Résolution de grands systèmes linéaires issus de la méthode des éléments finis sur des calculateurs massivement parallèles

Ibrahima Gueye

▶ **To cite this version:**

Ibrahima Gueye. Résolution de grands systèmes linéaires issus de la méthode des éléments finis sur des calculateurs massivement parallèles. Modélisation et simulation. École Nationale Supérieure des Mines de Paris, 2009. Français. NNT : 2009ENMP1658 . tel-00477653

HAL Id: tel-00477653

https://pastel.hal.science/tel-00477653v1

Submitted on 29 Apr 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

MINES ParisTech

ED n°432: Sciences des Métiers de l'Ingénieur

THÈSE

pour obtenir le grade de

DOCTEUR DE L'ÉCOLE NATIONALE SUPÉRIEURE DES MINES DE PARIS

Spécialité « **Mécanique** »

Présentée et soutenue publiquement par

Ibrahima GUEYE

le 15 décembre 2009

RÉSOLUTION DE GRANDS SYSTÈMES LINÉAIRES ISSUS DE LA MÉTHODE DES ÉLÉMENTS FINIS SUR DES CALCULATEURS MASSIVEMENT PARALLÈLES

Directeurs de thèse: Georges CAILLETAUD et François-Xavier ROUX

Jury

M.	Christian REY	Président	ENS Cachan
M.	David DUREISSEIX	Rapporteur	Université Montpellier II
M.	Frédéric HECHT	Rapporteur	Université Paris VI
M.	Pascal LANDEREAU	Examinateur	MICHELIN
M^{me}	Masha SOSONKINA	Examinatrice	SAMTECH
M.	Frédéric FEYEL	Examinateur	ONERA
M.	François-Xavier ROUX	Examinateur	ONERA
M.	Georges CAILLETAUD	Examinateur	MINES ParisTech

Je dédie cette thèse de doctorat
à tous mes proches et particulièrement, à mes parents pour leur patience et mes oncles (Paris 12^{e}) pour l'accueil chaleureux qu'ils m'ont réservé à mon arrivée en France.

Remerciements

A l'issue de ces travaux, je suis convaincu que la thèse est loin d'être un travail solitaire. En effet, je n'aurais jamais pu réaliser ces travaux sans le soutien d'un grand nombre de personnes dont la générosité, la bonne humeur et l'intérêt manifestés à l'égard de mes recherches m'ont permis de progresser dans cette phase délicate de l'***apprenti-chercheur***.

Je voudrais tout d'abord exprimer mes plus profonds remerciements à Georges CAILLETAUD et François-Xavier ROUX qui ont dirigé cette thèse dans la continuité de mon stage de Master II recherche. Tout au long de ces trois années, ils ont su orienter mes recherches aux bons moments, en me donnant des conseils judicieux et des idées innovantes. Les mots ne seront jamais assez explicites pour décrire leurs qualités humaines, leur sens de l'écoute et de compréhension.

Je tiens à remercier les rapporteurs de cette thèse, David DUREISSEIX et Frédéric HECHT, pour la rapidité avec laquelle ils ont lu mon manuscrit et l'intérêt qu'ils ont porté à mes travaux. Merci également aux autres membres du jury qui m'ont fait l'honneur d'accepter de juger ces travaux : Christian REY qui s'est sacrifié pour présider le jury de thèse et les examinateurs Masha SOSONKINA venue de Toulouse et Pascal LANDEREAU qui s'est déplacé depuis Clermont-Ferrand.

Pour reprendre un ordre plus chronologique, je voudrais remercier deux personnes qui ont joué un rôle fondamental dans cette thèse. Tout d'abord Xavier JUVIGNY, dont les travaux ont constitué le point de départ de ma thèse, a su éveiller en moi, dès mon stage de Master II recherche, une véritable passion pour l'algébre linéaire et les techniques de renumérotation, et un goût pour la programmation en C++. Grâce à lui, je garderai de mon passage à l'ONERA, de solides connaissances sur les méthodes de résolution directe de systèmes linéaires. Puis Frédéric FEYEL, l'expert ou l'un des experts pour développer dans le code de calcul ZéBuLon, est le guide de ma découverte de ce code. Il m'a sans aucun doute apporté, au travers des centaines d'heures passées ensemble, une aide pour développer dans ZéBuLon et y intégrer le solveur direct.

Je remercie particulièrement tous les autres membres de l'unité ***Calcul Hautes Performances*** du DTIM de l'ONERA pour leur bonne humeur et leur attitude à mon égard, Juliette, Alain, Riadh, Sylvain et sans oublier Nadia, ma petite sœur et compagne de bataille depuis le Master II et avec qui j'ai souvent pris des pauses thé les après-midis. Je n'oublierai pas aussi les doctorants des autres équipes du département, entre autre, Adrien, Anne-Marie, Antoine et Évangeline pour les bons moments passés ensemble. Je ne peux ne pas remercier Françoise notre secrétaire pour sa sympathie et sa joie de vivre, c'est une femme d'un bon cœur que je remercie très chaleureusement.

Mes remerciements vont aussi à tout le personnel du Centre des Matériaux (Evry) que j'ai côtoyé, plus particulièrement, les membres de l'équipe CoCas et ceux de l'équipe VAL (Stéphane, Farida, Laurent, Nikolay et Djamel) pour leur assistance et leur disponibilité, sans oublier notre équipe de football avec qui j'ai passé souvent des moments de plaisir et d'intensité les mercredis où je suis passé au centre.

Je voudrais également remercier mes amis et frères : Masseye et sa femme Maguette, Abdourahmane, Mouhamad, Ousseynou, Assane, Ange, Benjamin, Noureddine, Mountaga, Sabri, Amar, Bouba, Momar, El Seck (pardonnez-moi si j'ai oublié quelqu'un) pour leur support et leur soutien moral qu'ils m'auront apportés tout au long de la réalisation de ces travaux. Je n'oublierai pas non plus l'aide précieuse de Céline GERARD que je remercie tout particulièrement. Une pensée spéciale est adressée à une personne très chère, Mame Sophie, pour ses encouragements et pour sa confiance. Je la remercie de m'avoir tenu la main jusqu'aux dernières lignes de ce mémoire. En plus de me supporter dans des conditions "normales", ce qui n'est déjà pas une mince affaire je le conçois, elle a dû subir mon humeur en fin de thèse et se mettre en retrait. Et pourtant, quelle présence indispensable. Merci d'être là tous les jours.

Ces travaux ont bénéficié d'un accès aux moyens de calcul du CINES (Centre Informatique National de l'Enseignement Supérieur) au travers de l'allocation de ressources 2009-cmp6116 attribuée par la société civile GENCI (Grand Equipement National de Calcul Intensif). C'est pourquoi je témoigne ici toute ma reconnaissance au CINES et je tiens à remercier Saber El-AREM pour sa gentillesse et pour sa disponibilité pour effectuer les calculs.

YEEN GNËP, DIEUREU NGUEN DIEUFETI !

Ibrahima GUEYE,
Paris, le 15 décembre 2009.

Table des matières

Liste des figures **xi**

Liste des tableaux **xv**

Liste des algorithmes **xvii**

Introduction générale **1**

1 Mise en œuvre d'un solveur direct **5**
- 1.1 Le cadre général . . . 7
  - 1.1.1 Matrices creuses et remplissage . . . 7
  - 1.1.2 Techniques de renumérotation . . . 8
  - 1.1.3 Arbre d'élimination . . . 8
- 1.2 La phase d'analyse . . . 9
  - 1.2.1 Dissection emboîtée . . . 10
  - 1.2.2 Factorisation symbolique . . . 12
- 1.3 La phase numérique . . . 19
  - 1.3.1 Mise à l'échelle ou "scaling" de la matrice originale . . . 19
  - 1.3.2 Factorisation numérique . . . 19
  - 1.3.3 Condensation statique . . . 20
- 1.4 La résolution des systèmes triangulaires . . . 21
  - 1.4.1 Phase de descente . . . 21
  - 1.4.2 Résolution du problème condensé . . . 22
  - 1.4.3 Phase de remontée . . . 22
- 1.5 La prise en compte des systèmes non inversibles . . . 23
  - 1.5.1 Détection locale des singularités . . . 23
  - 1.5.2 Traitement des modes à énergie nulle . . . 23
  - 1.5.3 Illustration du déroulement de l'algorithme . . . 24
- 1.6 L'évaluation des performances . . . 26
  - 1.6.1 Comparaison des approches proposées dans la phase d'analyse . . . 26
  - 1.6.2 Évaluation du nombre optimal de sous-structures . . . 28
  - 1.6.3 Comparaison des temps d'exécution . . . 30
  - 1.6.4 Comparaison des temps de descente-remontée . . . 31
  - 1.6.5 Analyse de la complexité du solveur . . . 32
- Conclusion . . . 33

2 Parallélisation du solveur **35**
- 2.1 L'état de l'art . . . 37
  - 2.1.1 Solveurs destinés aux architectures à mémoire distribuée . . . 37
  - 2.1.2 Solveurs adaptés aux architectures à mémoire partagée . . . 38

2.1.3 Solveurs directs basés sur une programmation hybride . 38
2.1.4 Discussions 39
2.2 La programmation multi-cœurs 39
2.2.1 Motivation 40
2.2.2 Architectures multi-cœurs 40
2.2.3 Parallélisme à mémoire partagée : le multi-threading . . . 41
2.3 Les démarches pour la parallélisation du solveur . . . 45
2.3.1 Version multi-threads avec Pthreads 47
2.3.2 Version multi-threads avec OpenMP 54
2.3.3 Évaluation des deux versions multi-threads 54
2.3.4 Mise en place d'une version multi-threads hybride 61
2.4 Les performances en multi-threading 61
2.4.1 Analyse des performances 62
2.4.2 Comparaison avec DSCPack et MUMPS 63
Conclusion 64

3 Amélioration et évaluation des performances de la méthode FETI 65
Introduction 67
3.1 La présentation de la méthode FETI 68
3.1.1 Description de la version initiale de FETI 68
3.1.2 Résolution itérative du problème d'interface 70
3.1.3 Algorithme détaillé de FETI-1 72
3.2 Les facteurs critiques d'une méthode FETI 75
3.2.1 Choix d'un solveur local 75
3.2.2 Traitement des singularités et calcul des pseudo-inverses 76
3.2.3 Gestion du problème grossier 77
3.3 L'intégration d'un parallélisme à deux niveaux dans FETI 78
3.3.1 Extraction en parallèle des colonnes de $B^{(s)}$ 78
3.3.2 Calcul en parallèle des pseudo-inverses et pseudo-solutions 80
3.4 L'évaluation des performances de la méthode FETI . . . 82
3.4.1 Environnement d'évaluation 82
3.4.2 Critères de mesure de performances 84
3.4.3 Description des cas tests parallèles 85
3.4.4 Analyse de l'efficacité de FETI 85
3.4.5 Analyse de l'extensibilité de FETI 92
Conclusion 98

Conclusion générale 99

A Annexes 103
A.1 Factorisation par élimination de Gauss 105
A.1.1 Les généralités de la méthode 105
A.1.2 Les inconvénients de la méthode 107
A.2 Algorithmes de renumérotation 109
A.2.1 L'algorithme de dissection emboîtée 110
A.2.2 L'algorithme de degré minimum et ses variantes 110
A.2.3 L'algorithme de Cuthill - McKee 111
A.3 Présentation de la méthode des éléments finis 112
A.3.1 Le problème modèle 113

A.3.2 La mise sous forme variationnelle d'un problème d'EDP . 113
A.3.3 Le choix d'un maillage et discrétisation 114
A.3.4 Le problème sous forme matricielle 116
A.4 Analyse des performances des versions multi-threads du solveur Dissection 117
A.4.1 Les temps de calcul pour la phase d'analyse 117
A.4.2 Les temps de calcul pour la phase numérique 119
A.4.3 Les temps de calcul pour la phase de descente-remontée . 120
A.4.4 Les temps d'exécution en fonction du nombre de threads 121
A.4.5 Les conclusions 121

Bibliographie **123**

Notations **129**

Liste des figures

1.1 L'élimination du sommet j relie tous ses voisins entre eux. . 8
1.2 Effet de la renumérotation sur le phénomène de remplissage. 8
1.3 Exemple de système linéaire creux. 10
1.4 Découpage par bissection récursive. 11
1.5 Super-arbre d'élimination de hauteur 3 11
1.6 Partition d'un maillage éléments finis 14
1.7 Nœuds internes et interfaces des sous-domaines. 15
1.8 Super-arbre d'élimination et dépendances : 1^{re} approche. . . 15
1.9 Super-arbre d'élimination et dépendances : 2^{e} approche. . . 17
1.10 Structure de la matrice M factorisée. 18
1.11 Exemple de système linéaire singulier. 24
1.12 Temps CPU d'analyse : problème de 41472 ddl. 26
1.13 Temps CPU d'analyse : problème de 107811 ddl. 27
1.14 Temps CPU d'exécution : problème de 41472 ddl. 27
1.15 Temps CPU d'exécution : problème de 107811 ddl. 27
1.16 Première phase de comparaison des temps CPU d'exécution. 30
1.17 Deuxième phase comparaison des temps CPU d'exécution. 31
1.18 Comparaison des temps CPU de descente-remontée. 32

2.1 Architecture à mémoire distribuée. 37
2.2 Architecture à mémoire partagée. 38
2.3 Processeur d'Intel à 80 cœurs. 40
2.4 Exemple de fonctionnement multi-tâches. 40
2.5 Exemples d'architectures multi-cœurs et multi-processeurs. 41
2.6 Exemple de région parallèle : cinq threads sur quatre cœurs. 45
2.7 Indépendance du calcul des contributions apportées à Γ_1^1. . 46
2.8 Approche générale : répartition des tâches sur quatre cœurs. 46
2.9 Exemple de répartition de 8 sous-structures sur quatre cœurs. 49
2.10 Exemple de répartition de 9 sous-structures sur quatre cœurs. 49
2.11 Calcul par le thread 0 des contributions apportées par I_1 et I_2. 50
2.12 Exemple de répartition de charge des threads sur quatre cœurs. 52
2.13 Mise à jour du terme $B_{\Gamma_1^2}$ partagé par les threads 0 et 1. . . . 53
2.14 Différents types de temps dans un programme multi-threads. 55
2.15 Accélération dans la phase d'analyse du système. 55
2.16 Accélération dans l'étape de factorisation symbolique. . . . 56
2.17 Accélération dans l'étape de factorisation locale. 57
2.18 Temps de factorisation de Schur sur deux cœurs. 57
2.19 Temps de factorisation de Schur sur quatre cœurs. 58
2.20 Temps de factorisation de Schur sur huit cœurs. 58

2.21 Accélération sur l'exécution du complément de Schur. . . . 59
2.22 Accélération de la phase de descente-remontée. 60
2.23 Accélération des versions multi-threads. 60
2.24 Accélération du solveur multi-threads : problème à 206763 ddl. 62
2.25 Accélération du solveur multi-threads : problème à 397953 ddl. 62
2.26 Comparaison avec DSCPack et MUMPS : problème à 206763 ddl. 63
2.27 Comparaison avec DSCPack et MUMPS : problème à 397953 ddl. 63

3.1 Principe de la méthode FETI. 68
3.2 FETI-1 : solveur d'interface à deux niveaux sur un GCPC. . 74
3.3 Cluster du CdM (source : http ://www.mat.ensmp.fr). . . . 82
3.4 Cluster JADE du CINES (source : http ://www.cines.fr). . . 83
3.5 Architecture de JADE (source : http ://www.cines.fr). . . . 83
3.6 Découpage du problème de taille 285099 en 50 sous-domaines 86
3.7 Découpage du problème de taille 3371544 en 8 sous-domaines 86
3.8 Découpage du problème de taille 3371544 en 32 sous-domaines 86
3.9 Version exitante de FETI : efficacité dans ZéBuLon. 88
3.10 Accélération de la version existante de FETI. 89
3.11 Version améliorée de FETI : efficacité dans ZéBuLon. 91
3.12 Accélération de la version améliorée de FETI. 92
3.13 Découpage du cube de dimension $(1, 1, 1)$ en 2 sous-domaines. 92
3.14 Découpage de la poutre de dimension $(10, 1, 1)$ en 10 sous-domaines. 93
3.15 Version existante de FETI : extensibilité dans ZéBuLon. . . . 94
3.16 Scale-up de la version existante de FETI. 95
3.17 Version améliorée de FETI : extensibilité dans ZéBuLon. . . 96
3.18 Scale-up de la version améliorée de FETI. 97

A.1 Exemple classique de matrice dite "flèche". 108
A.2 Matrice "flèche" avec permutation inverse des inconnues. . . 108
A.3 Apparition d'une clique par élimination d'un sommet. . . . 110
A.4 Effet de la numérotation d'un sommet sur la largeur de bande. 112
A.5 Temps d'analyse du système en fonction du nombre de threads. 117
A.6 Comparaison des étapes d'analyse du système. 118
A.7 Temps de factorisation symbolique en fonction du nombre de threads. 118
A.8 Temps de factorisation locale en fonction du nombre de threads. 119
A.9 Temps de factorisation de Schur sur deux cœurs. 119
A.10 Temps de factorisation de Schur sur quatre cœurs. 119
A.11 Temps de factorisation de Schur sur huit cœurs. 120

A.12 Temps d'exécution de Schur en fonction du nombre de threads. 120
A.13 Temps de descente-remontée en fonction du nombre de threads. 120
A.14 Temps d'exécution en fonction du nombre de threads. . . . 121

Liste des tableaux

1.1 Nombre de super-nœuds par niveau. 15
1.2 Découpage du problème de taille 41472. 28
1.3 Découpage du problème de taille 107811. 28
1.4 Découpage du problème de taille 397953. 29
1.5 Temps d'exécution (s) en fonction du nombre de sous-structures. 29
1.6 Complexité du solveur dans la phase numérique. 32

2.1 Temps elapsed (s) des étapes d'analyse du système. 56

3.1 Efficacité de la version existante de FETI. 87
3.2 Accélération de la version existante de FETI. 88
3.3 Optimisation de la distribution des tâches. 89
3.4 Temps elapsed (s) dans ZéBuLon pour chaque type de distribution. 90
3.5 Efficacité de la version améliorée. 91
3.6 Extensibilité de la version existante de FETI. 94
3.7 Scale-up de la version existante de FETI. 95
3.8 Extensibilité de la version améliorée de FETI. 96
3.9 Scale-up de la version améliorée de FETI. 96
3.10 Temps (s) de traitement des mouvements de corps rigide. . 97

Liste des algorithmes

1.1 Mise en place de la structure d'un super-arbre 13
1.2 Construction d'un arbre avec l'algorithme génétique 14
1.3 Recherche des inconnues interfaces des séparateurs 16
1.4 Recherche des interfaces des sous-structures 17
2.1 Répartition des tâches dans la phase d'analyse 47
2.2 Répartition des tâches dans la phase numérique 51
2.3 Répartition des tâches dans l'étape de substitution 53
3.1 Vérification du noyau de projection $\widetilde{P} = \widetilde{N}^T \widetilde{N}$ 79
A.1 Étape $(k+1)$ de l'élimination de Gauss 105
A.2 Élimination de Gauss : phénomène de remplissage 108
A.3 Description algorithmique du degré minimum 111
A.4 Algorithme de Cuthill - McKee inverse 112

Introduction

La résolution de systèmes linéaires a toujours joué un rôle important dans la simulation numérique de nombreux problèmes scientifiques et industriels, en particulier ceux qui sont étudiés en mécanique des structures. Dans ce domaine, l'utilisation des codes de calcul par éléments finis conduit souvent à des systèmes linéaires creux qui peuvent atteindre des tailles de plusieurs dizaines de millions d'inconnues. La résolution de ces systèmes présente l'inconvénient de mener à des coûts prohibitifs en temps CPU et en espace mémoire.

Les méthodes de décomposition de domaine sont couramment utilisées pour résoudre ces systèmes linéaires en parallèle. Elles sont capables de réduire les coûts de calcul, et elles permettent ainsi d'envisager en mécanique des structures la simulation de problèmes de grande taille avec des lois de comportement de plus en plus complexes (prise en compte de nombreuses variables internes) et des géométries affinées (géométrie des composants, forme des microstructures). Ces méthodes se présentent comme le mélange de deux approches, l'une directe et l'autre itérative. D'une part, un algorithme itératif est utilisé pour résoudre le problème condensé aux interfaces entre sous-domaines. D'autre part, à chaque itération, un système linéaire est résolu avec un solveur direct au niveau de chacun des sous-domaines indépendamment des autres. La première méthode de cette sorte est attribuée à H. A. Schwarz en 1869 et sa forme la plus connue est analysée et étudiée par Lions (1988). Dans cette méthode, il y a un recouvrement partiel des sous-domaines. Sa convergence est très sensible aux hétérogénéités et elle duplique une partie non négligeable des calculs dans les zones de recouvrement. Cette technique reste donc marginale et beaucoup moins utilisée en mécanique des structures que les méthodes de décomposition de domaine sans recouvrement, où la taille du problème d'interface est de dimension deux pour les problèmes en $3D$. Le principe de ces dernières consiste à imposer aux interfaces entre sous-domaines des conditions de raccord, soit en déplacements (approche primale) c'est le cas notamment de Mandel (1993), Le Tallec (1994), soit en contraintes (approche duale) par exemple dans (Farhat et Roux 1994; Farhat et al. 2000b). D'autres méthodes de décomposition de domaine sont nées de la combinaison des deux approches précédentes : l'approche hybride (Farhat et al. 2001; Gosselet et Rey 2006). Parmi toutes ces méthodes, FETI est la plus utilisée. Elle est basée sur l'approche duale et présente l'avantage d'être robuste et bien adaptée aux problèmes étudiés en mécanique des structures. Cependant, des études faites sur de gros cas tests (plusieurs dizaines de millions d'inconnues) ont montré que l'efficacité de cette méthode se détériore au-delà d'un certain nombre de sous-structures (quelques centaines). Si ces gros modèles sont découpés en un nombre raisonnable de sous-structures, un autre problème survient. La taille des systèmes locaux émanant de ce découpage est très importante. En outre,

certains sont souvent associés à des sous-domaines flottants et sont donc singuliers. Les sous-domaines flottants sont des structures qui n'ont pas assez de conditions aux limites essentielles pour prévenir les mouvements de corps rigide. Équiper la méthode FETI d'un solveur direct capable de résoudre efficacement ces grands systèmes locaux peut apporter une solution à ces problèmes et par conséquent lui assurer une meilleure extensibilité parallèle.

D'autre part, avec l'évolution technologique des microprocesseurs, on voit naître de nouvelles architectures massivement multi-cœurs. L'intérêt essentiel ou la force des solutions multi-cœurs est de permettre l'exécution simultanée d'une tâche par cœur (voir Akhter et Roberts 2006). Certains algorithmes massivement parallèles peuvent tirer pleinement profit de ces nouvelles architectures.

C'est dans ce contexte que ces travaux ont été effectués. Ils consistent à mettre au point un parallélisme à deux niveaux pour résoudre de grands systèmes linéaires issus de la méthode des éléments finis sur des calculateurs massivement parallèles.

L'objectif de cette thèse a été de réaliser dans un premier temps, un solveur direct creux. Pour cela, un algorithme de dissection emboîtée est proposé pour analyser les systèmes linéaires et trouver un ordre d'élimination des inconnues durant la factorisation (voir les travaux de George 1973; George et Liu 1981; Charrier et Roman 1989; Juvigny 1997). Cette méthode de renumérotation est mieux adaptée au parallélisme que les algorithmes de degré minimum (Tinney et Walker 1967; Liu 1985; Amestoy et al. 1996). Elle permet de diviser la factorisation en autant d'étapes que de niveaux dans l'algorithme de dissection. Un dernier point pour atteindre le but de cette première partie a été la prise en compte des systèmes non inversibles en cas de présence de sous-structures flottantes dans la méthode FETI. Ce point est utile dans beaucoup de méthodes de décomposition de domaine, soit dans la construction d'un préconditionneur, soit pour la formulation de l'opérateur lui-même. La prise en compte a consisté à traiter automatiquement et proprement les singularités de ces systèmes en calculant leurs modes rigides ou plus généralement leurs modes à énergie nulle. Dans un deuxième temps, l'objectif a été de paralléliser le nouveau solveur direct mis en œuvre. Pour cela, une technique de parallélisme reposant sur la méthodologie du multi-threading est utilisée (Lewis et Berg 1998). Celle-ci est basée sur du multi-tâches et fonctionne sur des machines à mémoire partagée. De plus, elle permet au solveur parallélisé de tirer profit des nouveaux processeurs multi-cœurs. A notre connaissance, on ne trouve pas gratuitement ou sur le marché une bibliothèque qui satisfasse tous ces besoins. Dans un troisième temps, l'objectif a été de faire de la résolution locale parallèle dans la méthode FETI. Pour cela, le solveur direct parallèle est introduit au niveau des sous-domaines de FETI. Dans un quatrième temps, le but a été de revoir le plus haut niveau de parallélisme. C'est un parallélisme à gros grains entre les sous-domaines de la méthode FETI. Pour cela, la méthode itérative utilisée pour résoudre le problème d'interface de FETI est d'abord améliorée avec l'aide du solveur mis en place. Ensuite, les performances du solveur sont évaluées dans cette méthode de décomposition de domaine. De gros calculs sont ainsi simulés avec le code ZéBuLon. Ce dernier est un code de calcul par éléments finis

pour des problèmes de mécanique non linéaires. Il a été développé en collaboration avec l'ONERA, MINES ParisTech et la société NW Numerics, et a été parallélisé avec la méthode FETI (Feyel 1998; Gosselet et Rey 2006).

Dans cette thèse, nous exploitons un parallélisme à deux niveaux et l'intégrons dans une méthode de décomposition de domaine pour résoudre de très grands systèmes linéaires creux.

Le *premier chapitre* aborde la **mise en œuvre d'un solveur direct** pour inverser des systèmes linéaires creux (Guèye et al. 2007). Ces systèmes peuvent être symétriques ou non symétriques, réels ou complexes, à second membre simple ou multiple. Nous situons d'abord ce chapitre dans son cadre général en faisant une brève présentation des idées principales qui mènent au développement de solveurs directs creux robustes pouvant résoudre de tels systèmes. Nous nous appuyons ensuite sur ces idées pour réaliser notre solveur. Nous complétons cette réalisation par une étape indispensable dans son utilisation comme solveur local dans certaines méthodes de décomposition de domaine : la prise en compte des systèmes non inversibles. Nous traitons automatiquement et proprement les singularités de ces systèmes par un calcul de leurs modes à énergie nulle. Nous clôturons ce chapitre en évaluant les performances séquentielles du solveur et en le comparant avec d'autres solveurs disponibles dans le code de calcul ZéBuLon notamment les solveurs directs creux de référence : DSCPack (Raghavan 2001; 2002) et MUMPS (Amestoy et al. 2000).

Le *second chapitre* est consacré à la **parallélisation du solveur** mis en place (Guèye et al. 2009). Nous dressons d'abord l'état de l'art sur les bibliothèques permettant actuellement de résoudre par méthode directe de grands systèmes linéaires creux sur des calculateurs massivement parallèles. Nous exposons ensuite la motivation et l'intérêt de la programmation multi-cœurs que nous voulons choisir pour accroître les performances du solveur à travers un modèle de parallélisme à mémoire partagée. Nous discutons de l'avènement des architectures multi-cœurs et de la méthodologie du multi-threading (multi-programmation légère). Enfin, nous détaillons les différentes démarches que nous avons mises au point pour créer des threads dans le solveur direct. Nous terminons ce chapitre par la mesure des performances de la version multi-threads du solveur sur une machine multi-cœurs.

Le *troisième chapitre* porte sur l'**amélioration et l'évaluation des performances des méthodes** de décomposition de domaine de type **FETI**. Nous introduisons d'abord ces méthodes en essayant d'aborder leur problématique. Nous présentons ensuite la méthode FETI classique appelée aussi FETI-1 (*one-level* FETI en anglais) et nous soulevons les questions qui déterminent les points critiques de celle-ci. Puis, nous montrons comment un parallélisme à deux niveaux est intégré dans cette méthode. Nous inversons les problèmes locaux en parallèle et nous discutons de l'approche utilisée pour améliorer la phase itérative de FETI. Enfin, nous évaluons et analysons les performances de cette version améliorée des méthodes FETI en réalisant des tests parallèles, avec le code de calcul ZéBuLon, sur le cluster du Centre des Matériaux (MINES ParisTech) et le supercalculateur JADE du Centre Informatique National de l'Enseignement Supérieur (CINES).

Mise en œuvre d'un solveur direct

1

Sommaire

1.1 Le cadre général . . . 7
1.1.1 Matrices creuses et remplissage . . . 7
1.1.2 Techniques de renumérotation . . . 8
1.1.3 Arbre d'élimination . . . 8
1.2 La phase d'analyse . . . 9
1.2.1 Dissection emboîtée . . . 10
1.2.2 Factorisation symbolique . . . 12
1.3 La phase numérique . . . 19
1.3.1 Mise à l'échelle ou "scaling" de la matrice originale . . . 19
1.3.2 Factorisation numérique . . . 19
1.3.3 Condensation statique . . . 20
1.4 La résolution des systèmes triangulaires . . . 21
1.4.1 Phase de descente . . . 21
1.4.2 Résolution du problème condensé . . . 22
1.4.3 Phase de remontée . . . 22
1.5 La prise en compte des systèmes non inversibles . . . 23
1.5.1 Détection locale des singularités . . . 23
1.5.2 Traitement des modes à énergie nulle . . . 23
1.5.3 Illustration du déroulement de l'algorithme . . . 24
1.6 L'évaluation des performances . . . 26
1.6.1 Comparaison des approches proposées dans la phase d'analyse . . . 26
1.6.2 Évaluation du nombre optimal de sous-structures . . . 28
1.6.3 Comparaison des temps d'exécution . . . 30
1.6.4 Comparaison des temps de descente-remontée . . . 31
1.6.5 Analyse de la complexité du solveur . . . 32
Conclusion . . . 33

Dans ce chapitre nous mettons en œuvre un solveur direct pour inverser de grands systèmes linéaires creux $MX = B$. Ces systèmes peuvent être symétriques ou non symétriques, réels ou complexes, à second membre simple ou multiple. Nous allons d'abord situer ce chapitre dans son cadre général en faisant une brève présentation des idées principales qui mènent au développement de solveurs directs creux robustes

pouvant résoudre de tels systèmes. Nous nous appuyons ensuite sur ces idées pour réaliser notre solveur. Nous complétons cette réalisation par une étape indispensable dans son utilisation comme solveur local dans certaines méthodes de décomposition de domaine : la prise en compte des systèmes non inversibles par un calcul automatique et propre de leurs modes à énergie nulle. Nous clôturons ce chapitre par une évaluation des performances séquentielles du solveur mis en place et une comparaison avec d'autres solveurs existants dans le code de calcul ZéBuLon notamment les solveurs linéaires creux de référence : DSCPack (Raghavan 2001; 2002) et MUMPS (Amestoy et al. 2000).

1.1 Le cadre général

La résolution de grands systèmes linéaires ***creux*** $MX = B$ par méthodes directes est souvent basée sur l'élimination de Gauss (voir l'annexe A.1 ou bien Duff et Reid (1978); Lascaux et Théodor (2004a)). Plutôt que de traiter directement le système, la matrice $M = (m_{ij})$ est factorisée sous la forme LU, où $L = (l_{ij})$ est une matrice triangulaire inférieure et $U = (u_{ij})$ est une matrice triangulaire supérieure. Cette résolution est constituée en général de trois grandes phases :

1. Une **phase d'analyse** qui est effectuée en deux étapes.
   (a) La première étape consiste à trouver un ***ordre*** d'élimination des inconnues du système à traiter pour minimiser le ***remplissage*** lors de la factorisation numérique de M.
   (b) La deuxième étape consiste à effectuer une factorisation symbolique de M pour déterminer le nombre et la position des nouveaux termes qui seront créés au cours de la factorisation. Ce processus permet alors de connaître à l'avance la structure creuse des matrices triangulaires L et U. Cela réduit par conséquent les coûts du calcul effectif des termes de ces matrices.
2. Une **phase numérique** qui succéde à celle d'analyse. Elle permet de décomposer implicitement les coefficients de la matrice M en un produit LU. Cette étape s'appuie sur les informations qui ont été obtenues en analysant le système.
3. Une **phase de descente-remontée** qui permet de calculer la solution numérique X. Les systèmes triangulaires $LY = B$ et $UX = Y$ résultants de la phase numérique sont résolus. Dans le cas où les systèmes linéaires ont plusieurs seconds membres, il suffit d'effectuer successivement ou simultanément cette phase de descente-remontée qui est peu coûteuse par rapport à celle numérique.

1.1.1 Matrices creuses et remplissage

Il n'y a pas de définition précise d'une matrice creuse, mais on peut se contenter de la définition qui suit.

Définition 1.1 *Une matrice creuse M de dimension n est une matrice dont le nombre* **nonz** *de coefficients non nuls vérifie :* $\mathbf{nonz} \ll \mathbf{n^2}$

Généralement, les facteurs L et U sont plus denses que la matrice initiale M, dans le sens où ils possèdent plus de coefficients non nuls. C'est le phénomène de remplissage. Il est dû au fait qu'un terme nul dans M peut devenir non nul dans L ou U durant le processus de factorisation, c'est-à-dire que certains termes l_{ik} et u_{ki} sont différents de zéro alors que les termes initiaux m_{ik} et m_{ki} sont nuls. Ce remplissage a une interprétation en terme de graphe. Supposons dans ce cas que toutes les inconnues du système sont représentées par les sommets d'un graphe : on reliera les sommets i et k si et seulement si le terme m_{ik} est non nul. Si m_{ik} est nul, mais que i et k sont reliés à j alors l'élimination de j consistera à relier graphiquement i et k. La figure 1.1 illustre cette interprétation. Les arêtes en pointillé sont celles qui sont créées par l'élimination de l'inconnue j.

Elles correspondent à de nouveaux coefficients non nuls dans les matrices triangulaires L et U.

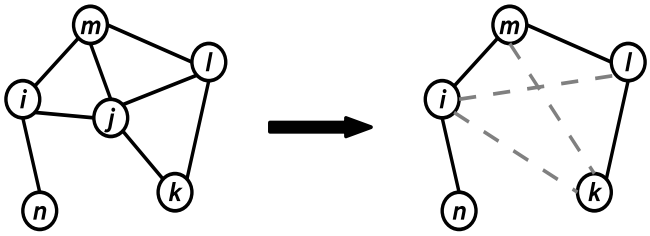


FIG. 1.1 – *L'élimination du sommet j relie tous ses voisins entre eux.*

La gestion du phénomène de remplissage de la matrice factorisée est donc essentielle pour les méthodes directes sur les systèmes creux. Il est directement lié à l'ordre d'élimination des inconnues. Il est donc nécessaire de trouver des ***techniques de renumérotation*** pour limiter celui-ci et réduire le nombre d'opérations à effectuer lors de la factorisation.

1.1.2 Techniques de renumérotation

Une renumérotation des inconnues d'un système linéaire creux, c'est-à-dire une permutation de ses lignes et de ses colonnes, permet de limiter le phénomène de remplissage comme le montre l'exemple de la figure 1.2. Lors de la factorisation de la matrice de gauche, une structure pleine est créée. En renumérotant les équations en ordre inverse du plus grand au plus petit, on obtient la structure de droite pour laquelle il n'y a aucun remplissage au cours de la factorisation.

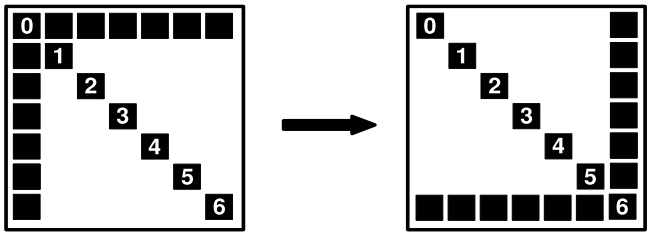


FIG. 1.2 – *Effet de la renumérotation sur le phénomène de remplissage.*

Plusieurs techniques de renumérotation heuristiques peuvent être utilisées. C'est notamment le cas des variantes de l'algorithme de degré minimum (Tinney et Walker 1967; Liu 1985; Amestoy et al. 1996) ou celles de la dissection emboîtée (George 1973; George et Liu 1981). Le résultat obtenu est un **arbre d'élimination** représentant les dépendances dans les calculs et permettant de mener à bien les phases suivantes. Dans la pratique, cet arbre peut être construit directement à partir du graphe représentant la structure creuse du système.

1.1.3 Arbre d'élimination

L'arbre d'élimination, introduit par Liu (1990), joue un rôle important dans plusieurs aspects de la factorisation des matrices creuses. Il exprime les dépendances relatives au processus de factorisation. L'arbre d'élimination concernait au début les matrices creuses symétriques. Puis, leur étude a

été étendue aux matrices non symétriques. Celle-ci a été faite en considérant pour toute la phase de construction de l'arbre, la matrice $M + M^T$ (symétrique) au lieu de la matrice M. Ainsi, on se ramène au cas symétrique pour générer l'arbre d'élimination.

Pour tout indice j, on note par C_j la j^e colonne de M. La définition 1.2 établit une relation de dépendance, notée $\rightarrow$, sur l'ensemble colonnes $\{C_j$ telles que $j \in \{1, 2, ..., n\}\}$ de M.

Définition 1.2 *Soit (i, j) dans $\{1, 2, ..., n\}^2$ tel que $i > j$. $C_j \rightarrow C_i$ (C_i **dépend** de C_j) si et seulement si le calcul de l'élément diagonal ou de l'un des éléments sous-diagonaux de C_i nécessite le calcul préalable de l'un des facteurs de C_j.*

Lemme 1.1 *$C_j \rightarrow C_i$ si et seulement si $l_{ij} \neq 0$. De même, par symétrie de la structure, $C_j \rightarrow C_i$ si et seulement si $u_{ji} \neq 0$.*

Définition 1.3 *Un **graphe orienté** est un graphe dont les arêtes sont définies par leur origine et leur extrémité, c'est-à-dire les arêtes sont orientées et munies d'un sens. Une arête d'un graphe orienté est définie par la donnée d'un couple de sommets.*

Définition 1.4 *Un **graphe de dépendance** est un graphe orienté dont les sommets sont les colonnes de M et dont les arêtes sont l'ensemble des couples (C_j, C_i) tels que $C_j \rightarrow C_i$.*

Lemme 1.2 *Soient i, j et k ($i < j < k$) vérifiant $C_j \rightarrow C_i$ et $C_i \rightarrow C_k$. On a alors $C_j \rightarrow C_k$.*

Définition 1.5 *Un graphe G^0 est une **réduction transitive** d'un graphe orienté G, si :*

1. *G^0 contient un chemin direct de i à j si et seulement si G contient un chemin direct de i à j ;*
2. *aucun graphe ayant moins d'arêtes que G^0 ne satisfait la condition* 1.

Théorème 1.1 *Le sous-graphe G^0, appélé **arbre d'élimination**, est la réduction transitive du graphe de dépendance.*

Etant donné une matrice creuse M, dont on connaît le remplissage, l'arbre d'élimination peut être construit comme suit :

- toutes les feuilles de l'arbre correspondent aux inconnues j, telles que $m_{ji} = 0$ pour tout $i < j$;
- un nœud j a pour père i, si i est le plus petit numéro de ligne tel que $m_{ij} \neq 0$.

Remarque 1.1 *La fabrication de cette arborescence doit prendre en compte les termes non nuls obtenus par remplissage au cours de l'élimination. On ne peut pas fabriquer l'arbre d'élimination uniquement à partir de la matrice creuse initiale : il faut connaître aussi les termes de remplissage.*

1.2 La phase d'analyse

Dans la phase d'analyse du solveur direct que nous mettons en œuvre, nous proposons l'algorithme de dissection emboîtée pour gérer le phénomène de remplissage. Le choix d'utiliser particulièrement cette méthode provient du fait qu'elle permet, d'une part, de faciliter le parallélisme plus

que les autres algorithmes de renumérotation (A.2) et que, d'autre part, elle conduit souvent à de meilleurs résultats.

1.2.1 Dissection emboîtée

La méthode de dissection emboîtée ordonne, avec l'aide de séparateurs, les sommets du graphe (ou nœuds du maillage) représentant la structure creuse de la matrice M du système. Le graphe non orienté $G(V, E)$ d'une matrice M de dimension n est composé d'un ensemble V de n sommets et d'un ensemble E d'arêtes reliant les sommets de V. Chaque sommet du graphe représente une inconnue du système linéaire. Chaque arête existant entre deux sommets i et j indique que le coefficient m_{ij} est non nul. Un séparateur est un petit ensemble de sommets qui découpe le reste du graphe en deux composants déconnectés $G(V_1, E_1)$ et $G(V_2, E_2)$.

Le principe

Le principe de la dissection emboîtée est basé sur la bissection récursive du graphe de la matrice M à factoriser. Une première bissection est effectuée en sélectionnant un ensemble de sommets formant un séparateur Γ. Ce séparateur est retiré du graphe et, cela génère une partition en deux sous-graphes déconnectés. Le séparateur est choisi de telle sorte que sa taille soit aussi petite que possible et que les tailles des sous-graphes obtenus soient équivalentes. Chacun de ces sous-graphes est alors découpé récursivement suivant le même processus, jusqu'à ce que les sous-structures I_i générées soient de tailles suffisamment petites.

Une illustration

Pour illustrer le principe de cette technique, nous considérons un petit système linéaire creux, où la structure de la matrice M est présentée sur la figure 1.3(a). Le graphe provenant du maillage associé aux inconnues du système est reproduit sur la figure 1.3(b).

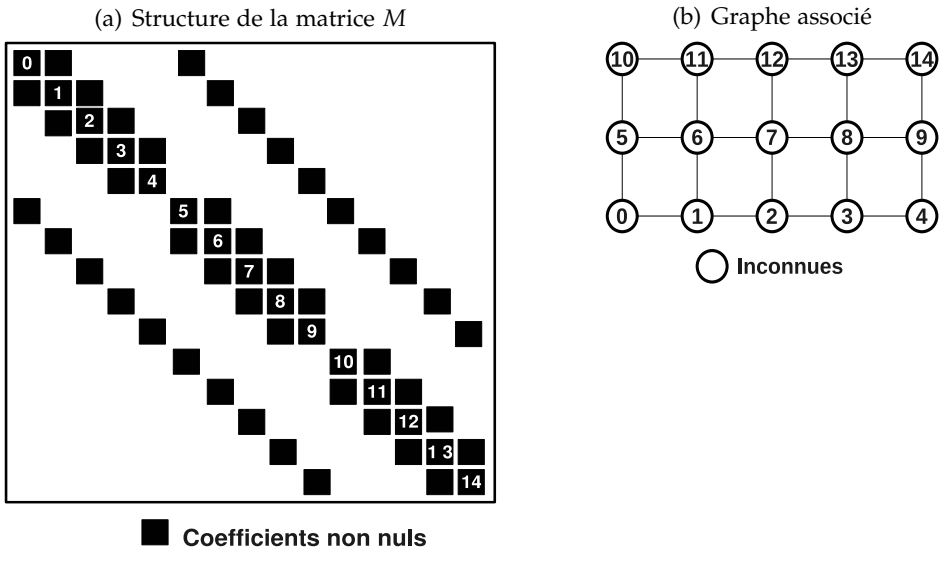


FIG. 1.3 – *Exemple de système linéaire creux.*

Sur cet exemple, nous effectuons d'abord une première bissection en sélectionnant l'ensemble des inconnues $\{2,7,12\}$. Avec ces nœuds nous formons le séparateur Γ_1^2. Ce séparateur est retiré du graphe et, cela génère une partition en deux sous-graphes déconnectés. Ensuite, nous découpons ces derniers suivant le même processus. Nous obtenons alors les séparateurs Γ_1^1 et Γ_2^1 qui sont respectivement formés des sommets $\{5,6\}$ et $\{8,9\}$. Enfin, nous générons avec les ensembles d'inconnues $\{0,1\}$, $\{10,11\}$, $\{3,4\}$ et $\{13,14\}$ les sous-structures I_i (figure 1.4). La bissection est effectuée avec l'aide de METIS (Karypis et Kumar 1995; 1999) qui est principalement un outil de découpage de graphes ou de maillages. Il vise à découper un graphe donné en k sous-graphes ayant des tailles proches pour assurer un bon équilibre lors d'une parallélisation sur k processeurs. D'autre part, il cherche à minimiser la taille des séparateurs pour réduire les communications résultantes.

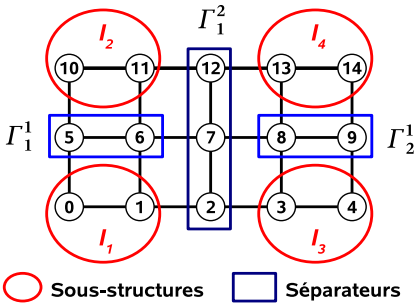


FIG. 1.4 – *Découpage par bissection récursive.*

Les sous-structures I_i et les séparateurs Γ_j^l générés sont utilisés pour réordonner les inconnues du système par paquets et ainsi construire un arbre d'élimination (figure 1.5), appelé **super-arbre d'élimination**. Les nœuds de cet arbre sont des super-nœuds, au sens où chacun d'eux est formé par l'ensemble des inconnues qui, au cours de l'élimination, ont les mêmes voisins au sens du graphe d'élimination. Les inconnues de chaque super-nœud seront éliminées tous ensemble lors de la factorisation numérique. Cela diminue le coût de la factorisation en regroupant les calculs (utilisation d'une bibliothèque d'algèbre linéaire de type BLAS-2 ou BLAS-3 (Dongarra et al. 1990)).

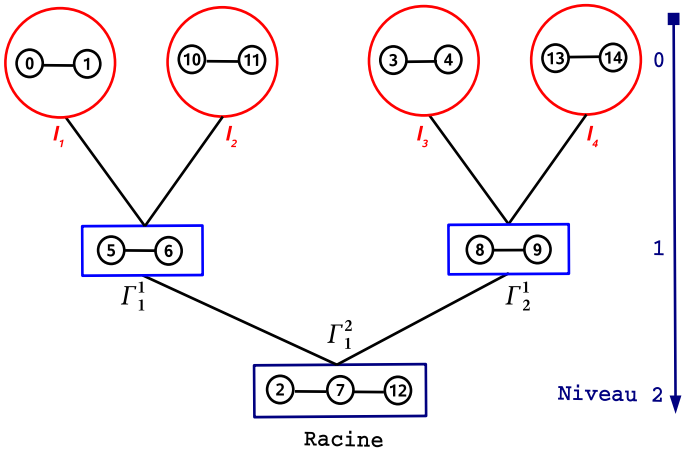


FIG. 1.5 – *Super-arbre d'élimination de hauteur* 3

Dans la technique de dissection emboîtée que nous avons exposée, le nombre de super-nœuds est une puissance de deux à chacun des niveaux **l** du super-arbre construit. Dans cet arbre, la racine se trouve au plus haut niveau et les feuilles sont au plus bas niveau. Si nous notons par **h**, la hauteur du super-arbre (le plus grand nombre de super-nœuds rencontrés pour aller d'une sous-structure à la racine). Le nombre de super-nœuds à un niveau **l** donné est alors égal à $\mathbf{2^{h-l-1}}$. Avec la stratégie de "division et conquête" de cette technique, la parallélisation du solveur peut se faire naturellement. Elle génère des super-arbres d'élimination larges et bien équilibrés permettant ainsi d'assurer un certain équilibrage des tâches de calcul. Ces arbres sont alors utilisés pour mener à bien les phases de mise en œuvre du solveur qui succèdent à celle-ci.

1.2.2 Factorisation symbolique

La factorisation symbolique de la matrice M consiste à déterminer le nombre et la position de ses nouveaux termes de remplissage qui seront créés au cours de la phase numérique. Cette procédure permet de construire la structure de données par blocs capable d'accueillir tous les coefficients de la matrice factorisée et donc d'optimiser le calcul effectif des coefficients des matrices triangulaires L et U. A priori, la factorisation symbolique de M requiert de faire toutes les itérations de l'algorithme d'élimination de Gauss en remplaçant le calcul des coefficients par un test sur leur pré-existence. Nous effectuons cela en incluant dans le super-arbre généré toutes les dépendances entre les inconnues du système linéaire.

Les super-arbres d'élimination et dépendances

La construction d'un super-arbre d'élimination qui contient toutes les dépendances entre les inconnues n'est pas une chose aisée si nous partons des outils traditionnels de partitionnement de graphes. Par exemple, METIS fournit toute une variété de programmes qui peuvent être utilisés pour partitionner des graphes ou des maillages, ainsi que des programmes pour renuméroter les inconnues d'un système linéaire creux. Dans ces programmes, on utilise un algorithme de partitionnement à plusieurs niveaux en cherchant à chaque niveau, un séparateur de taille minimale qui découpe le graphe en deux sous-graphes de tailles égales. Mais aucun de ces programmes ne fournit des informations sur ces séparateurs dont nous avons besoin pour la construction et la recherche des dépendances entre les inconnues. En effet, les calculs effectués sur une inconnue i d'un séparateur peuvent dépendre de l'élimination d'une inconnue j d'un supernœud situé à un niveau inférieur, si le terme m_{ij} est non nul. Nous voulons donc trouver pour chacun des super-nœuds dans l'arbre, toutes les inconnues qui sont en relation avec les siennes et celles de ses descendants. Nous qualifierons les inconnues propres à un super-nœud d'inconnues internes et celles qui sont dépendantes de ce super-nœud seront appelées inconnues interfaces.

Pour construire les super-arbres d'élimination renfermant toutes les dépendances, nous proposons donc deux approches qui dépendent du type de découpage choisi. Dans la première approche, le découpage est

effectué sur les éléments finis d'un maillage et il est géré par le partitionneur METIS. Cela ne nous permet pas de conserver toute l'historique sur la partition des éléments. Le nombre de sous-structures obtenues dans cette partition peut être une puissance de deux ou bien être quelconque. Dans la deuxième approche, nous effectuons directement le découpage sur les inconnues du système, et non plus sur les éléments du maillage. Cela assure le respect de l'ordre initial des inconnues, donc de la structure de la matrice. En découpant, nous intervenons dans les sous-programmes de METIS pour récupérer à chaque niveau les séparateurs trouvés. Le nombre de sous-structures présentes dans ce deuxième type de découpage ne s'écrit que sous la forme d'une puissance de deux.

La première approche consiste, en même temps, à reconstituer toute l'historique sur la partition du maillage éléments finis composée d'un nombre **ndoms** de sous-structuress I_i et à construire le super-arbre d'élimination contenant toutes les dépendances existantes entre les inconnues. Pour cela, nous mettons d'abord en place, grâce à l'algorithme 1.1, la structure de l'arbre en calculant sa hauteur **h** et le nombre **nbnode**(**l**) de super-nœuds situés à chacun de ses niveaux **l**. Ensuite, nous cherchons les nœuds (ou inconnues) internes et les nœuds (ou inconnues) interfaces de chacun des sous-domaines I_i que nous placerons au plus bas niveau de l'arbre ($l = 0$).

Algorithme 1.1 : Mise en place de la structure d'un super-arbre

Initialisation : $h = 1$ et $n = ndoms - 1$;
tant que $n \neq 0$ **faire**
  $h = h + 1$;
  calculer q, le quotient de la division de n par 2;
  $n = q$;
fin
$n = ndoms$;
$nbnode(0) = ndoms$;
pour $l = 1$ *à* $h - 1$ **faire**
  calculer q, le quotient de la division de n par 2;
  $nbnode(l) = q$;
  $rest = n - 2 * q$;
  $n = q + rest$;
fin

Nous bouclons cette première approche par l'application de l'algorithme 1.2 (défini à la page suivante) sur tous les nœuds interfaces des sous-structures. Nous formons alors les séparateurs Γ_j^l par la recherche de leurs nœuds internes et de leurs nœuds interfaces. Dans cette recherche, nous utilisons récursivement un **algorithme génétique** (Bui et Moon 1996) dans lequel chaque nouvelle génération **iter** permer de créer une population à partir de la précédente. La création de cette nouvelle population se fait par l'application des opérateurs génétiques que sont : la sélection, le croisement et la mutation. L'algorithme génétique se termine quand un certain critère d'arrêt est satisfait. Dans notre cas, ce critère est le nombre de générations **nbiter** que nous avons expérimentalement fixé à **10**.

Algorithme 1.2 : Construction d'un arbre avec l'algorithme génétique

Initialisation : $l = 1$ et $ncouples = ndoms$;
former tous les *couples* à partir des interfaces des $ndoms$ I_i;
tant que $l < h$ **faire**
  calculer q, le quotient de la division de $(ncouples + 1)$ par 2;
  créer une population initiale de *couples* de taille fixe;
  $ncouples = q$;
  $iter = 0$;
  tant que $iter < nbiter$ **faire**
    sélectionner les couples ayant plus de nœuds pour la reproduction;
    réaliser un croisement sur les *couples* sélectionnés;
    effectuer des mutations sur une faible proportion de *couples*;
    $iter = iter + 1$;
  fin
  pour $j = 1$ *à* $ncouples$ **faire**
    $n_1 = couples[j].first$;
    $n_2 = couples[j].second$;
    si $n_2 \geq 1$ **alors**
      Γ_j^l = l'ensemble des nœuds interfaces des super-nœuds ayant les positions n_1 et n_2 ;
    sinon
      $\Gamma_j^l = \emptyset$;
    fin
  fin
  chercher les nœuds internes et interfaces de chacun des $\Gamma_j^l \neq \emptyset$;
  placer au niveau l tous les super-nœuds $\Gamma_j^l \neq \emptyset$;
  former tous les *couples* à partir des interfaces des $ncouples$ Γ_j^l;
  $l = l + 1$;
fin

Pour rendre cette première approche plus claire, nous considérons donc l'exemple où nous disposons déjà d'un maillage éléments finis en dimension deux découpé en cinq sous-structures I_1, I_2, I_3, I_4 et I_5. Ce découpage est présenté sur la figure 1.6.

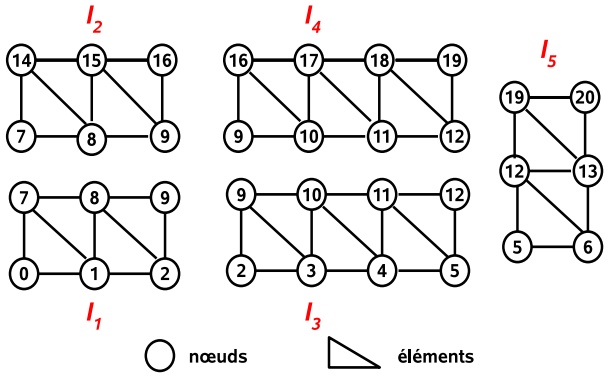


FIG. 1.6 – *Partition d'un maillage éléments finis*

Grâce au processus 1.1, la hauteur h du super-arbre est calculée et elle vaut 4. Quant au nombre de super-nœuds $nbnode(l)$ situés au niveau l, il est défini sur le tableau ci-dessous.

l	0	1	2	3
nbnode(l)	5	2	1	1

TAB. 1.1 – *Nombre de super-nœuds par niveau.*

L'illustration se poursuit par la recherche des nœuds internes et des nœuds interfaces de chacun des sous-domaines (figure 1.7). Par la suite, ces sous-domaines sont d'abord placés au niveau 0 du super-arbre et l'algorithme 1.2 est appliqué sur tous les interfaces de ces sous-domaines pour former les séparateurs Γ_j^l. A la fin, nous obtenons le super-arbre d'élimination présenté sur la figure 1.8.

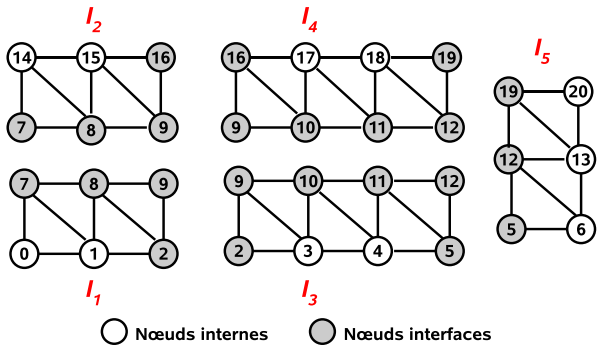


FIG. 1.7 – *Nœuds internes et interfaces des sous-domaines.*

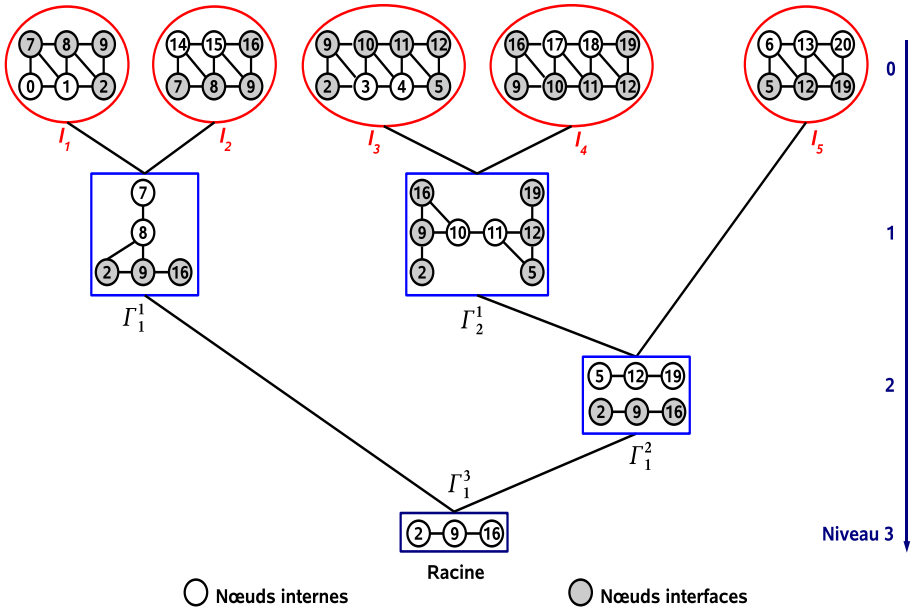


FIG. 1.8 – *Super-arbre d'élimination et dépendances : 1re approche.*

La deuxième approche consiste à compléter le super-arbre d'élimination, généré après un découpage par bissection récursive du graphe, en y incluant toutes les dépendances entre les inconnues. Pour cela, nous cherchons les inconnues interfaces de chacun des super-nœuds de l'arbre. Les

inconnues interfaces d'un super-nœud sont celles qui sont dépendantes de l'élimination de ses propres inconnues internes. A la racine, nous n'avons pas d'inconnues interfaces, puisqu'il n'y a aucune inconnue qui dépend de l'élimination de ses inconnues internes. Nous commençons donc la recherche aux séparateurs situés juste au-dessous de la racine et nous terminons par les sous-structures. Selon que nous sommes aux séparateurs ou aux sous-structures, la recherche est effectuée différemment. Pour la recherche, aux séparateurs nous utilisons l'algorithme 1.3 et, aux sous-structures nous utilisons l'algorithme 1.4.

Algorithme 1.3 : Recherche des inconnues interfaces des séparateurs

Entrées : h, la hauteur de l'arbre
Initialisation : l, un niveau dans l'arbre : $l = h - 2$;
tant que $l > 0$ **faire**
  si $2^{h-l-1} > 2$ **alors**
    pour $k = 1$ *à* 2^{h-l-1} **faire**
      Γ_p^{l+1} est le père de Γ_k^l;
      L2G = {inconnues internes et interfaces de Γ_p^{l+1}};
      allouer *mask*, un tableau d'entiers et l'initialiser à 1;
      pour $j \in$ **L2G faire**
        $mask(j) = 0$;
      fin
      $nf = 0$;
      pour $j \in \{\Gamma_k^l$ *ou ses descendants}* **faire**
        pour tous les i *tel que* $m_{ij} \neq 0$ *et* $mask(i) = 0$ **faire**
          $\mathbf{InterF}_k^l(nf) = i$;
          $mask(j) = 1$;
          $nf = nf + 1$;
        fin
      fin
    fin
  sinon
    L2G = {inconnues internes de la racine};
    pour $k = 1$ *et* 2 **faire**
      le tableau $\mathbf{InterF}_k^l$ de Γ_k^l est identique à **L2G** ;
    fin
  fin
  $l = l - 1$;
fin

A chaque itération l de l'algorithme 1.3, nous remplissons les tableaux $\mathbf{InterF}_k^l$ contenant les inconnues interfaces des séparateurs Γ_k^l ($1 \leq k \leq 2^{h-l-1}$). Si le nombre de séparateurs à un niveau l est supérieur à 2 alors chacun des tableaux $\mathbf{InterF}_k^l$ contiendra toutes les inconnues i appartenant à **L2G** ($mask(i) = 0$), telles que $m_{ij} \neq 0$. Le tableau **L2G** contient toutes les inconnues internes et interfaces de Γ_p^{l+1} (père de Γ_k^l) et l'indice j parcourt toutes les inconnues internes de Γ_k^l et celles de ses descendants. Dans le cas contraire, $\mathbf{InterF}_k^l$ ne contiendra que les inconnues internes de la racine.

Algorithme 1.4 : Recherche des interfaces des sous-structures

Entrées : h, la hauteur de l'arbre
pour $k = 1$ *à* 2^{h-1} **faire**
  allouer $mask$, un tableau d'entiers et l'initialiser à 0;
  pour $j \in I_k$ **faire**
    $mask(j) = 1$;
  fin
  $nf = 0$;
  pour $j \in I_k$ **faire**
    pour tous les i *tel que* $m_{ij} \neq 0$ *et* $mask(i) = 0$ **faire**
      $\mathbf{InterF}_k^0(nf) = i$;
      $nf = nf + 1$;
      $mask(i) = 1$;
    fin
  fin
fin

Dans l'algorithme 1.4, nous remplissons le tableau $\mathbf{InterF}_k^0$ par les inconnues interfaces de chacune des sous-structures I_k ($1 \leq k \leq 2^{h-1}$). Ce tableau contient toutes les inconnues i, non internes à I_k ($mask(i) = 0$) et telles que $m_{ij} \neq 0$.

En appliquant la deuxième approche sur l'exemple de la figure 1.5, nous obtenons sur la figure ci-dessous un super-arbre d'élimination qui inclut toutes les dépendances entre les inconnues. Les inconnues interfaces trouvées avec les algorithmes 1.3 et 1.4 sont en gris.

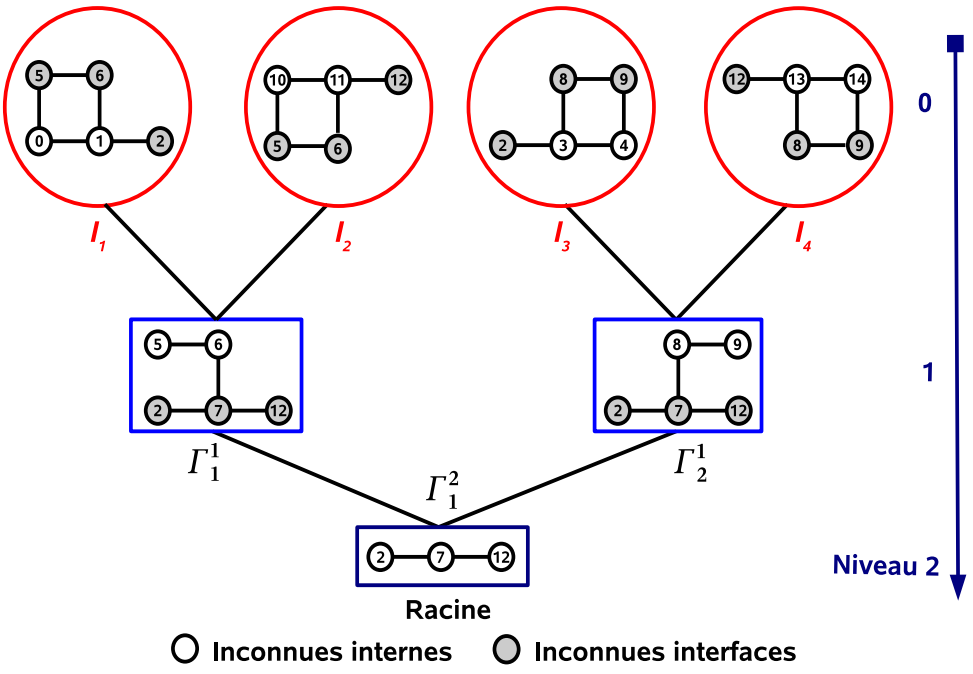


FIG. 1.9 – *Super-arbre d'élimination et dépendances : 2^e approche.*

La structure par blocs de la matrice factorisée

La structure par blocs de la matrice factorisée servira à stocker l'ensemble des coefficients de remplissage qui apparaîtront au cours de la factorisation numérique de la matrice globale M. Pour connaître à l'avance cette structure, nous allons opérer sur le super-arbre d'élimination qui a été construit avec l'une des deux approches précédentes. Cette opération se résume à associer à chacun des super-nœuds I_i ou Γ_j^l de ce super-arbre, un bloc matriciel diagonal correspondant à ses inconnues internes et des blocs extra-diagonaux correspondant aux interconnexions entre ses inconnues internes et celles de ses descendants. Les matrices $M_{I_iI_i}$ étant creuses, nous utilisons alors l'algorithme de Cuthill - McKee inverse (Cuthill et McKee 1969; Liu et Sherman 1975) pour renuméroter les inconnues associées aux sous-structures I_i. Cet algorithme permet de réduire efficacement la mémoire nécessaire au stockage en format "ligne de ciel" ou *skyline* du profil des matrices $M_{I_iI_i}$. Nous déterminons ainsi la structure de ces dernières sous la forme tridiagonale par blocs.

L'opération effectuée sur le super-arbre présenté sur la figure 1.9, par exemple, nous permet d'obtenir la structure de la matrice M factorisée (figure 1.10). Les blocs diagonaux $M_{I_iI_i}$ situés au niveau 0 correspondent aux matrices associées aux inconnues internes des sous-structures I_1, I_2, I_3 et I_4. Les blocs diagonaux $M_{\Gamma_j^l\Gamma_j^l}$ situés aux autres niveaux l correspondent aux matrices associées inconnues internes des séparateurs Γ_j^l. Les blocs extra-diagonaux $M_{I_i\Gamma_j^l}$ et $M_{\Gamma_j^lI_i}$ correspondent aux interconnexions entre les inconnues interfaces de I_i et celles internes à Γ_j^l ; les blocs extra-diagonaux $M_{\Gamma_i^l\Gamma_j^m}$ et $M_{\Gamma_j^m\Gamma_i^l}$ ($l < m$) correspondent aux interconnexions entre les inconnues interfaces de Γ_i^l et celles qui sont internes à Γ_j^m.

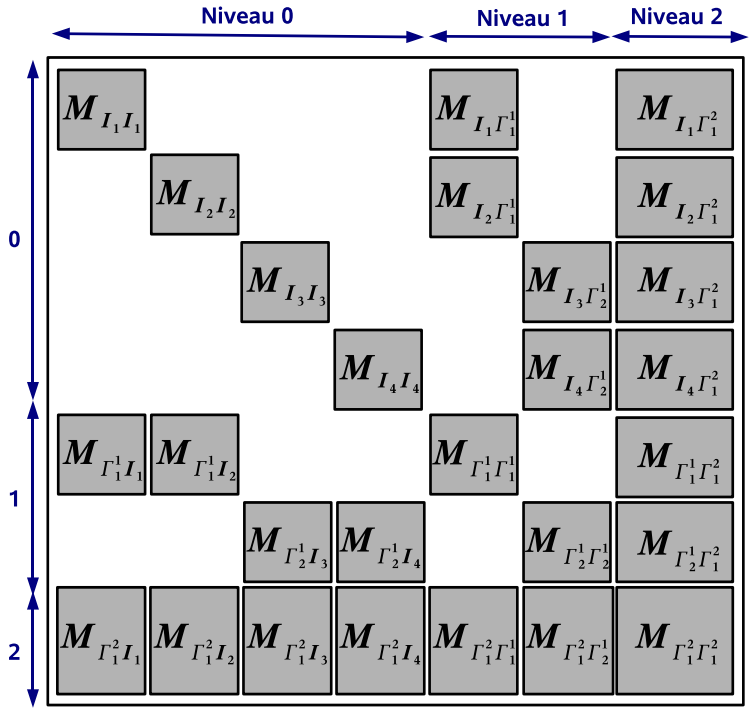


FIG. 1.10 – *Structure de la matrice M factorisée.*

1.3 La phase numérique

La phase numérique est une partie importante dans la mise en œuvre du solveur direct. Dans cette phase, nous alternons (niveau par niveau) l'étape de factorisation numérique et celle de condensation statique du système linéaire sur les inconnues appartenant aux séparateurs Γ_j^l. Avant d'effectuer ces deux étapes, nous prenons le soin de bien mettre à l'échelle la matrice originale M pour généraliser les types de systèmes linéaires que nous pouvons résoudre.

1.3.1 Mise à l'échelle ou "scaling" de la matrice originale

Pour la résolution de certains systèmes linéaires, il est souhaitable de les équilibrer en mettant à l'échelle la matrice M. En effet, quand l'ordre de grandeur des coefficients de M varie énormément d'un terme à l'autre, on risque de réaliser, au cours de la factorisation numérique, des opérations entre termes de tailles très différentes, ce qui peut faire échouer la résolution. Le *scaling* de M consiste à trouver une matrice diagonale inversible Δ telle que les éléments diagonaux de $\Delta M \Delta$ aient tous le même ordre de grandeur. Pour cela, nous choisissons les coefficients de la matrice Δ sous la forme donnée par la formule (1.1). La mise à l'échelle du système linéaire $MX = B$ se transforme alors en $(\Delta M \Delta)Y = \Delta B$, où $Y = \Delta^{-1}X$. Pour ne pas alourdir les notations dans la suite, nous désignerons par $MX = B$, le même système déjà mis à l'échelle.

$$\Delta_{ii} = \begin{cases} \dfrac{1}{\sqrt{|m_{ii}|}}, & \text{si } m_{ii} \neq 0 \\ \dfrac{1}{\sqrt{\max\limits_j |m_{ij}|}}, & \text{si } \max\limits_j |m_{ij}| \neq 0 \\ 1, & \text{sinon.} \end{cases} \tag{1.1}$$

1.3.2 Factorisation numérique

La factorisation numérique consiste à calculer implicitement la matrice triangulaire inférieure L et la matrice triangulaire supérieure U de M en s'appuyant sur le super-arbre généré dans la phase d'analyse. Pour cela, nous procédons d'abord à une factorisation creuse des matrices $M_{I_iI_i}$, associées aux sous-structures I_i, sous la forme $L_{I_iI_i}D_{I_iI_i}U_{I_iI_i}$ (ou $L_{I_iI_i}D_{I_iI_i}{L_{I_iI_i}}^T$ dans le cas symétrique). Nous utilisons une méthode frontale (Irons 1970) pour mieux tirer parti de la structure tridiagonale par blocs de ces matrices qui sont souvent très petites par rapport à la dimension de la matrice globale M. Nous passons ensuite à la factorisation de chacun des blocs matriciels $S_{\Gamma_j^l\Gamma_j^l}$ associés aux séparateurs Γ_j^l. Ces matrices sont créées à l'itération $l-1$ de l'étape de condensation statique et sont pleines. Leur factorisation consistera à les décomposer alors sous la forme $L_{\Gamma_j^l\Gamma_j^l}D_{\Gamma_j^l\Gamma_j^l}U_{\Gamma_j^l\Gamma_j^l}$ (dans le cas symétrique $U_{\Gamma_j^l\Gamma_j^l}$ est remplacée par ${L_{\Gamma_j^l\Gamma_j^l}}^T$) via une méthode d'élimination de Gauss avec pivotage complet. Durant cette phase de factorisation numérique, l'élimination des inconnues internes de chacun des super-nœuds

de l'arbre peut être effectuée dès lors que les calculs sur tous ses fils ont été réalisés.

Dans l'exemple de la section 1.2.1, nous décrivons les matrices triangulaires L et U sous une forme similaire à celle de M (figure 1.10), où les blocs diagonaux de L ont été décomposés sous la forme LDU (ou LDL^T dans le cas symétrique) :

$$L = \begin{pmatrix} M_{I_1I_1} & & & & & & \\ 0 & M_{I_2I_2} & & & & & \\ 0 & 0 & M_{I_3I_3} & & & & \\ 0 & 0 & 0 & M_{I_4I_4} & & & \\ M_{\Gamma_1^1I_1} & M_{\Gamma_1^1I_2} & 0 & 0 & S_{\Gamma_1^1\Gamma_1^1} & & \\ 0 & 0 & M_{\Gamma_2^1I_3} & M_{\Gamma_2^1I_4} & 0 & S_{\Gamma_2^1\Gamma_2^1} & \\ M_{\Gamma_1^2I_1} & M_{\Gamma_1^2I_2} & M_{\Gamma_1^2I_3} & M_{\Gamma_1^2I_4} & S_{\Gamma_1^2\Gamma_1^1} & S_{\Gamma_1^2\Gamma_2^1} & S_{\Gamma_1^2\Gamma_1^2} \end{pmatrix}$$

et

$$U = \begin{pmatrix} I & 0 & 0 & 0 & M_{I_1I_1}^{-1}M_{I_1\Gamma_1^1} & 0 & M_{I_1I_1}^{-1}M_{I_1\Gamma_1^2} \\ & I & 0 & 0 & M_{I_2I_2}^{-1}M_{I_2\Gamma_1^1} & 0 & M_{I_2I_2}^{-1}M_{I_2\Gamma_1^2} \\ & & I & 0 & 0 & M_{I_3I_3}^{-1}M_{I_3\Gamma_2^1} & M_{I_3I_3}^{-1}M_{I_3\Gamma_1^2} \\ & & & I & 0 & M_{I_4I_4}^{-1}M_{I_4\Gamma_1^2} & M_{I_4I_4}^{-1}M_{I_4\Gamma_1^2} \\ & & & & I & 0 & S_{\Gamma_1^1\Gamma_1^1}^{-1}S_{\Gamma_1^1\Gamma_1^2} \\ & & & & & I & S_{\Gamma_2^1\Gamma_2^1}^{-1}S_{\Gamma_2^1\Gamma_1^2} \\ & & & & & & I \end{pmatrix}$$

1.3.3 Condensation statique

L'idée de la condensation statique est d'inverser un système local pour chacun des nœuds de l'arbre pour assembler le complément de Schur global S. Pour cela, nous éliminons d'abord les inconnues dans chacune des sous-structures I_i. L'élimination de ces inconnues équivaut au calcul des compléments de Schur locaux ou des contributions $M_{\Gamma_j^l I_i} {M_{I_iI_i}}^{-1} M_{I_i\Gamma_k^m}$ de ces sous-structures. Ces contributions permettent d'assembler les blocs matriciels $S_{\Gamma_j^l\Gamma_k^m}$. Elles sont calculées efficacement avec l'aide de la bibliothèque BLAS-3. L'assemblage des blocs situés au niveau suivant ($l = 1$ et $m = 1$) est alors effectué complètement et celui des blocs situés aux niveaux plus élevés ($l > 1$ et $m > 1$) est effectué partiellement. La condensation statique se poursuit ensuite jusqu'à la racine de l'arbre. Nous éliminons les inconnues de chacun des nœuds Γ_j^l situé à un niveau $l > 0$ et nous calculons leurs contributions $S_{\Gamma_k^m\Gamma_j^l} {S_{\Gamma_j^l\Gamma_j^l}}^{-1} S_{\Gamma_j^l\Gamma_p^n}$ qui seront apportées aux blocs $S_{\Gamma_k^m\Gamma_p^n}$ localisés aux niveaux supérieurs m et n. A la racine (niveau $h-1$), nous n'aurons pas à calculer des contributions, mais nous assemblerons tout de même le bloc diagonal $S_{\Gamma_1^{h-1}\Gamma_1^{h-1}}$.

Dans le cas de l'exemple de la section 1.2.1, le complément de Schur S est défini par la formule :

$$S = \begin{pmatrix} S_{\Gamma_1^1\Gamma_1^1} & 0 & S_{\Gamma_1^1\Gamma_1^2} \\ 0 & S_{\Gamma_2^1\Gamma_2^1} & S_{\Gamma_2^1\Gamma_1^2} \\ S_{\Gamma_1^2\Gamma_1^1} & S_{\Gamma_1^2\Gamma_2^1} & S_{\Gamma_1^2\Gamma_1^2} \end{pmatrix}. \tag{1.2}$$

Les blocs matriciels $S_{\Gamma_j^l \Gamma_k^m}$ au niveau 1 sont assemblés par les équations (1.3) et (1.4). Le bloc matriciel $S_{\Gamma_1^2 \Gamma_1^2}$ à la racine (niveau 2) est assemblé par l'équation (1.5).

$$\begin{cases} S_{\Gamma_1^1\Gamma_1^1} = M_{\Gamma_1^1\Gamma_1^1} - \sum_{i=1}^{2} M_{\Gamma_1^1 I_i} {M_{I_i I_i}}^{-1} M_{I_i \Gamma_1^1} \\ S_{\Gamma_1^1\Gamma_1^2} = M_{\Gamma_1^1\Gamma_1^2} - \sum_{i=1}^{2} M_{\Gamma_1^1 I_i} {M_{I_i I_i}}^{-1} M_{I_i \Gamma_1^2} \\ S_{\Gamma_1^2\Gamma_1^1} = M_{\Gamma_1^2\Gamma_1^1} - \sum_{i=1}^{2} M_{\Gamma_1^2 I_i} {M_{I_i I_i}}^{-1} M_{I_i \Gamma_1^1} \end{cases} \tag{1.3}$$

$$\begin{cases} S_{\Gamma_2^1\Gamma_2^1} = M_{\Gamma_2^1\Gamma_2^1} - \sum_{i=3}^{4} M_{\Gamma_2^1 I_i} {M_{I_i I_i}}^{-1} M_{I_i \Gamma_2^1} \\ S_{\Gamma_2^1\Gamma_1^2} = M_{\Gamma_2^1\Gamma_1^2} - \sum_{i=3}^{4} M_{\Gamma_2^1 I_i} {M_{I_i I_i}}^{-1} M_{I_i \Gamma_1^2} \\ S_{\Gamma_1^2\Gamma_2^1} = M_{\Gamma_1^2\Gamma_2^1} - \sum_{i=3}^{4} M_{\Gamma_1^2 I_i} {M_{I_i I_i}}^{-1} M_{I_i \Gamma_2^1} \end{cases} \tag{1.4}$$

$$S_{\Gamma_1^2\Gamma_1^2} = M_{\Gamma_1^2\Gamma_1^2} - \sum_{i=1}^{4} M_{\Gamma_1^2 I_i} {M_{I_i I_i}}^{-1} M_{I_i \Gamma_1^2} - \sum_{j=1}^{2} S_{\Gamma_1^2\Gamma_j^1} {S_{\Gamma_j^1\Gamma_j^1}}^{-1} S_{\Gamma_j^1\Gamma_1^2} \tag{1.5}$$

1.4 La résolution des systèmes triangulaires

La solution du système linéaire creux $MX = B$ est déterminée en résolvant les systèmes triangulaires $LY = B$ et $UX = Y$. Les matrices triangulaires L et U sont issues de la factorisation numérique de M et, X, Y et B sont des matrices colonnes. Nous résolvons ce système linéaire à second membre simple ou multiple en trois phases :

- une phase de descente ;
- une phase de résolution d'un problème condensé ;
- une phase de remontée.

1.4.1 Phase de descente

Cette étape de descente consiste d'une part, à résoudre les systèmes locaux $(L_{I_i I_i} D_{I_i I_i} U_{I_i I_i}) Y_{I_i} = B_{I_i}$ associés aux sous-structures I_i et d'autre part, à mettre à jour les termes $B_{\Gamma_j^l}$ du second membre B. Si une sous-structure I_i est descendante d'un séparateur Γ_j^l alors nous retranchons le terme $M_{\Gamma_j^l I_i} Y_{I_i}$ à $B_{\Gamma_j^l}$. Dans l'exemple de la section 1.2.1, la mise à jour des termes $B_{\Gamma_1^1}$, $B_{\Gamma_2^1}$ et $B_{\Gamma_1^2}$ est effectuée de la façon suivante :

$$\begin{pmatrix} B_{\Gamma_1^1} \\ B_{\Gamma_2^1} \\ B_{\Gamma_1^2} \end{pmatrix} = \begin{pmatrix} B_{\Gamma_1^1} \\ B_{\Gamma_2^1} \\ B_{\Gamma_1^2} \end{pmatrix} - \begin{pmatrix} M_{\Gamma_1^1 I_1} Y_{I_1} + M_{\Gamma_1^1 I_2} Y_{I_2} \\ M_{\Gamma_2^1 I_3} Y_{I_3} + M_{\Gamma_2^1 I_4} Y_{I_4} \\ \sum_{i=1}^{4} M_{\Gamma_1^2 I_i} Y_{I_i} \end{pmatrix}. \tag{1.6}$$

1.4.2 Résolution du problème condensé

Dans cette étape, nous résolvons niveau par niveau le problème condensé aux inconnues appartenant aux séparateurs Γ_j^l. Les solutions $X_{\Gamma_j^l}$ sont obtenues par une phase élimination suivie d'une phase de substitution.

Dans la phase d'élimination, nous calculons à chacun des niveaux l les solutions $Y_{\Gamma_j^l}$ par $S_{\Gamma_j^l\Gamma_j^l} Y_{\Gamma_j^l} = B_{\Gamma_j^l}$. Puis, nous mettons à jour les termes $B_{\Gamma_k^m}$ aux niveaux supérieurs $m > l$ avec la formule (1.7), où les séparateurs Γ_j^l sont les descendants de Γ_k^m dans le super-arbre.

$$B_{\Gamma_k^m} = B_{\Gamma_k^m} - \sum_{j,l/l<m} S_{\Gamma_k^m\Gamma_j^l} Y_{\Gamma_j^l} \tag{1.7}$$

Dans la phase de substitution, nous calculons les solutions $X_{\Gamma_j^l}$ en commençant par celle associée à la racine. Nous descendons ensuite l'arbre jusqu'au niveau 1. A chacun des niveaux l, nous obtenons ces solutions par la formule (1.8) dans laquelle les séparateurs Γ_k^m sont les ascendants du super-nœud Γ_j^l de l'arbre.

$$X_{\Gamma_j^l} = Y_{\Gamma_j^l} - \sum_{k,m/m>l} S^{-1}_{\Gamma_j^l\Gamma_j^l} S_{\Gamma_j^l\Gamma_k^m} Y_{\Gamma_k^m} \tag{1.8}$$

Avec l'exemple de la section 1.2.1, les phases d'élimination et de substitution sont réalisées en résolvant respectivement les systèmes linéaires d'équations (1.9) et (1.10).

$$\begin{pmatrix} S_{\Gamma_1^1\Gamma_1^1} & 0 & 0 \\ 0 & S_{\Gamma_2^1\Gamma_2^1} & 0 \\ S_{\Gamma_1^2\Gamma_1^1} & S_{\Gamma_1^2\Gamma_2^1} & S_{\Gamma_1^2\Gamma_1^2} \end{pmatrix} \begin{pmatrix} Y_{\Gamma_1^1} \\ Y_{\Gamma_2^1} \\ Y_{\Gamma_1^2} \end{pmatrix} = \begin{pmatrix} B_{\Gamma_1^1} \\ B_{\Gamma_2^1} \\ B_{\Gamma_1^2} \end{pmatrix} \tag{1.9}$$

$$\begin{pmatrix} I & 0 & S^{-1}_{\Gamma_1^1\Gamma_1^1} S_{\Gamma_1^1\Gamma_1^2} \\ 0 & I & S^{-1}_{\Gamma_2^1\Gamma_2^1} S_{\Gamma_2^1\Gamma_1^2} \\ 0 & 0 & I \end{pmatrix} \begin{pmatrix} X_{\Gamma_1^1} \\ X_{\Gamma_2^1} \\ X_{\Gamma_1^2} \end{pmatrix} = \begin{pmatrix} Y_{\Gamma_1^1} \\ Y_{\Gamma_2^1} \\ Y_{\Gamma_1^2} \end{pmatrix} \tag{1.10}$$

1.4.3 Phase de remontée

La phase de remontée repose sur le calcul des solutions locales X_{I_i} associées aux sous-structures. Nous trouvons la solution rattachée à chacune des sous-structures I_i par la formule (1.11). Dans cette formule, les supernœuds Γ_j^l sont les ascendants de la sous-structure I_i.

$$X_{I_i} = Y_{I_i} - \sum_{j,l/l\geq 1} M^{-1}_{I_iI_i} M_{I_i\Gamma_j^l} X_{\Gamma_j^l} \tag{1.11}$$

En considérant l'exemple de la section 1.2.1, nous calculons les solutions X_{I_1}, X_{I_2}, X_{I_3} et X_{I_1}, avec la formule ci-dessous.

$$\begin{pmatrix} X_{I_1} \\ X_{I_2} \\ X_{I_3} \\ X_{I_4} \end{pmatrix} = \begin{pmatrix} Y_{I_1} \\ Y_{I_2} \\ Y_{I_3} \\ Y_{I_4} \end{pmatrix} - \begin{pmatrix} M^{-1}_{I_1I_1} M_{I_1\Gamma_1^1} X_{\Gamma_1^1} + M^{-1}_{I_1I_1} M_{I_1\Gamma_1^2} X_{\Gamma_1^2} \\ M^{-1}_{I_2I_2} M_{I_2\Gamma_1^1} X_{\Gamma_1^1} + M^{-1}_{I_2I_2} M_{I_2\Gamma_1^2} X_{\Gamma_1^2} \\ M^{-1}_{I_3I_3} M_{I_3\Gamma_2^1} X_{\Gamma_2^1} + M^{-1}_{I_3I_3} M_{I_3\Gamma_1^2} X_{\Gamma_1^2} \\ M^{-1}_{I_4I_4} M_{I_4\Gamma_2^1} X_{\Gamma_2^1} + M^{-1}_{I_4I_4} M_{I_4\Gamma_1^2} X_{\Gamma_1^2} \end{pmatrix} \tag{1.12}$$

1.5 La prise en compte des systèmes non inversibles

Il existe beaucoup de solveurs directs séquentiels ou parallèles, mais peu d'entre eux prennent en compte les singularités des systèmes non inversibles. Ces singularités sont, soit d'origine physique ou géométrique, soit dues au découpage issu de certaines méthodes de décomposition de domaine. Notre but est d'avoir un solveur qui traite automatiquement et proprement les modes à énergie nulle de ces systèmes.

1.5.1 Détection locale des singularités

La détection consiste à chercher les singularités locales durant la factorisation numérique de M en parcourant le super-arbre niveau par niveau. Nous commençons d'abord le processus de recherche sur les matrices $M_{I_iI_i}$ associées aux sous-structures. Ensuite, nous progressons jusqu'à la racine en appliquant le même processus sur les blocs $S_{\Gamma_j^l\Gamma_j^l}$. A chacune des itérations k de la factorisation LDU d'un bloc $M_{I_iI_i}$ (resp. $S_{\Gamma_j^l\Gamma_j^l}$), il s'agit de :

1. chercher si un pivot proche de zéro apparaît pendant l'élimination de l'inconnue k, c'est-à-dire vérifier si $D_{I_iI_i}(k,k)$ (resp. $D_{\Gamma_j^l\Gamma_j^l}(k,k)$), satisfait le critère (1.13). Dans le cas affirmatif,
2. renvoyer le traitement de l'équation k, dont le numéro global est k_g, à la fin de la factorisation de M :
   - annuler les lignes k de $M_{I_iI_i}$ et $M_{I_i\Gamma_j^l}$ (resp. $S_{\Gamma_j^l\Gamma_j^l}$ et $S_{\Gamma_j^l\Gamma_k^m}$, où $m > l > 0$) ;
   - annuler les colonnes k de $M_{I_iI_i}$ et $M_{\Gamma_j^lI_i}$ (resp. $S_{\Gamma_j^l\Gamma_j^l}$ et $S_{\Gamma_k^m\Gamma_j^l}$, où $m > l > 0$) ;
   - fixer à l'unité le terme diagonal $M_{I_iI_i}(k,k)$ (resp. $M_{\Gamma_j^l\Gamma_j^l}(k,k)$).

$$\begin{cases} |D(k,k)| < \boldsymbol{\varepsilon} * \max\limits_{1\leq p\leq k} |A(p,p)| \ \text{ avec } \ \boldsymbol{\varepsilon} \ \text{ constante fixée,} \\ D = D_{I_iI_i} \ (\text{resp. } D_{\Gamma_j^l\Gamma_j^l}) \ \text{ et } \ A = M_{I_iI_i} \ (\text{resp. } S_{\Gamma_j^l\Gamma_j^l}) \end{cases} \tag{1.13}$$

Dans la pratique, les pivots proche de zéro rencontrés lors de la décomposition LDU d'une matrice singulière A sont habituellement contrôlés par le critère (1.13). Pour les types de problèmes que nous traitons, la valeur $\boldsymbol{\varepsilon = 10^{-6}}$ permet de faire la distinction entre les pivots proches de zéro dus aux singularités et ceux qui sont grands. Mais, en général, la valeur idéale de $\boldsymbol{\varepsilon}$ est difficile, voire impossible à prédire sans erreurs. Si la valeur sélectionnée pour la constante arbitraire $\boldsymbol{\varepsilon}$ s'avère être trop petite ou trop grand, les modes à énergie nulle de A pourraient être mal déterminés. Cela peut conduire à un échec des calculs.

1.5.2 Traitement des modes à énergie nulle

Le traitement des modes à énergie nulle coïncide avec la fin de la factorisation et consiste à chercher les singularités globales du système. Pour cela, dans un premier temps, nous extrayons, pour chacune des singularités k détectée, la colonne (resp. ligne) k_g correspondante dans la matrice

globale M et nous la copions dans une matrice B^s (resp. C^s), où la ligne k_g (resp. colonne k_g) est mise à zéro. Le coefficient diagonal $m_{k_g k_g}$ est aussi extrait et copié dans une matrice A^s. Nous formons alors le système augmenté (1.14), où $\widetilde{M}$ est la matrice M modifiée pendant sa factorisation.

$$M^{aug} = \begin{pmatrix} \widetilde{M} & B^s \\ C^s & A^s \end{pmatrix} \tag{1.14}$$

Ce système est donc condensé sur toutes les singularités détectées pour obtenir en deux étapes une petite matrice pleine $S^s = A^s - C^s \widetilde{M}^{-1} B^s$. Nous calculons d'abord la matrice $\widetilde{N} = \widetilde{M}^{-1} B^s$ en effectuant simultanément autant de descentes-remontées qu'il y a de singularités. Ensuite, nous faisons le produit entre la matrice C^s et $\widetilde{N}$.

Dans un deuxième temps, nous effectuons une élimination de Gauss avec pivotage sur la matrice S^s. Les pivots nuls rencontrés lors de cette élimination sont alors les singularités globales. Elles sont donc utilisées pour extraire de $\widetilde{N}$ les colonnes sur lesquelles nous allons nous appuyer pour constituer une base N du noyau de M. Pour chacune des singularités globales k_g, nous mettons le terme $N(k_g, k)$ à 1.

1.5.3 Illustration du déroulement de l'algorithme

A titre d'illustration, nous considérons un petit système linéaire symétrique comportant neuf inconnues numérotées à partir de zéro. Le graphe représentant la structure creuse de ce système est montré sur la figure 1.11(a). Un découpage de ce graphe en quatre sous-structures I_1, I_2, I_3 et I_4 est donné sur la figure 1.11(b). Chacune de ces sous-structures ne contient qu'une seule inconnue.

(a) Graphe de M (b) Découpage en quatre sous-structures

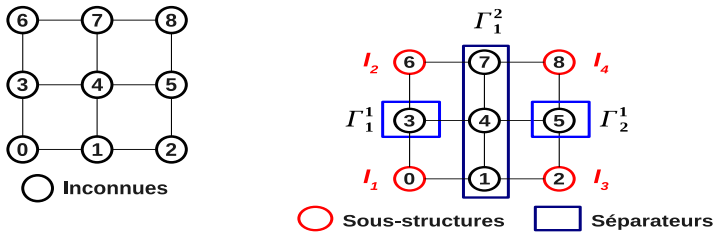


FIG. 1.11 – *Exemple de système linéaire singulier.*

Les coefficients diagonaux de la matrice initiale (l'équation (1.15) à gauche) et celles de la matrice renumérotée (l'équation (1.15) à droite) sont symboliquement non nuls, mais ils sont numériquement nuls rendant ainsi le système global non inversible. La dimension du noyau de ce système linéaire est 3.

$$M = \begin{array}{c|ccccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & & & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 3 & & & & 0 & 1 & 0 & 1 & 0 & 0 \\ 4 & & & & & 0 & 1 & 0 & 1 & 0 \\ 5 & & & & & & 0 & 0 & 0 & 1 \\ 6 & & & X & & & & 0 & 1 & 0 \\ 7 & & & & & & & & 0 & 1 \\ 8 & & & & & & & & & 0 \end{array} \qquad M = \begin{array}{c|ccccccccc} & 0 & 6 & 2 & 8 & 3 & 5 & 1 & 4 & 7 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 6 & & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 2 & & & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 8 & & & & 0 & 0 & 1 & 0 & 0 & 1 \\ 3 & & & & & 0 & 0 & 0 & 1 & 0 \\ 5 & & & & & & 0 & 0 & 1 & 0 \\ 1 & & & X & & & & 0 & 1 & 0 \\ 4 & & & & & & & & 0 & 1 \\ 7 & & & & & & & & & 0 \end{array} \tag{1.15}$$

Les termes $M_{I_iI_i}$ associés aux sous-structures étant nuls, les inconnues 0, 6, 2 et 8 appartenant à ces sous-structures sont alors détectées comme des singularités locales. Nous renvoyons donc le traitement des équations 0, 6, 2 et 8 à la fin de la factorisation de M en mettant à zéro les coefficients extra-diagonaux $M_{I_i\Gamma_j^1}$ $(j = 1,2)$ et $M_{I_i\Gamma_1^2}$ $(1 \leq i \leq 4)$. Nous en déduisons que les contributions apportées par l'élimination de chacune de ces inconnues sont nulles. Ceci nous permet de calculer de la manière suivante les compléments de Schur locaux sur les inconnues appartenant aux séparateurs Γ_1^1, Γ_2^1 et Γ_1^2 :

$$\begin{cases} S_{\Gamma_1^1\Gamma_1^1} = M_{\Gamma_1^1\Gamma_1^1} = 0 \text{ et } S_{\Gamma_1^1\Gamma_1^2} = M_{\Gamma_1^1\Gamma_1^2} = 0 \\ S_{\Gamma_2^1\Gamma_2^1} = M_{\Gamma_2^1\Gamma_2^1} = 0 \text{ et } S_{\Gamma_2^1\Gamma_1^2} = M_{\Gamma_2^1\Gamma_1^2} = 0 \end{cases}. \tag{1.16}$$

Avec les équations de (1.16), les inconnues 3 et 5 sont, à leur tour détectées comme des singularités locales lors de la décomposition LDL^t des blocs $S_{\Gamma_1^1\Gamma_1^1}$ et $S_{\Gamma_2^1\Gamma_2^1}$. Nous renvoyons alors le traitement des équations 3 et 5 à la fin en mettant à zéro les matrices $S_{\Gamma_1^1\Gamma_1^2}$ et $S_{\Gamma_2^1\Gamma_1^2}$. Les contributions apportées par l'élimination des 3 et 5 sont donc nulles. Nous en déduisons donc $S_{\Gamma_1^2\Gamma_1^2}$ par l'équation :

$$S_{\Gamma_1^2\Gamma_1^2} = M_{\Gamma_1^2\Gamma_1^2} = \begin{array}{c|ccc} & 1 & 4 & 7 \\ 1 & 0 & 1 & 0 \\ 4 & & 0 & 1 \\ 7 & X & & 0 \end{array}. \tag{1.17}$$

Lors de la décomposition LDL^T de la matrice $S_{\Gamma_1^2\Gamma_1^2}$, l'inconnue 7 est détectée comme étant la dernière singularité locale. Nous cherchons maintenant les singularités globales parmi celles détectées. Pour cela, nous formons d'abord, la matrice colonne B^s par une copie des colonnes 0, 2, 3, 5, 6, 7 et 8 de la matrice initiale M et nous construisons la matrice carrée A_s avec les lignes 0, 2, 3, 5, 6, 7 et 8 de B^s (l'équation à gauche de (1.18)). Par la suite, ces mêmes lignes sont mises à zéro dans B^s (l'équation à droite de (1.18)). La matrice M modifiée au cours de la factorisation, notée $\widetilde{M}$, est égale à l'identité.

$$A^s = \begin{array}{c|ccccccc} & 0 & 2 & 3 & 5 & 6 & 7 & 8 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 2 & & 0 & 0 & 1 & 0 & 0 & 0 \\ 3 & & & 0 & 0 & 1 & 0 & 0 \\ 5 & & & & 0 & 0 & 0 & 1 \\ 6 & & X & & & 0 & 1 & 0 \\ 7 & & & & & & 0 & 1 \\ 8 & & & & & & & 0 \end{array} \text{ et } B^s = \begin{array}{c|ccccccc} & 0 & 2 & 3 & 5 & 6 & 7 & 8 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} \tag{1.18}$$

Nous condensons ensuite le système linéaire global sur les singularités détectées. Nous obtenons alors la matrice pleine S^s de dimension 6 :

$$S^s = A^s - (B^s)^t \widetilde{M}^{-1} B^s = A^s - (B^s)^t B^s. \tag{1.19}$$

Nous effectuons enfin une factorisation par élimination de Gauss avec pivotage complet sur la matrice condensée S^s pour détecter les 3 vrais modes à énergie nullle du système linéaire global.

1.6 L'évaluation des performances

Le solveur direct a été intégré dans le code de calcul ZéBuLon. Dans cette section, nous évaluons ses performances. Les performances d'un solveur direct peuvent être évaluées de différentes manières. Concernant celui que nous avons mis en œuvre, nous nous intéressons aux temps CPU (en secondes) nécessaires pour effectuer la phase d'analyse, la phase numérique et la résolution des systèmes triangulaires. Pour cela, nous comparons d'abord les deux approches étudiées pour analyser un système linéaire donné. Puis, nous étudions l'influence du nombre de sous-structures, engendrées par la dissection emboîtée, sur les performances. Ensuite, nous comparons les temps de calcul en séquentiel du solveur avec ceux d'autres solveurs disponibles dans le code. Enfin, nous analysons la complexité du solveur dans la phase numérique.

Les tests d'évaluation portent sur des problèmes d'élasticité linéaire posés dans des domaines cubiques. Ces tests sont effectués en séquentiel sur une machine bi-processeurs Intel Quad-Core Xeon 64-bit, avec une mémoire vive de 32 Go et une fréquence de 3.16 GHz.

1.6.1 Comparaison des approches proposées dans la phase d'analyse

Dans la phase d'analyse, nous avons développé deux approches pour construire les super-arbres d'élimination renfermant toutes les dépendances entre les inconnues du système linéaire à résoudre. La première approche (**Approche 1**) part d'un découpage sur éléments finis du maillage associé au problème alors que la deuxième approche (**Approche 2**) intervient directement sur les inconnues du problème. Nous comparons ces deux approches selon deux critères : la taille des problèmes et le nombre de sous-structures issues de la dissection emboîtée. Pour ce faire, nous considérons deux problèmes de tailles différentes (**41472** et **107811** degrés de liberté) et nous évaluons les temps de calcul nécessaires pour analyser et exécuter ces problèmes en fonction du nombre de sous-structures et suivant l'approche que nous employons. Nous présentons les temps d'analyse de ces systèmes sur les figures 1.12 et 1.13, et les temps d'exécution totale sur les figures 1.14 et 1.15.

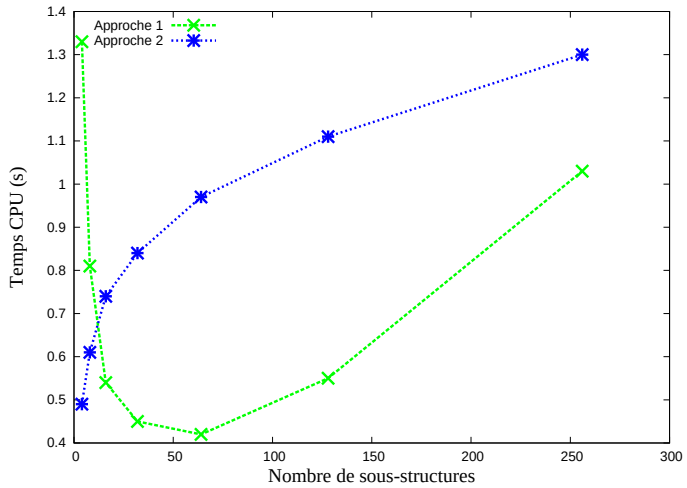


Fig. 1.12 – *Temps CPU d'analyse : problème de* 41472 *ddl.*

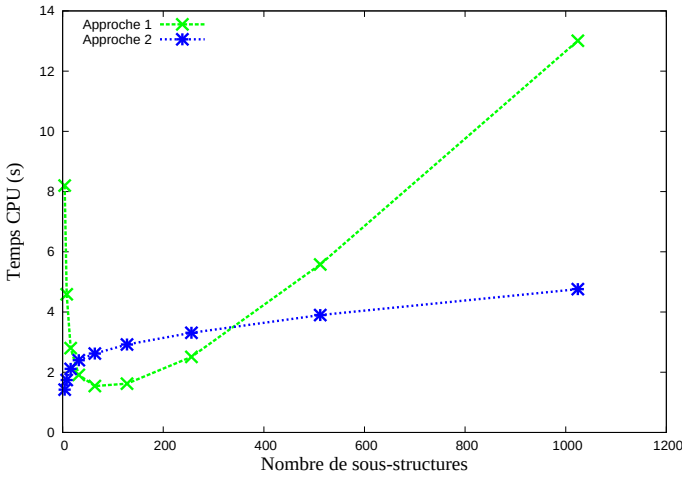


FIG. 1.13 – *Temps CPU d'analyse : problème de* 107811 *ddl.*

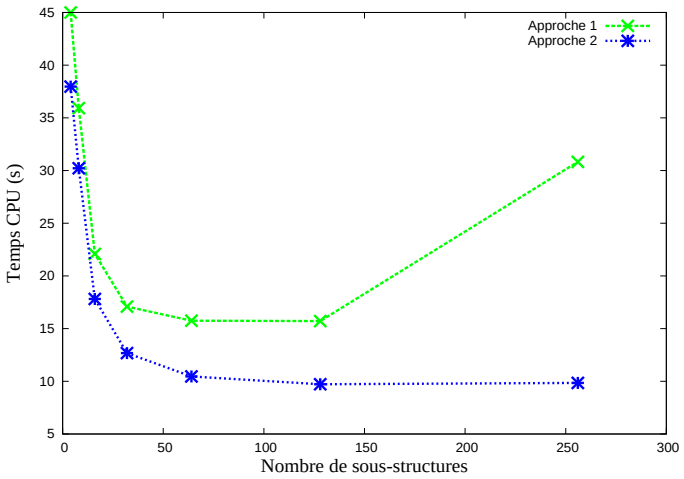


FIG. 1.14 – *Temps CPU d'exécution : problème de* 41472 *ddl.*

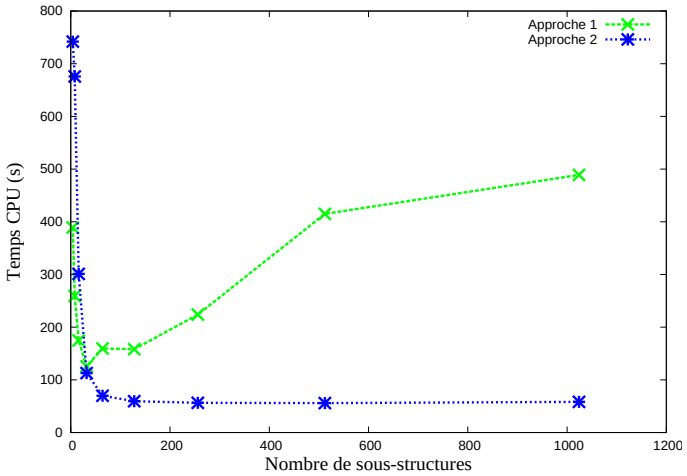


FIG. 1.15 – *Temps CPU d'exécution : problème de* 107811 *ddl.*

L'observation de ces résultats montre que, quelle que soit la taille du problème, les temps d'analyse sont faibles, voire négligeables par rapport aux temps d'exécution totale. Quand nous utilisons la deuxième approche

avec un nombre de sous-structures très élevé (supérieur à **128**), les temps d'exécution obtenus sont plus intéressants. En plus, c'est dans cette zone que les temps d'exécution des problèmes de grandes tailles sont optimaux avec cette deuxième approche même si les temps nécessaires pour analyser ces problèmes soit plus petits avec la première approche.

La comparaison effectuée montre que la deuxième approche offre une alternative intéressante à la première pour construire des super-arbres d'élimination dans la phase d'analyse du solveur. En particulier, les expériences menées prouvent que la deuxième approche est plus robuste et mieux adaptée dans la phase analyse des systèmes linéaires. Cette comparaison nous a donc permis d'adopter la deuxième approche dans cette phase. Neanmoins, pour un problème donné, il convient d'évaluer ou de donner une indication sur le nombre optimal de sous-structures dans le découpage qui mène à des temps d'exécution aussi petits.

1.6.2 Évaluation du nombre optimal de sous-structures

Afin de déterminer le nombre optimal de sous-structures engendrées par bissection récursive et conduisant à de meilleures performances du solveur, nous résolvons d'abord des problèmes de tailles **neqs** différentes (**41472**, **107811** et **397953** équations), où chacun d'eux est partitionné en un nombre **ndoms** de sous-domaines (**64**, **128**, **256**, **512** et **1024**). Ensuite, nous analysons les temps de calcul nécessaires pour leur résolution. Mais auparavant, nous porterons une attention particulière sur la qualité des découpages et leurs effets sur les temps écoulés dans la phase numérique.

Les tableaux 1.2, 1.3 et 1.4 présentent respectivement les résultats sur les découpages des problèmes de tailles **41472**, **107811** et **397953**. D'une part, ces résultats portent sur le nombre total **sepsz** d'inconnues internes aux séparateurs, le pourcentage **sepsz/neqs**, la taille **mid** (total des inconnues internes et interfaces) des sous-structures les plus représentées, la taille moyenne **avrg** de toutes les sous-structures, et le rapport **ratio** entre **mid** et **avrg**. D'autre part, ils nous informent sur les temps nécessaires pour réaliser la phase numérique : le temps **tloc** de factorisation des matrices locales associées aux sous-structures et celui **tschur** de construction et de factorisation du complément de Schur.

ndoms	sepsz	sepsz/neqs	mid	avrg	ratio	tloc	tschur
64	13722	0.33	1029	922	1.11	3.100	7.130
128	18672	0.45	588	527	1.12	1.460	8.030
256	23082	0.56	135	303	1.11	0.830	8.760

TAB. 1.2 – *Découpage du problème de taille* 41472.

ndoms	sepsz	sepsz/neqs	mid	avrg	ratio	tloc	tschur
64	26811	0.25	2430	2187	1.11	26.25	42.84
128	37146	0.34	1350	1212	1.11	11.41	47.32
256	46491	0.43	720	673	1.07	5.170	50.51
512	55083	0.51	450	375	1.20	2.720	52.37
1024	69723	0.65	240	223	1.08	1.800	55.52

TAB. 1.3 – *Découpage du problème de taille* 107811.

ndoms	sepsz	sepsz/neqs	mid	avrg	ratio	tloc	tschur
64	68508	0.17	7956	7421	1.07	415.5	471.7
128	94077	0.24	4290	3964	1.08	185.9	516.6
256	119346	0.30	2310	2129	1.08	85.02	546.9
512	144366	0.36	1272	1148	1.11	41.45	570.7
1024	185997	0.47	735	647	1.14	20.90	600.1

TAB. 1.4 – *Découpage du problème de taille* 397953*.*

Comme nous pouvons le voir sur ces tableaux, nous gagnons un facteur supérieur à deux sur le temps **tloc** quand nous continuons de doubler le nombre de sous-domaines jusqu'à une certaine valeur limite. Par exemple, pour le problème ayant **107811** équations, nous atteignons ce seuil lorsque **ndoms** est égal à **256**. Nous remarquons que cette limite coïncide aussi à une dimension **sepsz** du complément de Schur. Celle-ci représente environ **40**% du nombre total d'inconnues. Au-delà de cette dimension, le gain diminue et reste inférieur à deux. En cumulant les temps **tloc** et **tschur**, les performances que nous obtenons sont plus intéressantes à ce seuil. Nous pouvons aussi s'apercevoir que les découpages respectant ce taux sont assez réguliers puisque le ratio entre la taille **mid** des sous-structures les plus représentées et la taille **avrg** vaut environ **1.11**.

A présent, nous allons analyser l'influence du nombre de sous-structures engendrées par la bissection récursive sur les temps d'exécution des problèmes considérés. Pour chacun de ces problèmes, nous donnons le nombre **nonz** de termes non nuls dans les matrices associées. Sur le tableau 1.5, nous exposons les résultats obtenus en fonction du nombre de sous-structures **ndoms** variant de **64** à **1024**.

Problèmes		ndoms				
neqs	**nonz**	**64**	**128**	**256**	**512**	**1024**
41472	**1564236**	11.43	10.83	11.15	$***$	$***$
107811	**4160934**	72.44	62.39	59.80	59.86	63.01
397953	**15692256**	903.1	719.0	649.8	631.9	642.2

TAB. 1.5 – *Temps d'exécution (s) en fonction du nombre de sous-structures.*

En analysant les résultats présentés sur le tableau 1.5, nous constatons que le nombre optimal de sous-structures dans le découpage menant au temps d'exécution le plus petit possible est étroitement lié à **neqs**, taille du problème à résoudre. En effet, pour les problèmes comportant **41472**, **107811** et **397953** équations, ce nombre varie et vaut respectivement **128**, **256** et **512**. Dans chacun de ces trois découpages, la dimension du complément de Schur valait environ **40**% par rapport à la taille du problème global, c'est-à-dire que **sepsz** est équivalent à $\frac{2}{5} * \mathbf{neqs}$. Ce qui est donc en concordance avec les remarques que nous avons faites précédemment sur les découpages.

Pour conclure cette partie, nous pouvons dire que la recherche du nombre optimal de sous-structures menant au meilleur temps d'exécution d'un problème donné est un exercice très complexe. Mais nous avons tout de même mis en place une stratégie expérimentale permettant d'estimer ce nombre pour la majeure partie des problèmes que nous sommes

capables de traiter. Celle-ci consiste à chercher **ndoms** sous la forme $\mathbf{2^k}$ en déterminant **k** avec la formule (1.20). Dans cette formule, ***Limit*** est une estimation de la taille moyenne en inconnues internes des sous-structures et *MAX_P_LEVEL* désigne la valeur maximale que peut prendre **k**. Nous avons fixé ***Limit*** à **120** inconnues donnant une bonne approximation du nombre d'inconnues internes dans les sous-domaines et *MAX_P_LEVEL* à **8** ou **9** pour éviter d'avoir des super-arbres très larges qui pourraient ralentir les peformances du solveur.

$$k = min(log_2(\frac{3}{5} * \frac{neqs}{Limit}),\ MAX\_P\_LEVEL) \tag{1.20}$$

Dans le futur, l'idéal serait de trouver une relation pratique rendant possible l'évaluation du nombre optimal de sous-structures en fonction de **neqs** et de la dimension **sepsz** du complément de Schur.

1.6.3 Comparaison des temps d'exécution

Nous comparons en séquentiel les temps d'exécution CPU du solveur Dissection qui a été mis en place avec ceux d'autres solveurs directs existants dans ZéBuLon. Les résultats sont présentés sur les figures 1.16 et 1.17. Les solveurs DSCPack et MUMPS sont basés respectivement sur une approche similaire à celle de la dissection emboîtée et sur une approche multi-frontale mais un de leurs points faibles est que DSCPack ne prend pas en compte les systèmes singuliers et que MUMPS n'est pas très robuste sur l'inversion des systèmes linéaires provenant des structures mécaniques flottantes et très hétérogénes. Par contre, Sparse Direct et Frontal sont deux autres solveurs directs qui eux tiennent bien compte des singularités de ces systèmes.

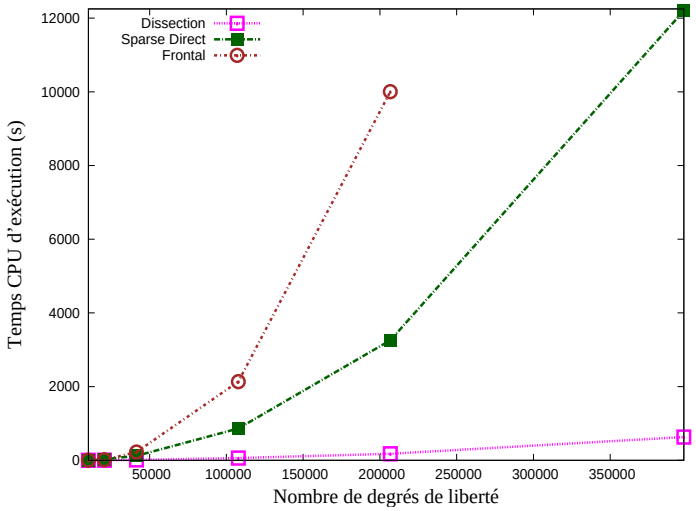


FIG. 1.16 – *Première phase de comparaison des temps CPU d'exécution.*

Sur les résultats présentés à la figure 1.16, nous pouvons observer que les méthodes de résolution les plus anciennes (Sparse Direct et surtout Frontal) s'écroulent pour des tailles de systèmes linéaires dépassant quelques dizaines de milliers de degrés de liberté alors que les résultats

obtenus avec le solveur "Dissection" que nous avons mis en œuvre sont très intéressants.

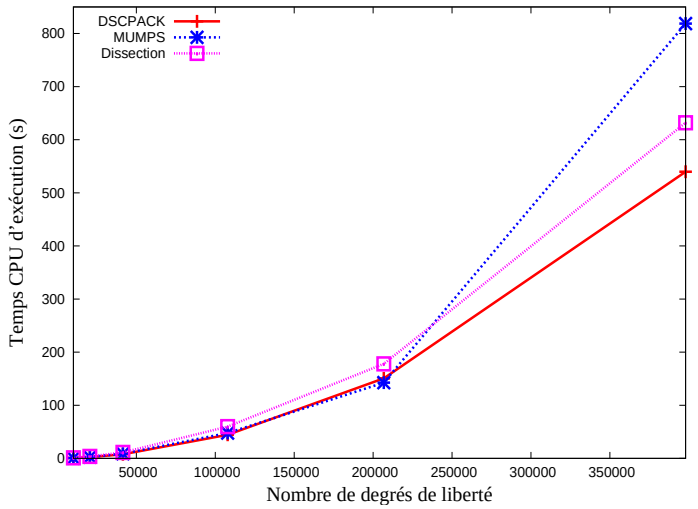


FIG. 1.17 – *Deuxième phase comparaison des temps CPU d'exécution.*

Les résultats présentés sur la figure 1.17 montrent que les performances de DSCPack et MUMPS sur des problèmes de tailles moyennes sont un peu meilleures que celles de notre solveur "**Dissection**". Par exemple, le ratio obtenu entre notre méthode et DSCPack vaut respectivement **1.42**, **1.54** et **1.46** pour les trois petits calculs (**10125**, **20577** et **41472** degrés de liberté ou ddl) et **1.37** et **1.18** pour les deux calculs moyens (**107811** et **206763** inconnues). Pour le calcul le plus gros (**397953** inconnues), la performance de DSCPack reste meilleure (**1.20** plus rapide). Par contre, celle de MUMPS devient moins intéressante que celle obtenue avec notre méthode (ratio de **0.79**). Toutes ces observations prouvent que notre méthode se comporte très honorablement par rapport aux méthodes de référence. Par ailleurs, elle offre sur les solveurs DSCPack et MUMPS l'avantage qu'elle est capable de prévenir les mouvements de corps rigide dans les sous-structures flottantes. Ce point sera mis à profit dans l'utilisation des approches de décomposition de domaine de type FETI.

1.6.4 Comparaison des temps de descente-remontée

Le solveur Dissection peut être utilisé pour résoudre des systèmes linéaires à seconds membres multiples. Sur la figure 1.18 suivante, nous montrons les performances de la phase de descente-remontée lors de la résolution d'un problème contenant **206763** degrés de liberté. Nous faisons varier le nombre de seconds membres **nrhs** de **1**, **50**, **100**, **150** et **200**. L'analyse des résultats présentés sur cette figure nous permet de voir que nous avons de bonnes performances avec le nouveau solveur "Dissection" dans la phase de descente-remontée. Cela pourrait être d'un grand intérêt pour l'amélioration de la résolution du problème d'interface de FETI, où plusieurs descentes-remontées sont effectuées successivement pour satisfaire la continuité aux interfaces entre sous-domaines.

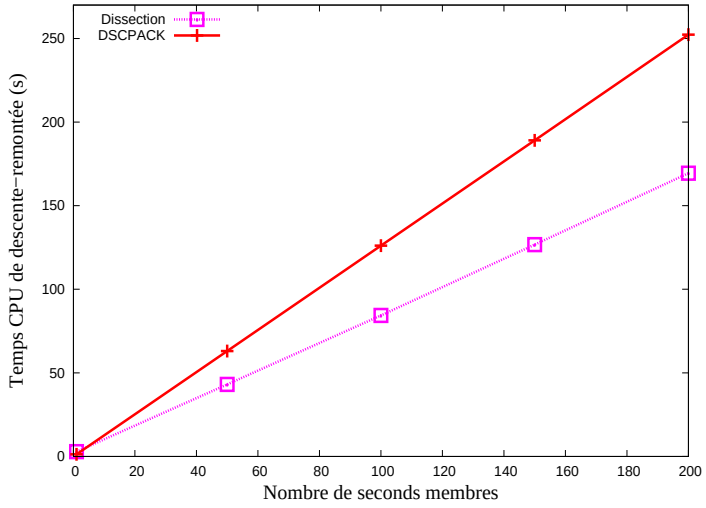


FIG. 1.18 – *Comparaison des temps CPU de descente-remontée.*

1.6.5 Analyse de la complexité du solveur

Pour analyser la complexité du solveur direct, nous comptons le nombre d'opérations en virgule flottante requises pour réaliser la phase numérique lors de l'inversion d'un système linéaire donné. Dans les méthodes directes, c'est dans cette phase que nous effectuons la majeure partie des instructions élémentaires. Pour ce faire, nous traitons des problèmes de différentes tailles (**107811**, **206763** et **397953** degrés de liberté). Sur le tableau 1.6, nous regroupons les résultats obtenus pour chacun de ces problèmes étudiés.

Problèmes		**Complexité**		
neqs	**nonz**	**nops**(x 10^9)	**nops/s** (GFLOPS)	**Temps (s)**
107811	**4160934**	89.86	1.61	55.97
206763	**8075406**	186.5	1.10	169.9
397953	**15692256**	380.2	0.62	611.9

TAB. 1.6 – *Complexité du solveur dans la phase numérique.*

En analysant ces résultats, nous remarquons que la vitesse du solveur en GigaFLOPS (GFLOPS) diminue en fonction de la taille des problèmes résolus. Ce qui veut dire que la variation du temps de factorisation est plus importante que celle du nombre d'opérations effectuées. Par exemple, en passant du premier problème (**107811**) au troisième (**397953**), les temps de factorisation sont presque multipliés par **11** alors que le nombre d'opérations n'est multiplié que par **4**. L'explication est que le pourcentage d'inconnues aux séparateurs est plus grand dans le premier (**43**%) que dans le deuxième (**36**%). La part de calcul des contributions apportées au complément de Schur devient alors plus coûteuse en opérations dans le premier problème que dans le deuxième. Tout de même, ces résultats démontrent que la vitesse du solveur en nombre d'opérations flottantes par seconde (FLOPS) est très satisfaisante par rapport à la fréquence de la machine (**3.16** GHz).

Conclusion

Nous avons mis en œuvre un solveur direct qui permet de résoudre de grands systèmes linéaires creux symétriques et réels, pouvant être à second membre simple ou multiple. Ce solveur a été évalué et comparé en séquentiel avec d'autres solveurs directs existants dans le code de calcul ZéBuLon et, les performances obtenues sont très encourageantes. Dans ce solveur direct, nous avons aussi pris en compte les systèmes singuliers en calculant leurs modes à énergie nulle. Ces modes jouent un rôle principal dans beaucoup de simulations numériques. La mise en œuvre du solveur est basée sur une technique de dissection emboîtée pouvant conduire à un parallélisme de haut niveau grâce à sa stratégie de "diviser pour mieux régner". Il convient de noter que les tailles des super-nœuds générés par cette technique ont joué un important rôle pour la détermination des performances du solveur. Pendant la factorisation numérique, nous avons remarqué que la taille optimale des blocs matriciels associés à ces super-nœuds dépendait du nombre d'inconnues du problème ce qui rend donc difficile le calcul de celle-ci. Dans le futur, il serait judicieux de trouver une relation permettant, en fonction de la taille du problème, de faciliter cette recherche. La mise en œuvre complète du solveur comprenant une prise en compte les systèmes linéaires symétriques complexes, non symétriques réels ou complexes sera aussi envisagée.

Parallélisation du solveur

2

Sommaire

2.1 L'état de l'art . . . 37
2.1.1 Solveurs destinés aux architectures à mémoire distribuée . 37
2.1.2 Solveurs adaptés aux architectures à mémoire partagée . . 38
2.1.3 Solveurs directs basés sur une programmation hybride . . 38
2.1.4 Discussions . . . 39
2.2 La programmation multi-cœurs . . . 39
2.2.1 Motivation . . . 40
2.2.2 Architectures multi-cœurs . . . 40
2.2.3 Parallélisme à mémoire partagée : le multi-threading . . . 41
2.3 Les démarches pour la parallélisation du solveur . . . 45
2.3.1 Version multi-threads avec Pthreads . . . 47
2.3.2 Version multi-threads avec OpenMP . . . 54
2.3.3 Évaluation des deux versions multi-threads . . . 54
2.3.4 Mise en place d'une version multi-threads hybride . . . 61
2.4 Les performances en multi-threading . . . 61
2.4.1 Analyse des performances . . . 62
2.4.2 Comparaison avec DSCPack et MUMPS . . . 63
Conclusion . . . 64

Ce chapitre est consacré à la parallélisation du solveur direct creux mis en place. Nous dressons d'abord l'état de l'art sur les bibliothèques permettant actuellement de résoudre par méthodes directes de grands systèmes linéaires creux sur des calculateurs massivement parallèles. Nous exposons ensuite la motivation et l'intérêt de la programmation multi-cœurs que nous voulons choisir pour accroître les performances du solveur à travers un modèle de parallélisme à mémoire partagée. Nous discutons de l'avènement des architectures multi-cœurs et de la méthodologie du multi-threading (multi-programmation légère). Enfin, nous détaillons les différentes démarches que nous avons mises au point pour créer des threads dans le solveur direct. Nous terminons ce chapitre par la mesure des performances de la version multi-threads du solveur sur une machine multi-cœurs.

2.1 L'état de l'art

Actuellement, nous trouvons de nombreuses bibliothèques académiques ou industrielles dédiées à la résolution de grands systèmes linéaires creux sur des calculateurs massivement parallèles. Certaines de ces bibliothèques sont spécialisées dans les méthodes directes. Les efforts se concentrent en gros sur la parallélisation de la factorisation de la matrice. En général, c'est l'étape la plus consommatrice en temps de calcul. Parmi les outils offrant des méthodes directes parallèles nous pouvons en citer quelques uns que nous classons en trois catégories : les bibliothèques portables sur les machines à mémoire distribuée, celles conçues pour les architectures parallèles à mémoire partagée et celles destinées à la fois aux deux types d'architectures précédemment citées. Toutes ces bibliothèques reposent pour la plupart sur l'approche multifrontale et l'approche supernodale qui découle directement des algorithmes d'élimination colonne par colonne réécrits par blocs. Pour les différences entre ces approches et leurs incidences sur les performances, nous renvoyons à l'article écrit par Heath et al. (1991) et à la thèse de Rothberg (1992).

2.1.1 Solveurs destinés aux architectures à mémoire distribuée

L'évolution des architectures des calculateurs scientifiques dans les dernières années a consacré l'usage systématique du parallélisme multiprocesseurs à mémoire distribuée pour les grands systèmes linéaires. Mémoire distribuée veut dire que chaque processeur (ou ensemble de processeurs) dispose d'une mémoire locale et ne peut accéder directement qu'à celle-ci. Les données résidant dans les différentes mémoires locales ne peuvent s'échanger que par des transferts réalisés par un réseau d'interconnexion (figure 2.1).

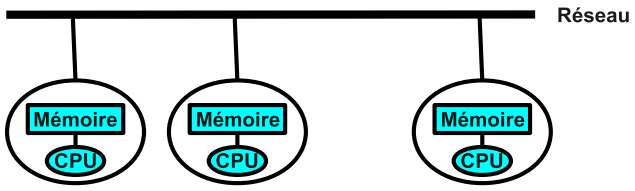


Fig. 2.1 – *Architecture à mémoire distribuée.*

Le parallélisme à mémoire distribuée offre une solution de conception pour certains solveurs directs creux parmi lesquels nous pouvons citer les codes PSPASES (Gupta et al. 1994) et DSCPack (Raghavan 2001) qui sont spécialisés dans les systèmes symétriques définis positifs (SDP), et SuperLU_Dist (Li et Demmel 2003) pour des systèmes non symétriques. Présentement, la seule bibliothèque parallèle adaptée à ces deux types de systèmes est MUMPS (Amestoy et al. 2000). PSPASES est l'un des premiers solveurs parallèles développé pour les systèmes SDP qui a réussi à résoudre en un temps réduit un problème comportant un million d'équations sur 256 processeurs d'un CRAY T3E. L'algorithme parallèle dans SuperLU_DIST est basé sur une partition en super-nœuds renfermant un parallélisme de haut niveau dans la mise à jour du complément de Schur. Des algorithmes d'élimination sur des graphes orientés acycliques sont

utilisés pour identifier la dépendance entre les super-nœuds. Le modèle de programmation parallèle utilisé dans ces solveurs est basé sur les librairies d'échanges de messages (MPI), ce qui les rend donc portables sur les calculateurs parallèles à mémoire distribuée.

2.1.2 Solveurs adaptés aux architectures à mémoire partagée

Les architectures à mémoire partagée sont composées de nœuds où chacun est un ensemble de processeurs qui se partagent la même mémoire physique (clusters ou machines multi-cœurs). Mais comme il est techniquement très difficile de réaliser une mémoire de grande taille capable d'alimenter en données plusieurs processeurs avec un débit suffisant, ces architectures possèdent différents niveaux de mémoires tampons intermédiaires appelés caches (figure 2.2).

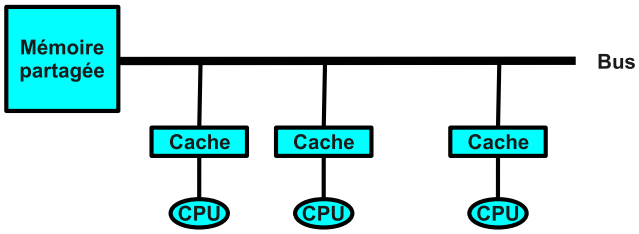


FIG. 2.2 – *Architecture à mémoire partagée.*

Certains des solveurs directs creux portables sur ces architectures sont aussi basés sur l'environnement MPI. C'est le cas, par exemple, du solveur direct SuperLU_MT (Demmel et al. 1999) qui est spécialisé sur les systèmes linéaires creux non symétriques. L'algorithme parallèle dans SuperLU_MT est significativement différent de celui dans SuperLU_Dist. La première version de SuperLU_MT a été développée avec les threads POSIX et destinée aux machines parallèles à mémoire distribuée de taille modeste (32 processeurs par exemple). Dans sa version la plus récente, des threads OpenMP ont été ajoutés. Lors de la factorisation LU, on utilise un algorithme parallèle classique avec un pivotage partiel. Cette algorithme, appelé *left-looking* en anglais, effectue une mise à jour d'une colonne de la matrice que lorsque le pivot de celle-ci doit être éliminé. Pour la parallélisation, un algorithme dynamique asynchrone est utilisé pour ordonnancer les tâches à gros grains et à grains fins. La tâche à gros grains consiste à factoriser les blocs indépendants dans les sous-arbres disjoints de l'arbre d'élimination, tandis que la tâche à grains fins est une mise à jour du complément de Schur par les super-nœuds précédemment calculés. L'ordonnancement facilite la transition entre les deux types de tâches et maintient dynamiquement l'équilibre des charges.

2.1.3 Solveurs directs basés sur une programmation hybride

Certains solveurs directs creux sont basés sur une approche révélante qui consiste à utiliser un modèle de programmation hybride MPI-Pthreads. Cette approche leur permet de tirer pleinement profit des machines à mémoire partagée. Parmi les bibliothèques basées sur cette approche, nous pouvons citer SPOOLES (Ashcraft et Grimes 1999) et le solveur

PaStiX (Hénon et al. 2002). Le code SPOOLES fournit un solveur direct creux capable d'inverser des systèmes symétriques définis positifs ou non-symétriques. Les parties multi-threads de ce code sont seulement la factorisation numérique et la phase de descente-remontée qui peuvent avoir de multiples threads POSIX partageant un même espace d'adressage. La version sous l'environnement MPI est plus complexe et concerne toutes les phases de résolution. Dans le solveur PaStiX, la mise en œuvre hybride est motivée par le fait que les communications au sein d'un nœud de calcul à mémoire partagée peuvent être avantageusement remplacées par des accès directs à la mémoire partagée entre les processeurs de celui-ci en utilisant des threads. Ce parallélisme s'effectue à l'intérieur des machines à mémoire partagée et il est automatiquement exploité par une version multi-threads de BLAS. Des algorithmes d'ordonnancement statique sont utilisés pour décider exactement comment la factorisation d'un super-nœud doit être découpée en des tâches de BLAS élémentaires distribuées entre les threads créés. Pour mettre en œuvre ces ordonnanceurs, de nombreux systèmes permettent de fixer les threads d'un processus MPI sur des ensembles de processeurs ainsi que de préciser sur quelle machine les allocations de zones de données doivent être effectuées.

2.1.4 Discussions

Dans toutes les bibliothèques citées, il n'y en a pas une seule qui soit mieux adaptée que les autres pour tous les types de systèmes linéaires. Certains solveurs directs sont ciblés pour des matrices symétriques définies positives, d'autres pour des cas plus généraux. Pour le même système linéaire, le code pourrait même décider d'utiliser un algorithme particulier ou la technique de mise en œuvre qui est la plus performante. Tous ces solveurs ont été installés sur une variété de plates-formes et testés intensivement sur de nombreuses applications issues du monde industriel. Mais à l'exception du solveur MUMPS, aucun d'entre eux ne prend en compte les systèmes non inversibles. D'ailleurs, ce dernier n'est pas très robuste pour les systèmes associés aux structures très hétérogénes. A notre connaissance aussi, aucun solveur direct n'a été validé comme solveur local parallèle au sein des méthodes de décomposition de domaine de type FETI à cause de l'absence de l'étape de détection des modes à énergie nulle des systèmes singuliers. En revanche, le solveur direct que nous avons mis en œuvre au chapitre 1 et que nous voulons paralléliser ici peut être une solution à cela. Nous effectuons sa parallélisation avec un modèle de programmation parallèle à mémoire partagée reposant sur la méthodologie du multi-threading (Akhter et Roberts 2006).

2.2 La programmation multi-cœurs

Durant des années, les concepteurs n'ont cessé d'améliorer les performances des microprocesseurs en augmentant leur vitesse d'exécution. Toutefois, la dernière évolution de la technologie des processeurs est celle des multi-cœurs. Les fabricants (Intel et AMD) investissent énormément dans le traitement multi-cœurs. Ils intègrent plusieurs microprocesseurs sur un seul circuit intégré. En 2006, Intel avait présenté le prototype d'un

processeur doté de 80 cœurs (figure 2.3) qu'il comptait commercialiser d'ici 2011. Ce processeur sera capable d'exécuter plus de mille milliards d'opérations à la seconde, soit une puissance de calcul supérieure à un TéraFLOPS (TFLOPS) (Intel 2006).

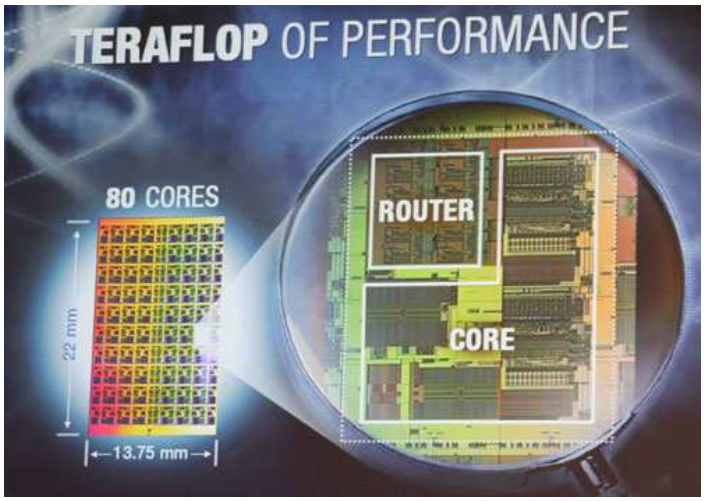
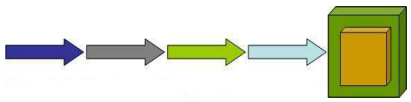


FIG. 2.3 – *Processeur d'Intel à 80 cœurs.*

2.2.1 Motivation

L'émergence des processeurs multi-cœurs et l'accroissement prévu du nombre de cœurs dans les futurs processeurs, va permettre un véritable fonctionnement multi-tâches (figure 2.4(b)). Sur des systèmes mono-cœur, le fonctionnement multi-tâches peut dépasser les capacités de la machine, entraînant une baisse des performances liée à la mise en attente des opérations à traiter (figure 2.4(a)).

(a) Processeur mono-cœur : tâches mises en attente



(b) Processeur bi-cœurs : tâches exécutées indépendamment

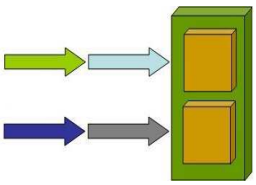


FIG. 2.4 – *Exemple de fonctionnement multi-tâches.*

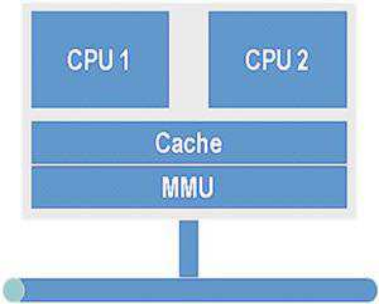
Notre motivation est de tirer donc pleinement profit de la puissance de calcul offerte par les prochaines architectures multi-cœurs en parallélisant le solveur direct mis en œuvre. Une telle réalisation est plus complexe que la mise en place de la version séquentielle destinée aux processeurs mono-cœur.

2.2.2 Architectures multi-cœurs

Les architectures multi-cœurs sont caractérisées par l'existence, dans un même processeur, de plusieurs cœurs d'exécution. Ces cœurs partagent hiérarchiquement la mémoire cache et l'unité de gestion mémoire (MMU)

avec des interconnexions courtes. Cela fournit suffisamment de ressources pour traiter en parallèle les tâches les plus exigeantes en calculs. La figure 2.5(a) donne une représentation schématique d'une architecture composée de deux cœurs et d'une mémoire cache. Celle-ci contient des données utilisées aussi bien par le premier que le second cœur. Les architectures multi-cœurs répondent aussi à de nombreux besoins non seulement en terme de consommation mais également en terme de performance. Elles constituent ainsi une amélioration des architectures multi-processeurs qui nécessitent des interconnexions longues (figure 2.5(b)). Les trajets parcourus par les signaux sont plus courts, il y a donc une meilleure synchronisation des données.

(a) Architecture bi-cœurs.



(b) Système bi-processeurs : mémoire cache divisée.

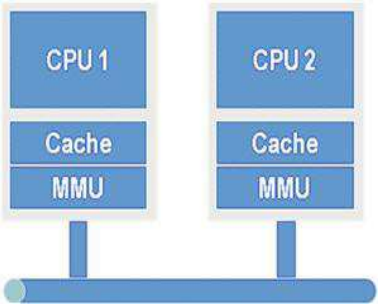


FIG. 2.5 – *Exemples d'architectures multi-cœurs et multi-processeurs.*

Les principaux inconvénients des architectures multi-cœurs se situent au niveau logiciel. Aujourd'hui, très peu d'applications sont développées dans l'esprit d'une parallélisation des instructions. Tout au plus, quelques processus légers sont lancés parallèlement à un processus plus important. L'effet produit est qu'un unique cœur est alors sollicité et le logiciel n'exploite donc pas toute la puissance de calcul des multi-cœurs.

Pour exploiter toute la puissance de calcul obtenue de ces nombreux cœurs, le développeur se doit dorénavant d'exhiber un parallélisme à grain très fin tout en portant une attention particulière à la répartition des traitements et des données auxquelles ceux-ci accèdent, afin de maximiser l'utilisation du cache et de minimiser les accès mémoire distants. Cette exploitation peut être bien menée grâce à la technique du multi-threading. Celle-ci est un modèle de parallélisme à mémoire partagée capable d'effectuer plusieurs tâches dans une seule application.

2.2.3 Parallélisme à mémoire partagée : le multi-threading

Le modèle de parallélisme à mémoire partagée reposant sur le multi-threading est très lié à la notion de thread ou processus léger. Un thread est une portion de code (fonction) qui se déroule en parallèle au processus principal (aussi appelé *main*). Les threads sont définis pour se lancer en même temps que le processus principal, ce qui permet de faire de la programmation multi-tâches. Le but est donc de permettre au programme de réaliser plusieurs tâches au même moment. C'est une bonne façon de mieux exploiter le parallélisme sur des architectures multi-cœurs. Nous allons commencer par présenter le concept général de cette technique de

parallélisme. Nous verrons ensuite les différents outils choisis pour mettre en œuvre cette technique dans le solveur direct.

Le concept général

La notion de parallélisme à mémoire partagée, fréquemment appelé multi-threading, est rendue nécessaire par l'utilisation qui est faite aujourd'hui des processeurs multi-cœurs. Dans cette approche, chaque thread a un accès partagé à la mémoire globale du processeur et les échanges de données entre les différents threads se font simplement par des lectures/écritures vers la même zone mémoire cache. Il faut s'assurer qu'aucune tâche ne peut lire une donnée qui n'est que partiellement mise à jour du fait que les opérations peuvent utiliser plusieurs cycles d'horloge. Si une partie de la cohérence des données peut être assurée par le matériel, la totalité de la consistance des données reste à la charge du développeur. La parallélisation des applications n'étant pas toujours une chose aisée, des outils et des compilateurs spécialisés ont vu le jour pour faciliter le multi-threading. Dans ce cas, le parallélisme est géré directement par ces outils. Le compilateur génère alors un code parallèle, utilisant le multi-threading en suivant les indications données par le programmeur.

Des outils pour le multi-threading

La parallélisation à mémoire partagée des applications repose très souvent sur l'utilisation des bibliothèques de multi-threading. Ces bibliothèques permettent à plusieurs tâches de partager le même espace d'adressage. Il existe maintenant des bibliothèques de multi-threading bien optimisées pour les architectures multi-cœurs. Les deux standards que nous allons exposer sont les threads POSIX (Lewis et Berg 1998) et OpenMP (Chandra et al. 2000). Ceux sont des outils simples et souvent efficaces pour mettre en œuvre le multi-threading dans des applications.

1. **Le standard POSIX**
   Ce standard définit le comportement et l'interface de bibliothèques de threads. Les bibliothèques qui suivent ce standard sont aussi appelées Pthreads. La norme POSIX donne une définition détaillée des threads et des proprietés qu'une implantation doit vérifier. Elle fournit aussi des primitives pour créer des processus légers et les synchroniser. La bibliothèque Pthreads est portable. Elle existe sur les systèmes d'exploitation tels que Linux et Windows. C'est pour cela que nous avons choisi d'utiliser en premier cette bibliothèque que nous allons décrire par la suite.

   (a) **La création et la fin des threads :**
   Un thread est créé avec l'appel de la fonction :

   $$\textbf{\textit{pthread\_create}}(th\_id, th\_attr, routine, param).$$

   Le premier paramètre *th_id* est un résultat, il s'agit de l'identifiant du thread créé. Le second paramètre *th_attr* détermine les attributs spécifiques du thread comme le type d'ordonnancement, la priorité et la taille de la pile. Le troisième *routine* est la procédure à exécuter. Celle-ci est en même temps la raison

de vivre du thread. Cet argument permet de transmettre un pointeur sur la fonction que le thread devra exécuter. Le dernier paramètre *param* représente la valeur du paramètre effectif à passer à la procédure que doit exécuter le thread. Chaque thread peut donc recevoir ce dernier paramètre sous la forme d'un pointeur. Pour passer plusieurs paramètres à un thread, il suffit de les regrouper dans une structure de données et de passer l'adresse de celle-ci.

Chaque thread créé est lancé immédiatement et exécute la procédure passée en troisième argument. L'exécution du thread se fait, soit jusqu'à la fin de la procédure associée, soit jusqu'à l'appel de la fonction ***pthread_exit()***. Cette fonction force la fin d'exécution d'un thread mais pas la fin du processus principal, ce qui permet ainsi aux autres threads créés de s'exécuter.

Lorsque nous créons des threads et puis que nous laissons continuer par exemple la fonction *main*, nous prenons le risque de terminer le programme complètement sans avoir pu exécuter tous les threads. Nous devons donc attendre que les différents threads créés se terminent. Pour cela, il existe la fonction :

$$\textbf{\textit{pthread\_join}}(th\_id, th\_return).$$

Le premier argument *th_id* est l'identifiant du thread à attendre et le deuxième *th_return* est la valeur de retour de la fonction. L'appel de cette fonction met en pause l'exécution du thread appelant jusqu'au retour de la fonction.

(b) **La synchronisation :**
Elle consiste à coordonner l'accès en lecture et en écriture sur une ou plusieurs données partagées entre les différents threads d'une application. Pour cela, plusieurs techniques peuvent être employées.

– Les **mutex** ou zones d'exclusion mutuelle sont un système de verrou donnant ainsi une garantie sur la viabilité des données manipulées par les threads. En effet, il arrive même très souvent que plusieurs threads accédent en lecture et/ou en écriture aux mêmes variables. Si un thread possède le verrou, lui seul peut lire et écrire sur les variables présentes dans la portion de code protégée (appelée aussi section critique). Lorsque le thread a terminé, il libère le verrou et un autre thread peut le prendre à son tour.
Pour créer un mutex, il faut tout simplement déclarer une variable du type ***pthread_mutex_t*** (verrou) et l'initialiser avec la constante ***PTHREAD_MUTEX_INITIALIZER*** soit par exemple : th_mutex. Pour le détruire et ainsi libérer les ressources qu'il détient, il faut faire un appel de la fonction $\textbf{\textit{pthread\_mutex\_destroy}}(th\_mutex)$.
Les mutex n'ont que deux états possibles, ils sont soit verrouillés, soit déverrouillés. Les deux fonctions ci-dessous sont utilisées pour changer les états. L'appel de la fonction $\textbf{\textit{pthread\_mutex\_lock}}(th\_mutex)$ permet de

déterminer le début d'une zone critique. La fonction ***pthread_mutex _unlock***(th_mutex) est utilisée pour relâcher le verrou *th_mutex* passé en argument.

– Les **sémaphores** protègent les données. Ce sont les primitives les plus couramment utilisées pour restreindre l'accès à des variables partagées dans un environnement de programmation concurrente. Les sémaphores sont créés par l'appel de ***sem_init***$(sem, pshared, value)$, qui place l'identificateur du sémaphore à l'endroit pointé par *sem*. La valeur initiale du sémaphore est dans *value*. Si l'argument *pshared* est nul, le sémaphore est local au processus principal.
Les méthodes ***sem_wait***(sem) et ***sem_post***(sem) sont respectivement utilisées pour l'attente d'une variable disponible et le rendu de la variable disponible à ceux qui l'attendent.

Nous pouvons donc conclure que les sémaphores POSIX sont mis en œuvre de façon assez similaire aux mutex, la principale différence réside dans la fait qu'un thread peut imbriquer plusieurs prises de sémaphores.

2. **Le standard OpenMP**
OpenMP est une interface standard de haut niveau pour une programmation parallèle sur machine à mémoire partagée permettant une gestion de la distribution des calculs entre plusieurs processeurs multi-cœurs. Basée sur les techniques du multi-threading, OpenMP peut être considérée comme l'un des grands standards au service du calcul scientifique. Cette interface de programmation est supportée sur la plupart des plate-formes, incluant Unix et Windows, pour les langages de développement Fortran, C et C++. Grâce à des directives insérées par le développeur dans le code et grâce à une analyse des dépendances entre les données, un compilateur OpenMP est capable d'extraire du parallélisme des boucles de calcul. Les directives ne sont que des commentaires activables par le compilateur grâce une option adéquate. Ainsi, un compilateur ne supportant pas OpenMP les ignorera. C'est ce qui permet de garantir la portabilité des codes OpenMP et la maintenance d'une version unique, séquentielle et parallèle du code.

La parallélisation des portions de code indépendantes va être construite par le compilateur en utilisant des threads. Ce standard est donc une approche simplifiée pour paralléliser des codes via des threads. Certaines librairies mathématiques sont déjà optimisées avec OpenMP pour s'exécuter sur tous les cœurs d'un nœud de calcul. C'est le cas, par exemple, de la librairie MKL d'Intel qui regroupe un ensemble de sous-programmes scientifiques telles que BLAS, LAPACK. Des variables d'environnement permettent de contrôler l'environnement d'exécution comme par exemple le nombre de threads. Par défaut, le nombre de threads utilisés dans BLAS est égal au nombre de cœurs de la machine. Dans certains cas, pour être optimal, il faut ajuster le nombre de threads à l'exécution avec la variable d'environnement ***OMP_NUM_THREADS*** ou bien faire appel à la fonction ***omp_set_num_threads()***.

Une application parallélisée avec OpenMP compte au moins un thread qui est tout simplement le code dans son exécution séquentielle. Quand le programme arrive à un point où se situe une région parallèle, d'autres threads sont créés. Le premier thread, appelé maître, a la maîtrise sur l'équipe de threads dont il fait lui-même partie et dont il est en quelque sorte le capitaine. Il est bon de savoir que ce thread maître porte toujours l'identifiant **0**. Lorsque l'exécution de la section est terminée, tous les threads sont détruits, à l'exception du maître qui reprend une exécution séquentielle du programme comme illustré sur la figure 2.6. A ce moment, le résultat du travail des autres membres de l'équipe peut être collecté si besoin.

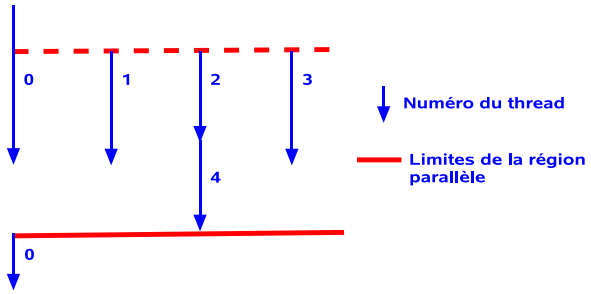


FIG. 2.6 – *Exemple de région parallèle : cinq threads sur quatre cœurs.*

Il est parfois nécessaire d'introduire une synchronisation entre les threads concurrents pour éviter, par exemple, que ceux-ci modifient dans un ordre quelconque la valeur d'une même variable partagée, c'est le cas par exemple des opérations de réduction. Pour ces opérations, il s'agit de calculer une somme, un maximum, etc, sur un ensemble de données d'un ou plusieurs tableaux.

Grâce aux deux outils de multi-threading que nous venons d'exposer, il est maintenant possible de définir des procédures permettant ainsi une programmation multi-cœurs efficace dans le solveur direct mis en place. Pour chacun de ces outils, la démarche effectuée pour la parallélisation du solveur peut être différente. Il faut aussi noter qu'il ne sert à rien d'avoir plus de threads que de cœurs disponibles sur un processeur, car le but d'un thread est d'alléger la tâche de ce dernier. Avoir plusieurs threads sur le même processeur n'allége donc rien sa charge. Nous allons maintenant détailler toutes ces démarches en tenant compte de ce dernier point.

2.3 Les démarches pour la parallélisation du solveur

Le solveur direct mis en œuvre au chapitre 1 utilise la méthode de dissection emboîtée. Cette méthode renferme un haut degré de parallélisme. En effet, le traitement d'un super-nœud, à un niveau donné du super-arbre généré, est totalement indépendant de celui des autres situés à ce niveau. Par exemple, le calcul des contributions locales $M_{\Gamma_1^1 I_1} {M_{I_1 I_1}}^{-1} M_{I_1 \Gamma_1^1}$ et $M_{\Gamma_1^1 I_2} {M_{I_2 I_2}}^{-1} M_{I_2 \Gamma_1^1}$ apportées par l'élimination des inconnues internes associées aux sous-structures I_1 et I_2 s'effectue indépendamment (voir la figure 2.7). Par conséquent, cette technique de renumérotation se prête bien à une parallélisation à mémoire partagée.

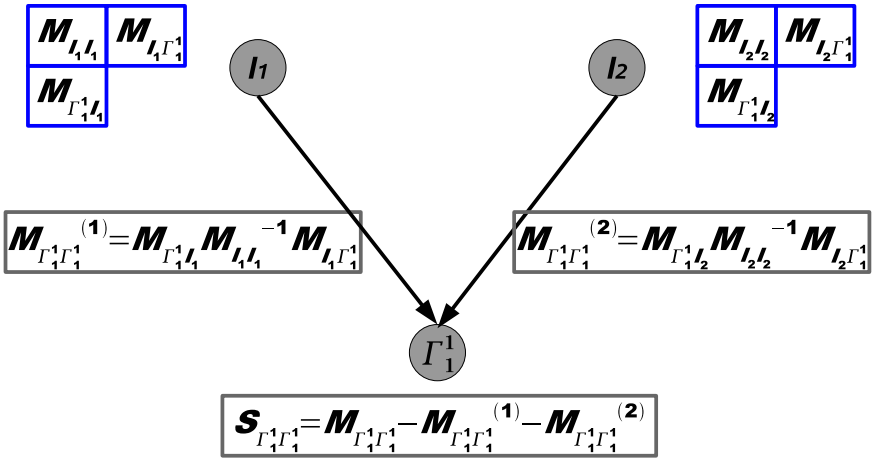


FIG. 2.7 – *Indépendance du calcul des contributions apportées à* Γ_1^1.

Notre approche générale pour la parallélisation du solveur direct consiste d'abord à considérer au maximum, à chacun des niveaux **l** du super-arbre, un nombre **ntask** de tâches de calcul ne dépassant pas **ncore** (le nombre de cœurs à notre disposition). Dans le cas où l'outil OpenMP est choisi, **ntask** reste identique à **ncore** puisque le traitement est automatiquement effectué à l'intérieur de chacun des super-nœuds. Dans le cas des Pthreads, chacune des tâches est alors composée de **nload** super-nœuds, où **nload** est le résultat de la division entière de **nbnode**(**l**) par **ntask**. La valeur **nbnode**(**l**) est le nombre de super-nœuds au niveau **l** pouvant être traités en parallèle. Mais, nous devons tenir compte dans le dernier cas du fait que le nombre **nbnode**(**l**) peut être quelconque ou diminuer de moitié au fur et à mesure que nous avançons dans l'arbre. Par ailleurs, les super-nœuds situés aux niveaux proches de celui de la racine ont souvent des tailles très importantes. En utilisant les threads POSIX, il est donc nécessaire de prendre en compte l'équilibre des charges dans la répartition des tâches. L'exemple de la figure 2.8 illustre la répartition des tâches sur une machine ayant quatre cœurs (**ncore** = **4**).

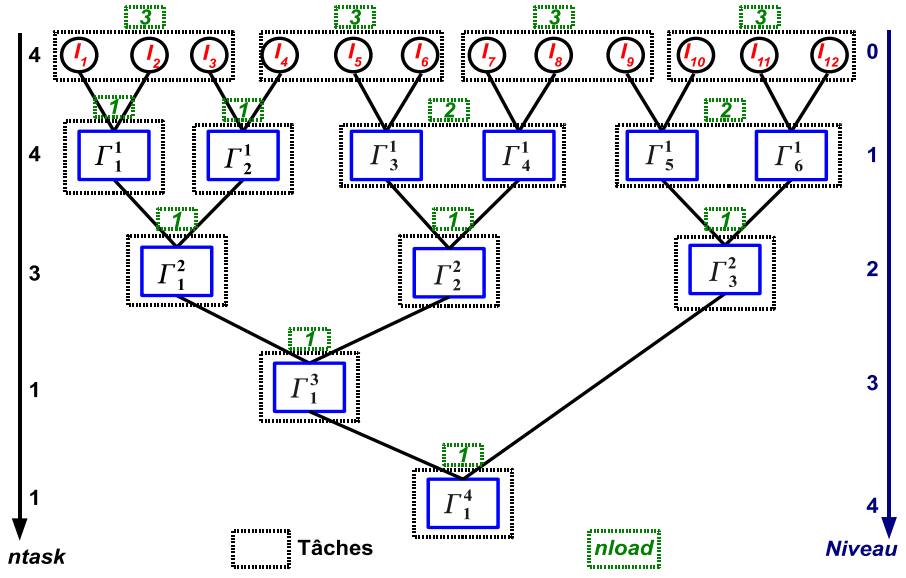


FIG. 2.8 – *Approche générale : répartition des tâches sur quatre cœurs.*

Nous poursuivons ensuite l'approche par la création de **ntask** threads. Nous définissons alors, sous la forme d'une structure de données, les paramètres de chacun des threads en fonction de **nload**. Ces paramètres seront passés à la procédure que allons construire par la suite et qui sera exécutée par tous les threads créés.

Nous allons d'abord commencer par mettre en place une version multi-threads avec Pthreads. Ce standard peut offrir une stratégie de parallélisation plus efficace aux niveaux **l** où le nombre **nbnode**(**l**) est plus grand que **ncore**. Nous utilisons ensuite la librairie MKL d'Intel (BLAS) optimisée avec OpenMP pour développer une version avec ce modèle de multi-threading. Cette bibliothèque peut permettre de mieux exploiter la parallélisation dans les super-nœuds dont les tailles sont plus importantes. Ces derniers sont localisés dans les plus hauts niveaux de l'arbre d'élimination. Une fois que ces deux versions sont mises en place et évaluées, notre objectif final est de tirer enfin profit du meilleur de ces deux dernières dans la phase numérique où les opérations sont pour la plupart effectuées par des sous-programmes de BLAS. Nous développons alors une version hybride en couplant les threads POSIX et le standard OpenMP dans la phase ciblée.

2.3.1 Version multi-threads avec Pthreads

La mise en place de cette version multi-threads consiste à créer des threads POSIX dans chacune des trois grandes phases qui ont permis au développement de la version séquentielle du solveur direct.

La création de threads lors de la factorisation symbolique

Il s'agit d'effectuer en parallèle la boucle qui porte sur la factorisation symbolique des blocs matriciels associés aux sous-structures I_i. Pour cela, nous découpons cette boucle en **ntask** tâches que nous répartissons de façon équilibrée sur les **ncore** cœurs disponibles. La charge **nload** de chacune des tâches est ainsi calculée en fonction du nombre de sous-structures **ndoms** égal à **nbnode**(**0**) et de **ncore** (voir l'algorithme 2.1).

Algorithme 2.1 : Répartition des tâches dans la phase d'analyse

Initialisation : $ntask = ncore$;
si $ntask > ndoms$ **alors**
| $ntask = ndoms$.
fin
calculer *rest*, le reste de la division de *ndoms* par *ntask*.
pour $ith = 1$ *à* $ntask - rest$ **faire**
| $nload = ndoms/ntask$.
fin
pour $ith = ntask - rest + 1$ *à* $ntask$ **faire**
| $nload = ndoms/ntask + 1$.
fin

Après avoir calculé la charge **nload** de chacun des threads **ith**, nous passons maintenant à la définition d'une structure de données **param** de

type ***SYM_FACT_PARAM*** à passer en paramètre à la procédure exécutée par les threads créés. Cette structure **param** regroupe les paramètres de début **begin** et de fin **end** de chacune des tâches.

```
struct SYM_FACT_PARAM {
                        int begin
                        int end
}
```

Pour chacun des threads **ith**, la structure **param** est définie de la manière suivante :

– si $1 \leq ith \leq ntask - rest$ alors $param$ est définie par :

$$\begin{cases} param[ith].begin = (ith-1) * nload \\ param[ith].end = param[ith].begin + nload \quad ; \\ \text{avec } nload = ndoms/ntask \end{cases}$$

– si $ntask - rest + 1 \leq ith \leq ntask$ alors $param[ith]$ est définie par :

$$\begin{cases} param[ith].begin = param[ith-1].end \\ param[ith].end = param[ith].begin + nload \quad . \\ \text{avec } nload = ndoms/ntask + 1 \end{cases}$$

Pour terminer cette parallélisation, nous construisons la procédure qui sera exécutée par chacun des threads **ith**. Dans cette procédure, nous effectuons la factorisation symbolique de **nload** blocs diagonaux $M_{I_iI_i}$, où $param[ith].begin \leq i \leq param[ith].end$.

Dans cette phase, les threads créés n'ont pas besoin d'être synchronisés puisqu'ils n'ont aucune donnée à échanger entre eux. En effet, la factorisation symbolique d'un bloc $M_{I_iI_i}$ est indépendante de celle des autres.

Il faut noter que nous ne pouvons pas profiter de tous les cœurs si leur nombre est supérieur à celui des sous-structures. Heureusement que ce cas n'arrive que très rarement dans la résolution de grands systèmes systèmes linéaires, où le nombre **ndom** est supérieur ou égal à **256**.

La création de threads dans la phase numérique

Dans un premier temps, nous commençons par créer des threads pour décomposer sous la forme $L_{I_iI_i}D_{I_iI_i}U_{I_iI_i}$ les matrices $M_{I_iI_i}$ et calculer les contributions $M_{\Gamma_j^l I_i}{M_{I_iI_i}}^{-1}M_{I_i\Gamma_k^m}$ apportées par l'élimination des inconnues internes des sous-structures I_i. L'approche est voisine de celle que nous avons utilisée pour mettre en place des threads dans la phase de factorisation symbolique. D'abord, nous appliquons l'algorithme 2.1 pour répartir les tâches sur les cœurs et calculer la charge **nload** de chacune d'elles. Nous regroupons ensuite les paramètres à passer à la procédure sous la forme d'une structure de données **param** de type ***NUM_FACT_PARAM***. Cette structure contient, d'une part les paramètres de début **begin** et de fin **end** de chacune des tâches et d'autre part, les variables **nsing**, **singVals** et **pivot** provenant de la détection locale des singularités dans les systèmes non inversibles. Les paramètres **begin** et **end** sont définis de la même façon que dans la phase de factorisation symbolique. La variable **nsing** est un entier qui représente le nombre de singularités locales détectées lors

de la décomposition $L_{I_iI_i}D_{I_iI_i}U_{I_iI_i}$ des **nload** blocs matriciels $M_{I_iI_i}$ effectuée par chacun des threads. La variable **singVals** est un tableau d'entiers, où sont stockées les **nsing** singularités. Le paramètre **pivot** est une variable globale représentant le plus petit pivot non nul rencontré lors de la factorisation des blocs diagonaux $M_{I_iI_i}$ ou $S_{\Gamma_j^l\Gamma_j^l}$.

```
struct NUM_FACT_PARAM {
                        int begin
                        int end
                        int nsing
                        int * singVals
                        double pivot
}
```

Dans les procédures que nous construisons ici, chacun des threads effectue d'abord une factorisation numérique de **nload** blocs matriciels $M_{I_iI_i}$ et calcule ensuite les contributions apportées par l'élimination des inconnues associées aux **nload** sous-structures I_i. Nous synchronisons les threads entre eux avec des mutex pour rafraîchir en mémoire la variable partagée $param[ith].pivot$. Pendant qu'un thread ith cherche parmi les pivots non nuls rencontrés en factorisant ses **nload** matrices celui qui est le plus petit et qu'il met à jour $param[ith].pivot$, les autres threads attendent.

Nous donnons deux exemples pour illustrer la répartition des tâches sur une machine ayant quatre cœurs (**ncore** $= 4$). Sur le premier (figure 2.9), le nombre de sous-structures I_i dans le super-arbre est une puissance de deux (**ndom** $= 2^3$) alors que celui dans le deuxième (figure 2.10) est quelconque (**ndom** $= 9$). En appliquant l'algorithme 2.1 sur chacun de ces deux exemples, nous obtenons le nombre de tâches à exécuter (**ntask** $= 4$). La charge des threads du premier exemple est égale à deux (**nload** $= 2$). Celle des threads du deuxième exemple varie entre deux et trois (**nload** $= 2$ ou 3). Nous effectuons alors sur chacun des quatre cœurs la factorisation numérique de **nload** blocs matriciels $M_{I_iI_i}$ et le calcul des contributions apportées par l'élimination des inconnues associées.

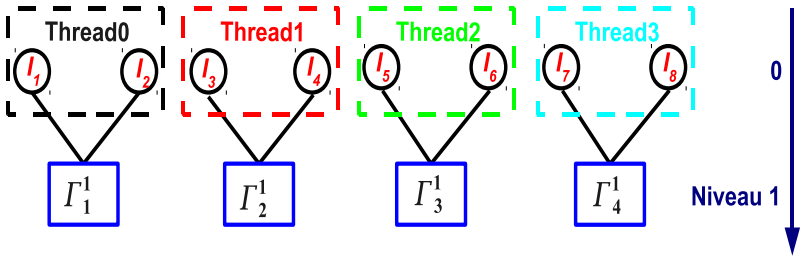


FIG. 2.9 – *Exemple de répartition de 8 sous-structures sur quatre cœurs.*

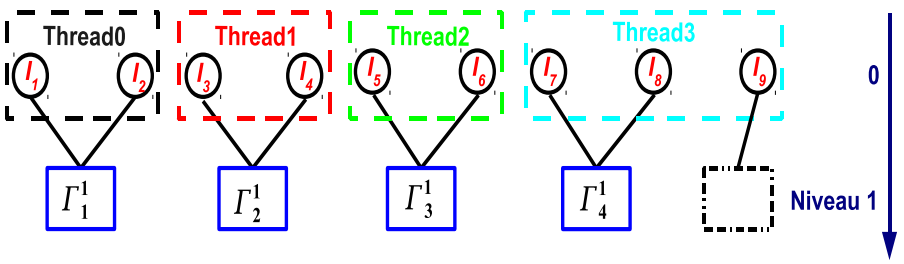


FIG. 2.10 – *Exemple de répartition de 9 sous-structures sur quatre cœurs.*

L'exemple sur la figure 2.11 illustre le calcul par le thread **0** (créé sur la figure 2.9) des contributions apportées par l'élimination des inconnues internes des sous-structures I_1 et I_2.

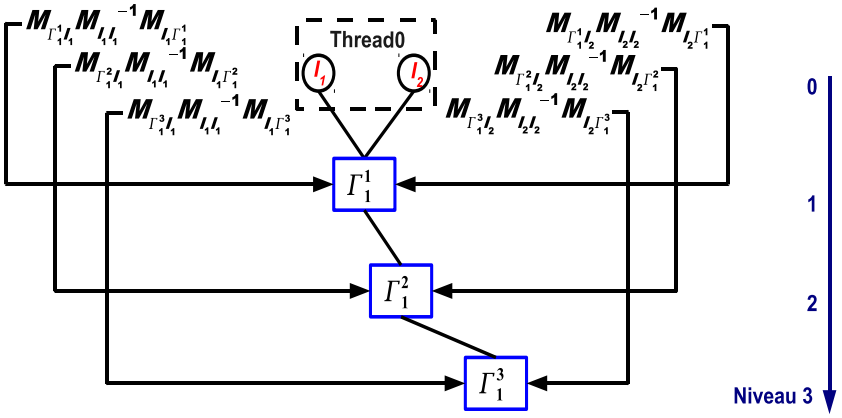


FIG. 2.11 – *Calcul par le thread* 0 *des contributions apportées par* I_1 *et* I_2.

Dans un deuxième temps, nous passons à la parallélisation de l'étape d'assemblage de tous les blocs $S_{\Gamma_j^l \Gamma_k^m}$ ($l > 0$ et $m > 0$) et celle de factorisation numérique du complément de Schur S formé par les matrices $S_{\Gamma_j^l \Gamma_j^l}$ associées aux séparateurs Γ_j^l (niveaux $l > 0$). Pour cela, nous créons un nombre **ntask** de threads à chacun des niveaux de ces deux étapes, où **ntask** est calculé en fonction du nombre de cœurs dont nous disposons et du nombre de super-nœuds à ce niveau. Le problème est que le nombre **ntask** de threads à créer dépend du niveau où nous nous retrouvons dans l'arbre. En effet, quand nous remontons l'arbre jusqu'à la racine, le nombre de super-nœuds par niveau décroît, ce qui fait décroître le nombre de tâches à effectuer. Et sachant que le nombre **ncore** de cœurs disponibles sur la machine est fixe, il arrive qu'aux plus hauts niveaux il n'y a plus assez de tâches à répartir sur les différents cœurs. Du coup, nous ne pouvons pas tirer profit de tous les **ncore** cœurs sur l'ensemble de la procédure.

Pour la création des threads dans les deux étapes, nous utilisons presque la même approche. La seule différence réside dans la construction des procédures que les threads exécuteront. A chacun des niveaux **l** de l'une ou l'autre des deux étapes, nous commençons d'abord par répartir les **ntask** tâches sur les cœurs grâce à l'algorithme 2.2.

Dans les procédures que nous construisons à l'étape d'assemblage, chacun des threads **ith** assemble, au niveau **l**, **nload** blocs diagonaux $S_{\Gamma_j^l \Gamma_j^l}$ et les blocs extra-diagonaux associés $S_{\Gamma_j^l \Gamma_k^m}$. Dans les procédures construites à l'étape de factorisation numérique du complément de Schur **S**, chacun des threads **ith** factorise au niveau **l**, **nload** blocs $S_{\Gamma_j^l \Gamma_j^l}$. L'indice j est compris entre $param[ith].begin$ et $param[ith].end$, où **param** est une structure de données de type ***NUM_FACT_PARAM***. Dans le cas où nous détectons des singularités locales dans certains blocs matriciels, nous les stockons dans les paramètres $param[ith].nsing$ et $param[ith].singVals$. Nous ordonnons l'exécution de toutes les tâches concurrentes pour mettre à jour la

variable partagée $param[ith].pivot$. Nous garantissons à l'aide de mutex la cohérence en lecture et en écriture de cette variable.

Algorithme 2.2 : Répartition des tâches dans la phase numérique

Entrées : *ncore*
Sorties : *nload*, le nombre de super-nœuds traités par chaque thread.
$nbdns = nbnode(1)$;
$ntask = ncore$.
si $ntask > nbdns$ **alors**
  $ntask = nbdns$.
fin
tant que *la racine n'a pas été traitée* **faire**
  calculer **rest**, le reste de la division entière de *nbdns* par *ntask*;
  pour $ith = 1$ *à* $ntask - rest$ **faire**
    $nload = nbdns/ntask$.
  fin
  pour $ith = ntask - rest + 1$ *à* $ntask$ **faire**
    $nload = nbdns/ntask + 1$.
  fin
  si $ntask > nbnds/2$ **alors**
    $ntask = nbnds/2$;
  fin
  $l = l + 1$;
  $nbdns = nbnode(l)$;
fin

Sur la figure 2.12, nous donnons un exemple pour illustrer la répartition des tâches sur quatre cœurs. Au niveau 1, chacun des quatre cœurs dont nous disposons assemble et factorise un des blocs matriciels associés aux séparateurs Γ_1^1, Γ_2^1, Γ_3^1 et Γ_4^1. Au niveau 2, seuls deux cœurs seront actifs pour traiter les blocs matriciels associés aux super-nœuds Γ_1^2 et Γ_2^2. Au niveau 3, un seul cœur s'occupe de l'assemblage et de la factorisation du bloc matriciel $S_{\Gamma_1^3\Gamma_1^3}$ associé à la racine Γ_1^3.

La création de threads dans la phase de résolution

Nous créons des threads dans chacune des trois étapes que comporte la phase de résolution : la descente, la résolution du problème condensé aux séparateurs et la remontée.

Dans l'étape de descente, nous résolvons indépendamment les systèmes locaux $(L_{I_iI_i}D_{I_iI_i}U_{I_iI_i})Y_{I_i} = B_{I_i}$ associés aux sous-structures I_i et mettons à jour chacun des termes $B_{\Gamma_j^l}$ du second membre. Un terme $B_{\Gamma_j^l}$ dépend de toutes les solutions locales Y_{I_i} calculées au sein des sous-structures qui sont descendantes de Γ_j^l. La création de threads à cette étape consiste à répartir, dans un premier temps, la boucle sur la résolution de ces systèmes locaux sur les cœurs. Cette répartition se fait de la même façon que dans la phase d'analyse. Nous définissons, dans un deuxième temps, les paramètres à passer à la procédure sous la forme d'une structure ***SOLVE_PARAM***. Celle-ci contient, d'une part les paramètres de début

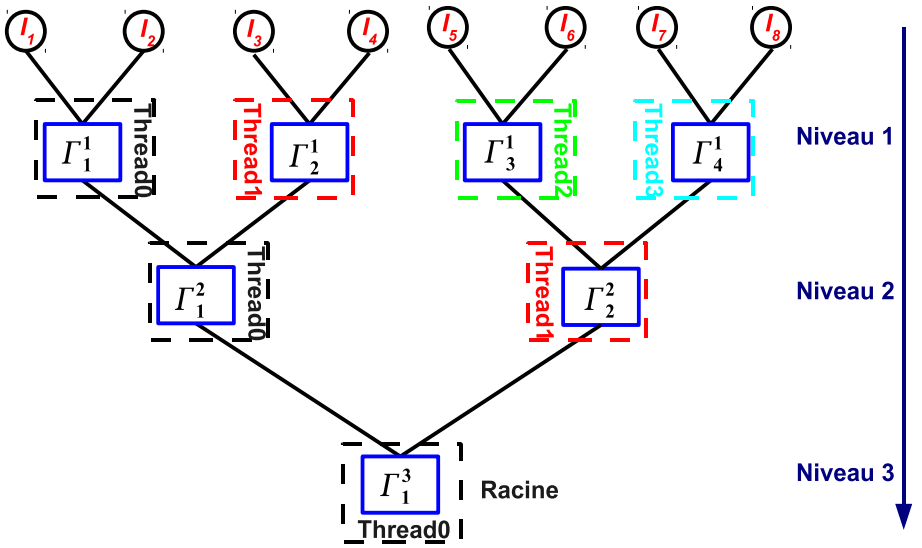


FIG. 2.12 – *Exemple de répartition de charge des threads sur quatre cœurs.*

begin et de fin **end** de chacun des threads et d'autre part, le nombre de seconds membres **nrhs** et la solution globale **X** du système $MX = B$.

```
struct SOLVE_PARAM {
        int begin
        int end
        int nrhs
        double * X
}
```

Dans un troisième temps, nous construisons la procédure à exécuter et passons en paramètre à celle-ci la structure **param** de type *SOLVE_PARAM*. Dans cette procédure, nous résolvons **nload** systèmes $(L_{I_iI_i}D_{I_iI_i}U_{I_iI_i})Y_{I_i} = B_{I_i}$. A la fin de la résolution de ces systèmes, chacun des threads met à jour tous les termes $B_{\Gamma_j^l}$ qui dépendent des solutions Y_{I_i}. Plusieurs threads peuvent à la fois accéder en lecture et en écriture à ces termes. Dans ce cas, nous créons un mutex pour définir la zone critique. Si un thread possède ce verrou, il lit et met à jour les termes $B_{\Gamma_j^l}$ concernés. Lorsqu'il a terminé, il libère le verrou, et un autre thread le prend à son tour.

Sur la figure 2.13, nous illustrons comment ce système de verrou peut être mis en place dans la phase de descente. Le découpage comporte huit sous-structures qui sont réparties sur quatre cœurs. Par exemple, les threads 0 et 1 calculent respectivement les solutions locales Y_{I_1}, Y_{I_2} et Y_{I_3}, Y_{I_4}, et partagent l'accès en lecture et en écriture sur les termes de $B_{\Gamma_1^2} = B_{\Gamma_1^2} - \sum_{i=1}^{4} M_{\Gamma_1^2 I_i} Y_{I_i}$.

Dans la résolution du problème condensé aux séparateurs, la solution est obtenue en deux étapes : une élimination et une substitution. Chacune de ces deux étapes est effectuée niveau par niveau. Dans l'étape d'élimination, nous calculons les solutions $Y_{\Gamma_j^l}$ à un niveau **l** et mettons à jour les termes $B_{\Gamma_j^m}$ aux niveaux supérieurs $\mathbf{m} > \mathbf{l}$. Les dépendances existant dans ces mises à jour sont très complexes. En plus, le calcul de la solution

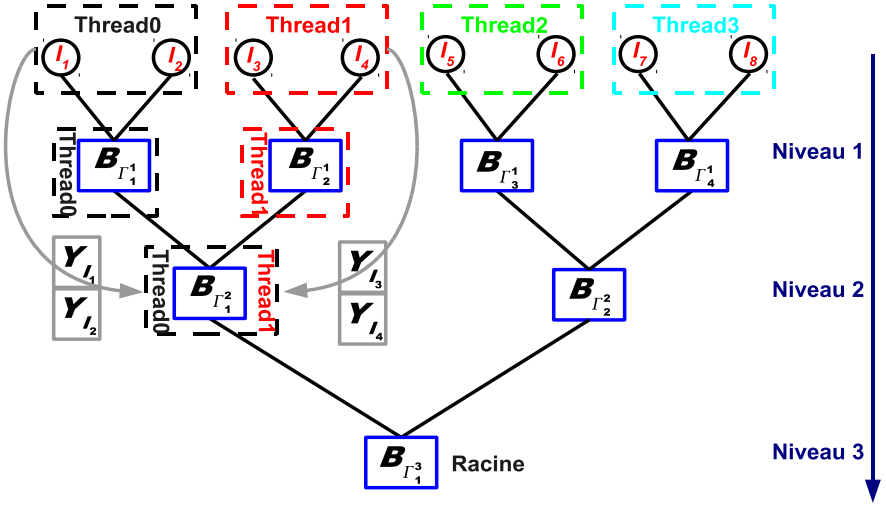


FIG. 2.13 – *Mise à jour du terme* $B_{\Gamma_1^2}$ *partagé par les threads* 0 *et* 1.

$Y_{\Gamma_j^l}$ associée à chacun des séparateurs Γ_j^l et la mise à jour des termes $B_{\Gamma_k^m}$ concernés sont faits dans une même fonction. Nous avons donc décidé de créer juste des threads dans l'étape de substitution. La création des threads consiste à :

1. répartir, à chacun des niveaux **l** du super-arbre, les tâches portant sur les séparateurs Γ_j^l en appliquant l'algorithme 2.3 ;
2. définir la structure **param** de type *SOLVE_PARAM* que nous passerons en paramètre ;
3. construire la procédure exécutée par chacun des threads dans laquelle sont calculées **nload** solutions $X_{\Gamma_j^l}$.

Algorithme 2.3 : Répartition des tâches dans l'étape de substitution

Initialisation : $ntask = 1$;
$nload$: le nombre de super-nœuds traités par tâche.
tant que $l > 0$ **faire**
  calculer $rest$, le reste de la division de $nbnode(l)$ par $ntask$;
  pour $ith = 1$ *à* $ntask - rest$ **faire**
    $nload = nbnode(l)/ntask$.
  fin
  pour $ith = ntask - rest + 1$ *à* $ntask$ **faire**
    $nload = nbnode(l)/ntask + 1$.
  fin
  $l = l + 1$;
  si $ntask > nbnode(l)$ **alors**
    $ntask = nbnode(l)$;
  sinon
    $ntask = ncore$.
  fin
fin

Dans l'étape de descente du problème global, nous calculons cha-

cune des solutions locales X_{I_i} indépendamment des autres. La création de threads consiste à :

1. utiliser l'algorithme 2.1 pour répartir les tâches portant sur la boucle de calcul de ces solutions X_{I_i} ;
2. définir la structure **param** de type *SOLVE_PARAM* que nous passerons en paramètre ;
3. construire la procédure exécutée par chacun des threads dans laquelle sont calculées **nload** solutions X_{I_i}.

2.3.2 Version multi-threads avec OpenMP

Nous utilisons des sous-programmes de la bibliothèque BLAS dans la plupart des trois grandes phases du solveur. Ces procédures sont optimisées avec le standard OpenMP. La mise en place de la version multi-threads avec OpenMP consiste simplement à activer la librairie BLAS optimisée dans le solveur. Pour ce faire, nous compilons le solveur direct avec l'option d'interprétation des directives OpenMP, c'est-à-dire **-fopenmp** pour les compilateurs de GNU (**gfortran**, **gcc**, **g++**) ou **-openmp** pour ceux d'Intel (**ifort**, **icc**, **icpc**). Nous générons alors un code parallèle dans les différentes parties du solveur où nous utilisons du BLAS. Ces parties sont la phase numérique et la phase de résolution des systèmes triangulaires. A l'exécution du solveur, nous définissons au moyen de la variable d'environnement *OMP_NUM_THREADS* le nombre maximum de threads à créer. Ce nombre est toujours égal à **ncore**, le nombre de cœurs disponibles.

2.3.3 Évaluation des deux versions multi-threads

Nous avons mis en place deux versions multi-threads du solveur : une version avec des threads POSIX et une version optimisée avec OpenMP. Dans cette partie, nous avons pour but d'évaluer et analyser les performances de ces deux versions sur une machine bi-processeurs quadri-cœurs offrant la possibilité d'exécuter jusqu'à huit tâches en parallèles. Pour cela, nous réalisons plusieurs tests sur des problèmes d'élasticité linéaire en $3D$ ayant différentes tailles. Dans cette sous-section, nous nous penchons sur les temps de résolution du problème ayant **397953** degrés de liberté. L'analyse des résultats obtenus sur un autre type de problème est faite dans l'annexe A.4. Le super-arbre généré lors de la résolution de ce problème est de hauteur de **10**. Il comporte donc **10** niveaux et **512** sous-structures au plus bas niveau. Nous mesurons, pour chacune des versions, les temps de calcul nécessaires pour réaliser chacune des trois grandes phases de mise en œuvre du solveur direct. Nous nous intéressons aux temps réels de calcul (*elapsed*) plutôt qu'aux temps CPU car les temps *elapsed* sont, en général, plus significatifs en multi-threading comme le montre l'exemple 2.14).

C'est à partir des temps *elapsed* que nous allons faire les calculs d'accélération sur lesquels nous nous penchons pour analyser les performances en multi-threading des deux versions exécutées avec **p** threads. Cette accélération est définie comme étant le rapport entre le temps réel de référence obtenu avec un thread et le temps écoulé pour une exécution du solveur sur **p** cœurs.

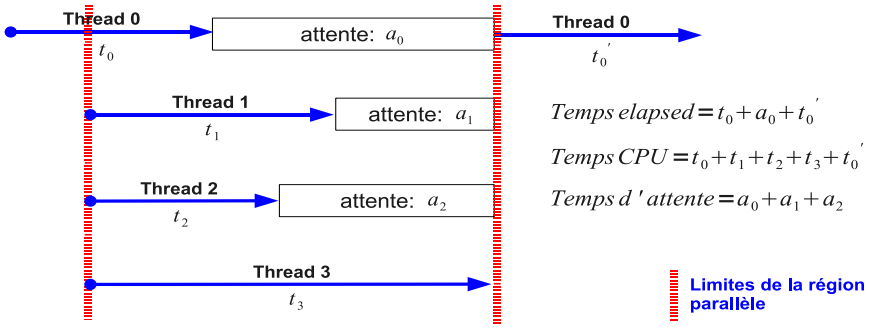


FIG. 2.14 – *Différents types de temps dans un programme multi-threads.*

Les temps de calcul pour la phase d'analyse

Les temps nécessaires pour analyser le système linéaire à résoudre concernent la renumérotation des inconnues (bissection récursive et construction du super-arbre d'élimination) et la factorisation symbolique des **512** matrices $M_{I_i I_i}$ associées aux sous-structures.

Sur la figure 2.15, nous présentons l'accélération des calculs quand le nombre de threads créés croît. Nous remarquons que l'accélération de la version optimisée avec OpenMP est, sans surprise, identique à **1** quel que soit le nombre de threads. Les temps obtenus en parallèle sont identiques à celui obtenu en séquentiel. La raison est que nous n'utilisons pas des sous-programmes de BLAS dans cette phase d'analyse. Nous observons aussi une faible évolution de l'accélération de la version Pthreads. Elle passe de **1** à **1.07** quand le nombre de threads passe de **1** à **8**. Cette faible amélioration des temps de calcul est due au fait que seule l'étape de factorisation symbolique (**SymbFact**) était parallélisable. Or, cette étape ne coûte pas grand chose comparée à celle de la construction de l'arbre (**BuildTree**) comme nous pouvons l'observer sur le tableau 2.1. Les bonnes performances en parallèle obtenues pendant la factorisation symbolique (figure 2.16) sont donc fortement influencées par l'étape de bissection récursive sur les inconnues.

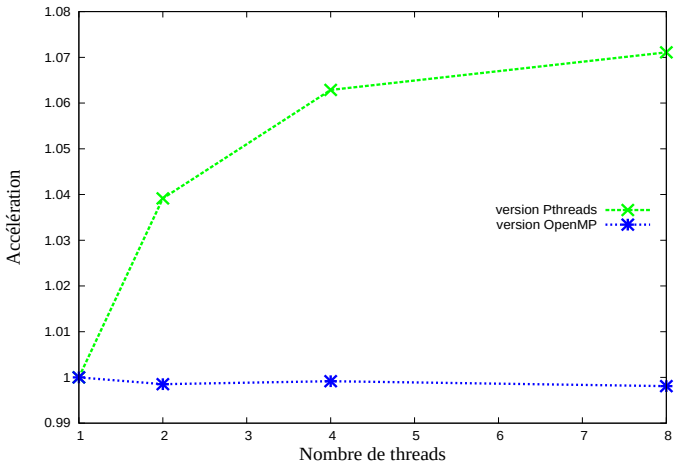


FIG. 2.15 – *Accélération dans la phase d'analyse du système.*

Étapes \ p	1	2	4	8
SymbFact	1.452	0.801	0.433	0.292
BuildTree	15.374	15.381	15.378	15.396

TAB. 2.1 – *Temps elapsed (s) des étapes d'analyse du système.*

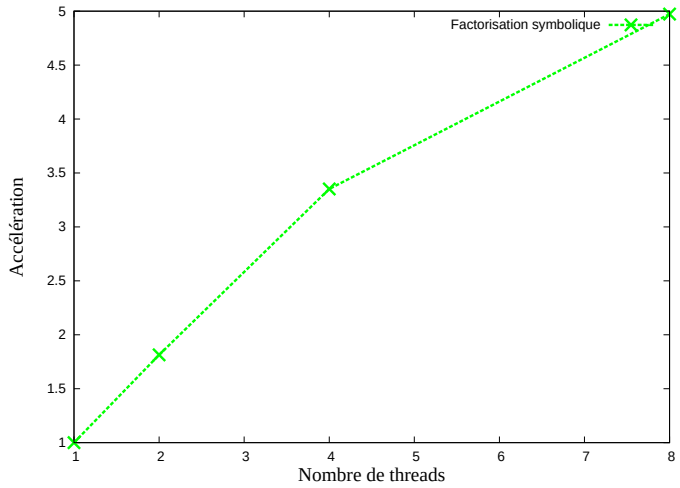


FIG. 2.16 – *Accélération dans l'étape de factorisation symbolique.*

Les temps de calcul pour la phase numérique

Dans la phase numérique, nous évaluons d'une part, les temps nécessaires pour effectuer la décomposition LDU des matrices associées aux sous-structures I_i et calculer les contributions locales $M_{\Gamma_j^l I_i} {M_{I_i I_i}}^{-1} M_{I_i \Gamma_k^m}$; nous mesurons d'autre part les temps nécessaires pour assembler et factoriser le complément de Schur S. Pour analyser ces derniers, nous évaluons d'abord, les temps de calcul nécessaires des deux versions multi-threads pour effectuer la factorisation par élimination de Gauss des blocs pleins $S_{\Gamma_j^l \Gamma_j^l}$ et calculer les contributions $S_{\Gamma_k^m \Gamma_j^l} {S_{\Gamma_j^l \Gamma_j^l}}^{-1} S_{\Gamma_j^l \Gamma_p^n}$ à chacun des niveaux l du super-arbre d'élimination ($\mathbf{1} \leq \mathbf{l} \leq \mathbf{9}$). Ensuite, nous mesurons les temps d'exécution (temps d'assemblage et de factorisation) du complément de Schur en fonction du nombre de cœurs.

Sur la figure 2.17, nous représentons, pour chacune des versions Pthreads et OpenMP, l'accélération des étapes de factorisation numérique des matrices $M_{I_i I_i}$ et de calcul des contributions locales en fonction du nombre de threads.

Des résultats présentés sur la figure 2.17, nous remarquons une réduction manifeste des temps de factorisation dans la version Pthreads. Les temps sont quasiment réduits par quatre pour une exécution sur quatre cœurs. Cependant, la version optimisée avec OpenMP est décevante. Nous ne gagnons presque rien en passant à huit threads et nous perdons même du temps sur deux ou quatre cœurs. Ce comportement peut s'expliquer facilement par le fait que le solveur a été bien optimisé en séquentiel avec des sous-programmes de BLAS sur des matrices de petites dimensions ($< \mathbf{120}$). Le gain devient alors négligeable quand nous passons à la version OpenMP multi-threads. Ces matrices de petites tailles sont les blocs

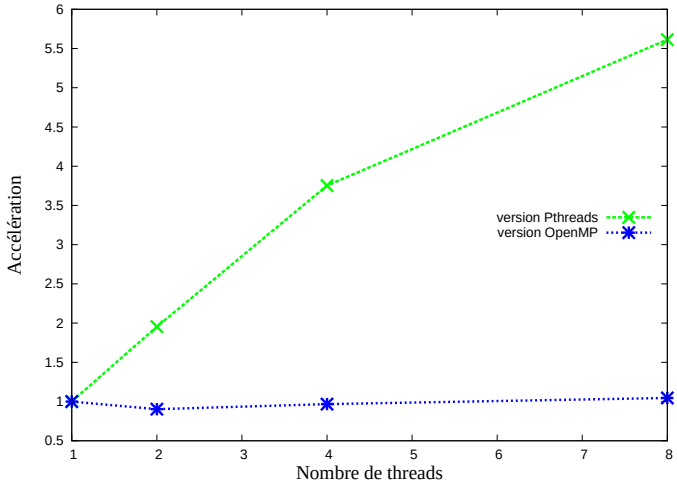


FIG. 2.17 – *Accélération dans l'étape de factorisation locale.*

de la structure tridiagonale des matrices associées aux sous-structures I_i. Nous rappellons que la taille moyenne **mid** des sous-structures les plus représentées est environ égale à **1272** inconnues internes et interfaces (voir le tableau 1.4).

Nous regroupons respectivement sur les figures 2.18, 2.19 et 2.20, les temps en fonction du niveau l et sur deux, quatre et huit cœurs pour la factorisation et le calcul de toutes les contributions des blocs matriciels $S_{\Gamma_j^l \Gamma_j^l}$ à apporter aux ascendants de Γ_j^l.

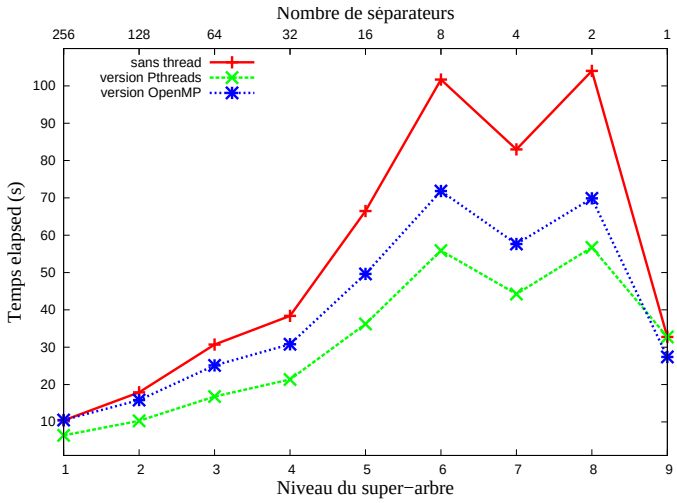


FIG. 2.18 – *Temps de factorisation de Schur sur deux cœurs.*

Sur les figures 2.18, 2.19 et 2.20, nous observons que si le nombre de super-nœuds ($\mathbf{2^{9-l}}$), à un niveau **l** de l'arbre d'élimination, est supérieur ou égal au nombre de cœurs disponibles, alors la version parallélisée manuellement avec les threads POSIX est meilleure que celle optimisée avec OpenMP. Dans le cas contraire, la tendance est inversée. Ceci est dû au fait qu'à ces niveaux, nous ne pouvons pas créer assez de threads. Du coup, nous ne tirons pas profit de tous ces cœurs avec la version Pthreads. Il est intéressant de noter que le gain de performances reste tout de même satisfaisant pour l'une ou l'autre des deux versions parallèles comparée à la version séquentielle du solveur. Au niveau **6** de l'arbre, ce gain peut

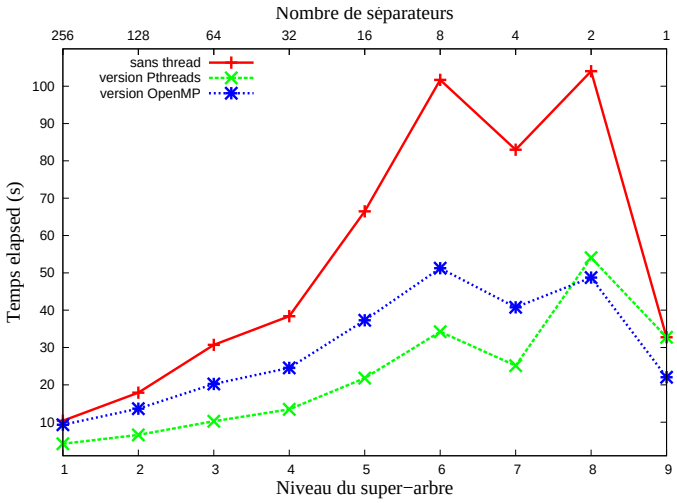


FIG. 2.19 – *Temps de factorisation de Schur sur quatre cœurs.*

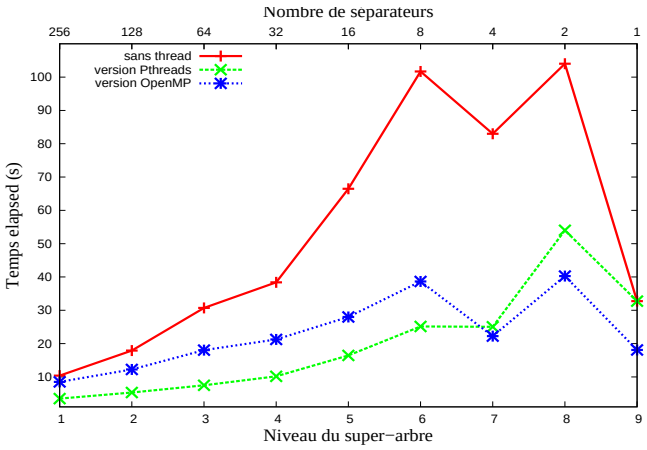


FIG. 2.20 – *Temps de factorisation de Schur sur huit cœurs.*

même atteindre jusqu'à un facteur **3** pour quatre threads, prouvant ainsi que l'extensibilité de ces version est bonne dans cette partie de la factorisation. Les faibles performances observées à la racine (niveau **9**) s'expliquent par le fait qu'il n'existe aucune contribution à calculer à cet endroit mais, nous effectuons tout de même une décomposition LDU du bloc $S_{\Gamma_1^9\Gamma_1^9}$.

Nous retenons de toutes ces observations que la version Pthreads exploite mieux le multi-threading aux niveaux de la factorisation où il y a un très grand nombre de super-nœuds à traiter. Quant à la version OpenMP, elle permet d'exploiter plus de parallélisme aux plus haut niveaux du super-arbre d'élimination.

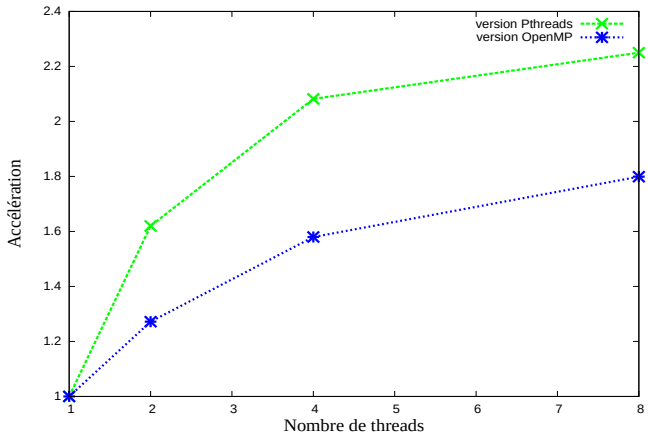


FIG. 2.21 – *Accélération sur l'exécution du complément de Schur.*

Sur la figure 2.21, nous avons présenté pour chacune des versions Pthreads et OpenMP, l'accélération des calculs pour l'assemblage et la factorisation du complément de Schur en fonction du nombre de threads. Ces résultats démontrent bien que la version Pthreads est plus performante quand nous exécutons le complément de Schur.

Les temps de calcul pour la phase de descente-remontée

Nous comparons les temps de calcul nécessaires pour effectuer en parallèle sur deux, quatre et huit cœurs, la phase de résolution des systèmes triangulaires. Les résultats présentés sur la figure 2.22 montrent cette fois-ci que la tendance est plutôt favorable à la version optimisée avec OpenMP même si les temps de résolution sont faibles. Le plus important de ce gain viendrait de la résolution du problème condensé aux séparateurs, où les blocs matriciels traités avec des procédures de BLAS sont de tailles plus grandes (voir tableau le 1.4).

Les temps d'exécution en fonction du nombre de threads

Dans cette partie, nous exhibons le gain en temps d'exécution totale de chacune des deux versions parallèles du solveur. Pour cela, nous mesurons les temps de calcul de chacune de ces versions en fonction du nombre de threads et nous analysons leur accélération. Les résultats obtenus sont présentés sur la figure 2.23.

En analysant les résultats sur la figure 2.23, nous observons qu'en utilisant la version Pthreads ou OpenMP, nous réduisons significativement

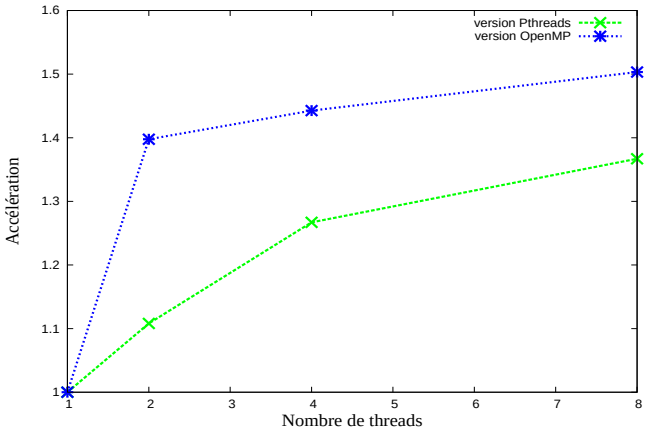


FIG. 2.22 – *Accélération de la phase de descente-remontée.*

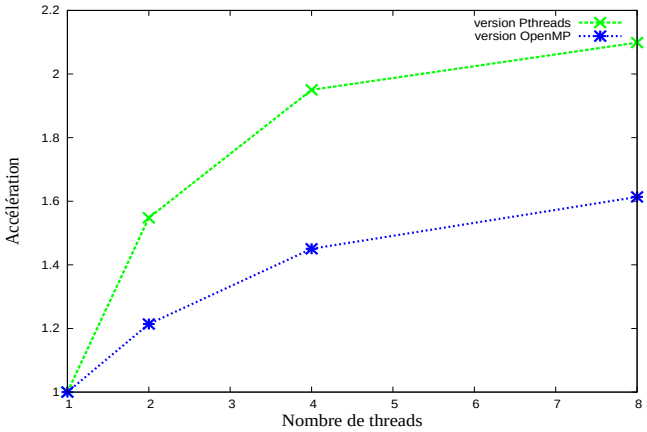


FIG. 2.23 – *Accélération des versions multi-threads.*

les temps d'exécution. Par exemple, la version Pthreads nous a permis de réduire par un facteur supérieur à **2** les temps de calcul sur quatre cœurs. Mais, les performances obtenues avec la version mise en place avec des threads POSIX restent quand même les meilleures. Les versions Pthreads et OpenMP du solveur nous ont permis certes de réduire significativement les temps de calcul sur une architecture multi-cœurs, mais la puissance de calcul reste relativement limitée. Nous pouvons bien travailler avec une de ces deux versions, mais la puissance de calcul potentielle accessible par une version hybride où nous alternons des threads POSIX et OpenMP pourrait être nettement plus importante. Il serait donc judicieux de pouvoir mettre en place cette version hybride afin de pouvoir tirer profit au maximum des machines multi-cœurs.

2.3.4 Mise en place d'une version multi-threads hybride

La mise en œuvre de la version multi-threads hybride consiste à prendre la meilleure version dans chacune des parties parallélisées. Pour cela, nous alternons donc niveau par niveau, les threads POSIX et OpenMP dans l'étape de factorisation du complément de Schur. En effet, c'est la partie du solveur la plus coûteuse en temps et dans laquelle la version optimisée avec OpenMP est parfois plus performante à certains niveaux. Nous avons alors réfléchi sur la manière de développer cette version hybride en compliquant aussi peu que possible le solveur, afin de coller au maximum à sa version séquentielle. La compatibilité entre les threads POSIX et OpenMP a déjà été étudiée et il est effectivement possible de les combiner.

Dans l'étape de factorisation du complément de Schur, le couplage des threads POSIX et OpenMP se fait facilement. La stratégie que nous avons adoptée à chacun des niveaux **l** ($\mathbf{l} \geq \mathbf{1}$), est la suivante :

- si le nombre de super-nœuds $\mathbf{nbnode}(\mathbf{l}) < \mathbf{ncore}$ alors nous activons la librarie BLAS optimisée avec le standard OpenMP et nous faisons appel au sous-programme ***omp_set_num_threads***$(ncore)$. Cette fonction, définie dans le fichier d'entête ***omp.h***, permet de spécifier un nombre **ncore** de tâches à l'entrée d'une région parallèle. Il est donc indispensable d'inclure ce fichier dans le solveur.
- sinon, nous basculons à la version parallélisée avec les Pthreads.

Une fois que cette stratégie est mise en place, nous compilons le solveur avec le flag **-fopenmp** (GNU OpenMP) ou **-openmp** (Intel MKL multi-threads basé sur les directives OpenMP) pour tenir compte des sous-programmes de OpenMP. L'édition de liens se fera par la suite avec l'option **-lgomp** (GNU OpenMP) ou **-liomp5** (Intel MKL multi-threads basé sur les directives OpenMP).

2.4 Les performances en multi-threading

Nous allons maintenant analyser les performances de la version multi-threads hybride basée sur les threads POSIX et la librairie MKL optimisée avec OpenMP. Pour ce faire, nous effectuons les mêmes calculs sur des problèmes d'élasticité linéaire en $3D$ ayant **206763** et **397953** degrés de liberté. Les calculs sont réalisés sur une machine bi-processeurs quadri-cœurs. Nous analysons les temps de calcul obtenus avec cette version

multi-threads hybride et nous les comparons à ceux des solveurs DSCPack et MUMPS. Ces deux derniers sont utilisés comme multi-threadés en activant juste la librairie BLAS optimisée avec OpenMP.

2.4.1 Analyse des performances

Nous présentons respectivement, sur les figures 2.24 et 2.25, les courbes sur l'accélération pour exécuter les problèmes de taille **206763** et **397953** degrés de liberté. Sur ces figures, nous constatons que, par rapport aux temps d'exécution de la version Pthreads, ceux de la version hybride sont encore légèrement réduits sur **4** et **8** cœurs.

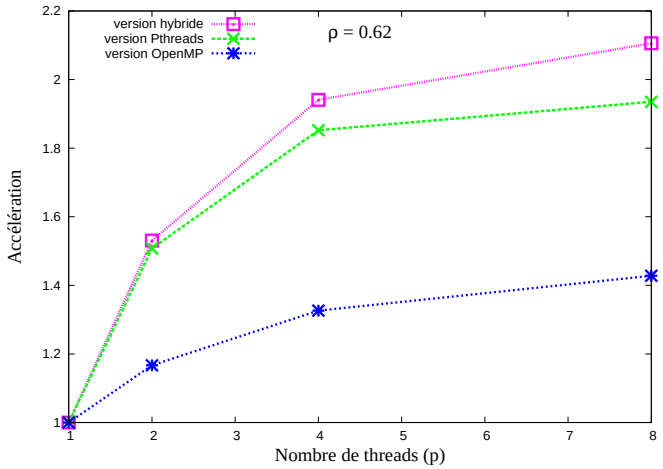


FIG. 2.24 – *Accélération du solveur multi-threads : problème à* 206763 *ddl.*

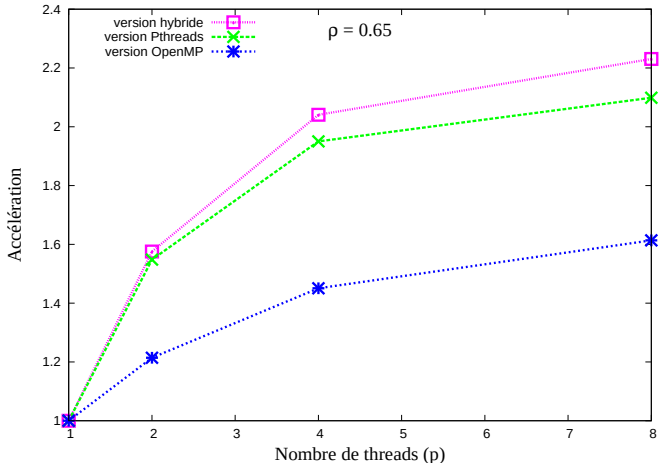


FIG. 2.25 – *Accélération du solveur multi-threads : problème à* 397953 *ddl.*

A partir du temps d'exécution total de la version hybride du solveur, nous pouvons connaître approximativement ρ, la portion du solveur multithreads (parallélisable). Cette approximation consiste à utiliser la fonction $T_p = \rho T_1/p + (1-\rho)T_1$ reliant $\mathbf{T_p}$, le temps nécessaire pour résoudre un problème donné sur $\mathbf{p}$ cœurs et $\mathbf{T_1}$, le temps d'exécution séquentielle sur un cœur, puis effectuer un ajustement (ou *fitting* en anglais) sur celle-ci. La portion que nous obtenons est alors égale à environ $\mathbf{2/3}$ ($\rho = \mathbf{62\%}$ pour le problème de taille **206763** ddl et $\rho = \mathbf{65\%}$ pour celui de taille **397953**

ddl). Ce qui veut dire qu'il reste au moins **1/3** du code qui n'est pas parallélisée. Bien que ces résultats soient satisfaisants, ils peuvent donc encore être améliorés.

2.4.2 Comparaison avec DSCPack et MUMPS

Dans cette partie, nous essayons de comparer les performances de la version hybride avec celles de DSCPack et MUMPS. Nous utilisons ces deux derniers comme solveurs multi-threads en activant juste la bibliothèque BLAS optimisée avec OPenMP. Nous montrons sur les figures 2.26 et 2.27 les performances que nous obtenons respectivement pour les problèmes à **206763** et **397953** inconnues. Sur ces résultats, nous observons que les écarts de temps qui existaient en séquentiel (mono-thread) entre le solveur que nous avons mis en œuvre et les autres sont réduits, ce qui prouve en partie que la stratégie de parallélisation que nous avons proposée se comporte bien. Le gain de performances maximal atteint est de l'ordre de **2.2** sur quatre cœurs.

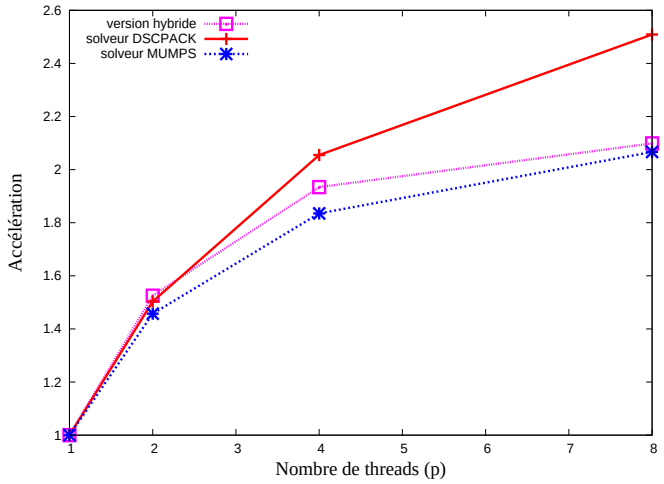


FIG. 2.26 – *Comparaison avec DSCPack et MUMPS : problème à* 206763 *ddl.*

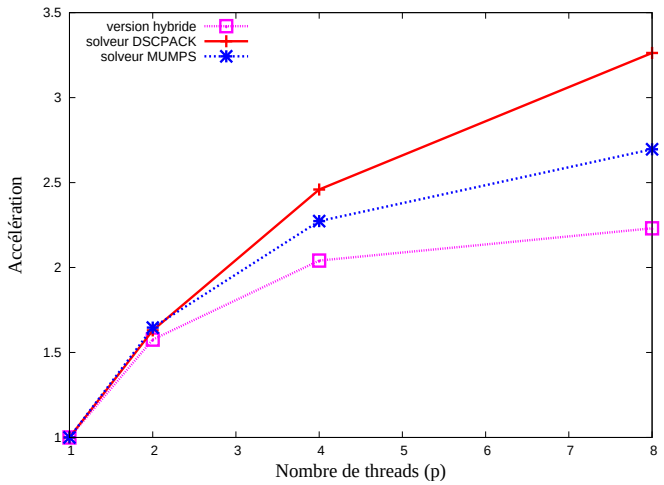


FIG. 2.27 – *Comparaison avec DSCPack et MUMPS : problème à* 397953 *ddl.*

Conclusion

Dans ce chapitre, nous avons proposé la mise en œuvre d'une version multi-threads hybride du solveur basée sur la bibliothèque Pthreads et celle de BLAS optimisée avec OpenMP. Malgré quelques difficultés rencontrées sur les dépendances des calculs, nous avons réussi à créer dans cette version, des threads dans plus de **2/3** des parties du solveur. Cette version a été portée sur une architecture à mémoire partagée. Ses performances ont été mesurées et analysées sur une machine bi-processeurs quadri-cœurs et la comparaison avec d'autres solveurs a été ainsi faite. Les analyses nous ont permis de voir que les gains sont plus importants avec un petit nombre de cœurs (**2** ou **4**) sur ce type de machine. Mais ces gains n'arrivent pas à dépasser **3** qui semble être une asymptote pour le moment. Ce serait bien d'exploiter la part séquentielle du solveur (**1/3**) pour envisager de tirer plus parti des machines disposant de **16** cœurs, voire **80** cœurs. La suite de ces travaux consistera à intégrer cette version hybride dans les méthodes de décomposition de domaine de type FETI pour inverser leurs problèmes locaux en parallèle.

3 Amélioration et évaluation des performances de la méthode FETI

Sommaire

Introduction . . . 67
3.1 La présentation de la méthode FETI . . . 68
3.1.1 Description de la version initiale de FETI . . . 68
3.1.2 Résolution itérative du problème d'interface . . . 70
3.1.3 Algorithme détaillé de FETI-1 . . . 72
3.2 Les facteurs critiques d'une méthode FETI . . . 75
3.2.1 Choix d'un solveur local . . . 75
3.2.2 Traitement des singularités et calcul des pseudo-inverses . 76
3.2.3 Gestion du problème grossier . . . 77
3.3 L'intégration d'un parallélisme à deux niveaux dans FETI 78
3.3.1 Extraction en parallèle des colonnes de $B^{(s)}$ . . . 78
3.3.2 Calcul en parallèle des pseudo-inverses et pseudo-solutions 80
3.4 L'évaluation des performances de la méthode FETI . . . 82
3.4.1 Environnement d'évaluation . . . 82
3.4.2 Critères de mesure de performances . . . 84
3.4.3 Description des cas tests parallèles . . . 85
3.4.4 Analyse de l'efficacité de FETI . . . 85
3.4.5 Analyse de l'extensibilité de FETI . . . 92
Conclusion . . . 98

Dans ce chapitre, nous introduisons d'abord les méthodes de décomposition de domaine sans recouvrement de type FETI de façon générique en montrant les caractéristiques communes à toutes les variantes. Nous présentons ensuite l'algorithme de FETI classique appelée aussi FETI-1 (*one-level* FETI en anglais) et nous soulevons les questions qui déterminent ses points critiques. Puis, nous montrons comment un parallélisme à deux niveaux peut s'intégrer dans cette méthode. Nous inversons les problèmes locaux en parallèle et nous discutons de l'approche qui est utilisée pour améliorer la phase itérative de FETI. Enfin, nous évaluons et analysons les performances de cette version améliorée des méthodes FETI en réalisant des tests parallèles, avec le code de calcul ZéBuLon, sur le cluster du Centre des Matériaux (MINES ParisTech) et le supercalculateur JADE du Centre Informatique National de l'Enseignement Supérieur (CINES).

Introduction

Les solveurs directs creux, qui ont été pendant des années le principal objet d'intérêt dans les logiciels de calcul pour la simulation numérique de structures par la méthode des éléments finis (MEF) (voir l'annexe A.3), continuent de jouer un rôle important dans ces codes. Néanmoins, avec le besoin pressant pour analyser les plus grands modèles mécaniques comportant plusieurs millions de degrés de liberté et les capacités de stockage requises par ces solveurs directs croissant rapidement avec la taille des problèmes, une grande partie de la communauté des mécaniciens utilisateurs et des développeurs de codes s'est orientée vers les méthodes itératives. Les importants progrès effectués au cours de ces deux dernières décennies dans la mise en œuvre d'algorithmes itératifs robustes pour la résolution des problèmes sur des structures mécaniques de type plaques et coques et l'avènement des calculateurs parallèles, sont deux autres facteurs clés de ce changement d'orientation. Ces deux derniers facteurs sont tellement liés que les méthodes itératives sont plus souples en parallèle que les méthodes directes. Les solveurs itératifs ont toutefois du mal à percer en mécanique des structures industrielles où sont souvent cumulés des hétérogénéités, des non-linéarités et des couplages de modèles qui dégradent le conditionnement des matrices de rigidité associées.

C'est ainsi qu'est venue l'idée de coupler ces deux types de solveur, direct et itératif, au sein d'une troisième catégorie, les méthodes de décomposition de domaine sans recouvrement, afin de tirer parti de leurs avantages respectifs. Ces techniques proposent de partitionner un domaine initial en plusieurs sous-domaines, pour lesquels on va pouvoir résoudre, de préférence sur des processeurs distincts, des problèmes locaux presque indépendants. Ces problèmes locaux sont de même nature que le problème global. Il est donc possible de les inverser tous par le même solveur. En général, par souci de robustesse, on fait appel à un solveur direct car les conditionnements des matrices associées à ces problèmes locaux sont souvent aussi défavorables que celui du problème initial et les tailles de ces matrices restent conséquentes. Parfois, un bon découpage peut diminuer significativement la largeur de bande de ces matrices de rigidité locales et donc accélérer d'autant leur inversion. Les résolutions locales étant faites, on opère ensuite sur la continuité des solutions locales aux interfaces entre sous-domaines *via* un solveur d'interface adéquat : inversion du complément de Schur pour la méthode itérative de Schur primale BDD ("Balanced Domain Decomposition") (Mandel 1993; Le Tallec 1994) ou construction de son opérateur dual pour la méthode FETI ("Finite Element Tearing and Interconnecting") (Farhat et Roux 1991; 1994). La matrice de ce problème d'interface est par contre bien mieux conditionnée que celle du problème global, de plus petite taille et elle combine les données condensées de part et d'autre. Pour faciliter le parallélisme et ne pas avoir à la construire explicitement, une méthode itérative de type gradient conjugué (CG) (Lascaux et Théodor 2004b) est utilisée. Les gains en performances de ces méthodes de décomposition de domaine jouent sur plusieurs facteurs notamment les temps CPU des solveurs directs et un parallélisme à gros grains et multi-niveaux (solveur d'interface, solveur local, ...).

La méthode itérative de Schur primale et la méthode duale FETI sont

celles qui se sont imposées le plus en mécanique des structures. Elles ont fait leurs preuves sur des études récentes pour le traitement de problèmes comportant des millions d'inconnues sur des machines à plusieurs milliers de processeurs (Bhardwaj et al. 2000). Elles ont ainsi démontré une scalabilité respectant en même temps la taille du problème et le nombre de sous-domaines. En particulier, la scalabilité parallèle de la méthode FETI et sa capacité à surpasser plusieurs algorithmes direct et itératif sur des calculateurs séquentiels et parallèles ont été largement prouvées (Farhat et al. 1994b; 1998). Leurs périmètres d'utilisation sont larges : mécanique non linéaire, analyse modale et vibratoire (Géradin et al. 1997), problème d'évolution, acoustique (Farhat et al. 2000a), etc. Conséquence de ce succès, chaque adaptation entraîne sa variante et plusieurs tentatives de généralisation ont tenté de les réunir, d'où une multitude d'approches : FETI-1, FETI-2, FETI-DP, mixte (Dureisseix et Ladevèze 1998), etc. Dans ce chapitre, nous nous limiterons à la présentation de la méthode FETI classique, FETI-1, qui est celle initialement développée dans le code ZéBuLon.

3.1 La présentation de la méthode FETI

La méthode FETI a été introduite par Farhat et Roux (1994). Le principe de cette méthode consiste à restreindre la résolution d'un problème global sur un domaine Ω à celle d'un problème d'interface plus petit et mieux conditionné. Les inconnues sont les forces λ à imposer aux interfaces Γ entre sous-domaines $\Omega^{(s)}$ de manière à assurer la continuité des solutions locales $x^{(s)}$, autrement dit, annuler le saut de déplacement aux interfaces entre sous-domaines. Ces forces sont déterminées par une méthode itérative, par exemple, un algorithme de gradient conjugué.

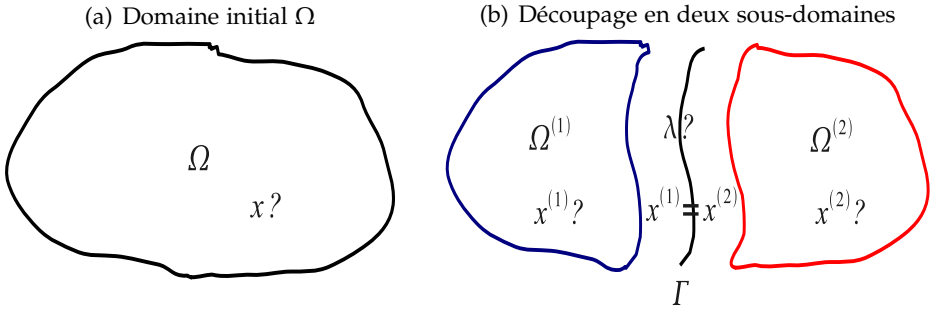


Fig. 3.1 – *Principe de la méthode FETI.*

Dans cette section, nous allons d'abord faire une description générale de la version initiale de FETI : FETI-1. Ensuite, nous parlons de la résolution itérative de son problème d'interface. Et pour terminer, nous détaillons l'algorithme classique de cette méthode.

3.1.1 Description de la version initiale de FETI

La simulation numérique des problèmes en mécanique des structures par la MEF conduit à la résolution d'un système linéaire de la forme :

$$Mx = b \tag{3.1}$$

où M est une matrice de rigidité symétrique définie ou semi-définie positive issue de la discrétisation par la MEF du problème sur un domaine initial Ω, x est le vecteur des déplacements recherchés et b le vecteur des forces imposées. La méthode FETI-1 consiste à, partitionner dans un premier temps, la structure du domaine de calcul Ω en N_s sous-domaines $\Omega^{(s)}$, et remplacer, dans un deuxième temps, le système linéaire (3.1) par le système d'équations suivant :

$$\begin{cases} M^{(s)}x^{(s)} = b^{(s)} - {R^{(s)}}^T\lambda, \ s = 1, 2, \ \ldots, \ N_s \\ \sum_{s=1}^{N_s} R^{(s)}x^{(s)} = 0 \end{cases} \tag{3.2}$$

où $M^{(s)}$ et $b^{(s)}$ sont respectivement les restrictions de M et b au sous-domaine $\Omega^{(s)}$, λ est le vecteur des multiplicateurs de Lagrange introduits pour assurer la continuité de chacune des solutions locales $x^{(s)}$ à l'interface $\Gamma^{(s)}$ entre $\Omega^{(s)}$ et les autres (voir la seconde équation de (3.2)). Les opérateurs $R^{(s)}$ sont des matrices booléennes signées qui décrivent les interconnectivités entre les sous-domaines. La première équation de (3.2) exprime l'équilibre local des sous-structures $\Omega^{(s)}$ avec des conditions aux limites essentielles héritées du problème global et des forces imposées (conditions aux limites de Neumann) aux interfaces $\Gamma^{(s)}$. En général, une partition du domaine Ω pourrait contenir un nombre N_f de sous-domaines flottants ($N_f < N_s$), c'est-à-dire que certaines sous-structures n'ont pas assez de conditions aux limites essentielles pour prévenir les mouvements de corps rigide et ainsi empêcher les matrices de rigidité locales $M^{(s)}$ d'être singulières. Dans ce cas, N_f des problèmes locaux de Neumann sont mal posés. Une condition nécessaire et suffisante pour que ces problèmes locaux aient au moins une solution est :

$$(b^{(s)} - {R^{(s)}}^T\lambda) \perp Ker(M^{(s)}), \ s = 1, 2, \ \ldots, \ N_f. \tag{3.3}$$

La condition (3.3) signifie que le second membre de la première équation de (3.2) doit appartenir à l'image de $M^{(s)}$. Si tel est le cas, les solutions locales du système (3.2) sont alors données par :

$$x^{(s)} = {M^{(s)}}^+(b^{(s)} - {R^{(s)}}^T\lambda) + B^{(s)}\alpha^{(s)}, \ s = 1, 2, \ \ldots, \ N_f \tag{3.4}$$

où ${M^{(s)}}^+$ désigne l'inverse de $M^{(s)}$ si elle est inversible ou le pseudo-inverse de $M^{(s)}$ si $\Omega^{(s)}$ est une sous-structure flottante. Ce type de pseudo-inverse n'est pas calculé explicitement. Dans le dernier cas, les colonnes de $B^{(s)}$ caractérisent les mouvements de corps rigide de $\Omega^{(s)}$, c'est-à-dire qu'elles forment une base du noyau $Ker(M^{(s)})$, et $\alpha^{(s)}$ est un ensemble d'amplitudes spécifiant la contribution de $B^{(s)}$ appportée à la solution $x^{(s)}$. L'introduction de nouvelles inconnues $\alpha^{(s)}$ est compensée par les équations supplémentaires résultant de la condition de solvabilité (3.3) :

$${B^{(s)}}^T(b^{(s)} - {R^{(s)}}^T\lambda) = 0, \ s = 1, 2, \ \ldots, \ N_f. \tag{3.5}$$

Le remplacement de la solution locale $x^{(s)}$, dans l'équation de continuité de (3.2), par son expression définie par (3.4) et la prise en compte de l'équation (3.5), conduisent au **problème d'interface** FETI :

$$\begin{bmatrix} F_I & -G_I \\ -{G_I}^T & 0 \end{bmatrix} \begin{bmatrix} \lambda \\ \alpha \end{bmatrix} = \begin{bmatrix} d \\ -e \end{bmatrix} \tag{3.6}$$

avec :

$$\begin{cases} F_I = \sum_{s=1}^{N_s} R^{(s)} {M^{(s)}}^{+} {R^{(s)}}^T, \\ G_I = [R^{(1)} B^{(1)} \ \cdots \ R^{(N_f)} B^{(N_f)}], \ \ \alpha = [{\alpha^{(1)}}^T \ \cdots \ {\alpha^{(N_f)}}^T]^T, \quad . \\ e = [{b^{(1)}}^T R^{(1)} \ \cdots \ {b^{(N_f)}}^T R^{(N_f)}]^T \ \text{ et } \ d = \sum_{s=1}^{N_s} R^{(s)} {M^{(s)}}^{+} b^{(s)} \end{cases}$$

Le problème d'interface (3.6) est appélé problème d'interface dual puisque le vecteur λ est le dual de la variable primale $x^{(s)}$ et que F_I est le dual du complément de Schur. La matrice de ce problème a été sciemment rendue symétrique afin de faciliter la résolution. Il faut noter qu'il est difficile d'assembler l'opérateur de flexibilité F_I pour des systèmes linéaires de grandes tailles. Cela rend complexe la résolution du problème d'interface par une méthode directe. Cependant, une méthode itérative est un choix efficace puisqu'elle ne nécessite que des produits matrice-vecteur.

3.1.2 Résolution itérative du problème d'interface

Dans la méthode FETI-1, le problème d'interface est résolu par un algorithme de gradient conjugué projeté (GCP). Plus précisément, le problème indéfini (3.6) est transformé, sous la contrainte ${G_I}^T \lambda = e$, en un problème de minimisation de la fonctionnelle :

$$\Phi(\lambda) = \frac{1}{2} \lambda^T F_I \lambda - \lambda^T (G_I \alpha + d). \tag{3.7}$$

La matrice F_I étant semi-définie ou définie positive, ce problème peut être résolu avec un algorithme de GCP. A chaque itération, il faut s'assurer que la valeur de λ satisfait la contrainte citée. C'est le cas en définissant par la formule (3.8), la valeur de début du processus itératif $\lambda = \lambda^0$ où Q est une matrice donnée.

$$\lambda^0 = Q G_I ({G_I}^T Q G_I)^{-1} e \tag{3.8}$$

Pour s'assurer que cette contrainte reste vérifiée au cours du processus, un opérateur $P(Q)$ de projection orthogonale sur le noyau de ${G_I}^T$ est utilisé pour éliminer des itérés $\lambda = \lambda^k$, les composantes qui ne satisfont pas la contrainte ${G_I}^T \lambda = e$. En d'autres termes, à chacune des itérations k, λ^k est mis à jour en adjoignant à λ^0 une composante $P(Q)\bar{\lambda}$ vérifiant ${G_I}^T P(Q) \bar{\lambda} = 0$.

$$\lambda^k = \lambda^0 + P(Q) \bar{\lambda} \tag{3.9}$$

Le projecteur $P(Q)$ est défini algébriquement par :

$$P(Q) = I - Q G_I ({G_I}^T Q G_I)^{-1} {G_I}^T \tag{3.10}$$

Il faut noter que l'opérateur de projection P vérifie les propriétés $P^2(Q) = P(Q)$ et ${G_I}^T P(Q) = 0$ pour toute matrice Q donnée, et sa construction requiert seulement la factorisation de la matrice carrée ${G_I}^T Q G_I$ qui est supposée être inversible. Cette matrice représente les mouvements de corps rigide de tous les sous-domaines flottants ; sa structure ne varie donc pas au cour du processus itératif. La matrice ${G_I}^T Q G_I$ peut alors s'interpréter comme un moyen de transmettre des informations globales à l'ensemble des sous-domaines par une **grille grossière**. Si cette technique peut avoir des points communs avec les méthodes multigrilles, contrairement à ces dernières, aucun maillage grossier n'est construit. La matrice ${G_I}^T Q G_I$ est symétrique et devient plus économique à gérer dès que Q est choisi symétrique. Sa dimension est au plus égale à $\mathbf{3xN_f}$ en dimension deux ou bien $\mathbf{6xN_f}$ en dimension trois. En effet, de manière générale, trois ou six mouvements de corps rigide au maximum apparaissent au sein de chaque sous-domaine selon sa dimension en espace.

Le choix de la matrice Q

On prend bien souvent Q égale à la matrice identité I pour les problèmes homogènes. La plupart des calculs effectués avec FETI et rapportés dans la littérature ont été effectués avec ce choix trivial. Cependant, pour les structures hétérogènes, il faut préconiser une matrice Q qui est physiquement homogène à une rigidité généralisée (Farhat et Roux 1994). Pour ces raisons, deux choix alternatifs ont été proposés pour Q :

1. **préconditionneur "lumped"** : $Q = Q^L = \bar{F}_I^{L^{-1}}$,
2. **préconditionneur de Dirichlet** : $Q = Q^D = \bar{F}_I^{D^{-1}}$.

Dans la suite, l'opérateur $P(Q)$ sera simplement noté par P pour des questions de clarté.

Le préconditionneur "lumped"

Il est prénommé ainsi parce que, mécaniquement, il correspond à la recherche des forces d'interaction grossières (*lumped*) permettant de reproduire des déplacements aux interfaces $\Gamma^{(s)}$ quand seuls les degrés de liberté appartenant à ces $\Gamma^{(s)}$ sont libres. En considérant que chaque matrice $M^{(s)}$ est structurée par blocs (c'est-à-dire que les degrés de liberté internes à $\Omega^{(s)}$, indexés par i, sont numérotés en premiers et sont suivis par ceux appartenant à l'interface $\Gamma^{(s)}$, indexés par f), alors le préconditionneur *lumped* s'écrit :

$$\bar{F}_I^{L^{-1}} = \sum_{s=1}^{N_s} R^{(s)} \begin{bmatrix} 0 & 0 \\ 0 & M_{ff}^{(s)} \end{bmatrix} {R^{(s)}}^T. \tag{3.11}$$

Quand la méthode FETI est équipée du préconditionneur *lumped*, un comportement de son conditionnement κ est donné par la formule ci-dessous où h est la taille des mailles de Ω et H est la taille des sous-structures. Il faut noter que pour un maillage donné, l'augmentation du nombre de sous-structures N_s entraîne une diminution de leurs tailles H.

$$\kappa = O(\frac{H}{h}) \tag{3.12}$$

Du point de vue mathématique, ce préconditionneur localisé n'est donc pas optimal. Cependant, il est économique puisqu'il ne requiert aucun stockage supplémentaire et n'a recours qu'à des produits matrice-vecteur de dimension égale au nombre de degrés de liberté appartenant à chaque $\Gamma^{(s)}$. Il est donc facilement parallélisable aussi bien sur les machines à mémoire partagée que sur les machines à mémoire distribuée.

Le préconditionneur de Dirichlet

Ce préconditionneur est basé sur une interprétation mécanique de la méthode FETI. A chaque itération, elle consiste à prescrire des forces d'interaction à l'interface de chacun des sous-domaines $\Omega^{(s)}$ et en déduire le saut de déplacement d'interface résultant (par le problème de Neumann). Donc, le problème inverse est construit en imposant un saut de déplacement à l'interface $\Gamma^{(s)}$ et en calculant les forces associées. Ceci conduit à un préconditionneur $\bar{F}_I^{D^{-1}}$ de la forme :

$$\bar{F}_I^{D^{-1}} = \sum_{s=1}^{N_s} R^{(s)} \begin{bmatrix} 0 & 0 \\ 0 & M_{ff}^{(s)} - M_{if}^{(s)^T} M_{ii}^{(s)^{-1}} M_{if}^{(s)} \end{bmatrix} R^{(s)^T}. \tag{3.13}$$

Lorsque la méthode FETI, appliquée aux problèmes de mécanique des structures, est équipée de ce préconditionneur, une estimation de son conditionnement κ ((3.14)) montre alors qu'il est borné.

$$\kappa = O(1 + log^2(\frac{H}{h})) \tag{3.14}$$

Le préconditionneur $\bar{F}_I^{D^{-1}}$ est donc considéré comme optimal. Cependant, sa mise en œuvre nécessite plus de mémoire et de temps pour calculer localement les compléments de Schur. A chaque application de ce préconditionneur, des descentes-remontées sont effectuées. La complexité de calcul est ainsi doublée. Pour ces raisons, le préconditionneur *lumped*, plus léger, lui est souvent préféré.

Le problème projeté

L'application de l'expression (3.9) de λ au problème indéfini (3.6) et une multiplication par P^T conduit au problème d'interface projeté :

$$(P^T F_I P)\bar{\lambda} = P^T(d - F_I \lambda^0). \tag{3.15}$$

La version initiale de FETI se ramène donc à la résolution du problème d'interface semi-défini symétrique (3.15). La solution est alors obtenue en appliquant l'algorithme de GCP à la matrice F_I, avec d comme second membre et λ comme inconnue.

3.1.3 Algorithme détaillé de FETI-1

En récapitulant les éléments des sous-sections précédentes, l'algorithme de FETI-1 s'écrit donc autour d'un gradient conjugué préconditionné du problème d'interface projeté (3.15) où $\bar{F}_I^{-1}$ désigne un préconditionneur

qui est une approximation de l'inverse de F_I. En prenant Q comme étant égale à I, l'algorithme simplifié s'effectue en deux étapes.

1. Une étape d'**initialisation** consistant à :

   – trouver les forces d'interaction initiales λ^0 :

   $$\lambda^0 = G_I(G_I{}^T G_I)^{-1} e \tag{3.16}$$

   – chercher la solution particulière du problème local de Neumann dans chacun des sous-domaines $\Omega^{(s)}$:

   $$x^{(s)0+} = M^{(s)+}(b^{(s)} - R^{(s)T}\lambda^0); \tag{3.17}$$

   – calculer le saut des solutions particulières initiales le long des interfaces entre sous-domaines, c'est-à-dire le gradient initial :

   $$g^0 = d - F_I\lambda^0 = \sum_{s=1}^{N_s} R^{(s)} x^{(s)0+}; \tag{3.18}$$

   – projeter le gradient initial g^0 sur le noyau de $G_I{}^T$:

   $$Pg^0 = g^0 + G_I\alpha^0 \text{ où } (G_I{}^T G_I)\alpha^0 = -G_I{}^T g^0; \tag{3.19}$$

   – préconditionner le gradient initial projeté Pg^0 :

   $$w^0 = \bar{F}_I^{-1} Pg^0; \tag{3.20}$$

   – calculer la direction de descente initiale p^0 :

   $$p^0 = Pw^0 = w^0 + G_I\tilde{w}^0 \text{ où } (G_I{}^T G_I)\bar{w}^0 = -G_I{}^T w^0. \tag{3.21}$$

2. Une étape de **réactualisation** qui correspond à une phase itérative du gradient conjugué. Connaissant les valeurs de λ^k, $x^{(s)k+}$, g^k et Pw^k à une itération $k \geq 0$ du gradient conjugué, à l'itération $k+1$, cette étape consiste à :
   – calculer les solutions locales des **problèmes de Neumann** :

   $$\Delta x^{(s)k+} = M^{(s)+} R^{(s)T} Pw^k; \tag{3.22}$$

   – faire le produit du complément de Schur dual F_I par Pw^k :

   $$F_I Pw^k = \sum_{s=1}^{N_s} R^{(s)} x^{(s)k+}; \tag{3.23}$$

   – calculer le nouvel itéré λ^{k+1} et le nouveau gradient g^{k+1} :

   $$\begin{cases} \lambda^{k+1} = \lambda^k + \rho^k Pw^k \\ g^{k+1} = g^k + \rho^k F_I Pw^k \\ \text{avec } \rho^k = -(g^k, Pw^k)/(F_I Pw^k, Pw^k) \end{cases}; \tag{3.24}$$

   – projeter le gradient g^{k+1} :

   $$Pg^{k+1} = g^{k+1} + G_I\alpha^{k+1} \tag{3.25}$$

   où α^{k+1} est obtenu en résolvant :

   $$(G_I{}^T G_I)\alpha^{k+1} = -G_I{}^T g^{k+1}; \tag{3.26}$$

– préconditionner le gradient projeté en passant par la résolution des problèmes :

$$z^{k+1} = \bar{F}_I^{-1} P g^{k+1}; \tag{3.27}$$

– projeter le gradient projeté et préconditionné :

$$Pz^{k+1} = z^{k+1} + G\bar{z}^{k+1} \text{ où } ({G_I}^T G_I)\bar{z}^{k+1} = -{G_I}^T z^{k+1}; \tag{3.28}$$

– calculer la nouvelle direction de descente Pw^{k+1} par conjugaison :

$$\begin{cases} Pw^{k+1} = Pz^{k+1} + \gamma^k Pw^k \\ \text{avec } \gamma^k = -(Pz^{k+1}, F_I Pw^k)/(F_I Pw^k, Pw^k) \end{cases}; \tag{3.29}$$

– déterminer les solutions locales particulières :

$${x^{(s)}}^{(k+1)+} = {x^{(s)}}^{k+} + \rho^k \Delta {x^{(s)}}^{(k+1)+}; \tag{3.30}$$

– corriger les solutions locales en cas de présence de mouvements de corps de rigide :

$${x^{(s)}}^{k+1} = {x^{(s)}}^{(k+1)+} + B^{(s)} {\alpha^{(s)}}^{k+1}. \tag{3.31}$$

Cette phase itérative de l'algorithme du gradient conjugué projeté et préconditionné (GCPC) est répétée jusqu'à un certain critère de convergence sur le gradient projeté ou le résidu des solutions locales.

La version initiale de la méthode FETI peut donc être considérée comme un solveur d'interface à deux niveaux sur un GCPC. Les problèmes locaux avec des conditions aux limites de Dirichlet sont résolus dans l'étape de préconditionnement, et les problèmes avec des conditions aux limites de Neumann associés aux sous-domaines sont résolus dans une deuxième étape. Cette deuxième étape se poursuit de manière itérative jusqu'à la convergence : équilibre du champ des efforts et annulation du saut de déplacement aux interfaces.

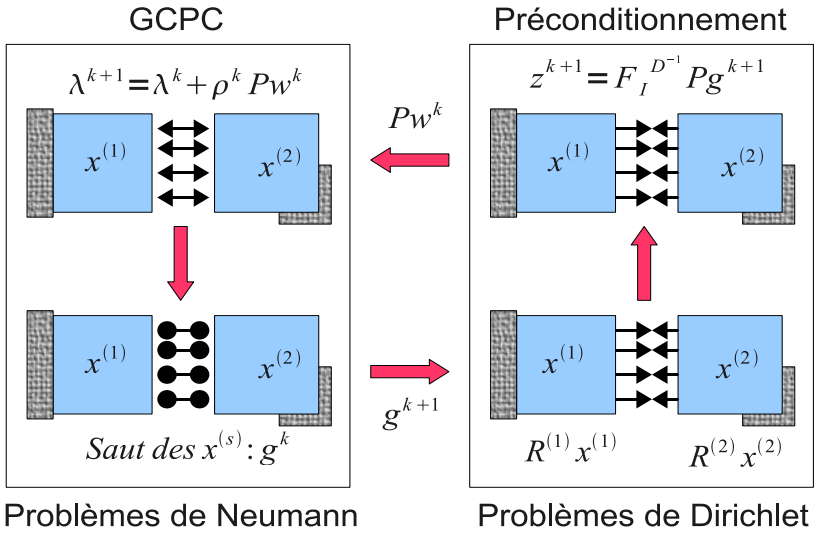


FIG. 3.2 – *FETI-1 : solveur d'interface à deux niveaux sur un GCPC.*

En présence de sous-structures flottantes, les matrices de rigidité locales ne sont plus inversibles et il faut alors prendre en compte, dans la

version initiale de FETI, une contrainte supplémentaire liée à leurs mouvements de corps rigide. Cette contrainte d'admissibilité est traitée en rajoutant un projecteur orthogonal P dans le calcul du saut des solutions locales aux interfaces et leur mise à jour. Chaque étape de projection nécessite la résolution d'un **problème grossier** du type :

$$(G_I{}^T Q G_I)\alpha = G_I{}^T g. \tag{3.32}$$

Le problème grossier (3.32) est très recherché par toutes les méthodes de décomposition de domaine car il permet de faire communiquer globalement des informations d'un bout à l'autre de la structure. Avec FETI-1, il est résolu deux fois par itération. Il propage ainsi l'erreur globalement à tout le domaine, accélérant la convergence du processus. Sa bonne résolution est cruciale pour le bon déroulement du processus itératif.

3.2 Les facteurs critiques d'une méthode FETI

La robustesse et les performances d'une méthode FETI dependent d'un ensemble de questions de mise en œuvre qui sont soulevées dans cette section. Ces questions déterminent les points critiques d'un solveur FETI : le choix d'un solveur local, le traitement des mouvements de corps rigide ainsi que le calcul des pseudo-inverses, et la gestion du problème grossier.

3.2.1 Choix d'un solveur local

Un solveur local est utilisé par les méthodes FETI pour effectuer deux tâches à chacune des itérations k de l'algorithme du GCP :

1. l'évaluation des solutions locales $x^{(s)^k}$ (voir (3.4)) associées aux forces d'interaction approchées λ^k ;
2. le calcul du gradient projeté préconditionné $\tilde{F}_I^{-1} P g^k$ par la résolution de problèmes locaux de Dirichlet quand le préconditionneur $\tilde{F}_I^{D^{-1}}$ est sélectionné (voir (3.13)).

Le choix d'une méthode iterative comme solveur local peut être en théorie attractif, car il réduit les besoins en mémoire des méthodes FETI et ouvre des thèmes de recherche variés dans le domaine des solveurs inexacts. Toutefois, un solveur itératif local accroît la complexité du résultat de robustesse puisqu'il introduit un critère de convergence supplémentaire dans la méthode FETI au niveau des sous-domaines. En outre, le solveur local est intégré dans une boucle itérative pour déterminer les forces d'interaction λ^k, et donc l'efficacité des calculs nécessite une bonne optimisation d'un solveur itératif local traitant des systèmes linéaires à seconds membres successifs. Même si une telle optimisation est possible (par exemple, voir Farhat et al. (1994a)), ce solveur itératif local a besoin d'un stockage supplémentaire qui neutralise son principal avantage.

D'autre part, le choix d'une méthode directe comme solveur local permet de réduire le nombre de questions sur la robustesse qui doivent être traitées lors de la mise en œuvre de FETI, et assure une meilleure efficacité de calcul global de FETI. En effet, puisque les méthodes FETI sont numériquement extensibles au nombre de sous-domaines N_s, les besoins

en mémoire d'un solveur direct local peut être considérablement réduit en découpant plus le domaine initial Ω, car une augmentation de N_s diminue la dimension des matrices de rigidité $M^{(s)}$. De plus, lorsque le nombre N_s est choisi assez grand, la taille des problèmes locaux tend vers une plage dans laquelle un solveur itératif ne peut pas rivaliser en vitesse de calcul avec un solveur direct.

Les solveurs directs sont donc d'excellents candidats pour être utilisés comme solveur local dans les méthodes FETI. Toutefois, certains problèmes locaux qui se posent dans les méthodes FETI sont souvent mal posés. Leur résolution nécessite donc de choisir des solveurs directs qui prennent en compte leurs singularités.

3.2.2 Traitement des singularités et calcul des pseudo-inverses

Le point le plus critique de la méthode FETI est le traitement des modes rigides des sous-structures flottantes et le calcul des pseudo-inverses associés. De leur bonne détermination dépend la validité de l'algorithme, puisqu'ils rentrent à la fois dans la construction de l'opérateur FETI et dans celle de l'opérateur de projection. Si, pour une raison quelconque, le nombre de mouvements de corps rigide d'un sous-domaine $\Omega^{(s)}$ flottant n'est pas très bien calculé ou bien qu'une représentation du noyau de $M^{(s)}$ (les colonnes de $B^{(s)}$), est mal spécifiée, la condition de solvabilité (3.3) sera violée, le projecteur P sera inadéquat (voir la formule (3.9)) et l'algorithme de FETI ne convergera pas vers la solution correcte du problème à résoudre. Des effets similaires se produisent quand un mauvais pseudo-inverse ${M^{(s)}}^+$ associé à la sous-structure flottante est calculé.

Différentes techniques de traitement des systèmes linéaires singuliers existent. Certaines sont mises en œuvre dans les solveurs Frontal, Sparse Direct, MUMPS et dernièrement le solveur Dissection (voir sa mise en œuvre au chapitre 1). Les solveurs Frontal et Sparse Direct étaient jusqu'à présent les deux seuls qui étaient disponibles pour être utilisés comme solveur local dans les méthodes FETI développées dans le code de calcul ZéBuLon. Le solveur Dissection a été mis en œuvre dans ces travaux pour traiter au mieux les modes rigides. La factorisation y est basée sur la procédure décrite dans le chapitre 1 et repose sur le calcul des modes à énergie nulle et de l'inverse généralisée de la matrice de rigidité associée à un sous-domaine flottant. La procédure exploite le fait que, si l_s est le nombre de colonne de $B^{(s)}$ alors l_s pivots proches de zéro apparaîtront au cours de la factorisation de $M^{(s)}$. Chaque pivot proche de zéro rencontré correspond à une équation redondante qui peut être supprimée du système linéaire local sans interrompre le processus de factorisation. A la fin de ce processus, les colonnes et les lignes non retirées correspondent à la forme factorisée la plus simple du pseudo-inverse ${M^{(s)}}^+$ que l'on peut construire et, une base du noyau de $M^{(s)}$ peut être récupérée en effectuant l_s descentes-remontées sur les l_s vecteurs associés aux l_s pivots proches de zéro rencontrés.

L'hypothèse que les mouvements de corps rigide d'un sous-domaine flottant calculés par la procédure résumée ci-dessus peuvent être suffisamment entachés d'erreurs par les arrondis accumulés au cours de la factorisation des matrices de rigidité locales et altérer la convergence de la

méthode FETI ou l'empêcher de produire une solution exacte du problème global, est souvent émise. Ceci est contredit par nos observations. En effet, si une matrice de rigidité locale $M^{(s)}$ est suffisamment mal conditionnée pour éviter que l_s descentes-remontées soient exactes, alors la matrice de rigidité globale M doit être encore plus mal conditionnée et devrait interdire de toutes façons une solution exacte du problème global. En fait, nous avons effectué un grand nombre d'expériences sur des problèmes d'élasticité linéaire et avons vérifié que les méthodes FETI équipées de la procédure décrite ci-dessus pour l'extraction des colonnes de $B^{(s)}$ et le calcul de ${M^{(s)}}^+$ peuvent toujours donner une solution exacte du problème global et être plus efficaces avec le solveur Dissection qu'avec les techniques de traitement des matrices singulières élaborées dans les solveurs Frontal et Sparse Direct.

3.2.3 Gestion du problème grossier

L'idée de la mise en œuvre d'un problème grossier, par rapport à celui qui est résolu sur la grille fine issue de la discrétisation par la MEF, est bien évidemment de ne faire intervenir qu'un petit nombre d'inconnues par sous-domaine pour coupler ces derniers. Il existe différents moyens d'exhiber des problèmes grossiers, par exemple en résolvant un problème où interviennent seulement les points multiples (Farhat et al. 1998), (Farhat et Mandel 1998), ou bien en vérifiant en moyenne une condition de continuité (Farhat et Roux 1991). Comme nous l'avons indiqué dans la section 3.1, la méthode FETI-1 incorpore naturellement une telle grille grossière induite par la prise en compte des sous-structures flottantes. Ce problème grossier va permettre de rendre la méthode numériquement extensible (*scalable* en anglais). Le terme *scalable* traduit l'adaptabilité du code aux problèmes traités (certains ont souvent plusieurs dizaines de sous-structures) sans dégradation notoire des performances. Toutefois, cette extensibilité numérique doit aller de pair avec le parallélisme. Or, la première peut nuire au second, voire être incompatible. En effet, la facilité de construction et de résolution du problème grossier (3.32), qui couple des informations provenant des sous-domaines flottants, peut être la clé de cette incompatibilité possible.

Deux possibilités peuvent être envisagées pour traiter le problème grossier. La première consiste à assembler explicitement la matrice ${G_I}^T Q G_I$ et la factoriser. La seconde possibilité consiste à résoudre ce problème à l'aide d'une méthode itérative pour laquelle seuls des produits matrice-vecteur sont nécessaires. Ces opérations peuvent se faire sans la construction explicite de la matrice ${G_I}^T Q G_I$. C'est le cas par exemple des méthodes FETI développées dans le code de calcul ZéBuLon. Une méthode directe a été choisie pour résoudre leur problème grossier. Son application est plus complexe que celle de la résolution par un algorithme itératif. Toutefois, une telle stratégie ne peut que renforcer la robustesse et la rapidité de ces méthodes FETI. En outre, lorsque le nombre de sous-domaines est très grand ($\mathbf{N_s > 100}$), il a été démontrée dans (Farhat et al. 1998) et (Farhat et al. 1995) que la résolution du problème grossier (3.32) par une méthode directe peut être plus efficace que sa résolution par un algorithme itératif.

L'assemblage explicite de la matrice ${G_I}^T Q G_I$ du problème grossier suppose quand même un nombre important d'échanges entre les processeurs qui traitent les sous-structures flottantes, mais surtout un surcoût en mémoire non négligeable. En effet, sa dimension est au moins égale à $\mathbf{3}$x$\mathbf{N_f}$ en $2D$ et $\mathbf{6}$x$\mathbf{N_f}$ en $3D$. De plus, sa topologie est creuse parce qu'elle rend compte des interactions qui se produisent entre les sous-domaines flottants. Ainsi, des termes nuls sont générés dans la matrice ${G_I}^T Q G_I$ dès que deux sous-domaines flottants ne sont pas connectés.

3.3 L'intégration d'un parallélisme à deux niveaux dans FETI

Jusque là, nous avons parcouru les principes de base de la méthode FETI-1 tout en privilégiant l'aspect matriciel. Il faut se reporter à la thèse de Roux (1989) pour des considérations plus mathématiques relatives à cette méthode. Maintenant, notre objectif est d'intégrer un parallélisme à deux niveaux dans cette méthode afin d'accélérer la résolution de son problème d'interface et ainsi améliorer ses performances. Pour cela, nous portons le choix du solveur local sur le solveur direct parallèle, mis en œuvre aux chapitres 1 et 2. Nous couplons donc la méthode FETI-1, implantée dans le code de calcul ZéBuLon et parallélisée seulement avec les bibliothèques MPI et PVM, et la version multi-threads de ce solveur direct avec des threads POSIX et des sous-programmes de BLAS optmisées en OpenMP. La stratégie de couplage consiste à effectuer, en parallèle avec le solveur direct, la construction si nécessaire d'une base du noyau de chacune des matrices de rigidité locales $M^{(s)}$, et le calcul des pseudo-inverses ${M^{(s)}}^+$ et des pseudo-solutions ${x^{(s)}}^+$. Cette stratégie peut s'avérer très précieuse et complémentaire du parallélisme à deux niveaux de la méthode classique de FETI.

3.3.1 Extraction en parallèle des colonnes de $B^{(s)}$

Les sous-structures $\Omega^{(s)}$ générées par le partitionnement d'un domaine initial Ω peuvent être totalement encastrées, partiellement encastrées ou libres. Dans le premier cas, les matrices de rigidité locales $M^{(s)}$ sont symétriques définies positives. Dans les deux derniers cas, les sous-structures sont flottantes et les matrices de rigidité associées ne sont pas inversibles. Pour chacune de ces sous-structures flottantes, une base du noyau $Ker(M^{(s)})$ formée par les colonnes de $B^{(s)}$, est donnée par les modes à énergie nulle de $M^{(s)}$.

Dans le code de calcul ZéBuLon, nous effectuons le calcul de ces modes à énergie nulle en deux temps. Nous partons d'abord des matrices S_s et $\widetilde{N}$ calculées à la sous-section 1.5.2 et dont les tailles sont respectivement $\mathbf{nsing^2}$ et $\mathbf{n}$x$\mathbf{nsing}$ où $\mathbf{nsing}$ est égal au nombre total de singularités locales détectées durant la factorisation de $M^{(s)}$ et $\mathbf{n}$ est la dimension associée. Puis, nous appliquons l'algorithme 3.1 pour vérifier le noyau de projection avec une dimension $\mathbf{n_1}$ croissante. Par la suite, nous en déduisons, pour chaque matrice singulière $M^{(s)}$, son rang $\mathbf{n_1}$ et la vraie dimension $\mathbf{n_{0\_opt}}$ de son noyau.

Algorithme 3.1 : Vérification du noyau de projection $\widetilde{P} = \widetilde{N}^T\widetilde{N}$

Initialisation : $\mathbf{n_{0\_opt}} = 0$, **eps_piv** et $\mathbf{r_{max\_opt}}$ donnés;
pour $n_{00} = nsing$ *à* 0 **faire**
  $\mathbf{n_0}$, le nombre de vecteurs à conserver dans le noyau;
  $\mathbf{n_0} = n_{00}$;
  $\mathbf{n_1} = nsing - n_0$;
  Effectuer une factorisation par élimination de Gauss avec pivotage complet sur le sous-bloc matriciel $\mathbf{S_s}(\mathbf{1} : \mathbf{n_1}, \mathbf{1} : \mathbf{n_1})$;
  Tester si un pivot nul est rencontré lors de l'étape précédente :
  pivot, le dernier pivot non nul rencontré;
  si $n_0 > 0$ **alors**
    $iker = nsing - n_0$;
    $dd = S_s(iker, iker)$;
    pour $iker = nsing - n_0 + 1$ *à* $nsing$ **faire**
      $dd = max(dd, S_s(iker, iker))$;
    fin
    si $dd >$ **eps_piv** $*$ *pivot* **alors**
      passer à l'itération $n_{00} = n_{00} - 1$;
    fin
  fin
  Examiner l'énergie correspondant à chaque vecteur v conservé dans le noyau :
  $n_1 = nsing - n_0$;
  $r_{max} = 0$;
  pour $jker = 1$ *à* n_0 **faire**
    $S_s v = S_s(n_1 + 1 : nsing, n_1 + jker)$;
    $r = ||S_s v||/||S_s(1 : n_1, 1 : n_1)||$ x $||v||$;
    $r_{max} = max(r_{max}, r)$;
  fin
  Vérifier la solution x de $S_s x = b$, avec $b = M^{(s)}$x$\mathbb{1}$:
  si $n_1 < nsing$ *et b projeté dans l'orthogonal du noyau* **alors**
    calculer le résidu condensé : $r = ||\widetilde{N}^T b||/||b||$;
    $r_{max} = max(r_{max}, r)$;
    calculer le résidu condensé approché : $r = ||Sx - \widetilde{P}b||/||\widetilde{P}b||$;
    $r_{max} = max(r_{max}, r)$;
    calculer le résidu exact : $r = ||M^{(s)}x - \widetilde{P}b||/||\widetilde{P}b||$;
    $r_{max} = max(r_{max}, r)$;
  sinon
    calculer le résidu condensé approché : $r = ||Sx - b||/||b||$;
    $r_{max} = max(r_{max}, r)$;
    calculer le résidu exact : $r = ||M^{(s)}x - b||/||b||$;
    $r_{max} = max(r_{max}, r)$;
  fin
  si $r_{max} < \mathbf{r_{max\_opt}}$ **alors**
    $\mathbf{r_{max\_opt}} = r_{max}$;
    la dimension du noyau de $S_s(1 : n_1, 1 : n_1)$ est :
    $\mathbf{n_{0\_opt}} = n_0$.
  fin
fin

A chaque itération n_{00} de l'algorithme de vérification, nous effectuons d'abord une élimination de Gauss avec pivotage complet sur le sous-bloc $S_s(1:n_1, 1:n_1)$, de dimension n_1 variable, pour mieux gérer les perturbations numériques. Nous testons ensuite si des pivots nuls sont rencontrés lors de cette factorisation. Si les termes diagonaux de $S_s(1:n_1, 1:n_1)$ sont assez grands, c'est-à-dire que le plus grand terme diagonal est supérieur à $pivot$x**eps_piv** où **eps_piv** est un paramètre absolu fixé à $\mathbf{10^{-2}}$, alors nous passons à l'itération suivante ($n_{00} - 1$). Dans le cas contraire, nous examinons alors dans un premier temps, l'énergie de chaque vecteur conservé dans le noyau du bloc $S_s(1:n_1, 1:n_1)$ et nous vérifions la solution du système $S_s x = b$ où b est le vecteur $M^{(s)}$x$\mathbb{1}$ projeté ou non dans le sous-espace orthogonal du noyau. Dans un deuxième temps, nous comparons le résidu maximal r_{max} par rapport à une valeur relative $\mathbf{r_{max\_opt}}$, égale par défaut à $\mathbf{10^{16}}$. Si r_{max} est plus petit que $\mathbf{r_{max\_opt}}$ alors la dimension du noyau de $S_s(1:n_1, 1:n_1)$ est bien $n_{0\_opt} = n_0$ et $\mathbf{r_{max\_opt}}$ est réinitialisée à r_{max}. Il faut noter que, pour calculer le résidu exact permettant de vérifier la solution x de $S_s x = b$, nous effectuons des descentes-remontées avec la version multi-threads du solveur local.

A la fin de la vérification du noyau de projection, deux situations se présentent : $n_{0\_opt} = nsing$ ou $n_s = nsing - n_{0\_opt} > 0$. Dans l'un ou l'autre de ces deux cas, nous calculons une base du noyau de $M^{(s)}$ de dimension n_{0_opt} et nous extrayons la matrice de projection P sur l'orthogonal de cet espace. Dans le premier cas, une base du noyau de $M^{(s)}$ est $N = \widetilde{N}$ et la matrice P factorisée est $P = \widetilde{P}$. Dans le deuxième cas, nous calculons N à partir d'une correction de la matrice $\widetilde{N}$. Cette correction consiste à :

- restaurer les n_s colonnes de la matrice $\widetilde{N}$ qui n'appartiennent pas au noyau de $M^{(s)}$;
- construire par correction une base de $Ker(M^{(s)})$ à partir des n_{0_opt} colonnes qui sont de nouveau dans le noyau.

D'autre part, nous extrayons la matrice de projection P à partir de la matrice $\widetilde{P}$ factorisée. Cette extraction consiste à prendre P telle que :

$$P(i,j) = \widetilde{P}(n_s + i, n_s + j), \quad 1 \leq i, j \leq n_{0\_opt}. \tag{3.33}$$

Quand la matrice $M^{(s)}$ est mise à l'échelle avec Δ, le noyau N construit est plutôt celui de $\Delta M^{(s)} \Delta$. Pour avoir le noyau de $M^{(s)}$, nous devons calculer N sous la forme ΔN. Pour la même raison, nous corrigeons la matrice de projection P à partir de la formule $P = (N^T \Delta^T)(\Delta N)$.

3.3.2 Calcul en parallèle des pseudo-inverses et pseudo-solutions

Les pseudo-inverses ${M^{(s)}}^+$ et les pseudo-solutions ${x^{(s)}}^+$ sont donnés par la version multi-threads du solveur local comme étant respectivement les matrices $\widetilde{M}^{-1}$ et $\widetilde{M}^{-1} b$. La matrice $\widetilde{M}$ est celle définie à la sous-section 1.5.2 et b est le second membre, où les composantes correspondant aux indices globales des modes rigides sont nulles. Nous supposons que le système local $M^{(s)} x^{(s)} = b^{(s)}$ peut s'écrire comme un système linéaire par blocs :

$$\begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix} \begin{pmatrix} x_1^{(s)} \\ x_2^{(s)} \end{pmatrix} = \begin{pmatrix} b_1^{(s)} \\ b_2^{(s)} \end{pmatrix} \tag{3.34}$$

où M_{11} est supposé être le plus grand bloc matriciel inversible de $M^{(s)}$ qui est donné par le solveur local et qui a pour dimension $n - nsing$. Les expressions de la matrice $\widetilde{M}^{-1}$ et de la pseudo-solution ${x^{(s)}}^+$ sont alors définies à partir des équations de (3.35).

$$\widetilde{M}^{-1} = \begin{pmatrix} M_{11}^{-1} & 0 \\ 0 & I \end{pmatrix} \text{ et } {x^{(s)}}^+ = \begin{pmatrix} M_{11}^{-1} & 0 \\ 0 & I \end{pmatrix} \begin{pmatrix} b_1^{(s)} \\ 0 \end{pmatrix} \tag{3.35}$$

La pseudo-solution obtenue est projetée dans le sous-espace orthogonal du noyau avec l'aide de la matrice de projecteur P. Cela conduit à :

$${x^{(s)}}^+ = {x^{(s)}}^+ - NP^{-1}N^T{x^{(s)}}^+. \tag{3.36}$$

Avec la dimension optimale n_{0_opt} du noyau, trouvée à la sous-section précédente, nous pouvons rencontrer le cas où $n_s = nsing - n_{0\_opt} > 0$. Dans ce cas, nous effectuons, avant l'étape de projection, une correction de la pseudo-solution ${x^{(s)}}^+$ avec les n_s colonnes qui ne font plus partie de la base de $Ker(M^{(s)})$. Pour ce faire, nous commençons par décomposer d'une part, les sous-blocs M_{22}, M_{12} et M_{21} et d'autre part, les vecteurs $x_2^{(s)}$ et $b_2^{(s)}$ sous les formes :

$$M_{22} = \begin{pmatrix} \underline{M}_{22} & \underline{M}_{23} \\ \underline{M}_{32} & \underline{M}_{33} \end{pmatrix}, \; M_{12} = \begin{pmatrix} \underline{M}_{12} \; \underline{M}_{13} \end{pmatrix} \text{ et } M_{21} = \begin{pmatrix} \underline{M}_{21} \\ \underline{M}_{31} \end{pmatrix} \tag{3.37}$$

et

$$x_2^{(s)} = \begin{pmatrix} \underline{x}_2^{(s)} \\ \underline{x}_3^{(s)} \end{pmatrix} \text{ et } b_2^{(s)} = \begin{pmatrix} \underline{b}_2^{(s)} \\ \underline{b}_3^{(s)} \end{pmatrix} \tag{3.38}$$

où les blocs diagonaux $\underline{M}_{22}$ et $\underline{M}_{33}$ ont pour dimension respectivement n_s et n_{0_opt}. Le système linéaire (3.34) devient alors :

$$\begin{cases} x_1^{(s)} = M_{11}^{-1}b_1^{(s)} - M_{11}^{-1}\underline{M}_{12}\underline{x}_2^{(s)} - M_{11}^{-1}\underline{M}_{13}\underline{x}_3^{(s)} \\ \\ Sx_2^{(s)} = \begin{pmatrix} S_{22} & S_{23} \\ S_{32} & S_{33} \end{pmatrix} \begin{pmatrix} \underline{x}_2^{(s)} \\ \underline{x}_3^{(s)} \end{pmatrix} = \begin{pmatrix} \underline{b}_2^{(s)} - \underline{M}_{21}M_{11}^{-1}b_1^{(s)} \\ \underline{b}_3^{(s)} - \underline{M}_{31}M_{11}^{-1}b_1^{(s)} \end{pmatrix} \\ \\ \text{avec } \; S_{ij} = \underline{M}_{ij} - \underline{M}_{i1}M_{11}^{-1}\underline{M}_{1j} \text{ pour } i,j = 1,2 \end{cases} \tag{3.39}$$

$$Scol = \begin{pmatrix} -M_{11}^{-1}\underline{M}_{12} \\ I \\ 0 \end{pmatrix}. \tag{3.40}$$

Une fois que ces décompostions sont faites, nous mettons à jour la pseudo-solution avec l'aide de la matrice $Scol$. Cette matrice est composée des n_s colonnes non conservées dans $\widetilde{N}$ et est donnée par la formule 3.40. Dans le cas symétrique (actuellement), cette mise à jour de ${x^{(s)}}^+$ s'effectue en trois étapes :

1. l'assemblage de $\underline{b}_2^{(s)} - \underline{M}_{21}M_{11}^{-1}b_1^{(s)} = Scol^T \text{x} b$;
2. le calcul de la contribution $\underline{x}_2^{(s)} = {S_{22}}^{-1} Scol^T \text{x} b$;

3. la correction de $x^{(s)^+} = x^{(s)^+} + Scol\underline{x}_2^{(s)} - \begin{pmatrix} 0 \\ \underline{x}_2^{(s)} \\ 0 \end{pmatrix}$.

3.4 L'évaluation des performances de la méthode FETI

Une stratégie d'amélioration de la méthode FETI a été proposée et déployée dans le code de calcul ZéBuLon. Après sa validation et sa stabilisation dans ce code, l'objectif de cette section est d'en mesurer les performances et d'en faire une analyse. Pour ce faire, nous présentons d'abord l'environnement d'évaluation sur laquelle nous effectuons tous nos simulations. Nous décrivons ensuite les cas tests réalisés dans cet environnement. Enfin, nous analysons les performances obtenues grâce à des critères d'efficacité et d'extensibilité.

3.4.1 Environnement d'évaluation

L'environnement d'évaluation est constitué de deux éléments principaux : les architectures parallèles sur lesquelles nous réalisons les tests et le code de calcul ZéBuLon que nous utilisons pour mesurer les performances de la méthode FETI.

La présentation des calculateurs parallèles utilisés

Tous les tests parallèles présentés dans ce chapitre ont été réalisés sur le cluster ou grappe de calcul du Centre des Matériaux (MINES ParisTech) et le supercalculateur JADE (cmp-200903446) du Centre Informatique National de l'Enseignement Supérieur (CINES).

Le cluster de calcul du Centre des Matériaux (figure 3.3) est constitué de 112 nœuds de calcul (232 cœurs). Chaque nœud est un bi-processeur AMD Opteron 248 en 64-bit cadencé à 2.2 GHz et pouvant possèder de 8 à 16 Go de mémoire vive. Au total, le cluster dispose de 1008 Go de RAM et d'une puissance théorique de 0.9 TFLOPS. L'ensemble des nœuds est relié par un réseau de type Ethernet Gigabit. Le cluster dispose d'une couche de calcul distribué (MPI).

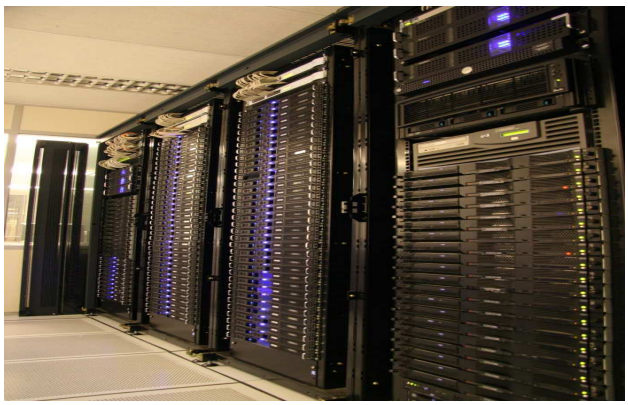


Fig. 3.3 – *Cluster du CdM (source : http ://www.mat.ensmp.fr).*

Le cluster SGI Altix ICE 8200 du CINES, baptisé JADE, est le plus puissant supercalculateur scalaire parallèle à l'échelle nationale (deuxième en

Europe). Il dispose d'une puissance de calcul maximale de 147 TFLOPS (figure 3.4). JADE comprend 25 racks dont un assurant la connexion à l'ensemble du cluster (*via* 3 nœuds de connexion) et 24 racks de calcul. Comme le décrit la figure 3.5, un rack comprend un nœud dit leader et 4 IRU. Chaque IRU est composé de 16 lames ou nœuds de calcul. Ces lames sont composées chacune de 2 processeurs Intel Quad-Core E5472. Les processeurs des nœuds de connexion sont des processeurs Intel Quad-Core X5472. Chacun de ces processeurs possède 4 cœurs et a une fréquence de 3.00 GHz. Par conséquent, leur puissance nominale est de 12 GFLOPS. Les cœurs disposent chacun de 4 Go de mémoire vive et d'une mémoire cache L1 de 32 Ko pour les données et de 32 Ko pour les instructions. Il existe deux caches L2 de 6 Mo par processeur. Chacun de ces deux caches est partagé par deux cœurs (figure 3.5). Au total, JADE est composé de 1536 nœuds de calcul et comprend donc 12288 cœurs et 46 To de mémoire vive. Concernant le calcul, le réseau connectant les racks entre eux, est un réseau InfiniBand (DDR) qui utilise un bus bi-directionnel à faible coût. Un "chemin" est utilisé pour les communications MPI et l'autre sert pour la gestion des entrées/sorties sur un système de fichiers distribué libre.

FIG. 3.4 – *Cluster JADE du CINES (source : http ://www.cines.fr).*

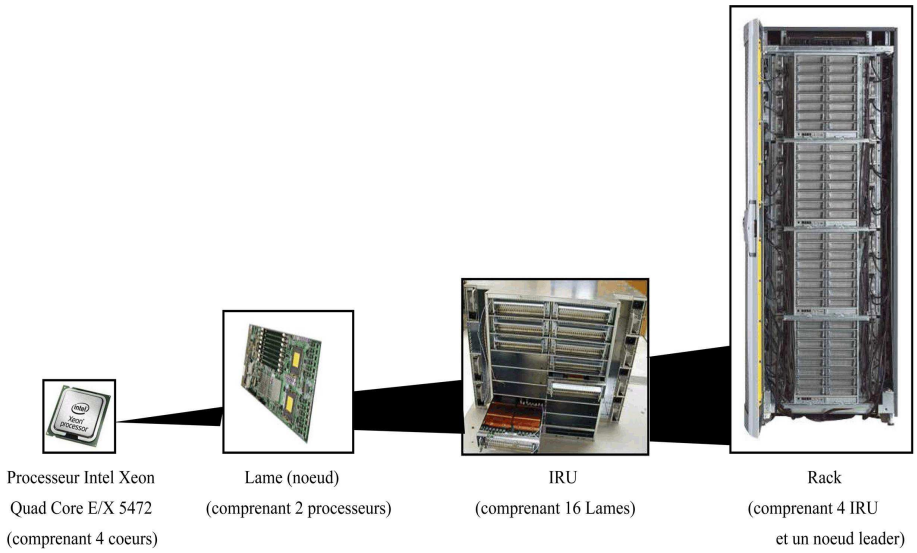


FIG. 3.5 – *Architecture de JADE (source : http ://www.cines.fr).*

Le code de calcul ZéBuLon parallèle

Les tests parallèles sont réalisés à partir du code ZéBuLon qui est un code de calcul de structures par éléments finis spécialisé dans les problèmes de mécanique non linéaire. Ce code est développé en étroite collaboration entre l'ONERA, MINES ParisTech et la société NW Numerics (USA). ZéBuLon est donc un code de recherche fortement évolutif et très proche des développeurs (chercheurs / ingénieurs). Le code de calcul ZéBuLon a été parallélisé avec la méthode FETI (Feyel 1998; Gosselet et Rey 2006) et a été installé, puis testé sur des architectures parallèles à mémoire distribuée : le cluster du Centre des Matériaux et le supercalculateur JADE du CINES.

3.4.2 Critères de mesure de performances

En résolvant un problème de taille $\mathbf{n}$ avec le code ZéBuLon parallèle sur $\mathbf{p}$ processeurs en un temps donné, deux cas de figures complémentaires se présentent. Nous souhaitons d'une part, réduire ce temps *elapsed* en augmentant $\mathbf{p}$ et d'autre part, nous passons à l'échelle sur un grand nombre de processeurs, c'est-à-dire augmenter $\mathbf{n}$ linéairement en fonction de $\mathbf{p}$. Il existe des critères pour chacun de ces deux cas pour mesurer les performances de la méthode FETI.

Dans le premier cas de figure, nous espérons une diminution quasi-linéaire du temps et nous parlons alors d'extensibilité forte (*strong scalability* en anglais). Elle se mesure par un critère appelé **accélération** ou ***speed-up***. L'accélération due à la parallélisation $\mathbf{S_p(n)}$ est définie par l'équation (3.41) comme étant le rapport entre le temps séquentiel $\mathbf{T_1(n)}$ et le temps d'exécution sur $\mathbf{p}$ processeurs $\mathbf{T_p(n)}$. L'accélération linéaire sur $\mathbf{p}$ processeurs vaut évidemment $\mathbf{p}$.

$$\mathbf{S_p(n)} = \frac{\mathbf{T_1(n)}}{\mathbf{T_p(n)}} \tag{3.41}$$

A ce critère, on préfère parfois la notion d'**efficacité** parallèle $\mathbf{E_p(n)}$ qui est le rapport entre l'accélération effective et l'accélération optimale.

$$\mathbf{E_p(n)} = \frac{\mathbf{S_p(n)}}{\mathbf{p}} \tag{3.42}$$

Dans le deuxième cas de figure, nous parlons d'**extensibilité** faible (*weak scalability* en anglais). Le critère utilisé est le ***scale-up***. Il est donné par l'équation (3.43) où $\mathbf{T_p(pn)}$ est le temps nécessaire pour traiter le problème de taille $\mathbf{p}$x$\mathbf{n}$ sur $\mathbf{p}$ processeurs.

$$\mathbf{A_p(n)} = \frac{\mathbf{T_1(n)}}{\mathbf{T_p(pn)}} \tag{3.43}$$

En théorie, avec le conditionnement estimé par la formule (3.14), si on fixe la taille du problème global tout en augmentant le nombre de sous-domaines (par exemple, pour profiter de plus de processeurs disponibles) alors le nombre d'itérations nécessaires pour la convergence croît. On a donc un bon *speed-up*. Si au contraire, on maintient un nombre de degrés de liberté constant par sous-domaine tout en augmentant la taille du

problème global *via* le nombre de sous-domaines alors le nombre d'itérations demeure inchangé. On a donc un bon *scale-up*. Sur le papier, la méthode FETI résout donc un problème **n** fois plus grand avec le même temps CPU, pourvu que le nombre de processeurs **p** connaisse la même croissance. Ce résultat théorique a été prouvé en pratique pour des applications numériques (Farhat et Roux 1994; Farhat et al. 1994b; 1995). L'extensibilité parallèle de FETI a été aussi démontrée sur une variété de calculateurs parallèles ayant un nombre de processeurs compris entre **2** et **1000** (Bhardwaj et al. 2000; Farhat et al. 2000b).

3.4.3 Description des cas tests parallèles

La mesure des performances de la méthode FETI dans le code de calcul ZéBuLon (au sens des critères définis au paragraphe 3.4.2) nécessite la génération, pour un problème donné, d'une suite de modèles d'éléments finis où le nombre total de degrés de liberté augmente proportionnellement avec le nombre de processeurs, afin de maintenir la taille par sous-domaine constante. La génération d'une telle suite de modèles d'éléments finis et de partitions de maillage correspondantes est en général une tâche fastidieuse. Pour cette raison, nous considérons ici des cas tests de référence en $3D$ qui sont faciles à générer et à manipuler pour les deux études d'efficacité et d'extensibilité. Ces cas tests de référence correspondent à des problèmes d'élasticité linéaire sur des structures homogènes uniformément discrétisées par des éléments de nature hexaédriques à **20** nœuds et découpées en sous-domaines. Dans les deux cas d'études, nous équipons la méthode FETI d'un projecteur direct ($\mathbf{P} = \mathbf{P}(\mathbf{I})$), du préconditionneur *lumped* et nous surveillons la convergence à l'itération k par le critère relatif suivant :

$$||\mathbf{P}z^k||/||r^0|| < \varepsilon \tag{3.44}$$

où z^k est le résidu préconditionné et projeté du problème d'interface, r^0 est le saut des solutions particulières initiales le long des interfaces entre sous-domaines et ε est un critère de précision fixé par défaut à $\mathbf{10^{-6}}$.

3.4.4 Analyse de l'efficacité de FETI

Le point de départ des travaux consiste à évaluer et analyser l'efficacité de la version existante de FETI en utilisant localement le solveur DSCPack car ce dernier s'est révélé être plus efficace quand les systèmes linéaires sont inversibles. En cas de sous-structures flottantes, les mouvements de corps rigide sont au préalable traités avec Sparse Direct. Le cas test étudié se base sur un domaine cubique avec **28**x**28**x**28** éléments et **285099** degrés de liberté. Ce domaine de taille fixe est découpé avec METIS en $\mathbf{N_s}$ sous-domaines où $\mathbf{N_s}$ prend les valeurs **2**, **4**, **8**, **16**, **25**, **50** et **100** (voir le cas **50** sur la figure 3.6). Nous évaluons ensuite la version améliorée de FETI en partant d'un gros cas test pour bien charger les nœuds de calcul. Nous considérons le cas test se basant sur un domaine cubique comportant **64**x**64**x**64** éléments et **3371544** degrés de liberté (ddl) et dont le découpage en sous-domaines bien équilibrés se fait manuellement et facilement. Les

découpages effectués sont constitués de **8** (2x2x2, figure 3.7), **16** (4x2x2), **32** (4x4x2, figure 3.8), **64** (4x4x4), **128** (8x4x4) et **256** (8x8x4) sous-domaines.

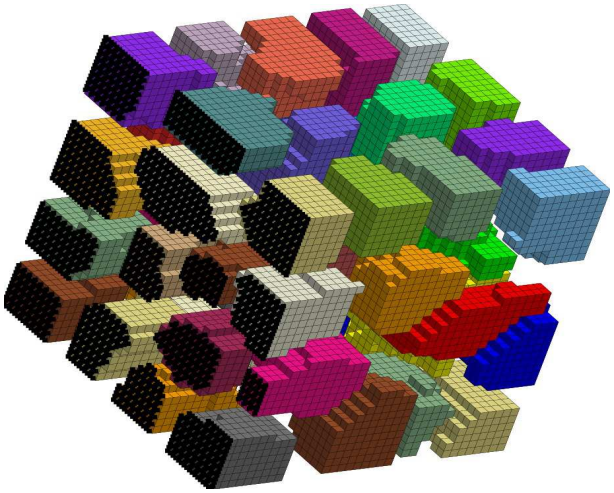


FIG. 3.6 – *Découpage du problème de taille* 285099 *en* 50 *sous-domaines .*

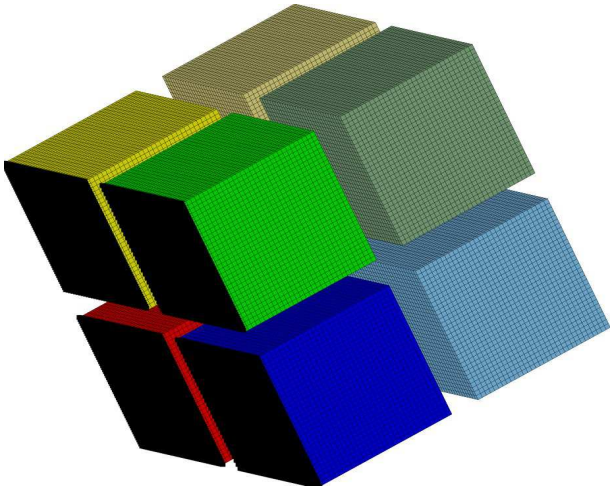


FIG. 3.7 – *Découpage du problème de taille* 3371544 *en* 8 *sous-domaines .*

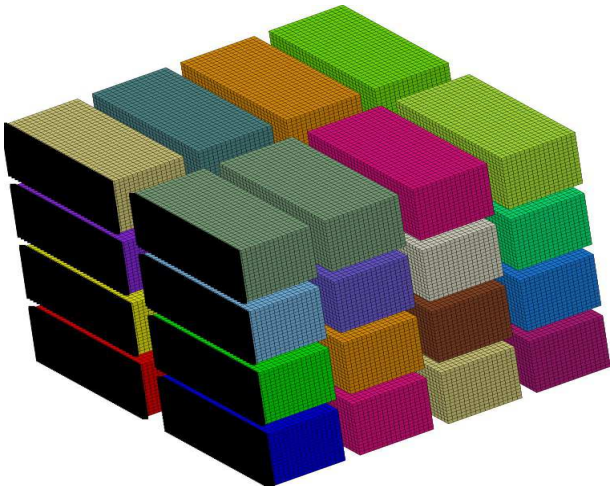


FIG. 3.8 – *Découpage du problème de taille* 3371544 *en* 32 *sous-domaines .*

Afin de bien quantifier l'efficacité de FETI, il n'est peut être pas tout à fait justifié d'évaluer la méthode FETI sur un processeur où elle se réduit à une résolution séquentielle. Nous avons donc pris comme point de repère le cas test $\mathbf{p = N_s = 2}$ pour la version existante et le cas test $\mathbf{p = N_s = 8}$ pour la version améliorée. Nous avons ensuite introduit plutôt les définitions suivantes de l'accélération :

$$\mathbf{S_p(n)} = \begin{cases} \frac{\mathbf{2*T_2(n)}}{\mathbf{T_p(n)}} & \text{pour la version existante} \\ \\ \frac{\mathbf{8*T_8(n)}}{\mathbf{T_p(n)}} & \text{pour la version proposée} \end{cases} . \tag{3.45}$$

La définition de l'accélération sur la formule (3.45) est de rigueur comparée à celle de (3.41). En effet, elle représente à la fois des concepts numériques et d'efficacité parallèle. Elle évalue les performances combinées de l'algorithme FETI, sa mise en œuvre parallèle et les calculateurs parallèles sur lesquels cet algorithme est exécuté. L'accélération et l'efficacité parallèle de chacune des deux versions de la méthode FETI sont analysées dans les paragraphes suivants.

L'efficacité de la version existante

La version existante de FETI a été évaluée sur le cluster du Centre des Matériaux. Chaque sous-domaine est attribué à un seul processeur et le multi-threading n'est pas activé dans le solveur local. Dans le tableau 3.1, nous reportons les temps d'exécution totale du code qui comprennent en grande partie des temps préparation des données et des temps passés dans l'algorithme FETI. Ces résultats caractérisent l'efficacité de cette version existante dans le code ZéBuLon. Ce type d'étude permet de voir le nombre optimal de sous-domaines pour lequel les temps d'exécution de la méthode FETI sont optimaux.

			Temps (s) dans ZéBuLon					
			Préparation		**FETI**		**Total**	
$\mathbf{N_s}$	$\mathbf{n_{max}}$	$\mathbf{N_{iter}}$	**CPU**	**elapsed**	**CPU**	**elapsed**	**CPU**	**elapsed**
2	147321	31	40.04	42.95	246.7	273.8	354.2	389.7
4	77805	46	24.73	27.16	85.53	102.4	138.5	164.4
8	40206	58	14.10	15.85	30.69	34.37	58.93	73.89
16	21327	83	8.610	19.72	18.37	25.02	33.72	61.02
25	14529	96	6.440	15.58	11.78	19.95	22.55	60.02
50	7965	119	4.230	18.91	6.530	20.08	12.84	81.04
100	4191	178	2.950	25.32	6.410	42.64	11.08	120.1

TAB. 3.1 – *Efficacité de la version existante de FETI.*

Comme nous pouvons le voir sur le tableau 3.1, le nombre $\mathbf{N_{iter}}$ d'itérations nécessaires pour la convergence de FETI devient de plus en plus important quand nous continuons de profiter du nombre de processeurs disponibles (**100**). Cela peut être interprété de la façon suivante : lorsque nous sous-structurons plus le domaine initial, la taille du problème d'interface devient plus grande et donc sa résolution réclame plus d'itérations pour converger. Les temps d'exécution *elapsed* de l'algorithme FETI et ceux

nécessaires pour le bon déroulement du calcul dans le code ZéBuLon sont quand même progressivement réduits. Mais, nous observons qu'au-delà de **25** sous-domaines ($\mathbf{N_s} \geq \mathbf{50}$), nous ne tirons plus profit du nombre de processeurs disponibles (voir la figure 3.9). Cela est dû au fait que les coûts de la communication sont grands par rapport à la charge de travail sur les processeurs. En d'autres termes, la taille $\mathbf{n_{max}}$ des sous-domaines est si petite ($\mathbf{n_{max}} \leq \mathbf{7965}$ ddl) que les processeurs finissent très rapidement leur travail et que les exigences des échanges de messages sont en légère croissance.

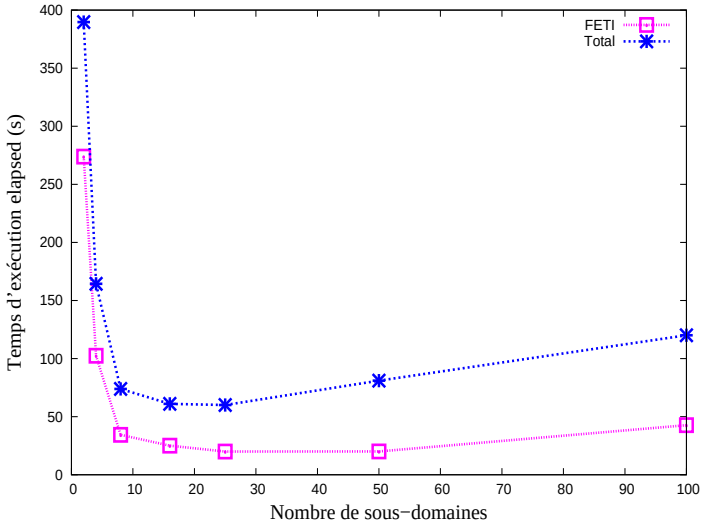


FIG. 3.9 – *Version exitante de FETI : efficacité dans ZéBuLon.*

Pour le calcul de l'accélération $\mathbf{S_p}$ et de l'efficacité $\mathbf{E_p}$ pour la version existante de la méthode FETI, nous avons pris comme temps CPU pour un calcul le plus important $\mathbf{T_p}$ des temps CPU des différents $\mathbf{p}$ processeurs nécessaires pour la convergence de l'algorithme. Nous avons reporté les résultats sur le tableau 3.2. Sur ce cas test, nous pouvons dire que le nombre optimal de sous-domaines $\mathbf{N_s}$ pour lequel les temps d'exécution de FETI sont optimaux ($\mathbf{Tp} = \mathbf{30.69}$ secondes) est égal à **8**. Cela correspond à la zone où l'efficacité est la plus élevée ($\mathbf{E_p} = \mathbf{200.9}\%$). Au-delà de cette zone, les accélérations sont moins importantes.

p	**2**	**4**	**8**	**16**	**25**	**50**	**100**
$\mathbf{T_p}$	246.7	85.53	30.69	18.37	11.78	6.530	6.410
$\mathbf{S_p}$	2.000	5.769	16.08	26.86	41.89	75.57	76.98
$\mathbf{E_p}$	1.000	1.442	2.009	1.679	1.676	1.511	0.770

TAB. 3.2 – *Accélération de la version existante de FETI.*

La courbe de l'accélération $\mathbf{S_p}$ tracée à partir des données du tableau 3.2 est présentée sur la figure 3.10. Nous observons sur cette figure que l'augmentation du nombre de processeurs a un effet négatif lorsque la taille des sous-domaines devient petite. nous constatons une dégradation de l'accélération quand le nombre $\mathbf{N_s}$ est égal à **100** ($\mathbf{S_p} = \mathbf{76.98}$). En effet, les explications sont les mêmes, les coûts de communication deviennent importants par rapport aux coûts de calcul. Lorsque les sous-

domaines sont de taille plus grande ($\mathbf{N_s} \leq \mathbf{50}$), les communications sont moins contraignantes car le temps passé à échanger des messages est petit par rapport aux temps de calcul. Pour ce type de problèmes, il faut chercher à éviter l'inactivité des processeurs ainsi qu'à obtenir une utilisation efficace de ceux-ci.

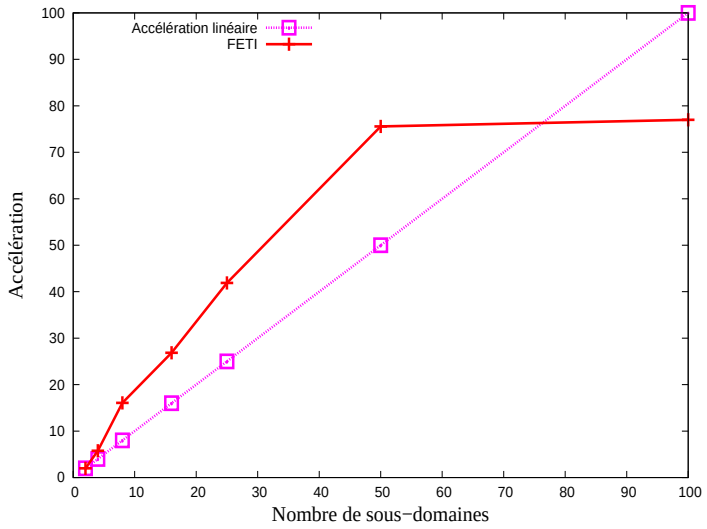


FIG. 3.10 – *Accélération de la version existante de FETI.*

L'efficacité de la version améliorée

Cette fois-ci, nous nous intéressons aux performances sur l'efficacité de la version améliorée évaluée sur le supercalculateur JADE. Pour chaque calcul, nous avons pris le plus grand temps *elapsed* passé dans les processeurs. Suivant le nombre de nœuds de calcul disponibles, nous décidons du nombre de processus FETI qu'il faut attribuer à chacun d'eux et donc le nombre de tâches à affecter au solveur local multi-threads. Nous avons alors la possibilité de rattacher au solveur local de **1** à **8** tâches. L'objectif de cette étude expérimentale est d'analyser le comportement de FETI pour un nombre $\mathbf{N_s}$ de sous-domaines différents et pour différents schémas de répartition de ces sous-domaines sur les $\mathbf{p}$ nœuds de calcul disponibles. Nous exploitons le parallélisme local sur un nombre de cœurs qui vaut respectivement **8**, **4**, **2**, et **1** si le nombre de nœuds $\mathbf{p}$ est égal à $\mathbf{N_s}$, $\mathbf{N_s/2}$, $\mathbf{N_s/4}$ et $\mathbf{N_s/8}$. Le tableau 3.3 présente cette distribution sur les tâches de calculs à exécuter.

$\mathbf{N_s}$ / $\mathbf{p}$	**8**	**16**	**32**	**64**	**128**	**256**
8	8	4	2	1		
16		8	4	2	1	
32			8	4	2	1
64				8	4	2
128					8	4
256						8

TAB. 3.3 – *Optimisation de la distribution des tâches.*

Le tableau 3.4 présente les temps *elapsed* en effectuant les calculs avec ZéBuLon pour chaque type de distribution. le temps nécessaire pour l'exécution totale du code est composé du temps de préparation, du temps passé dans l'algorithme FETI et du temps d'échange des données. Les temps sur chaque rangée sont obtenus avec le même nombre de nœuds de calcul et de cœurs. Pour chaque distribution, nous profitons du reste des cœurs disponibles pour accélérer la factorisation locale dans FETI. Les temps donnés sur chaque colonne informent sur l'importance de disposer de plus de machines pour réduire les temps d'exécution et de pouvoir tirer parti du parallélisme local.

			Nombre de sous-domaines ($\mathbf{N_s}$)					
		Phases	**8**	**16**	**32**	**64**	**128**	**256**
Nombre de nœuds de calcul (**p**)	**8**	**Préparation**	16.23	18.68	33.98	10.30		
		FETI	1186	366.5	179.1	172.4		
		Total	1299	438.4	242.3	201.8		
	16	**Préparation**		9.376	28.39	44.24	27.67	
		FETI		252.2	131.8	97.96	53.87	
		Total		310.6	188.9	160.1	93.23	
	32	**Préparation**			10.92	22.39	63.57	44.57
		FETI			86.63	79.75	33.21	24.18
		Total			125.4	118.4	114.0	79.15
	64	**Préparation**				11.73	18.90	54.82
		FETI				55.30	27.15	19.91
		Total				82.68	57.21	81.08
	128	**Préparation**					79.51	49.71
		FETI					21.70	17.61
		Total					114.6	81.14
	256	**Préparation**						167.2
		FETI						13.68
		Total						242.9

Tab. 3.4 – *Temps elapsed (s) dans ZéBuLon pour chaque type de distribution.*

En observant de près les résultats du tableau 3.4, nous constatons que les performances de l'algorithme FETI sont très dépendantes du nombre de cœurs disponibles pour traiter un sous-domaine. A mesure que nous diminuons le nombre de sous-domaines par nœud de calcul et pour un nombre total de sous-domaines fixé, les temps d'exécution de FETI s'améliorent puisque les attentes sont recouvertes par le parallélisme local. Avec un découpage en **64** sous-domaines, par exemple, nous sommes passés de **172.4** secondes (sur **8** nœuds) à **55.30** secondes (sur **64** nœuds). Ces résultats nous permettent de valider en partie que l'utilisation d'un solveur local s'appuyant sur des processus légers est plus que acceptable. D'autre part, nous remarquons que les temps nécessaires pour le calcul total dans ZéBuLon croissent au-delà de $\mathbf{p} = \mathbf{N_s} = \mathbf{64}$. Nous passons, par exemple, de **82.68** ($\mathbf{N_s} = \mathbf{64}$) à **242.9** secondes pour traiter le problème décomposé en **256** sous-domaines. Cela est dû au fait que les temps écoulés pour préparer les données du problèmes deviennent plus importants (**167.2** secondes) que les temps passés dans l'exécution de l'algorithme FETI (**13.68** secondes). Les coûts de communications sont négligeables par rapport aux

temps de calcul. Ces remarques peuvent être plus lisibles sur les courbes présentées sur la figure 3.11.

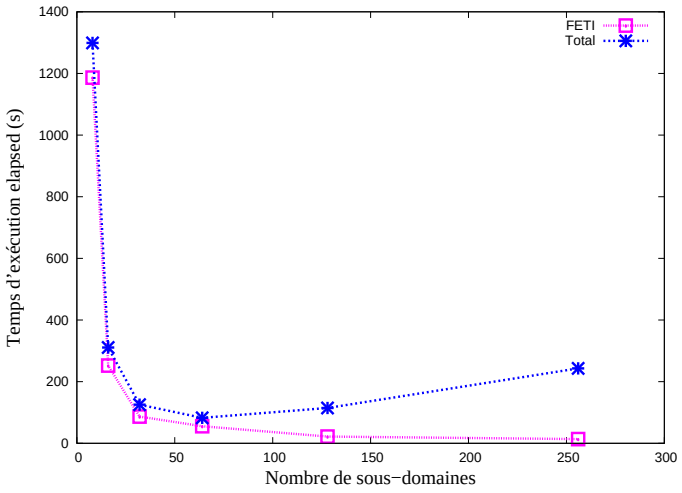


FIG. 3.11 – *Version améliorée de FETI : efficacité dans ZéBuLon.*

L'accélération $\mathbf{S_p}$ et l'efficacité $\mathbf{E_p}$ de la version améliorée de FETI ont été calculées en prenant les temps d'exécution des tests réalisés en utilisant au maximum **8** cœurs pour le parallélisme local. Nous avons donc pour chaque test autant de sous-domaines que de processeurs ($\mathbf{N_s} = \mathbf{p}$). Les résultats obtenus sont présentés sur le tableau 3.5.

p	**8**	**16**	**32**	**64**	**128**	**256**
$\mathbf{n_{max}}$	421443	215523	110211	56355	29427	15363
$\mathbf{N_{iter}}$	44	49	62	70	67	68
$\mathbf{T_p}$	1186	252.2	86.63	55.30	21.70	13.68
$\mathbf{S_p}$	8.00	37.62	109.5	171.5	437.2	693.5
$\mathbf{E_p}$	1.00	2.351	3.422	2.681	3.415	2.709

TAB. 3.5 – *Efficacité de la version améliorée.*

Sur les résultats du tableau ci-dessus, il peut sembler paradoxal que les cas avec **128** et **256** sous-domaines, et donc des problèmes d'interface plus grands, réclament moins d'itérations $\mathbf{N_{iter}}$ pour converger. En fait, cela s'explique par la forme (**H**) des sous-domaines qui influe énormément sur la convergence (voir le conditionnement du préconditionneur *lumped*, équation (3.12)). Un rapport de forme proche ou égal à **1** assure un meilleur taux de convergence, or nous sommes précisément dans les cas où les sous-domaines ont les configurations 8x16x16 et 8x8x16 éléments (leur taille **H** est donc légèrement plus petite). Le type de distribution pour laquelle l'efficacité est optimale (**342**%) correspond au calcul sous-structuré en **32** sous-domaines répartis sur **32** machines avec donc **8** cœurs par processus de FETI. Cela équivaut au total à **256** cœurs pour un temps de calcul égal à **86.63** secondes.

A partir des données du tableau 3.5, nous avons tracé, sur la figure suivante, la courbe de l'accélération $\mathbf{S_p}$ de FETI comparée à celle de l'accélération linéaire. Nous constatons sur la courbe FETI un effet de suraccélération produit par un découpage de plus en plus raffiné. Cette

courbe nous renseigne bien sur la super-linéarité de la version améliorée. Le choix d'un solveur local parallèle a bien amélioré les temps d'exécution.

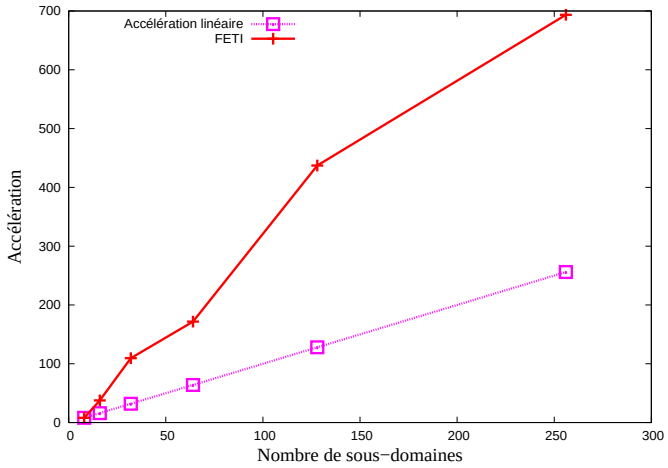


FIG. 3.12 – *Accélération de la version améliorée de FETI.*

Les résultats d'analyse sur l'efficacité de FETI que nous venons de présenter montrent que notre approche permet un maximum de parallélisme potentiel pour bien accélérer la convergence des méthodes de décomposition de domaine de type FETI. Cela permet de prédire une bonne efficacité lors de son utilisation.

3.4.5 Analyse de l'extensibilité de FETI

Le cas test étudié pour analyser l'extensibilité de la version existante de FETI est basé sur un domaine cubique de dimension $(\mathbf{1},\mathbf{1},\mathbf{1})$ et dont la taille globale augmente linéairement en fonction du nombre de sous-domaines. Nous partons d'une taille de sous-domaine égale à **141243** ddl puisque la taille optimale pour traiter un problème local est limitée à environ **140000** ddl sur chaque processeur des nœuds de calcul (cluster du Centre des Matériaux). Chaque sous-domaine est donc affecté à un seul processeur sans multi-threading. Sur la figure 3.13, nous donnons un exemple de découpage du problème de dimension $(\mathbf{1},\mathbf{1},\mathbf{1})$ en **2** sous-domaines dont les tailles sont **146700** et **147321** ddl.

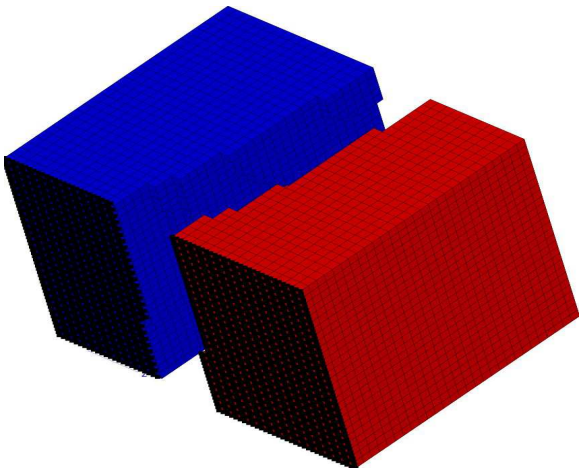


FIG. 3.13 – *Découpage du cube de dimension* $(1,1,1)$ *en* 2 *sous-domaines.*

Pour l'analyse de l'extensibilité de la version améliorée, nous avons considéré le cas test d'une poutre allongée et encastrée à une de ses extrémités. Cette poutre de dimension $(\mathbf{N_s}, \mathbf{1}, \mathbf{1})$ contient $\mathbf{N_s}$x32x32x32 éléments et elle est sous-structurée en $\mathbf{N_s}$ sous-domaines. Nous montrons, sur la figure 3.14, la poutre $(\mathbf{10}, \mathbf{1}, \mathbf{1})$ décomposée en **10** sous-domaines. Dans ce deuxième cas test, chaque sous-domaine est attribué à un seul nœud de calcul pour bien saturer toute la mémoire des nœuds de JADE que nous allons utiliser. Ici, nous utilisons donc autant de nœuds de calcul $\mathbf{p}$ que de sous-domaines $\mathbf{N_s}$ ($\mathbf{p} = \mathbf{N_s}$). Tous ces sous-domaines ont les mêmes caractéristiques locales et chacun d'eux est composé de **32**x**32**x**32** éléments et **421443** ddl.

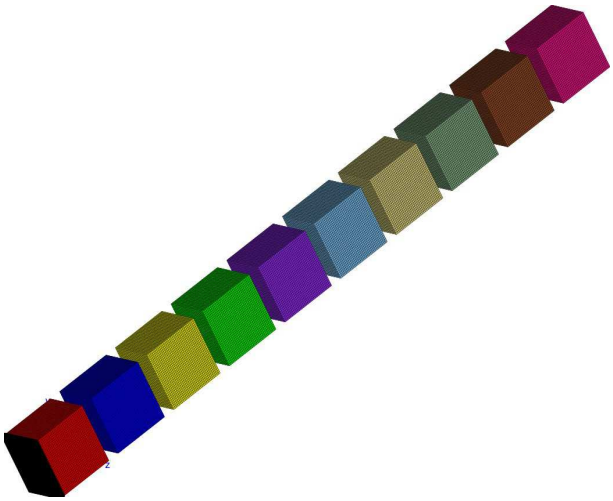


FIG. 3.14 – *Découpage de la poutre de dimension* $(10, 1, 1)$ *en* 10 *sous-domaines.*

Comme dans l'analyse de l'efficacité de FETI, en voulant avoir une idée sur l'extensibilité de la méthode FETI, nous avons pris comme point de repère le cas test $\mathbf{p} = \mathbf{N_s} = \mathbf{2}$ pour les deux versions et nous avons introduit plutôt la définition suivante sur l'extensibilité :

$$\mathbf{A_p(n)} = \frac{\mathbf{T_2(2n)}}{\mathbf{T_p(pn)}}. \tag{3.46}$$

L'extensibilité de la version existante

L'extensibilité de la version existante de FETI a été évaluée sur le cluster du Centre des Matériaux qui comptait à l'époque de **78** nœuds de calcul différents (**25** nœuds mono-processeur et **53** nœuds bi-processeurs). Nous avions donc la possibilité de partitionner le domaine global jusqu'à $\mathbf{N_s}$ égal à **131** sous-domaines. Les résultats des évaluations sur l'extensibilité sont présentés dans le tableau 3.6.

Tout d'abord, le fait d'augmenter linéairement la taille $\mathbf{n}$ du domaine global en fonction de $\mathbf{N_s}$ et de maintenir sa géométrie fixe revient à raffiner de plus en plus le maillage (faire tendre $\mathbf{h}$ vers **0**). Les résultats sur le nombre $\mathbf{N_{iter}}$ d'itérations est pratiquement ceux qui sont attendus, c'est-à-dire une augmentation du taux de convergence lorsque la taille du problème passe de **285099** ($\mathbf{N_{iter}} = \mathbf{31}$) à **12640860** degrés de liberté ($\mathbf{N_{iter}} = \mathbf{185}$). Cela s'explique par le conditionnement du problème

			Temps (s)					
			FETI		ZéBuLon + FETI			
					Préparation		Total	
N_s	n	N_{iter}	CPU	elapsed	CPU	elapsed	CPU	elapsed
2	285099	31	246.7	273.8	40.04	42.95	354.2	389.7
4	503475	46	248.6	286.0	74.82	90.69	382.1	444.9
8	1075275	64	350.4	403.6	183.1	199.7	598.2	680.2
16	1969275	81	381.2	436.4	330.4	378.9	751.1	905.3
25	3108864	105	512.3	589.4	541.0	627.1	1105	1470
50	6318243	141	605.8	956.9	1145	1303	1778	2526
80	9887484	157	696.8	1110	1763	1972	2428	3561
100	12640860	185	802.5	1365	2290	2613	3118	4763
131	16746240	168	4525	5285	3175	4697	7760	11592

TAB. 3.6 – *Extensibilité de la version existante de FETI.*

d'interface qui devient plus grand dès que **h** tend vers zéro. Quand le nombre de sous-domaines est égal à **131**, la chute du nombre d'itérations ($\mathbf{N_{iter} = 168}$) est due à un déraffinement du maillage qui a consisté à changer les dimensions du domaine global en passant de $(\mathbf{1,1,1})$ à $(\mathbf{100,100,100})$. Ce déraffinement est effectué pour éviter des problèmes d'instabilité qui proviendraient de la taille des mailles.

Du tableau précédent, nous avons tracé, fonction du nombre de sous-domaines, la courbe sur le temps d'exécution *elapsed* de FETI et du temps *elapsed* total dans le code ZéBuLon (voir la figure ci-dessous).

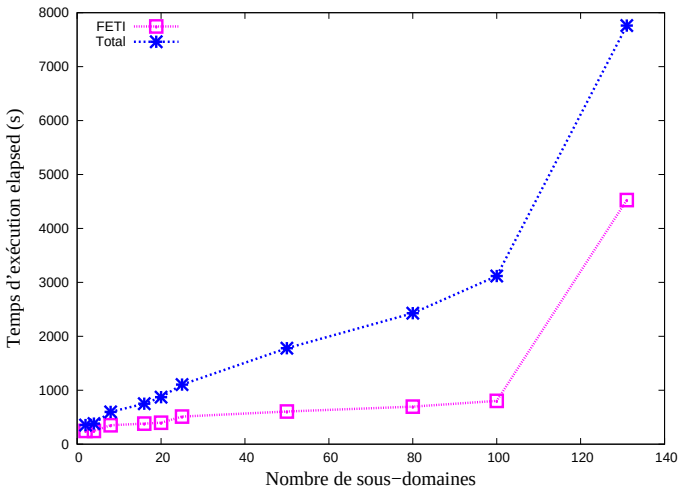


FIG. 3.15 – *Version existante de FETI : extensibilité dans ZéBuLon.*

Les courbes de la figure 3.15 montrent que le temps d'exécution croît linéairement en fonction du nombre de sous-domaines, ce qui confirme une bonne extensibilité de la méthode FETI jusqu'à **100** sous-domaines. En fait, lorsque le nombre de sous-domaines augmente, la taille du problème d'interface devient grande et exige plus d'itérations pour converger. Cela se traduit par une augmentation du coût des opérations relatives à l'interface. Nous observons aussi que ces temps explosent lorsque nous passons à **131** sous-domaines bien que le nombre d'itérations soit plus

petit. Cette perturbation des temps vient du nombre important de messages échangés entre les sous-domaines et aussi au fait que nous avons totalement saturé l'ensemble du cluster qui, en plus, n'a pas de réseau rapide. L'écart de temps noté entre le temps d'exécution de FETI et celui de la résolution totale est en gros le temps de chargement séquentiel des données du problème. Quand la taille du problème croît, il arrive parfois que ce temps de chargement soit plus important que le temps d'exécution de FETI détériorant ainsi les performances de ZéBuLon en parallèle.

p	**2**	**4**	**8**	**16**	**25**	**50**	**80**	**100**	**131**
$\mathbf{A_p}$	1.000	0.957	0.678	0.627	0.465	0.286	0.247	0.201	0.051

TAB. 3.7 – *Scale-up de la version existante de FETI.*

Le tableau 3.7 montre les résultats d'extensibilité de FETI pour des calculs effectués sur **2** à **131** processeurs. Nous avons évalué le *scale-up* et tracé la courbe associée sur la figure 3.16.

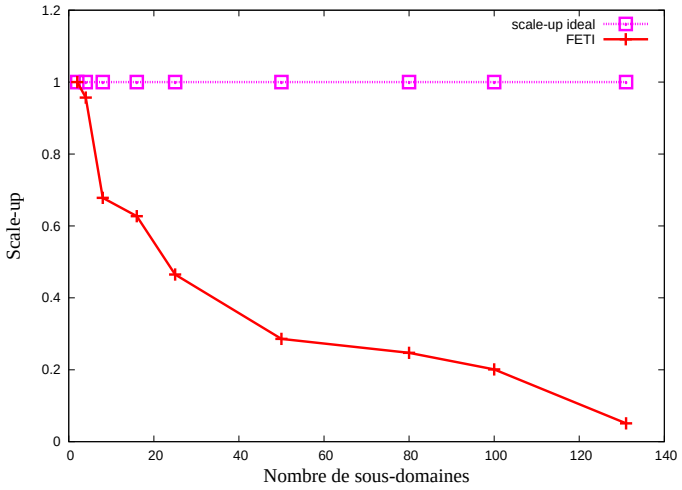


FIG. 3.16 – *Scale-up de la version existante de FETI.*

La figure 3.16 montre les résultats d'extensibilité de la version existante de FETI en fonction du nombre de processeurs. Les performances chutent considérablement pour les raisons évoquées précédemment. Quand nous passons de **4** à **131** sous-domaines, le *scale-up* $\mathbf{A_p}$ passe de **95.7**% à **5.1**%. Malgré cette perte partielle d'extensibilité sur ce type d'études, nous avons réussi à effectuer des tests jusqu'à atteindre la taille de **16746240** degrés de liberté en presque **3** heures d'horloge sur **131** processeurs.

L'extensibilité de la version améliorée

L'étude de l'extensibilité de la version améliorée de FETI met en évidence le fait qu'elle permet un choix aisé du nombre de sous-domaines et une répartition flexible de ces sous-domaines sur les nœuds de calcul. Cette flexibilité est très utile pour la mise en œuvre de notre approche dans FETI. Les tests sur l'analyse sont effectués sur le supercalculateur JADE. Nous rappelons que, pour un cas parfaitement extensible, les temps d'exécution doivent rester constants lorsque nous augmentons le nombre de nœuds de calcul. Le tableau 3.8 regroupe les résultats obtenus pour cette étude.

			Temps elapsed (s)		
				ZéBuLon + FETI	
$\mathbf{N_s}$	**n**	$\mathbf{N_{iter}}$	**FETI**	**Préparation**	**Total**
2	833283	16	1077	11.81	1187
10	4128003	17	1125	12.57	1235
30	12364803	15	1117	19.93	1237
50	20601603	13	1112	26.83	1244
80	32956803	13	1124	48.15	1288
100	41193603	12	1119	64.80	1304
200	82377603	12	1132	254.3	1544
300	123561603	12	1145	440.1	1773
400	164745603	12	1148	662.8	2018

TAB. 3.8 – *Extensibilité de la version améliorée de FETI.*

L'augmentation de la taille **n** du problème global n'entraîne que peu d'inflation sur le nombre d'itérations $\mathbf{N_{iter}}$. Ce nombre est légèrement variable (entre **12** et **17** itérations). Cela s'explique par le fait que nous travaillons à $\mathbf{H/h}$ constant, et donc avec un conditionnement κ (défini par la formule (3.12)) qui est très stable. Le nombre d'itérations nécessaire pour la convergence de FETI ne dépend que peu du nombre de sous-domaines. Mais le plus remarquable concerne les temps de calcul de FETI qui restent à peu près constants (figure 3.17).

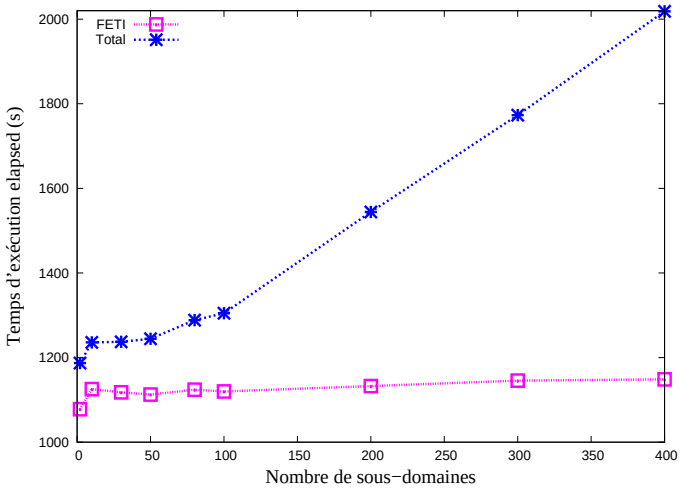


FIG. 3.17 – *Version améliorée de FETI : extensibilité dans ZéBuLon.*

Sur la figure 3.17, la différence de temps notée entre le temps d'exécution de FETI et le temps total provient du chargement du maillage et des coûts de communication entre les nœuds de calcul. En augmentant le nombre de sous-domaines $\mathbf{N_s}$, le maillage devient plus gros et son chargement et les échanges entre sous-domaines requièrent plus de temps.

p	**2**	**10**	**30**	**50**	**80**	**100**	**200**	**300**	**400**
$\mathbf{A_p}$	1.	0.957	0.964	0.968	0.958	0.962	0.951	0.941	0.938

TAB. 3.9 – *Scale-up de la version améliorée de FETI.*

Dans le tableau 3.9, nous avons reporté les résultats d'extensibilité de la version améliorée de FETI pour des calculs effectués sur **2** à **400** nœuds de calcul. Nous avons mesuré le *scale-up* en prenant comme temps de référence le temps $\mathbf{T_2(2n)}$ pour deux sous-domaines, puis nous avons tracé la courbe associée sur la figure 3.18.

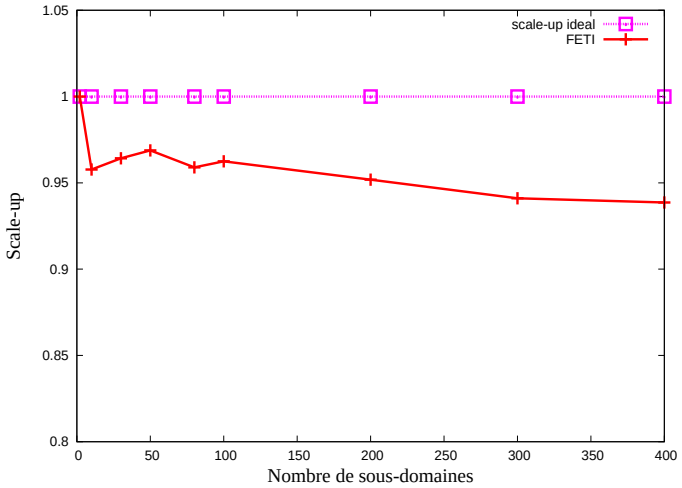


FIG. 3.18 – *Scale-up de la version améliorée de FETI.*

La figure 3.18 montre les résultats d'extensibilité de la version améliorée de FETI en fonction du nombre de cœurs. Les performances restent presque les mêmes pour les raisons évoquées plus haut. Quand nous passons de **2** à **400** sous-domaines, le *scale-up* $\mathbf{A_p}$ reste supérieur à $\mathbf{93,8}\%$. Nous pouvons dire que la méthode est numériquement extensible sur ce type d'études. Le plus gros calcul réalisé ici avec **400** nœuds de calcul (**3200** cœurs) comporte **164475603** degrés de liberté. Nous avons réussi cette réalisation en moins de **34** minutes, ce qui est hors de portée des codes commerciaux implicites classiques.

Nous avons évalué les temps de la phase de détection des modes à énergie nulle. Ces modes rigides sont très bien traités pour l'approche proposée, ce qui valide l'effort effectué dans le cadre de ces travaux. Dans le cas de la poutre fixée sur un de ses extrémités, le nombre de modes est en général égal à $\mathbf{3 + 6x(N_s - 2)}$. En effet, les sous-domaines internes au nombre de $\mathbf{N_s - 2}$ sont flottants et possèdent donc **6** mouvements de corps rigide chacun, l'un des deux sous-domaines situés aux extrémités est fixé et l'autre possède **3** mouvements de corps rigide.

Nous présentons sur le tableau 3.10 les temps de calcul nécessaires pour effectuer la phase de détection. Pour chaque calcul, ce temps est le temps passé pour construire le problème grossier associé aux sous-structures flottantes. Les modes rigides sont très bien traités pour l'approche proposée. De plus, les temps de calcul nécessaires pour leur traitement ne sont pas énormes comparés aux temps de résolution totale.

$\mathbf{N_s}$	**2**	**10**	**30**	**50**	**80**	**100**	**200**	**300**	**400**
Noyau	14.85	26.06	25.99	26.36	26.07	26.17	26.66	27.35	27.12

TAB. 3.10 – *Temps (s) de traitement des mouvements de corps rigide.*

Conclusion

Dans ce chapitre, nous avons proposé une approche pour accélérer la résolution dans les méthodes de décomposition de domaine de type FETI. Cette approche a consisté à utiliser comme solveur local dans ces méthodes la version multi-threads hybride du solveur direct que nous avons mis en œuvre et qui est basée sur les threads POSIX et la bibliothèque de BLAS optimisée avec OpenMP. L'intégration de cette approche dans la méthode FETI implantée dans le code de calcul ZéBuLon a été rendue possible du fait que, dans celle-ci, la détection des mouvements de corps rigide présents dans les sous-structures flottantes est prise en compte. Dans les nombreux cas tests étudiés avec ZéBuLon, la nouvelle méthode hybride s'avère être plus efficace que la méthode FETI existante. Ceci permet d'affirmer l'utilité des travaux effectués et l'intérêt de l'utilisation de cette version améliorée. Cela peut donner accès à de nouvelles gammes de problèmes en calcul des structures qui exploitent de façon efficace plus d'un millier de cœurs.

Conclusion générale

Dans ces travaux, nous nous sommes plaçés dans le cadre de la résolution de systèmes linéaires de grande taille issus de la méthode des éléments finis dans le but de traiter, sur des calculateurs massivement parallèles, de gros modèles en calcul des structures.

L'approche retenue est d'exploiter un parallélisme à deux niveaux et l'introduire dans les méthodes de décomposition de domaine sans recouvrement de type FETI qui sont capables de réduire les coûts de calcul.

Dans un premier temps, nous avons mis en œuvre un solveur direct pour l'inversion de systèmes linéaires creux symétriques et réels, à second membre simple ou multiple. Cette mise en œuvre est basée sur une technique de dissection emboîtée dont la stratégie est de "diviser pour mieux régner". Le solveur a été évalué et comparé en séquentiel avec d'autres solveurs directs disponibles dans le code de calcul ZéBuLon notamment les solveurs linéaires DSCPack et MUMPS. Le solveur DSCPack est basé sur une approche similaire à celle de la dissection emboîtée et MUMPS repose, quant à lui, sur une approche multi-frontale. Mais, un des points faibles de ces deux solveurs est que DSCPack ne prend pas en compte les systèmes singuliers et que MUMPS n'est pas très robuste sur l'inversion des systèmes linéaires provenant des structures mécaniques flottantes et très hétérogénes. Les performances obtenues avec ce nouveau solveur ont été très prometteuses. Dans ce solveur direct, nous avons aussi pris en compte les systèmes non inversibles en calculant leurs modes à énergie nulle. Ces modes jouent un rôle principal dans beaucoup de simulations numériques en mécanique.

Dans un deuxième temps, nous avons ainsi mis en place une version multi-threads hybride du solveur basée sur les threads POSIX (Pthreads) et la bibliothèque de BLAS optimisée avec OpenMP. Cette mise en place a consisté à prendre, dans chacune des phases du solveur direct, le standard Pthreads ou OpenMP qui exploite mieux le multi-threading même si chacun d'eux a permis bien sùr de réduire les temps d'exécution séquentielle. Nous avons alors réfléchi sur la manière de développer cette version hybride en compliquant aussi peu que possible le solveur, afin de coller au maximum à sa version séquentielle. Cette version fonctionne sur une architecture à mémoire partagée. Malgré quelques difficultés rencontrées sur les dépendances des calculs, nous avons réussi à créer dans cette version, des threads dans plus de $\mathbf{2/3}$ des parties du solveur. Ses performances ont été mesurées et analysées sur une machine bi-processeurs quadri-cœurs (**8** cœurs au total) et la comparaison avec les solveurs DSCPack et MUMPS, dont nous avons juste activé la librairie BLAS optimisée avec OPenMP, a été ainsi faite. Les analyses nous ont permis de voir que les gains sont plus importants avec un petit nombre de cœurs (**2** ou **4**) sur ce type de machine et que ces gains n'arrivent pas à dépasser la valeur **3** qui semble être une asymptote pour le moment.

Dans un troisième temps, nous avons proposé une approche pour accélérer la résolution dans les méthodes de décomposition de domaine sans recouvrement de type FETI. Avant tout, nous avons introduit ces dernières de manière générale tout en montrant les caractéristiques communes à toutes les variantes. L'approche proposée a consisté à utiliser comme solveur local dans ces méthodes la version multi-threads hybride du solveur direct que nous avons mis en œuvre et qui est basée sur Pthreads et OpenMP. L'intégration de cette approche dans la méthode FETI implantée dans le code de calcul ZéBuLon a été rendue possible du fait que, dans celle-ci, la détection des mouvements de corps rigide présents dans les sous-structures flottantes est prise en compte. Les premières validations de cette approche sont encourageantes. Les temps d'exécution pour les problemes d'élasticité linéaire sont largement réduits et les modes à énergie nulle sont très bien traités pour les systèmes locaux singuliers. De plus, ce traitement ne vaut pas grande chose comparé aux coûts de la résolution totale. Dans les nombreux cas tests étudiés avec le code ZéBuLon, la nouvelle méthode hybride s'est avèrée être plus efficace que la méthode FETI existante. Ceci permet d'affirmer l'utilité des travaux effectués et l'intérêt de l'utilisation de cette version améliorée.

Perspectives

Les résultats que nous avons obtenus à l'issue de ces travaux de thèse sur la résolution de systèmes linéaires de grande taille sont encourageants. Toutefois certains points sont encore à améliorer et de nombreux axes de travaux futurs apparaissent.

La mise en œuvre complète du solveur comprenant la prise en compte des systèmes linéaires symétriques complexes, non symétriques réels ou complexes sera envisagée à court ou moyen terme, et le faire fonctionner sur toutes les architectures et tous les systèmes d'exploitation disponibles actuellement sur le marché.

Concernant la génération des super-nœuds par dissection emboîtée, avons développé une stratégie expérimentale qui permet d'estimer le nombre optimal de sous-structures conduisant au meilleur temps d'exécution pour la majeure parti des problèmes que nous sommes capables de traiter. Mais, nous pensons qu'une relation judicieuse doit être trouvée pour rendre possible le calcul de leur taille optimale en fonction de celle du problème à traiter. Des études ont montré que les tailles de ces super-nœuds ont une grande influence sur les performances du solveur.

Dans la mise en place la version multi-threads, ce serait bien aussi d'exploiter la part du solveur direct non encore parallélisée (**1**/**3**) pour envisager de tirer mieux parti des machines disposant de **16** cœurs, voire **80** cœurs. Contrairement à ce que font les bibliothèques de solveurs directs parallèles standards, un effort particulier pourrait bien être fourni pour la parallélisation des résolutions simultanées ou successives, puisque, dans un contexte de résolution itérative par sous-domaines du problème initial ou global, chaque problème local devra être résolu un grand nombre de fois avec différents seconds membres.

Dans l'amélioration des performances des méthodes de décomposition de domaine de type FETI, nous pouvons revoir le parallélisme à gros grains entre les sous-domaines. Cette approche d'amélioration pourrait consister à coupler, au niveau global, la version du solveur, à second membre multiple, avec des méthodes itératives de type Krylov pour des calculs avec plusieurs seconds membres ou directions de descente. Ce couplage permetttra de diminuer la fréquence des tranferts de données entre groupe de processeurs et ainsi accélérer la convergence des méthodes FETI.

Annexes

A

Sommaire

A.1 Factorisation par élimination de Gauss 105
A.1.1 Les généralités de la méthode 105
A.1.2 Les inconvénients de la méthode 107
A.2 Algorithmes de renumérotation 109
A.2.1 L'algorithme de dissection emboîtée 110
A.2.2 L'algorithme de degré minimum et ses variantes 110
A.2.3 L'algorithme de Cuthill - McKee 111
A.3 Présentation de la méthode des éléments finis 112
A.3.1 Le problème modèle 113
A.3.2 La mise sous forme variationnelle d'un problème d'EDP . 113
A.3.3 Le choix d'un maillage et discrétisation 114
A.3.4 Le problème sous forme matricielle 116
A.4 Analyse des performances des versions multi-threads du solveur Dissection 117
A.4.1 Les temps de calcul pour la phase d'analyse 117
A.4.2 Les temps de calcul pour la phase numérique 119
A.4.3 Les temps de calcul pour la phase de descente-remontée . 120
A.4.4 Les temps d'exécution en fonction du nombre de threads 121
A.4.5 Les conclusions 121

A.1 Factorisation par élimination de Gauss

Partant de la constatation qu'il est facile de résoudre le système $Mx = b$ lorsque $M = (m_{ij})_{1 \leq i,j \leq n}$ est une matrice triangulaire inférieure ou supérieure, on cherche à décomposer la matrice initiale pleine par une factorisation de matrices triangulaires.

A.1.1 Les généralités de la méthode

Le principe de base est de rechercher une matrice régulière P, dite matrice de permutation, telle que le produit PM soit triangulaire, puis de résoudre $PMx = Pb$. Dans la pratique, P est déterminée par le produit de matrices élémentaires de permutation $P = P^k \dots P^2 P^1$. Les matrices P^i dépendent de la variante choisie, mais la matrice P n'est jamais calculée explicitement ; seuls les produits PM et Pb le sont.

La matrice M étant factorisée sous la forme générale LU (les matrices L et U sont respectivement triangulaire inférieure et triangulaire supérieure), on est amené à résoudre les deux systèmes linéaires : $Ly = b$ et $Ux = y$. Dans la méthode dite d'élimination de Gauss, on réalise simultanément la factorisation de M et la résolution de $Ly = b$. L'algorithme A.1 réalise l'élimination de Gauss et la résolution de $Ly = b$ à l'étape $(k+1)$. Dans cette écriture de l'algorithme d'élimination de Gauss, le second membre b est considéré comme une colonne supplémentaire de la matrice qui est alors traitée comme une matrice de taille nx$(n+1)$.

Algorithme A.1 : Étape $(k+1)$ de l'élimination de Gauss

pour $i = k+1$ *à* n **faire**
  pour $j = k+1$ *à* $n+1$ **faire**
    ${m_{ij}}^{(k+1)} = {m_{ij}}^{(k)} - {m_{ik}}^{(k)} * \left({m_{kk}}^{(k)}\right)^{-1} * {m_{kj}}^{(k)}$;
  fin
fin
pour $i = 1$ *à* k **faire**
  pour $j = 1$ *à* $n+1$ **faire**
    ${m_{ij}}^{(k+1)} = {m_{ij}}^{(k)}$;
  fin
fin
pour $i = k+1$ *à* n **faire**
  pour $j = 1$ *à* k **faire**
    ${m_{ij}}^{(k+1)} = 0.$
  fin
fin

La décomposition par l'élimination de Gauss n'est pas unique, mais si l'on spécifie la diagonale de L ou de U, alors on aboutit à son unicité.

Notion de pivot

La mise en œuvre de l'algorithme d'élimination de Gauss A.1 suppose implicitement que les termes diagonaux m_{kk}, appelés pivots, ne sont pas

nuls en cours de calcul. En cas de pivot nuls, on peut utiliser une des stratégies de pivotage qui suivent.

- **Le pivotage complet** : cette stratégie consiste à choisir comme pivot le plus grand terme dans le bloc restant, puis d'effectuer une permutation de ligne et de colonne. On a alors le système $PMQ(Q^T x) = Pb$, où P est la matrice de permutation des lignes et Q celle des colonnes. La solution trouvée est alors $y = Q^T x$. Il est donc nécessaire de conserver Q pour obtenir la solution cherchée $x = Qy$.
- **Le pivotage partiel** : le pivot est recherché comme étant le terme de valeur maximale, parmi les termes non encore traités, dans la colonne courante (la k-ième à l'étape k), puis on effectue une permutation de ligne.

Stabilité de la méthode

Une méthode numérique de résolution de système linéaire est dite mathématiquement **stable** lorsque, quelle que soit la matrice M régulière, l'algorithme aboutit. La méthode de Gauss avec une des deux stratégies de pivotage est mathématiquement stable si M est une matrice régulière. Sans pivotage, la méthode de Gauss est seulement stable pour les matrices réelles définies positives. Au cours de la factorisation de Gauss avec pivotage, si un pivot nul est détecté, alors la matrice M est singulière et le système $Mx = b$ n'a pas de solution unique.

Variante : la factorisation de Crout

La méthode de factorisation de Crout (Crout 1941; Lascaux et Théodor 2004a) est presque le même algorithme. Il nécessite le même nombre d'opérations et effectue le même remplissage de la matrice mais les calculs sont menés de façon différente. On se place dans le cas où la matrice est factorisable, ce qui est toujours le cas à une permutation près des lignes et des colonnes dès lors que la matrice est régulière. On a donc : $M = LU$. Si l_{ij} et u_{ij} sont les coefficients de L et U alors on a :

$$\begin{cases} m_{ij} = u_{ij} + \sum_{k=1}^{i-1} l_{ik} * u_{kj}, & \text{pour } 1 \leq i \leq j \leq n \\ m_{ij} = \sum_{k=1}^{j} l_{ik} * u_{kj}, & \text{pour } 1 \geq i > j \geq n \end{cases} . \tag{A.1}$$

En procédant par identification, les valeurs de l_{ij} et u_{ij} en fonction des coefficients m_{ij} s'écrivent :

$$\begin{cases} u_{1j} = m_{1j}, & \text{pour } j = 1, \ldots, n \\ l_{i1} = m_{i1}/u_{11}, & \text{pour } i = 1, \ldots, n \\ u_{ij} = m_{ij} - \sum_{k=1}^{i-1} l_{ik} * u_{kj}, & \text{pour } 1 \leq i \leq j \leq n \\ l_{ij} = \frac{1}{u_{jj}}(m_{ij} - \sum_{k=1}^{j-1} l_{ik} * u_{kj}), & \text{pour } 1 \geq i > j \geq n \end{cases} . \tag{A.2}$$

L'ordre des calculs n'est pas arbitraire. Il suffit juste de connaître les termes l_{ik} situés à gauche et les u_{kj} au-dessus de chaque terme à calculer.

On voit alors qu'à la k-ième étape, toutes les contributions antérieures sont reportées sur la k-ième ligne, laissant inchangées les lignes comprises entre $k+1$ et n. Cette variante de Crout, appelée aussi élimination de Gauss par colonnes, prévilégie l'opération de produit scalaire.

Cas des matrices symétriques

La décomposition de Gauss respecte la symétrie. Si M est une matrice symétrique, elle peut donc être factorisée sous la forme $M = LDL^T$, où D est une matrice diagonale et L une matrice triangulaire inférieure à diagonale unitaire. Cette décomposition, unique puisque l'on a fixé une diagonale, s'applique à toute matrice symétrique non singulière. Si la matrice M est définie positive, alors les termes de la diagonale sont strictement positifs et l'on peut utiliser la forme dite de Cholesky $M = LL^T = (LD^{1/2})(D^{1/2}L^T)$.

A.1.2 Les inconvénients de la méthode

Les inconvénients des méthodes de type Gauss sont essentiellement de trois types :

1. un nombre élevé d'opérations,
2. le remplissage de la matrice,
3. un mauvais conditionnement de la matrice.

Complexité de la méthode

Pour un système linéaire plein de taille n, à la k-ième étape, nous devons effectuer pour calculer les nouveaux coefficients l_{ij} et u_{ij} de la matrice M et le second membre b :

- $(n-k)$ divisions ;
- $(n-k+1)$x$(n-k)$ additions et multiplications.

Le nombre total d'opérations est donc :

$$\begin{aligned} &\sum_{k=1}^{k-1}(n-k) = \frac{n(n-1)}{2} \text{ divisions ;} \\ &\sum_{k=1}^{n-1}(n-k+1)(n-k) = \frac{n(n-1)(n-2)}{6} + \frac{n(n-1)}{2} \\ &\text{additions et autant de multiplications.} \end{aligned} \tag{A.3}$$

Soit $\frac{1}{3}n$x$(n-1)$x$(n+\frac{1}{2})$ opérations auxquelles il convient d'ajouter les n^2 opérations de la résolution des systèmes triangulaires. Pour tout résumer, la résolution d'un grand système linéaire plein par l'algorithme de Gauss nécessite de l'ordre de $\frac{1}{3}n^3$ opérations.

Remplissage de la matrice

Commençons par l'exemple classique d'une matrice M dite matrice "**flèche**". Les termes m_{ij} de cette matrice sont telles que $m_{1i} \neq 0$, $m_{i1} \neq 0$, $m_{ii} \neq 0$ et tous les autres termes sont nuls. La matrice M a donc l'allure suivante :

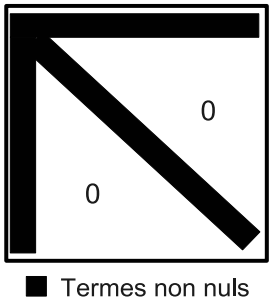


FIG. A.1 – *Exemple classique de matrice dite "flèche".*

Après la première étape de factorisation par élimination de Gauss, la matrice est pleine dans le sens où il n'y a plus de coefficients théoriquement nuls. En réécrivant l'algorithme de Gauss sous la nouvelle forme A.2, un regard plus formel permet de constater que le terme m_{ij} est non nul à la fin de la k-ième étape :

- s'il était non nul au début de cette k-ième étape,
- ou si les termes m_{ik} et m_{kj} sont tous les deux non nuls au début de la k-ième étape, et ceci indépendamment de la valeur initiale de m_{ij}.

Algorithme A.2 : Élimination de Gauss : phénomène de remplissage

boucle sur les étapes :
pour $k = 1$ *à* $n - 1$ **faire**
  boucle sur les lignes :
  pour $i = k + 1$ *à* n **faire**
    boucle sur les colonnes :
    pour $j = k + 1$ *à* n **faire**
      $m_{ij} = m_{ij} - m_{ik} * m_{kj} / m_{kk}$
    fin
  fin
fin

De plus, on voit que la méthode de Gauss remplit le profil de la matrice au cours des étapes de factorisation. Dans l'exemple de la matrice "flèche", le profil est la matrice pleine, d'où le résultat constaté. Cet exemple met en évidence l'importance de la numérotation des inconnues de la matrice puisqu'elle peut être réécrite, après permutation inverse des inconnues, sous une forme décrite sur la figure A.2, où le profil est la matrice elle même. Il n'y a donc pas de remplissage.

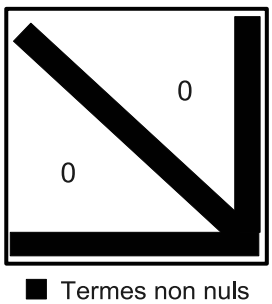


FIG. A.2 – *Matrice "flèche" avec permutation inverse des inconnues.*

Nous ne saurions trop insister sur le fait que des algorithmes de renumérotation "optimale" doivent être utilisés pour minimiser le remplissage de la matrice. Ces techniques reposent sur des heuristiques et sont spécialisées.

Conditionnement de la matrice

Un grand conditionnement de la matrice M conduit à des pertes de précision. Ces pertes proviennent du fait qu'en cours de décomposition, les pivots décroissent et qu'ils sont utilisés comme dénominateur pour les étapes suivantes. Le conditionnement intervient constamment pour les matrices carrées inversibles et il dépend de la norme matricielle choisie. Ce conditionnement est toujours supérieur ou égal à un, pour toute norme choisie et une matrice est d'autant mieux conditionnée que son conditionnement est proche de un.

En choisissant comme norme la norme euclidienne,

- le conditionnement d'une matrice M quelconque est alors

$$\kappa(M) = \frac{\mu_{max}}{\mu_{min}} \tag{A.4}$$

  où μ_{max} et μ_{min} sont respectivement la plus grande et la petite des valeurs propres de la matrice M^*M ;
- si la matrice M est symétrique (ou hermitienne), alors

$$\kappa(M) = \frac{max|\lambda_i(M)|}{min|\lambda_i(M)|} \tag{A.5}$$

  où les λ_i sont les valeurs propres de M.

La méthode la plus simple pour réduire le conditionnement est celle de la mise à l'échelle de M. Cela consiste à passer de M à $\Delta_1 M \Delta_2$. Les Δ_i sont des matrices diagonales et telles que le conditionnement de $\Delta_1 M \Delta_2$ soit meilleur que celui de M. Ceci est très théorique et il n'existe pas de méthode universelle pour déterminer Δ_1 et Δ_2. Il convient de noter que, si la matrice M est symétrique et s'il faut conserver cette propriété, il convient alors de prendre $\Delta_1 = \Delta_2$.

A.2 Algorithmes de renumérotation

La résolution d'un système linéaire $Mx = b$ par méthode directe est basée sur une factorisation par élimination de Gauss. Pour être plus précis, la matrice M est décomposée en facteurs triangulaires LU ou LDL^T en fonction de ses propriétés numériques et structurelles (positivité, densité, symétrie, ...). La solution x est alors obtenue par des phases de descente et remontée sur ces facteurs. Pour réduire le coût de la factorisation, les méthodes directes en algèbre linéaire creuse ne calculent que les valeurs non nulles. De même, uniquement les valeurs non nulles sont stockées pour réduire les besoins en mémoire. Généralement, les facteurs L et U sont plus denses que la matrice originale M à cause d'un phénomène appelé **remplissage** ("fill-in" en anglais) qui survient pendant la factorisation.

Ce phénomène de remplissage implique une occupation mémoire plus importante pour permettre le stockage de nouveaux coefficients dans L et U. De plus, l'apparition de ces éléments augmente le nombre d'opérations effectuées pendant la décompositon. Ainsi, de nombreux travaux ont été effectués pour réduire les effets du remplissage en permutant les inconnues du système linéaire. Ces techniques sont connues sous le nom d'algorithmes de renumérotation et elles s'expriment généralement comme des opérations sur des graphes décrivant la structure creuse de la matrice M. Un graphe est défini par un ensemble de sommets et d'arêtes. Ses sommets sont les numéros de colonne, c'est-à-dire les numéros d'équations du système, et ses arêtes représentent les coefficients non nuls de M.

Dans les sous-sections suivantes, nous allons donner une présentation des méthodes populaires de renumérotation symétrique qui peuvent aussi être appliquées à des structures non symétriques : l'algorithme de **dissection emboîtée**, les variantes basées sur l'algorithme de **degré minimum** et l'**algorithme de Cuthill - McKee**.

A.2.1 L'algorithme de dissection emboîtée

L'algorithme de dissection emboîtée (George 1973; George et Liu 1981) repose sur le principe de "diviser pour mieux régner" ("divide and conquer"). Il considère que la matrice initiale M est irréductible (c'est-à-dire le système ne peut se décomposer en deux sous-systèmes indépendants) et qu'elle est donc associée à un graphe connexe G. Le but de la dissection emboîtée est de faire interagir le moins possible des parties relativement indépendantes de M. Pour cela, en partant du graphe initial G, on cherche un séparateur qui permettra la décomposition du graphe en deux sous-graphes G_1 et G_2 qui pourront être traités en parallèle. La procédure est appliquée récursivement aux sous-graphes. Le parallélisme mis en évidence est tributaire de la taille des séparateurs ; plus ils sont petits, meilleur est le parallélisme.

A.2.2 L'algorithme de degré minimum et ses variantes

Le principe de l'algorithme de degré minimum ou "minimum degree" (Tinney et Walker 1967; George et Liu 1989) consiste à sélectionner à chaque étape, le sommet ayant le plus petit degré, c'est-à-dire le nombre d'arêtes qui lui sont incidentes, puis de l'éliminer du graphe en formant une clique avec ses voisins. Une illustration de l'apparition d'une clique est donnée sur la figure A.3 par les arêtes en tirets bleus.

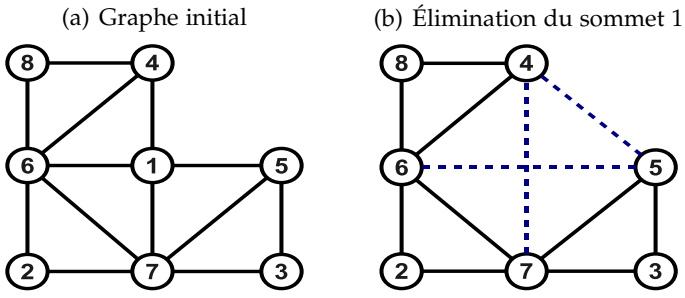


FIG. A.3 – *Apparition d'une clique par élimination d'un sommet.*

Chaque sommet sélectionné est renuméroté avec le plus petit numéro encore disponible. Le but de cette stratégie est de mettre en premier les inconnues du système qui ont potentiellement le moins de contributions à apporter au remplissage. En voici une description algorithmique simplifiée mais essentielle :

Algorithme A.3 : Description algorithmique du degré minimum

1 former le graphe initial associé à la matrice creuse à factoriser;
2 calculer le nombre de voisins de chaque sommet (son degré);
3 numéroter en premier (puis éliminer du graphe) le sommet ayant le moins de voisins (degré minimum);
4 remettre à jour le graphe de l'étape 1;
5 retourner à l'étape 2.

Au cours du déroulement de l'algorithme, on constate cependant que le degré d'un voisin du sommet à éliminer peut augmenter. Ce phénomène rend cette méthode très séquentielle, puisqu'on ne peut pas sélectionner à l'avance les sommets suivants. Cette heuristique est aussi un algorithme local puisqu'on ne s'intéresse qu'à un sommet et à ses voisins. Les arbres d'élimination ainsi produits comportent de nombreuses chaînes et ne sont donc que peu équilibrés et il en résulte généralement un mauvais parallélisme, ce qui peut être pénalisant dans la phase numérique.

Plusieurs variantes ont été proposées à l'algorithme de degré minimum. Elles ont conduit à l'amélioration de son temps d'exécution. Des exemples peuvent être donnés tels que MMD (Multiple Minimum Degree, voir Liu (1985)) ou AMD (Approximate Minimum Degree, voir Amestoy et al. (1996)). Bien que ces variantes ne produisent pas la même séquence d'élimination, elles sont généralement de qualité comparable. La qualité des renumérotations produites par ces variantes ainsi que leur efficacité en temps d'exécution font qu'elles sont très populaires. Toutefois, leur comportement théorique n'est pas encore bien compris et leurs performances dans le pire des cas peuvent être très éloignées de l'optimal. Enfin, il existe des variantes qui se basent sur la quantité de remplissage générée par l'élimination d'un sommet pour la construction de la permutation. Ces méthodes restent aussi purement locales et sont connues sous le nom de remplissage minimal ou "minimum fill" (en anglais) (Ng et Raghavan 1999). Il est à noter que la permutation produite par ce genre d'approche est complètement différente de celle qui peut être obtenue par l'algorithme de degré minimum.

A.2.3 L'algorithme de Cuthill - McKee

L'algorithme de Cuthill - McKee (Cuthill et McKee 1969; Liu et Sherman 1975; George et Liu 1981) est l'une des techniques de renumérotation les plus couramment utilisées. Son objectif principal est de réduire la largeur de bande (c'est-à-dire la distance maximale entre deux sommets adjacents) d'une matrice symétrique creuse en renumérotant les sommets du graphe associé. Cet algorithme est souvent efficace pour réduire la mémoire nécessaire au stockage en ligne de ciel ("skyline" en anglais) de la matrice

à factoriser. Sa procédure s'inspire de l'observation suivante. Soient y un sommet étiqueté et z un voisin sans étiquette. Afin de minimiser la largeur de bande de la ligne associée à z, il est évident que le sommet z doit être renuméroté dès que possible après y. La figure A.4 illustre ce point.

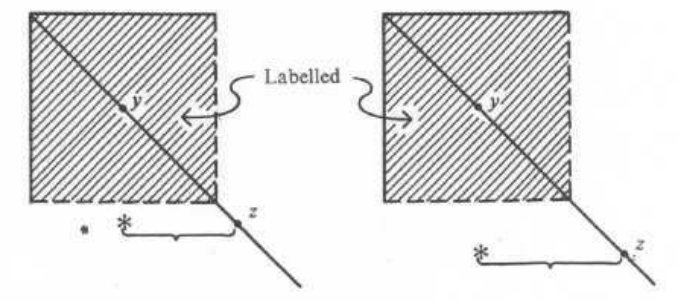


FIG. A.4 – *Effet de la numérotation d'un sommet sur la largeur de bande.*

La procédure de Cuthill - McKee réduit la largeur de bande d'une matrice $M = (m_{ij})$ via une minimisation locale des mesures $\beta_i(M)$ qui sont définies par les formules suivantes :

$$\begin{cases} f_i(M) = min\{j/m_{ij} \neq 0\} \\ \\ \beta_i(M) = i - f_i(M) \end{cases} . \tag{A.6}$$

Des études faites sur cette méthode de réduction des profils ont montré que la renumérotation obtenue en inversant l'ordre des sommets produit souvent un meilleur résultat. Cet algorithme est appelé l'**algorithme de Cuthill - McKee inverse** et il est décrit par A.4. La qualité des profils obtenus avec cet algorithme est conditionnée de manière substantielle par le choix du premier sommet.

Algorithme A.4 : Algorithme de Cuthill - McKee inverse

Choisir un premier nœud r et l'attribuer au sommet x_1.
pour $i = 1$ *à* n *(nombre de sommets du graphe)* **faire**
  trouver tous les voisins de x_i non encore numérotés;
  les numéroter dans l'ordre des degrés croissants.
fin
Effectuer la numérotation inverse :
pour $i = 1$ *à* n **faire**
  faire $y_i = x_{n-i+1}$.
fin

A.3 Présentation de la méthode des éléments finis

La méthode des éléments finis (MEF) (Dhatt et al. 2005) est utilisée pour résoudre numériquement des équations aux dérivées partielles (EDP) provenant de la modélisation de phénomènes physiques très variés notamment en mécanique des structures. Comme pour toute méthode de discrétisation, son objectif est le calcul des valeurs approchées du champ

étudié (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. La mise en œuvre de la MEF comprend les étapes suivantes :

1. analyse mathématique du problème continu avec, en particulier, la nécessité de mettre sous forme variationnelle le système d'EDP à résoudre ;
2. "triangulation" du domaine physique (construction d'un maillage) vérifiant certaines propriétés ;
3. définition des éléments finis et fonctions de base dont le choix est tel que la matrice de discrétisation construite après sera la plus creuse possible ;
4. assemblage de la matrice et du second membre à partir des contributions de chaque élément fini, avec prise en compte des conditions aux limites ;
5. résolution du système.

A.3.1 Le problème modèle

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

$$\begin{cases} -\Delta u = f \ \text{ dans } \ \Omega \\ u = 0 \ \text{ sur } \ \Gamma \end{cases} \tag{A.7}$$

où Δ est l'opérateur de Laplace défini en dimension 2 par : $\Delta u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$. La condition limite $u = 0$ sur la frontière Γ est appelée condition de Dirichlet homogène, elle signifie que le bord du domaine Ω est fixé. Lorsque f appartient à $L^2(\Omega)$, c'est-à-dire l'ensemble des fonctions de carré intégrable sommable sur Ω, alors le système d'EDP (A.7) a une unique solution u dans l'espace $V = {H_0}^1(\Omega)$ où ${H_0}^1(\Omega)$ est l'ensemble des fonctions de $L^2(\Omega)$ dont les dérivées sont dans $L^2(\Omega)$ et dont la trace est nulle sur Γ. Pour une analyse mathématique plus détaillée du problème, nous renvoyons à Raviart et Thomas (1998).

A.3.2 La mise sous forme variationnelle d'un problème d'EDP

Pour résoudre le problème par la MEF, on commence par écrire la formulation variationnelle correspondant au système (A.7). On multiplie la première équation de ce système par une fonction test $v \in V$ et on intègre par parties sur Ω. On obtient :

$$\int_\Omega \nabla u.\nabla v dx = \int_\Omega f v dx - \int_\Gamma \frac{\partial u}{\partial n}.v dn \tag{A.8}$$

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

$$\nabla u = \begin{pmatrix} \dfrac{\partial u}{\partial x} \\ \\ \dfrac{\partial u}{\partial y} \end{pmatrix}. \tag{A.9}$$

Etant donné que v appartient à ${H_0}^1(\Omega)$, alors v est nulle sur la frontière. Notre formulation variationnelle se réduit donc à :

$$\int_\Omega \nabla u.\nabla v dx = \int_\Omega f v dx. \tag{A.10}$$

La MEF utilise la formulation (A.10) pour déterminer des approximations de la solution u. Pour cela, on construit un sous-espace V_h de dimension finie inclus dans l'espace ${H_0}^1(\Omega)$, puis on cherche $u_h \in V_h$, solution approchée de u, comme solution de :

$$\int_\Omega \nabla u_h.\nabla v_h dx = \int_\Omega f v_h dx, \quad \forall v_h \in V_h. \tag{A.11}$$

A.3.3 Le choix d'un maillage et discrétisation

Un maillage d'un domaine Ω, noté $\mathcal{M}_h$, est un ensemble de N_{elem} éléments géométriques "simples" (également appélés 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 T_k et le domaine recouvert Ω_h est tel que $\Omega_h = \bigcup_{k=1}^{N_{elem}} T_k$.

Choix d'un maillage

Le choix de la géométrie des éléments dépend de la géométrie du domaine Ω et de la nature du problème à résoudre. En $2D$, ce sont par exemple des triangles ou des rectangles (respectivement des tétraèdres ou des parallélépipèdes en $3D$). 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 $3D$).

D'autres propriétés géométriques sont à vérifier pour que la MEF fournisse de bonnes approximations de la solution u. 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 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 dans certaines zones du domaine pour obtenir des solutions avec suffisamment de précision.

Fonctions de base

Le maillage du domaine physique n'est qu'un support géométrique pour la MEF. 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 $\mathcal{T}_h$.

Définition A.1 *Soit :*

- *une partie compacte T de $\mathbb{R}^2$, connexe et d'intérieur non vide (triangle, quadrangle, etc) ;*
- *un ensemble fini E^T 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 lorque, étant donné n scalaires réels quelconques α_j, il existe une unique fonction $p \in P^T$ telle que, en tout nœud a_j de E^T : $p(a_j) = \alpha_j,\ \forall j \in \{1, 2,\ ..., n\}$.

La fonction p peut également s'écrire $p(a_j) = \sum_{i=1}^{n} \alpha_i p_i(a_j)$, pour tout nœud a_j de E^T avec les fonctions $p_i,\ i = 1,\ ..., n$ vérifiant la condition :

$$p_i(a_j) = \begin{cases} 1 \text{ si } i = j \\ 0 \text{ si } i \neq j \\ \forall j = 1,\ ..., n \end{cases} \tag{A.12}$$

Les fonctions p_i 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 la présentation, regardons ce que signifient ces relations dans le cas du maillage bidimensionnel $\mathcal{M}_h$ constitué de sommets et d'éléments triangulaires. Nous définissons une triangulation $\mathcal{T}_h$ sur ce maillage en choisissant d'utiliser des éléments de type P_1, c'est-à-dire les éléments finis de Lagrange de degré un. 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 trois nœuds chaque nœud coïncidant avec les sommets du triangle T
- $P^T = P_1$, c'est-à-dire l'espace des polynômes de degré inférieur ou égal à un en x et y.

Il convient remarquer que les 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 ou des éléments dont les nœuds sont pris sur les arêtes ou même à l'intérieur de l'élément géométrique.

Les définitions que nous venons d'introduire sont généralisables à toute la triangulation $\mathcal{T}_h$. Lorsque l'on travaille avec la triangulation com-

plète, l'union des ensembles E^T associés à chacun des éléments de la triangulation forme un ensemble E de N nœuds a_j distincts définis sur $\mathcal{T}_h$. Les nœuds de E sont appelés degrés de liberté ; ce sont les inconnues de notre problème discret. A chaque nœud correspond une fonction de base continue, que nous allons noter à nouveau p_i vérifiant la condition :

$$p_i(a_j) = \begin{cases} 1 \text{ si } i = j \\ 0 \text{ si } i \neq j \\ \forall j = 1, \ ..., N \end{cases} \tag{A.13}$$

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

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 (déplacement u). Pour certains problèmes mécaniques tridimensionnels, par exemple,six degrés de liberté sont nécessaires pour définir les rotations et translations en X, Y et Z.

A.3.4 Le problème sous forme matricielle

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 des polynômes de degré inférieur ou égal à un. La dimension de l'espace V_h est égale au nombre total d'inconnues associées au maillage $\mathcal{T}_h$, les fonctions de V_h étant entièrement déterminées par leurs valeurs en chacun des nœuds de ces éléments. Dans cet espace, une approximation de la solution de (A.10) s'écrit donc :

$$u_h = \sum_{i=1}^{N} u_i p_i, \tag{A.14}$$

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 (A.10) s'écrit :

$$\int_{\Omega_h} \nabla u_h . \nabla p_j dx = \int_{\Omega_h} f p_j dx, \ \forall p_j \in V_h. \tag{A.15}$$

En utilisant l'écriture (A.14) de u_h dans V_h, l'équation (A.15) devient :

$$\sum_{i=1}^{N} \int_{\Omega_h} \nabla pi . \nabla p_j dx = \int_{\Omega_h} f p_j dx, \ \forall p_i, p_j \in V_h. \tag{A.16}$$

Le problème (A.16) est équivalent à la résolution d'un système linéaire :

$$AU = F, \tag{A.17}$$

où A est une matrice de dimension N dont les coefficients sont :

$$A_{ij} = \int_{\Omega_h} \nabla p_i . \nabla p_j dx, \ \ \forall (i,j) \in \{1, 2, \ ..., \ N\}^2, \tag{A.18}$$

U est le vecteur solution $u_1, u_2, \ ..., u_N$, et F est un vecteur de taille N ayant pour composantes :

$$F_j = \int_{\Omega_h} f p_j dx, \ \ \forall j \in \{1, 2, \ ..., \ N\}. \tag{A.19}$$

Le calcul des coefficients A_{ij} de la matrice A se fait en parcourant tous les éléments du maillage. La contribution $A_{ij}(T_k)$ de l'élément T_k au coefficient A_{ij} s'écrit :

$$A_{ij}(T_k) = \int_{T_k} \nabla p_i . \nabla p_j dx. \tag{A.20}$$

Les coefficients A_{ij} sont donc donnés par :

$$A_{ij} = \sum_{k=1}^{N_{elem}} A_{ij}(T_k). \tag{A.21}$$

La matrice A, dite matrice de rigidité, est creuse au sens où la plupart de ses coefficients sont nuls. En effet, $A_{ij}(T_k) \neq 0$ si et seulement si a_i et a_j sont des nœuds du triangle T_k.

A.4 Analyse des performances des versions multi-threads du solveur Dissection

Dans cette partie, nous complétons l'évaluation et l'analyse des performances de la version multi-threads avec des threads POSIX et celle optimisée avec OpenMP sur une machine bi-processeurs quadri-cœurs. Nous rappelons qu'elle offre la possibilité d'exécuter jusqu'à huit tâches en parallèles. Nous avons réalisé un autre test sur un problème d'élasticité linéaire en $3D$ ayant **206763** degrés de liberté.

A.4.1 Les temps de calcul pour la phase d'analyse

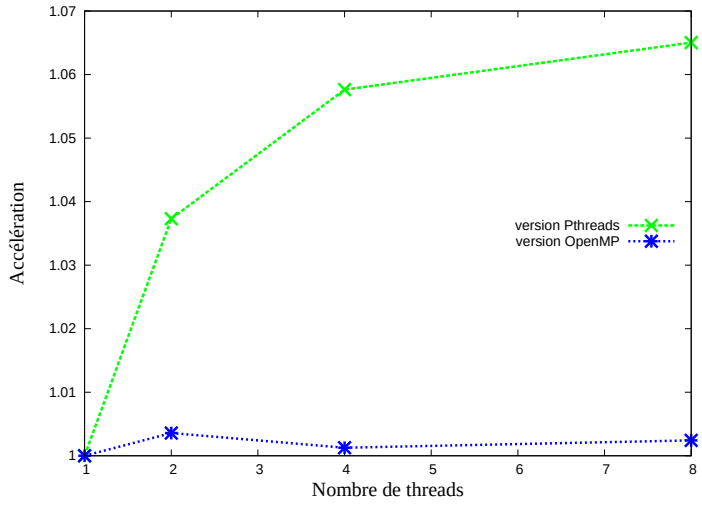


Fig. A.5 – *Temps d'analyse du système en fonction du nombre de threads.*

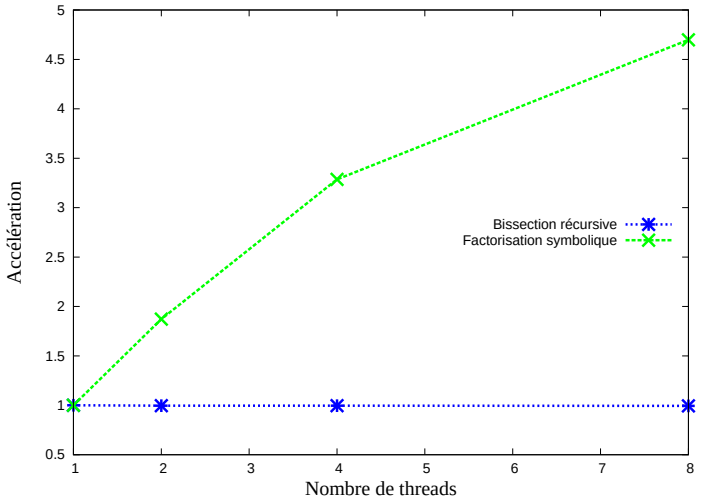


FIG. A.6 – *Comparaison des étapes d'analyse du système.*

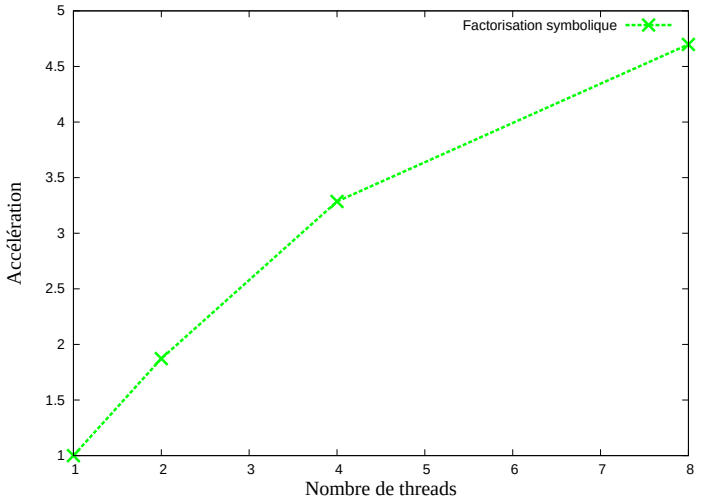


FIG. A.7 – *Temps de factorisation symbolique en fonction du nombre de threads.*

A.4.2 Les temps de calcul pour la phase numérique

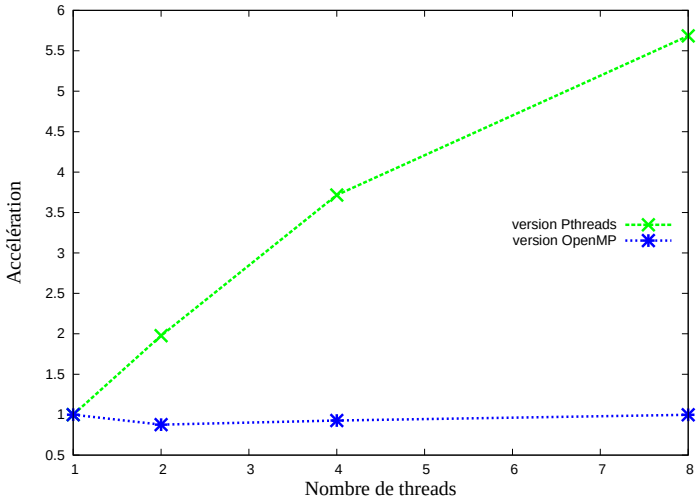


FIG. A.8 – *Temps de factorisation locale en fonction du nombre de threads.*

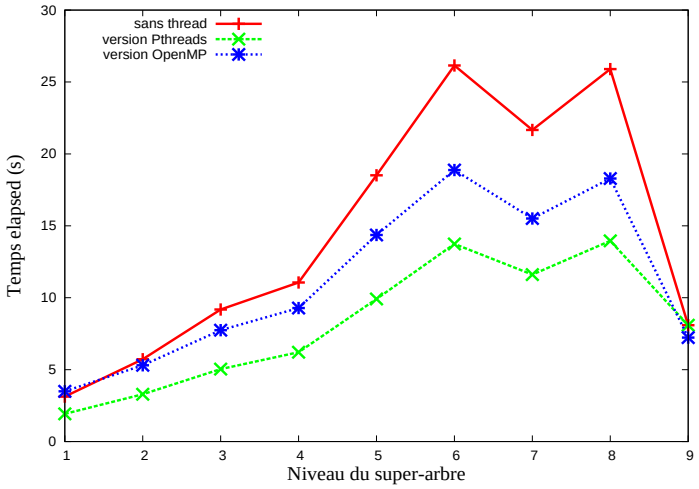


FIG. A.9 – *Temps de factorisation de Schur sur deux cœurs.*

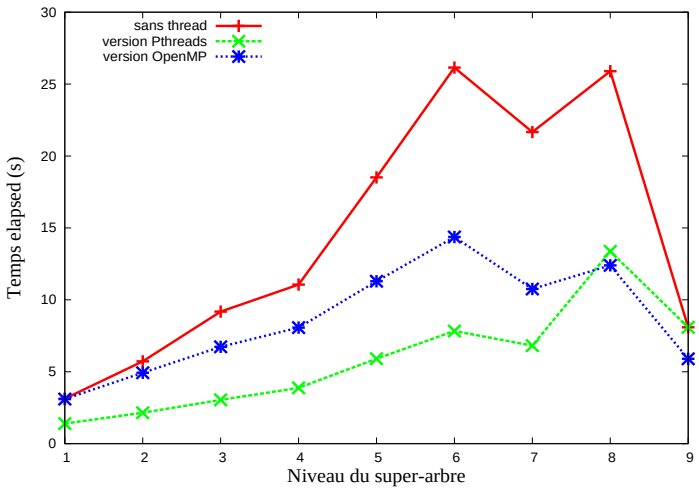


FIG. A.10 – *Temps de factorisation de Schur sur quatre cœurs.*

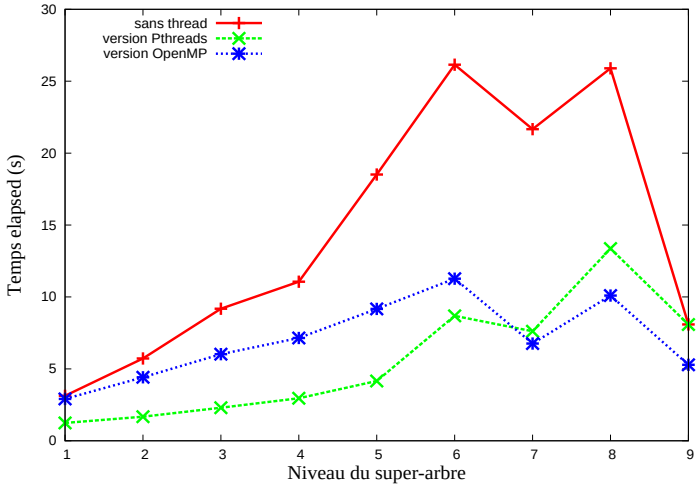


FIG. A.11 – *Temps de factorisation de Schur sur huit cœurs.*

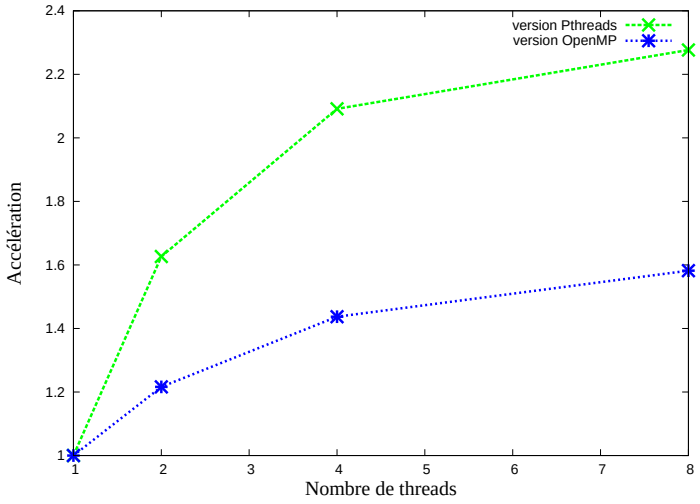


FIG. A.12 – *Temps d'exécution de Schur en fonction du nombre de threads.*

A.4.3 Les temps de calcul pour la phase de descente-remontée

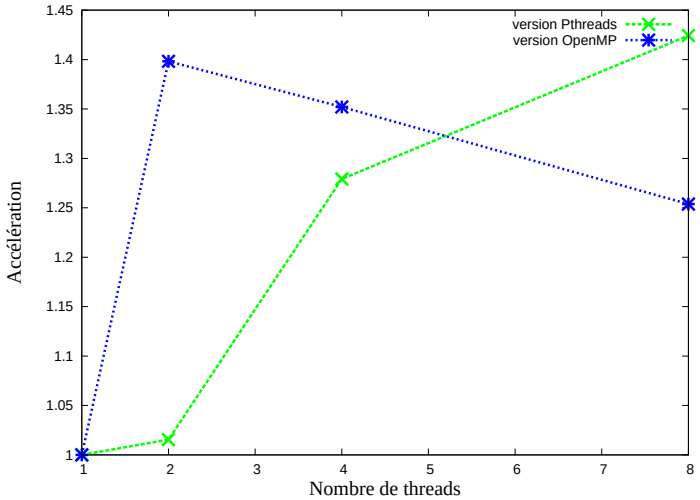


FIG. A.13 – *Temps de descente-remontée en fonction du nombre de threads.*

A.4.4 Les temps d'exécution en fonction du nombre de threads

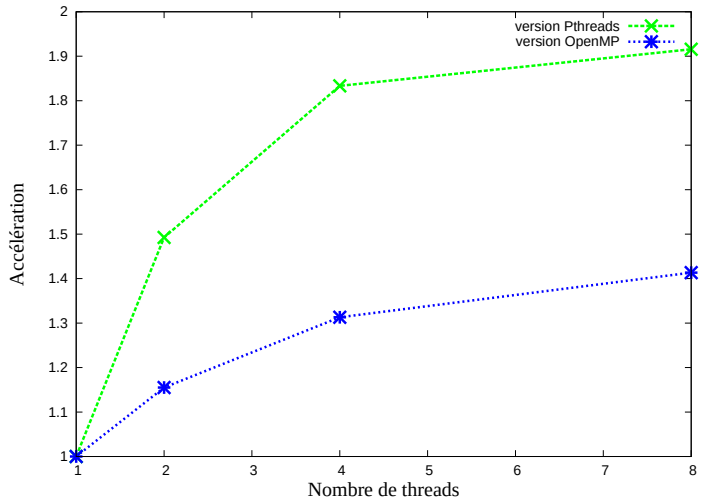


FIG. A.14 – *Temps d'exécution en fonction du nombre de threads.*

A.4.5 Les conclusions

Les résultats présentés dans la phase d'analyse confirment bien les remarques faites à la partie 2.3.3 sur le comportement de la version OpenMP et les bonnes performances obtenues dans la factorisation symbolique. Dans la phase numérique, les conclusions sont les mêmes, c'est-à-dire que la version Pthreads exploite mieux le multi-threading aux niveaux de la factorisation où il y a un très grand nombre de super-nœuds à traiter et que la version OpenMP permet quant à elle d'exploiter plus le parallélisme aux plus haut niveaux du super-arbre d'élimination. L'autre remarque confirmée est que la version Pthreads est plus performante quand nous exécutons le complément de Schur. Contrairement aux deux phases précédentes, nous remarquons dans celle de descente-remontée qu'il n'y a pas de tendance favorable qui se dégage entre la version avec Pthreads et celle avec OpenMP. D'ailleurs, dans cette phase, c'est ce comportement qui a rendu difficile le choix sur quelle version il faut basculer dans la version hybride. Mais nous avons finalement opté pour la version avec les threads POSIX qui est plus stable. Les temps d'exécution obtenus avec le problème de taille **206763** sont moins réduits. Ceci est dû au fait qu'il a des super-nœuds de tailles plus petites.

Bibliographie

S. Akhter et J. Roberts. *Multi-Core Programming : Increasing Performance Through Software Multithreading*. Intel Press, Santa Clara, CA, 2006.

P. Amestoy, T. A. Davis, et I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17 :886–905, 1996.

P. R. Amestoy, I. S. Duff, et J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Computer Methods in Applied Mechanics and Engineering*, 184 :501–520, 2000.

C. Ashcraft et R. Grimes. SPOOLES : An object-oriented sparse matrix library. *Proceedings of the 9th SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

M. Bhardwaj, D. Day, C. Farhat, M. Lesoinne, K. Pierson, et D. Rixen. Application of the FETI method to ASCI problems - scalability results on one thousand processors and discussion of highly heterogeneous problems. *International Journal for Numerical Methods in Engineering*, 47 : 513–535, 2000.

T. N. Bui et B.-R. Moon. Genetic algorithm and graph partitioning. *IEEE Transactions on Computers*, 45(7) :841–855, 1996.

R. Chandra, R. Menon, L. Dagum, D. Kohr, D. Maydan, et J. McDonald. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2000.

P. Charrier et J. Roman. Algorithmique et calculs de complexité pour un solveur de type dissections emboîtées. *Numerische Mathematik*, 55 :463–476, 1989.

Prescott D. Crout. A Short Method for Evaluating Determinants and Solving Systems of Linear Equations With Real or Complex Coefficients. *AIEE Transactions*, 60 :1235–1240, 1941.

E. Cuthill et J. McKee. Reducing the bandwidth of sparse symmetric matrices. *Proceedings of the 24th National Conference*, pages 157–172. ACM Press, New York, NY, USA, 1969.

J. W. Demmel, J. R. Gilbert, et X. S. Li. An asynchronous parallel supernodal algorithm for sparse Gaussian elimination. *SIAM Journal on Matrix Analysis and Applications*, 20(4) :915–952, 1999.

G. Dhatt, G. Touzot, et E. Lefrançois. *Méthode des éléments finis*. Hermès - Lavoisier, 2005.

J. J. Dongarra, I. S. Duff, J. Du Croz, et S. Hammarling. A set of Level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16 :1–17, 1990.

I. S. Duff et J. K. Reid. An implementation of Tarjan's algorithm for the block triangularization of a matrix. *ACM Transactions on Mathematical Software*, 4(2) :137–147, 1978.

D. Dureisseix et P. Ladevèze. A multi-level and mixed domain decomposition approach for structural analysis. *Contemporary Mathematics*, 218 : 246–253, 1998.

C. Farhat, P. S. Chen, et J. Mandel. A scalable Lagrange multiplier based domain decomposition method for implicit time-dependent problems. *International Journal for Numerical Methods in Engineering*, 38 :3831–3858, 1995.

C. Farhat, P. S. Chen, J. Mandel, et F.-X. Roux. The two-level FETI method - Part II : extension to shell problems, parallel implementation and performance results. *Computer Methods in Applied Mechanics and Engineering*, 155 :153–180, 1998.

C. Farhat, L. Crivelli, et F.-X. Roux. Extending substructure based iterative solvers to multiple load and repeated analyses. *Computer Methods in Applied Mechanics and Engineering*, 117 :195–200, 1994a.

C. Farhat, M. Lesoinne, P. Le Tallec, K. Pierson, et D. Rixen. FETI-DP : a dual-primal unified FETI method - part I : A faster alternative to the two-level FETI method. *International Journal for Numerical Methods in Engineering*, 50 :1523–1544, 2001.

C. Farhat, A. Macedo, M. Lesoinne, F.-X. Roux, F. Magoulès, et A. de La Bourdonnaye. Two-level domain decomposition methods with Lagrange multipliers for the fast iterative solution of acoustic scattering problems. *Computer Methods in Applied Mechanics and Engineering*, 184 : 213–239, 2000a.

C. Farhat et J. Mandel. The two-level FETI method for static and dynamic plate problems Part I : An optimal iterative solver for biharmonic systems. *Computer Methods in Applied Mechanics and Engineering*, 155 : 129–151, 1998.

C. Farhat, J. Mandel, et F.-X. Roux. Optimal convergence properties of the FETI domain decomposition method. *Computer Methods in Applied Mechanics and Engineering*, 115 :367–388, 1994b.

C. Farhat, K. Pierson, et M. Lesoinne. The second generation FETI methods and their application to the parallel solution of large-scale linear and geometrically non-linear structural analysis problems. *Computer Methods in Applied Mechanics and Engineering*, 184(2-4) :333–374, 2000b.

C. Farhat et F.-X. Roux. A method of finite element tearing and interconnecting and its parallel solution algorithm. *International Journal for Numerical Methods in Engineering*, 32 :1205–1227, 1991.

C. Farhat et F.-X. Roux. Implicit parallel processing in structural mechanics. *Computational Mechanics Advances*, 2(1) :1–124, 1994.

F. Feyel. *Application du calcul parallèle aux modèles à grands nombre de variables internes*. Thèse de doctorat, École Nationale Supérieure des Mines de Paris, 1998.

A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2) :345–363, 1973.

A. George et J. W.-H. Liu. *Computer solution of large sparse positive definite systems*. Prentice Hall, 1981.

A. George et J. W.-H. Liu. The evolution of the Minimum Degree Ordering Algorithm. *SIAM Review*, 31 :1–19, 1989.

M. Géradin, D. Coulon, et J.-P. Delsemme. Parallelization of the Samcef Finite Element Software Through Domain Decomposition and FETI Algorithm. *International Journal of High Performance Computing Applications*, 11(4) :286–298, 1997.

P. Gosselet et C. Rey. Non-overlapping domain decomposition methods in structural mechanics. *Archives of Computational Methods in Engineering*, 13(4) :515–572, 2006.

I. Guèye, X. Juvigny, F. Feyel, F.-X Roux, et G. Cailletaud. Mise en œuvre d'un solveur direct parallèle pour l'inversion des problèmes locaux au sein d'une méthode de décomposition de domaine. *9ème Colloque National en Calcul de Structures, Giens (France)*. Hal - ccsd : `http://hal.archives-ouvertes.fr/docs/00/39/95/57/PDF/gueye-cailletaud-Giens_09.pdf`, 2009.

I. Guèye, X. Juvigny, F.-X Roux, F. Feyel, et G. Cailletaud. Analyse et développement d'algorithmes parallèles pour la résolution directe de grands systèmes linéaires creux. *18ème Congrès Français de Mécanique*. AFM, Maison de la Mécanique, Courbevoie (France), 2007.

A. Gupta, G. Karypis, et V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. Rep. tech., IEEE Transactions on Parallel and Distributed Systems, 1994.

M. Heath, E. G. Ng, et B. Peyton. Parallel algorithms for sparse linear systems. *SIAM Review*, 33 :420–460, 1991.

P. Hénon, P. Ramet, et J. Roman. PaStiX : A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2) :301–321, 2002.

Intel. Tera-scale Computing Research Program. `http://techresearch.intel.com`, 2006.

B. M. Irons. A frontal solution program for finite element analysis. *International Journal for Numerical Methods in Engineering*, 2 :5–32, 1970.

X. Juvigny. *Résolution de grands systèmes linéaires sur machines massivement parallèles*. Thèse de doctorat, Université Paris VI, 1997.

G. Karypis et V. Kumar. METIS : Unstructured graph partitioning and sparse matrix ordering system. *http://www-users.cs.umn.edu/~karypis/metis*, 1995.

G. Karypis et V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graph. *SIAM Journal on Scientific Computing*, 20 (1) :359–392, 1999.

P. Lascaux et R. Théodor. *Analyse numérique matricielle appliquée à l'art de l'ingénieur*, volume 1. Dunod, 2004a.

P. Lascaux et R. Théodor. *Analyse numérique matricielle appliquée à l'art de l'ingénieur*, volume 2. Dunod, 2004b.

P. Le Tallec. Domain decomposition methods in computational mechanics. *Computational Mechanics Advances*, 1(2) :121–220, 1994.

B. Lewis et D. J. Berg. *Multithreaded Programming with Pthreads*. Prentice Hall, 1998.

X. S. Li et J. W. Demmel. SuperLU DIST : A scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Mathematical Software*, 29(2) :110–140, 2003.

P. L. Lions. On the Schwarz Alternating Method I. *First International Symposium on Domain Decomposition Methods*, pages 1–42. In Roland Glowinski, Gene H. Golub, Gerard A. Meurant, and Jacques Periaux, editors, SIAM, Philadelphia, PA, 1988.

J. W.-H. Liu. Modification of the Minimum-Degree algorithm by multiple elimination. *ACM Transactions on Mathematical Software*, 11(2) :141–153, 1985.

J. W.-H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal of Matrix Analysis and Applications*, 11(1) :132–172, 1990.

J. W.-H. Liu et A. Sherman. Comparative analysis of the Cuthill - McKee and the reverse Cuthill - McKee ordering algorithms for sparse matrices. *SIAM Journal on Numerical Analysis*, 13(2) :198–213, 1975.

J. Mandel. Balancing domain decomposition. *Communication in Applied Numerical Methods*, 9 :233–241, 1993.

E. G. Ng et P. Raghavan. Performance of greedy heuristics for sparse cholesky factorization. *SIAM Journal on Matrix Analysis and Applications*, 20 :902–914, 1999.

P. Raghavan. Page d'accueil DSCPack. *http://www.cse.psu.edu/~raghavan/Dscpack*, 2001.

P. Raghavan. DSCPack : Domain Separator Codes for the parallel solution of sparse linear systems. Rep. Tech. CSE-02-004, Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802-6106, 2002.

P. A. Raviart et J. M. Thomas. *Introduction à l'analyse numérique des équations aux dérivées partielles*. Dunod, 1998. Collection Mathématiques appliquées pour la maîtrise.

E. Rothberg. *Exploiting the memory hierarchy in sequential and parallel sparse Cholesky factorization*. Thèse de doctorat, Dept. of Computer Science, Stanford University, 1992.

F.-X. Roux. *Méthode de décomposition de domaine à l'aide de multiplicateurs de Lagrange et application à la résolution parallèle des équations de l'élasticité linéaire*. Thèse de doctorat, Université Paris VI, 1989.

W. F. Tinney et J. W. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55 (11) :1801–1809, 1967.

Notations

Γ_j^l	Séparateur à la position j et situé au niveau l de l'arbre
BLAS	Basic Linear Algebra Subprograms
CPU	Central Processing Unit ou l'Unité Centrale de Traitement
DDR	Double Data Rate
DSCPack	Domain Separator Codes Package
FETI	Finite Element Tearing and Interconnecting
FLOPS	FLoating point Operations Per Second
IRU	Independant Rack Unit
MKL	Math Kernel Library
MMU	Memory Management Unit
MUMPS	MUltifrontal Massively Parallel Sparse direct Solver
MUTEX	MUTual EXclusion
ONERA	Office National d'Études et de Recherches Aérospatiales
OpenMP	Open Multi-Processing
PaStiX	Parallel Sparse matriX package
POSIX	Portable Operating System Interface for UniX
PSPASES	Parallel SPArse Symmetric dirEct Solver
SPOOLES	SParse Object-Oriented Linear Equations Solver

Titre Résolution de grands systèmes linéaires issus de la méthode des éléments finis sur des calculateurs massivement parallèles

Résumé Cette étude consiste à résoudre de grands systèmes linéaires creux sur des calculateurs massivement parallèles. Ces systèmes linéaires, souvent rencontrés lors de la simulation numérique de problèmes de mécanique des structures par des codes de calcul par éléments finis, sont résolus avec des coûts très importants en temps de calcul et en espace mémoire.

Dans cette thèse, nous mettons au point un parallélisme à deux niveaux et l'intégrons dans les méthodes de décomposition de domaine de type FETI. La démarche s'est organisée autour de trois chapitres principaux. Dans un premier temps, nous mettons en œuvre un solveur direct pour inverser des systèmes linéaires creux qui peuvent être symétriques ou non symétriques, réels ou complexes, à second membre simple ou multiple. La mise en œuvre, basée sur une technique de renumérotation de type dissection emboîtée, est complétée par un point utile dans beaucoup de méthodes de décomposition de domaine (construction d'un préconditionneur ou formulation de l'opérateur de FETI) : la détection de modes à énergie nulle des systèmes singuliers. Dans un deuxième temps, nous parallélisons le solveur direct à travers un modèle de parallélisme à mémoire partagée (multi-threading) pour tirer profit des nouveaux processeurs multi-cœurs. Dans un troisième temps, nous intégrons cette version multi-threads du solveur dans les méthodes FETI pour inverser les problèmes locaux en parallèle.

Les résultats de cette étude mettent en évidence l'utilité des travaux effectués et l'intérêt d'utiliser comme solveur local dans les méthodes FETI un solveur direct parallèle robuste et efficace. Tout ceci peut donner accès à de nouvelles gammes de problèmes en calcul des structures.

Il serait intéressant de revoir le parallélisme à gros grains entre sous-domaines dans les méthodes FETI. Cela pourrait consister à utiliser la version du solveur direct à second membre multiple pour améliorer plus la méthode itérative utilisée dans la résolution du problème d'interface.

Mots-clés: Éléments finis ; grands systèmes linéaires ; parallélisme à deux niveaux ; méthode FETI ; solveur direct ; dissection emboîtée ; détection de modes à énergie nulle ; multi-threading.

Title Solving large linear systems arising in finite element approximations on massively parallel computers

Abstract This study is devoted to th resolution of large sparse linear systems on massively parallel computers. The computational effort for these linear systems, often encountered in the numerical simulation of structural mechanics problems by finite element codes, is very significant in terms of runtime and memory requirements.

In this work, we develop a two-level parallelism and integrate it into domain decomposition methods like FETI. The approach is organized around three main chapters. We first implement a direct solver for sparse linear systems which can be symmetric or non-symmetric, real or complex, with single or multiple right-hand sides. The implementation, based on a nested dissection technique, is completed by a useful point in many domain decomposition methods (building a preconditioner or formulation of the FETI operator) : handling of zero-energy modes of singular systems. As a second step, we parallelize the sparse direct solver through a model of shared memory parallelism (multi-threading) to take advantage of the recent multi-core processors. In a third step, we integrate this multi-threads version in FETI methods to solve local problems in parallel.

The results of this study highlight the usefulness of the work and interest to use as local solver in FETI methods a parallel direct solver which is robust and efficient. This can give access to new ranges of complex problems in structural mechanics.

It would be interesting to review the coarse-grained parallelism between subdomains in FETI methods. This could be to use the multiple right-hand sides version of the direct solver to improve the processing of the interface problem.

Keywords: Finite elements ; Large sparse linear systems ; Two-level parallelism ; FETI method ; Direct solver ; Nested dissection ; Handling of zero-energy modes ; Multi-threading.