afin que le modélisateur puisse s'en servir à son gré. Il serait intéressant de conjuguer les efforts faits dans d'autres disciplines pour produire un environnement qui exploite la puissance des MDD. Un tel environnement ne peut être justifié que s'il permet au modélisateur de spécifier et de combiner ces différentes techniques depuis la définition d'un problème jusqu'au choix de la machine où le programme doit être exécuté. C'est au sein de cet environnement que nous allons proposer une programmation des MDD.

Pour atteindre cet objectif, nous devons au moins satisfaire les règles suivantes:

1. Adaptabilité: on doit pouvoir spécifier une MDD quelque soit le problème à résoudre.

2. Simplicité: l'effort que doit fournir le modélisateur pour spécifier une MDD doit être moindre par rapport à la programmation de cette méthode.

3. Extensibilité: on doit pouvoir rajouter de nouvelles techniques d'accélération.

4. Haut niveau de spécification: la spécification d'une MDD doit être faite au niveau des problèmes. Elle est indépendante de la machine et du langage de programmation à utiliser.

Par conséquent, on doit dépasser le stade des environnements qui assistent à la parallélisation. Il nous faut un environnement qui "parle" le langage d'un modélisateur et "raisonne" comme un numéricien. Ici, l'informaticien doit faire la connexion entre les trois acteurs principaux de la modélisation numérique à savoir le modélisateur, le numéricien et la machine (voir la figure 3.1).

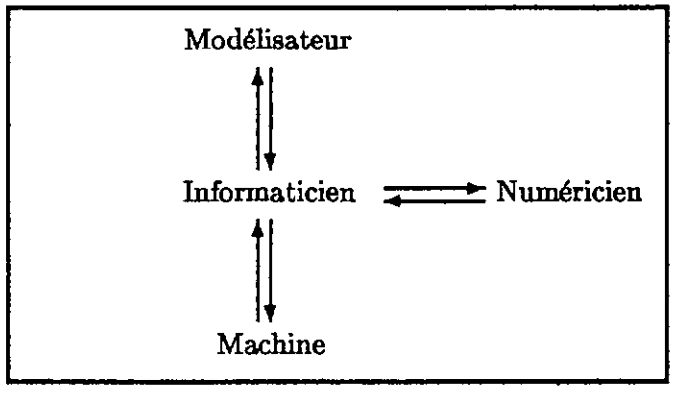

Figure 3.1 : la position de l'informaticien lors de la simulation numérique.

Notre contribution à cet ultime objectif est faite essentiellement au niveau de la programmation des MDD. Pour cela, les MDD sont définies comme un assemblage des composantes génériques qui sont, dans notre cas, des problèmes aux EDP. La transformation d'un problème crée un nouveau problème et donc une nouvelle composante. Cet assemblage de problèmes définit une MDD qui est spécifique au problème de départ.

En effet, à partir de la définition d'un problème supposé complexe, nous procédons à des transformations. Ces transformations génèrent un ou plusieurs nouveaux problèmes. Nous distinguons deux types de transformations:

1. **Transformation pour accélération:** un ou plusieurs nouveaux problèmes servent à accélérer la résolution du problème initial.

2. **Transformation par réduction:** on ramène un problème original à un ou plusieurs problèmes plus simples, en général de dimension plus petite (*i.e.*, le nombre de noeuds est moindre).

Après chaque transformation, nous établissons un lien, appelé **coordination**, entre le problème original et le ou les nouveaux problèmes. Ce lien permet de séparer les rôles que jouent ces derniers (accélération ou réduction) de la manière par laquelle ils sont résolus. Ainsi et indépendamment du rôle joué, ces nouveaux problèmes peuvent à leur tour être transformés. Par une succession de transformations, nous obtenons un arbre de dépendance dont les noeuds sont les problèmes et les arcs sont des coordinations entre problèmes.

Avec ce concept, le modélisateur peut spécifier avec un langage simple non seulement les MDD les plus connues, mais en créer de nouvelles.

Dans ce qui suit, nous allons définir un cadre de travail qui sera la base pour la programmation de notre environnement. Pour cela, nous allons donner une interprétation des rôles que jouent les MDD. Cette interprétation facilitera la mise en place de l'environnement que nous avons développé et qui sera étudié dans le chapitre suivant.

3.2 Transformations pour accélération

Nous allons nous intéresser dans cette section aux problèmes $\mathcal{P}(V, W, \mathcal{F})$ définis par la résolution de l'équation:

$$\mathcal{F}(x) = g, \qquad g \in W$$

où V et W sont deux espaces fonctionnels et $\mathcal{F}$ une application de V dans W.

Pour simplifier notre argumentation, nous supposons que $\mathcal{F}$ est linéaire et bijective. Nous notons $\overline{x} = \mathcal{F}^{-1}(g)$ la solution de ce problème.

En outre, nous supposons que ce problème est résolu par des algorithmes itératifs que nous pouvons écrire sous de la forme:

Algorithme 3.1 (Algorithme Itératif général)
étant donné une application $\mathcal{D} : W \longrightarrow V$.

(1) **iterer**
(2) $r^k = \mathcal{F}(x^k) - g$;
(3) $\delta^k = \mathcal{D}(r^k)$;
(4) $x^{k+1} = x^k - \delta^k$;
(5) **fin d'itérations**

La valeur x^0 est supposée connue. Sinon, nous la considérons nulle.

Intuitivement, dans l'instruction (2) l'équation permet de calculer dans W le déplacement qu'il faut apporter à $\mathcal{F}(x^k)$ pour atteindre g. L'application $\mathcal{D}$ nous donne l'image de cette correction dans V, à savoir, $\delta^k = \mathcal{D}(r^k)$. Enfin, dans l'instruction (4), nous calculons x^{k+1} en tenant compte de cette correction. Dans le cas matriciel, l'application $\mathcal{D}$ peut être représentée par la matrice pente S introduite dans la section 1.3.

Comme dans le cas de tous les algorithmes itératifs linéaires, en général, le meilleur choix de l'application $\mathcal{D}$ est $\mathcal{F}^{-1}$. Dans ce cas l'algorithme itératif converge en une seule itération.

Notre approche consiste à remplacer l'application $\mathcal{D}$ par la résolution d'un nouveau problème que nous appellerons, dans ce cas, "problème auxiliaire pour accélération" (PAA). Un problème $\mathcal{P}_a(V_a, W_a, \mathcal{F}_a)$ est un PAA au problème $\mathcal{P}(V, W, \mathcal{F})$ si:

1. il existe une suite entière croissante $\Pi = \{m_1, \ldots, m_n, \ldots\}$;
2. il existe deux applications $\mathcal{Q} : V_a \longrightarrow V$ et $\mathcal{R} : W \longrightarrow W_a$;

telles que si on remplace à l'itération $m_k \in \Pi$ de l'algorithme 3.1 l'application $\mathcal{D}$ par $\mathcal{Q} \circ \mathcal{F}_a^{-1} \circ \mathcal{R}$, le nouvel algorithme converge néanmoins vers la solution du problème $\mathcal{P}(V, W, \mathcal{F})$.

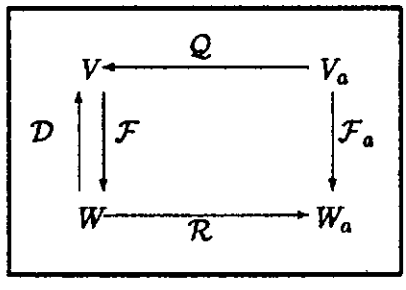

Figure 3.2 : le problème auxiliaire.

Dans ce cas, nous avons:

$$x^{k+1} = x^k - \mathcal{D}_k(\mathcal{F}(x^k) - g)$$

où

$$\mathcal{D}_k = \begin{cases} \mathcal{Q} \circ \mathcal{F}_a^{-1} \circ \mathcal{R} & \text{si} \quad k \in \Pi \\ \mathcal{D} & \text{sinon} \end{cases}$$

Le nouvel algorithme devient:

Algorithme 3.2 (Algorithme Itératif modifié)
étant donné une application $\mathcal{D} : W \longrightarrow V$.

(1) iterer
(2) $r^k = \mathcal{F}(x^k) - g$
(3) si $(k \in \Pi)$ alors
(4) transférer la direction r^k dans W_a: $r_a^k = \mathcal{R}(r^k)$;
(5) trouver la direction δ_a^k dans V_a: résoudre $\mathcal{F}_a(\delta_a^k) = r_a^k$;
(6) transférer la direction δ_a^k dans V: $\delta^k = \mathcal{Q}(\delta_a^k)$;
(7) sinon

(8) $\delta^k = \mathcal{D}(r^k)$;
(10) $x^{k+1} = x^k - \delta^k$;
(11) **fin d'itérations**

Comme nous pouvons le constater, le problème auxiliaire permet au problème original de trouver la direction en donnant une approximation du déplacement à prendre. En choisissant un problème auxiliaire adéquat, on peut arriver à la solution $\overline{x}$ plus rapidement et donc diminuer le nombre d'itérations de l'algorithme. Si tel est le cas, le problème auxiliaire joue le rôle d'un pré-conditionneur.

La notion de problème auxiliaire permet de formuler des pré-conditionneurs cette fois-ci au niveau des problèmes en approchant le problème original par un autre problème.

Dans la pratique, nous créons un problème auxiliaire par transformation du problème de départ. Ici, la notion de transformation est prise au sens large. Pour que cette transformation soit utile, il faut qu'elle constitue une simplification du problème. Nous distinguons deux cas:

1. **Transformation spatiale.** En gardant la même fonction $\mathcal{F}_a = \mathcal{F}$, on peut changer les ensembles V et W. Il s'agit de modifier la géométrie du domaine Ω ou le maillage, par exemple en modifiant le type d'éléments finis ou en grossissant le maillage, etc.

   L'exemple le plus utilisé – sans doute parce qu'il est le plus intuitif – est la programmation à deux niveaux, où il s'agit de prendre $V_a \subset V$. La fonction $\mathcal{F}_a$ est la restriction de $\mathcal{F}$ sur V_a et $W_a = \mathcal{F}(V_a)$ est l'image de V_a. Dans ce cas, $\mathcal{R}$ sera la projection de W dans W_a et $\mathcal{Q}$ la prolongation de V_a dans V. La correction dans l'espace V_a ne permet que d'ajuster certaines composantes de x^k. C'est pour cela que nous ne pouvons l'utiliser que dans une itération sur deux ou plus encore. Les expériences faites dans le domaine d'optimisation en général ont montré leurs effets en tant qu'accélérateurs (voir Karypis et Kumar, 1995, pour le découpage des domaines).

   La méthode multigrille (voir section 2.3.1) à deux niveaux rentre dans cette définition. En effet l'espace $V_a \subset V$ est construit par exemple, en grossissant le maillage. La suite $\Pi = \{n_1 + i \times n_2, i \geq 0\}$ où n_1 et n_2 sont les nombres d'itérations respectivement pour le pré-lissage et le post-lissage (voir l'algorithme 2.1).

2. **Transformation fonctionnelle.** En gardant les mêmes ensembles, c'est-à-dire en prenant $V_a = V$ et $W_a = W$, on approche la fonction $\mathcal{F}$ par une fonction $\mathcal{F}_a$ simple à inverser. Dans le cas des EDP, il s'agit de modifier l'opérateur (voir section 1.3 et l'article de Xu (1992)). En utilisant ce problème auxiliaire à chaque itération, l'algorithme résultant correspond à celui de Richardson (voir section 1.3 et l'article de Xu (1992)). De la même manière, nous pouvons définir les algorithmes de Jacobi ou de Gauss-Seidel.

Après chaque transformation, le problème auxiliaire devient lui-même un nouveau problème susceptible d'être résolu par un algorithme itératif. Cet algorithme itératif peut aussi être pré-conditionné en créant un nouveau problème auxiliaire et ainsi de suite. Ceci nous donne une relation en chaîne entre les problèmes. Nous appelons cette relation "coordination".

La **coordination pour accélération** (CA) entre un problème et son problème auxiliaire est définie par $< \mathcal{Q}, \mathcal{R}, \Pi >$ où $\mathcal{Q}$ (respectivement $\mathcal{R}$) est l'application qui permet le passage de V_a vers V (respectivement de W vers W_a) et Π la suite qui détermine la fréquence de l'utilisation du problème auxiliaire (voir ci-dessus).

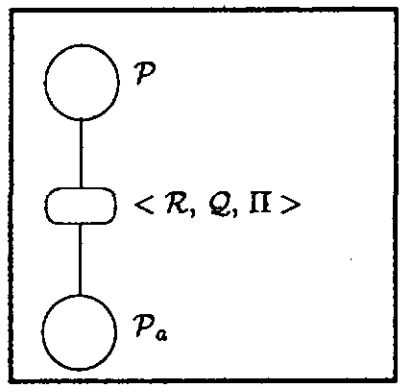

Figure 3.3 : coordination pour accélération.

Nous dirons qu'une CA est **forte** si à chaque itération de l'algorithme, nous faisons appel au problème auxiliaire. Dans ce cas, la suite $\Pi = \{0, 1, \dots\}$.

Dans la suite, sauf mention du contraire, nous supposons que les CA sont fortes. Dans ce cas, nous omettons l'écriture de la suite et la coordination sera désignée par $< \mathcal{R}, \mathcal{Q} >$.

Jusqu'ici, nous avons supposé que ces transformations ne génèrent qu'un seul problème auxiliaire. Dans le cas où une transformation donne naissance à plus d'un problème (par exemple subdivision du domaine), nous parlons de CA "multiple". Nous distinguons deux approches:

Soit $\mathcal{P}_j(V_j, W_j, \mathcal{F}_j)$, $j = 1, \dots, J$, les problèmes auxiliaires ainsi créés.

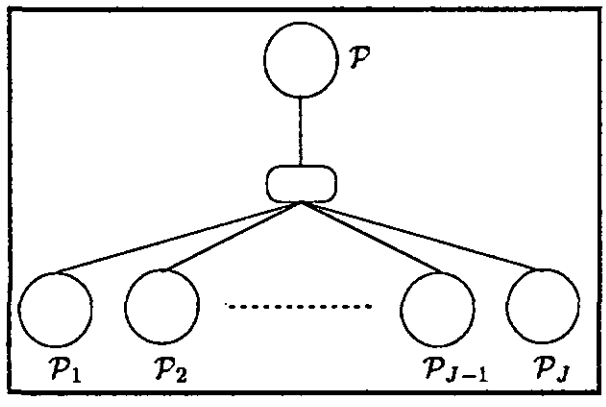

Figure 3.4 : coordination pour accélération multiple.

- **CA multiple successive** (séquentielle). Nous définissons pour chaque problème $\mathcal{P}_j$, les deux applications $\mathcal{Q}_j : V_j \longrightarrow V$ et $\mathcal{R}_j : W \longrightarrow W_j$, les applications de transfert de variables entre le problème $\mathcal{P}$ et le problème $\mathcal{P}_j$. Le but est de faire corriger la direction δ^k successivement par chaque problème $\mathcal{P}_j$, $j = 1, \dots, J$. Dans ce cas, à partir de la valeur x^k, et pour obtenir δ^k, nous définissons des variables intermédiaires $\delta^{k-1+j/J}$, $r^{k+(j-1)/J}$ et $x^{k+j/J}$, $\forall j = 1, \dots, J$, qui vérifient les égalités suivantes:

$$\begin{aligned} \delta^{k-1+\frac{j}{J}} &= \mathcal{Q}_j(\mathcal{F}_j^{-1}(\mathcal{R}_j(r^{k+\frac{j-1}{J}}))); \\ x^{k+\frac{j}{J}} &= x^{k-1+\frac{j}{J}} - \delta^{k-1+\frac{j}{J}}; \end{aligned}$$

avec $r^{n+\frac{j-1}{J}} = \mathcal{F}(x^{k+\frac{j-1}{J}}) - g$

Cette coordination est une abstraction de la variante multiplicative de Schwarz dans le cas des MDD. Lorsqu'on utilise une décomposition spatiale, on peut définir V comme le produit cartésien des espaces V_j. Ici, une variable x est écrite sous forme d'un vecteur $(x_1, \ldots, x_J)$, avec $x_j \in V_j$. Chaque problème $\mathcal{P}_j$ rapporte une correction à la composante x_j.

- **CA multiple simultanée** (parallèle). Dans le cas successif, les contributions de chaque problème sont prises indépendamment l'une de l'autre. Le problème transformé fait appel successivement à un problème auxiliaire à la fois. Dans le cas simultané, tous les problèmes auxiliaires vont participer en même temps à la recherche du nouveau déplacement. Pour cela, nous supposons avoir deux applications de transfert:

$$\begin{array}{rccl} \mathcal{R} & : & W & \longrightarrow & W_1 \times \cdots \times W_J \\ & & r & \longrightarrow & (\mathcal{R}_1(r), \ldots, \mathcal{R}_J(r)) \end{array}$$

et

$$\mathcal{Q} : V_1 \times \cdots \times V_J \longrightarrow V$$

Le déplacement est alors calculé de la manière suivante:

$$\delta^k = \mathcal{Q}(\mathcal{F}_1^{-1}(\mathcal{R}_1(r^k)), \ldots, \mathcal{F}_J^{-1}(\mathcal{R}_J(r^k)))$$

Ici, l'application $\mathcal{Q}$ détermine la manière de combiner les sous-déplacements générés par chaque problème $\mathcal{P}_j$ pour retrouver le déplacement dans V. On utilise souvent une combinaison additive où

$$\mathcal{Q}(x_1, \ldots, x_J) = \sum_{j=1}^{J} \mathcal{Q}_j(x_j)$$

avec $\mathcal{Q}_j$ une application définie de V_j dans V.

Dans le cas de la décomposition spatiale, on génère J sous-espaces fonctionnels $V_j \subset V$ $(j = 1, \ldots, J)$ de telle manière qu'on ait:

$$V = V_1 + V_2 + \cdots + V_J$$

(*i.e.*, $\forall\, u \in V; \exists (u_1, \ldots, u_J) \in V_1 \times V_2 \times \cdots \times V_J$ tels que $u = u_1 + u_2 + \cdots + u_J$.)

Cette technique est utilisée par la méthode additive de Schwarz (section 2.2.2), la méthode proximale (section 2.2.3) et les méthodes à bases hiérarchiques (section 2.3.2).

3.3 Transformations par réduction

Dans cette section, nous abordons un autre type de transformation du problème, où il ne s'agit pas d'accélérer un algorithme itératif, mais plutôt de transformer le problème $\mathcal{P}$ en un problème $\widetilde{\mathcal{P}}$ simple à résoudre. Pour retrouver la solution du problème de départ, le problème $\widetilde{\mathcal{P}}$ est muni d'un ensemble de contraintes.

Pour cela, nous supposons que $\mathcal{P}$ est défini sous forme de minimisation d'une fonction:

$$\text{trouver } \bar{u} = \underset{u \in V}{\operatorname{argmin}}\, \mathcal{F}(u)$$

Ce problème est transformé en un problème $\widetilde{\mathcal{P}}$:

$$\text{trouver } \overline{\widetilde{u}} = \underset{\widetilde{u}\in\widetilde{V}}{\operatorname{argmin}} \widetilde{\mathcal{F}}(\widetilde{u}) \tag{3.3.1}$$

Une solution $\overline{\widetilde{u}}$ du problème $\widetilde{\mathcal{P}}$ est aussi solution de $\mathcal{P}$ si et seulement si elle satisfait les contraintes:

$$\mathcal{B}(\overline{\widetilde{u}}) = c$$

Pour être concret, nous allons revenir au problème des EDP elliptiques et linéaires. Nous savons qu'après la discrétisation, nous obtenons le système d'équation $Au = F$ où A est une matrice SDP d'ordre N. La solution de ce système est aussi la solution du problème de minimisation:

$$\overline{u} = \underset{u\in\mathbb{R}^N}{\operatorname{argmin}} \left(\frac{1}{2}u\cdot(Au) - F\cdot u\right)$$

L'idée sous-jacente de cette transformation est de ramener l'équation $Au = F$ à une équation $\widetilde{A}\widetilde{u} = \widetilde{F}$ (de dimension $\widetilde{N}$) qui serait moins difficile à résoudre.

Pour cela, nous définissions une matrice R comme matrice de passage de $\mathbb{R}^N$ dans $\mathbb{R}^{\widetilde{N}}$. La matrice R^T, transposée de R, est alors la matrice de passage de $\mathbb{R}^{\widetilde{N}}$ dans $\mathbb{R}^N$.

En multipliant les deux membres de l'équation $Au = F$ par R, nous obtenons $RAu = RF$. Si nous posons $u = R^T\widetilde{u}$, nous avons l'équation:

$$\widetilde{A}\widetilde{u} = \widetilde{F} \tag{3.3.2}$$

où $\widetilde{A} = RAR^T$ et $\widetilde{F} = RF$. Si R est de rang $\widetilde{N}$, on constate que la matrice $\widetilde{A}$ est aussi une matrice SDP. Ce nouveau problème peut alors être transformé en un problème du minimisation:

$$\overline{\widetilde{u}} = \underset{u\in\mathbb{R}^{\widetilde{N}}}{\operatorname{argmin}} \left(\frac{1}{2}\widetilde{u}\cdot(\widetilde{A}\widetilde{u}) - \widetilde{F}\cdot\widetilde{u}\right) \tag{3.3.3}$$

Malheureusement dans la pratique, lorsqu'on procède à une simplification du problème, on a une "perte d'information". La solution du nouveau problème de minimisation (3.3.3) ne nous permet pas de retrouver la solution du problème de départ. Ceci vient du fait que la matrice R n'est généralement pas inversible. Pour remédier à cette lacune, on rajoute à ce nouveau problème un ensemble de contraintes qu'on peut écrire sous la forme:

$$B\widetilde{u} = c \tag{3.3.4}$$

B étant une matrice de dimension $M\times\widetilde{N}$ avec $\widetilde{N} > M > 0$, M étant le nombre de contraintes et c un vecteur dans $\mathbb{R}^M$.

Avec les contraintes adéquates, on peut dire que si $\overline{\widetilde{u}}$ est solution du problème de minimisation (3.3.3) avec les contraintes (3.3.4), alors $u = R^T\overline{\widetilde{u}}$ est la solution du problème de départ. D'un autre côté, en supposant que le rang de B vaut M, la solution $\overline{\widetilde{u}}$ de ce problème avec contraintes est caractérisée par l'existence de $\lambda\in\mathbb{R}^M$ tel que:

$$\begin{cases} \widetilde{A}\overline{\widetilde{u}} + B^T\lambda = \widetilde{F} \\ B\overline{\widetilde{u}} = c \end{cases} \tag{3.3.5}$$

En effet, $(\bar{\tilde{u}}, \lambda)$ représente le point-selle du *Lagrangien* associé au problème de minimisation (3.3.3) avec les contraintes (3.3.4). Ce lagrangien est défini par

$$\mathcal{L}(\tilde{u}, \mu) = \left(\frac{1}{2}\tilde{u} \cdot (\tilde{A}\tilde{u}) - \tilde{F} \cdot \tilde{u} + B^T\mu\right) . \tag{3.3.6}$$

pour tout $\tilde{u} \in \mathbb{R}^{\tilde{N}}$ et $\mu \in \mathbb{R}^M$; et son point-selle est caractérisé par

$$\mathcal{L}(\bar{\tilde{u}}, \lambda) = \min_{\tilde{u}} \max_{\mu} \mathcal{L}(\tilde{u}, \mu)$$

Par le fait que $\tilde{A}$ est une SDP et que B est de rang M, nous savons que $(\bar{\tilde{u}}, \lambda)$ est l'unique point-selle et que $\bar{\tilde{u}}$ est l'unique solution du problème avec contraintes. Les équations (3.3.5) sont obtenues en dérivant ce Lagrangien par rapport à $\tilde{u}$ et ensuite par rapport à μ (voir le livre de Fortin et Glowinski, 1983, pour plus de détails).

Comme nous avons supposé que la matrice $\tilde{A}$ est plus facile à inverser, nous allons éliminer $\bar{\tilde{u}}$ de ces équations. Nous obtenons alors un nouveau problème $\mathcal{P}_r$ où il s'agit de trouver λ solution de l'équation:

$$A_r\lambda = s_r \tag{3.3.7}$$

où $A_r = B\tilde{A}^{-1}B^T$ et $s_r = B\tilde{A}^{-1}\tilde{F} - c$.

En choisissant $M \ll N$, le problème de départ de dimension N est réduit à un problème d'ordre M. Pour résoudre le problème $\mathcal{P}_r$, nous n'avons pas besoin de calculer explicitement la matrice A_r. Dans ce cas, on tient compte du fait qu'on a, pour tout λ^k:

$$\begin{aligned} A_r\lambda^k - s_r &= B\tilde{A}^{-1}B^T\lambda^k - B\tilde{A}^{-1}\tilde{F} + c \\ &= B\tilde{A}^{-1}(B^T\lambda^k - \tilde{F}) + c \end{aligned}$$

$$A_r\lambda^k - s_r = -(B\tilde{u}^k - c) \tag{3.3.8}$$

où $\tilde{u}^k$ est solution de l'équation

$$\tilde{A}\tilde{u}^k = \tilde{F} - B^T\lambda^k$$

L'algorithme itératif appliqué à $\mathcal{P}_r$ est de la forme:

(1) λ^0 étant donné ou nulle;
(3) **iterer**
(4) **résoudre** $\tilde{A}\tilde{u}^k = \tilde{F} - B^T\lambda^k$;
(5) $\lambda^{k+1} = \lambda^k + \alpha_k(B\tilde{u}^k - c)$
(6) **fin d'itérations**

Cette réduction suppose l'existence et le moindre coût de calcul de la solution de l'équation:

$$\tilde{A}\tilde{u} = \tilde{F} - B^T\lambda$$

pour tout $\lambda \in \mathbb{R}^M$.

Les méthodes de réductions aux frontières peuvent être interprétées comme des méthodes utilisant cette transformation:

- **Méthodes sans recouvrement** (section 2.4.1). Après avoir subdivisé le domaine Ω en des sous-domaines Ω_j, il s'agit de s'assurer de la contrainte $\frac{\partial \widetilde{u}}{\partial \mathbf{n}_i} + \frac{\partial \widetilde{u}}{\partial \mathbf{n}_j} = 0$ sur la frontière $\Gamma_{i,j}$ entre les sous-domaines Ω_i et Ω_j. Ici λ représente les valeurs aux frontières que devaient avoir la fonction $\widetilde{u}$ sur les frontières internes, $\cup_{i,j}\Gamma_{i,j}$.
- **Méthodes d'enveloppement de domaine** (section 2.4.2). En partant d'un domaine Ω, il s'agit de résoudre les mêmes EDP sur un domaine $\widetilde{\Omega}$ ($\overline{\Omega} \subset \widetilde{\Omega}$) avec la contrainte $\widetilde{u}_{|\partial\Omega} = g$. Ici, λ représente la force qu'il faut exercer à la frontière $\partial\Omega$.

Cette transformation du problème, que nous appelons d'emblée **réduction** donne naissance à deux nouveaux problèmes auxilaires que nous appelons **problèmes auxilaires par réduction**: le problème $\widetilde{\mathcal{P}}$ (3.3.3) que nous supposons moins difficile à résoudre et le problème $\mathcal{P}_r$ (3.3.7) dont la dimension est très petite par rapport au problème de départ (voir la figure 3.5).

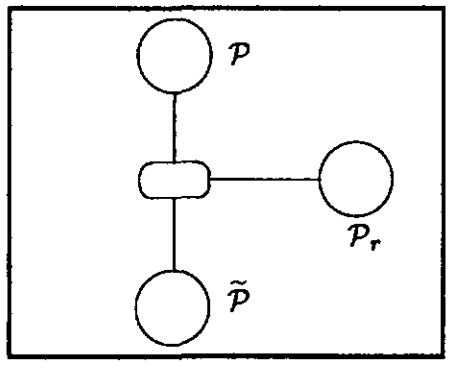

Figure 3.5 : coordination pour réduction du problème.

La coordination pour réduction (CR) issue de cette transformation s'établit principalement entre $\mathcal{P}_r$ et $\widetilde{\mathcal{P}}$ pour le calcul du résidu (3.3.8).

Les exemples donnés sont basés sur des transformations spatiales (*i.e.*, réduire aux frontières). Nous pouvons aussi appliquer des transformations fonctionnelles dans le cas où $A = LL^T$. Ici, la résolution de $Au = F$ se transforme en la résolution de deux problèmes:

$$Lp = F \quad \text{et } L^T u = p$$

autrement,

$$p - L^T u = 0 \quad \text{et } Lp = F$$

Ceci ressemble à l'équation (3.3.5). Au niveau fonctionnel, nous pouvons définir l'opérateur $\Delta u = \nabla^T \nabla u$. Cette technique basée sur une réduction fonctionnelle est utilisée dans les méthodes dites "mixtes" (voir Raviart et Thomas, 1977; Brezzi et Fortin, 1991).

3.4 Conclusions

Avec le concept défini ci-dessus que nous appelons un "cadre de travail", nous avons établi un langage à disposition du modélisateur. Ce langage utilise des mots comme "subdiviser", "grossir le maillage", "envelopper un domaine", etc. Ici, le modélisateur peut être un "homme de terrain", qui fera appel à ses expériences antérieures, ou à son intuition d'ingénieur pour

choisir les transformations adéquates ou tester de nouvelles transformations et ainsi accumuler d'autres expériences. Il peut être aussi un numéricien qui teste de nouvelles approches d'accélérations d'une classe de problèmes.

Pour programmer les MDD, ce cadre de travail nous permet d'atteindre au moins trois objectifs importants:

1. **Haut niveau de spécification.** Les MDD peuvent être spécifiées au niveau problème sans avoir de connaissances approfondies dans le domaine de l'analyse numérique. Nous laissons le modélisateur, comme un chef d'orchestre, composer les transformations qu'il pense être adéquates au problème traité.

2. **Simplicité.** Les MDD ne sont pas définies comme une seule action sur un problème, mais comme une succession de transformations simples appliquées à un problème.

3. **Diversification.** Les transformations permettent non seulement de spécifier les MDD les plus connues, mais aussi d'en créer de nouvelles qui seront adaptées aux problèmes à résoudre.

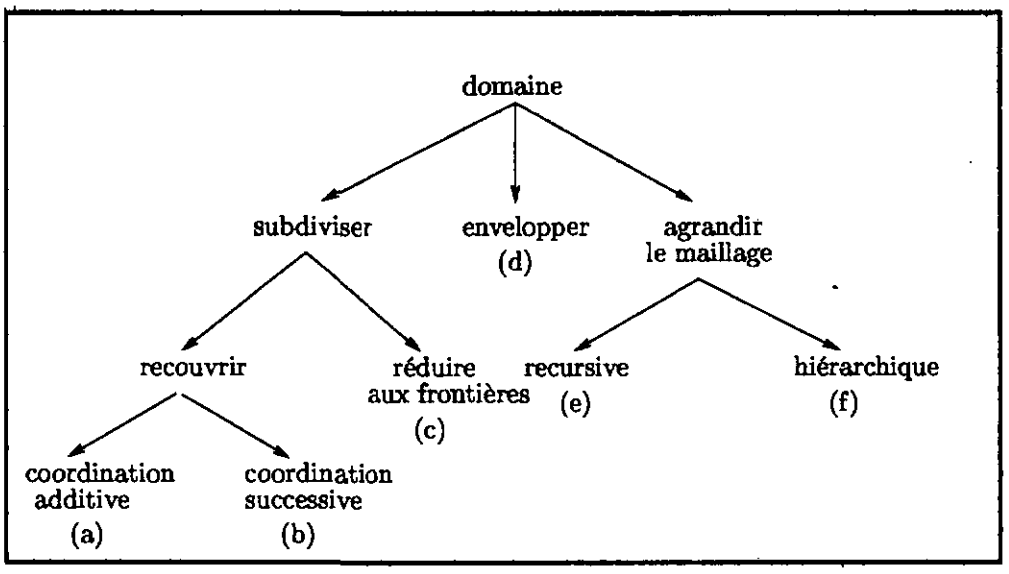

Figure 3.6 : quelques transformations spatiales

La figure 3.6 nous montre quelques transformations spatiales (*i.e.*, géométriques) mises à disposition du modélisateur. Suivant le choix et le déroulement de ces transformations, le modélisateur aboutit aux MDD:

(a) variante additive de Schwarz ou méthodes proximales si le modélisateur change l'opérateur local;

(b) variante multiplicative de Schwarz;

(c) méthodes sans recouvrement;

(d) méthodes d'enveloppement du domaine;

(e) méthodes multigrille;

(f) méthodes à bases hiérarchiques.

Finalement, grâce à ce cadre, nous pouvons répondre à trois questions importantes concernant une MDD:

1. son rôle: simplifier un problème ou accélérer un algorithme itératif appliqué à un problème;
2. sa spécification: ensemble de coordinations entre problèmes;
3. sa programmation: programmation des problèmes coopérants.

Chapitre 4

L'environnement de développement

4.1 Introduction

L'environnement de développement que nous proposons est un système de conception naturel dans son utilisation qui permet la programmation des MDD. Cet environnement est adressé aux modélisateurs. Il permet d'éviter à ces derniers les difficultés inhérentes à la programmation de ces méthodes numériques. Ainsi, il permet aux modélisateurs de se consacrer entièrement à leur travail. Nous attendons de cet environnement qu'il soit un "laboratoire" d'expériences pour les modélisateurs.

Cet environnement collecte les informations concernant les définitions des problèmes, les transformations à appliquer à ces problèmes et le type d'algorithmes numériques à utiliser. Après quoi, il produit un programme écrit dans un langage structuré, dans notre cas le langage C (voir figure 4.1).

Vu les progrès incessants dans les méthodes numériques, dans les librairies informatiques de stockage et de communication, cet environnement n'est pas clos, mais permet d'intégrer facilement ces nouvelles techniques; c'est pourquoi il a été programmé en un langage orienté objet (C++). D'un autre côté, le programme produit par cet environnement peut être intégré dans une application.

Nous allons montrer comment la notion de coordination a été greffée à l'intérieur d'un procédé classique de la simulation numérique. Nous divisons la description de cet environnement en deux parties correspondant à deux phases dans l'utilisation de l'environnement. La première, que nous appelons la **phase des spécifications**, est une étape interactive où l'environnement recueille les informations concernant le programme de simulation à créer: description du problème, des transformations à appliquer etc. La deuxième, que nous désignons par la **phase de la production**, est une étape automatique dans laquelle l'environnement utilise ces informations pour produire un programme qui peut être exécuté dans une machine parallèle ou séquentielle.

En nous basant sur le cadre défini dans le chapitre précédent, nous n'allons plus parler des MDD mais de coordinations entre problèmes.

4.2 Approches avec ou sans coordination

Pour situer notre contribution, nous allons montré la différence entre un environnement classique c'est-à-dire sans coordination et notre environnement dit avec coordinations.

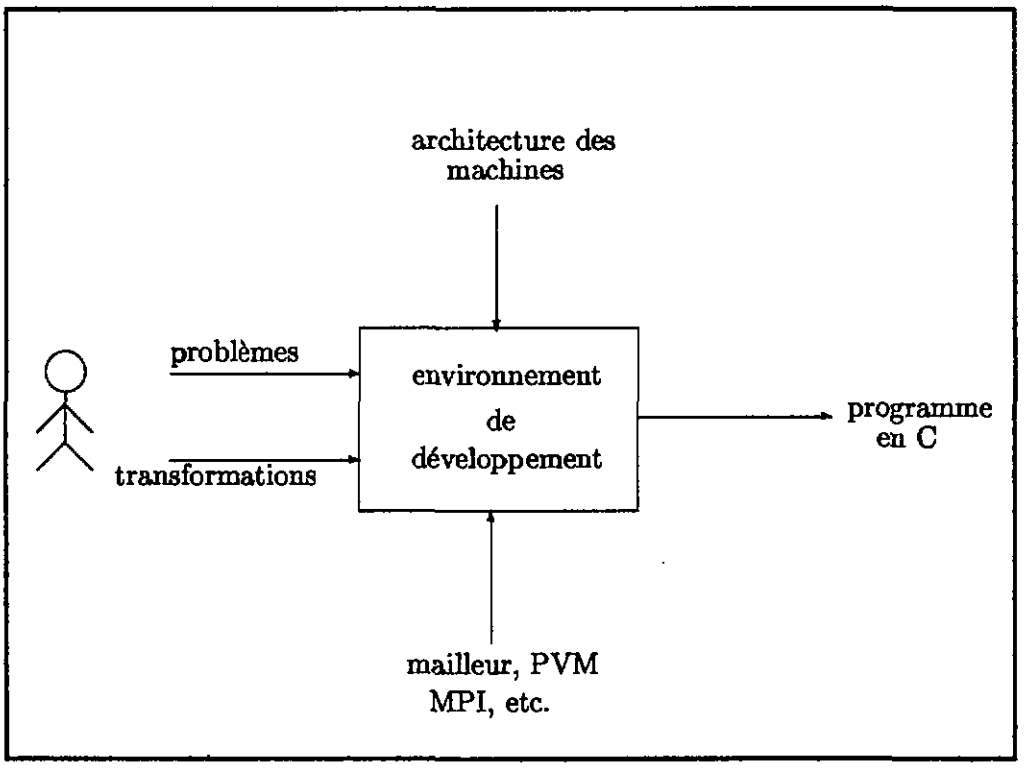

Figure 4.1 : vue globale de l'environnement.

4.2.1 Approche sans coordination

Dans une approche sans coordination, la simulation d'un problème se compose essentiellement de quatre étapes:

1. **Spécification du problème.** Il s'agit de définir le domaine Ω, l'opérateur ainsi que les conditions aux limites sur les frontières du domaine. Après avoir spécifié les noeuds sur les frontières, le modélisateur fait appel à un *mailleur* qui permet de discrétiser le domaine en fonction du type d'éléments finis choisis.

2. **Choix de l'algorithme numérique.** Le choix de l'algorithme numérique dépend du type d'opérateur (symétrique, linéaire, etc.) et du conditionnement de la matrice (voir section 1.3).

3. **Choix des structures des données.** Après la discrétisation des EDP, on obtient généralement des matrices de grande dimension et en plus elles sont creuses. Suivant que la méthode soit itérative ou directe, il convient de choisir la manière de stocker ces matrices. Un stockage adéquat permet non seulement d'économiser la mémoire, mais aussi de supprimer les opérations inutiles (multiplications par zéro, addition d'un membre nul, etc.).

4. **Production du code.** Il s'agit de produire le code correspondant à l'algorithme numérique et aux structures de données choisies.

Le modélisateur utilise des programmes tels que l'intégration numérique, la numérotation des noeuds pour diminuer la largeur de la matrice autour de la diagonale, et effectue

l'assemblage de la matrice. Il peut spécifier une bonne matrice pente qui jouera le rôle d'un pré-conditionneur matriciel et utiliser des librairies pour les opérations algébriques. Enfin dans le cas de la programmation parallèle, un utilitaire de découpage de domaine et les librairies de communications sont intégrés à son programme.

Tous ces utilitaires que nous avons cités existent dans le domaine public, ils sont généralement testés et adaptés à de nombreuses machines. Pour mettre en place un environnement interactif sans coordination, l'informaticien doit mettre en oeuvre un éditeur graphique pour la spécification du domaine, une base de données qui contient les opérateurs les plus utilisés, une base de données des éléments finis, une base de données pour les algorithmes et enfin une interface qui permet au modélisateur d'effectuer ses choix entre les différentes possibilités (algorithme, type d'élément fini, mailleur, etc.).

4.2.2 Approche avec coordinations

Il s'agit de rajouter un nouvel utilitaire de coordinations que nous appellerons **coordinateur** (voir la figure 4.2). Les aspects les plus importants de ce coordinateur est l'interactivité

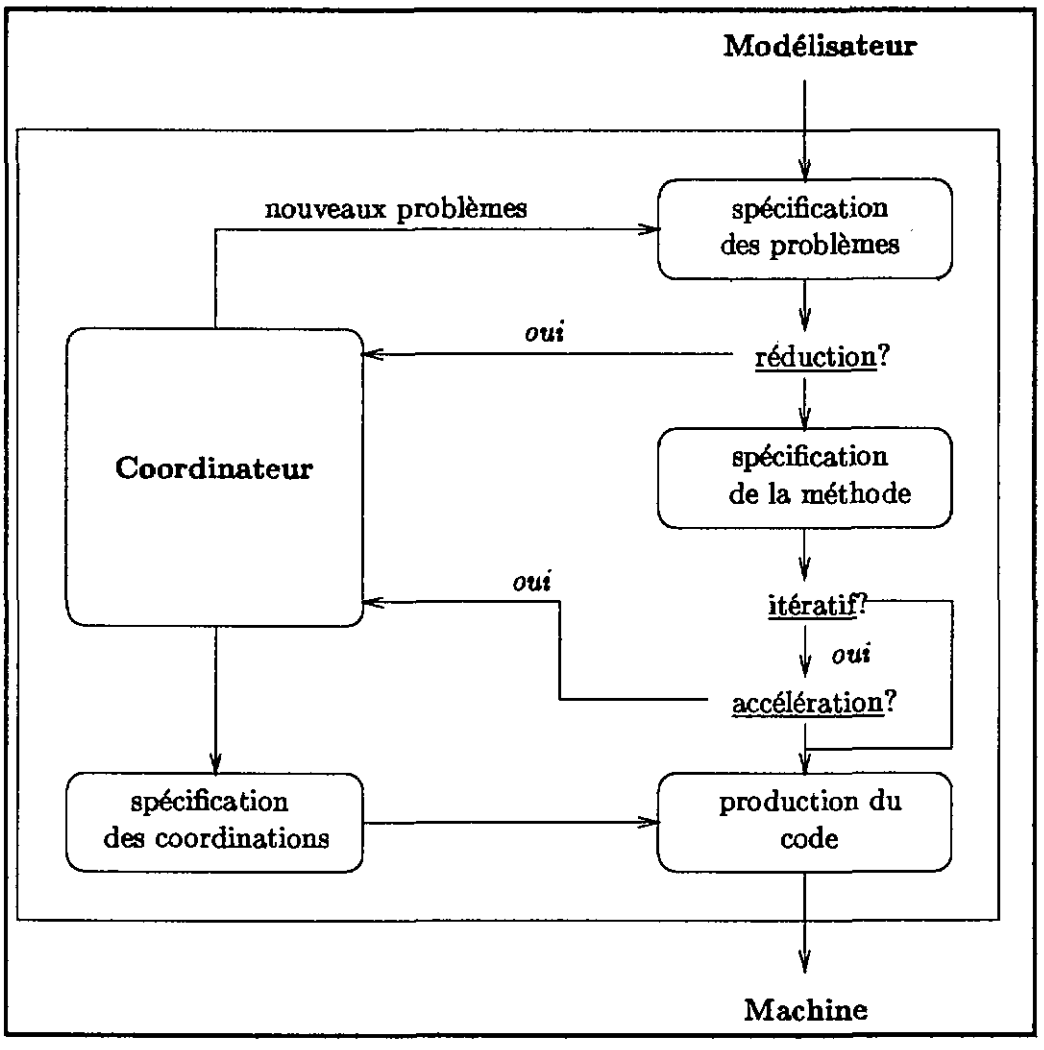

Figure 4.2 : coordinateur des problèmes.

et le graphisme. Être interactif oblige à ne faire aucune transformation du problème d'une manière automatique. Quant au graphisme, le but est de permettre au modélisateur de voir les transformations qu'il applique à chaque problème pour faire entrer en jeu la mémoire visuelle.

Muni de ses transformations, le coordinateur crée de nouveaux problèmes tout en établissant un lien, c'est-à-dire une coordination, entre ces problèmes et le problème transformé.

Nous distinguons deux types de transformations:

1. **Réduction.** Ces transformations peuvent être faites juste après la spécification d'un problème. Elles permettent de remplacer le problème à transformer par un nouveau problème muni d'un ensemble de contraintes (section 3.3). Pour l'instant, nous retenons comme exemple de transformation:

   - Envelopper et réduire aux frontières. Le domaine physique est remplacé par un nouveau domaine plus régulier qui contient le premier. Le nouveau problème est un prolongement sur ce domaine régulier et les contraintes s'appuient sur les valeurs que doit avoir la solution du problème transformé sur la frontière.
   - Subdiviser et réduire aux frontières. Le domaine physique est subdivisé en sous-domaines sans recouvrement. Nous obtenons alors plusieurs problèmes; chacun est une restriction du problème transformé sur le sous-domaine qui lui est associé. Les contraintes assurent soit la continuité de la solution soit l'annulation de la somme des dérivées normales.

2. **Accélération.** Ces transformations sont utilisées quand le modélisateur sait qu'il va choisir un algorithme itératif. Elles permettent de générer des problèmes auxiliaires qui joueront le rôle d'accélérateur. En s'appuyant sur les exemples vus dans le chapitre 2, voici quelques transformations:

   - Subdiviser et recouvrir. Après avoir subdivisé le domaine en sous-domaines avec recouvrement, nous obtenons autant de sous-problèmes que de sous-domaines. L'ordre et la manière de combiner les sous-solutions dépendent du type de la coordination (i.e., additive ou multiplicative).
   - Grossir le maillage avec intersection. Suivant le nombre de niveaux choisis, il s'agit de mettre en oeuvre les méthodes multigrille (section 2.3.1). Il y aura autant de coordinations qu'il a de niveaux. À chaque niveau le problème qui a moins de noeuds est un problème auxiliaire de celui qui a plus de noeuds.
   - Grossir le maillage sans intersection. Dépendant du nombre de niveaux choisis, il s'agit de mettre en oeuvre les méthodes à bases hiérarchiques (section 2.3.2) où chaque noeud n'appartient qu'à un niveau. Dans ce cas, il n'a qu'une seule coordination qui est à la fois multiple et additive.

Dans tous les cas et si aucune modification n'est explicitement demandée par l'utilisateur, les nouveaux problèmes héritent à la fois de l'opérateur et des conditions aux limites de la frontière du domaine transformé.

4.3 Phase des spécifications

Mis à part les cas évidents comme la restriction du problème à un sous-domaine, cette phase est entièrement interactive. Elle utilise des fenêtres graphiques, des fenêtres de dialogue, des menus et des éditeurs de texte. L'utilisateur suivra généralement, les étapes suivantes:

4.3.1 Spécification du problème

Spécifier un problème, c'est déterminer à la fois son aspect fonctionnel et son aspect géométrique. Dans le premier cas, il s'agit de spécifier les EDP (*i.e.*, l'opérateur) et les conditions aux limites. Tandis que dans le deuxième cas, il s'agit du domaine physique et du type d'éléments finis. La spécification devient complète lorsque les conditions aux limites sont définies. Nous allons voir les informations dont l'environnement a besoin pour cette spécification et sous quelles formes elle sont définies:

1. **Aspect fonctionnel.**

   (a) **L'opérateur.** Nous utilisons la forme

   $$-\sum_{i,j=1}^{d} \frac{\partial}{\partial x_i}(\alpha_{i,j}\frac{\partial u}{\partial x_j}) + \alpha_0 u \tag{4.3.1}$$

   Ici, l'opérateur est supposé symétrique.
   Grâce à des fenêtres de dialogue, l'utilisateur spécifie l'opérateur sous la forme d'une matrice (symétrique) de fonctions $\alpha_{i,j}$ et d'une fonction α_0. Ces fonctions peuvent être introduites sous une forme algébrique à l'aide d'un éditeur de texte. Nous laissons le soin à l'utilisateur de s'assurer que l'opérateur est elliptique, en particulier, que $\alpha_{i,j}$ et α_0 sont des fonctions bornées dans Ω.

   (b) **Les conditions aux limites.** Par rapport à la frontière du domaine ou à une portion Γ_c de la frontière, l'utilisateur choisit le type de conditions aux limites à appliquer sur Γ_c et une fonction g_c qui sera la valeur de ces conditions. Dans le cas où le type est de Dirichlet, nous supposons que $u = g_c$ sur Γ_c. Dans le cas Neumann, nous admettons que

   $$\sum_{i,j=1}^{d} \alpha_{i,j}\frac{\partial u}{\partial x_j}\mathbf{n}_i = g_c$$

   ($\mathbf{n}_i$ est la $i^{\text{ème}}$ composante de la normale unitaire sortante du domaine.)
   Dans ce cas, la forme variationnelle du problème devient:

   $$\sum_{i,j=1}^{d} \int_\Omega \alpha_{i,j}\frac{\partial u}{\partial x_i}\frac{\partial v}{\partial x_j} + \int_\Omega \alpha_0 u v = \int_\Omega f v + \int_{\Gamma_c} g_c v + \ldots$$

2. **Aspect géométrique.**

   (a) **Les domaines.** Dans cette phase l'utilisateur a recours à un éditeur graphique. En plus de la spécification du domaine, cet éditeur graphique permet:

   - De définir des noeuds sur les frontières. Ces noeuds seront utilisés au cours de la discrétisation.
   - De scinder la frontière externe Γ en des portions Γ_c. Chaque portion peut recevoir une condition aux limites différente.
   - De faire des transformations géométriques telles que la subdivision, l'enveloppement, le grossissement du maillage etc.

   (b) **La discrétisation.** Nous avons préalablement programmé des éléments finis de types triangles et carrés. L'utilisateur sélectionne parmi ces éléments ceux qu'il aimerait utiliser pour la discrétisation. À partir des noeuds sur la frontière et du type d'éléments finis, une discrétisation automatique du domaine est appliquée.

4.3.2 Spécification des algorithmes

Ici, nous avons choisi les algorithmes les plus utilisés. Dans le cas itératif, nous avons le gradient conjugué et GMRES (Saad et Schultz, 1986) suivant que la coordination soit additive ou multipicative. Dans le cas direct, nous utilisons la factorisation LU et en particulier, la factorisation de Cholesky.

4.3.3 Spécification des coordinations

L'éditeur graphique est muni d'un certain nombre de transformations géométriques. L'utilisateur peut sélectionner le domaine qu'il aimerait visualiser et ensuite choisir une des transformations. Après chaque transformation, l'éditeur appelle le coordinateur pour créer de nouveaux problèmes et générer la coordination adéquate. L'utilisateur peut accéder aux opérateurs des nouveaux problèmes s'il désire les modifier.

A l'aide d'une fenêtre de dialogue, l'utilisateur peut accéder aux paramètres d'une coordination pour spécifier sous forme d'opérateurs les applications des transferts ($\mathcal{R}$ et $\mathcal{Q}$), la fréquence d'utilisation des problèmes auxiliaires (Π) et le type de coordination (voir section 3.2). Nous supposons que toutes les fréquences Π sont définies sous formes $n_1 + i \times n_2$, $i \in I\!N$. Dans ce cas, l'utilisateur donnera les valeurs de n_1 et de n_2.

4.4 Phase de la production

Dans cette section, le problème traité est identifié par $\mathcal{P}$. Son domaine et l'ensemble des éléments finis seront désignés par Ω et $\mathcal{T}_h$. Nous notons Γ_D, la réunion de toutes les portions de $\partial\Omega$ admettant une condition aux limites de type Dirichlet et la fonction g_D la valeur de ces conditions. De la même façon, Γ_N pour les conditions de types Neumann et g_N la valeur de ces conditions. Nous supposons que $\partial\Omega = \Gamma_D \cup \Gamma_N$ avec $\Gamma_D \cap \Gamma_N = \emptyset$. En outre, nous admettons que $\Gamma_D \neq \emptyset$, c'est-à-dire il existe au moins un noeud où la valeur de la fonction u est fixée. Sinon, nous choisissons un noeud au hasard sur la frontière $\partial\Omega$ et fixons cette valeur à zéro (voir la section 1.2.4).

4.4.1 Construction des équations algébriques

Le but de cette section est de montrer le procédé utilisé par l'environnement pour construire la matrice A et le vecteur F du système d'équations algébriques d'un problème $\mathcal{P}$.

Assemblage de la matrice du système

La matrice qui correspond à l'opérateur défini dans (4.3.1) admet comme éléments:

$$a_{p,q} = \sum_{i,j=1}^{d} \int_{\Omega} \alpha_{i,j} \frac{\partial \varphi_p}{\partial x_i} \frac{\partial \varphi_q}{\partial x_j} + \int_{\Omega} \alpha_0 \varphi_p \varphi_q$$

Nous définissons la contribution de chaque élément fini $\mathcal{K}_e$ par:

$$a^e_{p,q} = \sum_{i,j=1}^{d} \int_{\mathcal{K}_e} \alpha_{i,j} \frac{\partial \varphi_p}{\partial x_i} \frac{\partial \varphi_q}{\partial x_j} + \int_{\mathcal{K}_e} \alpha_0 \varphi_p \varphi_q$$

Pour calculer cette contribution, nous avons préalablement mis à disposition deux procédures pour chaque type d'éléments finis:

1. $s^e_{p,q}(h,i,j)$ qui calcule par intégration numérique la valeur:

$$\int_{\mathcal{K}_e} h \frac{\partial \varphi_p}{\partial x_i} \frac{\partial \varphi_q}{\partial x_j}$$

   pour une fonction réelle h quelconque,

2. $m^e_{p,q}(h)$ qui calcule par intégration numérique la valeur:

$$\int_{\mathcal{K}_e} h \varphi_p \varphi_q$$

Les valeurs de $s^e_{p,q}(h,i,j)$ et $m^e_{p,q}(h)$ sont prises nulles si l'un des noeuds n_p ou n_q ne se trouve pas sur $\mathcal{K}_e$.

L'environnement calcule $a^e_{p,q}$ en se basant sur l'égalité:

$$a^e_{p,q} = m^e_{p,q}(\alpha_0) + \sum_{\substack{n_p \in \mathcal{K}_e \\ n_q \in \mathcal{K}_e}} \sum_{i,j=1}^{d} s^e_{p,q}(\alpha_{i,j}, i, j)$$

et finalement le calcul de $a_{p,q}$ se fait par:

$$a_{p,q} = \sum_{\mathcal{K}_e \in \mathcal{T}_h} a^e_{p,q}$$

Comme nous avons vu à la section 1.2.4, la matrice A est construite en ne tenant compte que des valeurs $a_{p,q}$ tels que $n_p \notin \Gamma_D$ et $n_q \notin \Gamma_D$ (*i.e.*, les noeuds où la valeur de la fonction n'est pas fixée).

Assemblage du vecteur F

Le vecteur F est calculé grâce à la somme de trois vecteurs:

1. F_f qui correspond à la fonction f, second membre des EDP, et dont les éléments sont $\int_\Omega f\varphi_p$,
2. F_N qui correspond à la contribution des conditions aux limites de type Neumann. Les éléments sont:
$$\int_{\Gamma_N} g_N\varphi_p \quad n_p \in \Gamma_N$$
3. $\mathrm{F}_D = -\mathrm{A}^{\mathrm{Frt}}\mathrm{u}_D$ qui correspond à la condition aux limites de type Dirichlet. Avec
$$\begin{array}{rcll} \mathrm{A}^{\mathrm{Frt}} & = & (a_{p,q}) & n_p \notin \Gamma_D, n_q \in \Gamma_D \\ \mathrm{u}_D & = & (g_D(n_q)) & n_q \in \Gamma_D \end{array}$$

Nous avons programmé pour chaque type d'éléments finis deux procédures:

1. $\mathrm{F}^e(h,p)$ qui calcule, pour tout $n_p \in \mathcal{K}_e$ et une fonction h, l'intégrale
$$\int_{\mathcal{K}_e} h\varphi_p$$
2. $\mathrm{F}^{e,\mathrm{Frt}}(h,p,\Gamma_c)$ pour calculer, pour tout $n_p \in \mathcal{K}_e$, une fonction h et une portion de frontière Γ_c, l'intégrale
$$\int_{\partial\mathcal{K}_e\cap\Gamma_c} h\varphi_p$$

La contribution de chaque élément fini $\mathcal{K}_e$ aux vecteurs F_f et F_N sont calculés respectivement par:
$$\begin{array}{rcl} (\mathrm{F}_f^e)_p & = & \mathrm{F}^e(f,p) \\ (\mathrm{F}_N^e)_p & = & \begin{cases} \mathrm{F}^{e,\mathrm{Frt}}(g_N,p,\Gamma_N) & n_p \in \Gamma_N \\ 0 & \text{sinon} \end{cases} \end{array}$$

Finalement F est calculé en additionnant les vecteurs F_f, F_D et F_N après avoir calculé $\mathrm{F}_D = -\mathrm{A}^{\mathrm{Frt}}\mathrm{u}_D$.

4.4.2 Résolution du problème

Pour faciliter l'intégration du code produit à un programme existant, nous avons défini deux procédures qui permettent de résoudre un problème $\mathcal{P}$ aux EDP:

1. au niveau fonctionnel, désignée par Résoudre_Func$(\mathcal{P}, f)$ où f est une fonction;
2. au niveau matriciel, que nous nommons Résoudre_Sys$(\mathcal{P}, \mathrm{F})$ où F est un vecteur.

Dans le premier cas, la construction du vecteur F est de la responsabilité de l'environnement, quant au deuxième cas, il est de la responsabilité de l'utilisateur du programme produit. La programmation de Résoudre_Func$(\mathcal{P}, f)$ est faite de la manière suivante:

Algorithme 4.1 (Résolution du problème au niveau fonctionnel)

(1) **procédure** Résoudre_Func($\mathcal{P}, f$) : **vecteur**;
(2) Contruire_Matrice($\mathcal{P}$);
(3) u_D = Contribution_Dirichlet($\mathcal{P}, \Gamma_D, g_D$);
(4) F_N = Contribution_Neumann($\mathcal{P}, \Gamma_N, g_N$);
(5) F_f = Contribution_Force($\mathcal{P}, f$);
(6) F = Construire_F_Global($\mathcal{P}, F_f, u_D, F_N$);
(7) Résoudre_Sys($\mathcal{P}$, F);
(8) $\overline{u}^0$ = Solution($\mathcal{P}$);
(9) $\overline{u} = \overline{u}^0 + u_D$; **commentaire** : rajouter les valeurs fixées.
(10) **retourner** $\overline{u}$;
(11) **fin procédure.**

Avec ce mode de programmation, un utilisateur peut intervenir à l'intérieur du programme pour redéfinir de nouvelles conditions aux limites ou changer la fonction f sans faire appel à l'environnement. Par contre, il n'aura pas le droit de changer les types (*i.e.*, Neumann ou Dirichlet) de ces conditions, car la construction des problèmes auxiliaires en dépend.

Résoudre_Sys($\mathcal{P}$, F) est la procédure à appeler pour résoudre le système Au = F. Dans le cas où un algorithme itératif est utilisé, elle doit être de la forme:

Algorithme 4.2 (Résolution du problème au niveau matriciel)

(1) **procédure** Résoudre_Sys($\mathcal{P}$, F);
(2) **iterer**
(3) z = CalculerAu($\mathcal{P}$, u^k);
(4) r^k = z − F;
(5) δ = Corriger($\mathcal{P}, r^k, k$);
(6) $u^{k+1} = u^k - \alpha_k \delta$;
(7) **fin d'itérations**
(8) Solution($\mathcal{P}$) ← u^k; **commentaire** : mémoriser la solution.
(9) **fin procédure.**

Le calcul de la multiplication de A par un vecteur u (instruction (3)) est laissé à la charge du producteur de code. Ceci, pour deux raisons:

1. dans certains problèmes la matrice A n'est pas calculée explicitement comme nous l'avons constaté dans les méthodes de réduction aux frontières;
2. dans le cas de la programmation parallèle, la matrice A est répartie sur un certain nombre de processeurs. Seul le producteur de code, pour l'instant, sait comment effectuer la multiplication en tenant compte de cette répartition.

L'appel de Corriger($\mathcal{P}, r^k, k$) permet d'appeler le coordinateur au cas où nous devons utiliser un pré-conditionneur à l'itération k (voir la section suivante).

4.4.3 Mise en oeuvre des coordinations

Nous avons vu au chapitre précédent qu'une coordination est caractérisée par les deux applications de transferts $\mathcal{R}$ et $\mathcal{Q}$, et par la suite Π. Dans cette section, nous allons montrer comment les matrices de transfert R et Q sont construites et les algorithmes utilisés pour chaque type de coordinations.

Les matrices de transfert

D'une manière générale, la matrice de l'application Q correspond au changement de base entre l'espace fonctionnel du problème auxiliaire et l'espace fonctionnel du problème original.

Soient $(\varphi_p)_{p=1,\ldots,N}$ (respectivement $(\tilde{\varphi}_q)_{q=1,\ldots,\tilde{N}}$) la base de l'espace fonctionnel V du problème original (respectivement V_a) et $\{n_p, p = 1,\ldots,N\}$ l'ensemble des noeuds du maillage relativement à V du problème auxiliaire.

Le changement de base de V_a vers V est défini par la matrice $\mathrm{Q} = (\mathrm{q}_{p,q})$ où $\forall p = 1,\ldots,N$ et $\forall q = 1,\ldots,\tilde{N}$, nous avons:

$$\mathrm{q}_{p,q} = \tilde{\varphi}_q(n_p)$$

En effet, pour toute fonction $\tilde{u} = \sum_{q=1}^{\tilde{N}} \tilde{\mathrm{u}}_q \tilde{\varphi}_q \in V_a$, sa $p^{\text{ème}}$ composante dans V est calculée par:

$$\mathrm{u}_p = \tilde{u}(n_p) = \sum_{q=1}^{\tilde{N}} \tilde{\mathrm{u}}_q \tilde{\varphi}_q(n_p) = \sum_{q=1}^{\tilde{N}} \mathrm{q}_{p,q} \tilde{\mathrm{u}}_q$$

Les opérateurs que nous utilisons sont définis de V dans V (*i.e.*, $W = V$).

Pour trouver la matrice de transfert R entre V et V_a, il faut revenir à la définition du vecteur F et à sa représentation $\tilde{\mathrm{F}}$ dans V_a.

Par définition, nous avons $\forall p = 1,\ldots,N$

$$\tilde{\mathrm{F}}_p = \int_\Omega \tilde{\varphi}_p f = \int_\Omega (\sum_{q=1}^{\tilde{N}} \tilde{\varphi}_p(n_q)\varphi_q) f = \sum_{q=1}^{\tilde{N}} \tilde{\varphi}_p(n_q) \int_\Omega f\varphi_q = \sum_{q=1}^{\tilde{N}} \tilde{\varphi}_p(n_q)\mathrm{F}_q$$

La matrice $\mathrm{R} = (\mathrm{r}_{p,q})$ est définie par: $\forall p = 1,\ldots,\tilde{N}$ et $\forall q = 1,\ldots,\tilde{N}$

$$\mathrm{r}_{p,q} = \tilde{\varphi}_p(n_q) = \mathrm{q}_{q,p} \tag{4.4.1}$$

Par conséquent, nous avons $\mathrm{Q} = \mathrm{R}^T$. On peut trouver le même résultat en se basant sur les projections orthogonales(voir Barry *et al.*, 1996, pages 16–17)

Dans le programme produit, seule la matrice R est calculée en utilisant l'équation (4.4.1).

Les algorithmes des coordinations

A partir des transformations effectuées par l'utilisateur, le coordinateur génère alors une arbre de dépendance entre problèmes. Le passage de la résolution d'un problème à la résolution de ses problèmes auxiliaires se fait à travers les appels de Corriger($\mathcal{P}, r, k$). Nous allons voir dans cette section comment le coordinateur va définir la procédure Corriger($\mathcal{P}, r, k$) pour chaque type de coordinations.

- **Cas d'une coordination pour accélération simple:** nous supposons que le problème $\mathcal{P}_a$ est le problème auxiliaire de $\mathcal{P}$. Dans ce cas, la coordination est produite de la manière suivante:

Algorithme 4.3 (Coordination pour accélération simple)

```
(1)  procédure Corriger(P, r, k) : vecteur;
(2)    si (n ∉ Π) alors retourner r;
(3)    r_a = Rr;
(4)    Résoudre_Sys(P_a, r_a);
(5)    δ_a = Solution(P_a);
(6)    δ = Qδ_a
(7)    retourner δ;
(8)  fin procédure.
```

Par la suite et pour des raisons de simplification, nous n'allons pas écrire l'instruction (2). Dans la pratique, nous avons généralement soit des coordinations fortes où cette instruction n'a aucun effet, soit des suites Π qui sont de la forme simple $\{n_1 + i \times n_2, i \in I\!N\}$.

- **Cas d'une coordination multiple additive:** nous supposons que $\mathcal{P}$ admet J problèmes auxiliaires $(\mathcal{P}_j)$ $j = 1, \ldots, J$

Algorithme 4.4 (Coordination pour accélération multiple additive)

```
(1)  procédure Corriger(P, r, k) : vecteur;
(2)    δ = 0;
(3)    pour tout j = 1 jusqu'à J faire
(4)      r_j = R_j r;
(5)      Résoudre_Sys(P_j, r_j);
(6)      δ_j = Solution(P_j);
(7)      δ = δ + Q_j δ_j;
(9)    retourner δ;
(10) fin procédure.
```

Remarque: la boucle (3) peut être parallélisée.

- **Cas d'une coordination multiple successive:** $(\mathcal{P}_j)$, $j = 1, \ldots, J$ étant les problèmes auxiliaires de $\mathcal{P}$.

  Dans le cas de cette coordination, l'ordre d'utilisation des solutions de chaque problème est important. Il peut être modifié par l'utilisateur.

Algorithme 4.5 (Coordination pour accélération multiple successive)

```
(1)  procédure Corriger(P, r, k) : vecteur;
(2)    r_1 = R_1 r;
(3)    Résoudre_Sys(P_1, r_1);
(4)    δ_1 = Solution(P_1);
(5)    δ = Q_1 δ_1;
(6)    pour tout j = 2 jusqu'à J faire
(7)      dr = CalculerAu(P, δ);
(8)      r_j = r_{j-1} − dr;
(9)      Résoudre_Sys(P_j, r_j);
```

```
(10)      δ_j = Solution(P_j);
(11)      δ = Q_j δ_j;
(13)    retourner δ;
(14)  fin procédure.
```

- **Cas d'une coordination par réduction:** nous supposons que le problème $\mathcal{P}$ est réduit au problème $\mathcal{P}_r$ défini par l'équation

$$B\tilde{A}^{-1}B^T\lambda = B\tilde{A}^{-1}\tilde{F} - c$$

Nous notons $\tilde{\mathcal{P}}$, le problème correspondant à la résolution du système $\tilde{A}\tilde{u} = \tilde{F}$. Il est aussi le problème obtenu par transformation de $\mathcal{P}$. B et c sont la matrice et le vecteur obtenus après discrétisation de la contrainte $\mathcal{B}(\tilde{u}) = c$. Nous notons R, la matrice de transfert du problème $\mathcal{P}$ dans $\tilde{\mathcal{P}}$.

Comme nous l'avons mentionné, la matrice $B\tilde{A}^{-1}B^T$ n'est pas calculée explicitement. Le producteur du code redéfinit la procédure CalculerAu($\mathcal{P}_r, \lambda$) de la manière suivante:

```
(1)   procédure CalculerAu(P_r, λ) : vecteur;
(2)     F̃ = B^T λ;
(3)     Résoudre_Sys(P̃, F̃);
(4)     ũ = Solution(P̃);
(5)     res = Bũ;
(6)     retourner res;
(7)   fin procédure.
```

Finalement, la résolution du problème $\mathcal{P}$ est programmée comme suit:

Algorithme 4.6 (Résolution d'un problème réduit)

```
(1)   procédure Résoudre_Sys(P, F);
(2)     F̃ = RF;
(3)     Résoudre_Sys(P̃, F̃);
(4)     ũ = Solution(P̃);
(5)     s_r = Bũ − c;
(6)     Résoudre_Sys(P_r, s_r);
(7)     λ̄ = Solution(P_r);
(8)     Résoudre_Sys(P̃, F̃ − B^T λ̄);
(9)     ũ = Solution(P̃);
(10)    ū = R^T ũ;
(11)    Solution(P) ← ū;
(12)  fin procédure.
```

4.5 La programmation parallèle

Il existe principalement deux types de machines parallèles: les machines à mémoire partagée et les machines à mémoires distribuées. Dans le premier type, tous les processeurs

partagent le même espace mémoire. Dans le deuxième cas, chaque processeur possède un espace mémoire local indépendant. Ceci inclut un ensemble de machines séquentielles reliées par un réseau.

Dans cette section, nous allons considérer les machines à mémoires distribuées et nous allons supposer que nous disposons d'une librairie de communications entre processeurs indépendamment de l'architecture de la machine. Des librairies telles que MPI et PVM obéissent à cette règle. Elles fournissent des procédures (**envoyer**, **recevoir**, etc.) qu'on peut utiliser sans se soucier du type d'architecture de la machine.

Pour produire un programme destiné à une machine parallèle, l'environnement:

1. sélectionne les problèmes pouvant être exécutés en parallèle;

2. distribue ces problèmes sur les processeurs mis à disposition.

L'attribution de deux ou plusieurs processeurs à la résolution d'un problème se justifie lorsque la dimension du système d'équations est assez importante. C'est dans cette optique que nous supposons que seuls les systèmes résolus par des méthodes itératives, et contrairement aux méthodes directes, seront candidats à la parallélisation.

Dans ce qui suit, nous allons montrer comment l'environnement procède à la parallélisation des méthodes itératives.

4.5.1 Parallélisation d'une méthode itérative

La parallélisation des méthodes itératives (MI) consiste, premièrement, à répartir les matrices et les vecteurs sur les processeurs et, ensuite, à élaborer des algorithmes permettant de faire des opérations algébriques qui tiennent compte de cette répartition. Ces opérations algébriques parallélisées (OAP) sont généralement utilisées pour effectuer la multiplication d'une matrice par un vecteur, l'addition (ou la soustraction) de deux vecteurs, le produit scalaire de deux vecteurs, etc.

Pour pouvoir résoudre un système d'équations $Au = F$, il faut élaborer des algorithmes pour les OAP qui dépendent de la manière dont la matrice A a été répartie entre les processeurs. Dans cette section, nous allons définir comment la répartition est faite et ensuite, nous élaborerons des algorithmes qui permettent d'effectuer ces OAP en tenant compte de cette répartition.

Découpage du domaine

Nous supposons avoir p processeurs que nous notons par $P_1, P_2, \ldots, P_p$ et une partition $(\mathrm{P}_1, \mathrm{P}_2, \ldots, \mathrm{P}_p)$ de l'ensemble des éléments finis . A chaque processeur P_j nous affectons la partie P_j du domaine Ω. Cette affectation définit la répartition du domaine Ω à travers les p processeurs.

Dans notre cas, le découpage (P_j), $j = 1, \ldots, p$, est supposé être une vraie partition, dans le sens mathématique, de l'ensemble des éléments finis. C'est-à-dire, qu'un élément fini ne doit être affecté qu'à un processeur. Les processeurs n'auront alors en commun que des noeuds, arêtes ou des faces suivant la dimension géométrique du domaine Ω (voir l'exemple de la figure 4.3 pour le découpage et la figure 4.4 pour la répartition).

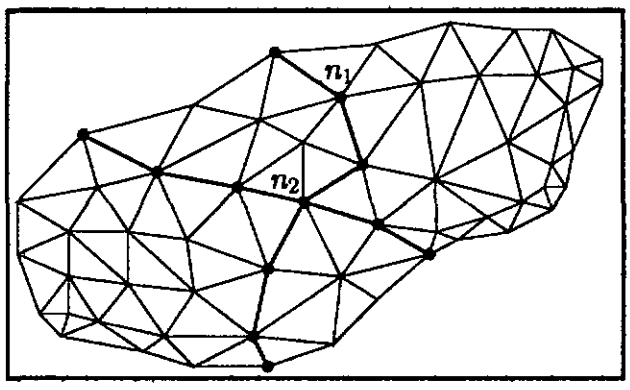

Figure 4.3 : découpage du domaine en quatre parties.

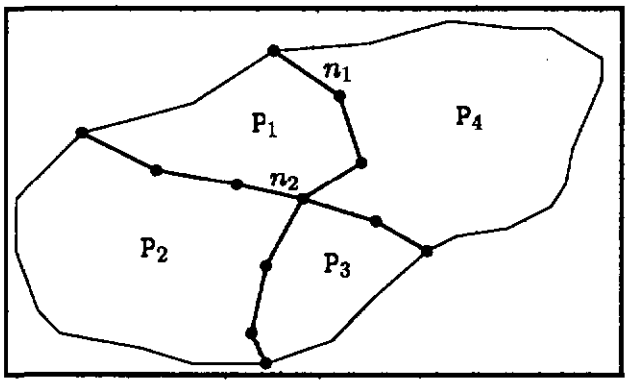

Figure 4.4 : attribution des parties aux processeurs.

Pour toute fonction $u_h \in V^h$, nous définissons $u_h^j = u_{h|P_j} = \sum_{n_i \in P_j} u_i \varphi_i$ la restriction de la fonction u_h à la partie P_j. De la même manière, nous définissons u^j, la restriction du vecteur u.

Pour alléger les notations, nous supposons que tous les vecteurs u^j ont la dimension N_h, les valeurs correspondantes aux noeuds qui ne se trouvent pas dans P_j étant par définition nulles.

Découpage du problème

Partant du découpage du domaine défini ci-dessus, la répartition du problème ne fait que restreindre le problème global aux parties P_j de chaque processeur. Pour cela, nous allons revenir à la forme variationnelle du problème (1.2.8) et appliquer cette restriction.

En effet, dans chaque processeur P_j, nous définissons le problème local suivant:

$$a_j(u_h^j, v_h^j) = f_j(v_h^j), \qquad \forall\, u_h,\ v_h \in V^h \tag{4.5.1}$$

avec

$$a_j(u_h^j, v_h^j) = \int_{P_j} \nabla u_h^j \cdot \nabla v_h^j \quad \text{et} \quad f_j(v_h^j) = \int_{P_j} f\, v_h^j$$

En appliquant $\int_\Omega = \sum_j^p \int_{P_j}$, nous avons:

$$\begin{aligned} a(u,v) &= \int_\Omega \nabla u_h \cdot \nabla v_h \\ &= \sum_{j=1}^p \int_{P_j} \nabla u_h \cdot \nabla v_h \\ &= \sum_{j=1}^p \int_{P_j} \nabla u_h^j \cdot \nabla v_h^j \qquad \text{(par définition)} \\ a(u,v) &= \sum_{j=1}^p a_j(u_h^j, v_h^j) \end{aligned}$$

de la même manière, nous avons:

$$f(v) = \sum_{j=1}^p f_j(v^j)$$

Le problème global devient "la somme" des sous-problèmes locaux. Au niveau matriciel, si A^j et F^j sont la matrice et le vecteur obtenus en discrétisant le sous-problème local (4.5.1) alors:

$$A = \sum_{j=1}^p A^j \quad \text{et } F = \sum_{j=1}^p F^j \tag{4.5.2}$$

L'intérêt de ce découpage est que la matrice du problème global ne sera pas assemblée dans un seul processeur, mais chaque processeur P_j fait l'assemblage de la sous-matrice correspondant à sa partie P_j.

Opérations algébriques parallélisées

Les principales opérations algébriques qu'une méthode itérative utilise sont: l'addition de deux vecteurs, la multiplication d'un réel et d'un vecteur, la multiplication d'un vecteur et d'une matrice, et le produit scalaire de deux vecteurs.

Bien entendu, au cours de l'exécution parallèle d'un algorithme, il faut s'assurer que toutes les données réparties (scalaire, vecteur ou matrice) sont cohérentes d'un point vue global. En d'autres termes, les variables devraient avoir les mêmes valeurs comme si l'algorithme était exécuté dans un seul processeur. Pour cela, il suffit que toute information partagée ait la même valeur dans tous les processeurs.

Nous dirons qu'une **variable est cohérente** si elle a la même valeur dans tous les processeurs. Par contre, pour assurer la cohérence d'un vecteur, nous devons nous intéresser aux valeurs correspondant aux noeuds se trouvant sur les frontières entre les parties P_j. Dans ce cas, nous dirons que u est un **vecteur cohérent** si, pour tout noeud $n_i \in P_j \cap P_k$, nous avons:

$$u_i^j = u_i^k$$

(u_i étant la valeur du vecteur u relativement au noeud n_i)

En tenant compte du découpage que nous avons défini ci-dessus, nous allons montrer comment les algorithmes des OAP sont programmés et monter que: a) ils effectuent les opérations voulues, b) ils assurent la cohérence du résultat.

1. **Addition de deux vecteurs**
   Pour obtenir globalement $z = u + v$, il suffit d'additionner localement $z^j = u^j + v^j$.

   Si u et v sont cohérents, alors le vecteur z défini par $z^j = z_{|P_j}$ est cohérent et il vaut la somme des vecteurs u et v. □

2. **Multiplication d'un réel et d'un vecteur**
   Pour obtenir globalement $z = \alpha u$, il suffit de multiplier localement $z^j = \alpha u^j$.

   Si le réel α et le vecteur V sont cohérents, alors le vecteur z est cohérent et est le produit voulu. □

3. **Multiplication d'un vecteur et d'une matrice**
   Pour tout noeud n_i, nous définissons $\mathcal{I}(n_i) = \{j | n_i \in P_j\}$, l'ensemble des indices des parties auxquelles le noeud appartient. Dans le cas de l'exemple de la figure 4.3, nous avons $\mathcal{I}(n_1) = \{1, 4\}$ et $\mathcal{I}(n_2) = \{1, 2, 3, 4\}$.

   Pour toute partie P_j, nous notons $\partial_{\text{int}}P_j$ l'ensemble des noeuds se trouvant sur les frontières internes (*i.e.*, partagées avec une ou plusieurs régions).

   Avant de définir l'algorithme de la multiplication, nous remarquons d'emblée que l'équation (4.5.2) nous suggère un algorithme qui permet d'additionner les contributions de toutes les sous-matrices locales.

Algorithme 4.7 (Addition des contributions locales)

```
(1)   procédure AddContributions( inout u^j : vecteur);
(2)     commentaire : j est l'indice local du processeur.
(3)     pour tout (n_i ∈ ∂_int P_j ∧ k ∈ I(n_i) ∧ k ≠ j) faire
(4)       envoyer  u_i^j au processeur P_k;
(6)     pour tout (n_i ∈ ∂_int P_j ∧ k ∈ I(n_i) ∧ k ≠ j) faire
(7)       recevoir  u_i^k du processeur P_k;
(8)       u_i^j = u_i^j + u_i^k;
(10)  fin procédure.
```

Remarques:

- L'envoi de données (4) est supposé asynchrone sinon il y a blocage.
- Les données échangées sont envoyées sous forme {*indice*, *valeur*}, où *indice* détermine l'indice du noeud concerné, et *valeur* la valeur du vecteur relativement à ce noeud. L'indexation est globale (*i.e.*, chaque noeud admet un indice unique).
- Au lieu d'envoyer à un processeur une valeur à la fois, les valeurs sont groupées dans un seul bloc et envoyées au processeur concerné. Ceci réduira le temps des communications.

Finalement, pour calculer z = Au, d'après l'égalité (4.5.2), on a:

$$z = Au = \sum_{j=1}^{p} A^j u = \sum_{j=1}^{p} A^j u^j$$

Nous pouvons en déduire que l'algorithme de la multiplication sera comme suit:

Algorithme 4.8 (Multiplication matrice-vecteur parallélisée)

(1) **procédure** *ParaMulti*(A^j : **matrice**, u^j : **vecteur**, **inout** z^j : **vecteur**);
(2) **multiplier localement:** $z^j = A^j u^j$
(3) **appeler** *AddContributions*(z^j)
(4) **fin procédure.**

Nous pouvons vérifier aisément, que si u est cohérent alors z = Au et que celui-ci est cohérent. □

Du fait que la matrice A^j soit creuse, seuls les éléments non nuls sont mémorisés. Pour stocker cette matrice, l'environnement utilise la forme dite "en lignes" qui consiste à attribuer à chaque ligne de la matrice une structure contenant deux tableaux: le tableau *indices* qui mémorise les indices colonnes correspondants aux éléments non nuls et le tableau *valeurs* dans lequel sont stockées les valeurs de ces éléments. Ici, nous supposons que les matrices sont constantes au cours de l'exécution, car ces structures ne seront pas adéquates au cas où nous aimerions accéder à un élément particulier de la matrice.

4. **Le produit scalaire de deux vecteurs**
Pour calculer $\alpha = u \cdot v$, nous allons introduire un *pseudo-produit* local que nous notons $\odot$ et définissons par:

$$u^j \odot v^j = \sum_{n_i \in P_j} \frac{u_i^j v_i^j}{| \mathcal{I}(n_i) |}$$

où $| \mathcal{I}(n_i) |$ est le nombre d'éléments de l'ensemble $\mathcal{I}(n_i)$, c'est-à-dire le nombre des parties auxquelles le noeud n_i appartient.

Si u et v sont cohérents, alors nous avons:

$$\begin{aligned} \alpha &= \sum_i u_i v_i \\ &= \sum_{j=1}^{p} \sum_{n_i \in P_j} \frac{u_i v_i}{| \mathcal{I}(n_i) |} \\ &= \sum_{j=1}^{p} u^j \odot v^j \end{aligned}$$

En effet, le passage de la première à la deuxième égalité est justifié par le fait que pour tout noeud n_i, nous avons:

$$\begin{cases} u_i^j v_i^j = u_i^k v_i^k = u_i v_i & k, j \in \mathcal{I}(n_i) \\ u_i^j v_i^j = 0 & j \notin \mathcal{I}(n_i) \end{cases}$$

et donc

$$\sum_{j=1}^{p} u_i^j v_i^j = \mid \mathcal{I}(n_i) \mid u_i v_i$$

□

Finalement, l'algorithme du produit scalaire parallélisé est défini par:

Algorithme 4.9 (Produit scalaire parallélisé)

```
(1)   procédure ParaProduit(u^j : vecteur, v^j : vecteur) : réel;
(2)     calculer    α_j = u^j ⊙ v^j;
(3)     envoyer     α_j à tout les les processeurs
(4)     α = α_j
(5)     pour tout (k ∈ {1,...,p} ∧ k ≠ j) faire
(6)       recevoir  α_k
(7)       α = α + α_k
(9)     retourner α;
(10)  fin procédure.
```

Avec cet algorithme, si u et v sont cohérents, alors nous avons $\alpha = u \cdot v$ et α est cohérent. □

4.6 La structure de l'environnment

Mise à part le module qui sert d'interface avec l'utilisateur, l'environnement se compose principalement de trois modules (voir la figure 4.5):

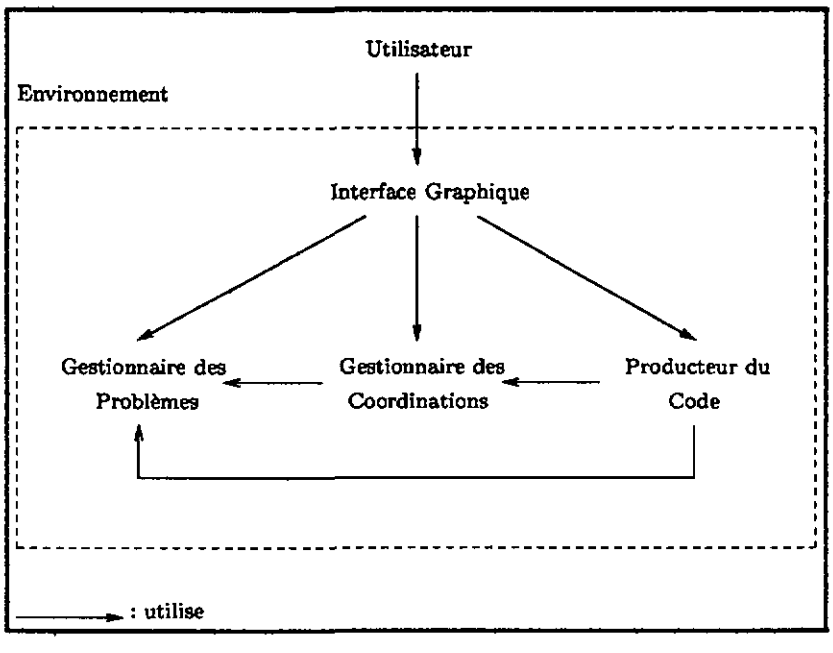

Figure 4.5 : structure de l'environnement

1. **Gestionnaire des problèmes.** Il permet de spécifier les différentes composantes d'un problème: le domaine, les éléments finis, l'opérateur, les conditions aux limites et l'algorithme numérique à utiliser.

2. **Gestionnaire des coordinations.** Après chaque transformation d'un problème, il crée la coordination adéquate et établit le lien entre ce problème et ses problèmes auxiliaires. Ces liens servent à générer un arbre de dépendances entre les problèmes.

3. **Producteur du code.** A partir de l'arbre de dépendances entre les problèmes, ce module génère le code qui mettra en oeuvre les algorithmes numériques et les coordinations spécifiées par l'utilisateur. Dans le cas d'une programmation parallèle, il répartit les problèmes sur les différents processeurs.

Ces trois modules ont été mis en place en utilisant la programmation orientée objets.
Nous allons décrire pour chacun de ces modules les principaux objets qui le composent et leurs interactions avec les autres modules.

4.6.1 Gestionnaire des problèmes

L'objet principal de ce module est l'objet *Probleme* qui consiste en un assemblage de quatre objets (voir la figure 4.6):

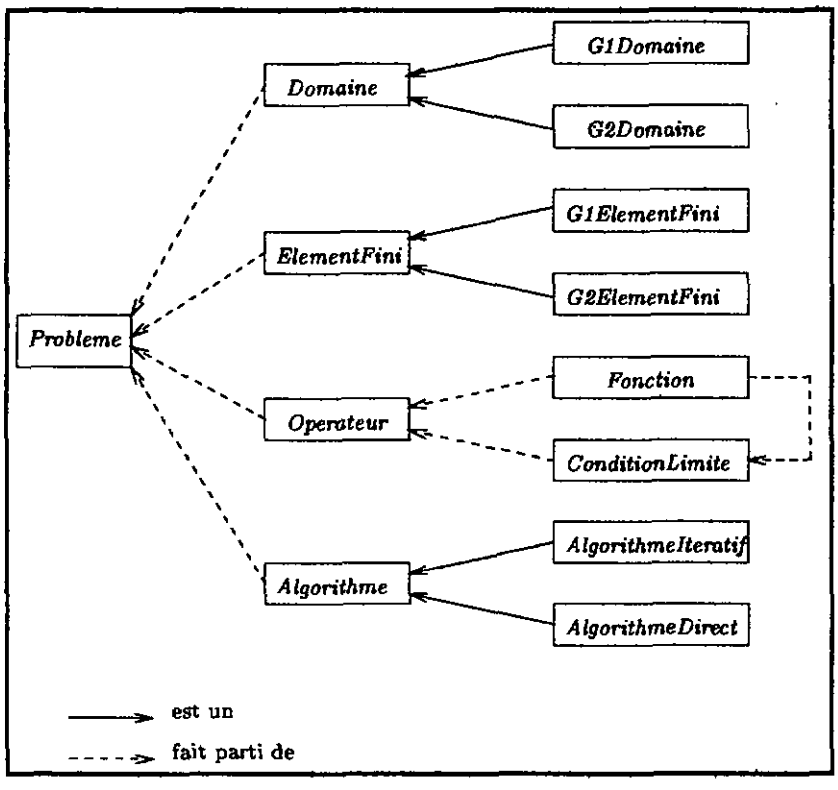

Figure 4.6 : gestionnaire des problèmes

1. Objet *Domaine.* Il permet de décrire la structure géométrique du domaine physique utilisé et sert à l'héritage des objets *G1Domaine* et *G2Domaine* qui sont spécifiques aux domaines de dimension un et de dimension deux. L'utilisation de cet objet intervient lors

de la discrétisation en donnant le type d'éléments finis et aussi lors de la transformation géométrique d'un domaine telle que le découpage, l'enveloppement, le grossissement de maillage etc.

2. Objet *ElementFini*. Il décrit le type d'éléments finis utilisés et sert à définir les fonctions de base pour l'assemblage de la matrice du problème (voir section 1.2.3). Il est aussi l'objet de base pour l'héritage des objets *G1ElementFini* et *G2ElementFini* respectivement dans le cas de dimension un et de dimension deux. D'autres objets sont définis suivant la géométrie (*i.e.*, triangle, rectangle etc.) et le degré de leurs fonctions de base (*i.e.*, linéaire, quadratique, etc.).

3. Objet *Algorithme*. Il décrit l'algorithme numérique utilisé pour la résolution du système d'équations algébriques (voir section 4.3.2). Il peut être itératif (objet *AlgorithmeIteratif*) ou direct (objet *AlgorithmeDirect*).

4. Objet *Operateur*. Il décrit l'opérateur elliptique utilisé (voir section 4.3.1) sous forme d'une matrice de fonctions – objet *Fonction* – et une ou plusieurs conditions aux limites – objet *ConditionLimite*.

Ce gestionnaire des problèmes peut recevoir des requêtes de création de problèmes à la fois de l'utilisateur et du gestionnaire des coordinations. Dans le premier cas, grâce à l'interface graphique, l'utilisateur peut spécifier les différentes composantes du problème. Dans le deuxième cas, c'est lors de la transformation d'un problème que le gestionnaire des problèmes crée automatiquement les problèmes auxiliaires.

4.6.2 Gestionnaire des coordinations

L'objet de base utilisé dans ce module est l'objet *Coordination*. Il permet de définir la relation entre un problème et ses problèmes auxiliaires. Nous avons défini les quatre variantes suivantes de l'objet *Coordination* (voir la figure 4.7):

- *Subdivision*: pour une subdivision avec recouvrement.
- *Reduction*: pour une subdivision avec réduction du problème aux frontières.
- *Enveloppement*: pour un enveloppement d'un domaine.
- *GrossSansRecouvrement*: pour un grossissement de maillage sans recouvrement.
- *GrossAvecRecouvrement*: pour un grossissement de maillage avec recouvrement.

Lorsque l'utilisateur choisit une transformation d'un problème, le gestionnaire de coordination fait appel au gestionnaire des problèmes pour la création des problèmes auxiliaires et attache à la fois le problème original et les nouveaux problèmes à la coordination.

Comme un problème auxiliaire peut subir à son tour une transformation, le gestionnaire des coordinations assure la cohérence entre ces transformations successives en créant un arbre de dépendances entre les problèmes.

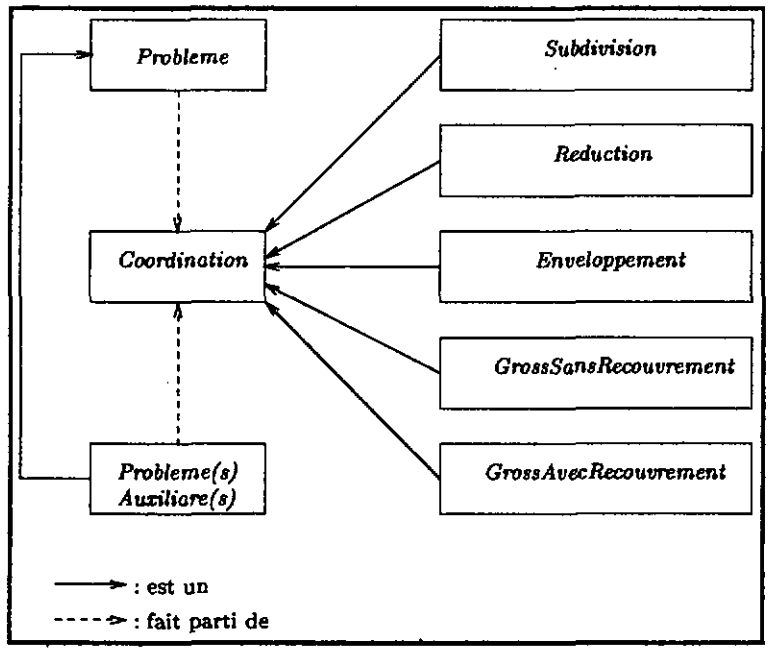

Figure 4.7 : gestionnaire des coordinations

4.6.3 Producteur du code

Ici, l'objet principal est l'objet *Processus* qui reflète la partie du programme destinée à être exécutée dans un processeur. Dans le cas d'une programmation parallèle, chaque processus reçoit un sous-domaine – objet *SousPartition* – du domaine partagé entre les processeurs (voir la figure 4.8). Chacun de ces sous-domaines définit un sous-problème – objet *SousProbleme* – qui n'est que la restriction du problème global à ce sous-domaine (voir section 4.5). Pour échanger les données entre les processus, nous distinguons deux types de communications:

1. Inter-communications (objet *InterComm*): où il s'agit d'échanger les données pour l'élaboration des opérations algébriques parallélisées.

2. Intra-communications (objet *IntraComm*): où il s'agit d'échanger les données entre un problème et ses problèmes auxiliaires. Si un problème est auxiliaire d'un autre problème, tous les sous-domaines sont rattachés aux sous-domaines du problème original grâce à l'objet *CoordHaut*. De la même manière, si ce problème admet des problèmes auxiliaires, nous définissons l'objet coordination – objet *CoordBas* – qui décrit la dépendance entre ses sous-domaines et les sous-domaines des problèmes auxiliaires.

De cette manière, le producteur du code génère un arbre de dépendance entre les sous-domaines – donc entre les processus – à partir de l'arbre de dépendances entre les problèmes.

Ce nouvel arbre est utilisé par la suite pour la production des programmes destinés aux différents processeurs.

4.7 Conclusions

Les tests de notre approche ont été effectués sur des domaines à deux dimensions, car nous n'avons pas de librairies permettant d'effectuer les transformations géométriques d'une

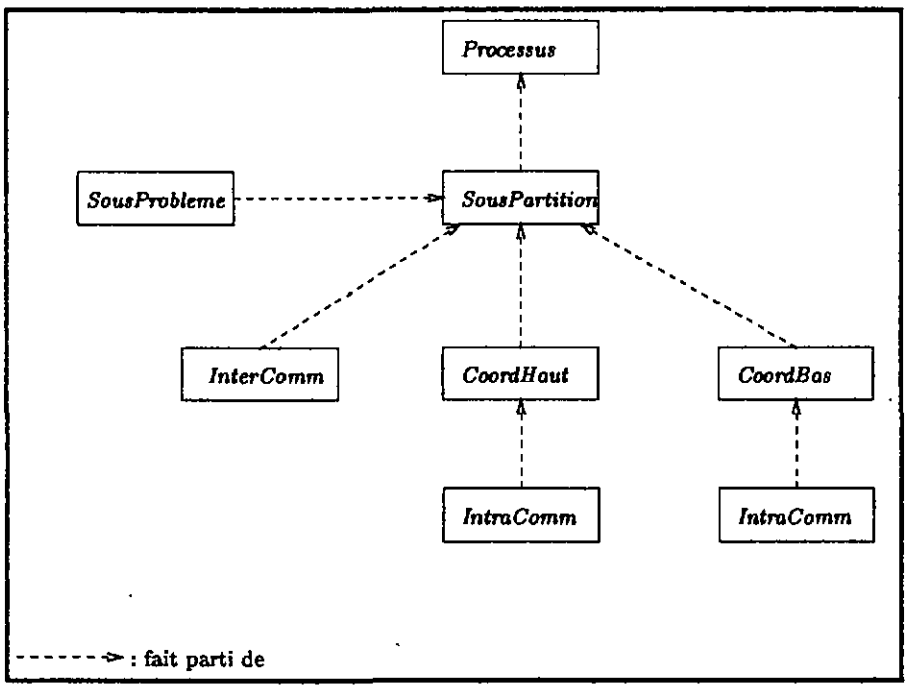

Figure 4.8 : producteur du code

manière interactive dans le cas à trois dimensions. Les logiciels commercialisés ne sont pas délivrés avec leurs sources pour pouvoir intégrer notre approche.

Nous avons dû programmer une librairie qui permet de faire ces transformations en nous restreignant à deux dimensions, notre objectif prioritaire étant de montrer la faisabilité de notre approche.

L'environnement (ou plutôt un prototype de l'environnement) tourne actuellement sur une machine SiliconGraphics avec le langage C++ et utilise un réseau d'une dizaine de stations de travail (Sun et SiliconGraphics) pour l'exécution des programmes.

L'environnement ne permet pas de contrôler si les transformations sont valides ou si elles ont un sens mathématique. Son rôle est surtout d'être "une plate-forme d'expériences" de laquelle peuvent naître de nouvelles méthodes.

L'utilisation de la programmation orientée objets permet d'ajouter de nouveaux opérateurs (EDP), de nouvelles méthodes numériques, de nouvelles coordinations, etc. Tout cela contribue à créer un laboratoire "virtuel" destiné aux simulations numériques.

Conclusions

Dans ce rapport nous avons présenté les connaissances de base qu'un informaticien doit avoir pour pouvoir à la fois dialoguer avec un numéricien et mettre en place un programme pour la simulation numérique. Ensuite, nous avons décrit, dans les deux derniers chapitres, notre contribution – en tant qu'informaticien – à la programmation des méthodes de décomposition de domaine (MDD).

Notre travail avait pour but de faciliter la programmation de ces méthodes. Pour cela, nous avons examiné les variantes les plus connues, présenté un concept pour unifier leurs rôles et développé un environnement qui permet de les exploiter.

En introduisant la notion de problèmes auxiliaires et de coordinations (voir chapitre 3), nous avons pu redonner une nouvelle définition aux MDD; à savoir, une coordination entre un problème de départ et ses problèmes auxiliaires. Une MDD est définie par le type de coordination et par les problèmes auxiliaires utilisés. Cette définition permet d'unifier les différentes MDD en un concept simple qui constitue le coeur de notre environnement.

Dans le chapitre 4, nous avons montré comment ce concept peut être intégré dans un environnement pour la simulation numérique et nous avons donné les algorithmes pour la mise en oeuvre des différents types de coordinations. Dans le cas d'une programmation parallèle, nous avons décrit les algorithmes des opérations algébriques parallélisées que l'environnement utilise.

Nous nous sommes restreints à des domaines de deux dimensions et à des problèmes elliptiques et linéaires. L'utilisation des domaines tridimensionnels peut être faite en intégrant un éditeur graphique adéquat. Quant aux problèmes elliptiques non linéaires, nous avons été limités par le manque des études théoriques sur l'application de ces MDD. À ce jour, seul le cas linéaire est maîtrisé.

Actuellement les problèmes non linéaires sont résolus en une suite de problèmes linéaires (*i.e.*, linéarisation). Notre environnement peut contribuer à l'accélération des algorithmes itératifs utilisés pour la résolution de ces problèmes linéaires.

Nous avons choisi la programmation orientée objets parce que notre concept est une abstraction de la plupart des MDD. Cela a facilité la mise en oeuvre de toutes les variantes des MDD que nous avons vues au chapitre 2.

Pour conclure, nous avons opté pour un environnement dont l'utilisateur n'est pas forcément un informaticien. Ce choix nous a certainement empêché d'élargir les champs d'application de cet environnement. Nous pensons, par exemple, à des nouveaux types de coordinations définis par l'utilisateur et non imposés par l'outil, ou à des opérateurs qui ne sont pas définis au moment de la phase de spécifications du problème mais générés par l'utilisateur au cours de l'exécution de son programme. Pour remédier à ces lacunes, une autre direction consistant à reprendre notre proposition d'unification et à créer une librairie informatique orientée objet en utilisant les "templates" est une des solutions possibles. Dans

ce cas, grâce à cette librairie, les programmeurs des MDD ne seraient pas restreints par les limites que leur imposent notre environnement.

Bibliographie

S. F. Ashby, T. A. Manteuffel et P. E. Saylor (1990). A taxonomy for conjugate gradient methods. *SIAM J. Numer. Anal.*, 27:1542–1568.

F. S. Barry, P. E. Bjørstad et W. D. Gropp (1996). *Domain Decomposition. Parallel Multilevel Methods For Elliptic Partial Differential Equations*. Cambridge University Press, Cambridge.

P. E. Bjørstad et O. B. Widlund (1986). Iterative methods for the solution of elliptic problems on regions partitioned into substructures. *SIAM J. Numer. Anal.*, 23:1097–1120.

C. Börgers (1989). The Neumann–Dirichlet domain decomposition method with inexact solvers on the subdomains. *Numer. Math.*, 55:123–136.

C. Börgers (1990). Domain imbedding methods for the Stokes equations. *Numer. Math.*, 57:435–452.

J. H. Bramble (1993). *Multigrid Methods*, tome 294 de *Pitman Research Notes in Mathematical Sciences*. Longman Scientific & Technical, Essex, England.

J. H. Bramble, J. E. Pasciak et A. H. Schatz (1989). The construction of preconditioners for elliptic problems by substructuring, IV. *Math. Comp.*, 53:1–24.

J. H. Bramble, J. E. Pasciak et J. Xu (1990). Parallel multilevel preconditioners. Dans *Third International Symposium on Domain Decomposition Methods for Partial Differential Equations*, edité par T. F. Chan, R. Glowinski, J. Périaux et O. B. Widlund, pages 341–357. SIAM, Philadelphia.

F. Brezzi et M. Fortin (1991). *Mixed and Hybrid Finite Element Methods*. Springer–Verlag, New York.

P. G. Ciarlet (1978). *The Finite Element Method for Elliptic Problems*. North–Holland, Amsterdam.

Y.-H. de Roeck et P. Le Tallec (1991). Analysis and test of a local domian decomposition. Dans *Fourth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, edité par R. Glowinski, Yu. A. Kuznetsov, G. A. Meurant, J. Périaux et O. B. Widlund, pages 112–128. SIAM, Philadelphia.

G. Dhatt et G. Touzot (1984). *Une présentation de la méthode des éléments finis*. Maloine S.A, Paris, deuxième édition.

Q. V. Dinh *et al.* (1992). Lagrange multiplier approach to fictitious domain methods: application to fluid dynamics and eletro-magnetics. Dans *Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, edité par D. E. Keyes, T. F. Chan, G. Meurant, J. S. Scroggs et R. G. Voigt, pages 151–194. SIAM, Philadelphia.

Q. V. Dinh, R. Glowinski et J. Périaux (1984). Solving elliptic problems by domain decomposition methods with applications. Dans *Elliptic Problem Solvers II*, edité par G. Birkhoff et A. Schoenstadt. Academic Press.

M. Dryja et O. B. Widlund (1990). Towards a unified theory of domain decomposition algorithms for elliptic problems. Dans *Third International Symposium of Domain Decomposition Methods for Partial Differential Equations*, edité par T. F. Chan, R. Glowinski, J. Périaux et O. B. Widlund, pages 3–21. SIAM, Philadelphia.

M. Dryja et O. B. Widlund (1992). Additive Schwarz methods for elliptic finite element problems in three dimensions. Dans *Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, edité par D. E. Keyes, T. F. Chan, G. Meurant, J. S. Scroggs et R. G. Voigt, pages 3–18. SIAM, Philadelphia.

M. Dryja et O. B. Widlund (1994). Domain Decomposition Algorithms with small overlap. *SIAM J. Sci. Comput.*, 15:604–620.

M. Fortin et R. Glowinski (1983). *Augmented Lagrangian Methods*. North-Holland, Amsterdam.

I. Fried (1971). Condition of finite element matrices generated from nonuniform meshes. *AIAA J.*, 10:219–221.

G. H. Golub et C. F. van Loan (1989). *Matrix Computations*. Johns Hopkins Univ. Press, Baltimore, MD, deuxième édition.

M. Griebel et V. Thurner (1993). The Efficient Solution of Fluid Dynamics problems by the Combination Technique. Rapport technique TUM-I9301, SFB-Bericht Nr.342/1/93 A, Institut für Informatik, Technische Universität München.

L. A. Hageman et D. M. Young (1981). *Applied Iterative Methods*. Academic Press, New York.

G. Karypis et V. Kumar (1995). A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. Rapport technique 95-035, University of Minnesota, Department of Computer Science, Minneapolis.

P. Le Tallec (1994). Domain Decomposition methods in computational mechanics. *Computational Mechanics Advances*, 1:121–220.

P.-L. Lions (1988). On the Schwarz Alternating Method I. Dans *First International Symposium on Domain Decomposition Methods for Partial Differential Equations*, edité par R. Glowinski, G. H. Golub, G. A. Meurant et J. Périaux, pages 1–42. SIAM, Philadelphia.

J. Mandel (1993). Balancing domain decomposition. *Comm. Numer. Meth. Engrg.*, 9:233–241.

L. D. Marini et A. Quarteroni (1989). A relaxation procedure for domain decomposition methods using finite elements. *Numer. Math.*, 55:575–598.

A. M. Matsokin et S. V. Nepomnyaschikh (1985). A Schwarz alternating method in subspaces. *In. Vuzov*, 10:61–66. Also in Soviet Mathematics, 10 (1985), pp. 78–84.

S. V. Nepomnyaschikh (1992). Decomposition and fictitious domains methods for elliptic boundary value problems. Dans *Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, edité par D. E. Keyes, T. F. Chan, G. Meurant, J. S. Scroggs et R. G. Voigt, pages 62–72. SIAM, Philadelphia.

S. Oualibouch et N. El Mansouri (1997). Proximal-Jacobi Domain Decomposition Algorithm and application to linear and nonlinear elliptic problems. Dans *Eight International Conference on Domain Decomposition Methods in Science and Engineering*, edité par R. Glowinski, J. Périaux, Z-C. Shi et O. Widlund, pages 91–98. John Wiley & Sons, West Sussex, UK.

O. Pironneau (1988). *Méthodes des éléments finis pour les fluides*. Masson, Paris.

A. Quarteroni et A. Valli (1994). *Numerical Approximation of Partial Differential Equations*. Springer–Verlag, Berlin.

P.-A. Raviart et J.-M. Thomas (1977). A mixed finite element method for second order elliptic problems. Dans *Mathematical Aspects of the Finite Element Method*, edité par A. Dold et B. Eckmann, tome 606 de *Lecture Notes in Mathematics*, pages 292–315. Springer–Verlag, Heidelberg.

Y. Saad et M. H. Schultz (1986). GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7:856–869.

U. Schumann, rédacteur (1978). *Fast Elliptic Solvers*. Advance Publications, London.

J.-P. Shao (1993). *Domain Decomposition Algorithms*. Thèse de doctorat, University of California, Los Angeles.

B. F. Smith (1992). An optimal domain decomposition preconditioner for the finite element solution of linear elasticity problems. *SIAM J. Sci. Stat. Comput.*, 13:364–378.

J. Xu (1989). *Theory of Multilevel Methods*. Thèse de doctorat, Cornell University.

J. Xu (1992). Iterative Methods by Space Decomposition and Subspace Correction. *SIAM Review.*, 34:581–613.

H. Yserentant (1986). On the multi–level splitting of finite element spaces. *Numer. Math.*, 49:379–412.