Multiprogrammation parallèle générique des méthodes de décomposition de domaine

Andréa Schwertner-Charão

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

T H È S E

pour obtenir le grade de

DOCTEUR DE L'INPG

***Spécialité :* « INFORMATIQUE : SYSTÈMES ET COMMUNICATIONS »**

préparée au **LABORATOIRE INFORMATIQUE ET DISTRIBUTION**

dans le cadre de l'École Doctorale

« MATHÉMATIQUES, SCIENCES ET TECHNOLOGIE DE L'INFORMATION »

présentée et soutenue publiquement

par

Andréa Schwertner CHARÃO

le 20 Septembre 2001

Multiprogrammation parallèle générique des méthodes de décomposition de domaine

Directeur de thèse :

Brigitte PLATEAU

Composition du Jury :

M. Jacques MOSSIÈRE,	*Président*
M. Jacques BAHI,	*Rapporteur*
M. Yves ROBERT,	*Rapporteur*
Mlle. Isabelle CHARPENTIER,	*Co-encadrant*
Mme. Brigitte PLATEAU,	*Directeur de thèse*

Remerciements

Je tiens tout d'abord à exprimer ma profonde gratitute à Isabelle Charpentier et Brigitte Plateau, qui m'ont encadrée et soutenue tout au long de ce travail. Leur esprit scientifique, leur clairvoyance et leur dynamisme hors pair font d'elles deux personnes que j'admire et respecte beaucoup. Au-dessus des liens professionnels, leurs qualités humaines m'ont particulièrement marquée et m'ont fait apprécier très sincèrement ces années de travail conjoint.

Je remercie aussi les autres membres de mon jury de soutenance : Monsieur Jacques Mossière pour l'avoir présidé, Messieurs Jacques Bahi et Yves Robert pour avoir rapporté sur ce travail de thèse. Leurs remarques et commentaires m'ont été très précieux, non seulement pour me permettre d'améliorer la qualité et la clarté de ce document, mais aussi en ce qui concerne les perspectives et débouchés de cette thèse.

Mes remerciements vont également à l'ensemble des membres passés et présents du Laboratoire Informatique et Distribution, pour l'excellente atmosphère qu'ils contribuent à maintenir. Je risque de commettre un oubli impardonnable, mais j'adresse quand même un merci tout particulier à Alexandre C., Gerson, Grégory, Gustavo, Hélène, Jacques C2K, Joëlle, Maurício, Nicolas, Olivier B., Roberta et Yves D., avec qui j'ai beaucoup apprécié de parcourir un bout de chemin.

Je ne pourrais pas non plus oublier les personnes qui m'ont accueillie au sein de l'équipe EDP du Laboratoire de Modélisation et Calcul pendant les derniers mois de rédaction de cette thèse. Un grand merci donc à Delia, Ioana, Isabelle (encore !), Philippe, Sylvia, Valérie et Yunqing pour leur soutien, leurs conseils et leur amitié. Je garderai pour toujours un très bon souvenir du septième et des moments plus qu'agréables passés avec eux à la Tour IRMA.

Enfin, je remercie Benhur pour m'avoir, de près comme de loin, supportée et encouragée pendant ces années. Je remercie également mes parents et amis qui ont toujours été là quand j'ai eu besoin. Un merci très spécial à Gustavo, Kelly, Maurício, Nicolas et Tati, dont l'amitié me sera toujours très précieuse.

Un dernier mot de remerciement va pour l'agence CAPES du gouvernement brésilien, pour m'avoir accordé l'aide financière sans laquelle cette thèse n'aurait pas pu voir le jour.

Table des matières

I Problématique et Techniques **19**

1 Le calcul parallèle **21**
1.1 Architectures pour le calcul parallèle 23
1.1.1 Architectures à mémoire partagée 23
1.1.2 Architectures à mémoire distribuée 24
1.1.3 Architectures hybrides 25
1.2 Programmation des machines parallèles 26
1.2.1 Méthodologie de parallélisation 26
1.2.2 Modèles de programmation parallèle 29
1.3 Multiprogrammation légère pour le calcul parallèle 32
1.3.1 Les processus légers en bref 33
1.3.2 Le bénéfice à tirer des processus légers 34
1.4 L'environnement de programmation ATHAPASCAN 35
1.4.1 Le noyau exécutif : ATHAPASCAN-0 36
1.4.2 L'interface de programmation haut niveau : ATHAPASCAN-1 37
1.5 Conclusion 38

2 La simulation numérique en parallèle **39**
2.1 Discrétisation de problèmes d'EDP 41
2.1.1 Le problème modèle et son analyse mathématique 41
2.1.2 Maillage du domaine physique 43
2.1.3 Formulation matricielle 44
2.1.4 Résolution du système 45
2.2 Adaptation de maillage 46
2.3 Méthodes de décomposition de domaine 48
2.3.1 Partitionnement de maillage 49
2.3.2 Méthodes avec recouvrement 50
2.3.2.1 Méthode de Schwarz multiplicative 51
2.3.2.2 Méthode de Schwarz additive 52

2.3.2.3 Comparaison entre Schwarz additive et Schwarz multiplicative . . . 52
2.3.2.4 Mise en œuvre parallèle . . . 53
2.3.3 Méthodes sans recouvrement . . . 54
2.3.3.1 Méthode du complément de Schur ou Schur primal 56
2.3.3.2 Méthode de Schur dual . . . 58
2.3.3.3 Remarques sur les méthodes sans recouvrement . 60
2.3.3.4 Méthode des joints . . . 61

II Solutions 63

3 Outils pour la résolution d'EDP en parallèle : état de l'art 65
3.1 Des approches complémentaires . . . 66
3.1.1 Outils issus des mathématiques . . . 66
3.1.1.1 Bibliothèques pour la résolution de systèmes linéaires . . . 67
3.1.1.2 Environnements pour la résolution numérique d'EDP en parallèle . . . 68
3.1.2 Outils informatiques . . . 69
3.1.2.1 Le paradigme objet pour le calcul scientifique numérique . . . 70
3.1.2.2 Outils pour la gestion de structures de données distribuées . . . 71
3.2 Étude de quelques outils représentatifs . . . 72
3.2.1 PSPARSLIB . . . 72
3.2.1.1 Structures de données . . . 72
3.2.1.2 Algorithmes parallèles . . . 75
3.2.1.3 Appréciation de l'outil . . . 76
3.2.2 POOMA . . . 77
3.2.2.1 Structures de données . . . 77
3.2.2.2 Algorithmes parallèles . . . 79
3.2.2.3 Appréciation de l'outil . . . 80
3.3 Présentation de AHPIK . . . 80

4 Structure du parallélisme pour la décomposition de domaine 83
4.1 Construction du graphe de tâches . . . 84
4.1.1 Identification des tâches : le cas général . . . 85
4.1.2 Coûts des tâches . . . 85
4.1.3 Interactions entre les tâches . . . 86
4.1.4 Placement des tâches . . . 87

4.1.5 Synchronisations 87
4.2 Extensions et cas spécifiques 88
4.2.1 Schémas asynchrones 88
4.2.2 Résolution globale par une méthode de Krylov 91
4.3 Mise en œuvre des processus légers dans le noyau de AHPIK 94
4.3.1 Motivations 94
4.3.2 Conception de l'interface 94
4.3.3 Écriture orientée objet 96
4.3.3.1 Affectation de tâches aux processus légers principaux 97
4.3.3.2 Affectation de données aux liens communicants . 99
4.3.3.3 Opérations nécessitant des communications collectives 101
4.3.3.4 Lancement des tâches de calcul d'interface 103
4.3.4 Mise en œuvre parallèle avec ATHAPASCAN-0 104
4.3.5 Mise en œuvre parallèle avec MPI 105

5 L'environnement objet de AHPIK 107
5.1 Objets pour les méthodes de décomposition de domaine 109
5.1.1 Classes caractérisant les problèmes d'EDP 109
5.1.2 Composants géométriques de la décomposition du maillage 110
5.1.3 Support au développement de solveurs 111
5.1.4 Les calculs spécifiques à chaque solveur 112
5.2 Mise en œuvre parallèle 114
5.3 Schémas parallèles génériques pour la décomposition de domaine . 115
5.3.1 Composition d'un schéma 115
5.3.1.1 Schéma pour solveurs reposant sur une méthode de point fixe 116
5.3.1.2 Schéma pour solveurs de type gradient conjugué . 118
5.3.2 Les différents schémas et leur représentation graphique . . . 121
5.4 Les méthodes de décomposition de domaine implantées 129
5.4.1 Schwarz additif synchrone 129
5.4.2 Schwarz additif asynchrone 131
5.4.3 Schur primal, méthode de point fixe 132
5.4.4 Schur dual, méthode de point fixe 133
5.4.5 Schur dual, méthode du gradient conjugué 135
5.4.6 Méthode des éléments joints 138

III Expérimentation **139**

6 Études expérimentales de problèmes d'EDP classiques **141**
6.1 Étude expérimentale du comportement de convergence des méthodes numériques . . . 142
6.1.1 Rapport d'aspect et décomposition géométrique . . . 143
6.1.2 « Scalabilité » de Schur dual . . . 145
6.2 Déséquilibres de charge et recouvrement : expériences avec un problème de convection-diffusion . . . 146
6.3 Aspects de la multiprogrammation d'une application réelle . . . 149
6.3.1 Un problème d'écoulement de fluide . . . 149
6.3.2 Description du code original . . . 151
6.3.3 Portage de l'application avec AHPIK . . . 154
6.3.4 Résultats numériques . . . 156
6.4 Conclusion . . . 157

A Méthode des éléments finis **163**
A.1 Maillage du domaine physique . . . 163
A.2 Définition des éléments finis . . . 165
A.3 Assemblage du système . . . 169
A.4 Prise en compte des conditions aux limites . . . 171

B Algorithmes de raffinement de maillage **173**
B.1 Adaptation par découpage . . . 174

C La méthode de Schur dual **177**
C.1 Réecriture du problème . . . 177
C.2 Schur dual . . . 177
C.3 Différence entre les 2 méthodes de Schur . . . 178

D La méthode du gradient conjugué **181**

Table des figures

1.1 Programmation classique vs. multiprogrammation d'un programme composé de tâches indépendantes. 35

2.1 (a) Maillage conforme, (b) Maillage non-conforme. 43
2.2 Exemple d'adaptation statique : maillage autour d'un profil d'aile d'avion. 47
2.3 Exemples de singularités. 48
2.4 Couplages de modèles. 49
2.5 Décomposition avec recouvrement. 51
2.6 Représentation graphique de la convergence de la méthode de Schwarz. 53
2.7 Importance du recouvrement pour la méthode de Schwarz. 54
2.8 Décomposition sans recouvrement. 55

3.1 (a) Profil d'une matrice creuse à 12 inconnues. (b) Graphe d'adjacence associé. 73
3.2 Partitionnement d'un graphe en 2 sous-graphes. 73
3.3 Vision globale d'une matrice creuse distribuée selon PSPARSLIB. . 75
3.4 Organisation modulaire de AHPIK. 81

4.1 Graphe de tâches pour une décomposition en trois sous-domaines. . 86
4.2 Diagramme de Gantt pour une décomposition en deux sous-domaines. 88
4.3 Liens communicants entre deux processeurs voisins. 95

5.1 Organisation de AHPIK. 108
5.2 Hiérarchie de classes pour les solveurs classiques de systèmes linéaires 110
5.3 Hiérarchie de classes pour la représentation des sous-domaines . . . 111
5.4 Déclaration d'une classe dérivée d'un schéma parallèle AHPIK . . . 112
5.5 Exemple d'implémentation de `LocalComputation`. 113
5.6 Schéma parallèle générique pour une décomposition en 3 sous-domaines 115
5.7 Schéma parallèle typique d'une méthode de point fixe. 117
5.8 Schéma parallèle typique d'une méthode de gradient conjugué. . . . 120

5.9 Exemple de code pour une tâche de calcul d'interface. 121
5.10 Visualisation du schéma de point fixe synchrone pour la méthode d'Uzawa : cas parfaitement équilibré. 122
5.11 Visualisation du schéma de point fixe synchrone pour la méthode d'Uzawa : déséquilibre de charge provenant de l'utilisation d'un solveur local itératif. 123
5.12 Visualisation d'exécution pour une décomposition en 4 sous-domaines : contrôle de convergence synchrone classique. 125
5.13 Visualisation d'exécution pour une décomposition en 4 sous-domaines : contrôle de convergence asynchrone. 126
5.14 Diagramme illustrant l'exécution de la méthode de Schwarz additif synchrone. 127
5.15 Visualisation d'exécution de la version asynchrone de la méthode de Schwarz additif. 128
5.16 Diagramme d'exécution avec deux sous-domaines par nœud de calcul. 130
5.17 Communications pour la méthode de Schwarz : distinction des nœuds correspondant aux frontières de recouvrement. 131
5.18 Schur dual, méthode point fixe : « zoom » sur les itérations. 135
5.19 Schur dual, méthode du gradient conjugué : « zoom » sur les itérations. 137

6.1 Décompositions en (a) bandes verticales, rapport d'aspect égal à ν et (b) blocs rectangulaires, rapport d'aspect égal à 1 ou 2. 144
6.2 Nombre d'itérations à l'intérieur des sous-domaines au cours des itérations de décomposition de domaine. Les décompositions considérées sont : (a) 2 sous-domaines, (b) 4 sous-domaines, (c) 8 sous-domaines et (d) 16 sous-domaines. 148
6.3 Pourcentages de variation du nombre d'itérations pour la résolution à l'intérieur des sous-domaines 148
6.4 Géométrie et conditions aux limites pour le problème d'écoulement autour du cylindre. 150
6.5 Décomposition de domaine du cylindre. 150
6.6 Code C++ correspondant à une tâche d'interface : relèvement sur le côté esclave de l'interface. 155
6.7 Résultats pour une distribution équitable des sous-domaines. 157
6.8 Résultats pour le code adaptatif. 158

A.1 (a) Maillage conforme, (b) Maillage non-conforme. 164
A.2 Maillage d'un domaine rectangulaire. 166
A.3 Élément fini P_1. 167
A.4 Représentation graphique des fonctions de base pour un domaine bidimensionnel. 168
A.5 Profil de la matrice de rigidité pour le problème de Poisson. 170

B.1 Raffinement par découpage. (a) Triangle à raffiner. (b) Premier raffinement. (c) Deuxième raffinement. 175

Liste des tableaux

6.1 Nombre d'itérations pour la convergence et temps de restitution pour la méthode d'Uzawa en séquentiel sur les décompositions en bandes et en blocs. 144
6.2 Temps de restitution parallèle pour les décompositions en bandes et en blocs. 144
6.3 Temps de restitution, problèmes de taille croissante. 146
6.4 Accélérations obtenues pour des différentes décompositions et différents schémas de placement 147

Introduction

La simulation numérique est un outil essentiel pour le développement technologique de disciplines très variées comme la mécanique des fluides, le calcul de structures et la météorologie. Un phénomène physique peut se traiter numériquement par la discrétisation du système d'Équations aux Dérivées Partielles (EDP) qui le modélise, le but étant d'obtenir un système matriciel linéaire. Numériquement, la précision d'une solution dépend de la finesse de la discrétisation utilisée ce qui influence fortement la taille du système à résoudre. En conséquence, les temps de calcul peuvent être de quelques milliers d'heures sur un ordinateur conventionnel lors de la simulation de problèmes en vraie grandeur. Les implémentations informatiques construites sur ce principe sont ainsi de grandes consommatrices de ressources de calcul et de mémoire.

Dans ce contexte, le calcul parallèle est de plus en plus utilisé comme une alternative permettant d'obtenir des diminutions substantielles du temps de restitution des calculs et/ou d'effectuer des études numériques de plus grande précision tout en gardant un temps de calcul raisonnable. Les possibilités de calcul parallèle sont aujourd'hui présentes à des degrés divers dans les systèmes informatiques. Ce peut être via des machines spécialement conçues pour ce type de calcul (CRAY T3E, SGI PowerChallenge, IBM SP), des serveurs ou stations multiprocesseurs telles que celles offertes par la plupart des constructeurs, ou bien une grappe, homogène ou non, de machines précédemment citées. Ces dernières sont de plus en plus fréquemment choisies car, construites en utilisant du matériel produit en grande série, les grappes sont plus abordables que les machines parallèles dédiées. Le problème se pose alors d'offrir au programmeur d'applications critiques les moyens d'exploiter le potentiel de puissance offert par ces architectures parallèles.

Dans le cadre de la résolution de systèmes issus de la discrétisation de problèmes d'EDP, une famille de méthodes se prêtant bien au calcul parallèle est connue sous le nom de **méthodes de décomposition de domaine**[149, 38]. L'idée de base, commune à ces méthodes, repose sur le découpage géométrique du domaine physique en sous-domaines de manière à travailler avec des problèmes d'EDP **locaux**, i.e.

associés aux sous-domaines, plus petits. Pour coupler les problèmes d'EDP posés sur chacun des sous-domaines, la méthode détermine itérativement des conditions aux limites à imposer aux frontières de ces domaines. Cette description succincte fait apparaître le caractère parallèle de la méthode puisque la résolution des sous-problèmes peut s'effectuer sur des processeurs différents, sous réserve d'un échange de données entre processeurs résolvant des sous-domaines "voisins" afin de déterminer/fournir les conditions aux limites. Les méthodes de décomposition de domaine sont un exemple typique de parallélisme à gros grain, se prêtant généralement bien aux architectures parallèles de type grappe car les communications sont peu volumineuses par rapport au volume de calcul.
L'obtention d'un parallélisme efficace lors de l'utilisation d'une méthode de décomposition de domaine dépend fortement d'une distribution équitable de la charge de calcul entre les processeurs. Lorsque les problèmes présentent une certaine régularité (mathématique et géométrique), l'équilibrage de charge est assez simple à réaliser. Par contre, ce n'est plus le cas dès lors que les problèmes mathématiques présentent une singularité ou que la géométrie est complexe.
De plus, dans certaines situations la charge associée à un sous-domaine varie dynamiquement au cours du calcul car les discrétisations peuvent être modifiées localement (plus fines ou plus grossières) en fonction du problème physique. Cette technique d'**adaptation de maillage** permet de les résoudre tout en évitant le coût insupportable en mémoire et temps de calcul d'une discrétisation globalement fine. Il en résulte des implémentations parallèles ayant un comportement dynamique (imprévisible), elles sont dites **irrégulières**[136, 80].

La prise en compte de l'irrégularité ajoute un degré de difficulté à la programmation d'applications parallèles, notamment si l'on souhaite que le programme soit à la fois efficace et portable. Une technique de programmation se prêtant bien à la mise en œuvre d'applications irrégulières est basée sur des réseaux de **processus légers communicants**. Un processus léger (ou « thread », en anglais)[107, 103] est une abstraction permettant d'exprimer la multiprogrammation (i.e. de multiples flots d'exécution indépendants) au sein d'un processus système. Le qualificatif "léger" vient du fait que la gestion de ces entités a un coût très faible par rapport aux processus système. Il est alors intéressant d'utiliser la multiprogrammation légère pour manipuler (démarrer, arrêter, suspendre, déplacer) les tâches d'un programme dynamiquement. Cette technique est également opportune pour recouvrir les temps de communication par du calcul utile. Le paradigme de processus légers communicants permet d'exploiter efficacement les architectures de type grappe de multiprocesseurs symétriques car les processus légers peuvent s'y exécuter en parallèle. Bien que la multiprogrammation légère soit une technique connue depuis longtemps, peu d'applications de simulation numérique en tirent parti.

Dans ce travail de thèse nous nous intéressons à l'apport du paradigme de pro-

cessus légers communicants pour l'exécution parallèle efficace d'applications de simulation numérique reposant sur des méthodes de décomposition de domaine. Ce travail s'inscrit dans une collaboration entre les projets APACHE[1] et IDOPT[2], localisés respectivement au sein du Laboratoire Informatique et Distribution (ID) et du Laboratoire de Modélisation et Calcul (LMC) de l'Institut de Mathématiques Appliquées de Grenoble (IMAG). Les thèmes de recherche du projet APACHE sont centrés sur le développement d'un environnement de programmation parallèle portable et efficace, nommé ATHAPASCAN [26, 44, 128]. Cet environnement comporte un noyau exécutif permettant l'utilisation de la multiprogrammation légère dans un contexte distribuée et une interface applicative implémentant un modèle de programmation parallèle haut niveau, basé sur un mécanisme de création de tâches communiquant entre elles par l'intermédiaire d'une mémoire partagée. Le projet IDOPT s'intéresse à la mise au point d'outils mathématiques et algorithmiques pour la simulation numérique de systèmes issus de la physique et des sciences de l'environnement.

Résultant d'une étroite coopération entre ces domaines, notre travail a une double vocation : d'une part utiliser l'environnement ATHAPASCAN pour la mise en œuvre d'une classe d'applications très représentative du calcul scientifique, d'autre part proposer des solutions informatiques mathématiquement correctes et facilement assimilables par la communauté d'utilisateurs. Dans ce cadre, nos principales contributions sont :

- une étude approfondie de l'apport du modèle de processus légers communicants dans le cadre de la résolution de problèmes d'EDP par des méthodes de décomposition de domaine ;
- un harnais informatique, appelé AHPIK, permettant une programmation aisée d'applications reposant sur les méthodes de décomposition de domaine. Sa conception orientée objet permet d'encapsuler les détails de gestion des processus légers et des communications, ce qui facilite l'implantation de nouvelles méthodes ;
- l'utilisation de cet environnement dans le cadre de la résolution de problèmes d'EDP classiques et notamment pour un problème en mécanique de fluides de grande taille.

Ce travail de thèse a fait objet de plusieurs publications acceptées [39, 40, 41] et un article est en cours d'évaluation [14].

Le manuscrit de thèse est organisé en trois parties. La première partie présente

[1] Algorithmique Parallèle et pArtage de CHargE, projet joint CNRS-INPG-INRIA-UJF.

[2] Identification, Optimisation et Simulation Numérique en Physique et en Environnement, projet joint CNRS-INPG-INRIA-UJF.

des concepts informatiques et mathématiques essentiels. Nous commençons (chapitre 1) par une introduction au calcul parallèle : des architectures parallèles actuelles et de la programmation de celles-ci. Cette partie comprend une introduction à la technique de multiprogrammation légère. Le chapitre 2 présente des généralités liées à la résolution de problèmes d'EDP en mettant l'accent sur les méthodes de décomposition de domaine. La deuxième partie discute de la programmation parallèle de ces méthodes, en présentant l'état de l'art dans le chapitre 3 puis en proposant, dans le chapitre 4, une approche systématique de programmation parallèle et un développement à l'aide de techniques de multiprogrammation légère. Cette partie se conclut par la présentation de l'environnement objet AHPIK dans le chapitre 5. La troisième partie (chapitre 6) présente des résultats expérimentaux.

Première partie

Problématique et Techniques

1

Le calcul parallèle

Sommaire

1.1 Architectures pour le calcul parallèle 23
1.1.1 Architectures à mémoire partagée 23
1.1.2 Architectures à mémoire distribuée 24
1.1.3 Architectures hybrides 25
1.2 Programmation des machines parallèles 26
1.2.1 Méthodologie de parallélisation 26
1.2.2 Modèles de programmation parallèle 29
1.3 Multiprogrammation légère pour le calcul parallèle 32
1.3.1 Les processus légers en bref 33
1.3.2 Le bénéfice à tirer des processus légers 34
1.4 L'environnement de programmation ATHAPASCAN 35
1.4.1 Le noyau exécutif : ATHAPASCAN-0 36
1.4.2 L'interface de programmation haut niveau : ATHAPASCAN-1 37
1.5 Conclusion 38

Les progrès remarquables dans le domaine de la micro-électronique fournissent des microprocesseurs toujours plus puissants, à un rythme souvent supérieur aux prévisions les plus optimistes[94]. Seulement cet accroissement en puissance demande des technologies de plus en plus complexes, engendrant des difficultés qui se font remarquer tant sur le plan technologique que sur le plan économique.

En même temps, les besoins des utilisateurs du calcul haute-performance sont habituellement au-delà des ressources disponibles. En effet, la conquête de la puissance nécessaire pour effectuer des calculs de pointe est naturellement suivie par le lancement de nouveaux challenges. Cela se traduit par des besoins toujours croissants en termes de performance arithmétique, capacité de stockage et rapidité d'accès aux données.

Le calcul parallèle constitue une réponse aux problèmes soulevés ci-dessus. Il consiste à utiliser simultanément les différentes ressources disponibles dans un système informatique (processeurs, mémoires, disques, réseaux de communication, etc.) pour l'exécution des calculs correspondants à une même application. Les motivations pour une telle technique ont à la fois un caractère temporel (travailler plus vite) et spatial (réaliser des plus gros travaux). De plus, le parallélisme peut être vu comme une façon de contourner les barrières technologiques et économiques qui s'imposent au développement de microprocesseurs plus puissants.

L'utilisation de multiples processeurs pour une même application implique une coopération entre eux. Ce fait est à l'origine des grandes difficultés liées au parallélisme : d'un point de vue matériel parce que cela nécessite des technologies permettant un accès rapide à des informations qui ne sont pas localement accessibles à un processeur, et d'un point de vue logiciel car il faut repenser les outils classiques de programmation afin de prendre en compte la coopération entre processeurs.

Dans ce chapitre nous tenons à introduire les concepts de base du calcul parallèle. Nous commençons (section 1.1) par présenter quelques catégories de machines parallèles, avec en particulier une discussion sur les tendances actuelles pour la construction de super-calculateurs. Nous présentons ensuite (section 1.2) quelques notions pour la conception d'algorithmes parallèles et leur implantation sur les différents types de machines. Les modèles les plus courants pour la programmation parallèle sont également évoqués dans cette section. La programmation basée sur processus légers communicants est présentée en détails dans la section suivante (1.3), où l'on met en évidence l'intérêt de cette technique pour la mise en œuvre d'applications irrégulières. Nous finissons (section 1.4) par présenter l'environnement ATHAPASCAN, qui constitue la plate-forme exécutive pour le travail que nous avons effectué.

1.1 Architectures pour le calcul parallèle

Les différentes architectures matérielles se prêtant au calcul parallèle peuvent être classées selon plusieurs critères. Flynn[69] a proposé dans les années 70 une classification fondée sur la multiplicité des flots d'instructions et de données, d'où la distinction des premières machines parallèles en SIMD (machine exécutant une même instruction simultanément sur plusieurs données) et MIMD (machine ayant plusieurs flots d'exécution simultanés sur plusieurs données). Certains auteurs [154, 157] ont par la suite subdivisé la classe MIMD en fonction du degré de couplage entre processeurs et mémoires. On distingue ainsi les multiprocesseurs (ou machines à mémoire partagée) des multiordinateurs (ou machines à mémoire distribuée).

Cette classification générale constitue un repère important lorsque l'on analyse l'évolution des architectures parallèles au fil des années. Dans la suite nous tenons à différencier les architectures à mémoire partagée des architectures à mémoire distribuée, puis nous discutons des tendances actuelles qui mènent à combiner ces deux modèles de mémoire pour la construction de super-calculateurs. Pour plus de détails sur les différentes architectures le lecteur pourra consulter [32].

1.1.1 Architectures à mémoire partagée

Les architectures parallèles à mémoire partagée, aussi appelées multiprocesseurs, sont composées d'un ensemble de processeurs ayant accès à une mémoire commune. La plupart des multiprocesseurs actuellement dans le marché sont des systèmes symétriques (SMP – *Symmetric MultiProcessing*), dans le sens où tous les processeurs ont les mêmes fonctions et sont en compétition pour les ressources système de façon égale.

L'existence d'une mémoire partagée facilite le travail de parallélisation des algorithmes. La mise en œuvre de la coopération entre processeurs est simplifiée car les communications peuvent s'exprimer par des lectures ou écritures de zones de mémoire partagées, se rapprochant de la programmation séquentielle classique. Mais l'accès à la mémoire commune est aussi le point faible de ces architectures. En effet, le couplage fort entre processeur et mémoire fait que l'augmentation du nombre de processeurs augmente aussi la contention d'accès au bus reliant processeurs et mémoires. De ce fait, les architectures à mémoire partagée ne permettent de relier qu'un nombre limité de processeurs, autrement dit elles ne sont pas adaptées pour le parallélisme à grande échelle.

Afin de limiter la charge dans le bus d'interconnexion, les architectures actuelles emploient plusieurs niveaux de mémoires cache, de façon à ce que le temps réel d'accès à une donnée dépende de sa localisation dans la hiérarchie de mémoires. Même avec l'utilisation de ces techniques, les architectures SMP restent aujourd'hui limitées à quelques dizaines de processeurs. Les machines Sun Enterprise 10000 sont actuellement les SMP de plus grande taille dans le marché. Malgré cette contrainte d'extensibilité, les architectures SMP se sont beaucoup répandues, en particulier les configurations à deux ou quatre processeurs qui sont aujourd'hui proposées par la plupart des constructeurs de stations de travail.

1.1.2 Architectures à mémoire distribuée

Les architectures à mémoire distribuée sont constituées d'un ensemble de nœuds, chaque nœud étant composé d'un processeur et d'une mémoire locale. Les nœuds sont connectés entre eux par un réseau de communication, au travers duquel ils peuvent coopérer. Grâce au couplage plus lâche entre processeurs et mémoires, les machines utilisant cette architecture sont plus extensibles à la fois en nombre de processeurs et en capacité mémoire. Cela a permis de construire des machines pouvant employer plus d'une centaine de processeurs. On parle alors d'architectures massivement parallèles (MPP – *Massively Parallel Processing*).

Les technologies d'interconnexion entre les nœuds sont très variées, tant du point de vue du matériel utilisé que des topologies proposées. Avec l'évolution des technologies standard (FDDI, ATM, etc.), les machines à mémoire distribuée ressemblent de plus en plus à des réseaux d'ordinateurs. Avec un bon rapport prix/performances, les grappes (« clusters » en anglais) construites par l'assemblage de matériel produit en grande série sont devenues très populaires.

Si l'absence d'une mémoire commune permet un parallélisme à plus grande échelle, la mise en œuvre d'applications parallèles sur les machines à mémoire distribuée est plus délicate. Aucun support aux accès directs à une mémoire distante n'est a priori fourni par ce type d'architecture, et la coopération entre processeurs est plus coûteuse par rapport aux architectures à mémoire partagée. Ceci est d'autant plus vrai pour les architectures de type grappe. Le programmeur doit par conséquence prendre en compte la localisation des données sur les mémoires locales et s'occuper explicitement de la distribution ou du mouvement de ces données lorsque nécessaire.

1.1.3 Architectures hybrides

La maturité des multiprocesseurs et l'évolution constante des réseaux d'interconnexion ont mis sur le marché des machines combinant les deux types d'architectures précédentes. Ces machines sont constituées d'un ensemble de nœuds multiprocesseurs reliés entre eux par un réseau d'interconnexion haut débit, l'exemple typique étant l'Origin 2000 fabriquée par SGI. Le modèle de mémoire de ces architectures est mixte : mémoire partagée au sein d'un même nœud, et mémoire distribuée entre les différents nœuds. Certaines machines ont une organisation mémoire où chaque nœud peut accéder à des zones mémoire appartenant à d'autres nœuds, mais la mise en œuvre efficace de ces mémoires virtuellement partagées n'est toujours pas bien maîtrisée.

Les architectures hybrides offrent une alternative pour contourner le problème d'extensibilité des machines à mémoire partagée. Elles ont aussi l'avantage d'une puissance de communication supérieure par rapport aux machines à mémoire distribuée, les premières ayant moins de nœuds que les dernières à une puissance de calcul égale (i.e. un même nombre de processeurs).

L'assemblage de multiprocesseurs en réseau s'est montré convenable pour la construction de systèmes de petite à grande taille. Avec l'ample dissémination des machines SMP sur le marché, les grappes et réseaux utilisent des unités à plusieurs processeurs. L'acronyme CluMPs (*Clusters of MultiProcessors* apparaît pour désigner ces grappes de multiprocesseurs. L'architecture des nouvelles machines massivement parallèles devient aussi très proche d'une grappe de SMP, la différence résidant surtout dans l'utilisation de réseaux propriétaires et dans la préservation de l'image d'un système homogène. Les systèmes de plus grande taille construits sur ce principe accommodent aujourd'hui près de dix mille processeurs, avec des performances dépassant le teraflops (10^{12} opérations de point flottant par seconde)[57].

Si les architectures hybrides peuvent mener à des performances sans précédent, elles représentent aussi des nouveaux challenges pour le calcul haute-performance. La mise en œuvre d'applications parallèles exploitant efficacement ce type d'architecture reste délicate puisqu'elle doit prendre en compte les différents niveaux de mémoire de la machine. La plupart des modèles de programmation parallèle classiques a été conçue pour les architectures soit à mémoire partagée soit à mémoire distribuée. Ces modèles correspondent mal à une organisation hybride de la mémoire. Le programmeur peut donc être amené à utiliser simultanément deux modèles de programmation différents.

1.2 Programmation des machines parallèles

L'utilisation des ressources d'une architecture parallèle pour la résolution d'un problème passe par l'extraction du parallélisme d'un algorithme et par la construction d'un programme parallèle exploitant efficacement ce parallélisme. La conception d'un algorithme parallèle est un travail méthodique où il faut prendre en compte successivement un certain nombre de points liés à l'expression du parallélisme. Nous présentons dans la section 1.2.1 les principales étapes d'une méthodologie de parallélisation. Les modèles visant la mise en œuvre du parallélisme sont ensuite présentés dans 1.2.2.

1.2.1 Méthodologie de parallélisation

Une représentation souvent utilisée pour décrire un programme parallèle est celle d'un **graphe de tâches**. Les nœuds d'un tel graphe sont les tâches d'un programme (des séquences de calculs) tandis que les arcs représentent les dépendances entre les tâches (il s'agit souvent de communications). Selon cette description d'un programme parallèle, la parallélisation d'un algorithme nécessite le découpage du travail en tâches potentiellement parallèles et la définition des interactions entre elles, suivi d'une distribution de ces tâches sur les processeurs de la machine parallèle.

Choix de la granularité des tâches

Lors du découpage d'un calcul, une question importante concerne la **granularité** des tâches, c'est à dire la taille de chaque tâche relativement à la taille totale du programme. On entend par "taille d'une tâche" la quantité de calculs associés à cette tâche ne nécessitant pas de communication avec les autres tâches. Dans un découpage à **grain fin**, le calcul est découpé en un grand nombre de tâches de petite taille. Inversement, la décomposition d'un travail en tâches peu nombreuses exigeant beaucoup de calculs caractérise un découpage à **gros grain**. Une granularité fine offre plus d'opportunités pour l'exploitation du parallélisme et pour l'équilibrage de charge. Mais un grain trop fin entraîne souvent des communications trop fréquentes entre processeurs. En revanche, le découpage à gros grain permet de réduire le temps nécessaire à la gestion du parallélisme, au prix d'une diminution du parallélisme potentiel. Le choix de la granularité doit donc satisfaire un compromis entre le maximum de parallélisme potentiel, un surcoût minimum de gestion du parallélisme, et un bon équilibre de charge entre les processeurs.

En pratique, le choix de la granularité dépend fortement de l'architecture cible et de l'application. Le coût de la communication entre processeurs joue un rôle important dans l'obtention d'un parallélisme efficace : plus ce surcoût est faible dans une architecture, plus elle sera apte à exécuter des tâches de granularité fine. Les architectures à mémoire partagée sont donc idéales pour exploiter le parallélisme à grain fin, tandis que les architectures à mémoire distribuée sont plus adaptées au parallélisme à gros grain. En revanche, les grappes de multiprocesseurs ont l'avantage de permettre l'exploitation simultanée de différents grains de parallélisme.

Expression des interactions entre tâches

La définition des interactions entre les tâches d'un programme, c'est-à-dire l'utilisation par une tâche d'une donnée calculée par autre tâche est aussi importante. Cela établit des contraintes de précédence entre les tâches. Le graphe de tâches agrémenté de cette information est un graphe orienté appelé **graphe de précédence**. Ce graphe définit un ordre partiel pour l'exécution des tâches. Les tâches non ordonnées par cet ordre partiel peuvent être exécutées en parallèle.

Une alternative au graphe de précédence consiste à représenter un flot de données sur le graphe. Dans cette représentation, dénommée **graphe de flot de données**, la précédence entre tâches est induite par les échanges de données. Typiquement, les arcs du graphe correspondent à des opérations de lecture ou d'écriture d'une donnée. Une différence fondamentale entre le graphe de précédence et le graphe de flot de données se trouve au niveau des synchronisations. Dans le cas d'un graphe de précédence, une tâche est prête à exécuter quand toutes les tâches qui la précèdent sont terminées, tandis que dans un graphe de flot de données une tâche est prête si toutes les données auxquelles elle accède en lecture sont disponibles.

La détermination d'un graphe de précédence ou d'un graphe de flot de données dépend de la connaissance du comportement de l'application en terme de phases de calcul et de communication, ainsi que de leurs durées. En fonction de cette connaissance préalable de leur comportement, les applications et les algorithmes parallèles peuvent être classés **réguliers** ou **irréguliers**[136, 80]. De manière générale, un algorithme est dit régulier si le graphe de tâches associé est statique, c'est-à-dire, si la durée des tâches de ce graphe est connue avant l'exécution, ou bien peut être prédite à chaque étape d'exécution. À l'inverse, on dit que l'algorithme est irrégulier s'il est impossible de prédire le comportement de l'algorithme avant son exécution, ou si cette prédiction reviendrait à exécuter l'application, ou encore si le comportement de l'algorithme change d'une exécution à l'autre en fonction des données.

Ordonnancement et placement des tâches

Le graphe de tâches constitue un élément central pour l'**ordonnancement** d'un programme parallèle. On entend par ordonnancement l'attribution d'une date de début d'exécution à chaque tâche composant le graphe. Cette attribution doit être compatible avec le nombre de processeurs disponibles et avec les contraintes de précédence entre les tâches. Cette étape est souvent combinée avec la distribution spatiale ou **placement** du graphe, c'est-à-dire à l'association d'un site d'exécution (un processeur) pour chacune de ces tâches. L'ordonnancement et le placement d'un graphe de tâches représentant un programme parallèle sont deux aspects fortement liées[1], tous les deux ayant pour but final l'obtention d'une exécution efficace où les communications entre processeurs sont minimisées et où le taux d'utilisation des processeurs est le plus élevé possible. La distribution des tâches de calcul sur les processeurs doit donc éviter que certains processeurs ne soient sous-utilisés alors qu'il reste du travail en attente. Cet objectif peut-être vu sous deux optiques légèrement différentes : le **partage de charge** qui consiste à répartir le travail de façon à ce que tous les processeurs soient actifs à tout instant, et l'**équilibrage de charge** qui consiste non seulement à garder les processeurs actifs, mais aussi à répartir équitablement le travail entre les processeurs.

L'équilibrage de charge au cours du calcul implique de déplacer des tâches, ce qui suppose des transferts de données entre le processeur surchargé et un processeur moins actif. Cette étape nécessite des communications, dont le coût doit être faible par rapport au gain espéré. En pratique, la décision d'équilibrer la charge est prise en fonction d'un seuil minimal dépendant de la taille des tâches et représentant un compromis acceptable. Ce compromis est généralement plus facile à atteindre lorsque la granularité des tâches est fine et que les données à communiquer sont "peu volumineuses".

Les stratégies d'ordonnancement et de placement sont fortement dépendantes des caractéristiques du graphe de tâches. Dans les cas où le graphe de précédence est complètement déterminé il est possible d'utiliser des heuristiques statiques classiques[46, 86]. Néanmoins, pour un grand nombre d'applications, le comportement du programme dépend des données et le graphe de tâches associé n'est pas prédictible. Ces applications typiquement irrégulières sont caractérisées par la création dynamique d'un nombre important de tâches ayant souvent des grains très différents. La réalisation d'un ordonnancement et d'un placement efficaces pour telles applications n'est donc pas trivial.

[1] En effet, dans la langue anglaise l'ordonnancement et le placement sont désignés par un seul mot : « scheduling ».

1.2.2 Modèles de programmation parallèle

Un modèle de programmation représente l'abstraction d'une machine dans le but de séparer les aspects concernant le développement de l'application de ceux liés à son exécution effective. Des modèles existent à différents niveaux d'abstraction, du parallélisme totalement explicite (où son expression est complètement à la charge du programmeur) au parallélisme implicite (parallélisation automatique). Les modèles explicites proches des architectures sont assez répandus pour les applications visant la haute-performance, mais la programmation basée sur ces modèles est plus difficile car tous les détails de la parallélisation sont apparents. Dans la suite nous faisons un tour d'horizon sur les modèles qui s'adaptent le mieux à chaque type d'architecture, puis nous introduisons quelques modèles à haut niveau d'abstraction. Cette présentation est loin d'être exhaustive. Pour une vision plus approfondie sur les modèles de programmation parallèle qui ont été proposés au fil des années, le lecteur pourra se référer par exemple à [148] ou [147].

Programmation des architectures à mémoire partagée

Sur architecture à mémoire partagée, la programmation parallèle est souvent fondée sur des techniques de multiprogrammation légère (« multithreading »)[107, 108, 103, 30]. Selon ce modèle, le programme parallèle est composé de multiples flots d'exécution (fils d'exécution) concurrents manipulant des données placées dans une mémoire commune. La notion de fil d'exécution ou processus léger est employée depuis longtemps au sein des systèmes d'exploitation afin d'exprimer des tâches concurrentes au sein d'un même programme. Si dans une architecture conventionnelle les processus légers sont en compétition pour l'utilisation du processeur, dans une machine SMP ils peuvent réellement s'exécuter en parallèle. Par définition, les processus légers appartenant à un même processus système partagent des zones mémoire allouées à ce processus, d'où l'emploi naturel de la multiprogrammation légère sur des machines SMP.

Étant donné que les lectures et écritures dans la mémoire commune se font en parallèle, la difficulté de ce modèle de programmation est de garantir la cohérence d'accès aux zones de mémoire partagée afin de préserver la sémantique correcte du programme. Pour cela le programmeur doit recourir à des primitives de synchronisation comme les verrous ou les variables de condition (voir par exemple [104]). La programmation se fait par l'intermédiaire de bibliothèques de processus légers qui sont disponibles dans la plupart des systèmes. Un jeu de primitives standard pour des bibliothèques offrant la gestion et la synchronisation de processus légers est définit par la norme POSIX[96]. La multiprogrammation légère peut aussi être offerte

au niveau du langage de programmation, comme dans Java[120].

Une approche plus récente pour la programmation des SMP est le standard OpenMP[49]. Il consiste essentiellement en un ensemble de procédures et directives que le programmeur ajoute à un programme écrit en Fortran ou C/C++ afin d'aider le compilateur à prendre des décisions de parallélisation. La multiprogrammation légère est une notion intrinsèque dans ce standard, mais les processus légers sont confinés au support exécutif de OpenMP, le programmeur n'ayant pas à les gérer lui même.

Programmation des architectures à mémoire distribuée

Dans une architecture à mémoire distribuée, la coopération entre processeurs se fait classiquement par échange de messages. Le programme parallèle est organisé comme une collection de processus s'exécutant sur un ensemble de processeurs, chaque processus possédant une zone de mémoire privée. L'échange de données entre processus n'est possible que par des opérations d'envoi et réception explicites. L'approche de programmation la plus répandue est d'utiliser une bibliothèque d'échange de messages telle que PVM[151] ou MPI[70]. Cette dernière est en fait un standard pour la programmation par échange de messages, visant à la fois la haute performance des communications et la portabilité des programmes parallèles reposant sur ce paradigme. Des bibliothèques suivant le standard MPI sont disponibles sur un grand nombre de plates-formes actuellement, et certains constructeurs en proposent des versions hautement optimisées pour une architecture en particulier.

La possibilité de mettre au point des applications efficaces sur des architectures extensibles est un point fort du paradigme d'échange de messages, mais cela se fait au prix d'une plus grande complexité de programmation parallèle. Comme le temps de transmission des données est grand par rapport à la puissance de calcul des processeurs, il faut chercher à réduire le plus possible les communications par un choix minutieux de la granularité des tâches et une distribution intelligente des données sur les mémoires locales.

Programmation des grappes de multiprocesseurs

En ce qui concerne les architectures parallèles ayant un modèle de mémoire hybride, le paysage des modèles de programmation est moins bien défini que celui des architectures purement à mémoire partagée ou à mémoire distribuée. Selon l'abstraction de l'organisation mémoire offerte au programmeur, deux approches

principales sont envisageables pour la programmation de CluMPs [114, 29] : un modèle de programmation uniforme, où le programmeur ne voit qu'un seul modèle mémoire (partagée ou distribuée), ou un modèle de programmation hybride, qui combine la multiprogrammation légère et l'échange de messages. Pour l'unification du modèle de mémoire, une alternative consiste à utiliser des environnements de programmation fournissant la fonctionnalité d'une mémoire virtuellement partagée [143, 150, 142, 63]. Les performances d'un tel modèle sont néanmoins très dépendantes du protocole de gestion de la cohérence de la mémoire. L'autre alternative est de programmer les CluMPs comme des machines à mémoire distribuée. On trouve ainsi des bibliothèques d'échange de messages mises au point pour une communication efficace à l'intérieur des nœuds [18, 113, 29]. Le modèle de programmation hybride, à son tour, constitue une des premières approches pour la programmation des CluMPs. En effet, l'association de la multiprogrammation légère à la communication par échange de messages suscitait déjà beaucoup d'intérêt avant que les CluMPs deviennent populaires, par exemple par les possibilités de masquer les temps de communication par des calculs utiles et de gérer simplement et élégamment l'indéterminisme des communications. Parmi les bibliothèques construites sur ce principe on peut citer ATHAPASCAN-0 [44, 26, 32], PM^2[119] et Nexus[71].

Programmation indépendante d'architecture

Afin de faciliter la compréhension et le développement d'applications parallèles, mais aussi d'augmenter la portabilité de ces applications, certains modèles de programmation travaillent avec une plus grande abstraction de l'architecture parallèle. Parmi les approches de plus haut niveau on trouve la parallélisation automatique ou semi-automatique, qui vise la génération d'un code parallèle à partir d'un programme séquentiel, par l'intermédiaire d'un compilateur capable d'extraire le parallélisme présent dans un tel programme. La génération du code parallèle est souvent guidée par des directives de compilation rajoutées dans le programme source afin de définir, par exemple, un partitionnement des données du programme. Une telle technique est employée dans le langage HPF (*High Performance Fortran*), entre autres. Cette approche est pratique pour le programmeur, mais les difficultés liées à la détection automatique du parallélisme et à la génération de codes efficaces sur les différentes architectures parallèles font que les outils de parallélisation automatique sont moins généraux et par conséquence moins répandus que les outils de programmation parallèle explicite. En effet, la parallélisation automatique est actuellement plus efficace sur des architectures à mémoire partagée que sur mémoire distribuée, et les bons résultats sont surtout obtenus pour des applications constituées de nids de boucles et pour lesquelles le parallélisme ne dépend pas des données d'entrée. Le lecteur intéressé trouvera dans [66] une revue des problèmes et techniques concer-

nant la génération automatique de codes pour les architectures parallèles.

Dans un niveau intermédiaire entre la parallélisation explicite et la parallélisation automatique, on trouve des paradigmes permettant d'abstraire certains aspects de la machine parallèle. Les environnements offrant la fonctionnalité d'une mémoire virtuellement partagée en sont un exemple : ils permettent d'abstraire le réseau de communication en fournissant des mécanismes de partage de données entre processeurs, occultant ainsi les communications inhérentes à une exécution sur architecture distribuée. La granularité des données partagées est variable : certains systèmes comme TreadMarks[4] et SAM[144] offrent un espace d'adressage unique de la mémoire, l'unité de partage étant le mot mémoire élémentaire. D'autres implantent le partage de données au niveau d'objets définis par le programmeur, c'est le cas d'Orca[9], Linda[36] et Jade[134], entre autres. L'aspect critique de cette approche pour la programmation parallèle réside dans sa mise en œuvre efficace sur une architecture distribuée, étant donnée que la préservation de la cohérence de la mémoire virtuellement partagée génère du trafic sur le réseau. Les systèmes offrant le partage au niveau de l'objet ont tendance à être plus efficaces sur architecture distribuée, puisque la granularité des données partagées est plus grosse et mieux adaptée à l'application[8].

Finalement, on trouve des systèmes permettant de séparer l'expression du parallélisme du placement réel des données et des calculs sur la machine. Il s'agit ici d'une abstraction des processeurs composant l'architecture parallèle : le programmeur définit explicitement les tâches du programme parallèle et les dépendances entre elles, sans pour autant prendre en compte l'architecture sur laquelle aura lieu l'exécution. Le système exécutif assure l'ordonnancement automatique des tâches et une exécution correcte de l'application basée sur une analyse du graphe de tâches ou du graphe de flot de données. Le programmeur ne gère pas explicitement les communications entre les tâches, ce qui facilite grandement la mise en œuvre parallèle. Parmi les systèmes offrant une telle abstraction on trouve Cilk[24, 98], Jade[134], NESL[23] et ATHAPASCAN-1 [78].

1.3 Multiprogrammation légère pour le calcul parallèle

Quand l'algorithmique du problème à paralléliser peut être conçue de manière à isoler un grand nombre de tâches, il est naturel d'envisager une exécution de celle-ci en utilisant des processus différents. Une technique adaptée à la gestion de ces processus, tant pour les architectures à mémoire partagée que pour les architectures

à mémoire distribuée, est la multiprogrammation légère. Pour les secondes, ces processus sont alors utilisés conjointement avec une bibliothèque de communication.

1.3.1 Les processus légers en bref

Un processus léger (ou PL) [2] est une abstraction permettant d'exprimer des multiples flots d'exécution indépendants au sein d'un processus système (e.g. UNIX, appelé processus lourd). Les différents flots d'exécution partagent le contexte de mémoire du processus lourd, leur gestion est donc moins coûteuse que celle des processus système. En effet, la gestion des processus légers nécessite seulement la manipulation de leur contexte d'exécution, tandis que les opérations sur les processus lourds demandent aussi la gestion de zones mémoire et de ressources (fichiers ouverts, verrous, etc.) allouées par le système. Par exemple, un changement de contexte pour un processus léger est de 10 à 1000 fois plus rapide que pour un processus lourd[43].

Les opérations usuellement disponibles pour la programmation avec des processus légers sont la création (sur le même processeur ou sur un autre processeur), la destruction (souvent l'auto-destruction), la communication entre PL (par messages, appels de procédure à distance, etc.) et la synchronisation (par sémaphores, verrous d'exclusion mutuelle, barrière de synchronisation, partage de données, etc.). La norme POSIX[96] définit un standard pour les opérations manipulant des processus légers et des implantations sont proposées par la plupart des grands constructeurs d'ordinateurs.

Bien qu'au niveau de l'interface de programmation les bibliothèques de PL offrent souvent des fonctionnalités similaires, leur mise en œuvre diffère sensiblement. Les différents choix d'implantation de ces bibliothèques et de leur intégration avec le système d'exploitation sont étudiés en détail dans [43] et [32], entre autre. Nous ne donnons ici qu'un aperçu des différentes techniques proposées au fil du temps. Ainsi, par exemple :

- certaines bibliothèques s'appuient sur des "PL système", qui sont connus du système d'exploitation sous-jacent. D'autres reposent sur des "PL utilisateurs", qui sont plus "légers" que les précédents. Ils ne sont pas connus du système d'exploitation mais seulement de l'environnement de programmation parallèle qui offre ces PL utilisateurs ;
- la multiprogrammation des PL peut être implantée avec différentes formes de

[2] Il existent plusieurs synonymes pour désigner les processus légers : "fil d'exécution", "flot d'exécution", « thread » ou « lightweight process ». L'acronyme PL n'est pas courant, mais nous l'utilisons pour alléger l'exposé.

préemption : préemption uniquement quand le PL est bloqué (e.g. en attente d'un message ou d'un sémaphore), préemption à la fin d'un quantum, etc. ;
- la politique d'ordonnancement des PL sur chaque processeur peut varier : premier arrivé-premier servi, utilisation de priorités, etc.

Lorsqu'un environnement de programmation parallèle utilise conjointement une bibliothèque de PL et une bibliothèque de communication (comme PVM[151] ou MPI[70]), il doit y avoir compatibilité (plus ou moins forte) entre les deux bibliothèques :
- si la bibliothèque de communication est « thread-safe », elle peut être utilisée de façon concurrente par plusieurs PL ;
- si une opération de communication est « thread-synchronous », elle peut bloquer le PL qui l'exécute sans bloquer les autres PL dans le même contexte ;
- si la bibliothèque de communication est « thread-aware », elle connaît la multiprogrammation des PL.

1.3.2 Le bénéfice à tirer des processus légers

Prenons l'exemple général d'un programme composé de tâches indépendantes (voir figure 1.1). L'utilisation de processus légers simplifie la programmation, car la coordination entre tâches est réduite. Les tâches sont exécutées de façon concurrente ou en parallèle et, si un processus léger est bloqué (par exemple, en attendant une lecture d'un fichier), les autres processus légers peuvent continuer. Le processeur est ainsi mieux utilisé.

Les PL constituent une technique naturelle pour l'implémentation des tâches d'une application parallèle. Cette vision d'un programme parallèle comme un ensemble de tâches/PL interagissant a plusieurs intérêts :
- elle permet de décrire l'application indépendamment de la machine cible, comme un ensemble de tâches multiprogrammées s'exécutant sur un nombre arbitraire de processeurs (mais en pratique inférieur au nombre de tâches potentiellement parallèles) ;
- elle simplifie la programmation des applications. Le programmeur n'a pas à gérer lui-même le masquage des attentes et des communications par du calcul, qui est pris en charge par la multiprogrammation de chaque processeur. Si l'environnement dispose d'un ordonnanceur global, celui-ci prend aussi en charge l'allocation des tâches/PL aux processeurs (statiquement ou dynamiquement) afin que tous les processeurs aient des tâches actives. Il faut noter que la structuration d'un programme sous forme de graphe de tâches est la bonne structure pour les algorithmes d'ordonnancement.

Les PL communicants sont donc le mécanisme d'abstraction d'une machine pa-

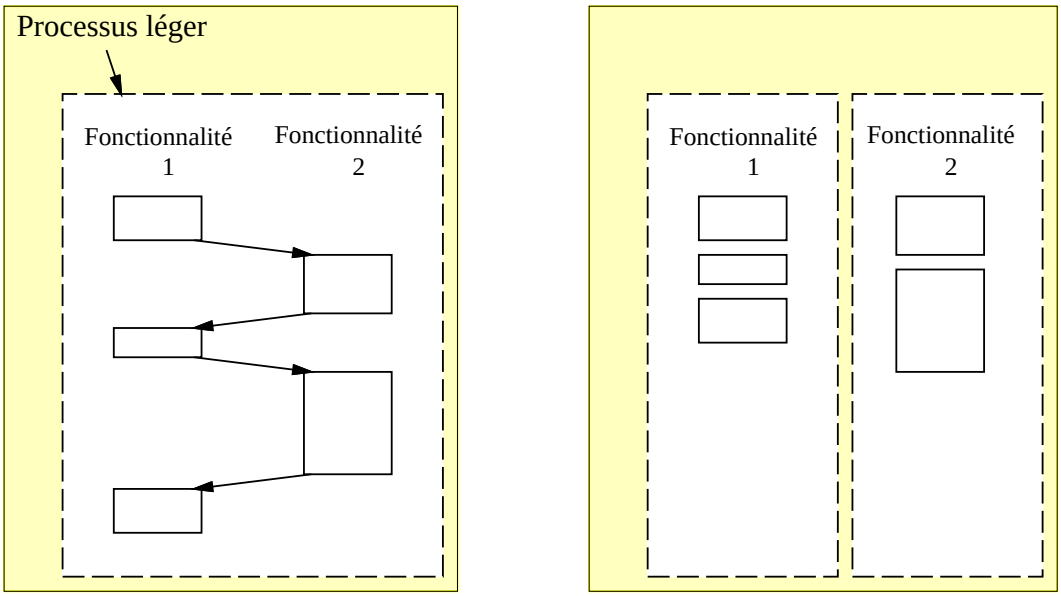

Figure 1.1 *Programmation classique vs. multiprogrammation d'un programme composé de tâches indépendantes.*

rallèle (à mémoire commune ou distribuée) permettant une programmation **flexible** et **réactive** d'un programme parallèle composé de tâches.

1.4 L'environnement de programmation ATHAPASCAN

ATHAPASCAN est un environnement de programmation parallèle conçu pour être à la fois portable et efficace sur CluMPs, et permettre de façon aisée le développement d'applications parallèles[26, 44, 128]. Cet environnement est constitué essentiellement par un noyau exécutif à base de processus légers communicants (ATHAPASCAN-0) et une interface de programmation parallèle haut niveau (ATHAPASCAN-1). La mise au point d'applications développées sur ATHAPASCAN est assurée par un outil de visualisation de programmes parallèles (PAJÉ [50, 51]), qui permet de représenter graphiquement toutes sortes d'entités et d'événements au niveau des deux modules de l'environnement ATHAPASCAN et au niveau de l'application programmée par l'utilisateur. De nombreuses applications ont été écrites au sein

du projet APACHE en utilisant l'environnement ATHAPASCAN. Citons entre autres une application de dynamique moléculaire[16], un code de chimie quantique[115], une bibliothèque de calcul symbolique[81] et un code de recherche combinatoire de type « branch and bound » [53].

1.4.1 Le noyau exécutif : ATHAPASCAN-0

ATHAPASCAN-0 est une bibliothèque permettant l'utilisation de la multiprogrammation légère dans un contexte distribué. L'abstraction de la machine parallèle offerte par ATHAPASCAN-0 est celle d'un réseau de nœuds [3] de calcul sur lesquels s'exécutent des processus légers. Les processus légers communiquent entre eux par des échanges de message entre nœuds distincts et par mémoire commune à l'intérieur d'un même nœud. Pour supporter un tel modèle, l'interface de programmation de ATHAPASCAN-0, écrite en langage C, offre des primitives permettant la gestion (création, destruction, suspension, etc.) des processus légers, la communication et la synchronisation entre processus légers.

Ce noyau exécutif permet d'exploiter quatre types de parallélisme pouvant être présents sur une architecture parallèle :

- le parallélisme entre les nœuds composant l'application, étant donné que l'exécution de chaque nœud se déroule indépendamment des autres nœuds (mise à part les synchronisations dues aux communications) ;
- le parallélisme à l'intérieur des nœuds, étant donné que chaque nœud peut comporter des multiples processus légers. Si le nœud réside physiquement sur une machine à plusieurs processeurs, le système d'exploitation gérant ce nœud se charge de répartir les processus légers entre ces processeurs ;
- le parallélisme entre calculs et communications, par l'intermédiaire de primitives asynchrones de communication offertes par la bibliothèque. Ce type de parallélisme est également exploitable en utilisant plusieurs processus légers par nœud : certains processus légers effectuent des calculs tandis que d'autres se chargent des communications, dans ce cas avec des primitives synchrones ;
- le parallélisme entre les communications, si un processus léger initie plusieurs communications non bloquantes ou si plusieurs fils d'exécution communiquent à partir d'un même nœud.

ATHAPASCAN-0 représente la couche de portabilité de l'environnement ATHAPASCAN, servant de support exécutif au module de plus haut niveau qui est ATHAPASCAN-1. Il sert aussi pour la programmation directe d'applications parallèles et peut être utilisé en tant que support exécutif pour d'autres langages de program-

[3] Un nœud ATHAPASCAN-0 est actuellement implanté comme un processus lourd Unix.

mation et/ou des bibliothèques parallèles (e.g. GIVARO[79]). Pour assurer sa portabilité, ATHAPASCAN-0 repose sur des bibliothèques standard pour la gestion de processus légers (POSIX ou équivalent) et pour la mise en œuvre de l'échange de messages (MPI). ATHAPASCAN-0 est actuellement disponible sur des nombreuses plates-formes, comme par exemple Intel/Linux, SPARC/Solaris, IBM SP/Aix, SGI MIPS/Irix.

1.4.2 L'interface de programmation haut niveau : ATHAPASCAN-1

ATHAPASCAN-1 est la couche de plus haut niveau de l'environnement ATHAPASCAN, construite pour faciliter la programmation d'applications parallèles de façon indépendante de l'architecture. Ceci est réalisé par une description d'un programme parallèle sous forme d'un graphe de tâches effectuant des lectures et écritures de données situées dans une mémoire virtuelle partagée. Le parallélisme est toujours explicite dans ATHAPASCAN-1 : le programmeur doit fixer les différents niveaux de découpe du travail en tâches, mais ce découpage est indépendant de l'architecture. En effet, ATHAPASCAN-1 abstrait les détails liés à la machine cible prenant en charge tout ce qui relève des mouvements et synchronisations de données. Le programmeur n'a pas à spécifier les communications et tous les échanges de données sont faits de manière implicite à partir d'une relation de dépendance de données décrite au moment de création de chaque tâche. À partir de cette description, ATHAPASCAN-1 construit un graphe de flot de données pour déterminer l'ordre d'exécution des tâches et le mouvement des données. Le placement réel des données et des calculs sur la machine est aussi pris en charge par ATHAPASCAN-1, qui permet l'emploi de différentes stratégies d'ordonnancement de tâches pour mieux répartir la charge d'une application sur l'architecture cible.

Pratiquement, ATHAPASCAN-1 se présente sous forme d'une librairie C++ générique. L'interface de programmation est réduite à deux primitives : `fork` et `shared`. Les procédures composant un programme ATHAPASCAN-1 utilisent la primitive `fork` pour créer des tâches de calcul potentiellement parallèles, et la primitive `shared` pour déclarer des données partagées et ses modes d'accès associés. Un `fork` est toujours une opération asynchrone. La création effective de la tâche dépend de la stratégie de régulation de charge : par exemple, une nouvelle tâche parallèle peut être crée si certains processeurs sont inactifs, sinon la tâche est exécutée comme un appel de procédure en séquence.

Ce mécanisme permet, à partir d'un programme donné, de déduire le parallélisme effectif, soit parce que le nombre de processeurs est restreint, soit à cause du

coût de gestion du parallélisme.

1.5 Conclusion

Dans ce chapitre, nous avons fait un tour d'horizon des concepts du parallélisme. Il faudra en retenir que pour la suite nous considérons la programmation des CluMPs avec des processus légers et des communications, et qu'un programme parallèle est un graphe orienté de type flot de données. Le chapitre suivant fait un état de l'art sur les méthodes de décomposition de domaine.

2

La simulation numérique en parallèle

Sommaire

2.1 Discrétisation de problèmes d'EDP 41
2.1.1 Le problème modèle et son analyse mathématique . . . 41
2.1.2 Maillage du domaine physique 43
2.1.3 Formulation matricielle 44
2.1.4 Résolution du système 45
2.2 Adaptation de maillage 46
2.3 Méthodes de décomposition de domaine 48
2.3.1 Partitionnement de maillage 49
2.3.2 Méthodes avec recouvrement 50
2.3.3 Méthodes sans recouvrement 54

L'étude de phénomènes physiques complexes tels que rencontrés en mécanique des fluides, calculs de structure, électromagnétisme, etc., nécessite souvent la résolution d'un système d'Équations aux Dérivées Partielles (EDP). En général, il est impossible de déterminer exactement les solutions d'un problème d'EDP donné. On fait alors appel à des méthodes numériques telles que la méthode de différences finies, la méthode de volumes finis ou la méthode des éléments finis pour discrétiser le problème d'EDP. Nous avons choisi la méthode des éléments finis.

Le préalable à la discrétisation des systèmes d'EDP est la construction d'un maillage dont les caractéristiques dépendent du problème physique et de la géométrie du domaine de calcul. Ce maillage est constitué d'éléments géométriques simples définis par leurs sommets. En utilisant ce maillage on transforme le problème continu en un problème discret dont la taille est proportionnelle au nombre de sommets. Les inconnues de ce problème sont les degrés de liberté définis sur les éléments géométriques.

Un bon moyen pour traiter les problèmes de grande taille consiste à utiliser une méthode de décomposition de domaine. Ces méthodes reposent sur le découpage géométrique du domaine physique en sous-domaines pour permettre de travailler avec des problèmes plus petits. Ces problèmes sont couplés par des conditions aux limites imposées à leurs frontières. La manière dont sont couplés les problèmes définit chacune des méthodes de décomposition de domaine. Le caractère parallèle de celle-ci en découle naturellement car les sous-problèmes peuvent être résolus localement sur des processeurs différents. Cela implique des communications entre processeurs traitant des sous-domaines voisins.

Parfois, il est nécessaire de construire des discrétisations très fines dans des zones où apparaissent des singularités de nature mathématique ou de nature géométrique. Dans ce cas, il est possible d'adapter dynamiquement la discrétisation en raffinant ou déraffinant le maillage pour éviter l'utilisation d'une discrétisation globalement fine, trop pénalisante en place mémoire et en temps calcul.

Dans le paragraphe 2.1 de ce chapitre nous présentons les principaux aspects de la méthode des éléments finis, d'autres détails sont présentés dans l'annexe A. L'adaptation de maillage et les problèmes de répartition de charge qui en découlent sont discutés dans le paragraphe 2.2. Le paragraphe 2.3 traite ensuite du partitionnement de maillages et des méthodes de décomposition de domaine classiques. Pour illustrer nos propos nous avons choisi un problème modèle bidimensionnel.

2.1 Discrétisation de problèmes d'EDP

La méthode des éléments finis est utilisée pour la résolution numérique d'EDP provenant de la modélisation de phénomènes physiques très variés (thermique, électromagnetisme, aérodynamique, mécanique des fluides, mécanique des structures, etc.). Comme pour toute méthode de discrétisation, son objectif est le calcul des valeurs approchées du champs étudié (température, champ magnétique, pression, vitesse, contraintes, déplacements, etc.) en certains points du domaine physique dans lequel est résolu le problème. On transforme ainsi le problème continu en un problème discret.

Quel que soit le problème à résoudre, la mise en œuvre de la méthode des éléments finis comprend les étapes suivantes :

1. analyse mathématique du problème avec, en particulier, l'écriture de la formulation variationnelle du système d'EDP à résoudre ;
2. construction d'un maillage du domaine physique vérifiant certaines propriétés ;
3. définition des éléments finis ;
4. assemblage de la matrice et du second membre à partir des contributions de chaque élément, avec prise en compte des conditions aux limites ;
5. résolution du système.

Dans ce paragraphe la discussion est centrée sur les points 1, 2 et 5. Les points 3 et 4 sont développés de manière exhaustive dans l'annexe A, on y trouve également une présentation plus détaillée du point 2.

2.1.1 Le problème modèle et son analyse mathématique

Le problème choisi modélise les déplacements verticaux d'une membrane élastique fixée par son bord et soumise à une force verticale donnée (voir par exemple [112]). Étant donné Ω une membrane de frontière $\Gamma = \partial\Omega$ suffisamment régulière, et f la force verticale appliquée sur cette membrane, le champ de déplacements u que l'on souhaite connaître est solution du **problème de Poisson** :

$$\left\{ \begin{array}{rcll} -\Delta u & = & f & \text{dans } \Omega, \\ u & = & 0 & \text{sur } \Gamma. \end{array} \right. \tag{2.1}$$

où Δ est **l'opérateur de Laplace** qui, en dimension 2, s'écrit :

$$\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}.$$

La condition limite $u = 0$ sur la frontière du domaine Ω est appelée **condition de Dirichlet homogène**, elle signifie que le bord du domaine Ω ne peut pas se déplacer.

Lorsque f est une fonction de $L^2(\Omega)$, c'est à dire lorsque f appartient à l'ensemble des fonctions de carré intégrable sur Ω, le système d'EDP (2.1) a une unique solution u dans l'espace $V = H_0^1(\Omega)$, défini par l'ensemble de fonctions de $L^2(\Omega)$ dont la dérivée est dans $L^2(\Omega)$ et dont la trace est nulle sur Γ. Pour une analyse mathématique plus détaillée du problème, nous renvoyons à [131].

Pour résoudre le problème par la méthode des éléments finis, on commence par écrire la formulation variationnelle correspondant au système (2.1). On multiplie (2.1) par une fonction test $v \in H_0^1(\Omega)$, comme u, et on intègre par parties. On obtient

$$\int_\Omega \nabla u \cdot \nabla v dx = \int_\Omega f v dx - \int_\Gamma \frac{\partial u}{\partial n} \cdot v dn$$

où n est la normale extérieure au domaine Ω, et ∇u désigne le vecteur

$$\nabla u = \begin{pmatrix} \dfrac{\partial u}{\partial x} \\ \\ \dfrac{\partial u}{\partial y} \end{pmatrix}.$$

Notre formulation variationnelle se réduit à

$$\int_\Omega \nabla u \cdot \nabla v dx = \int_\Omega f v dx \tag{2.2}$$

car, v appartenant à $H_0^1(\Omega)$, v est nulle sur la frontière.

La méthode des éléments finis utilise la formulation (2.2) pour déterminer des approximations de la solution u. Pour cela, on construit un sous-espace V_h (voir annexe A, paragraphe A.2) de dimension finie inclus dans l'espace $H_0^1(\Omega)$, et on cherche $u_h \in V_h$, solution approchée de u, comme solution de

$$\int_\Omega \nabla u_h \cdot \nabla v_h dx = \int_\Omega f v_h dx, \quad \forall v_h \in V_h. \tag{2.3}$$

2.1.2 Maillage du domaine physique

Un maillage d'un domaine Ω, noté $\mathcal{M}_h$, est un ensemble de N_k éléments géométriques "simples" (également appelés mailles) recouvrant Ω, définis par les N sommets du maillage. L'indice h est le pas de discrétisation du maillage, il correspond à la plus grande arête d'un élément appartenant à $\mathcal{M}_h$. Plus h est petit, meilleure sera l'approximation de la solution du problème posé sur Ω. Chaque élément est noté M_k et le domaine recouvert Ω_h est tel que $\Omega_h = \cup_{k=1}^{N_k} M_k$. Le choix de la géométrie des éléments dépend de la géométrie du domaine Ω et de la nature du problème à résoudre. En dimension 2, ce sont par exemple des triangles ou des rectangles (resp. des tétraèdres ou des parallélépipèdes en dimension 3). Les éléments constituant le maillage doivent satisfaire certaines conditions. En particulier, il est nécessaire que le maillage soit **conforme**, c'est à dire, que $\mathcal{M}_h$ vérifie les propriétés suivantes :

1. tout élément M_k de $\mathcal{M}_h$ est d'aire non nulle ;
2. l'intersection de deux éléments de $\mathcal{M}_h$ est soit vide, soit réduite à un sommet ou une arête complète (ou une face complète en dimension 3).

La figure 2.1 présente des maillages en triangles d'un domaine Ω, illustrant les différences entre un maillage conforme et un maillage non-conforme. La suite de l'exposé de la méthode des éléments finis discute uniquement de maillages conformes.

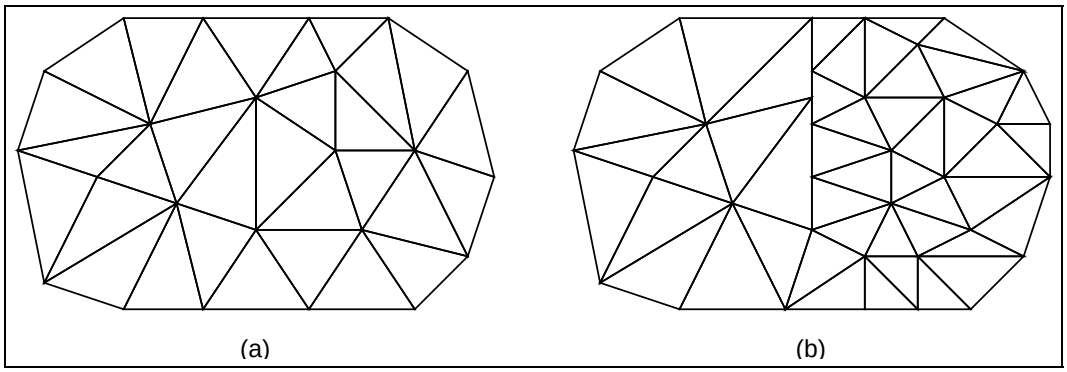

Figure 2.1 *(a) Maillage conforme, (b) Maillage non-conforme.*

D'autres propriétés géométriques sont à vérifier pour que la méthode des éléments finis fournisse de bonnes approximations de la solution (voir entre autres [131]). Par exemple, la taille des éléments doit varier progressivement, c'est-à-dire, sans présenter de discontinuités trop brutales, et les éléments triangulaires ou quadrangulaires ne doivent pas présenter d'angles trop obtus. Les caractéristiques géo-

métriques du maillage sont fondamentales pour la méthode des éléments finis, car elles permettent de démontrer les propriétés d'approximation de la méthode. Elles sont étroitement liées au problème physique à résoudre et à ses éventuelles singularités. En conséquence, la nature du problème conditionne le choix de mailles plus fines (i.e. une plus grande densité d'éléments) dans certaines zones du domaine pour obtenir des solutions avec suffisamment de précision.

En ce qui concerne la mise en œuvre informatique, un maillage est fondamentalement représenté par une liste de sommets et une liste définissant l'interconnexion entre les sommets (nous verrons plus tard que cette représentation ne suffit pas pour la méthode des éléments finis). Selon la topologie d'interconnexion des sommets, les maillages peuvent être classés comme structurés ou non-structurés[82]. Dans les maillages **structurés**, le schéma d'interconnexion est tel que la connaissance des indices d'un sommet en donne la position relative. C'est le cas, par exemple, des maillages en quadrangles lorsque l'on travaille en dimension 2. Dans un tel maillage, un sommet noté (i, j) a pour voisin de "gauche" le sommet $((i-1), j)$ et pour voisin de "droite" le sommet $((i+1), j)$. Dans les maillages **non-structurés**, en revanche, le schéma d'interconnexion des sommets est de type quelconque. Les maillages non-structurés sont donc plus généraux, ils permettent en particulier de recouvrir des domaines ayant une géométrie complexe. La méthode des éléments finis se prête bien à l'utilisation de maillages non-structurés, tandis que les maillages structurés sont plus souvent associés aux méthodes de discrétisation de type différences finies.

La génération de maillages adaptés aux problèmes à résoudre n'est pas une opération simple, surtout pour des domaines de géométrie irrégulière. Beaucoup de travaux de recherche portent sur des techniques permettant de générer des maillages automatiquement. Pour plus de détails sur ce sujet, nous renvoyons à [82].

2.1.3 Formulation matricielle

Sur ce maillage on utilise la méthode des éléments finis présentée dans l'annexe A pour écrire le problème (2.3) sous la forme matricielle. En notant ϕ_i les fonctions éléments finis, on obtient le système suivant :

$$AU = F, \tag{2.4}$$

où A est une matrice définie positive de taille $N \times N$, d´éléments

$$A_{ij} = \int_{\Omega_h} \nabla\phi_i \cdot \nabla\phi_j dx, \quad \forall i, j = 1, ..., N, \tag{2.5}$$

U est le vecteur solution $u_1, u_2, ..., u_N$, et F est un vecteur de dimension N, de composantes

$$F_j = \int_{\Omega_h} f \phi_j dx \quad \forall j = 1, ..., N. \tag{2.6}$$

En général la matrice A est creuse. Lorsque le maillage est structuré, elle comporte 5 diagonales dans le cas des éléments finis P_1 (voir annexe A). Lorsque le maillage est non-structuré, la matrice n'a pas de structure particulière. Toutefois il existe des algorithmes de renumérotation des sommets du maillage permettant de limiter la largeur de bande de la matrice.

2.1.4 Résolution du système

Les méthodes permettant de résoudre le système (2.4) sont classiquement regroupées en deux classes[84] :

Méthodes directes Ces méthodes sont basées sur une factorisation de la matrice A permettant d'aboutir à la solution en un nombre fini d'opérations. Elles consistent à déterminer une matrice de permutation P telle que la matrice PA puisse se factoriser en un produit d'une matrice L triangulaire inférieure et d'une matrice R triangulaire supérieure. Cela correspond à la méthode d'élimination de Gauss avec pivot total. Cette méthode a l'avantage d'être précise et stable numériquement mais sa complexité, mesurée par $O(N^3)$ opérations et $O(N^2)$ variables en mémoire, restreint son usage à des systèmes de petite taille ($N \leq 5000$ environ). Afin de réduire le coût en mémoire des méthodes directes, on exploite la structure creuse de la matrice en stockant peu, voire pas, de coefficients nuls[59]. Il existe de nombreux types de stockage creux : bande, profil, compressé par lignes, par colonnes, par diagonales, etc.

Méthodes itératives Ces méthodes sont basées sur le calcul d'approximations successives, de plus en plus précises, de la solution du système linéaire. Les méthodes de projection de Krylov telles que la méthode du gradient conjugué et la méthode GMRES (*Generalized Minimum RESidual*) sont parmi les plus étudiées actuellement[141]. L'avantage de ces méthodes est que la matrice A n'est utilisée que dans des produits matrice-vecteur. Théoriquement, ces méthodes convergent en $O(N)$ itérations, mais pratiquement on utilise des préconditionneurs pour accélérer la convergence. Ces méthodes sont en général supérieures aux méthodes directes pour les systèmes de grande taille ($N \geq 10000$ environ). Cependant la convergence des méthodes itératives est hautement dépendante de la disponibilité d'un bon préconditionneur.

Bien que ces algorithmes soient performants, le volume de données est important et les temps de restitution sont longs lorsque la taille du problème discret est très grande. On fait alors appel au parallélisme pour réduire les temps d'exécution soit en agissant au niveau matriciel (méthodes dites de « multisplitting » [121, 162]), soit en agissant au niveau de la discrétisation mathématique du problème. Dans ce cas, les numériciens proposent l'utilisation de méthodes de décomposition de domaine.

2.2 Adaptation de maillage

L'utilisation d'une méthode numérique approchée telle que la méthode des éléments finis, implique le choix d'un pas de discrétisation h adapté au problème à résoudre. Ce choix doit mener à des résultats d'une précision et d'un coût de calcul acceptables.

Pour des problèmes dont la solution est régulière, le choix du pas de discrétisation s'appuie sur une estimation d'erreurs *a priori*, résultant de propriétés théoriques de la méthode de discrétisation choisie. Par exemple, pour les éléments finis de type P_1, on sait que l'erreur commise lors du calcul d'une approximation u_h de la solution exacte u de notre problème modèle (2.1) est bornée par

$$||u - u_h|| \leq Ch, \tag{2.7}$$

où C est une constante indépendante de h.

Cependant, la solution peut présenter des singularités provenant de la géométrie du domaine (coin entrant, fissure, etc.) ou du système d'équations (matériaux différents, ondes de choc, etc.). Pour atteindre une précision acceptable sur tout le domaine, les estimations *a priori* imposent l'utilisation d'un maillage globalement fin. D'un point de vue pratique, il serait intéressant d'avoir une grande densité d'éléments dans les régions du domaine où la solution est singulière, et, éventuellement, de diminuer la densité des éléments dans les régions où la solution est très régulière.

Quand les singularités proviennent de la géométrie du domaine, il est encore facile d'identifier les régions critiques et ainsi construire un maillage adapté avant de démarrer les calculs (voir exemple de la figure 2.2).

Dans les autres cas, et notamment pour les problèmes d'évolution, les singularités peuvent se déplacer au cours du temps, ce qui suggère l'emploi de phases d'adaptation dynamique du maillage. Dans ce cas, chaque résolution du problème d'EDP est suivie d'une phase d'estimation d'erreur *a posteriori* qui fournit une

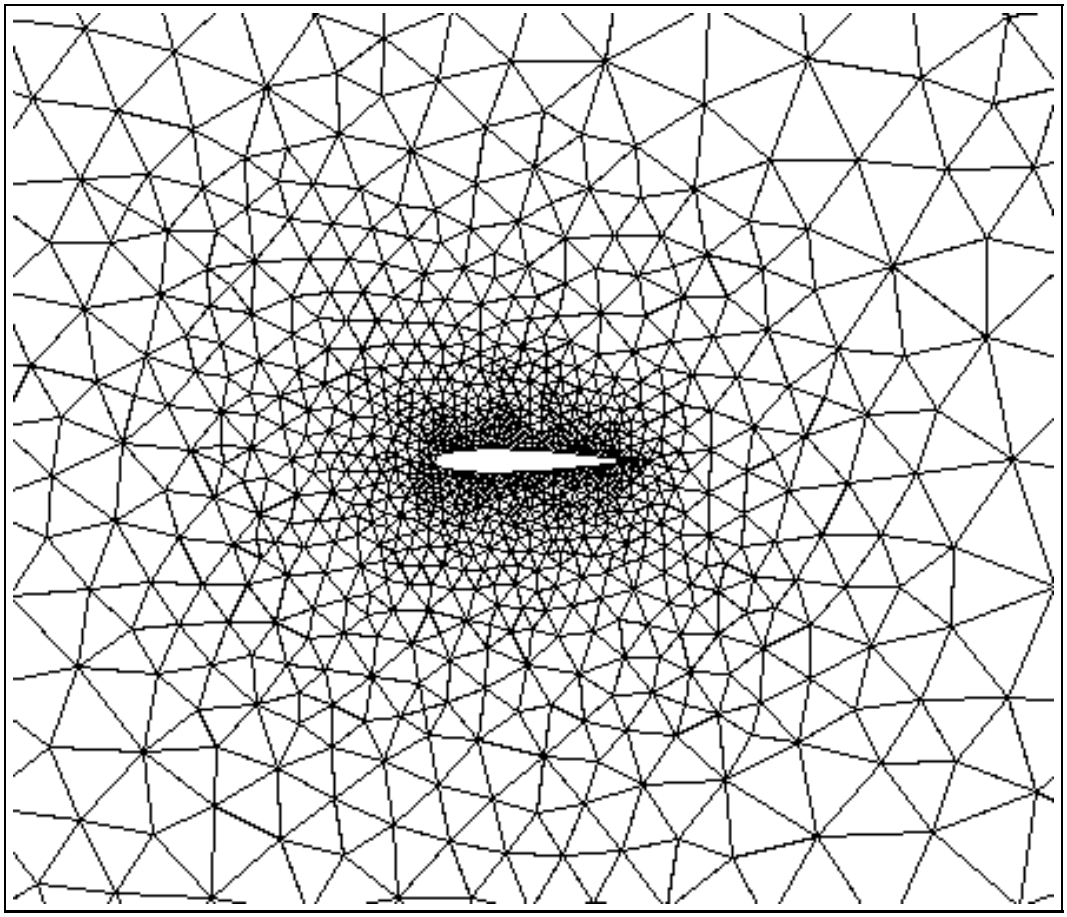

Figure 2.2 *Exemple d'adaptation statique : maillage autour d'un profil d'aile d'avion.*

carte des erreurs commises sur la discrétisation courante. Dans ce processus itératif, l'itération numéro p se décompose comme suit :

1. résolution du problème sur la triangulation $\mathcal{M}^p$ par le procédé habituel. Cela fournit une solution, notée U^p.
2. pour tout élément du maillage $\mathcal{M}^p$,
   - calcul d'un indicateur d'erreur local ;
   - si l'indicateur local est supérieur à un seuil donné, l'élément doit être raffiné,
3. si des éléments du maillage doivent être raffinés on modifie le maillage et on passe à l'étape 1, avec $p = p + 1$.

La construction de l'indicateur d'erreur local est basée sur l'analyse numérique des équations du problème à résoudre[160]. Pour chaque élément, il dépend de la

valeur de la solution dans l'élément considéré et la valeur de la solution dans ses éléments voisins. Des méthodes de raffinement de maillage sont discutées dans l'annexe B.

Dans le cadre des méthodes de décomposition de domaine (voir paragraphe 2.3), l'adaptation de maillage peut induire un déséquilibre de charge et nécessiter un repartitionnement du maillage.

2.3 Méthodes de décomposition de domaine

Généralement on utilise une méthode de décomposition de domaine lorsque le problème physique nécessite une résolution (éléments finis, différences finies, etc.) dans un espace discret de très grande dimension. Lorsque les problèmes comportent des singularités méritant un traitement particulier (coin, fissure, matériaux différents, cf. figure 2.3), on utilise un maillage avec un grand nombre de sommets ou des éléments finis de haut degré pour obtenir des solutions plus précises.

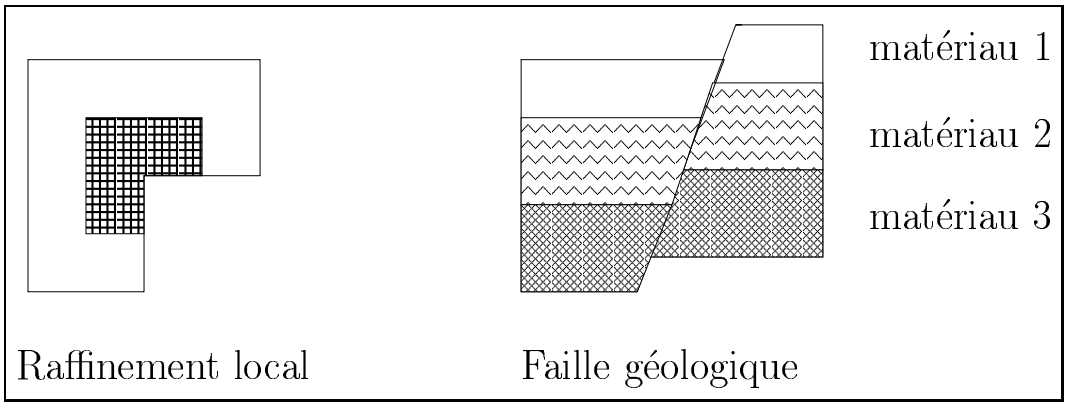

Figure 2.3 *Exemples de singularités.*

Les méthodes de décomposition de domaine sont également intéressantes pour les problèmes numériques impliquant un couplage de modèles sur des domaines naturellement disjoints car, dans ce cas, les équations à résoudre diffèrent d'un domaine à l'autre (océan/atmosphère, fluide/structure, cf. figure 2.4).

L'algorithmique des méthodes de décomposition de domaine suit le principe classique "Diviser pour Régner". Cela conduit à une parallélisation très naturelle, particulièrement adaptée à des architectures parallèles à mémoire distribuée. Il est intéressant de noter que ces méthodes peuvent être plus rapides, même sur calculateur séquentiel, que les méthodes itératives classiques [137].

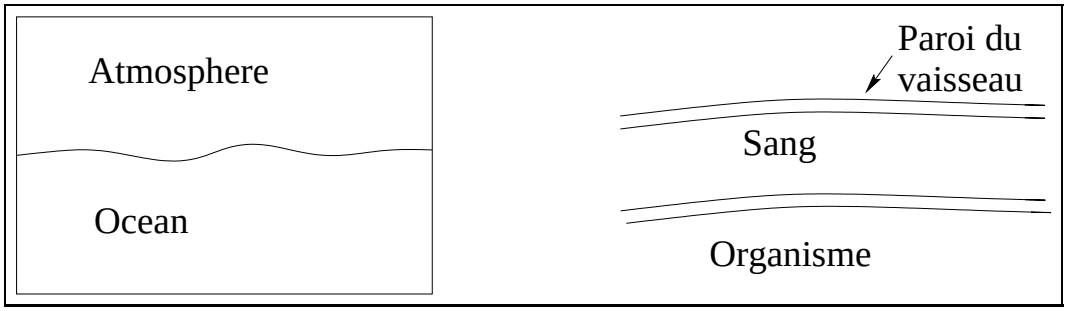

Figure 2.4 *Couplages de modèles.*

Après une courte discussion sur les manières de partitionner le maillage, préambule à toute méthode de décomposition de domaine, le prochain paragraphe présente des méthodes avec recouvrement (Schwarz) et des méthodes sans recouvrement (Schur primal, Schur dual, méthode des joints).

2.3.1 Partitionnement de maillage

Pour utiliser une méthode de décomposition de domaine il faut commencer par construire une partition du domaine (avec ou sans recouvrement) en tenant compte de critères mathématiques, qui ont une influence sur la convergence des méthodes, et de critères informatiques pertinents pour la parallélisation.

De manière générale, les outils de partitionnement de maillage existants utilisent des contraintes géométriques :

- répartition équitable du nombre d'arêtes. Cet aspect est important lors de l'utilisation d'une méthode de décomposition de domaine, car il est équivalent à équilibrer la charge de calcul associé à la résolution des sous-domaines si l'on considère que la charge par élément est homogène à travers le maillage ;
- nombre d'interfaces. Cela revient à minimiser le nombre de sous-domaines voisins pour chaque sous-domaine de manière à minimiser le nombre de communications de la résolution parallèle ;
- minimisation du nombre d'arêtes sur les interfaces d'un sous-domaine par rapport au nombre d'arêtes situées à l'intérieur de celui-ci ;

Ces critères conduisent en général à des sous-domaines relativement compacts, ce qui est numériquement favorable à l'efficacité des méthodes de décomposition de domaine.

Les outils existants sont essentiellement basés sur des algorithmes travaillant

à partir des coordonnées géométriques du maillage, ou des algorithmes utilisant le graphe associé à la matrice d'adjacence du maillage. Parmi les outils les plus connus on peut citer Metis[101], Scotch[126] et Chaco[91].

2.3.2 Méthodes avec recouvrement

Ces méthodes se caractérisent par le découpage de l'espace physique en un ensemble de domaines se recouvrant. Dans ce document, nous nous limiterons à la méthode de Schwarz dont nous présentons deux variantes : la méthode multiplicative et la méthode additive. Pour une discussion plus approfondie à propos de ces méthodes, nous renvoyons à [149] et [38].

Schwarz (1869) [145] a été le premier a introduire un schéma de décomposition de domaine applicable à la résolution de problèmes d'EDP, la convergence du schéma étant prouvée ultérieurement[110]. Dans cette méthode, on décompose le domaine ouvert, borné et connexe Ω en K sous-domaines ouverts Ω_k ($k = 1, .., K$) tels que

$$\Omega = \cup_{k=1,..,K} \Omega_k.$$

Pour illustrer nos propos nous considérons, par la suite, une décomposition du domaine principal en deux sous-domaines se recouvrant. Un exemple est proposé dans la figure 2.5.

Soit $\Gamma = \partial\Omega$ la frontière du domaine Ω, et on distingue les parties γ_k^l de frontière de Ω_k ($k = 1, .., K$) inclues dans les sous-domaines Ω_l ($l = 1, .., K, l \neq k$) des parties $\Gamma_k \subset \Gamma$, i.e. :

$$\begin{cases} \gamma_k^l & = & \partial\Omega_k \cap \Omega_l, \quad \forall l = 1, .., K, l \neq k, \\ \Gamma_k & = & \partial\Omega_k \cap \Gamma. \end{cases}$$

La méthode de Schwarz est une méthode itérative qui consiste à résoudre les problèmes d'EDP de façon alternée sur chaque sous-domaine, il n'y a pas de calcul d'interface (voir méthodes sans recouvrement, paragraphe 2.3.3). À chaque itération, des valeurs calculées sur Ω_l sont utilisées comme conditions aux limites imposées sur les parties de frontière γ_k^l sur Ω_k lors de l'itération suivante. L'itéré choisi pour mettre à jour les conditions aux limites détermine les variantes de cette méthode.

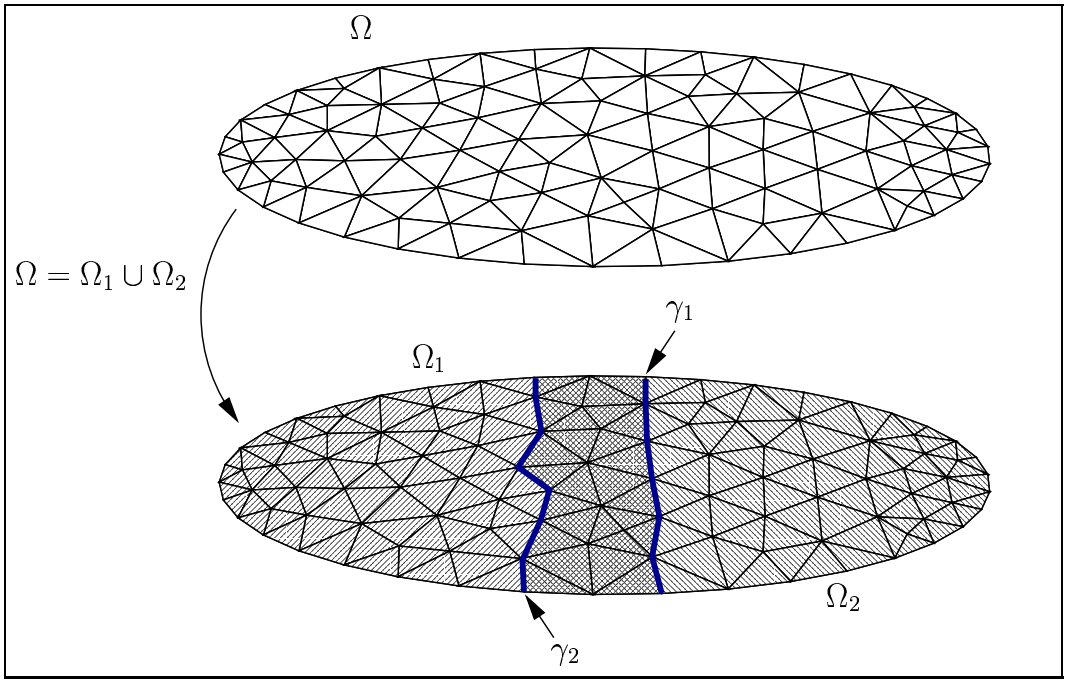

Figure 2.5 *Décomposition avec recouvrement.*

2.3.2.1 Méthode de Schwarz multiplicative

Pour initialiser la résolution on se donne u^0 dans tout le domaine et on construit les restrictions u_1^0 de u^0 dans Ω_1 et u_2^0 de u^0 dans Ω_2. On fait de même avec le second membre f. La méthode itérative consiste alors à calculer les $n^{èmes}$ itérés (u_1^n et u_2^n) comme suit.

On résout le problème

$$\left\{ \begin{array}{rcll} -\Delta u_1^n & = & f_1 & \text{dans } \Omega_1, \\ u_1^n & = & 0 & \text{sur } \Gamma_1, \\ u_1^n & = & u_2^{n-1} & \text{sur } \gamma_1^2 \end{array} \right.$$

pour u_1^n, puis le problème

$$\left\{ \begin{array}{rcll} -\Delta u_2^n & = & f_2 & \text{dans } \Omega_2, \\ u_2^n & = & 0 & \text{sur } \Gamma_2, \\ u_2^n & = & u_1^n & \text{sur } \gamma_2^1. \end{array} \right.$$

pour u_2^n.

Dans cette méthode le calcul est fait séquentiellement sur les sous-domaines, car on utilise la connaissance des solutions u_l^n déjà calculées pour calculer u_k^n ($l \neq k$). La figure 2.6 illustre l'exécution de l'algorithme sur le problème modèle 2.9.

Cette méthode est équivalente à une méthode de Gauss-Seidel par blocs. On en déduit que cette variante n'est pas, telle qu'elle est, adaptée au calcul parallèle. Des algorithmes de coloriage (rouge-noir, par exemple) sont utilisés pour compenser ce défaut.

2.3.2.2 Méthode de Schwarz additive

En initialisant la résolution de la même façon que dans le paragraphe précédent, on calcule les itérés successifs comme suit :

$$\begin{cases} & -\Delta u_k^n &=& f_k & \text{dans } \Omega_k, \\ \forall k = 1,2 & u_k^n &=& 0 & \text{sur } \Gamma_k, \\ & u_k^n &=& u_l^{n-1} & \text{sur } \gamma_k^l, \quad \forall l = 1,2,\ l \neq k. \end{cases} \tag{2.8}$$

Cette méthode est adaptée au calcul parallèle car les résolutions sur les sous-domaines peuvent être effectuées simultanément. En effet, le calcul de la solution à l'itéré n ne nécessite que la connaissance des solutions obtenues à l'itéré $n-1$. La figure 2.6 présente l'exécution de l'algorithme sur le problème modèle 2.9.

Cette méthode est équivalente à une méthode de Jacobi par blocs. Habituellement, les algorithmes asynchrones découlent de la méthode de Schwarz additive (voir paragraphe 4.2.1).

2.3.2.3 Comparaison entre Schwarz additive et Schwarz multiplicative

Les méthodes de Schwarz multiplicative et additive diffèrent en termes de vitesse de convergence. Pour examiner le comportement de convergence de ces méthodes, nous allons considérer la résolution du problème modèle dans un domaine unidimensionnel :

$$\begin{cases} -\Delta u &=& 0 & \text{dans }]0,1[, \\ u(0) &=& 0, & \\ u(1) &=& 0. & \end{cases} \tag{2.9}$$

dont la solution triviale est $u = 0$. Les solutions des problèmes locaux sont également des segments de droite.

La figure 2.6 illustre graphiquement la convergence des deux méthodes vers la solution exacte du problème. On vérifie que la méthode additive converge plus lentement que la méthode multiplicative. Selon [149], en pratique, pour des nombreux cas de décomposition en deux sous-domaines, la méthode de Schwarz additive exige le double du nombre d'itérations pour la convergence que la méthode de Schwarz multiplicatif.

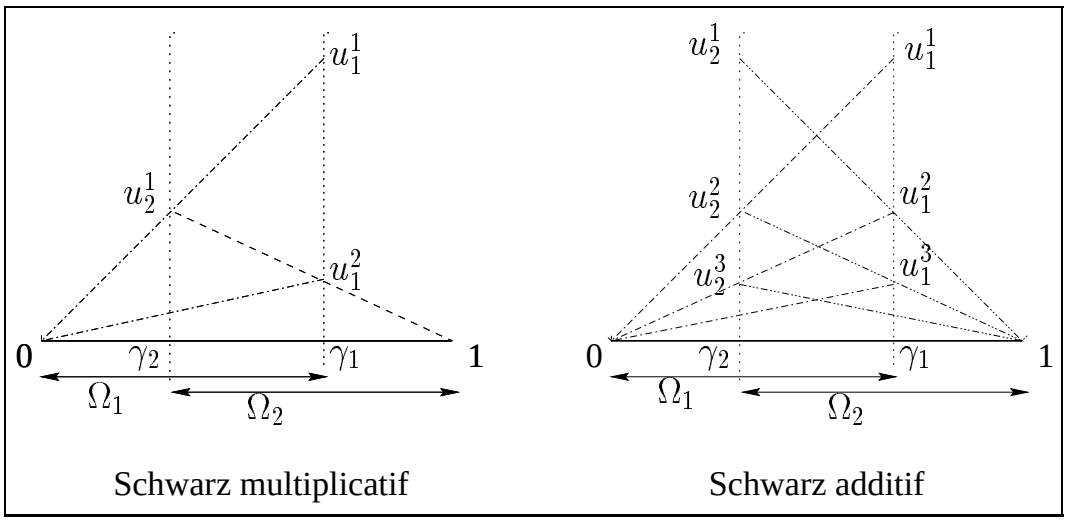

Figure 2.6 *Représentation graphique de la convergence de la méthode de Schwarz.*

La vitesse de convergence de ces méthodes est aussi fortement influencée par la taille du recouvrement. On vérifie, à l'aide des représentations graphiques de la figure 2.7, que la méthode converge plus vite lorsque le recouvrement est grand.

Le recouvrement implique une redondance de calculs car les degrés de liberté sont associés à plusieurs sous-domaines. On en déduit que le coût de calcul des problèmes locaux croît avec la taille du recouvrement. L'intérêt de ces méthodes dépend donc d'un bon compromis entre la vitesse de convergence et le volume de calculs redondants. On remarquera que la vitesse de convergence diminue lorsque l'on augmente le nombre de sous-domaines. Elle peut être améliorée en utilisant comme solution initiale la solution obtenue sur une grille grossière[153].

2.3.2.4 Mise en œuvre parallèle

Après discrétisation des problèmes locaux à l'aide d'une méthode d'éléments finis nous avons à résoudre, pour chaque sous-domaine Ω_k, le système algébrique

$$A_{kk}U_k = F_k.$$

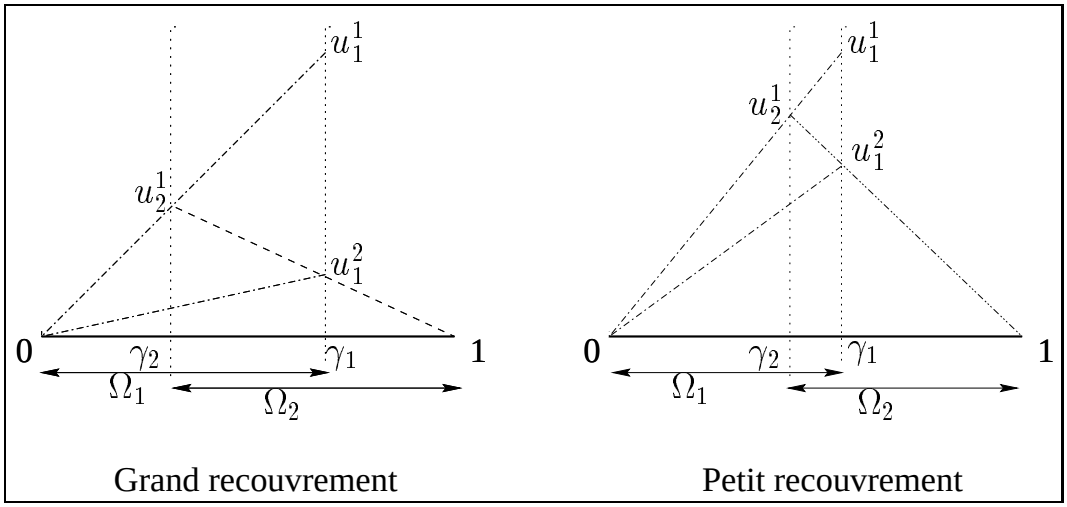

Figure 2.7 *Importance du recouvrement pour la méthode de Schwarz.*

Avec cette notation matricielle, la mise en œuvre parallèle des méthodes de Schwarz additive peut s'écrire sous la forme suivante :

Algorithme 1 : Schwarz additif en parallèle.

pour $n = 1$ **jusqu'à** la convergence **faire**
  pour $k = 1$ **jusqu'à** K (le nombre de sous-domaines) **faire [en parallèle]**
    U_k^n solution de $A_{kk}U_k^n = F_k$
    mise à jour des conditions aux limites entre sous-domaines voisins
  fin pour
fin pour

Cet algorithme est synchrone dans la mesure où le calcul de l'itéré $n+1$ nécessite de tous les résultats obtenus à l'itéré n dans le cas de la méthode de Schwarz additive. La méthode de Schwarz multiplicative s'écrit de manière semblable, avec prise en compte d'un coloriage des processeurs. Il est possible de désynchroniser cet algorithme. Nous en reparlerons dans le paragraphe 4.2.1.

2.3.3 Méthodes sans recouvrement

Ces méthodes ont été développées au cours de années 70 par des mécaniciens souhaitant effectuer des calculs sur des grandes structures. Les ressources de calcul disponibles étant généralement insuffisantes, les scientifiques ont eu l'idée d'effectuer des calculs par sous-structures[93, 47] pour tenir compte des performances des calculateurs de l'époque.

Dans une méthode sans recouvrement, on décompose Ω en K sous-domaines Ω_k ($k = 1, .., K$) tels que $\bar{\Omega} = \cup_{k=1,..,K}\bar{\Omega_k}$, et $\Omega_k \cap \Omega_l = \emptyset$ ($l = 1, .., K, l \neq k$). Lorsqu'elle existe, l'interface γ_{kl} entre les sous-domaines Ω_k et Ω_l ($k \neq l$) est définie par

$$\begin{cases} \gamma_{kl} & = & \partial\Omega_k \cap \partial\Omega_l, \quad \forall l = 1, .., K, l \neq k, \\ \Gamma_k & = & \partial\Omega_k \cap \partial\Omega. \end{cases}$$

Cette approche implique que les nœuds des interfaces γ_{kl} soient connus aux sous-domaines Ω_k et Ω_l. Cependant, ce découpage est qualifié de non recouvrant car les intérieurs des différents sous-domaines sont totalement disjoints.

Un exemple de décomposition à deux sous-domaines est proposé dans la figure 2.8. Il nous sert de référence pour expliquer les méthodes.

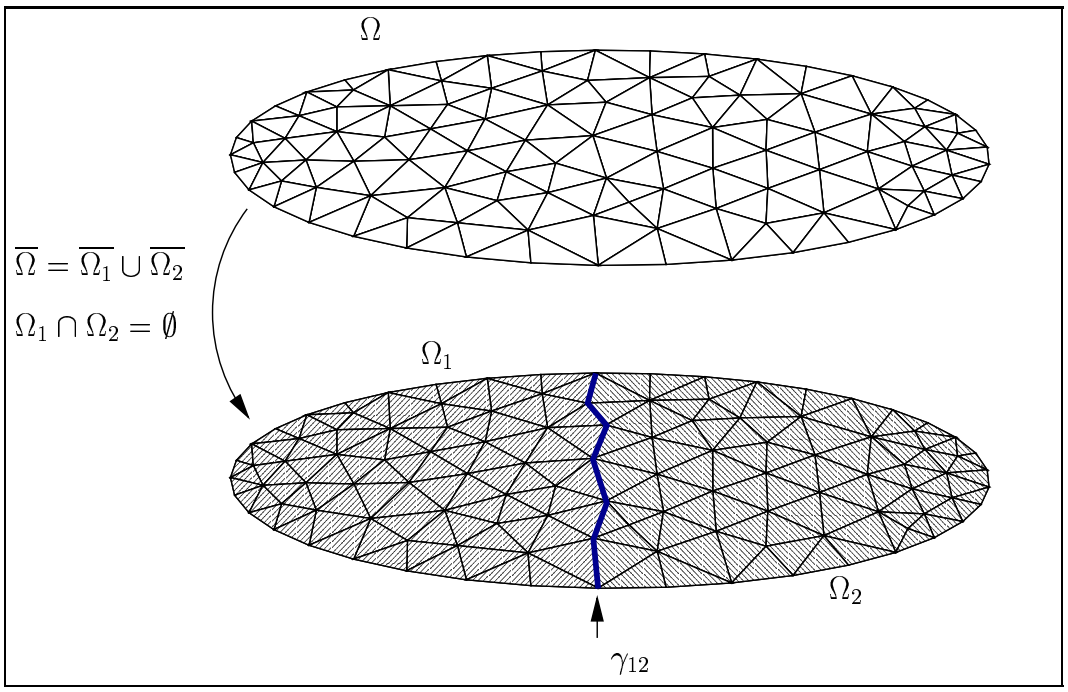

Figure 2.8 *Décomposition sans recouvrement.*

Toutes les méthodes sans recouvrement mènent à la construction d'un **système d'interface**. Une fois ce système identifié, les méthodes directes ou itératives classiques peuvent être utilisées pour le résoudre. Dans ce paragraphe nous présentons trois méthodes sans recouvrement : la méthode du complément de Schur primal, la méthode de Schur dual et la méthode des joints. Pour plus de détails à propos de ces méthodes, nous renvoyons à [64] et à [149].

2.3.3.1 Méthode du complément de Schur ou Schur primal

Après le partitionnement du maillage, on renumérote les degrés de liberté du système de manière à isoler les inconnues associées exclusivement à chacun des deux sous-domaines ainsi que les nœuds de l'interface γ_{12}. Le système algébrique

$$AU = F$$

peut alors s'écrire :

$$\begin{pmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} \begin{pmatrix} U_1 \\ U_2 \\ U_3 \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \\ F_3 \end{pmatrix} \tag{2.10}$$

Dans la décomposition en blocs 2.10 on distingue :

- les sous-blocs diagonaux A_{11} et A_{22} correspondant à la matrice de rigidité associée aux nœuds situés à l'intérieur des sous-domaines Ω_1 et Ω_2 (l'interface n'est pas comprise) ;
- le sous-bloc A_{33} correspondant à la matrice de rigidité associée aux nœuds disposés sur l'interface γ_{12} ;
- les sous-blocs A_{k3} et A_{3k} ($k = 1, 2$) sont identiques à une transposition près. Ils représentent les interactions entre les nœuds de Ω_k ($k = 1, 2$) et les nœuds de l'interface.

L'écriture (2.10) met en évidence l'absence de couplage direct entre les variables situées dans des sous-domaines différents : il n'est pas nécessaire d'assembler le système complet. Dans la résolution, on utilise alors les matrices de rigidité locales suivantes :

$$A_1 = \begin{pmatrix} A_{11} & A_{13} \\ A_{31} & A_{33}^{(1)} \end{pmatrix} \text{ et } A_2 = \begin{pmatrix} A_{22} & A_{23} \\ A_{32} & A_{33}^{(2)} \end{pmatrix}. \tag{2.11}$$

Dans l'écriture (2.11), la matrice $A_{33}^{(1)}$ (respectivement $A_{33}^{(2)}$) représente les interactions entre les nœuds de l'interface calculées à partir des éléments finis correspondant au domaine Ω_1 (respectivement Ω_2). Le sous-bloc A_{33} est la somme de ces deux contributions locales. Le vecteur f_3 peut être décomposé de la même manière.

En supposant que les matrices A_{kk} ($k = 1, 2$) sont inversibles (c'est le cas en éléments finis si la triangulation est régulière [131]), le système (2.10) s'écrit

$$\begin{cases} A_{11}U_1 + A_{13}U_3 = F_1 \\ A_{22}U_2 + A_{23}U_3 = F_2 \\ A_{31}U_1 + A_{32}U_2 + A_{33}U_3 = F_3. \end{cases}$$

Pour éliminer la troisième équation, on exprime U_1 et U_2 en fonction de U_3 :

$$\left\{ \begin{array}{l} A_{11}U_1 = F_1 - A_{13}U_3 \\ A_{22}U_2 = F_2 - A_{23}U_3, \end{array} \right. \tag{2.12}$$

on en déduit que U_3 est solution de

$$(A_{33} - A_{31}A_{11}^{-1}A_{13} - A_{32}A_{22}^{-1}A_{23})U_3 = F_{33} - A_{31}A_{11}^{-1}F_1 - A_{32}A_{22}^{-1}F_2.$$

En posant

$$\begin{array}{l} S = A_{33} - A_{31}A_{11}^{-1}A_{13} - A_{32}A_{22}^{-1}A_{23}, \\ G = F_{33} - A_{31}A_{11}^{-1}F_1 - A_{32}A_{22}^{-1}F_2, \end{array}$$

cette équation s'écrit plus simplement

$$SU_3 = G. \tag{2.13}$$

La matrice S est appelée **complément de Schur de** A. Cette transformation du système d'origine se généralise à plusieurs sous-domaines. La matrice S condensée à l'interface est dense, et pour la construire il faut calculer des inverses des systèmes dans chacun des sous-domaines. Cela est très coûteux pour des problèmes de taille importante.

Les problèmes sur les sous-domaines Ω_1 et Ω_2 pouvant être résolus indépendamment (2.12), on préfère alors résoudre le système d'interface (2.13) sans calculer explicitement le complément de Schur S. Si l'on choisit une méthode de gradient conjugué, les opérations sur S se limitent à des produits matrice-vecteur, qui peuvent être calculés sans l'assemblage explicite de S (voir [137] pour plus de détails). Une alternative pour résoudre (2.13) consiste à utiliser une méthode de point fixe avec préconditionnement[112], par exemple

$$\tilde{S}(U_3^{n+1} - U_3^n) = G - SU_3^n,$$

où $\tilde{S}$ est le préconditionneur de S. Pour la décomposition de référence à deux sous-domaines, un des choix possibles pour $\tilde{S}$ est

$$\tilde{S} = A_{33} - A_{31}(diag((A_{11}))^{-1}A_{13} - A_{32}(diag((A_{22}))^{-1}A_{23},$$

où $diag(A)$ est une matrice diagonale dont la diagonale est celle de A. L'algorithme 2 présente la résolution, par une méthode de point fixe, du système d'interface caractéristique de la méthode du complément de Schur. On considère une décomposition à K sous-domaines couplés par une interface globale $\mathcal{I}$.

Le calcul du vecteur $U_{\mathcal{I}}^n$ (ligne 6 de l'algorithme) peut se faire soit sur un processeur "coordinateur" soit sur tous les processeurs en même temps. Ce calcul nécessite, dans le premier cas, des communications entre le processeur coordinateur et les

processeurs calculant chacun des sous-domaines. Dans le deuxième cas, les communications se passent entre processeurs calculant des sous-domaines qui partagent au moins un nœud, de manière à garantir la continuité de la solution sur les interfaces. Pour la décomposition sans recouvrement de la figure 2.8, cette contrainte de continuité revient à chercher des solutions telles que

$$u_1 = u_2 \text{ sur } \gamma_{12}. \tag{2.14}$$

Algorithme 2 : Méthode de point fixe pour le complément de Schur en parallèle.

1: **pour** $n = 1$ **jusqu'à** la convergence **faire**
2: **pour** $k = 1$ **jusqu'à** K (le nombre de sous-domaines) **faire [en parallèle]**
3: U_k^n solution de $A_{kk}U_k^n = F_k - A_{k\mathcal{I}}U_{\mathcal{I}}^n$
4: $Y_k^n = A_{\mathcal{I}k}U_k^n$
5: **fin pour**
6: $\tilde{S}U_{\mathcal{I}}^n = F_{\mathcal{I}} - A_{\mathcal{I}\mathcal{I}}U_{\mathcal{I}}^{n-1} - \sum_{k=1}^{K} Y_k^n + \tilde{S}U_{\mathcal{I}}^{n-1}$
7: **fin pour**

2.3.3.2 Méthode de Schur dual

Cette méthode a été présentée dans[65, 64], elle est aussi connue sous le nom de FETI (*Finite Element Tearing and Interconnecting*). On distingue les deux méthodes de Schur, primal et dual, par la condition de "continuité" imposée sur les interfaces. Dans la méthode de Schur dual on relaxe la contrainte de continuité en cherchant des solutions telles que

$$\frac{\partial u_1}{\partial n_1} = \frac{\partial u_2}{\partial n_2} \text{ sur } \gamma_{12}. \tag{2.15}$$

Les dérivées normales sont habituellement calculées au milieu des arêtes, ce qui signifie que deux domaines ne s'échangeront des données que s'ils ont une arête en commun. Deux domaines partageant un unique point ne communiquent pas.

Comme l'indique le nom de la méthode, on utilise une formulation variationnelle pour déduire le système algébrique résultant. Dans cette formulation on introduit une variable supplémentaire, un multiplicateur de Lagrange λ, pour prendre en compte la contrainte à l'interface. On écrit alors un problème de minimisation (voir [112] et annexe C) dont la solution est (u_1, u_2, λ). Sous forme matricielle, la méthode de Schur dual consiste à trouver la solution du système algébrique

$$\begin{pmatrix} A_{11} & 0 & B_1^T \\ 0 & A_{22} & B_2^T \\ B_1 & B_2 & 0 \end{pmatrix} \begin{pmatrix} U_1 \\ U_2 \\ \Lambda \end{pmatrix} = \begin{pmatrix} F_1 \\ F_2 \\ 0 \end{pmatrix} \tag{2.16}$$

où l'on distingue :

- les sous-blocs A_{11} et A_{22} correspondant à la matrice de rigidité associée aux nœuds des sous-domaines locaux Ω_1 et Ω_2 (interface comprise) ;
- les sous-blocs B_1 et B_2 correspondant à des matrices booléennes signées permettant de localiser les nœuds de l'interface γ_{12} ainsi que les nœuds voisins de cette interface afin d'effectuer le calcul de la dérivée normale ;
- le vecteur des multiplicateurs de Lagrange Λ représentant les forces d'interaction entre les sous-domaines.

En supposant que les matrices A_{kk} ($k = 1, 2$) sont inversibles, on déduit de (2.16) les égalités

$$\begin{cases} A_{11}U_1 = F_1 - B_1^T\Lambda, \\ A_{22}U_2 = F_2 - B_2^T\Lambda \end{cases} \tag{2.17}$$

et

$$\{ \; B_1U_1 + B_2U_2 = 0. \tag{2.18}$$

En remplaçant U_1 et U_2 issus de (2.17) en fonction de Λ dans (2.18), on obtient un système d'interface dont les inconnues sont les multiplicateurs de Lagrange Λ

$$S\Lambda = G, \tag{2.19}$$

où

$$S = B_1A_{11}^{-1}B_1^T + B_2A_{22}^{-1}B_2^T \tag{2.20}$$

et

$$G = B_1A_{11}^{-1}F_1 + B_2A_{22}^{-1}F_2. \tag{2.21}$$

L'algorithme d'Uzawa, utilisé pour résoudre (2.19), fait apparaître le caractère parallèle de la méthode. Soit Λ_{12}^0 donnés, à l'itération n on suppose les Λ_{12}^n connus et on calcule U_1^n, U_2^n et Λ_{12}^{n+1} en résolvant (2.17), puis en calculant le multiplicateur de Lagrange comme suit

$$\Lambda^{n+1} = \Lambda^n + \rho(B_1U_1^n + B_2U_2^n). \tag{2.22}$$

On doit choisir ρ de manière que l'algorithme converge. Pour approfondir la question, se reporter à [45].

La méthode de Schur dual est plus facile a mettre en œuvre que la méthode de Schur primal, car il n'y a pas besoin de numéroter les variables du système de façon à séparer les nœuds internes des nœuds d'interface. De plus, cette méthode est adaptée au calcul parallèle, car les échanges de données pour le calcul d'interface sont limités aux sous-domaines partageant une arête. En ce sens, les interfaces entre sous-domaines sont gérées de manière disjointe.

2.3.3.3 Remarques sur les méthodes sans recouvrement

Dans les deux paragraphes précédents nous avons utilisé les formalismes mathématiques habituels. Au final, on cherche à résoudre un système linéaire du type

$$CX = D. \tag{2.23}$$

Dans le cas de deux sous-domaines, la matrice C et les vecteurs X et D sont définis comme suit :

	$SchurPrimal$	$SchurDual$
$C = \begin{pmatrix} C_{11} & 0 & C_{13} \\ 0 & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{pmatrix}$	$\begin{pmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix}$	$\begin{pmatrix} A_{11} & 0 & B_1^T \\ 0 & A_{22} & B_2^T \\ B_1 & B_2 & 0 \end{pmatrix}$
$X = \begin{pmatrix} X_1 \\ X_2 \\ X_3 \end{pmatrix}$	$\begin{pmatrix} U_1 \\ U_2 \\ U_3 \end{pmatrix}$	$\begin{pmatrix} U_1 \\ U_2 \\ \Lambda \end{pmatrix}$
$D = \begin{pmatrix} D_1 \\ D_2 \\ D_3 \end{pmatrix}$	$\begin{pmatrix} F_1 \\ F_2 \\ F_3 \end{pmatrix}$	$\begin{pmatrix} F_1 \\ F_2 \\ 0 \end{pmatrix}$
$S = C_{33} - \sum_{k=1}^{2} C_{3k} C_{kk}^{-1} C_{k3}$	$A_{33} - \sum_{k=1}^{2} A_{3k} A_{kk}^{-1} A_{k3}$	$\sum_{k=1}^{2} B_k A_{kk}^{-1} B_k^T$
$G = D_3 - \sum_{k=1}^{2} C_{3k} C_{kk}^{-1} D_k$	$F_3 - \sum_{k=1}^{2} A_{3k} A_{kk}^{-1} F_k$	$\sum_{k=1}^{2} B_k A_{kk}^{-1} F_k$

Ce formalisme permet d'utiliser les mêmes algorithmes de point fixe ou gradient conjugué pour l'une ou l'autre méthode. Comme nous le verrons ci-dessous, cela reste vrai pour la méthode des joints.

Remarque : les matrices A_{11} et A_{22} diffèrent d'une méthode à l'autre.

2.3.3.4 Méthode des joints

Les méthodes sans recouvrement que nous venons de présenter se prêtent aux maillages et décompositions conformes, c'est à dire, où les triangulations se raccordent aux interfaces, et où le découpage en sous-domaines suit aussi les principes de conformité (voir section 2.1.2). La méthode des éléments joints, proposée dans [17], permet de relaxer cette contrainte : les maillages ne sont plus nécessairement conformes aux interfaces, ce qui peut faciliter l'adaptation de maillage.

Le système d'interface correspondant à la méthode des joints est le même que celui correspondant à la méthode de Schur dual (2.19). Les différences entre les deux méthodes résultent du choix des fonctions de bases utilisées des deux côtés de l'interface. La méthode de Schur dual utilise les fonctions éléments finis classiques construites de manière identique des deux cotés de l'interface. La méthode des joints travaille avec des bases différant d'un côté (appelé "maître") à l'autre ("appelé esclave") de l'interface. L'espace des fonctions de discrétisation est constitué des fonctions éléments finis pour les noeuds internes à l'interface, et de deux fonctions spécifiques pour les noeuds extrêmes de celle-ci[112]. Lors de la construction des matrices d'interface, le calcul des éléments de B est réalisé à partir de la discrétisation la plus fine (côté maître). Une fois les matrices B_1 et B_2 construites, la résolution du système d'interface pour la méthode des joints est identique à celle employée par la méthode de Schur dual.

Deuxième partie

Solutions

3

Outils pour la résolution d'EDP en parallèle : état de l'art

Sommaire

3.1 Des approches complémentaires 66
3.1.1 Outils issus des mathématiques 66
3.1.2 Outils informatiques 69
3.2 Étude de quelques outils représentatifs 72
3.2.1 PSPARSLIB 72
3.2.2 POOMA 77
3.3 Présentation de AHPIK 80

Les progrès conjugués des méthodes numériques et des techniques informatiques assurent un renouvellement permanent du domaine de recherche consacré à la résolution de problèmes d'EDP en parallèle. Dans ce chapitre nous nous intéressons aux techniques et méthodes ayant abouti à la conception d'un outil informatique.

La plupart de ces outils n'emploient pas les techniques de multiprogrammation légère. Certains projets (voir paragraphe 3.2.2) s'appuient sur des noyaux exécutifs offrant des processus légers, mais l'intégration de la multiprogrammation légère est limitée à des multiprocesseurs à mémoire partagée. L'utilisation efficace de processus légers pour le parallélisme sur mémoire distribuée aussi bien que sur mémoire partagée reste un domaine de recherche ouvert.

Nous distinguons (paragraphe 3.1) trois groupes d'outils : les outils "algébriques" qui se concentrent sur la résolution de systèmes linéaires, les outils "numériques" qui résultent d'une réflexion mathématique menée à partir des équations continues du problème, et les modèles ou environnements de programmation adaptés, entre autres, à la résolution en parallèle par une méthode de décomposition de domaine. Ensuite, le paragraphe 3.2 présente, de manière plus précise, PSPARSLIB qui appartient à la classe des outils algébriques, POOMA qui est un environnement informatique dédié à la simulation numérique. Le dernier paragraphe est dédié à une courte introduction à notre logiciel, AHPIK, qui appartient à la classe des outils "numériques" de décomposition de domaine.

3.1 Des approches complémentaires

Pour construire un outil de résolution de problèmes d'EDP, les développeurs s'intéressent à des aspects mathématiques (problème à résoudre, choix de méthodes de discrétisation (§2.1), choix de méthodes de résolution (§2.1.4 et §2.3)), et à des aspects informatiques (choix du langage et du modèle de programmation).

3.1.1 Outils issus des mathématiques

Pour faciliter le développement d'applications de simulation numérique, certains logiciels fournissent un ensemble de composants intervenant dans la résolution de problèmes d'EDP. On trouve ainsi des environnements intégrant, par exemple, la discrétisation par éléments finis, l'adaptation de maillage, et la résolution de systèmes linéaires. Dans ce paragraphe nous présentons tout d'abord les outils issus de la recherche sur la résolution parallèle des systèmes linéaires, puis des logiciels

développés par des scientifiques des domaines d'application.

3.1.1.1 Bibliothèques pour la résolution de systèmes linéaires

Dans notre contexte, cette approche est abstraite au sens où elle ne permet pas de prendre en compte certains détails spécifiques au calcul d'EDP. Par exemple, les étapes d'adaptation de maillage ne peuvent être réalisées lorsque l'on ne dispose pas des coordonnées physiques du maillage. Il en va de même lorsque l'on veut utiliser une solution calculée sur une grille grossière pour initialiser la résolution itérative de la méthode de décomposition de domaine choisie [138].

On trouve actuellement de nombreuses bibliothèques dédiées à la résolution de systèmes linéaires sur machine parallèle. L'ensemble des techniques étant très vaste, ces bibliothèques sont en général spécialisées soit dans les méthodes itératives soit dans les méthodes directes.

Méthodes directes : En ce qui concerne les méthodes directes, les efforts se concentrent sur la parallélisation de la factorisation de la matrice et du second membre, ce qui représente le calcul le plus coûteux. Pour plus de détails sur la résolution parallèle de systèmes linéaires creux le lecteur peut se reporter à l'article [89] et au livre [56]. Les outils offrant des méthodes directes parallèles sont en général plus récents que les outils offrant des méthodes itératives. Parmi ces outils on peut citer CAPSS[90], PaStiX[92] et PSPASES[88], spécialisés dans les systèmes creux symétriques définis positifs, et SuperLU[109] et S+[76] pour des systèmes non-symétriques. D'après [2], actuellement les seules bibliothèques fournissant des solveurs adaptés à ces deux types de systèmes sont MUMPS[1] et SPOOLES[5].

Méthode itératives : Elles sont relativement faciles à paralléliser car seuls des produits matrice-vecteur et des produits scalaires interviennent dans leur algorithmique. Cependant la convergence de ces algorithmes dépend en grande partie de l'utilisation de préconditionneurs parallèles. Le passage à l'échelle d'un grand nombre de processeurs est un aspect critique pour ces méthodes car leur vitesse de convergence décroît avec le nombre de processeurs. Les outils offrant ce type de méthode sont nombreux, une revue des principales bibliothèques pour la résolution itérative de systèmes linéaires est disponible dans [60]. Parmi les bibliothèques parallèles on peut citer PSPARSLIB[139], Aztec[95], BlockSolve[99], PETSc[10], ParPre[37] et PIM[48].

Dans ces outils, les méthodes de décomposition de domaine fonctionnent comme des préconditionneurs : la solution du problème d'EDP original est typiquement obtenue à l'aide d'une méthode itérative de Krylov, précondi-

tionnée par un opérateur couplant les solutions des sous-problèmes. Les préconditionneurs basés sur la méthode de Schwarz sont les plus courants. Les bibliothèques PSPARSLIB et ParPre se distinguent par l'inclusion de préconditionneurs basés sur la méthode du complément de Schur.

Ces bibliothèques parallèles travaillent sur une représentation distribuée du système linéaire à résoudre. En général, cette distribution consiste à affecter un ensemble de lignes de la matrice creuse à chaque processeur : les processeurs peuvent recevoir des lignes contigües (par exemple dans PETSc et BlockSolve), ou bien un ensemble arbitraire de lignes (par exemple dans Aztec et PSPARSLIB).

Remarque : En ce qui concerne la mise en œuvre du parallélisme, l'approche la plus répandue est fondée sur l'utilisation de MPI pour l'échange de messages sur les architectures à mémoire distribuée. La bibliothèque SuperLU utilise des processus légers pour les architectures à mémoire partagée.

3.1.1.2 Environnements pour la résolution numérique d'EDP en parallèle

Il s'agit d'outils offrant les composants pour la simulation numérique par EDP : éléments finis, maillages adaptatifs, solveurs parallèles. Essentiellement conçus pour la résolution de problèmes donnés, ils ont été étendus progressivement. Les étapes de conception et d'amélioration de chacun d'entre eux sont ainsi très différentes d'un outil à l'autre. Nous avons décidé de présenter brièvement trois d'entre eux. La bibliothèque AHPIK, que nous avons construite, appartient à cette classe d'outils. Elle est détaillée dans le paragraphe 3.3.

Diffpack

Diffpack[27] est un environnement pour le développement d'applications nécessitant la résolution de problèmes d'EDP. Les bibliothèques composant cet environnement offrent une large gamme de fonctionnalités : maillages structurés ou non-structurés, adaptation de maillage, diverses méthodes de discrétisation et de résolution de systèmes linéaires. Le calcul parallèle n'a pas été pris en compte lors de la conception initiale de cet environnement (au début des années 90). La parallélisation des solveurs de systèmes linéaires dans Diffpack a suivi deux approches[31] : l'une est basée sur la parallélisation des opérations d'algèbre linéaire pour les solveurs itératifs (multiplications matrice-vecteur, produits scalaires de vecteurs), l'autre est fondée sur l'utilisation de méthodes de décomposition de domaine de type Schwarz.

Ces deux alternatives sont complémentaires, étant donné que les méthodes de décomposition de domaine peuvent être utilisées comme préconditionneurs pour les solveurs itératifs parallèles. Dans les deux approches, les communications entre processeurs ont été mises en œuvre en utilisant MPI.

PadFEM

PadFEM[54] est un environnement pour la mise en œuvre d'applications de simulation numérique utilisant la méthode des éléments finis et l'adaptation de maillage sur calculateur parallèle. Les facilités offertes par cet environnement incluent : génération et partitionnement de maillage, éléments finis pour les équations de Poisson et Navier-Stokes, solveurs basés sur décomposition de domaine, estimateurs d'erreurs, raffinement de maillage et répartition dynamique de charge. Les structures de données représentant les maillages et les éléments finis sont spécialement conçues pour les calculs adaptatifs en parallèle, facilitant le transfert d'éléments d'un sous-domaine à l'autre afin de répartir la charge de calcul. Les solveurs de systèmes linéaires utilisent des méthodes multi-grilles ou des méthodes itératives de type gradient conjugué, ces dernières pouvant être couplés avec un préconditionneur basé sur une méthode de décomposition de domaine de type Schur.

UG (*Unstructured Grids*)

UG[11] est un outil pour la résolution d'EDP sur maillages non-structurés utilisant des méthodes multi-grilles adaptatives. Cet environnement est construit de forme hiérarchique, s'appuyant sur un outil (DDD[22], voir section 3.1.2.2) permettant de manipuler des structures de données irrégulières et des objets distribués sur machine parallèle. Un ensemble de méthodes numériques de résolution d'EDP est implémenté sur cette couche de base, la distribution de données sur les processeurs étant transparente aux algorithmes utilisés.

3.1.2 Outils informatiques

Les applications du calcul scientifique sont de grandes consommatrices de puissance de calcul. Elles sont aussi à l'origine d'une demande croissante pour des logiciels numériques fiables et robustes, facilement adaptables à des différents problèmes. La « réutilisabilité », la flexibilité et l'expressivité des codes source sont des caractéristiques de plus en plus essentielles dans le développement de ces applica-

tions. On fait donc appel à des méthodes et outils informatiques généraux pour la programmation de celles-ci.

3.1.2.1 Le paradigme objet pour le calcul scientifique numérique

Au cours des dernières années, la programmation par objets dans le contexte de l'implémentation d'outils dédiés au calcul scientifique a suscité beaucoup d'intérêt (voir par exemple [132] ou [72]). Les langages orientés objet comme C++ et Java ont ainsi conquis un nombre considérable d'adeptes, dans un milieu où Fortran et C ont toujours été les langages favoris.

La programmation par objets offre des possibilités intéressantes pour le développement d'applications scientifiques. Les principaux atouts sont détaillés ci-dessous :

- La notion de **type abstrait de données** (représentée dans le langage C++ par le concept de **classe**) permet de séparer interface d'utilisation et mise en œuvre. Il est alors possible de travailler dans un niveau d'abstraction plus proche des formulations mathématiques et numériques, sans se préoccuper des détails d'implémentation. Les classes représentant matrices ou vecteurs sont un très bon exemple car l'utilisateur n'a besoin de connaître que les opérations (addition, multiplication, etc.) disponibles pour manipuler ces objets. Dans ce cas les structures de données et les algorithmes qui caractérisent la mise en œuvre restent cachés.
- Le **polymorphisme** offre la possibilité de travailler avec des différents objets implémentant une même fonction de l'interface applicative. Cela permet une meilleure lisibilité du code du fait de l'emploi d'un nom unique à une même fonction. On utilisera par exemple un nom générique pour désigner les fonctions effectuant un produit matrice-vecteur sans se soucier des formats particuliers de matrice et vecteur utilisés. En C++, le polymorphisme est réalisé par des **fonctions virtuelles** ou des **« templates »**[62].
- La notion d'**héritage** permet de définir de nouveaux objets par une combinaison des propriétés de différents types d'objets ou par un ajout de propriétés à un objet qui est déjà défini. Cela favorise évidemment la réutilisation de code et son extensibilité.

Les caractéristiques inhérentes au paradigme objet sont aussi très utiles dans le contexte du parallélisme. Ce paradigme sert notamment pour encapsuler les détails liées à la gestion du parallélisme dans une application, ce qui augmente sa portabilité et simplifie sa programmation. De nombreux environnements de programmation marient le parallélisme et les objets. Les exemples représentatifs sont MPI++[100], C++//[33], MPC++[97], Java//[35], Do ![106], JavaParty[127] et Eiffel//[34]. Le

lecteur est renvoyé à [77] pour une présentation des diverses extensions parallèles spécifiques à C++.

3.1.2.2 Outils pour la gestion de structures de données distribuées

Lors de la résolution de problèmes d'EDP en parallèle, les structures de données représentant les maillages, les matrices, les vecteurs, etc., sont souvent distribuées sur un ensemble de processeurs. La gestion de ces structures peut s'avérer une tâche complexe. Les difficultés proviennent en partie de la dualité entre les représentations locale et globale des ces structures. Par exemple, lorsque une matrice creuse est distribuée, chaque processeur a besoin de connaître la correspondance entre les numérotations locale et globale des éléments qui lui ont été affectés. La modification des structures au cours du calcul ajoute un degré de difficulté à leur gestion car la modification des structures locales implique généralement des communications pour préserver la cohérence avec la représentation globale. Dans ce contexte, un certain nombre d'outils ont été conçus pour faciliter la gestion de structures de données distribuées. Une grande partie d'entre eux fournissent des structures adaptées aux calculs évoluant dynamiquement. Ces outils intègrent alors des mécanismes de répartition dynamique de charge. De manière générale, ils peuvent être utilisés comme une couche logicielle de base pour des outils consacrés aux méthodes de résolution d'EDP.

Les outils permettant la gestion de grilles (maillages structurés) distribuées sont les plus nombreux. Ils sont particulièrement adaptés à la résolution de problèmes d'EDP par la méthode de différences finies. Parmi eux figurent A++/P++[125], DAGH[124], KeLP[67] et POOMA[133]. D'autres outils sont présentés dans [123]. Bien que les fonctionnalités varient d'un outil à l'autre, ils fournissent tous un support haut niveau pour la définition et la manipulation de grilles ou tableaux multidimensionnels distribués. Les détails concernant la mise en œuvre des communications et synchronisations sur machine parallèle à mémoire distribuée sont alors cachés à l'utilisateur.

La gestion de maillages non-structurés distribués est plus complexe et les outils conçus dans cet esprit sont peu fréquents. Une structure générale pour la représentation distribuée de maillages non-structurés est proposée dans [68] et incorporé à l'outil PMDB[122]. Celui-ci prend en charge les représentations locale et globale des maillages et de ses frontières, ainsi que le transfert d'éléments d'un processeur à l'autre suite à une adaptation de maillage. L'outil DDD[22] fournit, quant à lui, un support pour les structures de données irrégulières et les objets distribués.

3.2 Étude de quelques outils représentatifs

3.2.1 PSPARSLIB

PSPARSLIB[139] est une bibliothèque pour la résolution itérative de systèmes linéaires creux sur architecture parallèle. Les fonctionnalités offertes par PSPARSLIB sont basées sur une représentation distribuée du système à résoudre, autour de laquelle s'articulent outils de pré-traitement (partitionnement, etc.), noyaux de calcul algébrique (multiplication matrice-vecteur, produit scalaire, etc.), préconditionneurs et solveurs itératifs. Dans cette bibliothèque, les méthodes de décomposition de domaine sont employées sous forme de préconditionneurs pour accélérer la convergence de solveurs itératifs parallèles. Des méthodes avec et sans recouvrement sont disponibles dans PSPARSLIB (Schwarz additif et multiplicatif, et complément de Schur). Cette bibliothèque est écrite en Fortran et utilise MPI pour la mise en œuvre des échanges de messages entre processeurs.

3.2.1.1 Structures de données

Les solveurs parallèles de PSPARSLIB travaillent sur un système d'équations linéaires distribué sur plusieurs processeurs. Cette distribution a son origine dans le partitionnement du graphe d'adjacence associé à la matrice du système. Étant donné une matrice creuse A de dimension $N \times N$, le graphe $G = (V, E)$ associé à cette matrice est défini par

- V un ensemble de nœuds $V = \{1, 2, ..., N\}$, chaque nœud correspondant à une inconnue du système ;
- E un ensemble d'arêtes $E = \{(i, j) \mid a_{ij} \neq 0\}$ représentant les dépendances entre les équations du système.

Ce graphe est non-orienté si la matrice est symétrique, les arêtes (i, j) coïncidant avec les arêtes (j, i). La figure 3.1 illustre cette dualité entre la représentation matricielle d'un système à 12 inconnues et le graphe associés.

Étant donné un partitionnement de l'ensemble V en p sous-ensembles $V_1, ..., V_p$, le graphe G se décompose en p sous-graphes $G_i = (V_i, E_i)$. Les sous-ensembles E_i expriment les dépendances des nœuds appartenant à V_i par rapport à d'autres nœuds inclus ou non dans V_i. Les sous-graphes G_i s'apparentent à des sous-domaines. Dans une approche plus orientée vers les EDP, on effectue le partitionnement des éléments du maillage plutôt que de ses nœuds. Cela ne constitue pourtant pas une limitation à l'utilisation de PSPARSLIB, sachant qu'un partitionnement par éléments est équivalent au partitionnement des nœuds d'un graphe dual au graphe que

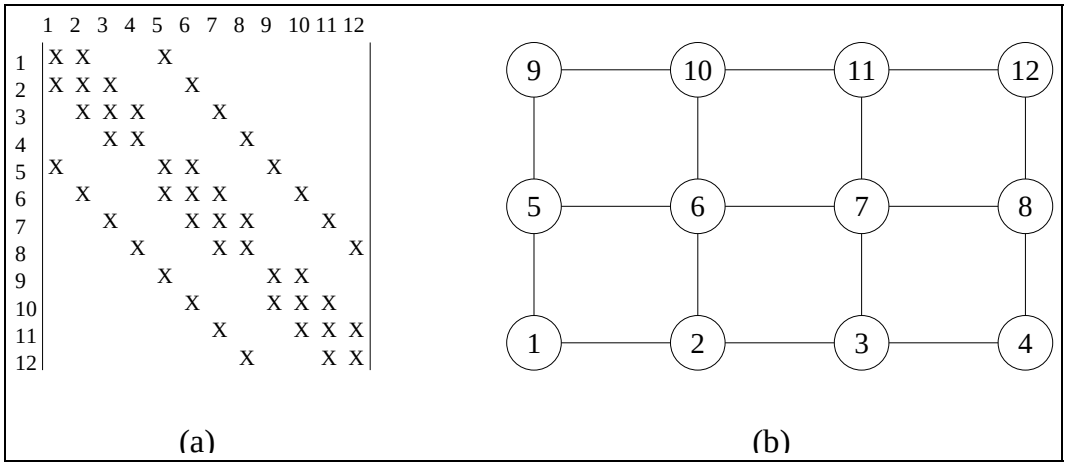

Figure 3.1 *(a) Profil d'une matrice creuse à 12 inconnues. (b) Graphe d'adjacence associé.*

nous avons défini précédemment. Le graphe dual de $G = (V, E)$ est le graphe $G' = (V', E')$ où $V' = E$ et E' est l'ensemble des paires d'arêtes de E qui partagent un nœud de V.

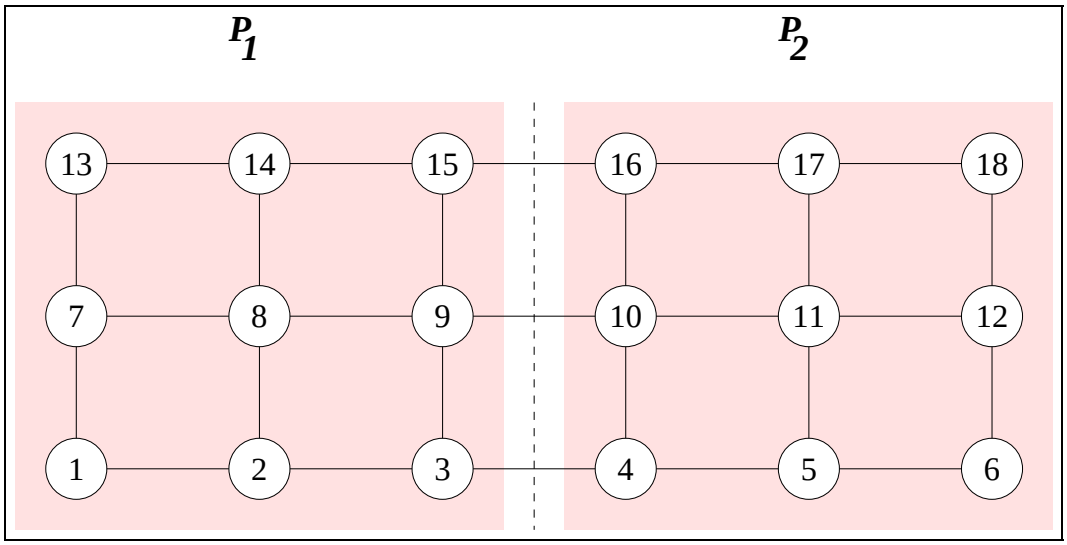

Figure 3.2 *Partitionnement d'un graphe en 2 sous-graphes.*

La figure 3.2 illustre le partitionnement d'un graphe à 18 nœuds en $p = 2$ sous-graphes. Dans PSPARSLIB, chaque sous-graphe est affecté à un processeur distinct, p désignant à la fois le nombre de sous-graphes et le nombre de processeurs. Cette distribution du graphe revient à placer un ensemble d'équations, i.e. de lignes du système linéaire, sur chaque processeur.

Étant donné un partitionnement convenable d'un graphe G, PSPARSLIB distingue plusieurs ensembles de nœuds :

- nœuds internes associés au sous-graphe G_i (ou au processeur i) : nœuds exclusivement connectés à d'autres nœuds appartenant à G_i ;
- nœuds d'interface locaux : nœuds connectés à des nœuds appartenant à G_i, mais aussi à d'autres sous-graphes ;
- nœuds d'interface externes : nœuds appartenant à d'autres sous-graphes mais qui sont connectés à des nœuds appartenant à G_i.

Dans le cas la figure 3.2, l'ensemble des nœuds internes à G_1 est $\{1, 2, 7, 8, 13, 14\}$, l'ensemble des nœuds d'interface locaux de G_1 est $\{3, 9, 15\}$ et l'ensemble de nœuds d'interface externes à G_1 est $\{4, 10, 16\}$. Dans ces ensembles la numérotation des nœuds est globale, mais PSPARSLIB autorise la création de ces ensembles à l'aide des numérotations locales.

Ces ensembles de nœuds jouent un rôle fondamental dans PSPARSLIB car leur identification permet de préparer les structures de données locales pour les communications nécessaires lors du calcul itératif. En effet, les matrices et les vecteurs locaux sont construits par permutation des équations affectées à chaque processeur de façon à séparer les nœuds internes des nœuds d'interface locaux. Les nœuds d'interface externe sont également pris en compte dans ces structures.

Quitte à procéder à une renumérotation des nœuds, la matrice creuse distribuée selon PSPARSLIB prend la forme présentée dans la figure 3.3.

On y distingue la matrice rectangulaire A^1_{loc} de dimension $N_1 \times N$ représentant les N_1 équations (ou nœuds) affectées au processeur P_1 et composée des deux blocs :

- B^1_{loc} est une matrice carrée de dimension $N_1 \times N_1$. Les éléments a_{ij} de ce bloc sont tels que j correspond à une variable locale, c'est-à-dire, à un nœud affecté au processeur i. Dans B^1_{loc} les lignes en pointillé séparent les nœuds internes à G_1 des nœuds d'interface locaux de G_1 ;
- B^1_{ext} : un bloc d'éléments a_{ij} tels que j correspond à une variable externe.

Tous les vecteurs associés au processus de résolution (vecteur de solution, second membre, etc.) sont partitionnés et organisés en conformité avec la matrice. Des structures de données additionnelles, notamment une liste de processeurs voisins et une liste de nœuds d'interface locaux, sont également employées pour gérer les communications.

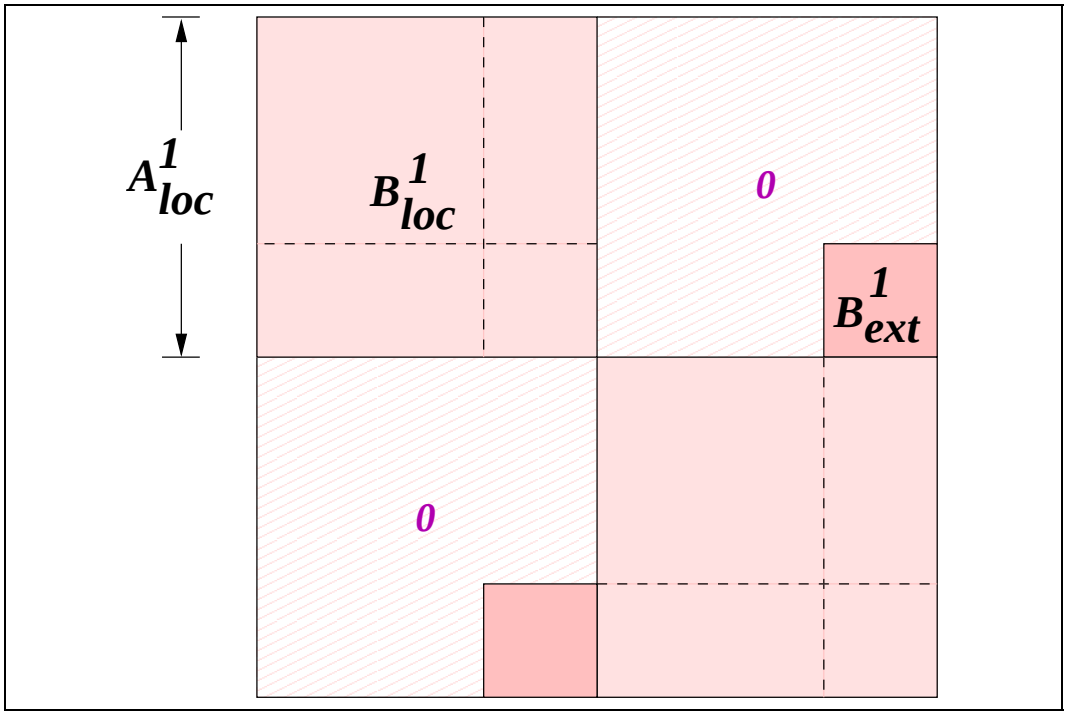

Figure 3.3 *Vision globale d'une matrice creuse distribuée selon PSPARSLIB.*

On notera que des structures de données similaires sont utilisées par les bibliothèques Aztec[95], BlockSolve[99] et PETSc[10].

3.2.1.2 Algorithmes parallèles

Dans PSPARSLIB, les algorithmes de résolution du système linéaire distribué s'articulent autour de solveurs itératifs basés sur des méthodes de Krylov telles que la méthode du gradient conjugué ou la méthode GMRES (*Generalized Minimum Residual*). Dans ce type de solveur, les calculs se résument à des combinaisons linéaires de vecteurs, produits scalaires et produits matrice-vecteur. PSPARSLIB implémente des versions parallèles de ces opérations en utilisant les structures de données décrites dans la section précédente.

Les combinaisons linéaires de vecteurs distribués sont des opérations totalement parallèles qui ne nécessitent pas de communications. Par exemple, une somme de deux vecteurs distribués n'a besoin que de la somme des représentations locales de ces vecteurs. Pour les produits scalaires distribués, l'algorithme consiste à effectuer le produit scalaire des vecteurs locaux, suivi d'une somme globale des résultats locaux. Dans PSPARSLIB, cette opération est réalisée à l'aide d'une primitive de

communication globale offerte par MPI.

Les produits matrice-vecteur sont parmi les opérations les plus coûteuses dans les solveurs itératifs contenus dans PSPARSLIB. Ils ne sont surpassés en volume de calcul que par la phase de préconditionnement de la matrice. Étant donnés une matrice et un vecteur distribués selon le format décrit en 3.2.1.1, l'algorithme parallèle pour le calcul du produit matrice-vecteur s'effectue comme suit :

1. multiplication du bloc B_{loc} de la matrice par le vecteur local ;
2. envoi des composants du vecteur local correspondant aux nœuds d'interface locaux ;
3. réception des valeurs correspondant aux nœuds d'interface externes ;
4. multiplication du bloc B_{ext} de la matrice par le vecteur de valeurs externes, le résultat obtenu est additionné à celui de la première multiplication.

L'algorithme mis en œuvre dans PSPARSLIB effectue simultanément les étapes 1, 2 et 3 à l'aide de primitives de communication asynchrones disponibles dans MPI. Les communications sont de type point-à-point, elles se déroulent entre processeurs voisins.

Le calcul d'un préconditionneur pour la matrice distribuée est nécessaire pour accélérer la convergence des solveurs itératifs disponibles dans PSPARSLIB. Les préconditionneurs parallèles basés sur méthodes de décomposition de domaine sont mis en œuvre selon des algorithmes très similaires à ceux décrits dans le chapitre 2. On remarque notamment que les communications se déroulent uniquement entre processeurs voisins. Le schéma précis de communication varie selon la méthode (par exemple, pour la méthode multiplicative de Schwarz, les communications obéissent à un algorithme de coloriage des processeurs). Habituellement, le préconditionnement ne nécessite que de quelques itérations de la méthode de décomposition de domaine, la convergence étant contrôlée par la méthode de résolution itérative globale (GMRES, par exemple).

3.2.1.3 Appréciation de l'outil

PSPARSLIB est parmi les seules bibliothèques à fournir une variété de préconditionneurs parallèles basés sur les méthodes de décomposition de domaine. L'approche est suffisamment générale pour permettre la résolution de systèmes provenant de différents types d'applications.

Cependant, cette bibliothèque ne fournit pas de méthode étroitement liée à la

résolution d'EDP. Pour l'instant, il n'y a pas de support pour le partitionnement dynamique des sous-domaines (création ou élimination de sous-domaines), ni pour la résolution par un schéma asynchrone.

3.2.2 POOMA

POOMA (*Parallel Object-Oriented Methods and Applications*) est un environnement exploitant les paradigmes de programmation par objets et de parallélisme de données pour faciliter le développement d'applications de calcul scientifique sur machines parallèles.

3.2.2.1 Structures de données

La bibliothèque POOMA s'articule autour de types de données (ou classes d'objets) orientés vers le calcul scientifique. Les objets de type `Array` sont les plus généraux : ils représentent des tableaux multidimensionnels supportant différents schémas d'indexation. Leur conception s'inspire des facilités fournies par le langage FORTRAN-90. Cette classe d'objets générique est notamment paramétrée par le nombre de dimensions et le type des données stockées. Par exemple, les déclarations

```
Array<1, int> V(20);
Array<2, float> M(10,10);
```

représentent, respectivement, un tableau unidimensionnel (vecteur) de 20 entiers et un tableau bidimensionnel (matrice) de taille 10×10 composé de réels. Un troisième paramètre, optionnel, permet de définir différents types de stockage pour ces tableaux (dense, creux, centralisé ou distribué, entre autres).

POOMA tire parti du paradigme de programmation par objets pour permettre l'expression naturelle des opérations manipulant les objets de type `Array`. Par exemple, sous réserve de la définition

```
Array<1, double> A(N), B(N), C(N), D(N);
```

on peut écrire

```
A = B + C * sin(D);
```

Ce type de notation permet de cacher les détails de mise en œuvre (boucles sur les indices de 1 à N), mettant en évidence les opérations mathématiques sur les vecteurs. La mise en œuvre performante de ces opérations dans POOMA est assurée par l'utilisation de techniques de programmation telles que les « expression templates » [158].

Cette bibliothèque offre plusieurs possibilités pour l'indexation d'un `Array`, permettant notamment de sélectionner un intervalle, contigu ou non, de valeurs dans un tableau. Cela se fait à l'aide des classes `Interval` et `Range` fournies par POOMA. Par exemple :

```
Array<2, double> M(N, N);
Range<1> I(0, N-1, 2);
Interval<1> J(0, N-1);
```

Ci-dessus, l'objet de type `Range` sert à sélectionner les lignes paires de la matrice (ou tableau bidimensionnel) `M`, tandis que l'objet de type `Interval` permet d'indexer tous les éléments de la première ligne de `M`. Il est ainsi possible d'exprimer une opération (affectation, dans cet exemple) sur un sous-ensemble d'éléments d'un tableau multidimensionnel. Par exemple :

```
M(I,J) = 0.5;
```

POOMA offre un support très intéressant pour les calculs basés sur « stencils » sur des espaces discrets. L'exemple typique de « stencil » est celui utilisé pour la discrétisation du Laplacien en différences finies : en effet, dans un schéma explicite d'ordre 1 on utilise les points immédiatement voisins d'un point donné pour mettre à jour sa valeur. Les calculs manipulant des « stencils » supposent établie une relation entre un domaine spatial de calcul et un ensemble de valeurs. Dans POOMA, le domaine spatial est traité par l'intermédiaire de la classe `Geometry` qui représente un ensemble de points définis par leurs coordonnées géométriques. La définition de valeurs dans le domaine discret se fait par l'application de « stencils » sur des objets de la classe `Field`. POOMA permet de définir des conditions aux limites ayant une influence sur un `Field` et fournit les moyens pour les prendre en compte automatiquement lorsqu'il est nécessaire.

POOMA fournit également un support aux techniques de simulation utilisant des particules. Ces techniques sont utilisées, par exemple, en conjonction avec des méthodes de Monte Carlo afin d'analyser statistiquement un système physique complexe. Dans ce cas, les particules représentent l'échantillon sur lequel travaillent ces méthodes. Dans POOMA, ces entités sont représentées par des objets de la classe `Particles`. Cette classe permet de gérer une collection dynamique d'attributs caractérisant un ensemble de particules. Les `Particles` sont utilisables pour simuler numériquement des systèmes composés d'entités discrètes telles que ions ou molécules.

3.2.2.2 Algorithmes parallèles

Le modèle d'exécution parallèle choisi par POOMA est basé sur le parallélisme de données, dans lequel une même instruction de calcul est appliquée simultanément à chacun des éléments d'un ensemble de données. Les algorithmes manipulant les structures de données fournies par POOMA ressemblent donc à des algorithmes séquentiels. Dans le code suivant

```
Array<2> V(N, N);
Array<2> b(N, N);
Interval<1> I(1, N-2), J(1, N-2);

V = 0.0;
b = 0.0;
for (int it = 0; it < 200; ++it)
{
   V(I,J) = 0.25 * (V(I+1,J) + V(I-1,J) +
                    V(I,J+1) + V(I,J-1) - b(I,J));
}
```

l'opération à l'intérieur de la boucle s'exécute en séquentiel ou en parallèle pour tous les éléments appartenant aux intervalles sélectionnés sur les vecteurs `V` et `b`. Le parallélisme n'apparaît pas explicitement, mais en pratique il est présent à niveau du noyau exécutif de POOMA. Ce modèle d'exécution exploite la multiprogrammation légère sur multiprocesseurs à mémoire partagée : le calcul de chaque élément est effectué par un processus léger et les données sont stockées dans une mémoire commune accessible à tous les processus. Pour cela, POOMA s'appuie sur un noyau exécutif appelé SMARTS[156], qui gère les tâches parallèles (implémentées par des processus légers) et les dépendances de données entre elles.

Dans sa version la plus récente, POOMA étend son modèle d'exécution aux architectures à mémoire distribuée. Le stockage des données relatifs aux objets POOMA est flexible : le passage d'un format de stockage contigu à un format distribué n'exige que des légères modifications au moment de la création des objets par l'utilisateur. Dans cette version, la mise en œuvre des communications nécessaires à la gestion des structures de données distribuées utilise la bibliothèque d'échange de messages nommée CHEETAH.

3.2.2.3 Appréciation de l'outil

POOMA utilise des techniques de programmation C++ de pointe pour une mise en œuvre aisée d'applications de simulation numérique. C'est un outil puissant pour la résolution d'EDP par la méthode des différences finies, avec des abstractions qui pourraient être étendues à la méthode des éléments finis, sous réserve de pouvoir travailler avec des maillages non-structurés et de résoudre les problèmes matriciels par les algorithmes classiques. Les objets de type `Array` peuvent être utilisés pour représenter les matrices et vecteurs nécessaires à la résolution d'un système linéaire. Une résolution basée sur une méthode de décomposition de domaine peut être développée en utilisant les formats de stockage distribuée des objets POOMA. De même, on peut envisager une spécialisation des classes `Geometry` et `Field` adaptée à la méthode des éléments finis.

POOMA met en œuvre une approche haut niveau pour l'expression du parallélisme. Si ce paradigme permet aisément de passer du séquentiel au parallèle, il ne permet pas de contrôler la distribution des tâches et des données sur les processeurs. Le parallélisme étant caché, cet outil est peu flexible pour effectuer des expérimentations (nombre de sous-domaines, placement sur les processeurs). La portabilité est un autre point faible car les noyaux exécutifs SMARTS et CHEETAH ne sont pas disponibles sur une grande variété de plates-formes.

3.3 Présentation de AHPIK

L'environnement AHPIK est composé d'un ensemble de classes C++ fournissant les diverses abstractions impliquées dans le développement de solveurs de problèmes d'EDP basés sur une méthode de décomposition de domaine. Certaines de ces abstractions sont explicitement implantées dans AHPIK (par exemple maillages, éléments finis, solveurs parallèles), pour les autres nous avons développé des interfaces avec des bibliothèques externes (Metis, SparseLib++, MV++, IML++, Su-

perLU). Dans AHPIK, l'utilisation du paradigme objet (voir chapitre 5) permet à AHPIK d'offrir un support fortement modulaire pour le développement de solveurs parallèles en cachant à l'utilisateur les détails d'une mise en œuvre parallèle à base de processus légers.

La figure 3.4 illustre l'organisation modulaire de AHPIK. Les classes de l'environnement sont réparties dans trois couches établies sur des bibliothèques fournissant quelques "briques" de base. La couche 1 correspond au **noyau** de AHPIK (paragraphe 4.3) ; cette couche générique encapsule les données et les algorithmes qui peuvent être réutilisés par des différents solveurs de décomposition de domaine. Les schémas de communication et synchronisation spécifiques à chaque méthode sont définis dans la couche 2 (paragraphe 5.3) ; ils utilisent les fonctionnalités fournies par le noyau de AHPIK. Finalement, les calculs spécifiques à chaque solveur sont définis par l'utilisateur dans la couche 3. Dans les chapitres 4 et 5 nous décrivons ces trois couches en détail.

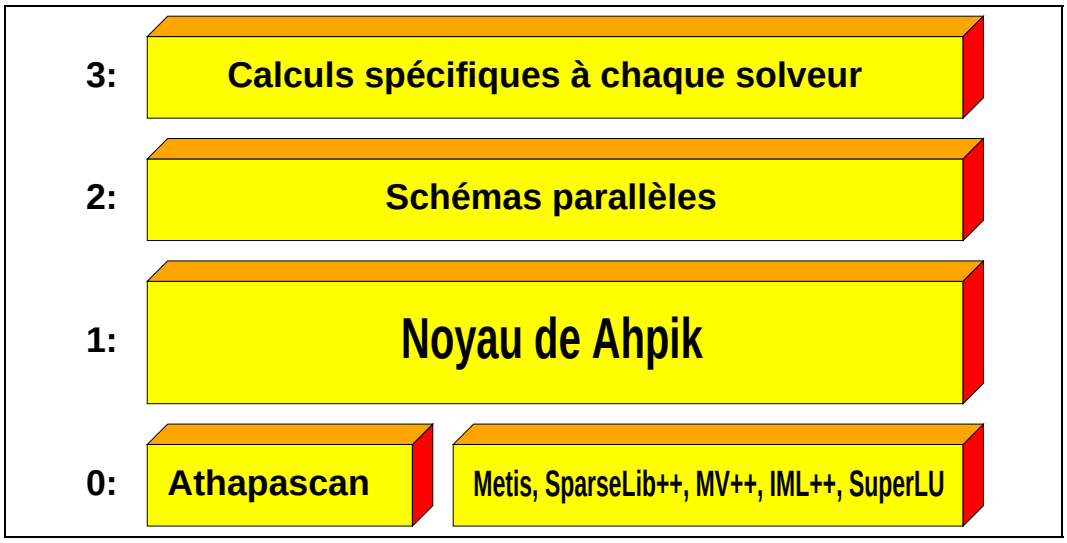

Figure 3.4 *Organisation modulaire de* AHPIK.

En quelques mots, le noyau de AHPIK est une couche logicielle générique gérant le parallélisme et les communications dans le cadre des applications reposant sur une méthode de décomposition de domaine. Ce noyau fournit une interface de programmation indépendante de la bibliothèque de communication utilisée (ATHAPASCAN-0, MPI – voir résultats dans le paragraphe 6.3.4), et permet d'exprimer des schémas parallèles servant à différentes méthodes de décomposition de domaine, qu'elles soient synchrones ou asynchrones. Ces schémas peuvent aussi servir dans le cadre de problèmes reposant sur une méthode avec découpe géométrique du domaine de résolution, par exemple lors des simulations en dynamique moléculaire.

4

Structure du parallélisme pour la décomposition de domaine

Sommaire

4.1 Construction du graphe de tâches 84
4.1.1 Identification des tâches : le cas général 85
4.1.2 Coûts des tâches 85
4.1.3 Interactions entre les tâches 86
4.1.4 Placement des tâches 87
4.1.5 Synchronisations 87
4.2 Extensions et cas spécifiques 88
4.2.1 Schémas asynchrones 88
4.2.2 Résolution globale par une méthode de Krylov 91
4.3 Mise en œuvre des processus légers dans le noyau de AHPIK 94
4.3.1 Motivations 94
4.3.2 Conception de l'interface 94
4.3.3 Écriture orientée objet 96
4.3.4 Mise en œuvre parallèle avec ATHAPASCAN-0 104
4.3.5 Mise en œuvre parallèle avec MPI 105

Les méthodes mathématiques de décomposition de domaine (chapitre 2, paragraphe 2.3) reposent sur une décomposition géométrique du domaine de calcul permettant de briser le problème d'EDP complet en un ensemble de problèmes d'EDP plus petits. La manière dont ces problèmes sont couplés définit la méthode de décomposition de domaine, une parallélisation naturelle en découle.

Dans ce cadre, le schéma parallèle (paragraphe 4.1) définit l'entrelacement entre les tâches de calculs locaux et les tâches de calcul d'interface. Ces différentes étapes de calcul sont liées algorithmiquement par des communications. La méthode itérative choisie pour résoudre les problèmes couplés détermine des points de synchronisation. Si deux méthodes ont les mêmes points de synchronisation elles ont le même schéma parallèle. Le paragraphe 4.2 discute des aspects concernant la synchronisation du schéma parallèle.

Nous avons choisi d'implémenter ces schémas parallèles à l'aide de processus légers pour relaxer autant que possible les contraintes de synchronisation. Des détails sur la conception du noyau de AHPIK sont exposés dans le paragraphe 4.3. Des exemples de schémas sont proposés dans le chapitre 5.

4.1 Construction du graphe de tâches

Pour toutes les méthodes de décomposition de domaine que nous avons étudiées, il est possible d'extraire une structure de calcul commune liée à la décomposition d'un domaine Ω en K sous-domaines Ω_k, $k \in \{1, ..., K\}$. Cette partition crée des interfaces, nous utilisons la notation γ_{kl} pour désigner génériquement l'interface entre deux sous-domaines Ω_k et Ω_l. Dans le cas de méthodes sans recouvrement, l'interface représente géométriquement la partie de frontière commune aux deux sous-domaines. Ces méthodes permettent alors de condenser le problème d'EDP en un problème posé sur les interfaces. Dans les méthodes avec recouvrement, une "interface" représente géométriquement le recouvrement (intersection) entre deux sous-domaines donnés, les valeurs aux frontières du recouvrement, internes à un des sous-domaines, sont utilisées algorithmiquement comme des conditions aux limites pour l'autre sous-domaine, et *vice-versa*.

Après la définition des différentes tâches (paragraphe 4.1.1) et de l'évaluation de leur coût (paragraphe 4.1.2), nous déterminons les dépendances de données entre ces tâches pour construire le graphe de précédence (paragraphe 4.1.3). Ce graphe est commun aux différentes méthodes. Dans un second temps nous discutons le placement des tâches (paragraphe 4.1.4) et la synchronisation des algorithmes (paragraphe 4.1.5).

4.1.1 Identification des tâches : le cas général

Pour chaque domaine de la décomposition le calcul d'une itération de la méthode comprend deux types de tâches :

- les tâches T_Ω^k qui effectuent les calculs locaux associés au sous-domaine Ω_k. Les tâches de calculs locaux nécessitent les éléments (matrices, vecteurs, préconditionneurs) relatifs à un sous-domaine Ω_k. En général, les calculs locaux s'articulent autour de la résolution d'un système linéaire. Ces tâches produisent des données servant au calcul ou à la mise à jour des valeurs aux interfaces entre Ω_k et ses sous-domaines voisins Ω_l ;
- les tâches T_γ^{kl}, responsables du calcul des variables associées à l'interface γ_{kl} et des communications des valeurs de celle-ci. On remarque que T_γ^{kl} et T_γ^{lk} sont en général identiques (sauf pour la méthode des joints), que le calcul sur l'interface peut éventuellement être vide (méthode de Schwarz), et que les transferts de données entre ces tâches vont dépendre très fortement de la méthode de résolution.

Remarque : Cette définition de tâches peut être raffinée car les solveurs locaux peuvent être eux-mêmes parallèles : il s'agit alors d'un parallélisme à grain plus fin. Le coût de gestion des processus légers étant faible, on peut les utiliser pour la mise en œuvre des solveurs locaux (voir par exemple [58]). L'approche que nous avons retenue n'exploite pas cette possibilité car nous utilisons des solveurs locaux séquentiels pour matrices creuses offerts par des bibliothèques spécialisées.

4.1.2 Coûts des tâches

Le coût de chaque tâche T_Ω^k peut être estimé à partir du nombre d'inconnues associées au problème local posé dans le sous-domaine Ω_k. Il dépend de la méthode (itérative ou directe) utilisée. Lorsque le solveur local utilise une méthode directe, on peut évaluer de manière sûre le coût des calculs. Lorsque l'on utilise un solveur itératif, on connaît sa complexité en fonction de certaines propriétés de la matrice. Ces propriétés dépendent du problème à résoudre : par exemple, le coût est en $O(\sqrt{N})$ pour la méthode du gradient conjugué appliquée au problème de Poisson. En pratique, cela est une indication approximative du nombre d'itérations, ce qui peut mener à des différences de coût entre les résolutions locales. D'un point de vue parallélisme cela induit un déséquilibre de charge.

De même, le coût des tâches des calculs relatif à l'interface γ_{kl} est fonction du nombre de nœuds situés sur celle-ci. En général, le coût des calculs d'interface est

bien plus petit que le coût des tâches de calculs locaux. On remarque que si les sous-domaines sont trop petits, le coût des calculs locaux est dominé par le coût des transferts, i.e. le coût de gestion du parallélisme (les communications) est supérieur au coût des calculs utiles (calculs locaux et d'interface). Dans ces conditions, la parallélisation ne peut pas être efficace.

4.1.3 Interactions entre les tâches

Afin d'exprimer les dépendances entre les tâches nous utilisons la notation $T_1 \ll T_2, T_3$ qui signifie que la tâche T_1 dépend des tâches T_2 et T_3. Les dépendances s'expriment d'une itération à l'autre de la méthode de décomposition de domaine. On note $T(i)$ l'exécution d'une tâche à l'itération i et ν_k l'ensemble d'indices de sous-domaines ayant une interface commune avec le sous-domaine Ω_k. Avec ces notations, le graphe de précédence typique pour une itération de décomposition de domaine se présente comme suit :

- les calculs locaux correspondant au sous-domaine Ω_k à l'itération i nécessitent les résultats des calculs d'interface obtenus à l'itération précédente. On a donc les précédences suivantes :
$$T_\Omega^k(i) \ll T_\gamma^{kl}(i-1) \quad \forall l \in \nu_k$$
- le calcul correspondant à une interface γ_{kl} à l'itération i a besoin des résultats des calculs locaux sur le sous-domaine Ω_k ainsi que sur le sous-domaine voisin Ω_l. Les précédences s'écrivent donc :
$$T_\gamma^{kl}(i) \ll T_\Omega^k(i), T_\Omega^l(i) \forall l \in \nu_k$$

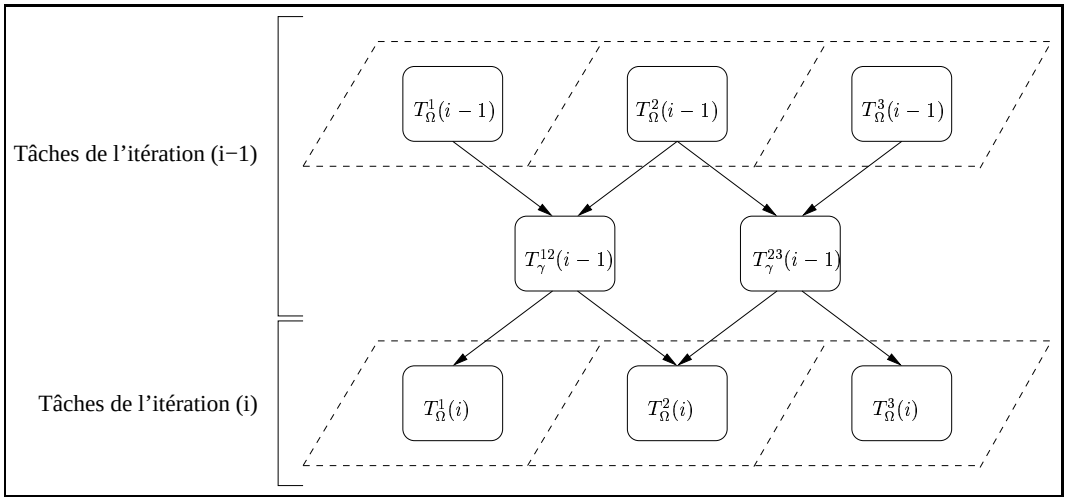

Figure 4.1 *Graphe de tâches pour une décomposition en trois sous-domaines.*

La figure 4.1 illustre les précédences entre tâches pour une décomposition d'un domaine rectangulaire en trois sous-domaines. Ce graphe de précédence est caractéristique des méthodes synchrones.

Les échanges de données sont en général symétriques car chaque processeur fait un envoi de ses contributions locales suivi d'une réception de données extérieures. Des méthodes telles que la méthode des joints ont une conception mathématique asymétrique au niveau des calculs d'interface, les communications peuvent alors être soit symétriques soit asymétriques.

4.1.4 Placement des tâches

Généralement, la décomposition en sous-domaines prend en compte le problème mathématique, des contraintes géométriques et des contraintes liées à la convergence numérique de la méthode de décomposition de domaine. Il est naturel de considérer une décomposition permettant d'affecter au moins un sous-domaine par nœud. Dans ce cas les tâches T_Ω^k et T_γ^{kl} sont placées sur le même processeur.

Bien que les tâches T_γ^{kl} et T_γ^{lk} soient en général identiques, ce qui implique un calcul redondant à l'interface, on les duplique de manière à réduire le nombre de communications et le nombre de points de synchronisation. Le gain est double car le coût de ce calcul est souvent inférieur au coût d'une communication.

Afin d'obtenir un recouvrement des communications par des calculs et un meilleur équilibrage de charge, on place plusieurs sous-domaines par processeur. Dans ce cas on place des sous-domaines voisins sur le même processeur de manière à réduire le volume de communications entre processeurs. Il est aussi possible de prendre en considération la topologie du réseau pour définir un bon placement des tâches, mais cela conduit à des solutions peu portables[116].

4.1.5 Synchronisations

L'algorithmique classique des méthodes de décomposition de domaine est synchrone : une tâche T_Ω (calculs locaux) nécessite des résultats des tâches T_γ (calculs d'interface) obtenus à l'itération précédente. Les calculs d'interface impliquent des communications entre processeurs. Ils représentent des points de synchronisation pour chacun des processeurs qui doit attendre une donnée provenant d'un autre processeur. L'efficacité parallèle d'une méthode de décomposition de domaine dépend ainsi d'un bon équilibrage de charge entre les processeurs. En effet, si le travail n'est

pas distribué de façon équitable, certains processeurs finissent leurs calculs avant les autres. Les temps d'attente ainsi créés dégradent les performances de l'algorithme. Ce comportement est illustré dans la figure 4.2 pour une décomposition en deux sous-domaines.

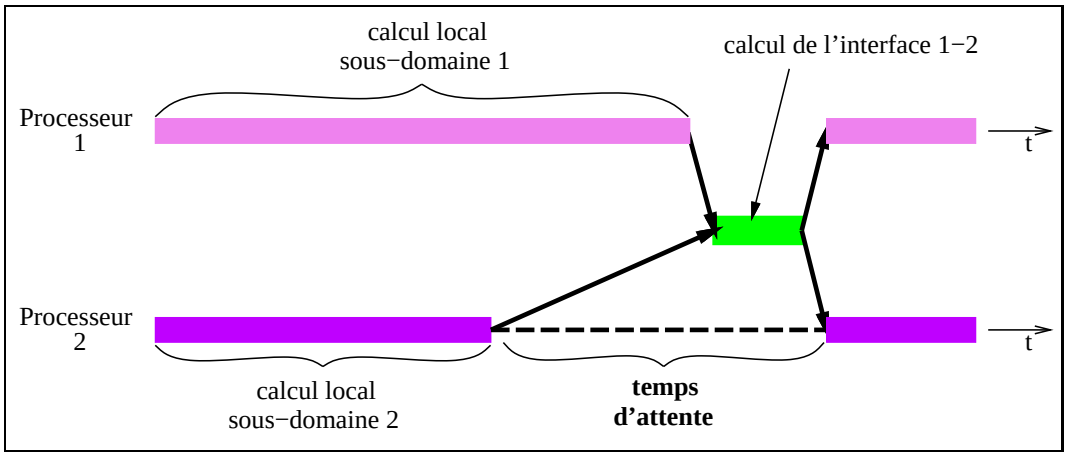

Figure 4.2 *Diagramme de Gantt pour une décomposition en deux sous-domaines.*

Les méthodes asynchrones permettent d'éviter les temps d'attente en éliminant autant que possible les points de synchronisation. Elles se prêtent aux problèmes composés des tâches ayant des coûts de calcul très différents. Le prix que l'on paie pour cette flexibilité au niveau des communications est une perte en vitesse de convergence doublée d'une certaine "redondance" de calcul. Ces méthodes sont avantageuses lorsque la charge de calcul n'est pas bien équilibrée ou lorsque les communications entre processeurs sont lentes[75].

Même si l'asynchronisme permet de contourner le problème de l'équilibrage de charge entre les processeurs, les pertes éventuelles sur la vitesse de convergence de la méthode numérique itérative peuvent révéler qu'il est préférable de choisir un schéma synchrone classique.

4.2 Extensions et cas spécifiques

4.2.1 Schémas asynchrones

Dans le cadre du calcul parallèle, les algorithmes asynchrones sont utilisés pour modéliser les calculs itératifs s'effectuant sur plusieurs processeurs coopérant. Ces

algorithmes tolèrent les éventuels délais de calcul et de communication entre les différents processeurs : ils autorisent quelques processeurs à calculer plus vite et effectuer plus d'itérations que d'autres, ainsi qu'à échanger des données plus fréquemment que d'autres. Les algorithmes asynchrones ont initialement été étudiés dans le cadre des systèmes linéaires, la première référence connue remontant à l'article de Chazan-Miranker publié en 1969[42]. Depuis ce travail pionnier, plusieurs auteurs se sont intéressés à la théorie et à l'application des algorithmes asynchrones[117, 135, 12, 19, 7, 118].

Dans ce paragraphe nous fournissons une introduction à cette famille d'algorithmes, notre but étant de donner au lecteur non-spécialiste les éléments essentiels caractérisant les modèles d'algorithmes asynchrones étudiés au fil des années. Notre introduction est basée sur [75]. D'autres survols sur les itérations asynchrones se trouvent par exemple dans [3, 21, 20].

On se donne un espace $E = E_1 \times ... \times E_m$ et une application $H : E \to E$. Les composants de H sont notés H_i, c'est-à-dire, on a

$$H : E \to E, \quad x = (x_1, ..., x_m) \to (H_1(x), ..., H_m(x)),$$

où $x_i, H_i(x) \in E_i, i = 1, ..., m$. On veut trouver un point fixe de H. Pour cela, le schéma standard consiste à approcher ce point fixe en calculant

$$x^{k+1} = H(x^k), \quad k = 0, 1, ... \tag{4.1}$$

Une version parallèle de la formulation (4.1) peut se construire comme suit. Étant donnée une machine parallèle comportant p processeurs $(p \leq m)$, on associe un bloc de composants $J_j \subseteq \{1, ..., m\}$ à chaque processeur $P_j (j = 1, ..., p)$. Le calcul sur le processeur P_j s'effectue ensuite selon l'algorithme 3.

Algorithme 3 : Algorithme parallèle pour l'approximation d'un point fixe.

1: **pour** $k = 1$ **jusqu'à** convergence **faire**
2: obtenir x
3: **pour** $i \in J_j$ **faire**
4: $x_i^{new} = H_i(x)$
5: fournir x_i^{new} afin de remplacer les composants x_i de x
6: **fin pour**
7: **fin pour**

Il convient remarquer que l'algorithme 3 est valable quel que soit le modèle de mémoire (partagée ou distribuée) de la machine parallèle. Lorsqu'il s'agit d'une machine à mémoire partagée, les étapes 2 et 5 de l'algorithme se traduisent respectivement par une lecture et une écriture dans la mémoire commune. Lorsque l'on

travaille sur une machine à mémoire distribuée, ces étapes se font par échange de messages entre les processeurs concernés.

Dans une mise en œuvre parallèle synchrone de (4.1), chaque processeur attend que l'itération k soit finie sur les autres processeurs avant de passer à l'itération suivante. Dans un schéma asynchrone, les processeurs n'attendent pas les autres. Comme les temps d'exécution de la boucle sur i peuvent différer d'un processeur à l'autre, il apparaît des déphasages : le processus itératif est moins structuré. Après convergence, les différents processeurs auront effectué plus ou moins d'itérations. On remarque qu'il n'y a pas de temps d'inactivité, puisque les processeurs n'attendent jamais les autres.

L'analyse mathématique de l'algorithme 3 donne des éléments théoriques fondamentaux pour prouver la convergence d'un calcul itératif asynchrone. Soit k un compteur d'itérations ($k \in \mathbb{N}$) incrémenté de 1 à chaque fois qu'un processeur P_j obtient la valeur de x. À cet instant, la valeur de chaque composant de x provient d'un calcul effectué lors d'une itération précédente, c'est-à-dire, nous avons $x = (x_1^{s_1(k)}, ..., x_m^{s_m(k)})$, où $s_l(k) \in \mathbb{N}_0, l = 1, ..., m$ sont des indices d'itérations antérieures à k indiquant l'itération au cours de laquelle le composant x_l a été calculé. On définit aussi un ensemble I^k de composants calculés à l'itération k. Avec ces définitions, et étant donné $x_0 \in E = E_1 \times ... \times E_m$, une itération asynchrone s'appliquant à l'algorithme 3 peut-être modélisée comme suit[73, 152] :

$$x_i^k = \begin{cases} H_i(x_1^{s_1(k)}, ..., x_m^{s_m(k)}) & \forall i \in I^k \\ x_i^{k-1} & \forall i \notin I^k. \end{cases} \tag{4.2}$$

La formulation 4.2 doit satisfaire certaines hypothèses :

– seuls les composants calculés antérieurement sont utilisés à l'itération courante, i.e.,
$$s_i(k) \leq k - 1 \quad \forall i \in \{1, ..., m\};$$

– au fur et à mesure où le calcul itératif avance, des valeurs plus récentes peuvent être obtenues pour chacun des composants, i.e.,
$$\lim_{k \to \infty} s_i(k) = \infty \quad \forall i \in \{1, ..., m\};$$

– aucun composant ne manque d'être mis à jour au cours du processus itératif, i.e.,
$$|\{k \in \mathbb{N} : i \in I^k\}| = \infty \quad \forall i \in \{1, ..., m\}.$$

Un théorème général de convergence pour le modèle d'itération asynchrone (4.2) se trouve dans [19, 155]. Bien que ce modèle fondamental ait de nombreuses

applications, il admet plusieurs extensions afin de tenir compte de situations soit plus spécifiques soit plus générales. Par exemple, certains auteurs imposent des conditions sur les délais $d_i(k) = k - s_i(k)$ (voir par exemple [6, 111, 129]). De même, certains auteurs se sont intéressés aux cas où l'application H a des composants H_i qui sont eux-mêmes approchés, par exemple lorsque l'on travaille avec des méthodes itératives à deux niveaux (« two-stage iterations »). Ce cadre d'application est traité par les modèles d'itérations asynchrones avec communications flexibles[61, 118], qui autorisent l'envoi de nouvelles valeurs de x_i avant même que le calcul itératif plus interne ait convergé.

La mise en œuvre des itérations asynchrones est en général plus difficile par rapport aux méthodes itératives synchrones classiques. Un des problèmes à traiter à chaque implémentation est la terminaison des algorithmes. La convergence d'un algorithme itératif est basée sur une information globale sur l'avancement de l'exécution, cependant dans une exécution asynchrone chaque processeur a sa propre vision du processus itératif. Bien que de nombreux résultats théoriques montrent les avantages des itérations asynchrones, l'expérimentation autour de ces méthodes reste encore faible. Selon [102], les démarches expérimentales sont un facteur décisif pour justifier l'utilisation des méthodes asynchrones dans le cadre du calcul haute-performance. Parmi les travaux fournissant des résultats expérimentaux sur les méthodes asynchrones on peut citer [28, 74, 87].

4.2.2 Résolution globale par une méthode de Krylov

Les méthodes de décomposition de domaine peuvent intervenir dans la résolution de grands problèmes matriciels de deux manières.

Dans la première, elles servent à construire un problème d'interface. Dans ce cas on résout par exemple les problèmes locaux à l'aide d'une méthode directe et le problème d'interface à l'aide d'une méthode de point fixe ou d'une méthode de Krylov. Dans la seconde, elles peuvent être utilisées comme préconditionneurs pour une méthode de Krylov. Les deux cas se ressemblent du point de vue de l'exécution parallèle, nous étudions la première alternative en utilisant le formalisme introduit dans le paragraphe 2.3.3 pour les méthodes sans recouvrement. Génériquement, ces méthodes cherchent à résoudre, dans le cas de 2 sous-domaines,

$$\begin{pmatrix} C_{11} & 0 & C_{13} \\ 0 & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ X_3 \end{pmatrix} = \begin{pmatrix} D_1 \\ D_2 \\ D_3 \end{pmatrix} \tag{4.3}$$

avec le système condensé à l'interface

$$SX_3 = G$$

où

$$S = C_{33} - \sum_{k=1}^{2} C_{3k} C_{kk}^{-1} C_{k3}$$

et

$$G = C_{33} - \sum_{k=1}^{2} C_{3k} C_{kk}^{-1} D_k$$

L'algorithme 4 présente la résolution de ce système par la méthode du gradient conjugué. Le lecteur trouvera une présentation plus détaillée dans l'annexe D. Toutes les instructions composant cet algorithme utilisent des matrices et vecteurs locaux.

Algorithme 4 : Algorithme CG générique pour le sous-domaine $\Omega_k, k = 1, 2$.

1: $X_3^0 = 0$ {initialisation}
2: X_k solution de $C_{kk} X_k = D_k - C_{k3} X_3^0$
3: $\xi_k = C_{3k} X_k$
4: $\boxed{p^0 = r^0 = D_3 - C_{33} X_3^0 - \xi_1 - \xi_2}$ {communications}
5: $\boxed{\epsilon^0 = \|r^0\|^2}$ {communications globales}

6: **pour** $i = 0, 1, ...$ **jusqu'à** la convergence **faire**
7: z_k solution de $C_{kk} z_k = C_{k3} p^i$
8: $\xi_k = C_{3k} z_k$
9: $\xi_3 = C_{33} p^i$
10: $\boxed{\psi = \xi_3 - \xi_1 - \xi_2}$ {communications}
11: $\boxed{\alpha^i = \|r^i\|^2 / (\psi, p^i)}$ {communications globales}
12: $X_3^{i+1} = X_3^i + \alpha^i p^i$
13: $r^{i+1} = r^i - \alpha^i \psi$
14: $\boxed{\beta^{i+1} = \|r^{i+1}\| / \|r^i\|^2}$ {communications globales}
15: $p^{i+1} = r^{i+1} - \beta^{i+1} p^i$
16: **fin pour**

Nous avons caché l'aspect parallèle de l'algorithme au sein des opérations algébriques. Les instructions (4,5,10,11 et 14), que nous avons entourées, nécessitent des communications. Les autres instructions s'effectuent de manière locale. Les instructions 2 et 7 représentent les calculs locaux sur les sous-domaines. Ces calculs

requièrent l'utilisation d'un solveur local. Les calculs d'interface sont présents dans les instructions 3–4 et 8–10, ces produits matrice-vecteur nécessitent des communications.

La vérification de la convergence (point de synchronisation globale) est effectuée au niveau de l'instruction 14, qui nécessite des calculs de norme (instructions 5 et 14). Ces calculs imposent des points de synchronisation globale additionnels. Le calcul du pas de descente (instruction 11) nécessite le calcul d'un produit scalaire global.

Une fois que la solution X_3 condensée à l'interface est obtenue, on résout à nouveau les systèmes locaux

$$C_{kk}X_k = D_k - C_{k3}X_3$$

afin de calculer les solutions locales X_k. On peut également les obtenir au cours de la résolution itérative. Dans ce cas on utilise la solution calculée à l'instruction 2 comme solution initiale X_k^0 et on effectue le calcul additionnel

$$X_k^{i+1} = X_k + \alpha^i z_k$$

après le calcul du pas de descente α (instruction 11).

L'algorithme se généralise aisément pour $K \neq 2$. Le problème est alors résolu en considérant une interface unique (essentiellement pour la méthode de Schur primal) ou plusieurs interfaces (Schur dual, ou Schur primal avec le domaine décomposé en "tranches"). Dans le premier cas les communications nécessaires pour les étapes 4 et 10 peuvent être respectivement groupées avec les communications globales des étapes 5 et 11. On réduit ainsi les points de synchronisation de l'algorithme. Dans le deuxième cas, on peut utiliser la liste des sous-domaines voisins (communications point-à-point) de Ω_k pour effectuer les étapes 4 et 10. Le choix du mode de communication va en fait dépendre du schéma parallèle effectif.

Dans la méthode du gradient conjugué, les normes calculées sont utilisées pour mettre à jour la direction de descente nécessaire à l'itération suivante. Il est périlleux, la preuve théorique n'existe pas à notre connaissance, de calculer β^{i+1} de manière asynchrone sous peine d'affecter la convergence de la méthode de gradient. Cette limitation n'existe pas dans les méthodes de point fixe, il est donc possible d'effectuer le calcul de norme de manière asynchrone sur un processus "coordinateur".

4.3 Mise en œuvre des processus légers dans le noyau de AHPIK

4.3.1 Motivations

Le développement de ce noyau a été guidé par les préoccupations suivantes :

- réutiliser la même couche logicielle de base pour la mise en œuvre de différentes méthodes ;
- changer facilement le comportement en terme de synchronisation d'une méthode de résolution. En général, il est difficile de prévoir l'impact de l'utilisation d'un schéma synchrone ou asynchrone dans les performances parallèles d'une méthode de résolution. C'est plus spécialement le cas lorsque les calculs sont déséquilibrés. L'expérimentation avec différents schémas de synchronisation est alors très utile pour la mise au point d'une application de simulation numérique. Nous verrons plus tard (paragraphe 5.3) comment ce point est traité dans AHPIK ;
- exprimer une méthode de résolution indépendamment de la bibliothèque de communication. D'un point de vue programmation parallèle, il est instructif de comparer une mise en œuvre basée sur ATHAPASCAN-0 avec une mise en œuvre basée sur MPI. Cela permet de mettre en évidence les applications pour lesquelles l'utilisation du paradigme de processus légers communicants offre des avantages.

4.3.2 Conception de l'interface

Le noyau de AHPIK est responsable de la gestion d'un réseau de processus légers communicants. À partir d'une topologie de décomposition en sous-domaines et d'un placement des sous-domaines sur les processeurs, le noyau crée un « pool » de processus légers sur chaque processeur. À chaque sous-domaine correspond au moins un processus léger chargé de coordonner l'enchaînement des calculs sur ce sous-domaine. Le noyau s'occupe aussi des liaisons entre ces processus légers. Il initialise un support pour les communications parallèles, celui-ci est caractérisé par l'établissement de **liens communicants** entre sous-domaines voisins. Ce support autorise la modification du placement des sous-domaines au cours de l'exécution : un protocole de reconfiguration du réseau de processus légers peut être mis en place. Pour simplifier l'exposé, nous supposons pour l'instant que chaque processeur gère un seul sous-domaine, les liens communicants relient donc des processeurs voisins.

À chaque sous-domaine correspond un groupe de liens communicants, en nombre généralement égal au nombre d'interfaces du sous-domaine. Ces liens sont créés au lancement de AHPIK à partir du placement géométrique et algorithmique des sous-domaines. Ils établissent les contacts entre processeurs gérant des sous-domaines géométriquement voisins. Génériquement ce lien est appelé à effectuer des opérations de communication d'objets (écriture, lecture, échange) contenant des données nécessaires à la méthode de résolution. Par souci de réutilisabilité, la programmation de ce lien est indépendante du caractère synchrone ou asynchrone de la méthode de résolution.

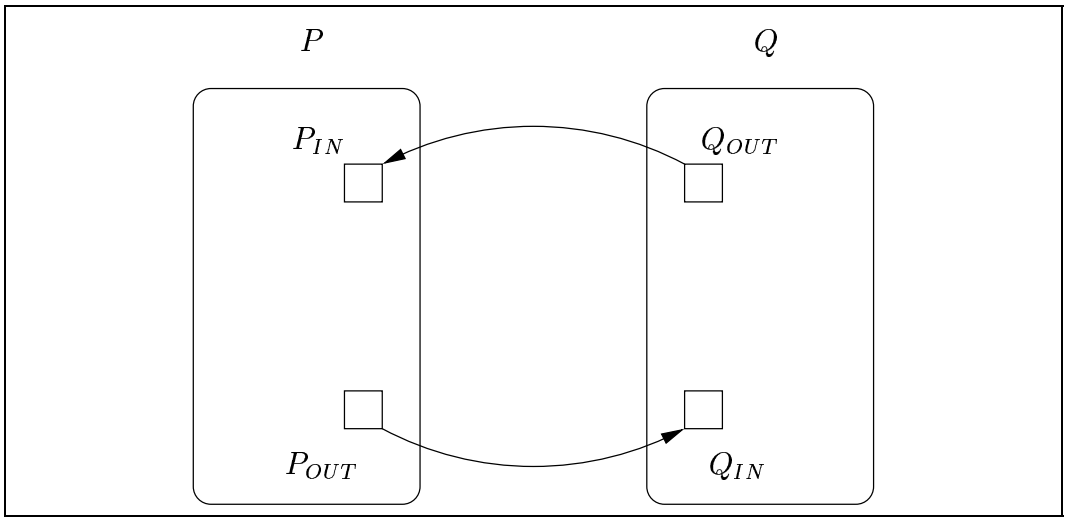

Figure 4.3 *Liens communicants entre deux processeurs voisins.*

De manière plus précise, la création d'un lien implique la création d'au maximum deux processus de communication pour chacun des deux sous-domaines concernés. Si l'on note P et Q les processus légers principaux traitant ces sous-domaines, on crée deux processus d'envoi de données P_{OUT} et Q_{OUT} chargés d'envoyer le contenu d'un espace mémoire lorsque nécessaire. À ces processus d'envoi correspondent deux processus de réception, respectivement Q_{IN} et P_{IN}, dont les fonctions dépendent du caractère de la communication (voir figure 4.3). Étudions par exemple deux comportements possibles du processus P_{IN} :

– Lorsque la communication est synchrone, le processus P_{IN} attend que les données soient arrivées. Une fois que les données sont reçues, il attend que le processus principal l'interroge afin de récupérer les données. Il ne peut pas recevoir de nouvelles données tant que le processus P ne l'a pas interrogé sur les précédentes. Le processus P est bloqué dans son exécution si P_{IN} ne peut lui fournir de données au moment de l'interrogation.
– Lorsque la communication est asynchrone, le processus P_{IN} est toujours en

attente de données, il contient les dernières arrivées. L'arrivée de nouvelles données efface les précédentes. Lorsque le processus principal P a besoin de données provenant du sous-domaine voisin, il interroge le processus P_{IN}. Toute interrogation est immédiatement satisfaite par P_{IN} : le processus P n'est jamais bloqué en attente de données.

Le caractère synchrone ou asynchrone de la résolution se manifeste par la selection d'une routine, synchrone ou asynchrone, implémentant les processus P_{IN} et P_{OUT}. Lors d'une mise en œuvre sans processus légers, ces routines contiennent simplement des appels à des primitives de communication telles que celles proposées par MPI. Le processus P fonctionne toujours de façon identique, quel que soit le comportement choisi pour les processus P_{IN} et P_{OUT}.

Dans le cadre des méthodes de décomposition de domaine, les liens communicants sont principalement utilisés pour les communications des données d'interface. Cependant, ils peuvent être utilisés pour la communication de données lors de l'initialisation de l'application (par exemple, pour mettre en œuvre un algorithme distribué de coloriage des sous-domaines pour la méthode de Schwarz multiplicative).

Pour chaque sous-domaine, l'ensemble de liens communicants définit un **voisinage communicant**. Ce concept permet de démarrer des communications pour tout le voisinage en même temps. En présence de calculs découplés dans un même sous-domaine (voir section 6.3.2), on peut utiliser plusieurs voisinages communicants pour un même sous-domaine.

4.3.3 Écriture orientée objet

Au début de l'exécution parallèle, le noyau de AHPIK crée un processus léger principal par sous-domaine en fonction du schéma de placement sur les processeurs. Il s'agit par exemple de la création des processus P et Q introduits dans la section précédente. En général, ces processus légers principaux s'occupent de l'enchaînement des tâches de calculs locaux et des tâches de calculs d'interface associés à un sous-domaine. Ils sont la base des schémas de calcul caractérisant une ou plusieurs méthodes de décomposition de domaine. Des schémas pour les méthodes que nous avons traitées dans cette thèse sont déjà disponibles dans AHPIK. Cette section explique les grandes lignes des outils que AHPIK fournit pour leur mise en œuvre.

4.3.3.1 Affectation de tâches aux processus légers principaux

La définition de ces tâches obéit à certaines règles. De manière générale, une tâche correspond à une classe-fonction C++ (« function class » [62]), c'est-à-dire, une classe pouvant être utilisée comme une fonction. Il s'agit d'une classe surchargeant l'operateur '()', dont la forme la plus simple s'écrit comme suit :

```
class user_task
{
  public :
    void operator() ( [...paramètres...] ) {
        [... corps de la fonction ... ]
    }
};
```

En C++, un appel séquentiel à cette classe-fonction s'écrit :

```
user_task()( [...paramètres effectifs...] );
```

Dans AHPIK, cette classe-fonction doit être dérivée de la classe `DomainTask` prédéfinie par le noyau. Ainsi chaque tâche hérite, du noyau de AHPIK, de certains attributs la distinguant des autres tâches (numéro du sous-domaine auquel elle correspond, par exemple). Cela s'écrit comme suit :

```
class user_task : public DomainTask
{
  public :
    void operator() ( [...paramètres...] ) {
        [... corps de la fonction ... ]
    }
 };
```

En général, une classe dérivée de `DomainTask` spécifie, dans la fonction surchargeant l'opérateur '()', le déroulement des calculs relatifs à un sous-domaine. Cela comprend par exemple la lecture du maillage, l'assemblage du système d'équations algébriques, l'initialisation de la méthode de décomposition de domaine et

l'appel au schéma parallèle correspondant à la résolution globale itérative. Nous reparlerons plus tard (paragraphe 5.3) des schémas parallèles accessibles aux classes dérivées de `DomainTask`.

Avec ces définitions, un programme très simple utilisant directement les fonctionnalités offertes par le noyau de AHPIK a la forme :

```
#include <ahpik.h>

void
main(int argc, char** argv)
{
  ahpik::Init(argc, argv, "map", "dom");
  ahpik::SpawnDomainTasks(user_task());
  ahpik::Terminate();
}
```

La première fonction AHPIK appelée doit être ahpik::Init(). Cette fonction initialise la bibliothèque de communication sur laquelle repose AHPIK, puis s'occupe de la création du réseau de processus légers correspondant à la décomposition géométrique donnée par le fichier "dom", avec le placement défini dans le fichier "map". Les liens communicants sont établis par cette fonction. Le nombre de processus légers créés sur chaque processeur pendant l'initialisation de AHPIK est égal au nombre de sous-domaines placés sur ce processeur. Ces processus légers restent en attente d'une tâche à exécuter.

La fonction ahpik::SpawnDomainTasks() affecte des tâches aux processus légers crées sur un processeur P lors de l'appel à ahpik::Init(). Dans ce cas il s'agit de tâches de type user_task. Ces tâches ne reçoivent pas de paramètre. Elles s'exécutent en parallèle lorsqu'il y a plus d'un sous-domaine affecté au processeur P. Cette fonction retourne lorsque toutes les tâches ont fini leur exécution : elle bloque donc le fil d'exécution principal. Cette fonction est équivalente au code suivant :

```
for (i = 0; i < ahpik::DomainCount(); i++)
{
    ahpik::DomainThread(i).NewTask(user_task());
}
for (i = 0; i < ahpik::DomainCount(); i++)
{
    ahpik::DomainThread(i).WaitDone();
}
```

La fonction ahpik : :DomainCount() retourne le nombre de sous-domaines affectés au processeur P. Cette information est obtenue à partir du placement décrit dans le fichier spécifié au moment de l'initialisation. L'objet ahpik : :DomainThread(i) désigne le processus léger correspondant au $i^{ème}$ sous-domaine affecté au processeur P. La fonction NewTask() affecte une tâche à ce processus léger. Il s'agit d'une tâche asynchrone : son exécution se déroule en parallèle avec le fil d'exécution principal. La fonction WaitDone() permet ensuite d'attendre que le processus léger sélectionné ait fini l'exécution de la tâche qui lui a été affectée. Cette écriture est plus flexible que l'écriture utilisant ahpik : :SpawnDomainTasks() car elle permet de gérer le lancement et la synchronisation à la fin d'une tâche de manière découplée, ainsi que de contrôler les paramètres passés aux tâches. On suppose que de nouvelles tâches ne sont générées que lorsque les tâches lancées ont été achevées. Ainsi tout appel à NewTask() avant que le processus léger ait fini sa tâche bloque le fil d'exécution principal.

4.3.3.2 Affectation de données aux liens communicants

Nous allons maintenant nous concentrer sur les opérations accessibles à un objet d'une classe dérivée de DomainTask. Rappelons qu'un objet de cette classe représente une tâche travaillant dans le contexte d'un seul sous-domaine. Cette tâche ne connaît que les données correspondant à ce sous-domaine. Elle est reliée aux sous-domaines voisins par l'intermédiaire du support exécutif encapsulé dans la classe DomainTask. Ce support exécutif permet de gérer les liens communicants et le voisinage communicant correspondant à ce sous-domaine, ainsi que de déclencher des communications globales (communications collectives nécessaires au contrôle de convergence de la méthode itérative globale). L'association d'un contexte à un objet dérivé de DomainTask a lieu au moment du lancement de la tâche, c'est-à-dire, au moment où l'on l'affecte au processus léger choisi (par l'intermédiaire de la fonction NewTask()).

Un comportement typique d'une classe dérivée de DomainTask consiste à associer des données aux liens communicants, puis à effectuer des itérations alternant des calculs locaux et des calculs d'interface entrelacés par des échanges de données à travers les liens communicants. Pour l'association de données aux liens communicants on définit, pour chaque interface, les données qui seront échangées au cours d'une phase du calcul. Cela est illustré par l'extrait de code suivant :

```
class user_task : public DomainTask
{
  public :
    void operator()()
    {
        // ...
        domain : :SelectCommMode(SyncCommMode) ;
        for (i = 0; i < domain : :IfaceCount() ; i++)
        {
            domain : :InterfaceDataOut(i, x[i]) ;
            domain : :InterfaceDataIn(i, y[i]) ;
        }
        // ...
    }
};
```

Avant l'association de données aux liens communicants on doit définir le mode de communication (synchrone ou asynchrone) pour l'échange de données à travers les liens communicants. La fonction domain : :SelectCommMode() est utilisée pour cela. La fonction domain : :IfaceCount() retourne le nombre d'interfaces du sous-domaine. La donnée qui sera envoyée au $i^{ème}$ sous-domaine voisin est spécifiée par la fonction domain : :InterfaceDataOut(), la donnée qui sera reçue est spécifiée par domain : :InterfaceDataIn(). Les données à échanger sont en général des vecteurs. Leur type n'a pas besoin d'être spécifié : le polymorphisme inhérent au paradigme de programmation par objets permet d'utiliser le même nom de fonction pour des différents types de données prédéfinies.

Les fonctions domain : :InterfaceDataOut() et domain : :InterfaceDataIn() créent les processus légers de communication. Sur un processeur P, il s'agit des processus P_{IN} et P_{OUT} mentionnés dans le paragraphe précédent. Le comportement de ces processus dépend du choix préalable d'un mode de communication synchrone ou asynchrone.

Les opérations à travers les liens communicants sont soit des écritures, soit des lectures, soit des échanges (lectures puis écritures) de données. Ces opérations peuvent être déclenchées soit pour chaque interface séparément (lignes 7 à 11 de l'extrait de code ci-dessous), soit pour toutes les interfaces du sous-domaine au même temps (ligne 12 de cet exemple).

```
1   class user_task : public DomainTask
2   {
3     public :
4       void operator()()
5       {
6           // ...
7           for (i = 0; i < domain : :IfaceCount() ; i++)
8           {
9               domain : :Write(i) ;
10              domain : :Read(i) ;
11          }
12          domain : :ExchangeAll() ;
13          // ...
14      }
15  };
```

Les liens communicants ne servent qu'à des communications point-à-point entre deux processus légers traitant des sous-domaines voisins. Lorsque la méthode de résolution implique des échanges de données entre plus de deux processeurs (par exemple lorsqu'il existe une partie d'interface partagée par plus de deux sous-domaines), il faut traiter chaque envoi et chaque réception séparément, puis traiter les éventuelles redondances de données reçues. Cela devient contraignant pour les méthodes avec recouvrement appliquées à des décompositions ayant des zones géométriques communes à plusieurs sous-domaines. Pour les méthodes sans recouvrement, les parties d'interface partagées se résument à des points communs entre deux sous-domaines au minimum.

4.3.3.3 Opérations nécessitant des communications collectives

Les tâches de type `DomainTask` peuvent aussi effectuer des opérations globales, i.e., des opérations nécessitant des communications collectives. Dans AHPIK, il s'agit d'opérations de réduction équivalentes à la primitive MPI_ALLREDUCE offerte par MPI. Elles servent à effectuer des produits scalaires globaux ou bien

des calculs de norme pour le contrôle de convergence de la méthode itérative globale. Classiquement, ces opérations représentent un point de synchronisation globale car elles nécessitent des données de tous les processeurs. Dans AHPIK, les réductions peuvent se dérouler de manière synchrone ou asynchrone. La sélection du mode de fonctionnement de ces opérations se fait à l'aide de la fonction SelectControlMode(), comme indiqué dans l'extrait de code suivant :

```
class user_task : public DomainTask
{
  public :
    void operator()()
    {
        // ...
        Vector v1, v2 ;
        domain : :SelectControlMode(SyncControlMode) ;
        // ...
        local_dot = v1.dot(v2) ;
        global_dot = domain : :Reduce(local_dot, Sum) ;
        // ...
    }
};
```

Dans cet exemple, l'opération de réduction sert à calculer un produit scalaire global entre deux vecteurs distribués. Cette opération se déroule de façon synchrone : chaque processus léger appelant Reduce bloque jusqu'à ce que tous les autres processus aient exécuté cette opération. Lorsque l'on choisit un mode de fonctionnement asynchrone, l'opération ne bloque pas. Elle retourne la valeur de la dernière réduction achevée. Ce mode de fonctionnement est utile lorsque l'opération se trouve à l'intérieur d'une boucle, c'est le cas du calcul de la norme de deux vecteurs servant à déterminer l'erreur globale entre deux itérations consécutives. Cela suppose que la boucle ne contienne qu'une opération de réduction. Lorsque ce n'est pas le cas, les réductions asynchrones se font par l'intermédiaire d'objets permettant de distinguer chaque opération en cours. Il s'agit d'objets de type Reductor, qui doivent être initialisés (lignes 7 et 8 de l'exemple ci-dessous) et ensuite registrés par le support exécutif de AHPIK (lignes 10 et 11).

```
class user_task : public DomainTask
{
  public :
    void operator()()
    {
      domain : :SelectControlMode(AsyncControlMode) ;
      // ...
      Reductor<double> rnorm1(/* valeur initiale */) ;
      Reductor<double> rnorm2(/* valeur initiale */) ;
      // ...
      domain : :NewReductor(rnorm1) ;
      domain : :NewReductor(rnorm2) ;
      for (i = 0 ; i < max_iter ; i++)
      {
        // ...
        norm1 = rnorm1.Reduce(resid1, Sum) ;
        norm2 = rnorm2.Reduce(resid2, Sum) ;
        // ...
      }
    }
};
```

La mise en œuvre des opérations globales utilise un processus léger “coordinateur” responsable de la collecte des contributions de chaque sous-domaine et du calcul du résultat qui sera ensuite retourné aux processus légers concernés. Lorsque les opérations sont synchrones, la fonction Reduce implémentée envoie la contribution locale à l’opération de réduction, puis attend le résultat provenant du coordinateur. Dans le cas des réductions asynchrones, la fonction Reduce envoie la contribution locale et retourne immédiatement la valeur de la dernière réduction achevée. Ce mode de fonctionnement utilise, pour chaque sous-domaine, un processus léger additionnel chargé de recevoir le résultat du coordinateur.

4.3.3.4 Lancement des tâches de calcul d’interface

Les tâches de type `DomainTask` peuvent également lancer des tâches de calcul d’interface, i.e., d’affecter des tâches aux processus légers dédiés à ce propos (voir paragraphe 5.2). La définition d’une tâche d’interface suit le même principe décrit au paragraphe 4.3.3.1 : il s’agit d’une classe-fonction C++ surchargeant l’opérateur ’()’. Cette classe-fonction doit être dérivée de la classe `IfaceTask` prédéfinie par

le noyau de AHPIK. L'affectation de ces tâches aux processus légers de calcul d'interface se fait à l'aide de la fonction domain : :SpawnIfaceTasks() appelé par une tâche de type `DomainTask`. Cela s'écrit comme suit :

```
class
UserIfaceTask : public IfaceTask
{
    public :
        void operator()( [... paramètres ...] )
        {
            [... corps de la fonction ...]
        }
};
class
UserDomainTask : public DomainTask
{
    public :
        void operator()()
        {
            // ...
            domain : :SpawnIfaceTasks(UserIfaceTask()) ;
            // ...
        }
};
```

4.3.4 Mise en œuvre parallèle avec ATHAPASCAN-0

La mise en œuvre de AHPIK s'appuie fortement sur le paradigme de processus légers communicants. Nous avons choisi d'utiliser la bibliothèque ATHAPASCAN-0 pour deux raisons : premièrement, la couche de plus haut niveau ATHAPASCAN-1 était encore en développement au début de ce travail. Deuxièmement, nous avons voulu profiter des fonctionnalités de plus bas niveau offertes par ATHAPASCAN-0 pour un contrôle plus fin du placement et de l'ordonnancement des processus légers, ainsi que des communications entre eux.

Afin d'obtenir un maximum d'indépendance de la bibliothèque sous-jacente, les appels aux fonctions ATHAPASCAN-0 sont complètement encapsulées dans le noyau de AHPIK. Cela permet de suivre l'évolution de l'environnement ATHAPASCAN, ou même d'utiliser une autre bibliothèque mariant les processus légers et les

communications (par exemple lorsque des implémentations de MPI-2[85] seront disponibles).

L'ensemble des fonctionnalités de ATHAPASCAN-0 que nous avons utilisé n'est pas très vaste. La création de processus légers est encapsulée dans une classe permettant de mettre en œuvre des objets "actifs" : il s'agit d'objets ayant une fonction `Start` qui démarre l'exécution en parallèle d'une fonction prédéfinie. Tous les paramètres nécessaires à l'exécution de cette fonction sont passés à cet objet lors de sa création ou initialisation. La création du processus légers a donc toujours la même forme car les paramètres n'ont pas besoin d'être spécifiés à ce moment.

Étant donné que AHPIK emploie des processus légers spécialisés dans les communications, nous utilisons essentiellement des primitives d'envoi et réception synchrones offertes par ATHAPASCAN-0. Des primitives de réception asynchrones ne sont utilisées que pour mettre en œuvre la terminaison des processus légers récepteurs qui sont indéfiniment en attente de données (dans ce cas on utilise une réception asynchrone multiple : le processus léger redevient actif soit avec la réception des données d'interface, soit avec un message de terminaison).

4.3.5 Mise en œuvre parallèle avec MPI

Nous avons également mis en œuvre une version de AHPIK basée sur la bibliothèque de communication MPI. Un programme écrit à l'aide de AHPIK peut s'exécuter sur l'une ou l'autre version sans modification du code source. Cependant, la version MPI a une limitation : elle ne permet, en l'état, d'affecter plusieurs sous-domaines par processeur.

Cette version de AHPIK est totalement basée sur des primitives de communication asynchrones offertes par MPI afin de reproduire le même comportement de la version basée sur les processus légers communicants.

5

L'environnement objet de AHPIK

Sommaire

5.1 Objets pour les méthodes de décomposition de domaine . . . 109
5.1.1 Classes caractérisant les problèmes d'EDP 109
5.1.2 Composants géométriques de la décomposition du maillage 110
5.1.3 Support au développement de solveurs 111
5.1.4 Les calculs spécifiques à chaque solveur 112
5.2 Mise en œuvre parallèle 114
5.3 Schémas parallèles génériques pour la décomposition de domaine 115
5.3.1 Composition d'un schéma 115
5.3.2 Les différents schémas et leur représentation graphique . 121
5.4 Les méthodes de décomposition de domaine implantées . . . 129
5.4.1 Schwarz additif synchrone 129
5.4.2 Schwarz additif asynchrone 131
5.4.3 Schur primal, méthode de point fixe 132
5.4.4 Schur dual, méthode de point fixe 133
5.4.5 Schur dual, méthode du gradient conjugué 135
5.4.6 Méthode des éléments joints 138

L'environnement AHPIK a été conçu pour répondre à deux objectifs. Tout d'abord, AHPIK est un outil modulaire dédié à la mise en œuvre rapide des méthodes mathématiques de décomposition de domaine et d'adaptation de maillage permettant la résolution de problèmes d'EDP complexes. Le second but est de proposer un environnement parallèle construit autour des techniques de multiprogrammation légère pour faciliter l'ordonnancement des tâches lorsque les calculs sont déséquilibrés, l'utilisation d'algorithmes asynchrones devenant naturelle.

Dans ce cadre, la programmation par objets est intéressante car elle permet d'exprimer aisément une conception modulaire et générique pour le support au développement de solveurs parallèles. Ce paradigme de programmation nous a permis de cacher de l'utilisateur les détails d'une mise en œuvre parallèle à base de processus légers, et d'utiliser un même noyau parallèle pour la mise en œuvre d'un ensemble de méthodes.

Dans ce chapitre, nous présentons d'abord (paragraphe 5.1) les différents objets fournis par AHPIK pour la résolution de problèmes d'EDP par une méthode de décomposition de domaine. L'exposé relie les différentes classes d'objets aux couches logicielles que nous avons introduites dans le paragraphe 3.3, figure 3.4 (afin de faciliter la lecture, nous reproduisons cette illustration dans la figure 5.1 ci-dessous). La mise en œuvre parallèle générique basée sur les processus légers est présentée dans le paragraphe 5.2. On y distingue les interactions entre les différentes classes d'objets. Finalement, le paragraphe 5.3 présente les schémas de décomposition de domaine implantés.

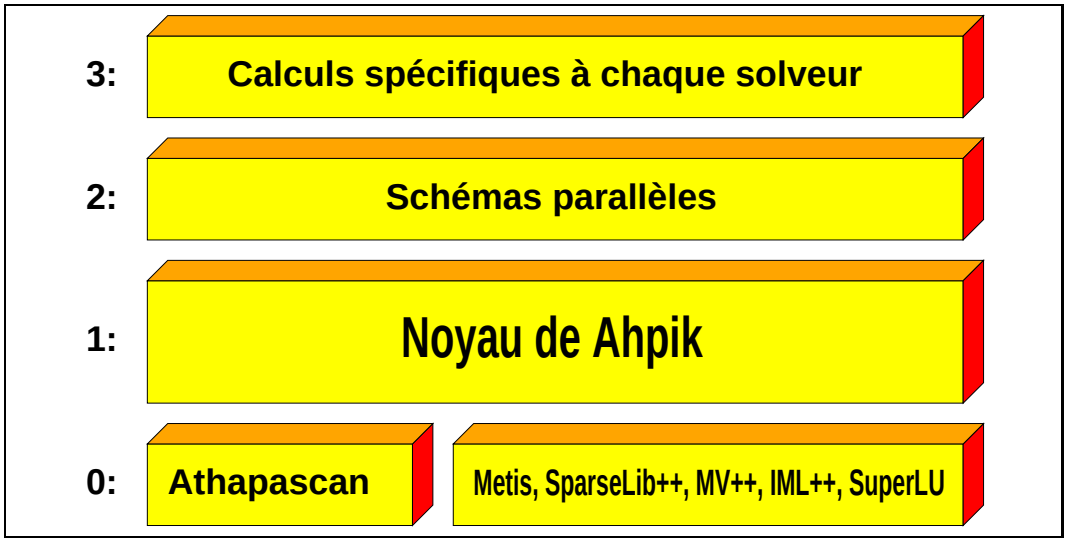

Figure 5.1 *Organisation de* AHPIK.

5.1 Objets pour les méthodes de décomposition de domaine

Les éléments mathématiques intervenant dans la résolution de problèmes d'EDP par une méthode de décomposition de domaine sont très différents les uns des autres : il est intéressant de choisir une représentation objet pour ceux-ci. Cette conception modulaire facilite l'extension des couches 2 et 3 de la bibliothèque AHPIK pour permettre la résolution de nombreux problèmes d'EDP. La couche 1, le noyau de AHPIK, reste valide quels que soient le problème d'EDP et la méthode de décomposition de domaine choisis.

Les paragraphes 5.1.1 et 5.1.2 présentent les principales classes d'objets intervenant dans la définition d'un problème d'EDP dans un sous-domaine (ou dans le domaine complet dans le cas sans décomposition de domaine). Les classes relatives à l'enchaînement de tâches spécifiques aux solveurs (point fixe, gradient conjugué) sont décrites dans le paragraphe 5.1.3. Elles implémentent les schémas parallèles utilisés pour résoudre itérativement le problème issu de la modélisation par une méthode de décomposition de domaine. Les classes décrivant les calculs spécifiques à une itération sont présentées dans le paragraphe 5.1.4.

5.1.1 Classes caractérisant les problèmes d'EDP

AHPIK propose deux classes pour la discrétisation de problèmes d'EDP : la classe `FEM` pour une méthode d'éléments finis et la classe `FDM` pour les schémas de différences finies. La fonction principale de la classe `FEM` est de construire le système linéaire (classe `LinearEqSystem`) correspondant au problème d'EDP discrétisé à l'aide d'un maillage (classe `Mesh`). Dans AHPIK il y a également une classe nommée `Adaptive2DMesh` qui est utilisée pour des problèmes avec adaptation de maillage. La classe `FEM` encapsule les procédures de base de la méthode de discrétisation : construction des fonctions de base, procédures d'intégration, etc. La classe `FDM` diffère de la classe `FEM` dans la mesure où la construction du système n'a pas besoin du maillage. L'assemblage est fait de manière classique à l'aide de « stencils » (voir section 3.2.2). Les classes `FEM` et `FDM` servent à définir d'autres classes, elles sont dites *abstraites* car elles ne peuvent pas être instanciées directement.

La hiérarchie de classes pour les solveurs classiques [27, 10] de systèmes linéaires (figure 5.2) est d'un niveau d'abstraction proche des formulations mathématiques.

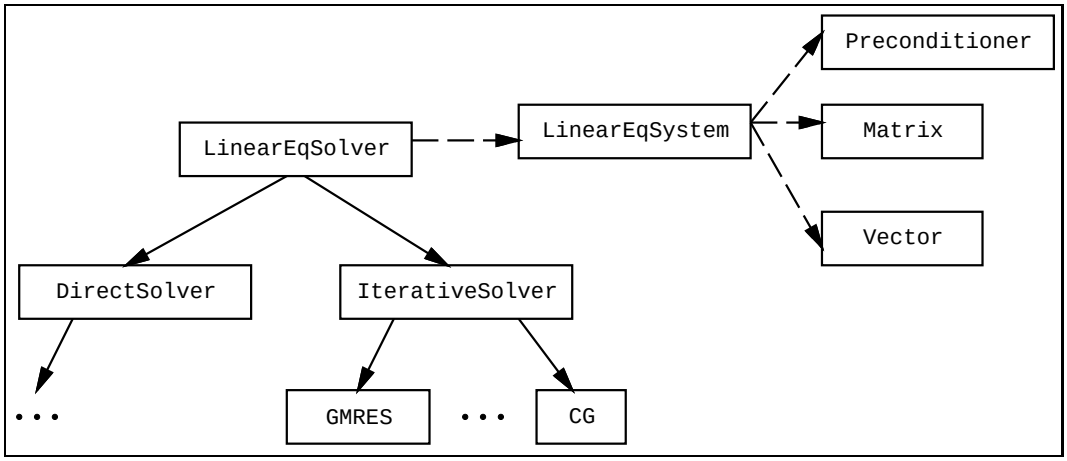

Figure 5.2 *Hiérarchie de classes pour les solveurs classiques de systèmes linéaires*

Dans la figure 5.2, les solveurs itératifs et directs sont dérivés de la classe abstraite `LinearEqSolver` qui utilise la classe abstraite `LinearEqSystem` pour la représentation du système linéaire à résoudre. Les différents composants des systèmes linéaires (matrices, vecteurs, etc.) sont les classes `Matrix`, `Vector` et `Preconditioner`. Cette organisation hiérarchique en classes a permis l'emploi de bibliothèques externes telles que SparseLib++[55] pour le stockage et la manipulation de matrices creuses et de vecteurs, IML++[130] pour les méthodes itératives de résolution de systèmes linéaires, et SuperLU[52] pour les méthodes directes.

À chaque nouveau problème, le programmeur doit écrire une classe dérivée de `FEM` ou de `FDM`. Cette classe définit les fonctions calculant les contributions de chaque nœud ou de chaque élément du maillage aux matrices de rigidité et de masse, ainsi que les conditions aux limites. Ces fonctions sont utilisées pour l'assemblage du système linéaire correspondant au problème d'EDP discret.

5.1.2 Composants géométriques de la décomposition du maillage

D'un point de vue informatique, l´étape géométrique de décomposition du domaine est traité à l'aide de la classe `SubDomain`. Elle contient la description géométrique d'un maillage (nœuds, éléments et arêtes) à laquelle on rajoute les descriptions des parties frontières, des interfaces et, pour les méthodes avec recouvrement, les frontières de recouvrement. Chacun des sous-domaines est associé à un objet de cette classe.

La construction hiérarchique de ces objets géométriques est illustrée dans la figure 5.3. Dans cette figure, une flèche en trait plein de A vers B indique que la classe B hérite de la classe A, tandis qu'une flèche en pointillée de A vers B indique que la classe B utilise la classe A pour sa construction.

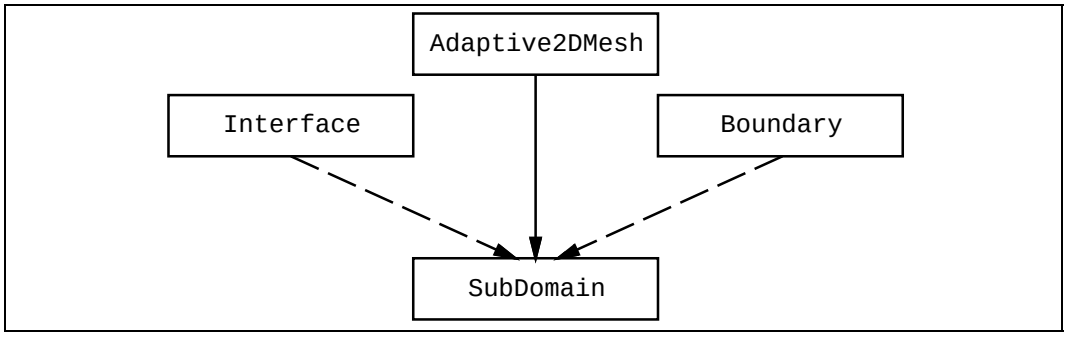

Figure 5.3 *Hiérarchie de classes pour la représentation des sous-domaines*

Plus précisément, la classe `Adaptive2DMesh` code la description géométrique adéquate pour représenter des maillages adaptatifs (éléments finis) structurés ou non-structurés en deux dimensions. Selon les exigences du problème à résoudre (par exemple maillage en trois dimensions), elle peut être remplacée par une autre. La classe `Boundary` fournit les informations sur les points de discrétisation situés sur la frontière des sous-domaines commune au domaine global. La classe `Interface` décrit l'ensemble des points de discrétisation appartenant aux interfaces, celle-ci contient plus ou moins d'informations, elle est adaptée à la méthode de décomposition de domaine.

Chacun des sous-domaines est associé à un objet de la classe `SubDomain`.

5.1.3 Support au développement de solveurs

Dans AHPIK, le support au développement de solveurs parallèles de décomposition de domaine est fourni par la classe de base `DomainTask` (voir paragraphe 4.3.3). Les classes de la couche 2 (voir figure 5.1) dérivent de `DomainTask`. Elles précisent l'enchaînement des étapes de résolution (point fixe, paragraphe 5.3.1.1, ou gradient conjugué, paragraphe 5.3.1.2) du problème issu de la modélisation par une méthode de décomposition de domaine.

La classe `DomainTask` appartient au noyau de AHPIK. Au démarrage du processus de résolution, le noyau de AHPIK crée le réseau de processus légers et détermine le graphe des communications nécessaire au fonctionnement d'un objet dérivé

de la classe `DomainTask`. Cette classe est donc la couche logicielle commune à tous les solveurs de décomposition de domaine, elle est abstraite car elle ne contient pas les instructions définissant le schéma de résolution par une méthode de décomposition de domaine.

5.1.4 Les calculs spécifiques à chaque solveur

Dans AHPIK, les aspects numériques sont complètement dissociés de la gestion du parallélisme. Étant donné un schéma parallèle, l'implantation d'une nouvelle méthode de décomposition de domaine, i.e. d'un nouveau **solveur**, implique la définition d'une classe C++ fournissant quelques fonctions pré-définies qui "remplissent" le schéma parallèle. Cette classe diffère d'une méthode à l'autre à cause des spécificités mathématiques : les structures de données et les opérateurs élémentaires sont adaptés à la méthode considérée.

Prenons comme exemple un schéma de résolution du problème d'interface par une méthode de point fixe (voir 5.3.1.1). La figure 5.4 illustre la définition d'un solveur basé sur ce schéma.

```
class NewSolver
{
    public :
        // Calcul local sur les variables intérieures au sous-domaine.
        template<class LinearEqSystem>
        void LocalComputation(LinearEqSystem& system);

        // Calcul/mise a jour des variables associées α l'interface d'indice k.
        // Le vecteur v représente la contribution externe au calcul d'interface.
        template<class Vector>
        void InterfaceComputation(int k, const Vector& v);

        // Définition de la contribution locale.
        template<class Vector>
        void LocalContribution(int k, Vector& v);

};
```

Figure 5.4 *Déclaration d'une classe dérivée d'un schéma parallèle* AHPIK

Les principales fonctions à fournir pour compléter cette classe sont :
- `LocalComputation` : cette fonction spécifie la méthode utilisée pour la résolution d'un problème d'EDP dans un sous-domaine, ce qui permet d'utiliser AHPIK dans le cas d'applications nécessitant le couplage de modèles différents ;
- `LocalContribution` : cette fonction extrait les informations sur l'interface ;
- `InterfaceComputation` : cette fonction calcule et/ou met à jour les variables associées à une interface ;

Ces trois fonctions manipulent la matrice de rigidité et les vecteurs représentant le second membre, la solution à l'itération antérieure et la solution actuelle, qui sont des objets appartenant à la classe `LinearEqSystem`.

La figure 5.5 illustre nos propos en prenant pour exemple la fonction `LocalComputation` qui effectue la résolution d'un système linéaire par une méthode directe. On remarque l'appel à la fonction `SolveLU` définie dans la classe `DirectSolver`. Ses paramètres sont les composants du système linéaire : matrice, vecteur des inconnues et second membre.

```
template<class LinearEqSystem>
void
NewSolver : :LocalComputation(LinearEqSystem& system)
{
    // Lecture de l'indice de la solution α l'itération actuelle.
    int curr = system.CurrentSolutionIndex() ;

    // Résolution de Ax=b par une méthode directe.
    SolveLU(system.SelectMatrix(), // matrice de rigidité
            system.SelectSolution(curr), // solution cherchée
            system.SelectRHS()) ;    // vecteur du second membre
}
```

Figure 5.5 *Exemple d'implémentation de* `LocalComputation`.

Lors de l'implantation d'une nouvelle méthode, le programmeur travaille donc sur un objet de la classe `LinearEqSystem`. Il n'a *a priori* pas à refaire le schéma d'échange de messages. La méthode utilisée pour la résolution du système (gradient conjugué, relaxation, etc.) est définie par l'utilisateur. Il peut choisir une des méthodes déjà disponibles dans AHPIK ou bien fournir ses propres routines de calcul.

La structure modulaire que nous venons de présenter a été utilisée pour la mise en œuvre d'une méthode sans recouvrement, synchrone (les interfaces étant traitées par une méthode de Schur duale) et de méthodes avec recouvrement (Schwarz) synchrone et asynchrone. Notre objectif n'est pas de développer la partie numérique de ces codes, mais d'offrir un outil d'implantation flexible et modulaire pour des exécutions parallèles. Le portage d'autres codes numériques est envisagé.

5.2 Mise en œuvre parallèle

Une application AHPIK est globalement organisée dans un modèle SPMD (*Single Program Multiple Data*). Les solveurs parallèles de EDP sont implantés comme un ensemble de tâches interagissantes, chaque tâche étant exécutée par un processus léger spécialisé.

Dans AHPIK, un processus lourd est associé à chaque processeur[1]. Dans la méthode de décomposition de domaine, ce processus gère un (ou plusieurs) sous-domaine pour lequel sont définis un ensemble de processus légers parmi les processus suivants :

- processus léger de **calculs locaux** : il exécute la tâche T_Ω^k pour le sous-domaine Ω_k en appelant la fonction `LocalComputation`, puis il extrait les données d'interface en appelant la fonction `LocalContribution`, ce sont les fonctions mentionnées dans 5.1.4. Ce processus appelle régulièrement les processus légers de **calcul d'interface** et de **communication**, et occasionnellement le processus léger de **contrôle de convergence** (par exemple pour les méthodes de point fixe) ;
- processus léger de **calculs d'interface** : il exécute la tâche T_γ^{kl} pour l'interface γ_{kl} entre les sous-domaines Ω_k et Ω_l. Il appelle la fonction `InterfaceComputation`, mentionnée en 5.1.4 ;
- processus légers de **communication** (envoi ou réception) : ils envoient ou reçoivent les contributions locales/externes pour le calcul de l'interface γ_{kl}.
- processus léger de **contrôle de convergence** : il calcule l'erreur locale correspondant à l'itération courante dans le domaine Ω_k.

Ces processus légers sont définis dans des classes du noyau de AHPIK. Ils sont ordonnancés selon la disponibilité des données dont ils dépendent. Quand un sous-domaine a plus d'une interface, des calculs d'interface peuvent être exécutés en parallèle par les différents processus légers dès que leurs données d'entrée sont

[1] Un seul processeur pourrait gérer plusieurs processus lourds.

disponibles.

Pour compenser l'effet négatif de la synchronisation inhérente à certaines méthodes, AHPIK permet d'affecter plus d'un sous-domaine par nœud. Cette technique est utile pour recouvrir les communications avec du calcul.

Génériquement, le comportement des communications et synchronisations entre les processus légers est défini par un des schémas parallèles (voir paragraphe 5.3). Ce comportement est illustré dans la figure 5.6. On y retrouve tous les processus légers cités précédemment.

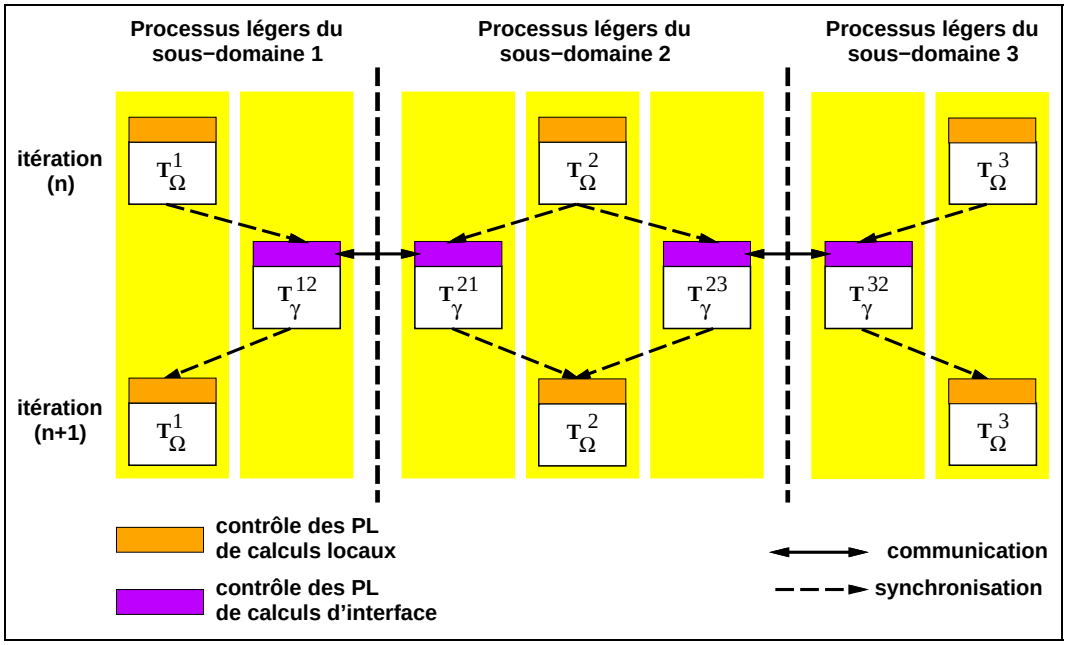

Figure 5.6 *Schéma parallèle générique pour une décomposition en 3 sous-domaines*

5.3 Schémas parallèles génériques pour la décomposition de domaine

5.3.1 Composition d'un schéma

Dans AHPIK, les schémas parallèles définissent l'enchaînement des tâches (calculs et communications) au cours du processus de résolution itératif. Ils comprennent la sélection d'un mode de communication (synchrone ou asynchrone) pour les calculs d'interface et pour le contrôle de convergence (opérations globales).

Les schémas sont génériques. Ils sont implantés sous forme de fonctions « templates » C++ paramétrées par des classes représentant le sous-domaine, les éléments du système linéaire à résoudre, et les calculs spécifiques à une méthode de décomposition de domaine. Cela permet, par exemple, d'utiliser un même schéma pour des problèmes bidimensionnels ou tridimensionnels à partir du moment où les classes correspondantes existent. De même, les schémas sont indépendants des structures de données représentant le système linéaire. L'utilisateur peut ainsi opter pour ses propres classes de vecteurs, matrices et préconditionneurs ou bien utiliser les classes disponibles dans AHPIK.

Cette construction générique implique l'utilisation de classes ayant quelques fonctionnalités prédéfinies. Ces fonctionnalités varient selon la méthode itérative globale qui caractérise un solveur de décomposition de domaine. Nous allons approfondir la question dans les deux paragraphes suivants. Le premier présente les solveurs itératifs basés sur une méthode de point fixe. Nous présentons ensuite les spécificités caractérisant les solveurs basés sur une méthode de Krylov en nous appuyant sur la méthode du gradient conjugué.

5.3.1.1 Schéma pour solveurs reposant sur une méthode de point fixe

Les schémas itératifs se caractérisent par le calcul d'approximations successives d'une fonction $f(x) = x$, la séquence générée ayant la forme

$$x^{k+1} = x^k + A(x^k),$$

où $A(x) = f(x) - x$. Dans ce cas, trouver le point fixe de $f(x)$ revient à résoudre $A(x) = 0$.

Le schéma parallèle AHPIK effectuant le calcul itératif global d'un solveur de décomposition de domaine par une méthode de point fixe est présenté dans la figure 5.7. On y distingue :

- la sélection du mode de contrôle de convergence (ligne 10). Dans cet exemple, il s'agit d'un contrôle de convergence asynchrone où les opérations de réduction (ligne 23) ne bloquent pas le processus léger appelant. Une nouvelle itération peut ainsi s'initier avant que tous les calculs locaux soient terminés pour tous les sous-domaines ;
- la sélection du mode de communication entre voisins (ligne 11). Dans cet exemple, les communications sont synchrones ;
- la définition des liens communicants associés à chaque interface du sous-domaine traité (lignes 12 à 16) ;

– le calcul itératif (lignes 18 à 29) ;
– l'appel à la fonction de calculs locaux (ligne 20) ;
– le lancement des tâches de calcul d'interface (ligne 21). Ces tâches s'exécutent en parallèle pour chaque interface entre le sous-domaine traité et un sous-domaine voisin ;
– le calcul du résidu local (ligne 22) ;
– le calcul de l'erreur correspondante au calcul itératif global (ligne 23).

```
1  template<class DDMethod, class System>
2  int
3  DDSyncFPDriver(const DDMethod& method, System& system,
4                 int& max_iter, double& tolerance)
5  {
6      int i, k, iter;
7      double err;
8      Residual resid;
9
10     SelectControlMode(AsyncControlMode);
11     SelectCommMode(SyncCommMode);
12     for (i = 0; i < IfaceCount(); i++)
13     {
14        InterfaceDataOut(i, method.DataOut(i));
15        InterfaceDataIn(i, method.DataIn(i));
16     }
17
18     for (iter = 1; iter <= max_iter; iter++)
19     {
20        method.LocalComputation(system);
21        SpawnIfaceTasks(IfaceTask(), method, system);
22        resid = system.CalcResidual(L2Norm);
23        err = Reduce(resid, L2Norm);
24
25        if (err <= tolerance)
26        break;
27
28        system.NewCurrentSolution();
29     }
30 }
```

Figure 5.7 *Schéma parallèle typique d'une méthode de point fixe.*

Lorsque l'on utilise un schéma comme celui de la figure 5.7, la classe repré-

sentant la méthode de décomposition de domaine doit obligatoirement fournir les fonctions suivantes :

- `DataOut` et `DataIn` qui retournent respectivement la donnée à envoyer au $i^{ème}$ sous-domaine voisin et l'adresse de la place mémoire pour la donnée qui sera reçue de ce sous-domaine. Ces fonctions fournissent les paramètres d'entrée pour les fonctions `InterfaceDataOut` et `InterfaceDataIn` du noyau de AHPIK qui créent les processus légers de communication (cf. paragraphe 4.3.3.2) ;
- `LocalComputation` et `InterfaceComputation` qui définissent respectivement les calculs locaux et les calculs d'interface, ainsi que `LocalContribution` qui calcule la donnée à envoyer au $i^{ème}$ sous-domaine voisin (cf. paragraphe 5.1.4).

5.3.1.2 Schéma pour solveurs de type gradient conjugué

Les méthodes de type gradient conjugué sont souvent utilisées pour accélérer la convergence du calcul itératif global d'un solveur de décomposition de domaine. Lorsque l'on utilise ce type de méthode, le schéma parallèle diffère légèrement de celui présenté au paragraphe précédent. En effet, les produits scalaires et le calcul de norme (voir paragraphe 4.2.2) introduisent des points de synchronisation globale supplémentaires. Or un schéma comme celui de la figure 5.7 ne contient qu'un seul point de synchronisation globale. Nous avons donc inclus dans AHPIK un schéma parallèle spécifique pour les solveurs de type gradient conjugué. Ce schéma suppose que les interfaces entre les sous-domaines sont gérées de manière disjointe : il résout plusieurs systèmes d'interface au lieu d'un système unique (voir ci-dessous pour plus de détails sur l'algorithme). Nous avons utilisé ce schéma pour la mise en œuvre d'un solveur basé sur l'algorithme d'Uzawa pour la méthode de Schur dual (voir paragraphe 5.4.5). Ce schéma est néanmoins générique et peut par exemple s'appliquer à la méthode du complément de Schur primal lorsque les interfaces ne possèdent pas de points en commun. La figure 5.8 présente le code C++ correspondant. On y distingue :

- la sélection d'un comportement synchrone pour le contrôle de convergence et la communication entre voisins (lignes 9 et 10). L'algorithme du gradient conjugué parallèle est typiquement synchrone, la convergence de la méthode n'étant pas prouvée, à notre connaissance, pour des itérations asynchrones ;
- la définition des liens communicants associés à chaque interface du sous-domaine traité (lignes 13 et 14) ;
- l'initialisation de la méthode (lignes 17 à 21), puis le calcul itératif (lignes 23

à 29) ;
- les appels aux fonctions de calculs locaux (lignes 17, 25 et 30). Les deux premières correspondent à la résolution d'un système linéaire (cf. algorithme au paragraphe 4.2.2). La troisième correspond à la mise à jour du vecteur de solution correspondant au sous-domaine traité ;
- le lancement des tâches de calcul d'interface (lignes 18, 26, 31 et 37) ;
- le calcul du résidu initial (ligne 19) puis du résidu à chaque itération (ligne 32), ainsi que le calcul d'un produit scalaire global (ligne 27) pour déterminer le pas de descente α (ligne 28) à chaque itération.

Les tâches lancées par `SpawnIfaceTasks` s'exécutent en parallèle pour chaque interface entre le sous-domaine traité et un sous-domaine voisin. La figure 5.9 fournit un exemple du code C++ exécuté par ces tâches. Elles comprennent typiquement :
- l'appel à une fonction qui calcule la contribution locale au calcul d'interface (ligne 8) ;
- un échange de données entre sous-domaines voisins (ligne 9) ;
- l'appel à une fonction effectuant un calcul d'interface (ligne 10).

Remarque 1 : L'algorithme générique du gradient conjugué présenté au paragraphe 4.2.2 admet plusieurs décompositions en tâches de calculs locaux et de calculs d'interface. La décomposition que nous avons choisi met en évidence les points de synchronisation globales. Le schéma de la figure 5.8 ne s'occupe que de l'enchaînement des tâches, les calculs effectifs sont fournis par une classe (ici `DDMethod`) externe au schéma. On pourrait tirer parti du caractère générique de l'algorithme 4 pour construire un schéma contenant aussi les calculs et les structures de données représentant les vecteurs inhérents à la méthode du gradient conjugué. Dans ce cas, un objet de la classe `DDMethod` n'aurait besoin que d'indiquer les différents sous-blocs du système, conformément à la décomposition générique présenté au paragraphe 2.3.3.3.

Remarque 2 : On peut envisager un schéma similaire où le traitement des interfaces est fait de manière globale, i.e., où l'on résout un système d'interface unique. Nous ne l'avons pas programmé, mais l'identification des tâches pour un tel schéma peut se faire facilement.

```
1  template<class DDMethod, class System>
2  int
3  DDSyncCGDriver(const DDMethod& method, System& system,
4                 int& max_iter, double& tolerance)
5  {
6      int i, iter;
7      double normgd0, normgd1, psi_dot_p, alpha, beta;
8
9      SelectControlMode(SyncControlMode);
10     SelectCommMode(SyncCommMode);
11     for (i = 0; i < IfaceCount(); i++)
12     {
13        InterfaceDataOut(i, method.DataOut(i));
14        InterfaceDataIn(i, method.DataIn(i));
15     }
16
17     method.LocalComputation1(system);
18     SpawnIfaceTasks(IfaceTask1(), method, system);
19     normgd0 = Reduce(method.CalcResidual(), Sum);
20     if (normgd0 <= tolerance)
21         return 0;
22
23     for (iter = 1; iter <= max_iter; iter++)
24     {
25         method.LocalComputation2a(system);
26         SpawnIfaceTasks(IfaceTask2a(), method, system);
27         psi_dot_p = Reduce(method.CalcDot(), Sum);
28         alpha = normgd0 / psi_dot_p;
29
30         method.LocalComputation2b(system, alpha);
31         SpawnIfaceTasks(IfaceTask2b(), alpha, method, system);
32         normgd1 = Reduce(method.CalcResidual(), Sum);
33         if (normgd1 <= tolerance)
34             return iter;
35
36         beta = normgd1 / normgd0;
37         SpawnIfaceTasks(IfaceTask2c(), beta, method, system);
38         normgd0 = normgd1;
39     }
40 }
```

Figure 5.8 *Schéma parallèle typique d'une méthode de gradient conjugué.*

```
1  class
2  IfaceTask1 : public IfaceTask
3  {
4      public :
5          template<class DDMethod, class System>
6          void operator()(DDMethod& method, System& system)
7          {
8              method.LocalContribution1(IfaceId(), system) ;
9              IfaceExchange() ;
10             method.InterfaceComputation1(IfaceId(), system) ;
11         }
12 };
```

Figure 5.9 *Exemple de code pour une tâche de calcul d'interface.*

5.3.2 Les différents schémas et leur représentation graphique

Dans le paragraphe précédent, nous avons montré que AHPIK offre des mécanismes flexibles pour la composition des schémas parallèles. Les multiples possibilités pour l'enchaînement des tâches et les modes de communication entre elles conduisent à des programmes parallèles ayant des comportements différents face aux éventuels déséquilibres de charge. Cela s'observe surtout sur plusieurs itérations de la méthode de décomposition de domaine. De même, chaque itération d'un solveur (point fixe ou gradient conjugué, par exemple) a un comportement parallèle caractéristique qui résulte des points de synchronisation globale et des points de synchronisation entre sous-domaines voisins.

Dans ce paragraphe, nous illustrons le comportement de différents schémas parallèles à l'aide d'une représentation graphique de leur exécution. Les diagrammes que nous allons présenter ont été générés à l'aide de l'outil PAJÉ [2][51], développé au sein du projet APACHE. Cet outil permet la visualisation de traces d'exécution de programmes parallèles basés sur le paradigme de processus légers communicants. Les traces que nous allons visualiser ont été générées par une version instrumentée du noyau de AHPIK.

La fenêtre principale de PAJÉ fournit un diagramme espace-temps montrant

[2]L'URL de la page officielle du projet est la suivante : `http://www-apache.imag.fr/software/paje`

l'activité des fils d'exécution dans les nœuds de la machine parallèle. Ce diagramme combine les états de chaque fil d'exécution et les communications entre eux. L'axe horizontal représente le temps. Les fils d'exécution sont groupés par nœuds suivant l'axe vertical. Les communications sont représentées par des flèches, tandis que l'état des fils d'exécution est représenté par des rectangles. Le diagramme utilise des couleurs pour indiquer le type de communication ou l'activité des fils d'exécution.

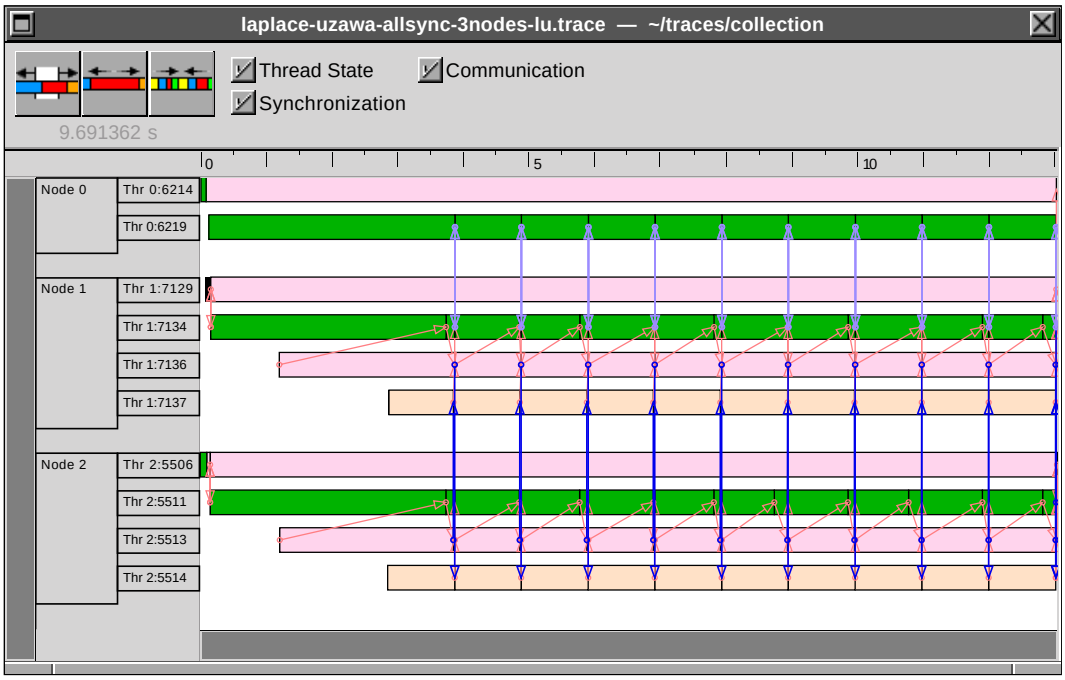

Figure 5.10 *Visualisation du schéma de point fixe synchrone pour la méthode d'Uzawa : cas parfaitement équilibré.*

La figure 5.10 illustre l'exécution d'un programme utilisant la méthode d'Uzawa pour résoudre le problème de Poisson sur un domaine décomposé en 2 sous-domaines. Afin de faciliter la compréhension du diagramme, nous avons utilisé 3 nœuds pour cette exécution, le nœud 0 étant consacré uniquement au calcul de l'erreur globale. Le premier processus léger de chaque nœud est responsable de l'initialisation de AHPIK, il n'intervient pas au cours du calcul. Sur les nœuds 1 et 2 on trouve, dans l'ordre d'apparition, les processus légers suivants :

- un processus léger responsable des tâches de calculs locaux
- un processus léger responsable des tâches de calcul d'interface, qui fait aussi les envois de données aux sous-domaines voisins
- un processus léger de réception des données d'interface

Chaque trait horizontal représente les différents états d'un processus léger. Les rectangles plus clairs indiquent les intervalles de temps où le processus léger est bloqué (en attente de communication ou lors d'une synchronisation). Les rectangles plus foncés correspondent aux intervalles où le processus léger est « activable ».

Le programme correspondant à la figure 5.10 effectue 10 itérations pour la convergence de la méthode de décomposition de domaine. La charge de calcul est bien équilibrée car le maillage d'origine a été découpé en deux maillages de taille identique et la résolution des problèmes locaux est faite par une méthode directe (factorisation LU). Cela se traduit par des échanges de données très régulières, avec des temps d'attente négligeables par rapport aux temps où les processeurs sont actifs.

On observe dans la figure 5.11 que cet équilibre est perturbé lorsque l'on utilise une méthode itérative pour la résolution des problèmes locaux, i.e. lorsque l'on travaille avec une méthode 2 niveaux (« two-stage methods »). Pour le problème en question, le solveur local sur le nœud 2 converge moins vite que sur le nœud 1. Par conséquent, ce dernier reste inactif entre deux itérations, en attente de communication avec le voisin.

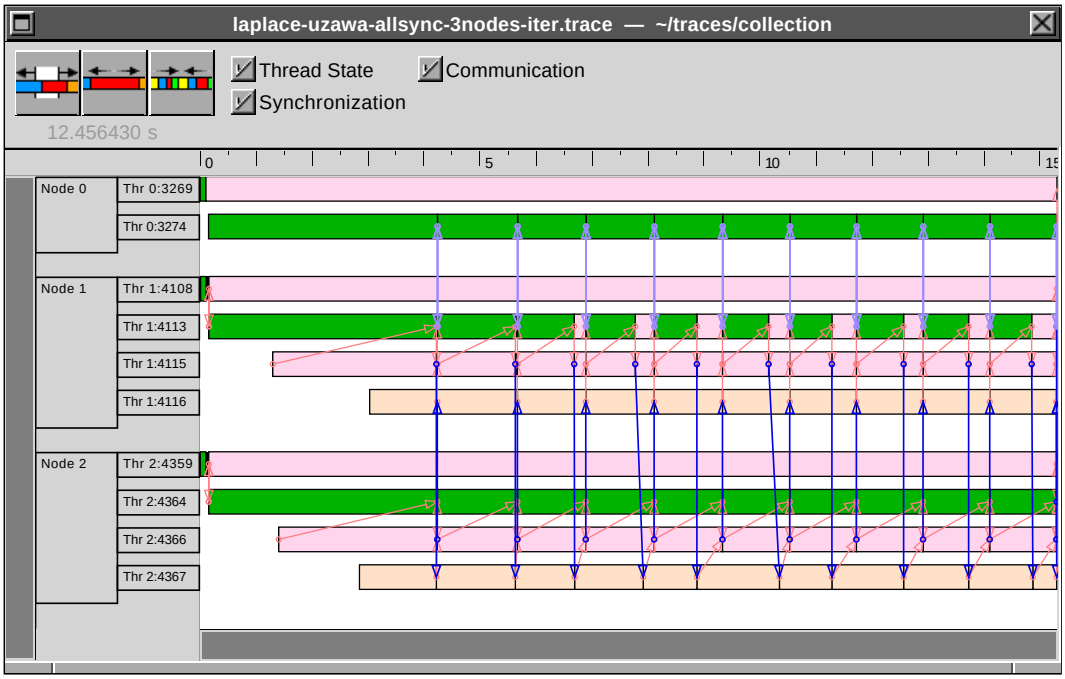

Figure 5.11 *Visualisation du schéma de point fixe synchrone pour la méthode d'Uzawa : déséquilibre de charge provenant de l'utilisation d'un solveur local itératif.*

Sur ces exemples, le contrôle de convergence (calcul de l'erreur globale) est effectué à l'aide d'un point de synchronisation entre processeurs après chaque itération de la méthode de décomposition de domaine. Dans ces diagrammes, les messages pour la mise en œuvre des réductions globales se confondent avec les communications entre voisins. La figure 5.12 montre l'exécution du même programme des figures précédentes pour une décomposition en 4 bandes verticales. Le sous-domaine 1 est affecté au nœud 1 et ainsi de suite. Le découpage géométrique est tel que les sous-domaines du milieu (2 et 3) ont 2 voisins, tandis que ceux situés aux extrémités (1 et 4) ne possèdent qu'un seul. Dans cet exemple, le maillage du premier sous-domaine est plus fin (1625 nœuds) que celui des autres sous-domaines (1105 nœuds dans chacun des maillages correspondants). Cela engendre des temps d'attente visibles sur les processus légers exécutant les tâches de calculs locaux. La synchronisation globale à la fin de chaque itération fait que l'exécution soit cadencée par le processeur plus chargé.

Il convient de remarquer que pour une méthode de point fixe, le calcul de l'erreur sert uniquement à arrêter le calcul itératif. La norme calculée n'entre pas dans les calculs de la prochaine itération, comme c'est le cas lorsque l'on utilise une méthode de Krylov. On peut donc effectuer le contrôle de convergence de façon asynchrone : au lieu d'attendre la valeur de l'erreur globale, chaque processeur ayant fini une itération entame le calcul de la prochaine (les communications entre voisins restent synchrones). Pour la décomposition en 4 bandes verticales citée précédemment, le choix d'un contrôle de convergence asynchrone produit le diagramme de la figure 5.13. On remarque sur ce diagramme un processus léger additionnel sur chaque nœud : il s'agit d'un processus léger qui attend que le processus léger "coordinateur" envoie les résultats des opérations globales (réductions) asynchrones. L'exécution est maintenant cadencée par les échanges de données entre sous-domaines voisins. Cela fait que les nœuds éloignés du nœud plus chargé effectuent plus d'itérations dans un même intervalle de temps.

Itérations asynchrones Nous nous intéressons maintenant à montrer le comportement parallèle d'un schéma de calcul effectuant des itérations asynchrones. À titre comparatif, nous commençons par présenter, dans la figure figure 5.14, l'exécution d'un programme utilisant une méthode de Schwarz synchrone classique pour résoudre le problème d'EDP

$$\left\{ \begin{array}{rcll} -10^{-2}\Delta u + 0.5\frac{\partial u}{\partial x} + 1.5\frac{\partial u}{\partial y} + 10u & = & f & \text{dans } \Omega, \\ u & = & 0 & \text{sur } \Gamma. \end{array} \right.$$

Le domaine d'origine est décomposé en 4 sous-domaines (bandes verticales) se recouvrant. Comme dans les exemples précédents, on affecte un sous-domaine par

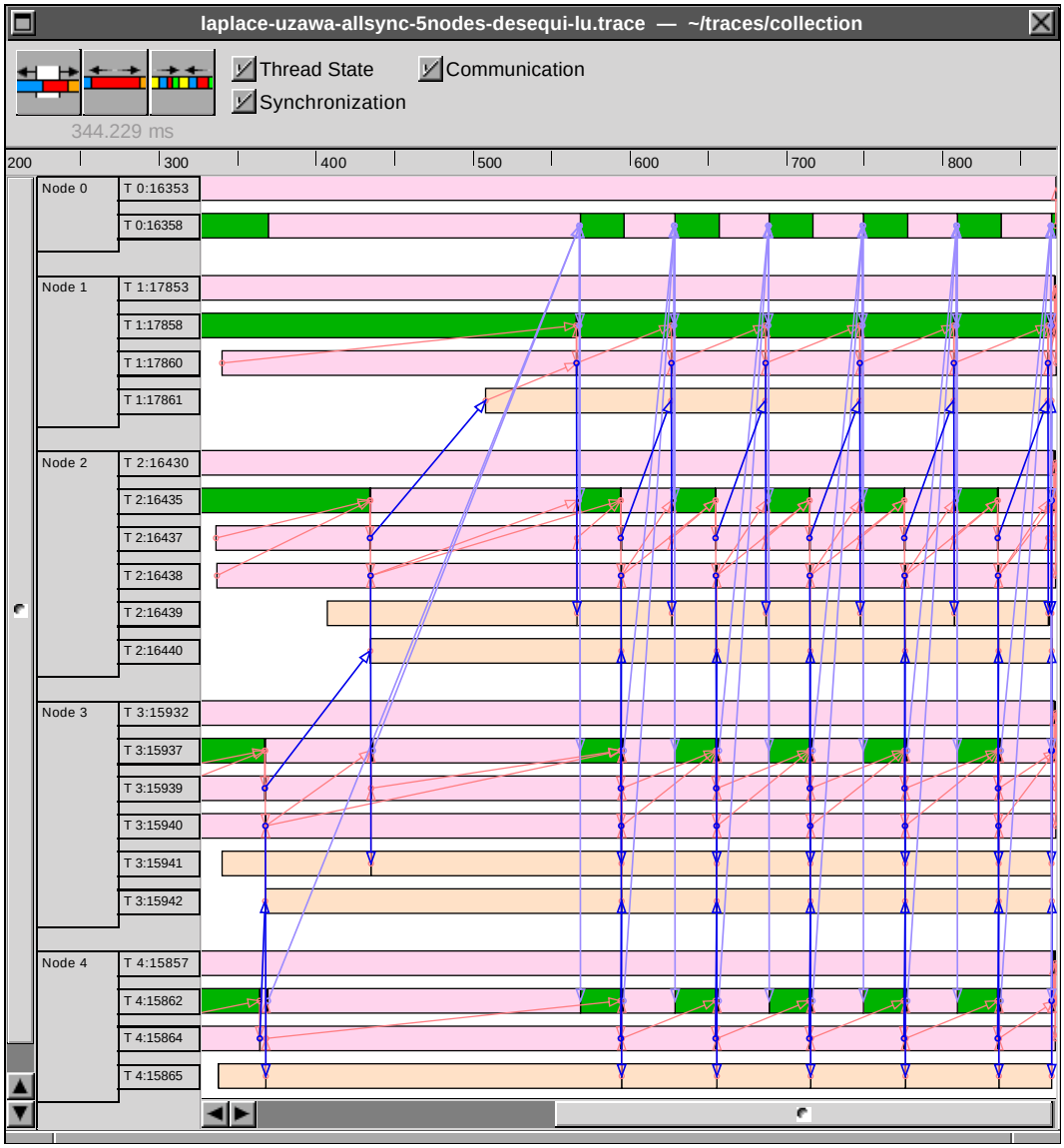

Figure 5.12 *Visualisation d'exécution pour une décomposition en 4 sous-domaines : contrôle de convergence synchrone classique.*

nœud de calcul, le sous-domaine 1 sur le nœud 1 et ainsi de suite. La charge de calcul est légèrement déséquilibrée entre les nœuds, ce qui induit les temps d'attente visibles surtout sur le nœud 4. Sur ce diagramme, on distingue facilement la fin de chacune des 11 itérations : il suffit de repérer les communications globales qui synchronisent tous les nœuds.

Lorsque l'on utilise des communications asynchrones (globales et entre voisins)

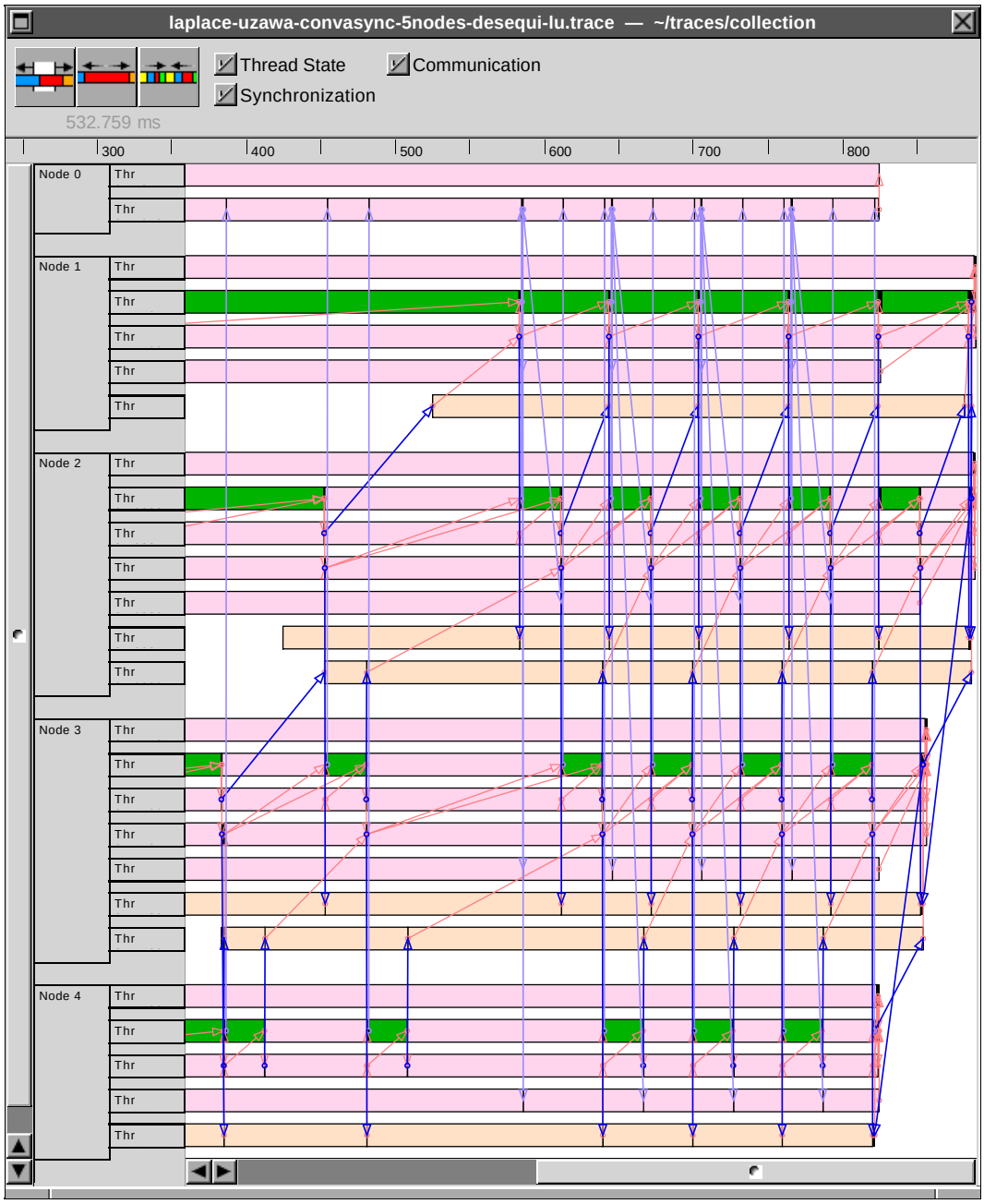

Figure 5.13 *Visualisation d'exécution pour une décomposition en 4 sous-domaines : contrôle de convergence asynchrone.*

dans ce solveur, le diagramme d'exécution parallèle a la forme présentée dans la figure 5.15. Deux remarques importantes ressortent de cette illustration : premièrement, le processus itératif est moins structuré que celui de la figure 5.14. En effet,

Figure 5.14 *Diagramme illustrant l'exécution de la méthode de Schwarz additif synchrone.*

les itérations "se mélangent" à cause de la désynchronisation des communications, il n'est plus possible de déterminer l'itération courante de manière globale. Deuxièmement, on remarque que les processus légers exécutant les tâches de calculs locaux sont toujours actifs. Ils ne bloquent jamais en attente de données.

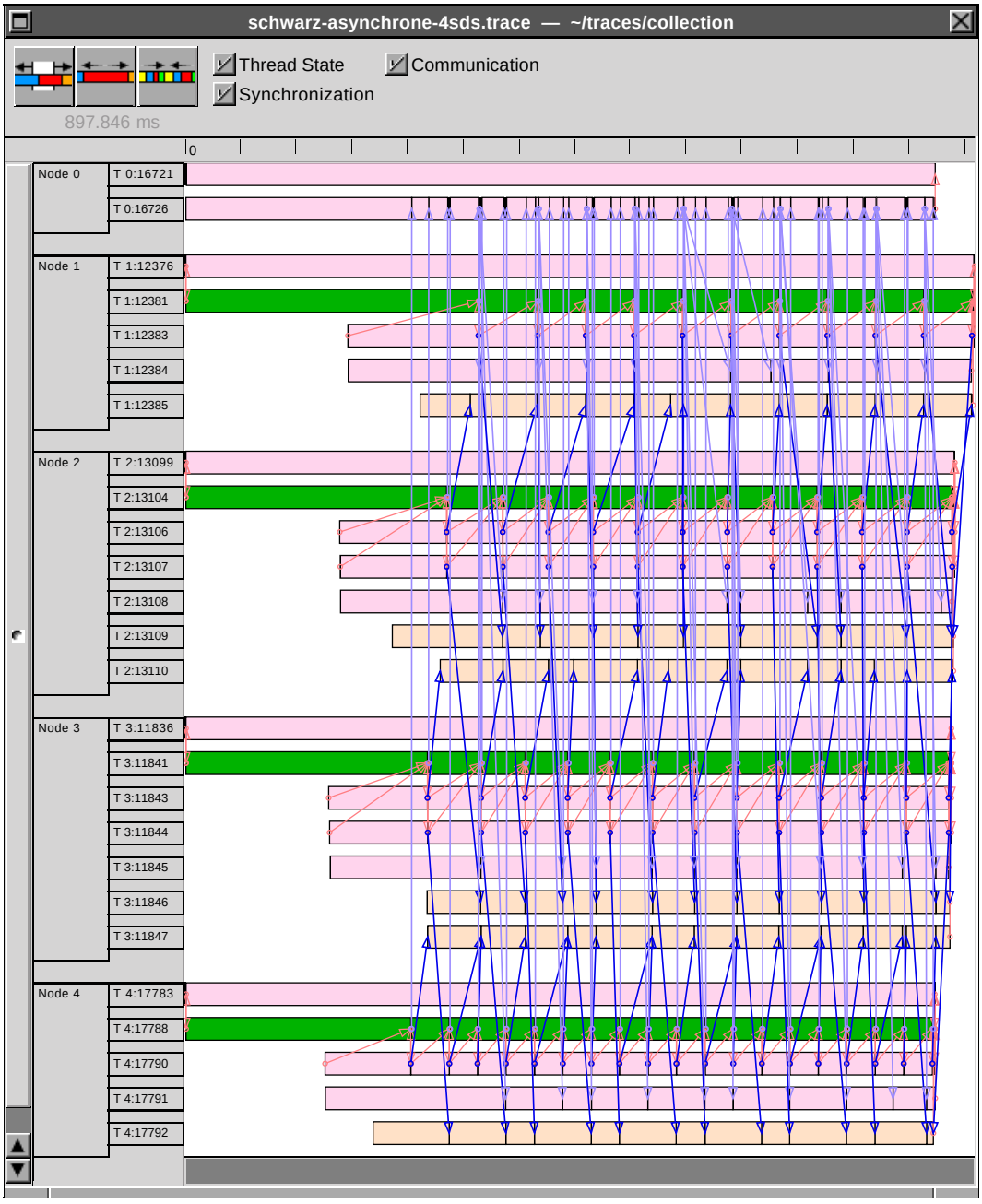

Figure 5.15 *Visualisation d'exécution de la version asynchrone de la méthode de Schwarz additif.*

Multiples sous-domaines par nœud de calcul Un des atouts de AHPIK réside dans la facilité d'affecter plusieurs sous-domaines par nœud de calcul. Cela est illustré dans la figure 5.16 pour la résolution du problème de Poisson sur un

domaine décomposé en 4 sous-domaines. Nous utilisons 2 nœuds pour cette exécution. Chaque nœud traite 2 sous-domaines et il n'y a pas de nœud dédié au calcul de l'erreur globale : le processus léger correspondant partage le processeur avec les processus légers responsables des tâches de calcul locaux, de calcul d'interface, etc. La méthode de décomposition de domaine choisie pour cet exemple est celle de Schur dual (algorithme d'Uzawa), avec une résolution globale par méthode de point fixe.

Les sous-domaines ayant des tailles différentes, on effectue un placement de manière à équilibrer la charge sur les 2 nœuds. On remarque sur la figure 5.16 que les processeurs sont bien utilisés : à un instant donné, il y a pratiquement toujours un processus léger actif pour les tâches de calculs locaux. Ces calculs sont déclenchés automatiquement avec la disponibilité des données.

5.4 Les méthodes de décomposition de domaine implantées

Dans ce paragraphe, nous nous attachons à montrer les calculs (locaux et d'interface) caractérisant chaque méthode de décomposition de domaine implantée à présent dans AHPIK.

Tous les solveurs comprennent une procédure d'initialisation qui construit les structures de données spécifiques à chaque méthode.

5.4.1 Schwarz additif synchrone

Pour notre mise en œuvre de la méthode de Schwarz additif, la systématique de découpage en tâches du chapitre 4 nous donne, pour chaque sous-domaine Ω_k :

– une tâche de calculs locaux T^k_Ω qui effectue la résolution du système linéaire issu de la discrétisation par différences finies du problème d'EDP posé dans ce sous-domaine. À chaque itération $n = 1, 2, ...$, il s'agit de résoudre

$$A_{kk} U^n_k = F^{n-1}_k.$$

Le second membre F_k prend en compte les conditions aux limites, il est mis à jour à chaque itération de la méthode de décomposition de domaine. La tâche T^k_Ω de l'itération n nécessite les valeurs du second membre F_k à l'itération $n-1$;

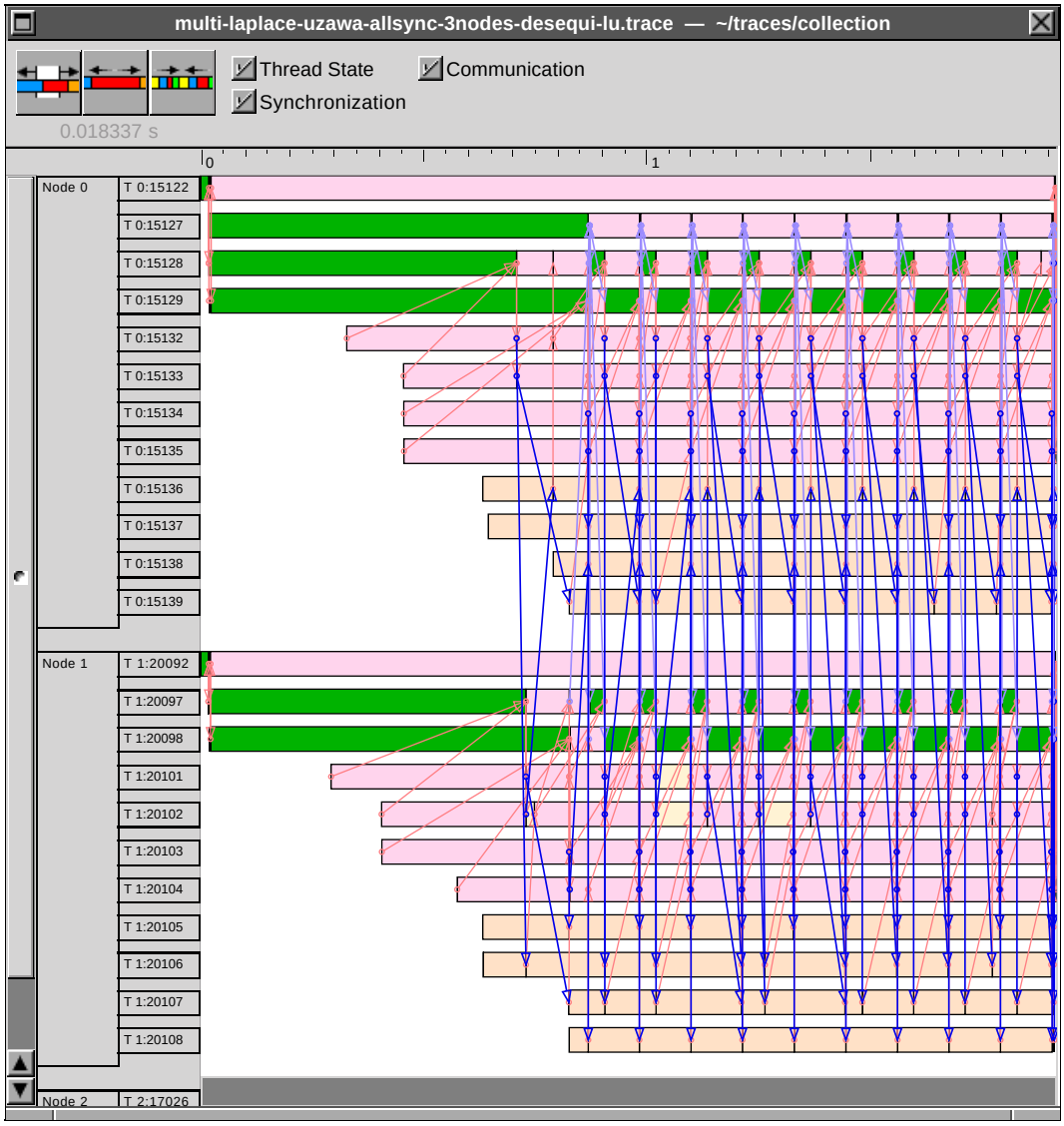

Figure 5.16 *Diagramme d'exécution avec deux sous-domaines par nœud de calcul.*

– un ensemble de tâches T_γ^{kl}, en nombre égal au nombre de sous-domaines ayant une interface commune avec Ω_k. Ces tâches extraient l'information nécessaire pour mettre à jour les seconds membres pour le calcul du nouvel itéré dans les sous-domaines Ω_l voisins de Ω_k. Il s'agit des valeurs de la solution U_k^n calculée dans le sous-domaine Ω_k restreintes à l'interface γ_l^k. Il est donc nécessaire de distinguer les éléments de U_k et de F_k correspondant aux frontières de recouvrement, comme illustré dans la figure 5.17. Dans ce dessin nous avons utilisé une numérotation locale cartésienne des nœuds de chaque sous-domaine. Pour plus de lisibilité, seuls quelques numéros sont indiqués.

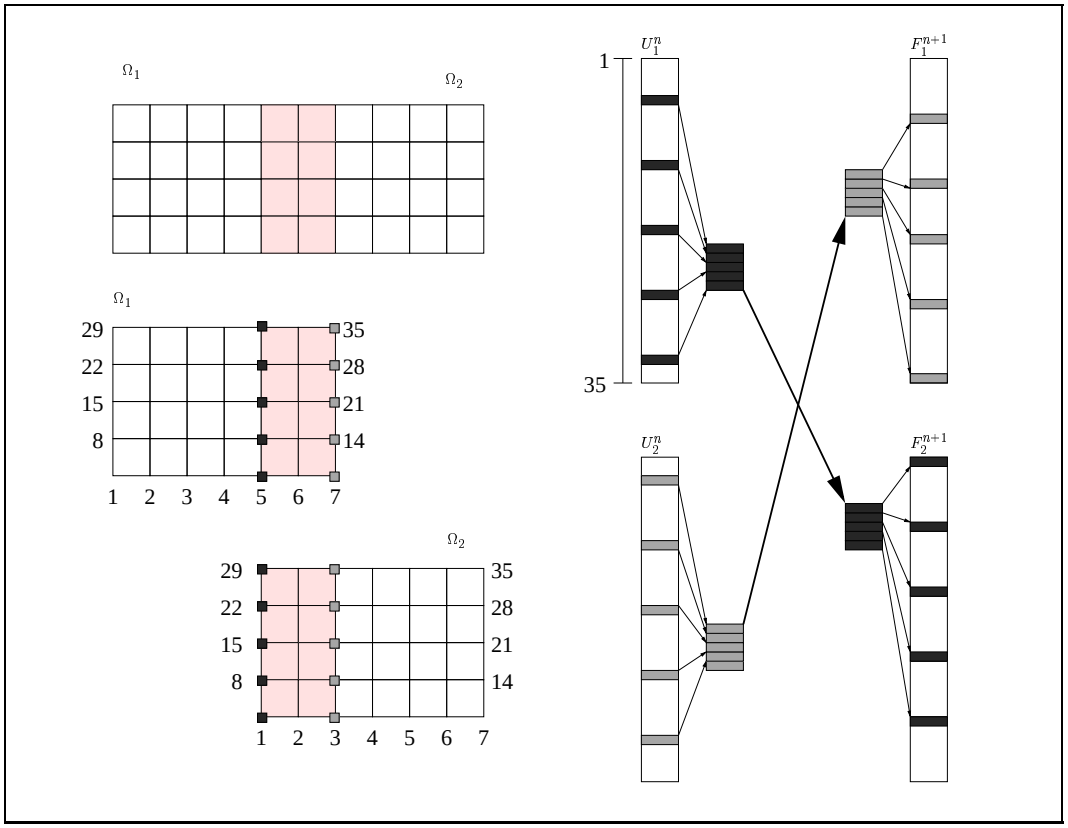

Figure 5.17 *Communications pour la méthode de Schwarz : distinction des nœuds correspondant aux frontières de recouvrement.*

Dans AHPIK, les calculs spécifiques à ce solveur sont codés dans la classe `SchwarzSolver`. Elle définit entre autre :

- des fonctions permettant d'identifier les valeurs du vecteur solution correspondant aux nœuds de la frontière de recouvrement interne, ainsi que les valeurs du vecteur du second membre correspondant aux nœuds de la frontière de recouvrement externe ;
- les calculs correspondant aux tâches T_Ω et T_γ (fonctions `LocalComputation` et `InterfaceComputation`).

5.4.2 Schwarz additif asynchrone

La classe `SchwarzSolver` contenant les calculs spécifiques à la méthode de Schwarz est indépendante du caractère synchrone ou asynchrone de la résolution. Le changement de comportement de la méthode résulte du choix du schéma parallèle avec communications asynchrones entre sous-domaines voisins.

5.4.3 Schur primal, méthode de point fixe

Dans la méthode du complément de Schur primal, les résolutions des systèmes locaux sont couplées par la résolution du système d'interface

$$SU_{\mathcal{I}} = G, \tag{5.1}$$

où $U_{\mathcal{I}}$ est le vecteur solution du système d'interface,

$$S = A_{\mathcal{I}\mathcal{I}} - \sum_{k=1}^{K} A_{\mathcal{I}k} A_{kk}^{-1} A_{k\mathcal{I}} \tag{5.2}$$

et

$$G = F_{\mathcal{I}} - \sum_{k=1}^{K} A_{\mathcal{I}k} A_{kk}^{-1} F_k. \tag{5.3}$$

Le sous-bloc $A_{\mathcal{I}k}$ correspond à la partie de la matrice de rigidité qui gère la relation entre les nœuds internes au sous-domaine Ω_k et les nœuds situés sur les parties de frontière de Ω_k communes à un autre sous-domaine (ce sont les nœuds appartenant aux interfaces γ_{kl} entre Ω_k et ses sous-domaines voisins Ω_l).

Informatiquement, le vecteur $U_{\mathcal{I}}$ est une collection de vecteurs de solution, chacun correspondant à une interface γ_{kl} entre deux sous-domaines Ω_k et Ω_l. Lorsque les interfaces n'ont pas de point en commun, ces vecteurs sont disjoints.

Nous avons mis en œuvre un solveur qui résout ce système itérativement à l'aide d'une méthode de point fixe avec préconditionnement de la matrice S (voir paragraphe 2.3.3.1). Le préconditionneur utilisé, noté $\tilde{S}$, est donné par

$$\tilde{S} = A_{\mathcal{I}\mathcal{I}} - \sum_{k=1}^{K} A_{\mathcal{I}k} (diag((A_{kk}))^{-1} A_{k\mathcal{I}},$$

où $diag(A)$ est une matrice diagonale dont la diagonale est celle de A.

Pour ce solveur nous avons identifié, pour chaque sous-domaine Ω_k :

– une tâche de calculs locaux T_Ω^k qui effectue la résolution du système linéaire :

$$A_{kk} U_k^n = F_k - A_{k\mathcal{I}} U_{\mathcal{I}}^{n-1}.$$

La tâche T_Ω^k a besoin du vecteur $U_{\mathcal{I}}$ solution du système d'interface à l'itération $n-1$. Lorsque les interfaces n'ont pas de point en commun, cela revient

à obtenir, pour chaque sous-domaine Ω_l voisin de Ω_k, le vecteur de solution U_l (solution restreinte aux nœuds situés aux interfaces γ_{kl}) à l'itération $n-1$. Cette tâche fournit le vecteur solution U_k^n correspondant aux nœuds internes, nécessaire aux calculs d'interface ;

– une tâche de calcul d'interface $T_\gamma^{\mathcal{I}}$. Cette tâche a besoin du vecteur $U_{\mathcal{I}}^{n-1}$, ainsi que des contributions des K sous-domaines au calcul d'interface (il s'agit des produits $A_{\mathcal{I}k}U_k^n$). Elle résout le système

$$\tilde{S}U_{\mathcal{I}}^n = F_{\mathcal{I}} - A_{\mathcal{I}\mathcal{I}}U_{\mathcal{I}}^{n-1} - \sum_{k=1}^{K} A_{\mathcal{I}k}U_k^n + \tilde{S}U_{\mathcal{I}}^{n-1}$$

Dans le cas où les interfaces sont totalement disjointes, cette tâche se décompose en un ensemble de tâches T_γ^{kl}. Chacune d'entre elles est associée à l'interface entre Ω_k et l'un des sous-domaines voisins Ω_l.

Les calculs spécifiques à ce solveur sont codés dans la classe `SchurFPSolver` disponible dans AHPIK. Cette classe définit entre autre :

– des fonctions de rénumérotation des systèmes locaux, chargées de séparer les nœuds internes des nœuds d'interface ;
– les calculs correspondant aux tâches T_Ω et T_γ (fonctions `LocalComputation` et `InterfaceComputation`).

5.4.4 Schur dual, méthode de point fixe

Mathématiquement, il s'agit d'appliquer une méthode de point fixe pour la résolution du système d'interface

$$S\Lambda = G,$$

où nous avons, pour une décomposition en K sous-domaines :

$$S = \sum_{k=1}^{K} B_k A_{kk}^{-1} B_k^T,$$

et

$$G = \sum_{k=1}^{K} B_k A_{kk}^{-1} F_k.$$

La méthode de Schur dual permet de résoudre séparément les problèmes sur les interfaces. Nous avons donc, pour chaque interface $\mathcal{I} = \gamma_{kl} = \gamma_{lk}$:

$$S_{\mathcal{I}}\Lambda_{\mathcal{I}} = G_{\mathcal{I}},$$

où

$$S_{\mathcal{I}} = \sum_{k=1}^{K} B_{\mathcal{I}} A_{kk}^{-1} B_{\mathcal{I}}^{T},$$

et

$$G_{\mathcal{I}} = \sum_{k=1}^{K} B_{\mathcal{I}} A_{kk}^{-1} F_k.$$

Chaque système d'interface est résolu par l'algorithme d'Uzawa, présenté dans le paragraphe 2.3.3.2. Avec la méthode systématique de découpage en tâches du chapitre 4 on extrait, pour chaque sous-domaine Ω_k :

– une tâche de calculs locaux T_Ω^k qui effectue la résolution du système linéaire suivant :

$$A_{kk}U_k^n = F_k - \sum_{\mathcal{I}=1}^{N_i} B_{\mathcal{I}}^T \Lambda_{\mathcal{I}}^n$$

La tâche T_Ω^k a besoin des valeurs de $\Lambda_{\mathcal{I}}$ obtenues à l'itération $n-1$. Elle fournit le vecteur U_k^n, nécessaire au calcul d'une interface.

– les tâches de calcul d'interface T_γ^{kl}. Elles ont besoin des valeurs de la solution U_k^n et U_l^n pour calculer :

$$\Lambda_{\mathcal{I}}^{n+1} = \Lambda_{\mathcal{I}}^n + \rho(B_{\mathcal{I}(k)}U_k^n - B_{\mathcal{I}(l)}U_l^n).$$

La contribution d'une tâche T_Ω^k pour le calcul d'une interface $\mathcal{I}$ est donc $U_k^n B_{\mathcal{I}(k)}$. Les calculs d'interface sont les mêmes des deux côtés de l'interface. Les tâches de calculs d'interface T_γ^{kl} ont besoin des valeurs de la solution U_k^n et U_l^n.

Au contraire de la méthode de Schur primal, le système n'est pas numéroté de façon à séparer les nœuds internes des nœuds d'interface.

Dans AHPIK, les calculs spécifiques à cette méthode se trouvent dans la classe `UzawaFPSolver`. Cette classe définit :

– les fonctions construisant les matrices B pour chaque interface
– les calculs correspondant aux tâches T_Ω et T_γ (fonctions `LocalCOmputation` et `InterfaceComputation`) conformément à ce que nous venons de présenter.

Diagramme d'exécution Afin d'illustrer le comportement parallèle d'un solveur basé sur une méthode de point fixe, nous fournissons dans la figure 5.18 un diagramme d'exécution avec « zoom » sur quelques itérations de la méthode de décomposition de domaine. Il s'agit de la résolution d'un problème de très petite taille, où les temps de communication sont très visibles, dépassant même le temps du calcul d'interface. On distingue à la fin de chaque itération l'échange d'un message avec le processus léger "coordinateur", nécessaire pour le contrôle de convergence global.

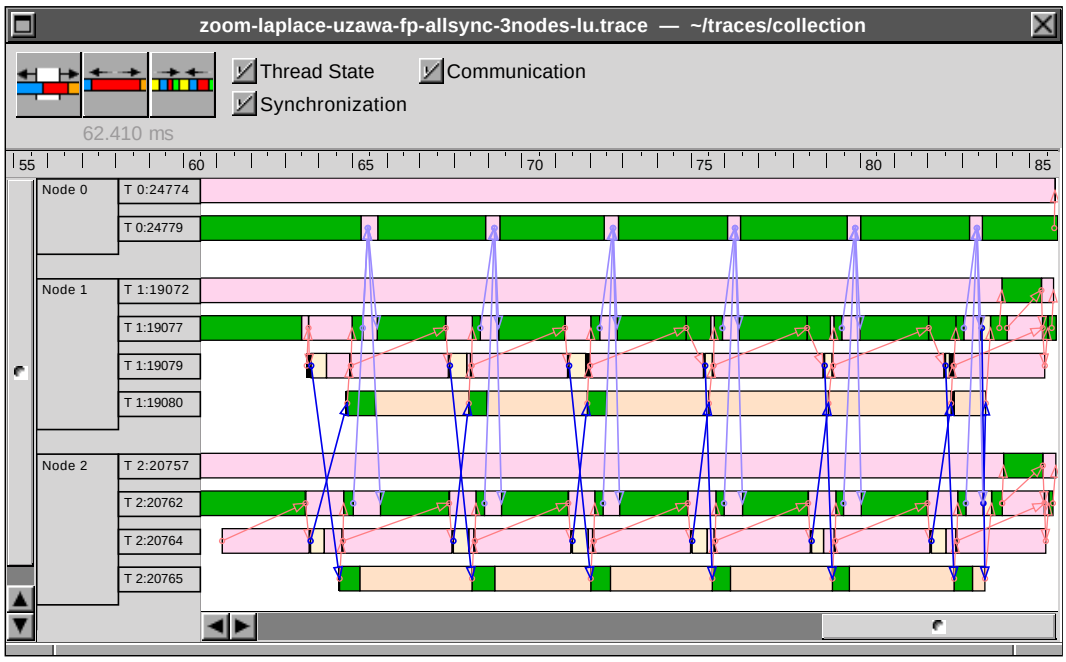

Figure 5.18 *Schur dual, méthode point fixe : « zoom » sur les itérations.*

5.4.5 Schur dual, méthode du gradient conjugué

Dans AHPIK, la classe `UzawaCGSolver` réunit les calculs spécifiques à l'algorithme d'Uzawa appliqué à la méthode de Schur dual avec une résolution par gradient conjugué sur les interfaces. En plus des structures de données propres à la méthode de résolution, cette classe définit les fonctions qui remplissent le schéma du gradient conjugué parallèle présenté au paragraphe 5.3.1.2. Il s'agit des calculs effectués par les tâches T_Ω et T_γ associées à sous-domaine Ω_k, suivant l'algorithme 9 présenté à l'annexe D.

Pour cette méthode, la méthode systématique de découpage en tâches du cha-

pitre 4 nous donne :

– une tâche de calculs locaux T_Ω^k à l'initialisation (`LocalComputation1`) :

$$A_{kk}U_k = F_k - \sum_{\mathcal{I}=1}^{N_i} B_\mathcal{I}\Lambda_\mathcal{I}^0$$

La contribution pour le calcul de l'interface $\mathcal{I}$ est

$$\xi_{\mathcal{I}(k)} = B_\mathcal{I}U_k$$

– une tâche de calculs d'interface T_γ^{kl} (`InterfaceComputation1`) à l'initialisation. Cette tâche retourne le résidu à l'interface $\mathcal{I}$, donné par

$$r_\mathcal{I} = \xi_{\mathcal{I}(k)} + \xi_{\mathcal{I}(l)}$$

– une tâche de calculs locaux T_Ω^k (`LocalComputation2a`) pour l'itération i. Elle a besoin du résidu sur toutes les interfaces entre le sous-domaine Ω_k et ses sous-domaines voisins Ω_l pour résoudre le système :

$$A_{kk}z_k^i = \sum_{\mathcal{I}=1}^{N_i} B_\mathcal{I}^T p_\mathcal{I}^i,$$

fournissant le vecteur de variables intermédiaires z_k pour le calcul du pas de descente. La contribution pour le calcul de l'interface $\mathcal{I}$ est

$$\xi_{\mathcal{I}(k)} = B_\mathcal{I}z_k$$

– une tâche de calculs d'interface T_γ^{kl} (`InterfaceComputation2a`) pour l'itération i. Elle calcule la contribution de l'interface $\mathcal{I}$ au calcul du pas de descente α^i :

$$\psi = \xi_{\mathcal{I}(k)} + \xi_{\mathcal{I}(l)}$$

– une deuxième tâche de calculs locaux T_Ω^k (`LocalComputation2b`) pour l'itération i . Elle calcule les solutions locales à l'itération $i+1$:

$$X_k^{i+1} = X_k^i + \alpha^i z_k$$

– une deuxième tâche de calculs d'interface T_γ^{kl} (`InterfaceComputation2b`) pour l'itération i . Elle calcule la direction de descente correspondant à l'interface $\mathcal{I}$:

$$X_\mathcal{I}^{i+1} = X_\mathcal{I}^i + \alpha^i p_\mathcal{I}^i$$

– une troisième tâche de calculs d'interface T_γ^{kl} (`InterfaceComputation2c`) pour l'itération i . Elle calcule le résidu correspondant à l'interface $\mathcal{I}$:

$$r_{\mathcal{I}}^{i+1} = r_{\mathcal{I}}^{i} + \alpha^i \psi_{\mathcal{I}}$$

On remarque que le nombre de tâches pour l'algorithme du gradient conjugué est supérieur à celui d'une méthode de point fixe. Toutefois, à chaque instant il n'y qu'une seule tâche de calcul local active par sous-domaine. Ces tâches sont donc exécutés par le même processus léger. Il en est de même pour ce qui concerne les tâches de calcul d'interface : les tâches associées au même sous-domaine s'exécutent en parallèle, mais il n'y a qu'une seule tâche active par interface.

Diagramme d'exécution La figure 5.19 fournit un diagramme d'exécution avec « zoom » sur quelques itérations de la méthode de décomposition de domaine. Ce diagramme illustre le comportement parallèle caractéristique d'une résolution couplée par la méthode du gradient conjugué. On remarque que ce schéma de résolution nécessite plus de communications qu'une méthode de point fixe. Il se caractérise par deux points de synchronisation globale à la fin de chaque itération.

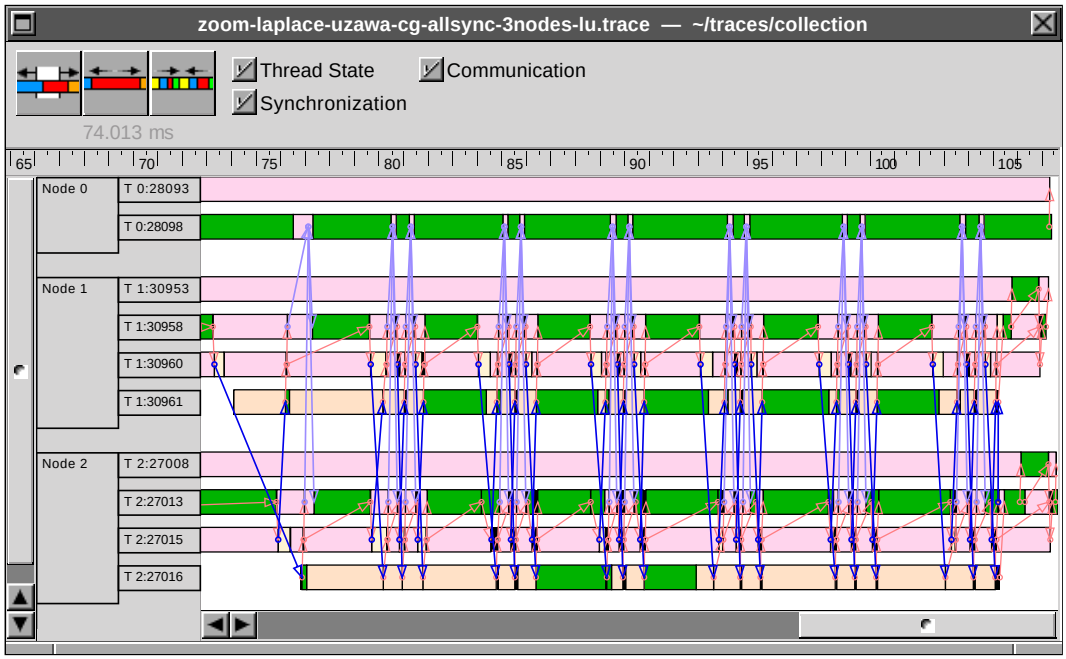

Figure 5.19 *Schur dual, méthode du gradient conjugué : « zoom » sur les itérations.*

5.4.6 Méthode des éléments joints

Les calculs spécifiques à la méthode des éléments joints diffèrent peu de ceux que nous venons de décrire pour la méthode de Schur dual. La principales différences viennent de la construction des matrices d'interface ainsi que des échanges de données asymétriques provenant de la méthode numérique qui distingue, pour chaque interface, le côté (sous-domaine) maître du côté esclave. Nous donnons plus de précisions sur la mise en œuvre de cette méthode dans le paragraphe 6.3.

Troisième partie

Expérimentation

6

Études expérimentales de problèmes d'EDP classiques

Sommaire

6.1 Étude expérimentale du comportement de convergence des méthodes numériques 142
6.1.1 Rapport d'aspect et décomposition géométrique 143
6.1.2 « Scalabilité » de Schur dual 145
6.2 Déséquilibres de charge et recouvrement : expériences avec un problème de convection-diffusion 146
6.3 Aspects de la multiprogrammation d'une application réelle . 149
6.3.1 Un problème d'écoulement de fluide 149
6.3.2 Description du code original 151
6.3.3 Portage de l'application avec AHPIK 154
6.3.4 Résultats numériques 156
6.4 Conclusion 157

Un des résultats les plus importants de ce travail est l'écriture d'une bibliothèque flexible pour la mise en œuvre de méthodes de décomposition de domaine pour la résolution en parallèle de problèmes d'EDP de grande taille. Ce chapitre discute les performances de la bibliothèque AHPIK pour plusieurs méthodes de décomposition de domaine. Nous avons décidé de présenter les résultats de manière progressive sur des problèmes dont la complexité mathématique va croissante.

Les premières expériences ont pour but de montrer le comportement des méthodes tant d'un point de vue de la convergence mathématique que d'un point de vue de l'efficacité parallèle des algorithmes. Pour réaliser ces expériences, nous avons choisi le problème le plus classique en EDP : le problème de Poisson décrit dans la section 2.1.1.

Dans un deuxième temps, le comportement inhomogène des méthodes de résolution locale itératives est mis en évidence par l'intermédiaire de la résolution d'un problème de convection-diffusion. Il apparaît notamment que le nombre d'itérations nécessaires pour la convergence de la résolution locale par une méthode de type gradient conjugué est variable, ce qui introduit de légers déséquilibres de charge.

Suite à ces problèmes académiques nous avons considéré le portage d'un code parallèle implémenté à l'aide de la bibliothèque MPI, et résolvant les équations de Navier-Stokes. La discrétisation du problème requiert environ 250 000 degrés de liberté répartis, après décomposition, en 92 sous-domaines, et affectés sur 22 processeurs. Pendant le portage, il est apparu que ce code est peu flexible car conçu pour cette unique application. De taille réelle, celle-ci constitue cependant un test pertinent pour évaluer l'intérêt de AHPIK.

6.1 Étude expérimentale du comportement de convergence des méthodes numériques

Les méthodes de décomposition de domaine ont un comportement de convergence très dépendant des considérations géométriques relatives à la décomposition du domaine. Il a été prouvé que le nombre de sous-domaines et leur rapport d'aspect ont une forte influence sur la vitesse de convergence des méthodes[64].

Bien que les résultats présentés dans ce paragraphe soient connus des utilisateurs et développeurs de méthodes de décomposition de domaines, il nous a semblé pertinent de les rappeler afin de mettre en évidence le dilemme concernant le nombre de sous-domaines. En effet, on a potentiellement plus de parallélisme lorsque le nombre de sous-domaines est grand, mais on perd en efficacité car ces méthodes

mathématiques sont itératives et impliquent un surcroît de calculs dans ce cas.

Pour illustrer nos propos, nous avons choisi le problème de Poisson bidimensionnel décrit dans la section 2.1.1. Notre but n'étant pas de comparer les différentes méthodes numériques, nos résultats sont uniquement basés sur la méthode de Schur dual. Nous l'avons choisi car sa simplicité au niveau du traitement des interfaces permet de manipuler facilement différentes décompositions géométriques. Les découpes en sous-domaines seront précisées au fur et à mesure.

6.1.1 Rapport d'aspect et décomposition géométrique

Pour les domaines bidimensionnels, le rapport d'aspect d'un sous-domaine est caractérisé par le rapport entre sa longueur et sa largeur. Pour une discrétisation régulière, un rapport d'aspect grand implique potentiellement des interfaces comportant plus de nœuds. Nous allons étudier son rôle dans la convergence des méthodes par l'intermédiaire de deux décompositions classiques. Le domaine de calcul est le carré unitaire $\Omega =]0,1[\times]0,1[$ discrétisé à l'aide de triangles définis sur une grille cartésienne comportant 257 points équidistribués dans chacune des directions. Au final, le probléme comporte 66 049 inconnues. Pour ce même domaine, nous considérons :

- une décomposition en ν bandes rectangulaires d'égale largeur. Le rapport d'aspect de ces rectangles est égal à ν comme montré dans la figure 6.1(a) ;
- une décomposition en rectangles dont le rapport d'aspect est égal à 1 ou 2 (figure 6.1(b)).

La résolution du problème d'interface est réalisé par la méthode de point fixe, les problèmes locaux sont résolus par une factorisation LU. Le tableau 6.1 présente le nombre d'itérations nécessaires pour la convergence du solveur Schur dual dans chacune des décompositions considérées, ainsi que les temps de restitution pour la résolution du problème **en séquentiel** sur chaque décomposition (il s'agit d'une version séquentielle du solveur de décomposition de domaine).

On observe que le nombre d'itérations pour la convergence croît effectivement avec l'augmentation du nombre de sous-domaines, et cet effet est plus négatif pour la décomposition en bandes. Cependant, le solveur local choisi est plus rapide, dans le cas considéré, lors de la résolution des systèmes issus de la décomposition en bandes. Pour ces expériences, nous avons utilisé un critère d'arrêt égal à 10^{-4}, ce qui explique en partie le petit écart entre le nombre d'itérations pour les décompositions en 8 et 16 sous-domaines.

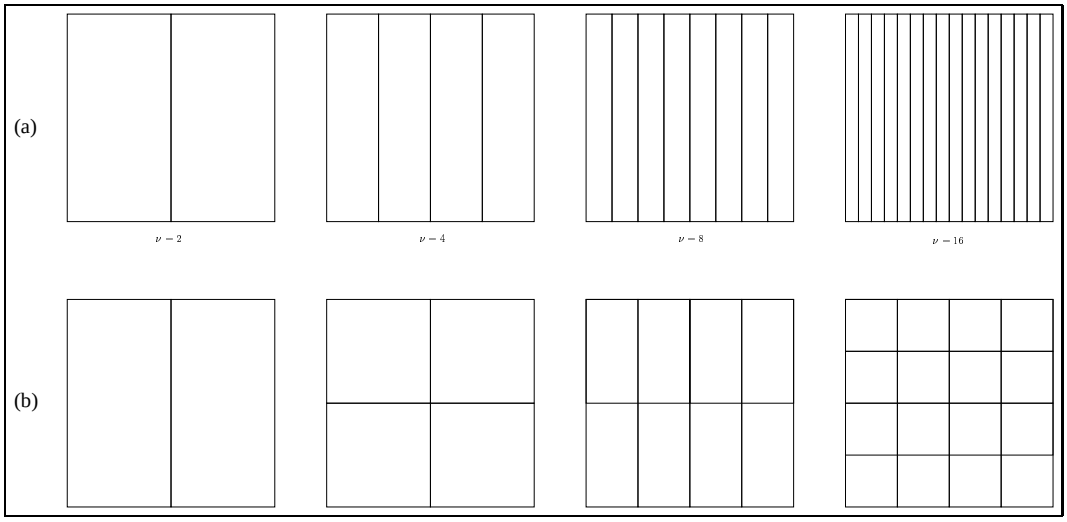

Figure 6.1 *Décompositions en (a) bandes verticales, rapport d'aspect égal à ν et (b) blocs rectangulaires, rapport d'aspect égal à 1 ou 2.*

Nombre de sous-domaines	Nombre d'itérations		Temps de restitution (s)	
	bandes	blocs	bandes	blocs
2	2	2	82.1	82.1
4	19	2	109.2	82.7
8	22	19	59.2	107.8
16	25	21	43.1	107.4

Tableau 6.1 *Nombre d'itérations pour la convergence et temps de restitution pour la méthode d'Uzawa en séquentiel sur les décompositions en bandes et en blocs.*

Nombre de processeurs	Temps de restitution (s)		Accélérations	
	bandes	blocs	bandes	blocs
2	41.3	41.3	1.98	1.98
4	46.9	20.7	3.89	3.99
8	12.8	23.1	7.83	7.84
16	4.7	12.8	15.06	15.39

Tableau 6.2 *Temps de restitution parallèle pour les décompositions en bandes et en blocs.*

L'influence du comportement de convergence sur les efficacités parallèles est illustrée dans le tableau 6.2. Les temps de restitution parallèle fournis dans ce ta-

bleau sont obtenus en affectant un sous-domaine par processeur. Les accélérations sont calculées par rapport à la version séquentielle du solveur Uzawa (point fixe) pour chacune des décompositions (cf. tableau 6.1). On observe que, dans ce cas où la charge de calcul est parfaitement équilibrée, les accélérations obtenues sont très satisfaisantes.

Les résultats que nous venons de présenter ne se prêtent évidemment pas à une généralisation, cependant ils mettent en évidence que le compromis entre la vitesse de convergence et le maximum de parallélisme potentiel dépend de plusieurs facteurs et n'est pas toujours facile à prédire. Dans ce sens, les voies de recherche pour obtenir de bonnes efficacités lors de l'utilisation d'une méthode de décomposition de domaine sont multiples : certains auteurs s'intéressent aux heuristiques permettant d'optimiser le rapport d'aspect lors du partitionnement du maillage (voir par exemple [161]), d'autres travaillent sur des méthodes numériques hybrides permettant d'accélérer la convergence de la méthode de décomposition de domaine (voir par exemple [140] qui présente un exemple de solveur associant les méthodes de décomposition de domaine aux méthodes multiniveaux algébriques). Il est donc important de disposer d'outils informatiques flexibles permettant une mise en œuvre parallèle rapide de nouvelles méthodes numériques.

6.1.2 « Scalabilité » de Schur dual

Pour évaluer la « scalabilité » de la méthode de Schur dual, le domaine de calcul est construit par juxtaposition d'autant de répliques du carré unitaire que de processeurs utilisés pour l'expérience. Nous considérons deux tailles différentes pour la grille représentant le carré, avec les nombre de nœuds respectivement égaux à 33×33 et 65×65. Les décompositions sont formées de P carrés alignés, où P correspond au nombre de processeurs utilisé pour chaque exécution. La taille du problème croît donc avec P. Comme dans le paragraphe précédent, on affecte un sous-domaine par processeur. Pour un cas parfaitement « scalable », les temps d'exécution doivent rester constants lorsque l'on augmente le nombre de processeurs. Le tableau 6.3 regroupe les résultats obtenus pour cette expérience.

On observe sur le tableau 6.3 que l'augmentation du nombre de processeurs a un effet plus négatif lorsque les sous-domaines sont de taille plus petite. En effet, dans ce cas le coût des communications devient important par rapport au coût des calculs. Lorsque les sous-domaines sont de taille plus grande, les communications sont moins contraignantes car le temps passé à échanger des messages est petit par rapport aux temps de calcul. Pour ces problèmes, il faut chercher à éviter l'inactivité des processeurs ainsi qu'à obtenir une utilisation efficace de ceux-ci.

Nombre de processeurs	Temps de restitution (s)	
	33×33	65×65
2	0.59	5.12
4	0.61	5.15
8	0.67	5.20
16	0.77	5.33
24	0.86	5.42
32	0.97	5.52
48	1.28	5.69
64	3.44	5.88

Tableau 6.3 *Temps de restitution, problèmes de taille croissante.*

6.2 Déséquilibres de charge et recouvrement : expériences avec un problème de convection-diffusion

Les équations de convection-diffusion sont une version linéarisée des équations de Navier-Stokes. À ce titre, elles sont utilisées dans de nombreux problèmes de mécanique des fluides. Parmi eux figurent des problèmes d'environnement tels que le transport et diffusion de polluants dans l'atmosphère ou l'océan, de sédiments dans les fleuves, etc. La discrétisation de ces problèmes, souvent tridimensionnels, s'effectue dans un espace de grande dimension, ce qui justifie l'utilisation d'une méthode de décomposition de domaine. Pour simplifier l'exposé nous nous plaçons dans le cadre académique suivant.

En notant $\Omega =]0,1[\times]0,1[$ le domaine de frontière Γ, f une fonction de $L^2(\Omega)$, et α_i ($i = 1,..,2$), $\beta \geq 0$ et $\mu > 0$ des réels, le problème d'EDP modélisant la convection-diffusion s'écrit

$$\begin{cases} \text{Trouver } u \in H_0^1(\Omega) \text{ tel que} \\ -\mu\Delta u + \displaystyle\sum_{i=1}^{2} \alpha_i \frac{\partial u}{\partial x_i} + \beta u = f \text{ dans } \Omega, \\ u = 0 \text{ sur } \Omega. \end{cases} \tag{6.1}$$

La solution discrète de (6.1) est cherchée dans l'espace d'approximation des fonctions éléments finis P^1. La méthode de décomposition de domaine choisie calcule la solution à l'intérieur des sous-domaines à l'aide d'un gradient conjugué préconditionné par une factorization LU incomplète. La résolution aux interfaces est

faite par une méthode de Schur primale. Le domaine de calcul est identique à celui présenté dans le paragraphe précédent (carré unitaire, grille régulière 257x257). Les sous-domaines sont des bandes verticales. La solution initiale est sur une grille grossière. Ces tests ont été effectués sur une machine parallèle IBM-SP2.

Nombre de sous-domaines	Nombre de processeurs			
	2	4	8	16
2	0.81	-	-	-
4	1.22	1.83	-	-
8	1.78	2.77	3.99	-
16	1.86	3.57	6.07	8.01

Tableau 6.4 *Accélérations obtenues pour des différentes décompositions et différents schémas de placement*

L'objectif de cette étude expérimentale est d'analyser le comportement de AHPIK pour des différents nombres de sous-domaines et des différents schémas de placement de ces sous-domaines sur les processeurs disponibles. Nous considérons les décompositions du domaine original en 2, 4, 8 et 16 sous-domaines non-recouvrants de taille identique. Le tableau 6.4 montre les accélérations du solveur parallèle de décomposition de domaine pour des exécutions sur 2 à 16 processeurs. Pour calculer ces accélérations, nous avons fait une version séquentielle du solveur de décomposition de domaine, et nous avons mesuré le temps de restitution pour chaque décomposition. Ensuite, nous avons choisi l'exécution séquentielle la plus rapide (avec 16 sous-domaines) comme référence. En effet , il a été prouvé[64] que la méthode de décomposition de domaine agit comme un préconditionneur susceptible d'accélérer la convergence d'un calcul d'EDP.

Dans le tableau 6.4, on remarque que les accélérations sont peu satisfaisantes quand un seul sous-domaine est affecté à chaque processeur. Pour interpréter ce résultat, il faut se pencher sur le calcul lui même. Pour mettre en évidence le motif de ce comportement, nous avons tracé le nombre I d'itérations nécessaires à la convergence de la méthode de décomposition de domaine et le nombre d'itérations i_k pour la résolution à l'intérieur de chaque sous-domaine $\Omega_k, k = 1, ..., K$. Les courbes de la figure 6.2 montrent que le nombre d'itérations i_k peut varier d'un sous-domaine à l'autre (même s'ils sont de même taille). On observe aussi que ce nombre décroît lorsqu'on se rapproche de la solution. Ces différences sur le nombre d'itérations viennent du choix d'une méthode itérative (gradient conjugué) pour la résolution du problème d'EDP à l'intérieur des sous-domaines. Ce choix d'un solveur itératif induit une incertitude sur la durée de chaque itération. La figure 6.3 trace les variations maximales (en pourcentages) de ces nombres d'itérations.

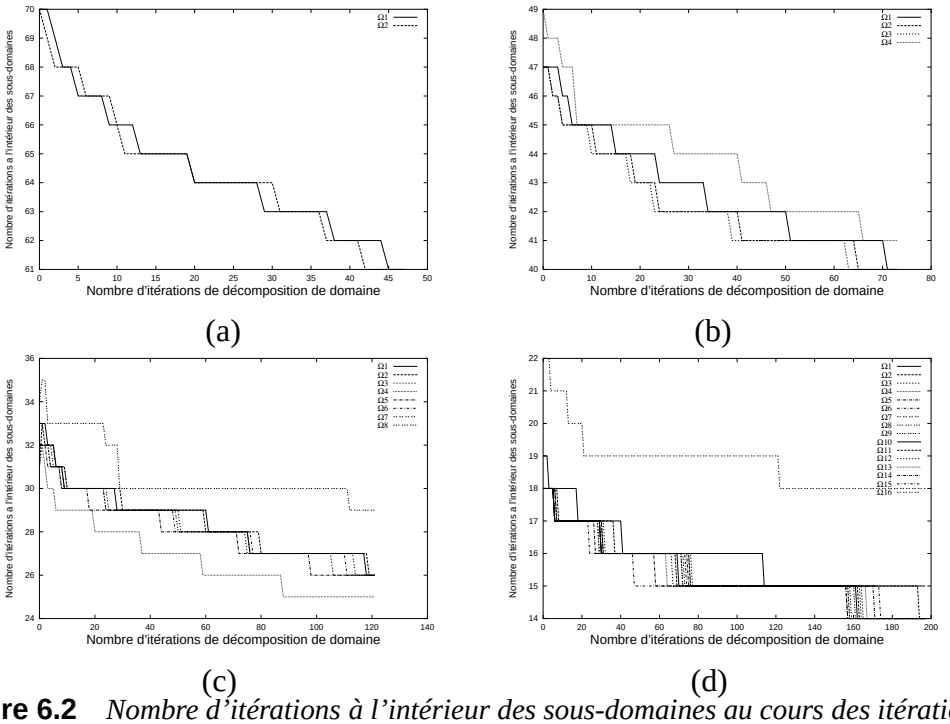

Figure 6.2 *Nombre d'itérations à l'intérieur des sous-domaines au cours des itérations de décomposition de domaine. Les décompositions considérées sont : (a) 2 sous-domaines, (b) 4 sous-domaines, (c) 8 sous-domaines et (d) 16 sous-domaines.*
Les figures sont tracées à une échelle différente pour une meilleure visibilité des courbes.

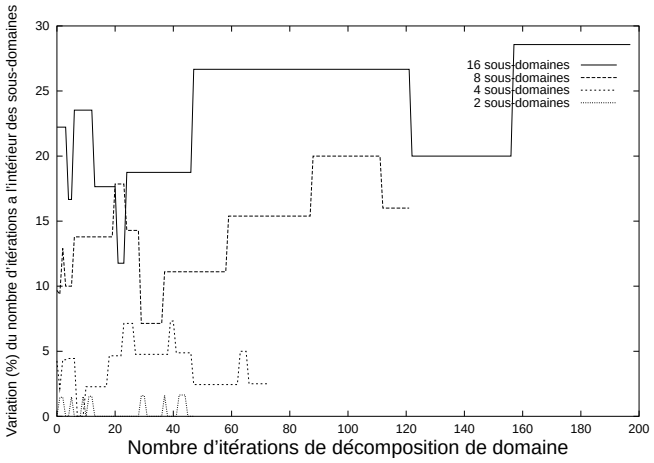

Figure 6.3 *Pourcentages de variation du nombre d'itérations pour la résolution à l'intérieur des sous-domaines*

L'incertitude du nombre d'itérations i_k introduit un déséquilibre de charge (donc une inactivité) entre les nœuds de calcul qui est très visible à cause de la nature synchrone de la méthode de décomposition de domaine. À mesure que nous augmentons le nombre de sous-domaines par processeur et pour un nombre total de sous-domaines fixé, les accélérations s'améliorent puisque les attentes dues au schéma synchrone sont recouvertes par du calcul. Ces résultats nous permettent de vérifier que le surcoût lié à l'utilisation d'un environnement générique s'appuyant sur des processus légers communicants est acceptable. Cette étude expérimentale met aussi en évidence le fait que AHPIK permet un choix aisé du nombre de sous-domaines et un placement flexible de ces sous-domaines sur les processeurs. Cette flexibilité est très utile pour l'implantation de mécanismes de répartition dynamique de charge.

6.3 Aspects de la multiprogrammation d'une application réelle

6.3.1 Un problème d'écoulement de fluide

Le problème que nous avons traité modélise l'écoulement d'un fluide incompressible autour d'un cylindre. Cette simulation est très courante en mécanique des fluides et a fait l'objet de plusieurs « benchmarks » (voir par exemple [146]). L'écoulement est régi par les équations de Navier-Stokes écrites ci-dessous sous forme adimensionnelle :

$$\begin{aligned} \frac{\partial \underline{u}}{\partial t} + \underline{u} \cdot \nabla \underline{u} &= \frac{1}{Re} \Delta \underline{u} - \nabla p, \\ \text{div } \underline{u} &= 0, \end{aligned} \tag{6.2}$$

où $\underline{u} = (u, v)$ et p sont respectivement les champs de vitesse et de pression cherchés. La géométrie du problème ainsi que les conditions aux limites sont spécifiées dans la figure 6.4. Les conditions aux limites de sortie sont

$$p|_{sortie} = 0, \quad \left.\frac{u}{x}\right|_{sortie} = 0, \quad v|_{sortie} = 0. \tag{6.3}$$

La condition d'entrée est donnée par

$$u(0, y) = 6\frac{y(H - y)}{H^2}, \quad v|_{sortie} = 0, \tag{6.4}$$

où $H = 4.1$ est la hauteur adimensionnelle du canal.

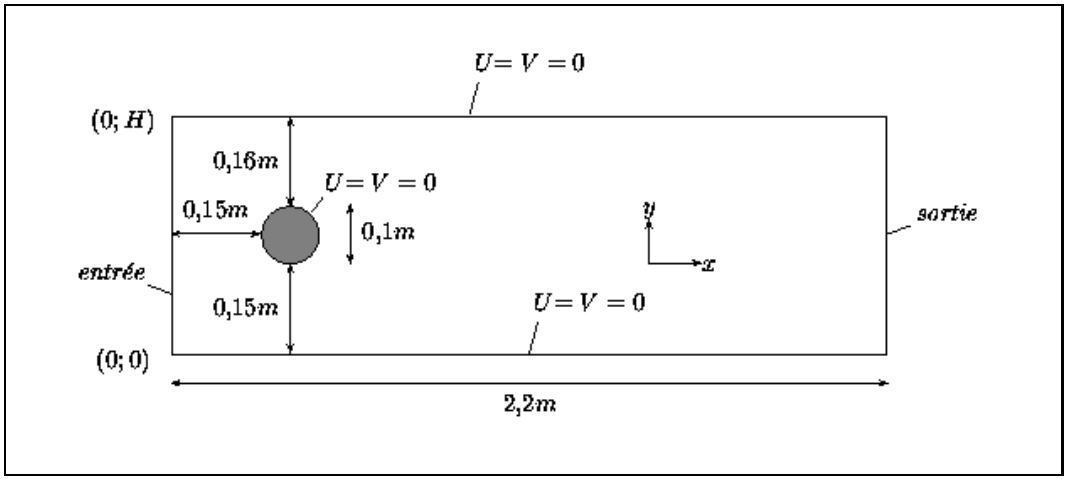

Figure 6.4 *Géométrie et conditions aux limites pour le problème d'écoulement autour du cylindre.*

Le domaine de calcul est subdivisé en 92 sous-domaines non-recouvrants (cf. figure 6.5). La décomposition est géométriquement conforme : l'intersection entre deux sous-domaines Ω_k et Ω_l est soit vide, soit réduite à un sommet ou à un côté. Chaque sous-domaine possède au plus 4 sous-domaines voisins.

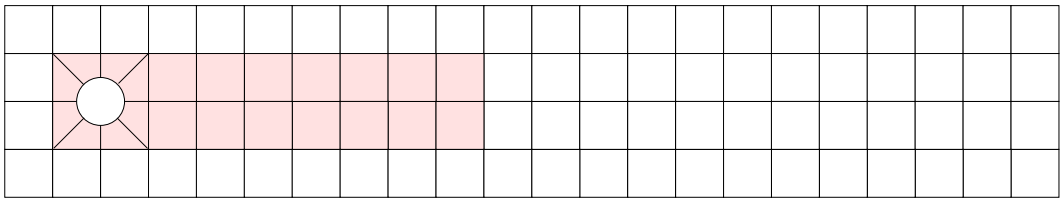

Figure 6.5 *Décomposition de domaine du cylindre.*

Sur chaque sous-domaine, la représentation locale de la vitesse et de la pression s'effectue à l'aide d'éléments finis mixtes P_1-iso-P_2/P_1 (voir Girault-Raviart[83] ou Brezzi[25]). Ces éléments impliquent l'utilisation d'un maillage triangulaire pour l'interpolation de la pression et d'un maillage deux fois plus fin pour le calcul de la vitesse. Le nombre total de nœuds du maillage de vitesse est de 113121 ($\approx$ 2,9 10^4 nœuds de pression). Le maillage est non-conforme : il est plus fin dans les sous-domaines qui entourent le cylindre et en aval de celui-ci (1681 nœuds par sous-domaine, ce sont les sous-domaines hachurés dans la figure 6.5). Il est plus grossier ailleurs (1089 nœuds par sous-domaine).

La simulation consiste à calculer le régime instationnaire établi pour un nombre de Reynolds $Re = 100$.

6.3.2 Description du code original

Le code original est un code parallèle pour la résolution des équations de Navier-Stokes en 2 dimensions. Il est écrit en Fortran 90 et utilise la bibliothèque MPI pour les communications entre processeurs. Le problème est résolu à l'aide de la méthode des éléments joints. Dans la suite, nous nous concentrons sur l'aspect algorithmique de ce code, les précisions concernant les méthodes numériques étant simplifiées autant que possible. Pour plus de détails sur les aspects numériques nous renvoyons à [13].

Avant chaque simulation, un premier programme se charge de préparer les données. Pour chaque sous-domaine on détermine le maillage pour la vitesse (construit à partir de celui de la pression), puis les interfaces du sous-domaine et les indices des sous-domaines voisins. Le résultat est stocké dans des fichiers. Chaque processeur lit les fichiers de données correspondant aux sous-domaines qui lui sont attribués. Il construit ensuite des matrices de rigidité locales, des préconditionneurs et des matrices d'éléments joints (qui vont servir lors des échanges de données des interfaces).

Un des points particuliers de cette application est que viscosité et incompressibilité du fluide sont traitées en deux phases séparées. À chaque itération en temps (δt), le solveur Navier-Stokes résout deux systèmes linéaires pour la vitesse (composantes horizontale et verticale du champ de vitesse) et un système pour la pression. La structure du programme principal est donnée de manière simplifiée par l'algorithme 5.

Algorithme 5 : Structure du programme principal du code Navier-Stokes.

```
lecture des données
initialisations
pour it = 1 jusqu'à Niter faire
   calcul de la composante horizontale de la vitesse
   calcul de la composante verticale de la vitesse
   calcul de la pression
fin pour
```

L'algorithme 5 suggère un ordre séquentiel dans la résolution des trois systèmes linéaires, mais en réalité il n'existe pas de dépendance de données entre les calculs des composantes horizontale et verticale du champ de vitesse. Les systèmes correspondants peuvent donc être calculés en parallèle, les résultats étant ensuite utilisés pour le calcul de la pression. Cette possibilité n'est pas exploitée dans le code original.

Solveur de pression

Le solveur de pression est un solveur entièrement local, c'est-à-dire qu'il ne nécessite pas de communication entre processeurs. Le système posé dans chaque sous-domaine est résolu de manière indépendante à l'aide d'une méthode de gradient conjugué préconditionné par le processeur traitant ce sous-domaine,

Solveur de vitesse

La mise en œuvre du solveur de vitesse est basée sur celle proposée dans l'article originel de la méthode des éléments joints[17] (voir aussi [15]). Ce solveur résout le système matriciel :

$$Q^T AQU = Q^T F. \tag{6.5}$$

La matrice A est une matrice symétrique définie positive. Elle a une structure diagonale par blocs, où chaque bloc A_{kk} correspond à la discrétisation par éléments finis du problème dans le sous-domaine Ω_k. Les vecteurs U et F sont décomposés de manière analogue. La matrice Q est une matrice d'interface qui s'apparente à la matrice B apparaissant dans la formulation matricielle de la méthode de Schur dual (voir paragraphe 2.3.3.2). Elle possède aussi une structure par blocs, chacun d'entre eux matérialisant la connexion entre les sous-domaines Ω_k et Ω_l voisins à travers une interface γ_{kl}. La multiplication par Q peut être vue comme une opération de relèvement sur l'interface du côté esclave. Celle par Q^T correspond à une opération de restriction.

Le système 6.5 est résolu par la méthode du gradient conjugué selon l'algorithme 6 [13]. On note que cet algorithme ne nécessite pas de solveur de système linéaire local. Les calculs locaux se résument en effet à des produits matrice-vecteur.

Les opérations que nous avons entourées nécessitent des communications. La mise en œuvre est similaire à celle du gradient conjugué pour l'algorithme d'Uzawa. Toutefois, dans ce solveur parallèle, le comportement des processeurs n'est pas symétrique car on distingue le "coté maître" du "coté esclave" pour traiter les contributions des interfaces dans la méthode des éléments joints. Pour une interface donnée, le sous-domaine maître envoie l'information à l'esclave lors de la multiplication par la matrice Q. Les opérations sont inversées lors de la multiplication par la transposée Q^T. L'algorithme 7 fournit de manière synthétique les opérations nécessaires pour effectuer la multiplication par Q.

Algorithme 6 :Algorithme CG pour la résolution de 6.5 sur le sous-domaine $\Omega_k, k = 1, ..., K$.

1: $U^0_{k_{maitre}} = 0$ {initialisation}
2: $\boxed{p^0 = r^0 = Q^T F_k}$
3: $\boxed{\epsilon^0 = \|r^0\|^2}$

4: **pour** $i = 0, 1, ...$ **jusqu'à** la convergence **faire**
5: $\quad\boxed{z = Qp^i}$
6: $\quad\xi = A_{kk}z$
7: $\quad\boxed{\psi = Q^T\xi}$
8: $\quad\boxed{\alpha^i = \|r^i\|^2/(\psi, p^i)}$
9: $\quad U^{i+1}_{k_{maitre}} = U^i_{k_{maitre}} + \alpha^i p^i$
10: $\quad r^{i+1} = r^i - \alpha^i\psi$
11: $\quad\boxed{\beta^{i+1} = \|r^{i+1}\|/\|r^i\|^2}$
12: $\quad p^{i+1} = r^{i+1} - \beta^{i+1}p^i$
13: **fin pour**
14: $\boxed{U = QU_{k_{maitre}}}$

Algorithme 7 : Multiplication par Q.

pour tout sous-domaine Ω_k **faire**
  initialisations
  pour tout interface γ_{kl} de Ω_k **faire**
    si γ_{kl} est esclave pour Ω_k **alors**
      multiplication matrice-vecteur
      envoi de contribution au processeur traitant Ω_l
    sinon
      réception de contribution du processeur traitant Ω_l
      multiplication matrice-vecteur
      résolution de système tridiagonal
    fin si
  fin pour
fin pour

6.3.3 Portage de l'application avec AHPIK

Dans un premier temps, nous nous sommes intéressés au portage de l'application à l'aide de AHPIK. Nous l'avons ensuite transformé en un code permettant des résolutions adaptatives avec l'ajout d'une estimation d'erreurs *a posteriori* et de procédures d'adaptation dynamique de maillage.

Le travail de portage a débuté avec la traduction du code en langage C++ à l'aide de la bibliothèque Blitz++[159]. Une alternative aurait été d'isoler des parties de code séquentiel en Fortran 90 puis de les appeler à partir du programme principal en C++. Cependant, par souci de portabilité, nous avons préféré la première solution.

La partie la plus importante du travail de portage concerne le solveur de vitesse car le solveur de pression ne nécessite pas de communications. Le découpage en tâches adopté est similaire à celui du gradient conjugué pour la méthode de Schur dual (paragraphe 5.4.5). Au niveau du schéma parallèle, une des principales différences entre ces deux méthodes réside dans l'asymétrie des communications entre sous-domaines voisins : les processus légers traitant les interfaces effectuent soit des envois soit des réceptions de données contrairement aux échanges (envois puis réceptions) typiques des autres schémas. La figure 6.6 fournit un exemple du code C++ correspondant à une tâche d'interface qui effectue une opération de relèvement sur le côté esclave d'une interface (multiplication par Q).

Dans le programme original, un même processeur gère plusieurs sous-domaines. Pour ce faire, le code exécuté par chaque processeur est composé de suites de boucles sur les sous-domaines qui lui ont été affectés. Les boucles sont entrelacées par des communications, ce qui permet de regrouper les données dans le même message lors des opérations nécessitant des communications globales. Lors du portage sur AHPIK, les boucles sur les sous-domaines disparaissent. En effet, la gestion de plusieurs sous-domaines par nœud est transparente dans AHPIK : les solveurs spécifient les calculs associés à un seul sous-domaine, tandis que le noyau prend en charge le regroupement des données lorsque nécessaire. Nous avons aussi tiré parti de la flexibilité offerte par la multiprogrammation légère pour exprimer en parallèle les calculs indépendants des composantes horizontale et verticale du champ de vitesse.

Le travail a poursuivi avec la transformation du code C++ en un code adaptatif. Pour ce faire, nous avons remplacé les structures de données représentant le maillage dans le code original par la classe `Adaptive2DMesh` offerte par AHPIK. Ensuite, nous avons ajouté l'estimation d'erreurs *a posteriori* basé sur [160] dans le processus de résolution. Tout raffinement géométrique du maillage est suivi de la mise à jour des structures de données représentant les matrices et vecteurs des

```
1  class
2  IfaceMultQTask1 : public IfaceTask
3  {
4      public :
5          template<class DDMethod, class Domain, class System>
6          void operator()(DDMethod& method,
7                          Domain& domain,
8                          System& system)
9          {
10             if (domain.IsSlave(IfaceId())) {
11                 method.LocalContributionMultQ1(IfaceId(), system) ;
12                 IfaceWrite() ;
13             } else {
14                 IfaceRead() ;
15                 method.InterfaceComputationMultQ1(IfaceId(), system) ;
16             }
17         }
18 } ;
```

Figure 6.6 *Code C++ correspondant à une tâche d'interface : relèvement sur le côté esclave de l'interface.*

systèmes à résoudre.

Difficultés rencontrées

Lorsque l'on utilise une bibliothèque développée par ailleurs on doit faire attention à ce qu'elle soit « thread-safe ». Ce n'est pas le cas de la bibliothèque Blitz++. Nous avons donc ajouté une couche logicielle à cette bibliothèque afin de permettre à plusieurs processus légers d'appeler les fonctions de manipulation de tableaux.

6.3.4 Résultats numériques

Notre première expérience consiste à comparer l'application originale avec la version reposant sur AHPIK dans un cas où la charge de calcul est bien distribuée sur les sous-domaines. Cette étude expérimentale a été menée sur deux plates-formes différentes : une grappe de PCs monoprocesseurs et une grappe de SMP composée de PCs biprocesseurs. Les deux grappes sont homogènes, mais les processeurs de la grappe de SMP sont plus rapides que ceux de la grappe de monoprocesseurs. Nous utilisons 22 nœuds de ces grappes pour chaque exécution parallèle, la plupart des nœuds ont donc 4 sous-domaines à résoudre. La figure 6.7 montre le temps d'une itération pour l'application originale sur MPI et pour l'application reposant sur AHPIK.

On voit que la version reposant sur AHPIK offre des performances légèrement inférieures à celles obtenues par la mise en œuvre originale (MPI) sur la grappe de PCs monoprocesseurs. Cela s'explique en partie par le bon équilibrage de charge qui caractérise cette expérience. En effet, lorsque le taux d'utilisation des processeurs est haut, l'utilisation de processus légers introduit un surcoût. Dans le cas de ATHAPASCAN-0, ce surcoût est dû à l'emploi d'un processus léger responsable de la scrutation des messages sur chaque nœud. Pour plus de détails sur ce sujet nous renvoyons à [32]. Sur un nœud multiprocesseur, la version reposant sur AHPIK obtient de meilleures performances que l'application originale pour des paramètres d'exécution identiques. On voit que la version combinant processus légers et communications s'adapte automatiquement aux multiprocesseurs, tandis que le code original n'exploite qu'un seul processeur.

Notre deuxième expérience consiste à incorporer l'adaptation de maillage aux deux versions de l'application. Cela introduit des déséquilibres de charge dès que le nombre de degrés de liberté associés au maillage varie d'un processeur à l'autre au cours de la résolution temporelle. Pour simplifier la mise en œuvre, on raffine

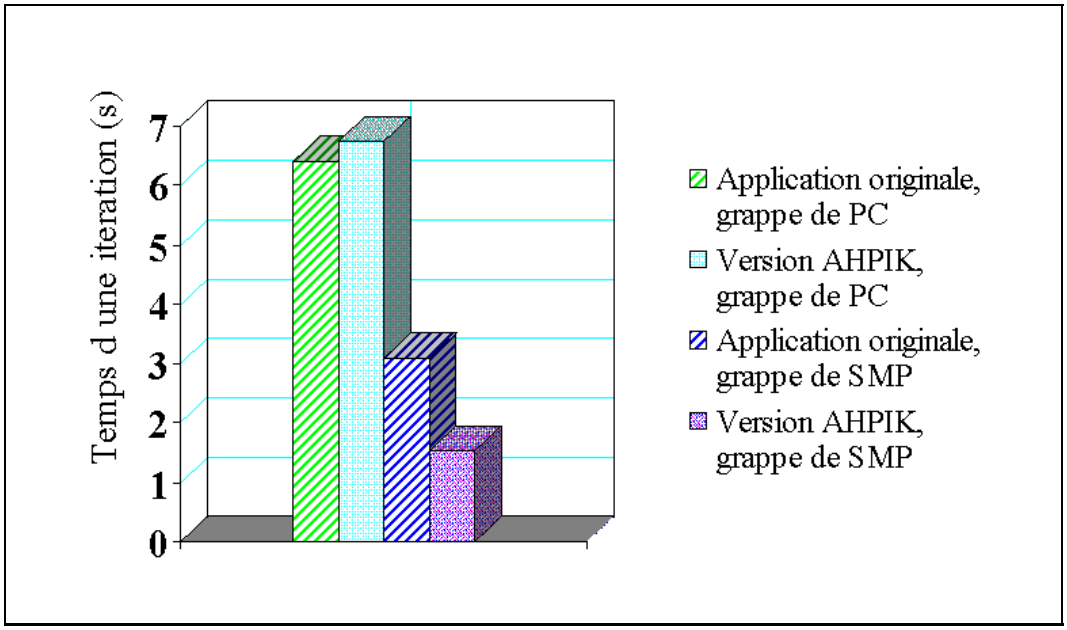

Figure 6.7 *Résultats pour une distribution équitable des sous-domaines.*

toujours un sous-domaine entier, ainsi la configuration originale du maillage est atteinte en un petit nombre d'adaptations. La figure 6.8 montre les résultats obtenus avec l'exécution des codes adaptatifs sur chacune des plates-formes considérées. Si les résultats sur la grappe de SMP reproduisent le comportement observé lors de notre première expérience, les résultats sur la grappe de monoprocesseurs montrent que, dans le cas où la charge de calcul n'est pas équilibrée, la version AHPIK peut atteindre ou même dépasser les performances du code adaptatif basé sur MPI.

On remarque que cette expérience représente un cas critique pour la multiprogrammation légère car les calculs locaux sont de gros grain et le schéma de recollement des interfaces requiert des synchronisations globales fréquentes. De meilleures performances pour la version reposant sur AHPIK sont attendues si ces synchronisations peuvent être relaxées.

6.4 Conclusion

La mise en œuvre basée sur les processus légers offre un degré de liberté supplémentaire au niveau de l'ordonnancement, et permet d'utiliser efficacement une plateforme SMP. Dans AHPIK, cette technique a aussi permis d'optimiser les communications entre les processus légers correspondant à des sous-domaines affectés

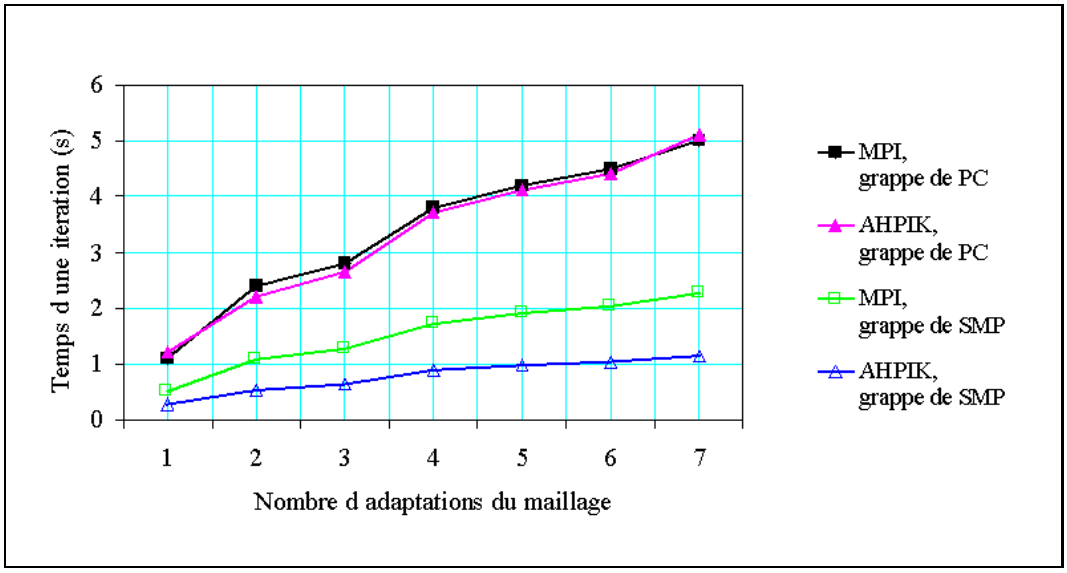

Figure 6.8 *Résultats pour le code adaptatif.*

à un même processeur : dans ce cas, les échanges de données se font dans la mémoire partagée, et non pas par le réseau. Ces détails étant encapsulés dans le noyau de AHPIK, leur exploitation dans les différentes applications n'exige pas d'efforts de la part de l'utilisateur.

Conclusion et perspectives

Dans cette thèse, nous nous sommes intéressés à la parallélisation de problèmes d'EDP par des méthodes mathématiques de décomposition de domaine, et plus particulièrement à l'utilisation du paradigme de processus légers communicants pour la mise en œuvre parallèle de ces méthodes. Ce paradigme constitue la base de l'environnement de programmation ATHAPASCAN développé au sein du projet APACHE, dans lequel s'inscrit notre travail. L'orientation vers les problèmes d'EDP a son origine dans une étroite coopération avec le projet IDOPT.

Les environnements intégrant communication et multiprogrammation légère suscitent beaucoup d'intérêt pour la mise en œuvre d'applications critiques car, au niveau matériel, les architectures de type grappe de machines à mémoire partagée (CluMPs) sont de plus en plus fréquemment choisies par les décideurs. On remarque cependant que ce paradigme n'est pas réellement exploité dans les bibliothèques parallèles applicatives (voir l'état de l'art au chapitre 3).

Dans le cadre de la parallélisation de problèmes d'EDP à l'aide de méthodes de décomposition de domaine, un problème critique est de trouver un compromis entre le maximum de parallélisme potentiel et le minimum de perte en convergence. En effet, bien que le degré de parallélisme croisse lorsque le problème est découpé en un grand nombre de sous-problèmes, ce découpage est en général nuisible à la convergence des méthodes numériques. L'équilibrage de la charge, souci permanent dans toute programmation parallèle, est d'autant plus crucial que les méthodes de décomposition de domaine sont classiquement synchrones. Ainsi il est parfois souhaitable d'augmenter le nombre de sous-domaines pour mieux distribuer la charge de calcul entre les processeurs, même si cela se fait au détriment de la vitesse de convergence. On affecte alors plusieurs sous-domaines par nœud de calcul. Les méthodes itératives asynchrones sont également une alternative pour aborder le problème puisqu'il s'agit de désynchroniser les calculs pour éviter les temps d'attente.

Après des chapitres introductifs sur la multiprogrammation (chapitre 1), sur les méthodes de décomposition de domaine (chapitre 2) et un état de l'art (chapitre 3),

nous avons abordé ces problèmes sous plusieurs angles. Cela nous a, entre autres, conduit à l'écriture d'une bibliothèque flexible, AHPIK, pour la résolution en parallèle de problèmes d'EDP de grande taille afin de montrer que le paradigme de processus légers communicants est adapté à ce type de méthodes. S'appuyant sur la multiprogrammation légère, cette bibliothèque permet d'exploiter aisément les différentes formes de parallélisme présentes dans une architecture parallèle (cf. paragraphe 1.4.1).

Dans les chapitres 4 et 5, nous avons montré que le paradigme de processus légers communicants s'adapte bien à cette stratégie d'équilibrage de charge. En proposant d'utiliser un ensemble de processus légers pour les calculs et communications correspondants à un sous-domaine, nous avons codé des solveurs parallèles de manière générique. Pour utiliser ceux-ci, il suffit de coder le solveur du problème d'EDP écrit dans le sous-domaine : cette partie du travail est du ressor de l'utilisateur, elle est indépendante des aspects informatiques de la gestion du parallélisme. En effet, dans AHPIK, l'utilisateur n'a pas à organiser explicitement les séquences calculs/communications car deux schémas parallèles génériques d'entrelacement des tâches (point fixe et gradient conjugué) s'en chargent. La programmation orientée objet de AHPIK a largement facilité l'écriture générique des solveurs et schémas parallèles. Le support aux méthodes asynchrones constitue également un des atouts de l'environnement AHPIK car, le parallélisme n'étant pas explicite, les schémas parallèles peuvent être exécutés suivant l'un ou l'autre mode de synchronisation.

Nous avons montré graphiquement le comportement parallèle d'applications reposant sur une méthode de décomposition de domaine. Les diagrammes d'exécution présentés couvrent une large gamme de comportements parallèles pour les applications reposant sur une méthode de décomposition de domaine. Ces résultats sont bien connus de la communauté "informatique", mais pas forcément de la communauté "mathématique". Nous jugeons que les diagrammes présentés peuvent contribuer aux rapprochement des deux disciplines.

Perspectives et travaux futurs

Les perspectives ouvertes à la suite de nos travaux ont la même double orientation qui s'est manifestée au cours de cette thèse. Elles concernent des problèmes relatifs à l'algorithmique parallèle ainsi qu'à la mise en œuvre des méthodes numériques.

L'extension des fonctionnalités de AHPIK pour le support aux calculs avec des maillages adaptatifs constitue, à notre avis, une des voies les plus intéressantes pour

la poursuite de ce travail. En effet, AHPIK contient un support géométrique pour l'adaptation de maillage (voir chapitre 5). Celui-ci a été utilisé dans l'application en mécanique des fluides décrite au paragraphe 6.3. Il conviendrait d'étendre AHPIK par un ensemble de méthodes permettant la répartition dynamique de la charge de calcul pour remédier aux déséquilibres résultant d'étapes d'adaptation. Dans ce sens, les processus légers encapsulés dans le noyau de AHPIK offrent un degré de liberté important pour la mise en œuvre et l'évaluation de différentes stratégies de répartition de charge.

Encore visant la répartition dynamique de charge, il serait intéressant d'envisager une mise en œuvre de AHPIK basée sur une couche logicielle telle que ATHAPASCAN-1, l'interface de programmation haut niveau développée dans le cadre du projet APACHE. En effet, l'expression du parallélisme sous forme d'un graphe de tâches est conforme au paradigme de programmation proposé par ATHAPASCAN-1. Dans sa version actuelle, AHPIK laisse au programmeur la définition de la granularité des tâches. Une version basée sur ATHAPASCAN-1 pourrait tirer parti d'un découpage dynamique en tâches (ainsi que d'un ordonnancement de ces tâches) s'adaptant automatiquement à la plate-forme parallèle hôte.

Toujours en ce qui concerne le support aux calculs adaptatifs, une piste complémentaire à suivre serait l'inclusion, dans AHPIK, de mécanismes de gestion de structures de données distribuées pouvant évoluer dynamiquement (paragraphe 3.1.2.2). En l'état, les schémas de calcul offerts par AHPIK sont complètement indépendants des structures de données utilisées pour représenter les matrices et vecteurs distribués. Le challenge serait donc d'incorporer dans AHPIK une couche logicielle gérant des structures de données dynamiques tout en gardant le caractère générique des schémas de calcul.

Une autre perspective qui s'ouvre grâce à la réalisation de AHPIK est l'expérimentation autour des méthodes numériques. Dans ce contexte s'insèrent, par exemple, l'étude et l'utilisation des méthodes asynchrones, la recherche autour des préconditionneurs parallèles et l'extension de AHPIK à d'autres algorithmes où l'on pourrait utiliser la notion de sous-domaine (méthodes de « multisplitting » en algèbre linéaire, par exemple). Cette voie est particulièrement intéressante par son caractère multidisciplinaire.

La poursuite de l'évaluation expérimentale de AHPIK constitue aussi un autre point pouvant être développé à l'avenir. Au cours du développement de AHPIK, nous nous sommes essentiellement concentrés sur la conception générique des schémas de calcul parallèles, en évitant de nous lancer sur une optimisation prématurée des codes. Une évaluation expérimentale plus poussée mettrait en évidence les sources d'inefficacité provenant directement de notre mise en œuvre, permettant

ainsi d'ouvrir la voie pour l'amélioration des performances parallèles de AHPIK. Un aspect méritant certainement d'être repensé concerne l'algorithme centralisé pour les opérations de réduction globales : bien que simple à mettre en œuvre, un tel algorithme constitue un goulot d'étranglement lorsque l'on utilise un grand nombre de processeurs.

Il ne faut cependant pas perdre de vue que les performances de AHPIK seront toujours dépendantes du noyau de communication sous-jacent. À ce titre, un autre axe pour la continuation de ce travail serait le portage de AHPIK sur une autre bibliothèque que ATHAPASCAN-0. Une possibilité qui se dévoile est l'utilisation du noyau de communication en cours de conception dans le cadre des projets APACHE et MultiCluster, ce dernier s'intéressant au déploiement de grappes et grilles de grappes pour le calcul parallèle et distribué. Un tel portage serait suivi de nouvelles mesures de performance sur les grappes disponibles. Cela permettrait de vérifier le passage à l'échelle de centaines de processeurs, ainsi que de contribuer à l'évaluation de ce nouveau noyau de communication pour une classe d'applications très représentative du calcul scientifique.

Annexe A

Méthode des éléments finis

A.1 Maillage du domaine physique

Un maillage $\mathcal{M}_h$ d'un domaine Ω, est un ensemble de N_k éléments géométriques "simples" (également appelés mailles) recouvrant Ω. L'indice h est le pas de discrétisation du maillage, il correspond à la plus grande arête d'un élément appartenant à $\mathcal{M}_h$. Plus h est petit, meilleure sera l'approximation de la solution du problème posé sur Ω. Chaque élément est noté T_k et le domaine recouvert Ω_h est tel que $\Omega_h = \cup_{k=1}^{N_k} T_k$. Le choix de la géométrie des éléments dépend de la géométrie du domaine Ω et de la nature du problème à résoudre. En dimension 2, ce sont par exemple des triangles ou des rectangles (resp. des tétraèdres ou des parallélépipèdes en dimension 3). Les éléments constituant le maillage doivent satisfaire certaines conditions. En particulier, il est nécessaire que le maillage soit *conforme*, c'est à dire, que $\mathcal{M}_h$ vérifie les propriétés suivantes :

1. tout élément T_k de $\mathcal{M}_h$ est d'aire non nulle ;
2. l'intersection de deux éléments de $\mathcal{M}_h$ est soit vide, soit réduite à un sommet ou une arête complète (ou une face complète en dimension 3).

La figure A.1 présente des maillages en triangles d'un domaine Ω illustrant les différences entre un maillage conforme et un maillage non-conforme. La suite de l'exposé de la méthode des éléments finis discute uniquement de maillages conformes.

D'autres propriétés géométriques sont à vérifier pour que la méthode des éléments finis fournisse de bonnes approximations de la solution (voir entre autre

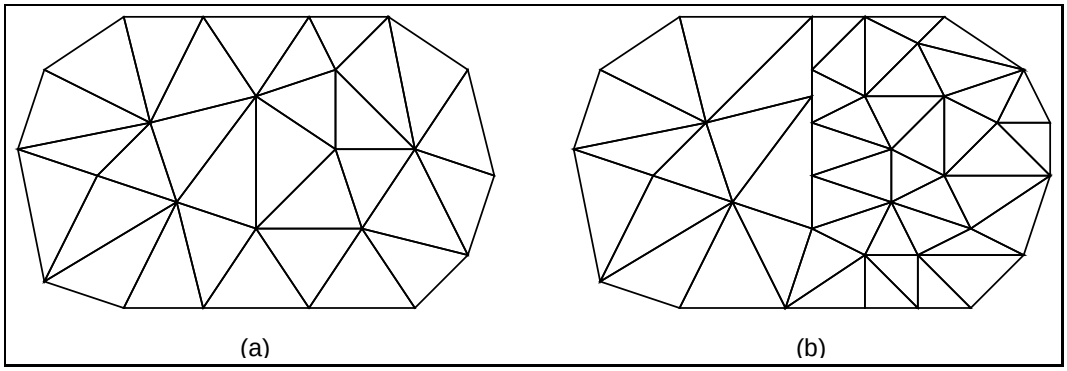

Figure A.1 *(a) Maillage conforme, (b) Maillage non-conforme.*

[131]). Par exemple, la taille des éléments doit varier progressivement, c'est-à-dire, sans présenter de discontinuités trop brutales, et les éléments triangulaires ou quadrangulaires ne doivent pas présenter d'angles trop obtus ou trop aigus. De même, il est souhaitable que la rondeur ρ_k des éléments soit petite. En dimension 2, ρ_k s'écrit :

$$\rho_k = \frac{h_k}{g_k}$$

où h_k est la taille de la plus grande arête de l'élément T_k (équivalente au diamètre du cercle circonscrit à T_k), et g_k est le diamètre du cercle inscrit dans T_k. Les caractéristiques géométriques du maillage sont fondamentales pour la méthode des éléments finis, car elles permettent de démontrer les propriétés d'approximation de la méthode. Elles sont étroitement liées au problème physique à résoudre et à ses éventuelles singularités. En conséquence, la nature du problème conditionne le choix de mailles plus fines (i.e. une plus grande densité d'éléments) dans certaines zones du domaine pour obtenir des solutions avec suffisamment de précision.

En ce qui concerne la mise en œuvre informatique, un maillage est fondamentalement représenté par une liste de sommets et une liste d'arêtes définissant l'interconnexion entre les sommets (nous verrons plus tard que cette représentation ne suffit pas pour la méthode des éléments finis). Selon la topologie d'interconnexion des sommets, les maillages peuvent être classés comme structurés ou non-structurés[82]. Dans les maillages **structurés**, le schéma d'interconnexion est tel que la connaissance des indices d'un sommet en donne la position relative. C'est le cas, par exemple, des maillages en quadrangles lorsque l'on travaille en dimension 2. Dans un tel maillage, un sommet noté (i, j) a pour voisin de "gauche" le sommet $((i-1), j)$ et pour voisin de "droite" le sommet $((i+1), j)$. Dans les maillages **non-structurés**, en revanche, le schéma d'interconnexion des sommets est de type quelconque. Les maillages non-structurés sont donc plus généraux, ils permettent en particulier de recouvrir des domaines ayant une géométrie complexe. La méthode

des éléments finis se prête bien à l'utilisation de maillages non-structurés, tandis que les maillages structurés sont plus souvent associés aux méthodes de discrétisation de type différences finies.

La génération de maillages adaptés aux problèmes à résoudre n'est pas une opération simple, surtout pour des domaines de géométrie irrégulière. Beaucoup de travaux de recherche portent sur des techniques permettant de générer des maillages automatiquement. Pour plus de détails sur ce sujet, nous renvoyons à [82].

A.2 Définition des éléments finis

Le maillage du domaine physique n'est qu'un support géométrique pour la méthode des éléments finis. Pour que les éléments géométriques du maillage deviennent des "éléments finis", quelques définitions doivent être ajoutées. Un maillage $\mathcal{M}_h$ agrémenté de telles définitions est appelé **triangulation** et noté $\mathcal{T}_h$.

Soit :
- une partie compacte T de $\mathbb{R}^2$, connexe et d'intérieur non-vide (triangle, quadrangle, etc.) ;
- un ensemble fini $E^T = \{a_j\}_{j=1}^n$ de n points ou nœuds distincts définis sur T ;
- un espace vectoriel P^T de dimension finie et composé de fonctions définies sur T à valeurs réelles.

Le triplet (T, P^T, E^T) est appelé **élément fini de Lagrange** lorsque, étant donné n scalaires réels quelconques $(\alpha_j)_{j=1}^n$, il existe une unique fonction $p \in P^T$ telle que, en tout nœud a_j de E^T :

$$p(a_j) = \alpha_j, \quad \forall j = 1, ..., n.$$

La fonction p peut également s'écrire, pour tout nœud a_j de E^T :

$$p(a_j) = \sum_{i=1}^{n} \alpha_i p_i(a_j).$$

où les fonctions $p_i, i = 1, ..., n$ vérifient la condition :

$$p_i(a_j) = \begin{cases} 1 & \text{si} \quad i = j \\ 0 & \text{si} \quad i \neq j \end{cases} \quad \forall j = 1, ..., n \tag{A.1}$$

✍ *intérpretation : pour toute fonction v, il existe une fonction $p \in P_T$ et une seule qui interpole v sur E_T*

Les fonctions $p_i, i = 1, ..., n$ sont appelées **fonctions de base** pour l'élément fini de Lagrange (T, P^T, E^T). Dans la littérature anglo-saxonne, ces fonctions sont désignées sous le nom de « shape functions » ; c'est pourquoi ces fonctions sont aussi parfois appelées fonctions de forme.

Pour simplifier l'exposé, regardons ce que signifient ces relations dans le cas du maillage bidimensionnel illustré dans la figure A.2, recouvrant un domaine rectangulaire $]-3, 3[\times]-1, 1[$. Ce maillage est constitué de 35 sommets et 48 éléments triangulaires.

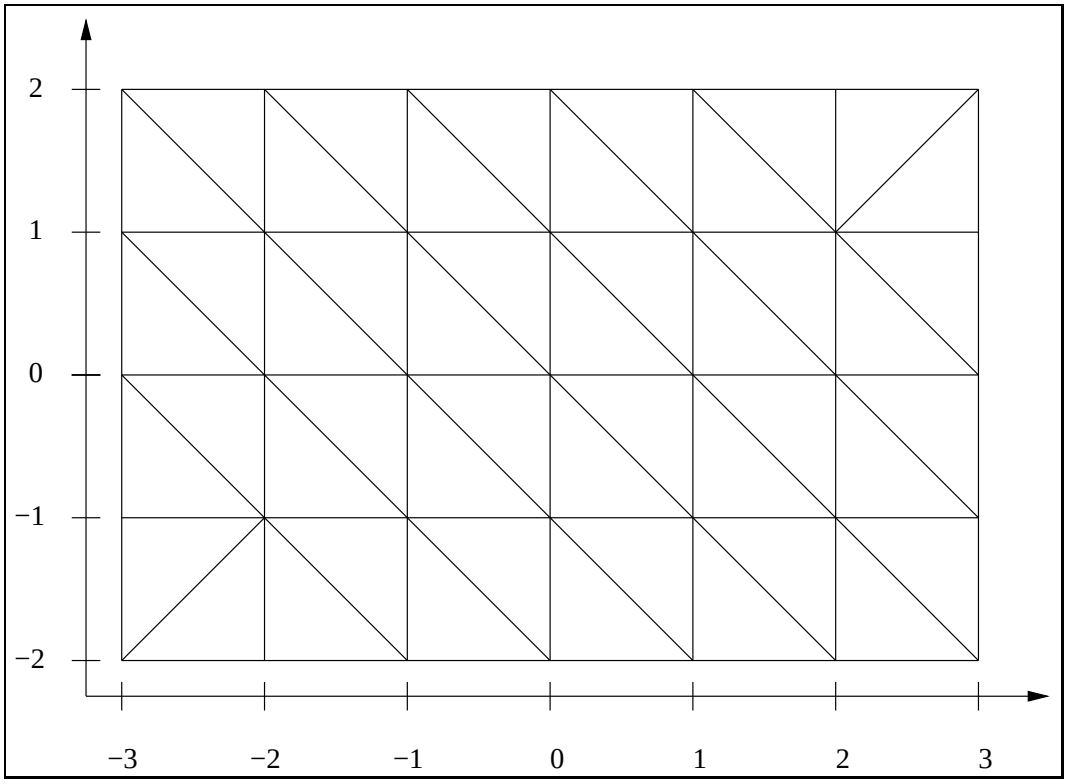

Figure A.2 *Maillage d'un domaine rectangulaire.*

Nous allons maintenant définir une triangulation $\mathcal{T}_h$ sur le maillage de la figure A.2. Pour cela nous avons choisi d'utiliser les éléments de type P_1, c'est-à-dire, les éléments finis de Lagrange de degré 1. Un élément P_1 est caractérisé par le triplet (T, P^T, E^T) où

- T est un triangle appartenant à un maillage $\mathcal{M}_h$;
- $E^T = \{a_1, a_2, a_3\}$, c'est-à-dire, un ensemble de 3 nœuds, chaque nœud coïncidant avec les sommets du triangle M_k ;
- $P^T = P_1$, c'est-à-dire, l'espace des polynômes de degré inférieur ou égal à 1 en x et y.

L'élément P_1 de référence est présenté dans la figure A.3.

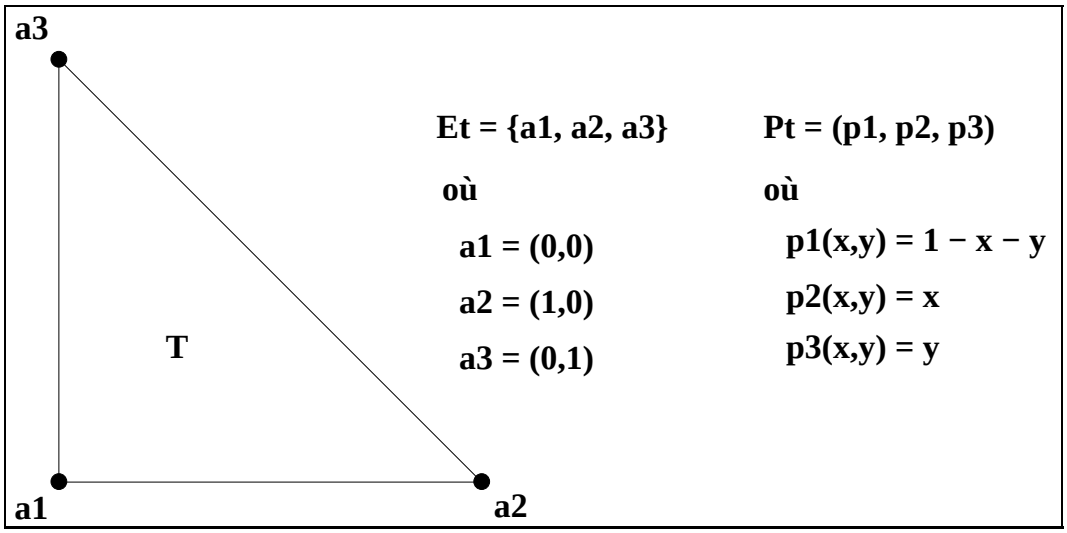

Figure A.3 *Élément fini P_1.*

Il convient remarquer que nœuds des éléments finis ne coïncident pas toujours avec les sommets des éléments géométriques. Il existe des éléments où les sommets ne sont pas des nœuds (élément fini de type hybride primal, par exemple), ou des éléments dont les nœuds sont pris sur les arêtes ou même à l'intérieur de l'élément géométrique (élément fini de Lagrange de degré 3).

Les définitions que nous venons d'introduire sont généralisables à toute la triangulation $\mathcal{T}_h$. Lorsque l'on travaille avec la triangulation complète, l'union des ensembles E^T associés à chacun des éléments de la triangulation forme un ensemble $E = \{a_j\}_{j=1}^N$ de N nœuds distincts définis sur $\mathcal{T}_h$. Les nœuds de E sont appelés **degrés de liberté** ; ce sont les inconnues de notre problème discret. À chaque nœud correspond une fonction de base continue, que nous allons noter à nouveau p_i, vérifiant la condition

$$p_i(a_j) = \begin{cases} 1 & \text{si} \quad i = j \\ 0 & \text{si} \quad i \neq j \end{cases} \quad \forall j = 1, ..., N$$

Chacune de ces fonctions a pour support l'ensemble des éléments contenant le nœud a_j et vaut 0 sur le restant de $\mathcal{T}_h$ (voir figure A.4).

Le nombre de degrés de liberté par nœud dépend en réalité de la nature du problème traité. Notre problème modèle admet en effet un seul degré de liberté par nœud (déplacements verticaux). Pour certains problèmes d'élasticité, par exemple, six degrés de liberté sont nécessaires pour définir les rotations et translations en X, Y et Z.

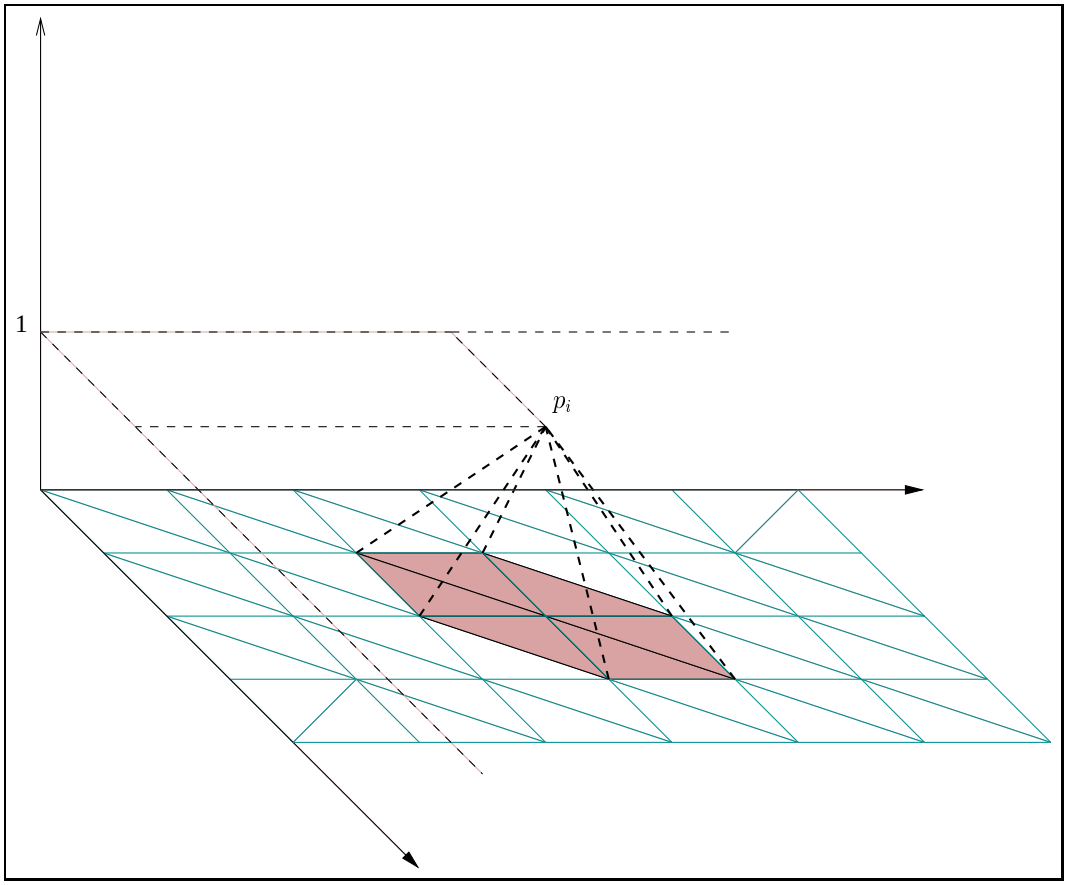

Figure A.4 *Représentation graphique des fonctions de base pour un domaine bidimensionnel.*

Nous pouvons maintenant introduire l'espace V_h sur lequel nous allons déterminer notre solution approchée u_h. Il s'agit d'un espace de fonctions v continues sur Ω_h, telles que la restriction de v à tout triangle $T \in \mathcal{T}_h$ appartient à l'ensemble de polynômes de degré inférieur ou égal à 1. La dimension de l'espace V_h est égale au nombre total d'inconnues associeés au maillage $\mathcal{T}_h$, les fonctions de V_h étant entièrement déterminées par leurs valeurs en chacun de nœuds de ces éléments. Dans cet espace, l'approximation à la solution de (2.2) s'écrit donc :

$$u_h = \sum_{i=1}^{N} u_i p_i \tag{A.2}$$

où N est le nombre total d'inconnues (ou degrés de liberté) associées au domaine discrétisé Ω_h et $u_1, u_2, ..., u_N$ sont les valeurs de la solution approchée pour chaque inconnue.

Avec ces notations, la formulation variationnelle discrète associée à (2.2) s'écrit

$$\int_{\Omega_h} \nabla u_h \cdot \nabla p_j dx = \int_{\Omega_h} f p_j dx, \quad \forall p_j \in V_h. \tag{A.3}$$

Cette égalité est vraie pour toutes les fonctions de base $p_j, j = 1, ..., N$ appartenant à l'espace V_h.

En utilisant l'écriture (A.2) de u_h dans la base V_h, la formule (A.3) devient

$$\sum u_i \int_{\Omega_h} p_i p_j dx = \int_{\Omega_h} f p_j dx, \quad \forall p_i, p_j \in V_h. \tag{A.4}$$

Le problème (A.4) est équivalent à la résolution d'un système linéaire

$$AU = F, \tag{A.5}$$

où A est une matrice de taille $N \times N$, d´éléments

$$A_{ij} = \int_{\Omega_h} \nabla p_i \cdot \nabla p_j dx, \quad \forall i, j = 1, ..., N, \tag{A.6}$$

U est le vecteur solution $u_1, u_2, ..., u_N$, et F est un vecteur de dimension N, de composantes

$$F_j = \int_{\Omega_h} f p_j dx, \quad \forall i, j = 1, ..., N. \tag{A.7}$$

A.3 Assemblage du système

Le calcul des coefficients A_{ij} de la matrice A selon la définition (A.6) se fait en parcourant tous les éléments du maillage. La contribution de l'élément T au coefficient A_{ij} s'écrit

$$A_{ij}(T) = \int_{M_k} \nabla p_i \cdot \nabla p_j dx, \tag{A.8}$$

et le coefficient A_{ij} est calculé par la formule

$$A_{ij} = \sum_{k=1}^{K} A_{ij}(T).$$

La matrice A, dite **matrice de rigidité**, est creuse au sens où la plupart de ses coefficients est nulle. En effet, $A_{ij}(T) \neq 0$ si et seulement si a^i et a^j sont des nœuds

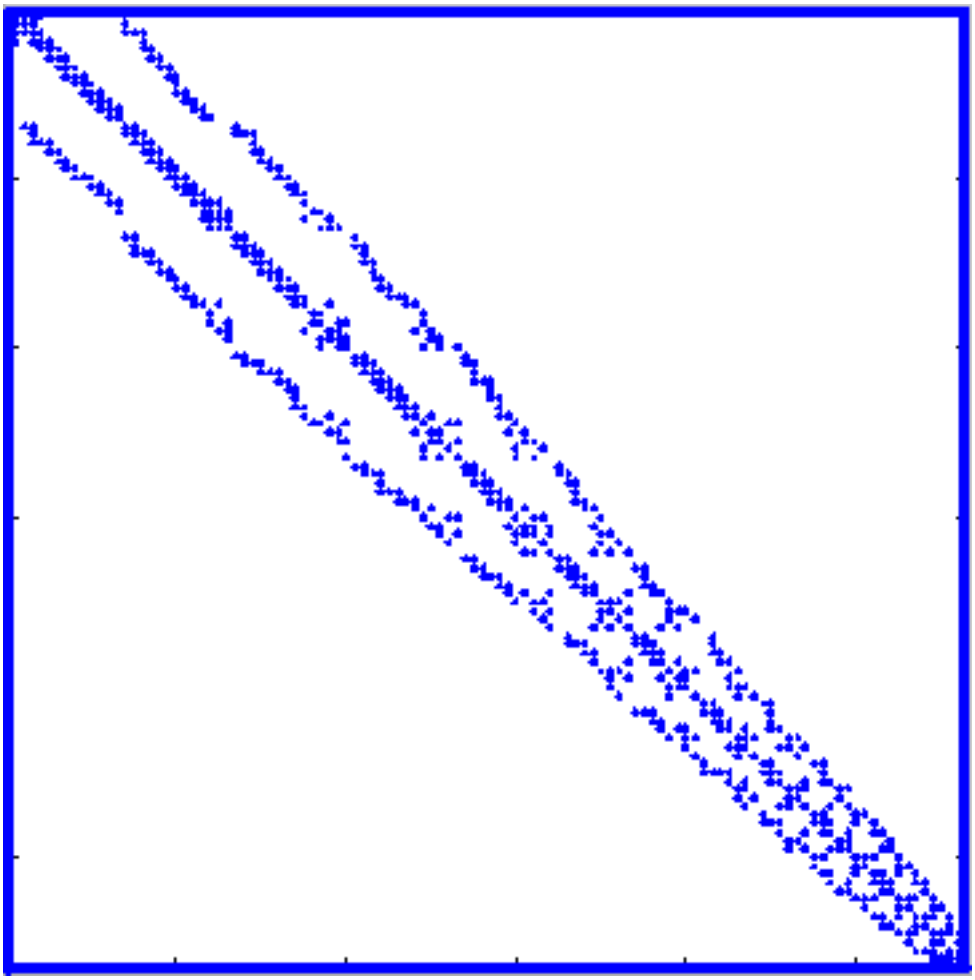

Figure A.5 *Profil de la matrice de rigidité pour le problème de Poisson.*

du triangle M_k. Le profil de A pour le problème modèle que nous avons choisi est illustrée dans la figure A.5 (les coefficients non-nuls sont représentés par des carrés). Ce profil est étroitement lié à la numérotation des nœuds choisie.

Dans (A.7) figurent des intégrales qu'on ne peut, en général, calculer de manière exacte. On utilise donc des formules de quadrature, c'est-à-dire, on calcule les intégrales de manière approchée.

En procédant ainsi, on ne calcule pas les éléments nuls de la matrice.

Algorithme 8 : Algorithme générique pour l'assemblage de la matrice de rigidité.

$A_{ij} \leftarrow 0$
pour $k = 1$ **jusqu'à** nombre d'éléments du maillage **faire**
  initialisation des constantes pour la formule de quadrature (points, poids)
  pour $i = 1$ **jusqu'à** nombre de degrés de liberté du $k^{ème}$ élément **faire**
    pour $j = 1$ **jusqu'à** nombre de degrés de liberté du $k^{ème}$ élément **faire**
      $A_{ij} \leftarrow A_{ij} + \int_k p_i p_j dx$
    fin pour
    $F_i \leftarrow F_i + \int_k p_i f dx$
  fin pour
fin pour

A.4 Prise en compte des conditions aux limites

L'algorithme d'assemblage décrit dans la section précédente rempli la matrice A et le vecteur F pour tout degré de liberté défini sur le domaine discret Ω_h, y compris les nœuds situés sur la frontière Γ. Pour compléter la procédure, il faut ensuite "ajuster" les valeurs correspondantes aux nœuds frontière afin de prendre en compte les conditions aux limites. Il s'agit d'assurer, pour toute inconnue i associée à un nœud situé sur la frontière Γ, la condition $u_i = f(a_i)$. Pour ce faire, une technique consiste à remplir cette condition de manière approchée en multipliant l'élément diagonal A_{ii} de la matrice et l'élément F_i du second membre par une très grosse constante e.

Annexe B

Algorithmes de raffinement de maillage

Il existe plusieurs techniques qui permettent de modifier un maillage au cours d'un calcul adaptatif. Le choix de la technique est fait en fonction de la nature du problème. Pour des problèmes stationnaires, la tendance est d'utiliser une technique optimale, qui est plus coûteuse mais fournit des meilleurs résultats. Pour des problèmes qui dépendent du temps, un grand nombre d'adaptations peuvent être nécessaires au cours du calcul : l'algorithme d'adaptation doit être rapide et doit être capable d'effectuer raffinements et déraffinements. L'interpolation entre un maillage et l'autre doit être précise.

Il convient de rappeler que la qualité d'une solution obtenue avec la méthode des éléments finis est fonction du type d'élément choisi (ordre des polynômes de base) et de la finesse du maillage. Les techniques d'adaptation agissent alors en essayant d'améliorer l'un ou l'autre de ces critères :

Raffinement p Consiste à augmenter les degrés des polynômes de base. En d'autres mots, on associe plus de nœuds à chaque élément, sans changer sa représentation géométrique.

Raffinement h Consiste à augmenter le nombre de nœuds et d'éléments composant le maillage, de façon à réduire la taille des éléments discrétisant le domaine. La mise en œuvre passe par un découpage des éléments, suivi généralement d'une division des éléments voisins afin de garder la conformité du maillage. Cette méthode est rapide, et possède l'avantage de permettre le retour à la grille grossière en renversant le processus de découpage. Cette méthode permet aussi une interpolation précise entre les mailles emboîtées.

Elle est idéale pour les problèmes dépendant du temps. La connectivité du maillage change, ainsi cette méthode est seulement appropriée aux maillages non structurées.

Raffinement r Consiste à déplacer (*relocaliser*) les sommets afin d'obtenir une meilleure solution avec un nombre fixe d'inconnues. Cette méthode peut seulement faire face à des petits réglages, sans introduire des éléments très dégénérés, mais elle est rapide et permet tant le raffinement comme le deraffinement du maillage. Elle est appropriée pour les maillages structurés, qui exigent une connectivité fixe, et pour les problèmes dépendant du temps car la vitesse et la capacité de raffiner et deraffiner sont importantes. ✍ *Réfs : Dufour*

Raffinement m Consiste à changer les équations ou le modèle physique selon le comportement local de la solution approchée. Par exemple, on peut utiliser des équations linéarisées si les termes non-linéaires du modèle physique sont négligeables.

Remaillage Consiste à générer le maillage avec un nouveau pas de discrétisation. Cette méthode est plus générale, mais elle est plus lente, et il est difficile de renverser le raffinement.

Dans ce travail nous avons utilisé le raffinement h décrit dans le paragraphe suivant.

B.1 Adaptation par découpage

On dispose d'une discrétisation du domaine en triangles. On note $\mathcal{T}_k$ le maillage de départ – c'est une triangulation conforme. Une phase d'estimations d'erreur[] détermine un ensemble S_T de triangles où l'erreur locale est grande. Pour améliorer la qualité de la solution du problème d'EDP, on raffine les triangles de S_T. La construction d'une triangulation adaptée T^{k+1} par la technique de découpage se fait en deux phases :

- découpage des éléments qui ont été désignés par l'estimateur d'erreur : chaque triangle de S_T est découpé en quatre triangles homothétiques (figure B.1(b)). Cela produit un maillage T^{k+1} qui n'est pas conforme entre la zone raffinée et une zone non raffinée ;
- découpage de mise en conformité : s'applique aux triangles ayant un nœud nouvellement crée sur l'une de leurs arêtes. Une technique consiste à ajouter des triangles "verts". On joint le milieu de l'arête coupé en deux au sommet opposé pour former deux nouveaux triangles (figure B.1(b)). Ces découpages produisent des éléments de qualité moindre par rapport aux éléments

introduits dans la phase de découpage standard. Dans des découpages successifs, les triangles “verts”sont supprimés avant l’étape de raffinement (figure B.1(c)).

La figure B.1 illustre l’application de cette technique pour deux raffinements successifs.

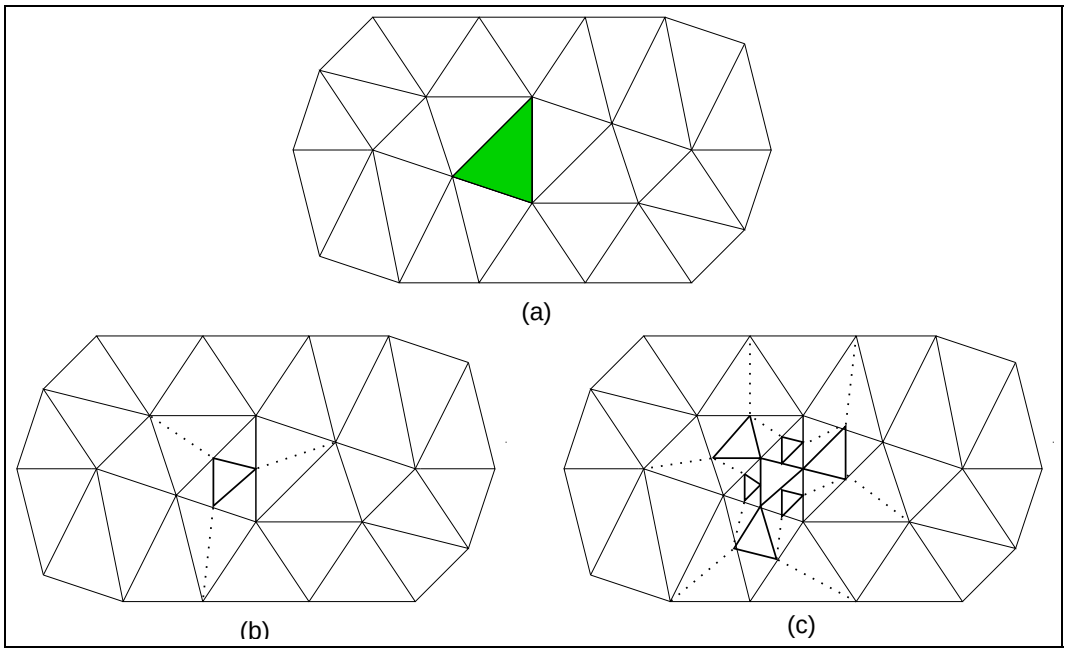

Figure B.1 *Raffinement par découpage. (a) Triangle à raffiner. (b) Premier raffinement. (c) Deuxième raffinement.*

Annexe C

La méthode de Schur dual

C.1 Réecriture du problème

Soit $a(.,.)$ une forme bilinéaire, continue et coercive de $H_0^1(\Omega)$. Soit $L(.)$ une forme linéaire de $L^2(\Omega)$. Alors le problème

$$Trouver\ u \in H_0^1(\Omega)\ tel\ que\ a(u,v) = L(v) \quad \forall v \in H_0^1(\Omega) \tag{C.1}$$

admet une unique solution appartenant à $H_0^1(\Omega)$.

Lorsque $a(.,.)$ est symétrique i.e. $a(u,v) = a(v,u)\ \forall u, v \in H_0^1(\Omega)$, le problème (C.1) équivaut au problème de minimisation

$$Trouver\ u \in H_0^1(\Omega)\ tel\ que\ J(u) \leq J(v) \quad \forall v \in H_0^1(\Omega). \tag{C.2}$$

où J est la fonctionnelle quadratique définie par

$$J(v) = a(v,v) - L(v) \quad \forall v \in H_0^1(\Omega). \tag{C.3}$$

C.2 Schur dual

Pour simplifier, on suppose que

$$a(u,v) = \int_\Omega \nabla u.\nabla v dx \quad \forall u, v \in H_0^1(\Omega).$$

et on pose

$$\begin{cases} V_1 = \{v_1 \in H^1(\Omega_1) \,|v_1 = 0 \text{ sur } \partial\Omega \cap \partial\Omega_1\}, \\ V_2 = \{v_2 \in H^1(\Omega_2) \,|v_2 = 0 \text{ sur } \partial\Omega \cap \partial\Omega_2\}, \\ V = \{(v_1, v_2) \in V_1 \times V_2 \mid v_1 = v_2 \text{ sur } \gamma_{12}\}. \end{cases}$$

Pour résoudre le problème (C.1) sur Ω par la méthode de décomposition de domaine, on utilise les fonctionnelles

$$J_k(v_k) = \frac{1}{2}\int_{\Omega_k} |\nabla v_k|^2 dx - \int_{\Omega_k} f v_k dx, \quad \forall v_k \in V_k,\ \forall k = 1, 2$$

et on résout le problème de minimisation sous la contrainte présente dans la définition de V :

$$\begin{aligned} Trouver \quad & (u_1, u_2) \in V \; tels \; que \\ & J_1(u_1) + J_2(u_2) \leq J_1(v_1) + J_2(v_2) \quad \forall (v_1, v_2) \in V. \end{aligned} \tag{C.4}$$

Pour prendre en compte la contrainte à l'interface dans le problème de minimisation, on introduit un multiplicateur de Lagrange μ. Le Lagrangien $\mathcal{L} : V \times L^2(\gamma_{12}) \to I\!R$ associé au problème (C.4) est défini comme suit :

$$\mathcal{L}(v_1, v_2, \mu) = J_1(v_1) + J_2(v_2) + \int_{\gamma_{12}} \mu(v_1 - v_2) ds.$$

On peut montrer que le point $(u_1, u_2, \lambda) \in V \times L^2(\gamma_{12})$ est point selle de $\mathcal{L}$ i.e.

– le point $(u_1, u_2) \in V$ est un minimum de $L(., ., \lambda)$

$$\mathcal{L}(u_1, u_2, \lambda) = \inf_{(u_1, u_2) \in V} \mathcal{L}(v_1, v_2, \lambda),$$

– le point $\lambda \in L^2(\gamma_{12})$ est un maximum de $\mathcal{L}(u_1, u_2, .)$

$$\sup_{\mu \in L^2(\gamma_{12})} \mathcal{L}(u_1, u_2, \mu) = \mathcal{L}(u_1, u_2, \lambda),$$

est solution du problème de minimisation.

C.3 Différence entre les 2 méthodes de Schur

On considère une décomposition sans recouvrement de Ω telle que

$$\overline{\Omega} = \overline{\Omega_1} \cup \overline{\Omega_2}, \text{ tel que } \Omega_1 \cap \Omega_2 = \emptyset.$$

Les méthodes de décomposition de Schur, primale et duale, se distinguent par la manière d'imposer la "continuité" à l'interface :

– Schur primal suppose

$$u_1 = u_2 \text{ sur l'interface ;}$$

– Schur dual suppose (dualisation de la contrainte)

$$\int_\gamma \mu(u_1 - u_2)ds = 0 \quad \forall \mu \in L^2(\gamma_{12}). \tag{C.5}$$

Annexe D

La méthode du gradient conjugué

L'algorithme de gradient conjugué générique présenté au chapitre 4 est dérivé de l'algorithme figurant dans Lascaux-Théodore [105]. Pour résoudre le problème

$$Ax = b, \tag{D.1}$$

on effectue les opérations suivantes

1. Initialisations
   (a) x^0 donné,
   (b) $p^0 = r^0 = b - Ax^0$,
   (c) $\beta^0 = ||r^0||^2$;
2. Itérations, pour $i = 0, 1, ...$
   (a) calcul du pas de descente :
   $$\alpha^i = ||r^i||^2/(Ap^i, p^i),$$
   (b) calcul de la direction de descente :
   $$x^{i+1} = x^i + \alpha^i p^i,$$
   (c) calcul du résidu :
   $$r^{i+1} = r^i - \alpha^i Ap^i,$$
   (d) calcul de l'erreur :
   $$\beta^{i+1} = ||r^{i+1}||^2/||r^i||^2,$$

(e) calcul de la direction de descente conjuguée :

$$p^{i+1} = r^{i+1} - \beta^{i+1} p^i.$$

Nous avons montré au chapitre 2 que les méthodes de décomposition de domaine sans recouvrement (Schur primal, Schur Dual, méthode des joints) reviennent à résoudre le système d'interface

$$SX_3 = G,$$

qui s'écrit de manière développée :

$$[C_{33} - C_{31}C_{11}^{-1}C_{13} - C_{32}C_{22}^{-1}C_{23}]X_3 = D_3 - C_{31}C_{11}^{-1}D_1 - C_{32}C_{22}^{-1}D_2.$$

En adaptant les étapes précédentes, la méthode de gradient conjugué s'écrit comme suit. Les étapes indiquées en caractères gras seront implémentées informatiquement.

1. Initialisations
   (a) $\mathbf{X_3^0}$ **donné**,
   (b) on calcule

$$\begin{aligned} p^0 &= r^0 \\ &= G - SX_3^0 \\ &= (D_3 - \textstyle\sum_{k=1,2} C_{3k}C_{kk}^{-1}D_k) - \Big[- \sum_{k=1,2} C_{3k}C_{kk}^{-1}C_{k3}X_3^0 + C_{33}\Big] X_3^0, \\ &= D_3 - \textstyle\sum_{k=1,2} C_{3k}C_{kk}^{-1}[D_k - C_{k3}X_3^0] - C_{33}X_3^0. \end{aligned} \tag{D.2}$$

On remarque que le terme entre crochet correspond à des résolutions sur les sous-domaines :

$$D_k - C_{k3}X_3^0 = C_{kk}X_k, \tag{D.3}$$

on en déduit

$$\begin{aligned} p^0 = r^0 &= D_3 - \textstyle\sum_{k=1,2} C_{3k}C_{kk}^{-1}C_{kk}X_k - C_{33}X_3^0 \\ &= D_3 - \textstyle\sum_{k=1,2} C_{3k}X_k - C_{33}X_3^0. \end{aligned} \tag{D.4}$$

Les vecteurs p^0 et r^0 sont de la même dimension que X_3^0.
D'un point de vue informatique, on procède comme suit (sur chaque processeur $k = 1, 2$) :

i. **calcul de $\mathbf{X_k}$ (variable locale) par résolution de**

$$\begin{aligned} &\mathbf{C_{kk}X_k = D_k - C_{k3}X_3^0}, \\ &\xi_\mathbf{k} = \mathbf{C_{3k}X_k}, \end{aligned} \tag{D.5}$$

ii. **calcul du résidu :**

$$\mathbf{p^0 = r^0 = D_3 - C_{33}X_3^0 - \xi_1 - \xi_2}, \tag{D.6}$$

(c) **calcul de l'erreur :**

$$\epsilon^0 = ||\mathbf{r^0}||; \tag{D.7}$$

2. Itérations, pour $i = 0, 1, ...$

(a) Dans le **calcul du pas de descente**, on veut évaluer Sp^i par la formule

$$Sp^i = (C_{33} - C_{31}C_{11}^{-1}C_{13} - C_{32}C_{22}^{-1}C_{23})p^i \tag{D.8}$$

sans inverser les matrices C_{kk} ($k = 1, 2$). Pour le faire, on utilise des variables intermédiaires z_k ($k = 1, 2$) telles que

$$z_k = C_{kk}^{-1}C_{k3}p^i \tag{D.9}$$

i.e. telles que z_k soient **solutions des problèmes locaux :**

$$\mathbf{C_{kk}z_k = C_{k3}p^i}. \tag{D.10}$$

Une fois ces calculs locaux réalisés, on pose

$$\begin{cases} \xi_\mathbf{k} = \mathbf{C_{3k}z_k}, \quad \mathbf{k = 1, 2}, \\ \xi_\mathbf{3} = \mathbf{C_{33}p^i}, \end{cases} \tag{D.11}$$

et on évalue

$$\psi = \mathbf{SX_3^i} = \xi_\mathbf{3} - \xi_\mathbf{1} - \xi_\mathbf{2}. \tag{D.12}$$

Le pas de descente est

$$\alpha^\mathbf{i} = ||\mathbf{r^i}||^\mathbf{2}/(\psi, \mathbf{p^i}), \tag{D.13}$$

(b) **calcul de la direction de descente :**

$$\mathbf{X_3^{i+1} = X_3^i} + \alpha^\mathbf{i}\mathbf{p^i}, \tag{D.14}$$

(c) **calcul du résidu :**

$$\mathbf{r^{i+1} = r^i} - \alpha^\mathbf{i}\psi, \tag{D.15}$$

(d) **calcul de l'erreur :**

$$\beta^\mathbf{i+1} = ||\mathbf{r^{i+1}}||^\mathbf{2}/||\mathbf{r^i}||^\mathbf{2}, \tag{D.16}$$

(e) **calcul de la direction de descente conjuguée :**

$$\mathbf{p^{i+1} = r^{i+1}} - \beta^\mathbf{i+1}\mathbf{p^i}. \tag{D.17}$$

L'algorithme que nous venons de présenter résout un système d'interface global, c'est-à-dire qu'il traite simultanément toutes les interfaces d'une décomposition donnée. Dans la suite, nous nous attachons à étudier le cas où toutes les interfaces entre sous-domaines sont traitées de manière disjointe. C'est le cas par exemple de la méthode de Schur dual et de la méthode de Schur primal lorsque le domaine est décomposé en "tranches". On veut montrer que, dans ce cas particulier, nous avons plusieurs systèmes d'interface. Si l'on note N_i le nombre d'interfaces (disjointes) d'une décomposition donnée, nous avons donc

$$S_{\mathcal{I}} X_{\mathcal{I}} = G_{\mathcal{I}}, \quad \mathcal{I} = 1, ..., N_i$$

Le processus de résolution se décompose alors en un ensemble de calculs d'interface pouvant être effectués indépendamment les uns des autres. Cela nous permet d'exprimer l'algorithme CG générique pour un sous-domaine Ω_k ayant un nombre quelconque d'interfaces, ce qui facilite l'identification des tâches relatives au processus de résolution sur un sous-domaine.

Pour illustrer nos propos, prenons comme exemple la décomposition d'un domaine rectangulaire en 3 "tranches" non recouvrantes. Si l'on réorganise le système linéaire résultant de façon à séparer les nœuds internes des nœuds situés aux interfaces, nous avons le système suivant :

$$C = \begin{pmatrix} C_{11} & 0 & 0 & C_{14} & 0 \\ 0 & C_{22} & 0 & C_{24} & C_{25} \\ 0 & 0 & C_{33} & 0 & C_{35} \\ C_{41} & C_{42} & 0 & C_{44} & 0 \\ 0 & C_{52} & C_{53} & 0 & C_{55} \end{pmatrix} \quad X = \begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \end{pmatrix} \quad D = \begin{pmatrix} D_1 \\ D_2 \\ D_3 \\ D_4 \\ D_5 \end{pmatrix}$$

Cette formulation matricielle nous donne les systèmes d'interface

$$S_4 X_4 = G_4 \quad \text{et} \quad S_5 X_5 = G_5$$

où

$$S_4 = C_{44} - \sum_{k=1}^{2} C_{4k} C_{kk}^{-1} C_{k4}, \quad G_4 = D_4 - \sum_{k=1}^{2} C_{4k} C_{kk}^{-1} D_k$$

et

$$S_5 = C_{55} - \sum_{k=2}^{3} C_{5k} C_{kk}^{-1} C_{k5}, \quad G_5 = D_5 - \sum_{k=2}^{3} C_{5k} C_{kk}^{-1} D_k.$$

Rappelons que les matrices S_4 et S_5 sont composées des blocs de la matrice globale ayant une influence sur les interfaces respectives. Les vecteurs G_4 et G_5 sont décomposés de façon analogue.

Ces notations mettent en évidence que les calculs de X_4 et X_5 sont indépendants l'un de l'autre. L'algorithme 9 fournit une écriture générique du gradient conjugué pour K sous-domaines lorsque les interfaces sont disjointes (résolution de plusieurs systèmes d'interface au lieu d'un seul).

Algorithme 9 : Algorithme CG générique pour le sous-domaine $\Omega_k, k = 1, ..., K.$

1: $X_{\mathcal{I}}^0 = 0, \quad \forall \mathcal{I} = 1, ..., N_i$ {initialisation}
2: X_k solution de $C_{kk}X_k = D_k - \sum_{\mathcal{I}=1}^{N_i} C_{k\mathcal{I}}X_{\mathcal{I}}^0$
3: $\xi_{\mathcal{I}(k)} = C_{\mathcal{I}k}X_k, \quad \forall \mathcal{I} = 1, ..., N_i$
4: $\boxed{p_{\mathcal{I}}^0 = r_{\mathcal{I}}^0 = D_{\mathcal{I}} - C_{\mathcal{I}\mathcal{I}}X_{\mathcal{I}}^0 - \xi_{\mathcal{I}(k)} - \xi_{\mathcal{I}(l)}}, \quad \forall \mathcal{I} = 1, ..., N_i$
5: $\boxed{\epsilon^0 = \sum_{\mathcal{I}=1}^{N_i} \|r_{\mathcal{I}}^0\|^2}$

6: **pour** $i = 0, 1, ...$ **jusqu'à** la convergence **faire**
7: z_k solution de $C_{kk}z_k = \sum_{\mathcal{I}=1}^{N_i} C_{k\mathcal{I}}p_{\mathcal{I}}^i$
8: $\xi_{\mathcal{I}(k)} = C_{\mathcal{I}k}z_k, \quad \forall \mathcal{I} = 1, ..., N_i$
9: $\xi_{\mathcal{I}} = C_{\mathcal{I}\mathcal{I}}p_{\mathcal{I}}^i, \quad \forall \mathcal{I} = 1, ..., N_i$
10: $\boxed{\psi_{\mathcal{I}} = \xi_{\mathcal{I}} - \xi_{\mathcal{I}(k)} - \xi_{\mathcal{I}(l)}}, \quad \forall \mathcal{I} = 1, ..., N_i$
11: $\boxed{\alpha^i = \sum_{\mathcal{I}=1}^{N_i} \|r_{\mathcal{I}}^i\|^2 / (\psi_{\mathcal{I}}, p_{\mathcal{I}}^i)}$
12: $X_{\mathcal{I}}^{i+1} = X_{\mathcal{I}}^i + \alpha^i p_{\mathcal{I}}^i, \quad \forall \mathcal{I} = 1, ..., N_i$
13: $r_{\mathcal{I}}^{i+1} = r_{\mathcal{I}}^i - \alpha^i \psi_{\mathcal{I}}, \quad \forall \mathcal{I} = 1, ..., N_i$
14: $\boxed{\beta^{i+1} = \sum_{\mathcal{I}=1}^{N_i} \|r_{\mathcal{I}}^{i+1}\| / \|r_{\mathcal{I}}^i\|^2}$
15: $p_{\mathcal{I}}^{i+1} = r_{\mathcal{I}}^{i+1} - \beta^{i+1} p_{\mathcal{I}}^i, \quad \forall \mathcal{I} = 1, ..., N_i$
16: **fin pour**

Bibliographie

[1] Amestoy (P. R.), Duff (I. S.) et L'Excellent (J. Y.). – *A fully asynchronous multifrontal solver using distributed dynamic scheduling*. – Rapport technique RT/APO/99/2, ENSEEIHT-IRIT, 1999.

[2] Amestoy (P. R.), Duff (I. S.), L'Excellent (J. Y.) et Li (X. S.). – *Analysis, Tuning and Comparison of Two General Sparse Solvers for Distributed Memory Computers*. – Rapport technique RT/APO/00/3, ENSEEIHT-IRIT, 2000.

[3] Amitai (D.), Averbuch (A.), Israeli (M.), Itzikowitz (S.) et Turkel (E.). – A survey of asynchronous finite-difference methods for parabolic pdes on multiprocessors. *Applied Numerical Mathematics*, vol. 12, 1993, pp. 27–45.

[4] Amza (C.), Cox (A. L.), Dwarkadas (S.), Keleher (P.), Lu (H.), Rajamony (R.), Yu (W.) et Zwaenepoel (W.). – Treadmarks : Shared memory computing on networks of workstations. *IEEE Computer*, vol. 29, n2, février 1996, pp. 18–28.

[5] Ashcraft (C.) et Grimes (R. G.). – SPOOLES : An object oriented sparse matrix library. *In : Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. – mars 1999.

[6] Bahi (J.). – Algorithmes parallèles asynchrones pour des systèmes singuliers. *In : C.R. l'Acad. Sci. Sér. 326*, pp. 1421–1425. – 1998.

[7] Bahi (J.), Miellou (J.-C.) et Rhofir (K.). – Asynchronous multisplitting methods for nonlinear fixed point problems. *Numerical Algorithms*, vol. 15, 1997, pp. 315–345.

[8] Bal (H. E.), Bhoedjang (R.), Hofman (R.), Jacobs (C.), Langendoen (K.) et Ruhl (T.). – Performance evaluation of the Orca shared-object system. *ACM Trans. on Computer Systems*, vol. 16, n1, février 1998, pp. 1–40.

[9] Bal (H. E.), Kaashoek (M. F.) et Tanenbaum (A. S.). – Orca : A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, vol. 18, n3, mars 1992, pp. 190–205.

[10] Balay (S.), McInnes (W. Gropp L. C.) et Smith (B.). – Efficient management of parallelism in object-oriented numerical software libraries. *In : Modern*

Software Tools in Scientific Computing, éd. par Arge (E.), Bruaset (A. M.) et Langtangen (H. P.). – Birkhauser Press, 1997.

[11] Bastian (P.) et Wittum (G.). – Adaptive multigrid methods : The UG concept. *In : Adaptive Methods – Algorithms, Theory and Applications*. pp. 17–37. – Braunschweig, 1994.

[12] Baudet (Gérard M.). – Asynchronous iterative methods for multiprocessors. *Journal of the ACM*, vol. 25, n2, avril 1978, pp. 226–244.

[13] Ben-Abdallah (A.). – *Méthode de projection pour la simulation de grandes structure turbulentes sur calculateurs parallèles*. – Thèse de doctorat, Université Pierre et Marie Curie, Paris, 1998.

[14] Ben-Abdallah (A.), Charão (A. S.), Charpentier (I.) et Plateau (B.). – Ahpik : A Parallel Multithreaded Framework Using Adaptivity and Domain Decomposition Methods for Solving PDE Problems. *In : 13th International Conference on Domain Decomposition Methods*. – Lyon, France, october 2000. A paraître.

[15] Ben-Belgacem (F.). – *The mortar element method with Lagrange multipliers*. – Rapport technique 94-1, Mathématiques pour l'Industrie et la Physique, Université Paul Sabatier, 1994.

[16] Bernard (Pierre-Eric), Gautier (Thierry) et Trystram (Denis). – Large scale simulation of parallel molecular dynamics. *In : Proceedings of Second Merged Symposium IPPS/SPDP 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing*. – San Juan, Puerto Rico, avril 1999.

[17] Bernardi (Christine), Maday (Yvon) et Patera (Anthony T.). – A new non conforming approach to domain decomposition : The mortar element method. *In : Collège de France Seminar*, éd. par Brezis (Haim) et Lions (Jacques-Louis). – Pitman, 1994.

[18] Bernaschi (M.). – Efficient message passing on shared memory multiprocessors. *Lecture Notes in Computer Science*, vol. 1156, 1996.

[19] Bertsekas (D. P.). – Distributed asynchronous computation of fixed points. *Math. Programming*, vol. 27, 1983, pp. 107–120.

[20] Bertsekas (D. P.) et Tsitsiklis (J. N.). – Some aspects of parallel and distributed iterative algorithms – A survey. *Automatica*, vol. 27, 1991, pp. 3–21.

[21] Bertsekas (Dimitri P.) et Tsitsiklis (John N.). – *Parallel and Distributed Computation : Numerical Methods*. – Prentice-Hall Inc., 1989.

[22] Birken (K.) et Rühle (R.). – Dynamic distributed data : Efficient, portable and easy to use. *In : Parallel Computing : State-of-the-Art and Perspectives, Proceedings of the Conference ParCo'95, 19-22 September 1995, Ghent, Bel-*

gium, éd. par D'Hollander (E. H.), Joubert (G. R.), Peters (F. J.) et Trystram (D.). pp. 303–310. – Amsterdam, février 1996.

[23] Blelloch (Guy E.). – *NESL : A Nested Data-Parallel Language (Version 3.1)*. – Rapport technique CMU-CS-95-170, Computer Science Department, Carnegie Mellon University, septembre 1995.

[24] Blumofe (R. D.), Joerg (C. F.), Kuszmaul (B. C.), Leiserson (C. E.), Randall (K. H.), Zhou (Y. C. E.), Randall (K. H.) et Zhou (Y.). – Cilk : an efficient multithreaded runtime system. *ACM SIGPLAN Notices*, vol. 30, n8, août 1995, pp. 207–216.

[25] Brezzi (F.). – On the existence, uniqueness and approximation of saddle-point problems arising from lagrangian multipliers. *Revue Franc. Automat. Inform. Rech. Operat.*, vol. 8, nR-2, 1974, pp. 129–151.

[26] Briat (J.), Ginzburg (I.) et Pasin (M.). – ATHAPASCAN-0B : un noyau exécutif parallèle. *Lettre du Calculateur Parallèle*, vol. 10, n3, 1998, pp. 273–293.

[27] Bruaset (Are Magnus) et Langtangen (Hans Petter). – Object-oriented design of preconditioned iterative methods in Diffpack. *ACM Transactions on Mathematical Software*, vol. 23, n1, mars 1997, pp. 50–80.

[28] Buchholz (P.), Fischer (M.) et Kemper (P.). – Distributed steady state analysis using kronecker algebra. *In : Numerical Solution of Markov Chains (NSMC'99)*, éd. par Plateau (B.) et Stewart (W. J.). pp. 76–95. – Prensas Universitarias de Zaragoza, 1999.

[29] Bugge (Hakon O.) et Husoy (Per O.). – Efficient SAR processing on the SCALI system. *In : International Parallel Processing Symposium*. – 1997.

[30] Butenhof (David R.). – *Programming with POSIX threads*. – Reading, MA, USA, Addison-Wesley, 1997, 381p.

[31] Cai (X.). – Two object-oriented approaches to the parallelization of Diffpack. *In : Proceedings of the HiPer'99 Conference*. – Tromsø, Norway, 1999.

[32] Carissimi (Alexandre). – *Le noyau exécutif Athapascan-0 et l'exploitation de la multiprogrammation légère sur les grappes de stations multiprocesseurs*. – Thèse de doctorat en informatique, Institut National Polytechnique de Grenoble, France, novembre 1999.

[33] Caromel (D.), Belloncle (F.) et Roudier (Y.). – The C++// language. *In : Parallel Programming Using C++*, éd. par Wilson (G.) et Lu (P.), pp. 257–296. – Cambridge (MA), USA, MIT Press, 1996.

[34] Caromel (Denis). – Towards a method of object-oriented concurrent programming. *Communications of the ACM*, vol. 36, n9, 1993, pp. 90–102.

[35] Caromel (Denis) et Vayssière (Julien). – A Java framework for seamless sequential, multi-threaded, and distributed programming. *In : Proceedings*

of the ACM Workshop on Java for High-Performance Network Computing. – mars 1998.

[36] Carriero (Nicholas) et Gelernter (David). – How to write parallel programs : A guide for the perplexed. *ACMCS*, vol. 21, n3, septembre 1989, pp. 323–357.

[37] Chan (T.) et V.Eijkhout. – *Design of a library of parallel preconditioners*. – Rapport technique 97-58, UCLA CAM, 1997.

[38] Chan (Tony F.) et Mathew (Tarek P.). – Domain decomposition algorithms. *In : Acta Numerica 1994*, pp. 61–143. – Cambridge University Press, 1994.

[39] Charão (A. S.), Charpentier (I.) et Plateau (B.). – Un environnement modulaire pour l'exploitation des processus légers dans les méthodes de décomposition de domaine. *In : 11ème Rencontres francophones du parallélisme, des architectures et des systèmes*, éd. par Pazat (J.-L.) et Quinton (P.), pp. 145–150. – Rennes, France, june 1999.

[40] Charão (A. S.), Charpentier (I.) et Plateau (B.). – A framework for parallel multithreaded implementation of domain decomposition methods. *In : Parallel Computing : Fundamentals and Applications*, éd. par D'Hollander (E. H.), Joubert (G. R.), Peters (F. J.) et Sips (H. J.). pp. 95–102. – Imperial College Press, 2000.

[41] Charão (A. S.), Charpentier (I.) et Plateau (B.). – Programmation par objet et utilisation de processus légers pour les méthodes de décomposition de domaine. *Technique et Science Informatiques*, vol. 19, n5, 2000.

[42] Chazan (D.) et Miranker (W.). – Chaotic relaxation. *Linear Algebra and Its Applications*, vol. 2, 1969, pp. 199–222.

[43] Christaller (M.). – *Vers un Support d'Exécution Portable pour Applications Parallèles Irrégulières : Athapascan-0*. – Thèse de doctorat, Université Joseph Fourier, Grenoble I, novembre 1996.

[44] Christaller (M.), Briat (J.) et Rivière (M.). – Athapascan-0 : concepts structurants simples pour une programmation parallèle efficace. *Calculateurs Parallèles*, vol. 7, n2, 1995, pp. 173–196.

[45] Ciarlet (P. G.). – *Introduction à l'analyse matricielle et à l'optimisation*. – Masson, 1994, *Collection Mathématiques Appliquées pour la Maîtrise*.

[46] Cosnard (M.) et Robert (Y.). – Algorithmique parallèle : une étude de complexité. *Technique et Sciences Informatiques*, 1987.

[47] Craig (R.) et Bampton (M. C. C.). – Coupling of substructures for dynamic analysis. *AIAA J.*, vol. 4, 1968, pp. 1313–1321.

[48] da Cunha (Rudnei Dias) et Hopkins (Tim). – The Parallel Iterative Methods (PIM) package for the solution of systems of linear equations on parallel computers. *Applied Numerical Mathematics : Transactions of IMACS*, vol. 19, n1–2, décembre 1995, pp. 33–50.

[49] Dagum (Leonardo) et Menon (Ramesh). – OpenMP : An industry-standard API for shared-memory programming. *IEEE Computational Science & Engineering*, vol. 5, n1, janvier/mars 1998, pp. 46–55.

[50] de Oliveira Stein (B.) et Chassin de Kergommeaux (J.). – Interactive visualisation environment of multi-threaded parallel programs. *In : Parallel Computing : Fundamentals, Applications and New Directions*. pp. 311–318. – Elsevier, 1998.

[51] de Oliveira Stein (Benhur). – *Visualisation interactive et extensible de programmes parallèles à base de processus légers*. – Thèse de doctorat, Université Joseph Fourier, France, octobre 1999.

[52] Demmel (James W.), Gilbert (John R.) et Li (Xiaoye S.). – An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, vol. 20, n4, octobre 1999, pp. 915–952.

[53] Denneulin (Y.). – Granularity and on-line scheduling of branch & bound tasks. *In : 12th Conference of the European Chapter on Combinatorial Optimization*. – 1999.

[54] Diekmann (R.), Dralle (U.), Neugebauer (F.) et Romke (T.). – PadFEM : A portable parallel FEM-tool. *In : High-Performance Computing and Networking (HPCN'96 Europe)*, éd. par Liddell (H.), Colbrook (A.), Hertzberge (B.) et Sloot (P.), pp. 580–585. – Springer-Verlag, 1996.

[55] Dongarra (J.), Lumsdaine (A.), Pozo (R.) et Remington (K.). – A sparse matrix library in C++ for high performance architectures. *In : Proceedings of the 2nd Object Oriented Numerics Conference*, pp. 214–218. – 1992.

[56] Dongarra (J. J.), Duff (I. S.), Sorensen (D. C.) et van der Vorst (H. A). – *Numerical Linear Algebra for High-Performance Computers*. – SIAM, 1998.

[57] Dongarra (Jack J.), Meuer (Hans W.) et Strohmaier (Erich). – TOP500 supercomputer sites, 11th edition. *In : Supercomputing 2000*. – Dallas,USA, novembre 2000.

[58] Doreille (Mathias). – *Athapascan-1 : vers un modèle de programmation parallèle adapté au calcul scientifique*. – Thèse de doctorat, Institut National Polytechnique de Grenoble, 1999.

[59] Duff (I. S.), Erisman (A. M.) et Reid (J. K.). – *Direct Methods for Sparse Matrices*. – Oxford, UK, Clarendon Press, 1986, 341p.

[60] Eijkhout (Victor). – Overview of iterative linear system solver packages. *NHSE Review*, vol. 3, n 1, 1998. – http ://www.nhse.org/NHSEreview/98-1.html.

[61] El-Baz (D.), Gazen (D.), Miellou (J.-C.) et Spiteri (P.). – Mise en œuvre de méthodes itŕatives asynchrones avec communication flexible – aplication à la résolution d'une classe de problèmes d'optimisation. *Calculateurs Parallèles*, vol. 8, 1996, pp. 393–410.

[62] Ellis (M. A.) et Stroustrup (B.). – *The Annotated C++ Reference Manual*. – Addison-Wesley, 1990.

[63] Erlichson (Andrew), Nuckools (Neal), Chesson (Greg) et Hennessy (John). – Softflash : Analysing the performance of clustered distributed virtual shared memory. *In : Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII), Computer Architecture News*, pp. 210–220. – octobre 1996.

[64] Farhat (C.) et Roux (F.-X.). – Implicit parallel processing in structural mechanics. *Computational Mechanics Advances*, vol. 2, 1994, pp. 1–124.

[65] Farhat (Charbel) et Roux (Francois-Xavier). – A Method of Finite Element Tearing and Interconnecting and its Parallel Solution Algorithm. *Int. J. Numer. Meth. Engng.*, vol. 32, 1991, pp. 1205–1227.

[66] Feautrier (Paul). – Compiling for massively parallel architectures : a perspective. *Microprogramming and Microprocessors*, vol. 41, 1995, pp. 425–439.

[67] Fink (S. J.), Baden (S. B.) et Kohn (S. R.). – Efficient run-time support for irregular block-structured applications. *Journal of Parallel and Distributed Computing*, vol. 50, n1, 1998, pp. 61–82.

[68] Flaherty (J.), Loy (R.), Ozturan (C.), Shephard (M.), Szymanski (B.), Teresco (J.) et Ziantz (L.). – *Parallel structures and dynamic load balancing for adaptive nite element computation*. – Rapport technique SCOREC 22-1996, Sci. Comp. Res. Ctr., Rensselaer Polytechnic Institute, 1996.

[69] Flynn (Michael J.). – Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, vol. C-21, n9, septembre 1972, pp. 948–960.

[70] Forum (Message Passing Interface). – MPI : a message-passing interface standard. *International Journal of Supercomputer Applications and High Performance Computing*, vol. 8, n3/4, 1994, pp. 159–416.

[71] Foster (Ian), Kesselman (Carl) et Tuecke (Steven). – The Nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, vol. 37, n1, 1996, pp. 70–82.

[72] Fox (Geoffrey C.) et Furmanski (Wojtek). – Java for parallel computing and as a general language for scientific and engineering simulation and modeling. *Concurrency : Practice and Experience*, vol. 9, n6, juin 1997, pp. 415–425.

[73] Frommer (A.). – *Parallele asynchrone iterationen*, chap. 4, pp. 187–231. – Berlin, Wissenschaftliches Rechnen, Akademie, 1995.

[74] Frommer (A.), Schwandt (H.) et Szyld (D. B.). – Asynchronous weighted additive Schwarz methods. *Electron. Trans. Numer. Anal.*, vol. 5, 1997, pp. 48–61.

[75] Frommer (Andreas) et Szyld (Daniel B.). – On asynchronous iterations. *Journal of Computational and Applied Mathematics*, vol. 23, 2000, pp. 201–216.

[76] Fu (C.), Jiao (X.) et Yang (T.). – Efficient sparse LU factorization with partial pivoting on distributed memory architectures. *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, n2, février 1998, pp. 109– ? ?

[77] Furmento (Nathalie), Roudier (Yves) et Siegel (Günther). – *Parallélisme et Distribution en C++. Une revue des langages existants.* – Rapport technique RR 95-02, Sophia Antipolis, France, I3S, 1995.

[78] Galilée (F.), Roch (J. L.), Cavalheiro (G. H.) et Doreille (M.). – Athapascan-1 : On-line building data flow graph in a parallel language. *In : Pact'98*. – 1998.

[79] Gautier (T.). – *Calcul formel et parallélisme : Conception du Systéme Givaro et Applications au Calcul dans les Extensions Algébriques*. – Thèse de doctorat, Institut National Polytechnique de Grenoble, 1996.

[80] Gautier (T.), Roch (J-L.) et Villard (G.). – Regular versus irregular problems and algorithms. *In : Proc. of IRREGULAR'95, Lyon, France*. – Springer-Verlag, Septembre 1995.

[81] Gautier (Thierry). – *Calcul formel et parallélisme : Conception du Système Givaro et Applications au Calcul dans les Extensions Algébriques*. – Thèse de doctorat, Institut National Polytechnique de Grenoble, France, Juin 1996.

[82] George (Paul Louis). – *Génération Automatique de Maillages. Applications aux Méthodes d'Élements Finis*. – Paris, Masson, 1991, *Collection Recherches en Mathématiques Appliquées*.

[83] Girault (Vivette) et Pierre-Arnaud Raviart. – *Finite Element Methods for Navier-Stokes Equations*. – New York, Springer-Verlag, 1986.

[84] Golub (Gene H.) et Loan (Charles F. Van). – *Matrix computations*. – Baltimore, MD, USA, The Johns Hopkins University Press, 1996, third édition, *Johns Hopkins Studies in the Mathematical Sciences*, 698p.

[85] Gropp (William), Lusk (Ewing) et Thakur (Rajeev). – *Using MPI-2 : Advanced Features of the Message-Passing Interface*. – Cambridge, MA, MIT Press, 1999.

[86] Guinand (F.). – *Ordonnancement avec Communications pour Architectures Multiprocesseurs dans Divers Modèles d'Exécution*. – Thèse de doctorat, Institut National Polytechnique de Grenoble, juin 1995.

[87] Guivarch (R.). – *Résolution parallèle de problèmes aux limites couplés par des méthodes de sous-domaines synchrones et asynchrones*. – Thèse de doctorat, Institut National Polytechnique de Toulouse, 1997.

[88] Gupta (Anshul), Karypis (George) et Kumar (Vipin). – Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, n5, mai 1997, pp. 502–520.

[89] Heath (M. T.), Ng (E.) et Peyton (B. W.). – Parallel algorithms for sparse linear systems. *SIAM Review*, vol. 33, n3, 1991, pp. 420–460.

[90] Heath (M. T.) et Raghavan (P.). – Performance of a fully parallel sparse solver. *The International Journal of Supercomputer Applications and High Performance Computing*, vol. 11, n1, Spring 1997, pp. 49–64.

[91] Hendrickson (Bruce) et Leland (Robert). – A multilevel algorithm for partitioning graphs. *In : Proc. Supercomputing 95*. – 1995.

[92] Henon (P.), Ramet (P.) et Roman (J.). – A mapping and scheduling algorithm for parallel sparse fan-in numerical factorization. *In : EuroPar'99 Parallel Processing*. pp. 1059–1067. – Springer-Verlag, 1999.

[93] Hurty (W. C.). – Dynamic analysis of structural systems using component models. *AIAA J.*, vol. 4, 1965, pp. 678–685.

[94] Hutcheson (G. Dan) et Hutcheson (Jerry D.). – Technology and economics in the semiconductor industry. *Scientific American*, vol. 274, n1, janvier 1996, pp. 40–46.

[95] Hutchinson (S. A.) et al. – *Aztec User's Guide : Version 2.0*. – Rapport technique, Sandia National Laboratories, 1998.

[96] IEEE. – *IEEE Standard for Multithreaded Programming POSIX.1c*. – IEEE Computer Society Press, 1995.

[97] Ishikawa (Y.) et al. – RWC massively parallel software environment and an overview of MPC++. *In : Proceedings of Workshop on Parallel Symbolic Languages and Systems*. – 1995.

[98] Joerg (C. F.). – *The Cilk system for parallel multithreaded computing*. – Cambridge, MA, USA, Thesis (ph.d.), Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1996, 199p.

[99] Jones (M. T.) et Plassmann (P. E.). – *BlockSolve95 users manual : Scalable library software for the solution of sparse linear systems*. – Rapport technique ANL-95/48, Argonne National Lab., 1995.

[100] Kafura (Dennis) et Huang (Liya). – MPI++ : A C++ language binding for MPI. *In : MPI Developers Conference*. – University of Notre Dame, juin 1995.

[101] Karypis (George) et Kumar (Vipin). – A fast and high quality multilevel scheme for partitionning irregular graphs. *SIAM Journal of Scientific Computing*, 1997.

[102] Keyes (D.). – Trends in algorithms for nonuniform applications on hierarchical distributed architectures. *In : Proceedings of the Workshop on Computational Aerosciences in the 21st Century*, éd. par Salas (M. D.) et Anderson (W. K.). pp. 103–137. – Kluwer, 1999.

[103] Kleiman (Steve), Shah (Devang) et Smaalders (Bart). – *Programming With Threads*. – Mountain View, CA, USA, SunSoft Press, 1995, 534p.

[104] Krakowiak (S.). – *Systemes d'exploitation : principes et fonctions*. – Techniques de l'ingénieur, 1996.

[105] Lascaux (P.) et Théodor (R.). – *Analyse numérique matricielle appliquée à l'art de l'ingénieur : Méthodes itératives*. – Masson, 1987.

[106] Launay (Pascale) et Pazat (Jean-Louis). – *A Framework for Parallel Programming in Java*. – Rapport technique RR-3319, INRIA, 1997.

[107] Lewis (Bil) et Berg (Daniel J.). – *Threads primer : a guide to multithreaded programming*. – Mountain View, CA, USA, SunSoft Press, 1996, 319p.

[108] Lewis (Bil) et Berg (Daniel J.). – *Multithreaded programming with pthreads*. – Mountain View, CA, USA, Sun Microsystems, 1998, 382p.

[109] Li (X. S.) et Demmel (J. W.). – A scalable sparse direct solver using static pivoting. *In : Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. – mar 1999.

[110] Lions (Pierre Louis). – On the Schwarz alternating method. I. *In : First International Symposium on Domain Decomposition Methods for Partial Differential Equations*, éd. par Glowinski (Roland), Golub (Gene H.), Meurant (Gérard A.) et Périaux (Jacques). – Philadelphia, PA, USA, 1988.

[111] Lubachevsky (B.) et Mitra (D.). – A chaotic asynchronous algorithm for computing the fixed point of a nonnegative matrix of unit spectral radius. *Journal of the ACM*, vol. 33, 1986, pp. 130–150.

[112] Lucquin (B.) et Pironneau (O.). – *Introduction au Calcul Scientifique*. – Paris, Masson, 1996.

[113] Lumetta (Steven S.), Mainwaring (Alan) et Culler (David E.). – Multi-protocol active messages on a cluster of SMPs. *In : Proceedings of Supercomputing'97 (CD-ROM)*. – San Jose, CA, novembre 1997.

[114] Lusk (E. L.) et Gropp (W. W.). – A taxonomy of programming models for symmetric multiprocessors and SMP clusters. *In : Proceedings of Programming Models for Massively Parallel Computers*, pp. 2–7. – Dallas,USA, 1995.

[115] Maillard (N.), Roch (J.-L.) et Valiron (P.). – Parallélisation du calcul ab-initio de l'énergie de corrélation électronique. *In : RenPar'9 — 9èmes rencontres francophones du parallélisme*, p. 45. – Lausanne, Suisse, mai 1997.

[116] McManus (Kevin). – *A Strategy for Mapping Unstructured Mesh Computational Mechanics Programs onto Distributed Memory Parallel Architectures*. – University of Greenwich, Wellington St., London, SE18 6PF, Thèse de doctorat, Computing and Mathematical Science, mars 1996.

[117] Miellou (J.-C.). – Algorithmes de relaxation chaotique à retards. *Revue Francaise d'Automatique, d'Informatique et Recherche Op'erationnelle (RAIRO)*, avril 1975, pp. 55–82.

[118] Miellou (J. C.), El-Baz (D.) et Spitéri (R.). – A new class of asynchronous iterative algorithms with order intervals. *Mathematics of Computation*, vol. 67, 1998, pp. 237–255.

[119] Namyst (R.) et Méhaut (J.-F.). – PM^2 : Parallel multithreaded machine. A computing environment for distributed architectures. *In : Parallel Computing : State-of-the-Art and Perspectives, Proceedings of the Conference ParCo'95, 19-22 September 1995, Ghent, Belgium*, éd. par D'Hollander (E. H.), Joubert (G. R.), Peters (F. J.) et Trystram (D.). pp. 279–285. – Amsterdam, février 1996.

[120] Oaks (Scott) et Wong (Henry). – *Java Threads*. – Newton, MA, USA, O'Reilly & Associates, Inc., 1999, 319p.

[121] O'Leary (Dianne P.) et White (R. E.). – Multisplittings of matrices and parallel solution of linear systems. *SIAM Journal on Algebraic and Discrete Methods*, vol. 6, n4, 1985, pp. 630–640.

[122] Ozturan (Can). – *Distributed Environment and Load Balancing for Adaptive Unstructured Meshes*. – Thèse de doctorat, Computer Science Department, Rensselaer Polytechnic Institute, 1995.

[123] Parashar (M.). – Scalable distributed dynamic grids : A survey of existing support. – http ://www.ticam.utexas.edu/ parashar/Papers/survey/.

[124] Parashar (M.) et Browne (J. C.). – *DAGH : A data-management infrastructure for parallel adaptive refinement techniques*. – Rapport technique, Department of Computer Science, University of Texas at Austin, 1995.

[125] Parsons (R.) et Quinlan (D.). – A++/P++ array classes for architecture independent finite difference computations. *In : Proceedings of the Second Annual Object-Oriented Numerics Conference*. – 1994.

[126] Pellegrini (F.) et Roman (J.). – SCOTCH : A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. *In : High-Performance Computing and Networking (HPCN'96 Europe)*, éd. par Liddell (H.), Colbrook (A.), Hertzberge (B.) et Sloot (P.), pp. 493–498. – Springer-Verlag, 1996.

[127] Philippsen (Michael) et Zenger (Matthias). – JavaParty – transparent remote objects in Java. *Concurrency : Practice and Experience*, vol. 9, n11, novembre 1997, pp. 1225–1242.

[128] Plateau (Brigitte) et al. – *Présentation d'APACHE*. – Rapport APACHE 1, Grenoble, IMAG, octobre 1993.

[129] Pott (M.). – On the convergence of asynchronous iteration methods for nonlinear paracontractions and consistent linear systems. *Linear Algebra Appl.*, vol. 283, 1998, pp. 1–33.

[130] Pozo (Roldan) et al. – *IML++ WWW Home Page*. – Disponible sur le Web à http ://gams.nist.gov/acmd/Staff/RPozo/iml++.html, 1997.

[131] Raviart (P. A.) et Thomas (J. M.). – *Introduction à l'analyse numérique des équations aux dérivées partielles*. – Masson, 1994, *Collection Mathématiques Appliquées pour la Maîtrise*.

[132] Reis (Gabriel Dos). – *Vers une nouvelle approche du calcul scientifique en C++*. – Rapport technique RR-3362, INRIA, Institut National de Recherche en Informatique et en Automatique, 1998.

[133] Reynders (J. V. W.) et al. – POOMA : A framework for scientific simulations of paralllel architectures. *In : Parallel Programming using C++*. – Cambridge, MA, 1996.

[134] Rinard (M. C.) et Lam (M. S.). – The design, implementation and evaluation of Jade. *ACM Trans. Prog. Lang. and Sys.*, vol. 20, n3, mai 1998, pp. 483–545. – Rinard's PhD Thesis, Comp Sci, Stanford 1994.

[135] Robert (F.), Charnay (M.) et Musy (F.). – Itérations chaotiques série-parallèle pour des équations non-linéares de point fixe. *Appl. Math.*, vol. 20, 1975, pp. 1–38.

[136] Roucairol (C.) et al. – *Stratageme : Une méthodologie de programmation parallèle pour les problèmes non structurés*. – Rapport technique, PRiSM, Versailles, décembre 1995.

[137] Roux (F.-X.). – Méthodes de décomposition de domaine pour des problèmes elliptiques. *Calculateurs Parallèles*, vol. 7, n3, 1995, pp. 237–253.

[138] Saad (Y.). – *Data Structures and Algorithms for Domain Decomposition and Distributed Sparse Matrix Computations*. – Rapport technique 95-014, Minneapolis, Department of Computer Science, University of Minnesota, 1995.

[139] Saad (Y.) et Malevsky (A.). – PSPARSLIB : A portable library of distributed memory sparse iterative solvers. *In : Proceedings of Parallel Computing Technologies (PaCT-95)*, éd. par M. (V. E.) et al. – St. Petersburg, Russia, septembre 1995.

[140] Saad (Y.) et Suchomel (B.). – *ARMS : An algebraic recursive multilevel solver for general sparse linear systems*. – Rapport technique UMSI-99-107, Minnesota Supercomputer Institute, University of Minnesota, Minneapolis, 1999.

[141] Saad (Yousef). – *Iterative Methods for Sparse Linear Systems*. – Boston, PWS Publishing, 1996.

[142] Samanta (R.), Bilas (A.), Iftode (L.) et Singh (J. P.). – Home-based SVM protocols for SMP clusters : Design and performance. *In : Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*. – février 1998.

[143] Scales (D. J.), Gharachorloo (K.) et Aggarwal (A.). – Fine-grain software distributed shared memory on SMP clusters. *In : Proc. of the 4th IEEE Symp. on High-Performance Computer Architecture (HPCA-4)*. – février 1998.

[144] Scales (Daniel J.) et Lam (Monica S.). – The design and evaluation of a shared object system for distributed memory machines. *In : Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI'94)*, pp. 101–114. – 1994.

[145] Schwarz (H. A.). – Über einige Abbildungsaufgaben. *Ges. Math. Abh.*, vol. 11, 1869, pp. 65–83.

[146] Shaefer (M.) et Turek (S.). – Benchmark computations of laminar flow around cylinder. *In : Flow Simulation with High-Performance Computers II*, éd. par Hirschel (E.H.). – Vieweg, 1996.

[147] Skillicorn (D. B.) et Talia (D.). – Models and Languages for Parallel Computation. *ACM Computing Surveys*, vol. 30, n2, 1998, pp. 123–169.

[148] Skillicorn (David B.). – *Foundations of parallel programming*. – Cambridge University Press, 1995, *International series on parallel computation*.

[149] Smith (B.), Bjorstad (P.) et Gropp (W.). – *Domain Decomposition : Parallel Multilevel Methods for Elliptic Partial Differential Equations*. – Cambridge University Press, 1996.

[150] Stets (R.), Dwarkadas (S.), Hardavellas (N.), Hunt (G.), Kontothanassis (L.), Parthasarathy (S.) et Scott (Michael). – Cashmere-2L : Software coherent shared memory on a clustered remote-write network. *In : Proc. of the 16th ACM Symp. on Operating Systems Principles (SOSP-16)*. – octobre 1997.

[151] Sunderam (V. S.). – PVM : a framework for parallel distributed. *Concurrency, practice and experience*, vol. 2, n4, décembre 1990, pp. 315–339.

[152] Szyld (D. B.). – Different models of parallel asynchronous iterations with overlapping blocks. *Computational Applied Mathematics*, vol. 17, 1998, pp. 101–115.

[153] Tallec (P. Le) et Tidriri (Moulay D.). – *Convergence Analysis of Domain Decomposition algorithms with full overlapping for the advection-diffusion problems*. – Rapport technique RR-2435, INRIA Rocquencourt, 1994.

[154] Tanenbaum (Andrew S.). – *Modern Operating Systems*. – Englewood Cliff, NJ, Prentice Hall, 1992.

[155] Üresin (A.) et Dubois (M.). – Sufficient condiions for the convergence of asynchronous iterations. *Parallel Computing*, vol. 10, 1989, pp. 83–92.

[156] Vajracharya (Suvas), Karmesin (Steve), Beckman (Peter), Crotinger (James), Malony (Allen), Shende (Sameer), Oldehoeft (Rod) et Smith (Stephen). – SMARTS : Exploiting temporal locality and parallelism through vertical execution. *In : Proceedings of the 1999 Conference on Supercomputing*. pp. 302–310. – ACM Press, juin 1999.

[157] van der Steen (Aad J.) et Dongarra (Jack J.). – *Overview of Recent Supercomputers*. – Rapport technique UT-CS-96-325, Department of Computer Science, University of Tennessee, avril 1996.

[158] Veldhuizen (T.). – Expression templates. *C++ Report*, vol. 7, n5, juin 1995, pp. 26–31.

[159] Veldhuizen (T.). – Arrays in Blitz++. *In : Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*. – Springer-Verlag, 1998.

[160] Verfürth (Rüdiger). – *A Review of A Posteriori Error Estimation and Adaptive Mesh-Refinement Techniques*. – Wiley and Teubner, 1996.

[161] Walshaw (C.), Cross (M.), Diekmann (R.) et Schlimbach (F.). – Multilevel mesh partitioning for optimising aspect ratio. *Lecture Notes in Computer Science*, vol. 1573, 1999, pp. 285–300.

[162] White (R. E.). – Multisplitting of a symmetric positive definite matrix. *SIAM Journal on Matrix Analysis and Applications*, vol. 11, n1, janvier 1990, pp. 69–82.

Résumé : Les applications de simulation numérique nécessitant la résolution de problèmes d'Équations aux Dérivées Partielles (EDP) sont souvent parallélisées à l'aide d'une méthode de décomposition de domaine. Ces méthodes mathématiques sont naturellement ouvertes au parallélisme, cependant leur exploitation efficace sur les machines parallèles devient difficile lorsque les applications ont un comportement irrégulier. C'est le cas par exemple lorsque les problèmes mathématiques sont résolus dans des domaines géométriques complexes ou lorsque l'on utilise des techniques d'adaptation de maillage. Une technique de programmation se prêtant bien à la mise en œuvre d'applications irrégulières est la multiprogrammation basée sur des réseaux de processus légers communicants. Dans cette thèse nous réalisons une étude approfondie de l'apport de ce paradigme de programmation à la résolution de problèmes d'EDP par des méthodes de décomposition de domaine et nous montrons qu'il existe une écriture algorithmique générique de celles-ci. Une de nos principales contributions réside dans la conception et réalisation d'un harnais informatique, appelé Ahpik, permettant une programmation aisée d'applications reposant sur les méthodes de décomposition de domaine. Ce harnais fournit un support générique adaptable à de nombreuses méthodes mathématiques, qu'elles soient synchrones ou asynchrones, avec ou sans recouvrement. Une conception orientée objet permet d'encapsuler les détails de gestion des processus légers et des communications, ce qui facilite l'implantation de nouvelles méthodes. Nous avons utilisé l'environnement Ahpik dans le cadre de la résolution de problèmes d'EDP classiques et notamment pour un problème en mécanique de fluides de grande taille.

Mots clés : Multiprogrammation légère distribuée, méthodes de décomposition de domaine, résolution d'Équations aux Dérivées Partielles en parallèle, programmation générique, grappes de multiprocesseurs.

Title: Generic parallel multithreaded programming of domain decomposition methods

Abstract: Numerical simulation applications requiring the resolution of Partial Differential Equation (PDE) problems are often parallelized using domain decomposition methods. These mathematical methods are well adapted to parallel computing, however their effective exploitation on parallel machines becomes difficult when the applications have an irregular behavior. This is the case for example when the mathematical problems are solved over complex geometries or when one uses mesh refinement techniques. A programming technique that is useful to cope with irregular parallel applications is multithreading. In this thesis we perform a thorough study on the use of this programming paradigm for solving PDE problems through domain decomposition methods, and we show that a generic algorithmic writing of this methods is possible. One of our main contributions resides in the design and implementation of a programming harness called Ahpik, allowing for easy development of applications relying on domain decomposition methods. This programming environment provides a generic support that is adaptable to many mathematical methods, which can be synchronous or asynchronous, overlapping or non-overlapping. Its object-oriented design allows to encapsulate implementation details concerning the management of threads and communications, which eases the task of developing new methods. We validate the Ahpik environment in the context of the resolution of some classical PDE problems and in particular for one large problem in computational fluid dynamics.

Keywords: parallel multithreaded programming, domain decomposition methods, parallel solution of Partial Differential Equations, generic programming, clusters of multiprocessors

Laboratoire Informatique et Distribution – ENSIMAG, antenne de Montbonnot. ZIRST – 51, av. Jean Kuntzmann, 38330 Montbonnot Saint Martin.