Optimisations des solveurs linéaires creux hybrides basés sur une approche par complément de Schur et décomposition de domaine

Astrid Casadei

▶ **To cite this version:**

Astrid Casadei. Optimisations des solveurs linéaires creux hybrides basés sur une approche par complément de Schur et décomposition de domaine. Autre [cs.OH]. Université de Bordeaux, 2015. Français. NNT : 2015BORD0186 . tel-01228520

HAL Id: tel-01228520

https://theses.hal.science/tel-01228520

Submitted on 13 Nov 2015

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.

N° d'ordre : XXXX

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Astrid CASADEI**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Optimisations des solveurs linéaires creux hybrides basés sur une approche par complément de Schur et décomposition de domaine

---

Soutenue le : 19 octobre 2015

Après avis des rapporteurs :

Pierre Manneback .	Professeur, Faculté Polytechnique, Univ. de Mons, Belgique
Esmond G. Ng	Directeur de recherche, Lawrence Berkeley Nat. Lab., USA

Devant la commission d'examen composée de :

Frédéric Desprez ...	Directeur de recherche, Inria	Examinateur
Pierre Manneback .	Professeur, Faculté Polytechnique, Univ. de Mons, Belgique	Rapporteur
Esmond G. Ng	Directeur de recherche, Lawrence Berkeley Nat. Lab., USA	Rapporteur
François Pellegrini	Professeur, Université de Bordeaux	Directeur de thèse
Pierre Ramet	Maître de conférences, Université de Bordeaux	Co-encadrant
Jean Roman	Professeur, Inria et Bordeaux INP	Examinateur

– 2015 –

Optimisations des solveurs linéaires creux hybrides basés sur une approche par complément de Schur et décomposition de domaine

Résumé :

Dans cette thèse, nous nous intéressons à la résolution parallèle de grands systèmes linéaires creux. Nous nous focalisons plus particulièrement sur les solveurs linéaires creux hybrides directs-itératifs tels que HIPS, MaPHyS, PDSLIN ou ShyLU, qui sont basés sur une décomposition de domaine et une approche « complément de Schur ». Bien que ces solveurs soient moins coûteux en temps et en mémoire que leurs homologues directs, ils ne sont néanmoins pas exempts de surcoûts. Dans une première partie, nous présentons les différentes méthodes de réduction de la consommation mémoire déjà existantes et en proposons une nouvelle qui n'impacte pas la robustesse numérique du précondionneur construit. Cette technique se base sur une atténuation du pic mémoire par un ordonnancement spécifique des tâches de calcul, d'allocation et de désallocation des blocs, notamment ceux se trouvant dans les parties « couplage » des domaines. Dans une seconde partie, nous nous intéressons à la question de l'équilibrage de la charge que pose la décomposition de domaine pour le calcul parallèle. Ce problème revient à partitionner le graphe d'adjacence de la matrice en autant de parties que de domaines désirés. Nous mettons en évidence le fait que pour avoir un équilibrage correct des temps de calcul lors des phases les plus coûteuses d'un solveur hybride tel que MaPHyS, il faut à la fois équilibrer les domaines en termes de nombre de nœuds et de taille d'interface locale. Jusqu'à aujourd'hui, les partitionneurs de graphes tels que Scotch et MeTiS ne s'intéressaient toutefois qu'au premier critère (la taille des domaines) dans le contexte de la renumérotation des matrices creuses. Nous proposons plusieurs variantes des algorithmes existants afin de prendre également en compte l'équilibrage des interfaces locales. Toutes nos modifications sont implémentées dans le partitionneur Scotch, et nous présentons des résultats sur de grands cas de tests industriels.

Mots clés :

Calcul haute performance, algèbre linéaire creuse, solveur parallèle, méthode hybride directe-itérative, décompostion de domaine, complément de Schur, réduction du pic mémoire, équilibrage de la charge, partitionnement de graphe, bipartitionnement récursif.

Discipline :

Informatique

Equipe-projet commune HiePACS (`https://team.inria.fr/hiepacs/`)
Inria
LaBRI UMR CNRS 5800
Université de Bordeaux
Bordeaux INP

Optimizations of hybrid sparse linear solvers relying on Schur complement and domain decomposition approaches

Abstract :

In this thesis, we focus on the parallel solving of large sparse linear systems. Our main interest is on direct-iterative hybrid solvers such as HIPS, MaPHyS, PDSLIN or ShyLU, which rely on domain decomposition and Schur complement approaches. Althrough these solvers are not as time and space consuming as direct methods, they still suffer from serious overheads. In a first part, we thus present the existing techniques for reducing the memory consumption, and we present a new method which does not impact the numerical robustness of the preconditioner. This technique reduces the memory peak by doing a special scheduling of computation, allocation, and freeing tasks in particular in the Schur coupling blocks of the matrix. In a second part, we focus on the load balancing of the domain decomposition in a parallel context. This problem consists in partitioning the adjacency graph of the matrix in as many domains as desired. We point out that a good load balancing for the most expensive steps of an hybrid solver such as MaPHyS relies on the balancing of both interior nodes and interface nodes of the domains. Through, until now, graph partitioners such as MeTiS or Scotch used to optimize only the first criteria (*i.e.*, the balancing of interior nodes) in the context of sparse matrix ordering. We propose different variations of the existing algorithms to improve the balancing of interface nodes and interior nodes simultaneously. All our changes are implemented in the Scotch partitioner. We present our results on large collection of matrices coming from real industrial cases.

Keywords :

High-performance computing, sparse linear algebra, parallel solver, direct-iterative hybrid method, domain decomposition, Schur complement, memory peak reduction, load balancing, graph partitioning, recursive bipartitioning.

Discipline :

Computer science

Project-team HiePACS (`https://team.inria.fr/hiepacs/`)
Inria
LaBRI UMR CNRS 5800
Université de Bordeaux
Bordeaux INP

Table des matières

Introduction **1**

1 Etat de l'art des méthodes hybrides **5**

1.1 Rappels sur les méthodes directes 7

1.2 Méthode de décomposition de domaine avec complément de Schur 11

1.2.1 Décomposition de domaine et renumérotation 12

1.2.2 Factorisation des intérieurs, calcul d'un préconditionneur et résolution itérative 13

1.2.3 Calcul d'une factorisation approchée d'une matrice 15

1.3 Principaux solveurs de type complément de Schur 18

1.3.1 Le solveur HIPS 19

1.3.2 Le solveur PDSLIN 20

1.3.3 Le solveur ShyLU 21

1.3.4 Le solveur MaPHyS 22

1.4 Positionnement de la thèse 23

2 Réduction de la mémoire dans les méthodes hybrides **25**

2.1 Méthodes de réduction mémoire préexistantes 25

2.1.1 Seuillage numérique des matrices de couplage (HIPS et PDSLIN) 26

2.1.2 Restriction du remplissage du complément de Schur à un « pochoir » (ShyLU) 28

2.1.3 Allocation et libération des matrices de couplage par domaine (HIPS) . . 29

2.2 Optimisation de la gestion des matrices de couplage 30

2.2.1 Variantes fan-in, fan-out, right-looking et left-looking 30

2.2.2 Gestion optimisée de la mémoire 32

2.2.3 Expériences numériques 36

3 Amélioration du partitionnement pour un meilleur équilibrage de la charge **45**

3.1 Algorithme de bipartitionnement récursif 47
3.2 Multiniveau 49
3.3 Raffinement d'un séparateur : l'algorithme de Fiduccia-Mattheyses 53
3.4 Algorithmes de partitionnement 59
3.4.1 Greedy graph growing 60
3.4.2 Double greedy graph growing 64
3.4.3 Halo-first greedy graph growing 69
3.5 Résultats 75
3.5.1 Etude de scalabilité 76
3.5.2 Résultats dans le solveur hybride MaPHyS 78

Conclusion et perspectives **87**

Bibliographie **91**

A Liste des publications **97**

Liste des Figures

1 Exemple de maillage. 1

1.1 Notations pour les algorithmes de factorisation de matrices creuses par blocs denses. 10
1.2 Arbre d'élimination par bloc-colonne. 11
1.3 Décomposition de domaine (visions graphique et matricielle). 12
1.4 Algorithme de construction du préconditionneur. 14
1.5 Extraction des matrices de domaine. 19

2.1 Réduction mémoire par seuillage numérique de W, G et S. 27
2.2 Matrices d'exemple pour la restriction du remplissage de S à un pochoir. 28
2.3 Réduction de la mémoire de S par utilisation d'un pochoir : exemple de coloriage. 28
2.4 Variantes fan-in et fan-out de l'algorithme LU right-looking. 31
2.5 Les trois niveaux de parallélisme exploitables par une approche hybride. 33
2.6 Illustration de la $1^{\text{ère}}$ méthode de réduction de la mémoire du couplage. 35
2.7 Illustration de la 2^{e} méthode de réduction de la mémoire du couplage. 37
2.8 Comparaison des deux méthodes de réduction de la consommation mémoire. . . . 39
2.9 Evolution de la mémoire économisée en fonction du nombre de processus par domaine. 41
2.10 Evolution de la mémoire économisée en fonction du nombre de threads par processus. 43

3.1 Bipartitionnement récursif avec et sans halo. 48
3.2 Méthode de multiniveau. 50
3.3 Heuristique du *heavy-edge matching*. 52
3.4 Déplacement d'un sommet dans l'heuristique de Fiduccia-Mattheyses. 55
3.5 Impact du choix de la graine pour l'heuristique du *greedy graph growing*. 63
3.6 Illustration d'un cas où le *greedy graph growing* modifié marche mal. 64
3.7 Choix de la graine dans l'algorithme DG : 1^{er} contre-exemple. 66
3.8 Choix de la graine dans l'algorithme DG : 2^{e} contre-exemple. 66
3.9 Problème lors de la construction d'un halo reconnecté. 70
3.10 Reconnexion du halo : l'algorithme présenté ne fournit pas un ensemble minimal de sommets. 71
3.11 Comparaison du GG, $GG^{\star}$, DG et HF 72
3.12 Comparaison visuelle des algorithmes GG, DG et HF sur le graphe `mario002`. . 76
3.13 Scalabilité du déséquilibre de l'interface et des intérieurs en fonction du nombre de domaines. 79
3.14 Description d'un des quatre déca-cœurs de la machine *riri*. 80

Liste des Tableaux

2.1 Matrices de test du chapitre 2. 39
2.2 Optimisation mémoire : influence du nombre de processus par domaine (matrice `audi`). 41
2.3 Optimisation mémoire : influence du nombre de processus par domaine (matrice `haltere`). 42
2.4 Optimisation mémoire : influence du nombre de threads par processus. 42

3.1 Matrices de test du chapitre 3. 61
3.2 Résultats de DG en comparaison avec GG. 68
3.3 Résultats de HF en comparaison avec GG. 73
3.4 Comparaison des algorithmes DG et HF. 74
3.5 Etude de scalabilité de GG, DG et HF sur la matrice `Almond`. 77
3.6 Etude de scalabilité de GG, DG et HF sur la matrice `NICE-7`. 77
3.7 Etude de scalabilité de GG, DG et HF sur la matrice `10millions`. 77
3.8 Comparaison des algorithmes d'équilibrage dans MaPHyS (matrice `MHD`). 81
3.9 Comparaison des algorithmes d'équilibrage dans MaPHyS (matrice `inline`). . . 82
3.10 Comparaison des algorithmes d'équilibrage dans MaPHyS (matrice `Almond`). . . 82
3.11 Comparaison des algorithmes d'équilibrage dans MaPHyS (matrice `10millions`). 83

Liste des Algorithmes

1 scalaireLU 7
2 rightLookingLU 9
3 calculRemplissageILU(k) 16
4 calculRemplissageILUT(τ) 18

5 leftLookingLU 32
6 leftRightLookingLU 38
7 fanInRecv 39

8 bipartitionnementRécursif 47
9 bipartitionnementRécursifAvecHalo 49
10 multiniveau 51
11 fiducciaMattheyses 54
12 fiducciaMattheysesModifié 58
13 greedyGraphGrowing 62
14 doubleGreedyGraphGrowing 67
15 construireHaloReconnecté 70
16 haloFirstGreedyGraphGrowing 72

Introduction

La résolution des systèmes linéaires creux est un problème qui apparaît fréquemment dans les simulations numériques de phénomènes physiques. Les domaines d'application sont multiples ; on peut par exemple les utiliser pour étudier l'écoulement de l'air autour d'une aile d'avion, pour étudier la propagation d'un séisme ou d'un tsunami, ou bien encore pour prévoir l'évolution du climat terrestre dans des prochaines décennies. Elles permettent de faire des prévisions pour une situation dont le comportement dépend des lois de la physique, tout en évitant les coûts liés à la réalisation d'une simulation effective sur une maquette ou un prototype onéreux à fabriquer.

Le protocole général pour effectuer une simulation numérique est le suivant. Tout d'abord, la surface ou l'espace étudié est discrétisé en un maillage à n nœuds tel que représenté sur la figure 1. Une modélisation du phénomène physique mis en jeu est ensuite mise en œuvre afin de décrire l'évolution de la grandeur physique que l'on souhaite étudier ; dans le cas d'une aile d'avion, par exemple, cette donnée pourrait être la pression de l'air ou la vitesse locale d'écoulement. L'état à un instant donné t de tous les nœuds du maillage vis-à-vis de cette grandeur est représenté par un vecteur b de taille n ; l'état de ces mêmes nœuds à l'instant suivant $t + 1$ est noté x, et constitue l'inconnue que l'on désire calculer. Dans le cas d'un schéma numérique dit implicite, ces deux vecteurs sont reliés par une matrice A de taille $n \times n$ vérifiant l'équation $Ax = b$ et décrivant les interactions des nœuds avec leurs voisins. Puisque seuls les voisins proches interagissent, la matrice A contient généralement un grand nombre de valeurs nulles, d'où sa dénomination de matrice creuse.

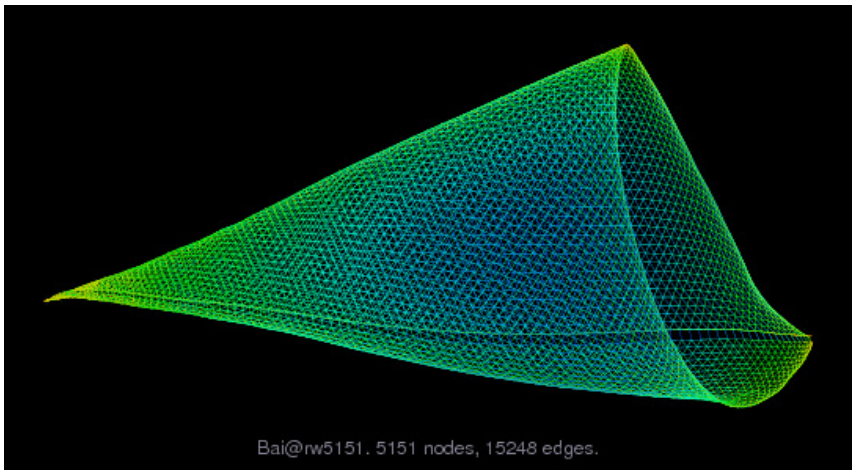


FIGURE 1 – Exemple de maillage, provenant de la matrice `rw5151` de la collection [Dav].

En pratique, de très nombreux problèmes se réduisent sous la forme d'un système linéaire

creux $Ax = b$. Les tailles de ces systèmes peuvent atteindre plusieurs dizaines ou centaines de millions d'inconnues avec un nombre de coefficients non-nuls dépassant le milliard. La résolution demande alors d'utiliser des super-calculateurs, les plus performants d'entre eux pouvant effectuer plus de dix millions de milliards (10^{16}) d'opérations à la seconde grâce aux centaines de milliers de cœurs qui les équipent. Cependant, afin d'exploiter de façon satisfaisante ces derniers, il est nécessaire de développer des algorithmes adaptés qui explicitent de manière précise la division du travail en tâches susceptibles d'être effectuées en parallèle.

Ce sujet ayant fait l'objet d'abondantes recherches au cours des dernières décennies, de nombreuses solutions sont aujourd'hui disponibles. L'état de l'art évoquera notamment les méthodes directes, consistant à factoriser préalablement la matrice A en deux matrices L et U respectivement triangulaire inférieure et supérieure, et les méthodes itératives, reposant sur une suite de vecteurs convergeant vers x. Aucune de ces deux familles de méthodes n'est entièrement satisfaisante ; les premières sont robustes mais coûteuses en termes de temps de calcul et de consommation mémoire. Quant aux secondes, elles souffrent du fait qu'il n'est pas aisé de trouver des suites qui convergent rapidement pour tous les types de problèmes. Une technique pour améliorer leur convergence consiste à trouver une matrice M inversible « bien choisie », appelée *préconditionneur*, telle que le système $M^{-1}Ax = M^{-1}b$ soit mieux conditionné, c'est-à-dire plus facile à résoudre. La question de trouver un bon préconditionneur pour tous les types de problèmes demeure cependant ouverte. Plusieurs méthodes intermédiaires ont donc été développées afin de combiner les avantages des deux extrêmes que constituent les méthodes directes et itératives. Cette thèse se focalise sur l'une d'elle, prometteuse, appelée méthode hybride par décomposition de domaine et basée sur une approche « complément de Schur ».

L'idée générale de ces méthodes hybrides est la suivante. Le maillage de la simulation physique (tel que sur la figure 1) est décomposé en plusieurs sous-ensembles appelés *domaines*. À chaque domaine est associé un système de taille égale au nombre de nœuds qu'il contient. Ces systèmes peuvent être résolus indépendamment les uns des autres avec une méthode directe. Pour trouver la solution du système global, il est ensuite nécessaire de résoudre le système associé à l'ensemble des nœuds qui séparent les domaines les uns des autres, appelé interface. Une méthode itérative préconditionnée est utilisée pour réaliser cette opération.

En dépit du fait que les méthodes hybrides proposent un compromis intéressant entre les méthodes directes et les méthodes itératives, elles ne sont pas dépourvues d'inconvénients. L'un des facteurs limitant leur utilisation est notamment la quantité de mémoire qu'elles requièrent, même si celle-ci est réduite en comparaison avec celle utilisée par un solveur direct. Effectuer un découpage en davantage de domaines n'est plus une solution effective au-delà d'un certain point, car l'interface totale grossit et la méthode itérative a alors davantage de difficultés à converger. D'autre part, le découpage des domaines induit un problème d'équilibrage de la charge, dans la mesure où les différents domaines ont vocation à être traités par différentes unités de calcul. L'objectif de cette thèse est de proposer des solutions face à ces différents écueils.

Dans le premier chapitre, nous détaillons le fonctionnement des méthodes hybrides qui nous intéressent. Les méthodes directes, itératives et autres alternatives y sont abordées afin de mettre en lumière leurs avantages et inconvénients respectifs. Puis, le fonctionnement des méthodes hybrides basées sur une décomposition de domaine et un complément de Schur est détaillé. Les spécificités des différents solveurs existants que sont HIPS, PDSLIN, ShyLU et MaPHyS sont explicitées.

Le chapitre 2 pose le problème de la consommation mémoire. Les solutions qui ont déjà été proposées y sont détaillées. Certaines d'entre elles ne sont applicables que dans certains cas et/ou économisent de la mémoire au détriment de la robustesse numérique du préconditionneur. Nous exposons ensuite notre propre contribution pour la réduction de la mémoire. Celle-ci est basée sur un ordonnancement particulier des tâches d'allocation, de désallocation et de calcul lors de l'étape dite de factorisation des domaines. Des expérimentations permettent de connaître la quantité de mémoire économisée par notre approche.

Le chapitre 3 explore quant à lui l'aspect de l'équilibrage de la charge. Ce problème revient à partitionner le maillage de la simulation en k parties de tailles équivalentes. Nous montrons que pour obtenir un meilleur équilibrage, il est également nécessaire d'équilibrer les k interfaces locales des domaines formés (c'est-à-dire les nœuds de l'interface adjacents à chaque domaine). Les différents algorithmes existants sont ensuite passés en revue et nous proposons des variantes pour chacun afin de tenir compte de ce nouveau critère. Finalement, nous exposons une série de tests afin de valider notre approche.

Enfin, nous résumons les résultats obtenus tout au long de cette thèse et donnons les pistes que nous jugeons intéressant d'approfondir par la suite.

Chapitre 1

Etat de l'art des méthodes hybrides

Sommaire

1.1 Rappels sur les méthodes directes 7
1.2 Méthode de décomposition de domaine avec complément de Schur 11
1.2.1 Décomposition de domaine et renumérotation 12
1.2.2 Factorisation des intérieurs, calcul d'un préconditionneur et résolution itérative 13
1.2.3 Calcul d'une factorisation approchée d'une matrice 15
1.2.3.1 Factorisation incomplète $ILU(k)$ 15
1.2.3.2 Factorisation incomplète $ILUT(\tau)$ 17
1.3 Principaux solveurs de type complément de Schur 18
1.3.1 Le solveur HIPS 19
1.3.2 Le solveur PDSLIN 20
1.3.3 Le solveur ShyLU 21
1.3.4 Le solveur MaPHyS 22
1.4 Positionnement de la thèse 23

La résolution de systèmes linéaires creux de la forme $Ax = b$ est un problème central apparaissant dans de nombreux problèmes physiques de simulation numérique. De multiples méthodes de résolution existent [HNP91] et se classent en plusieurs grandes catégories :

- les méthodes directes [DER86, Gup01, Dav06] : LU, Cholesky, LL^t, Crout LDL^t ;
- les méthodes itératives [Saa03] : Jacobi, Gauss-Seidel, gradient-conjugué, GMRES [SS86], les méthodes multigrilles [Hac85, Bra86, BHM00, HKMR06], etc. ;
- les factorisations incomplètes [Man80, Saa94, NR00, HRR08] ;
- les méthodes hybrides, notamment celles basées sur une décomposition de domaine [TW05] et un complément de Schur (les références associées aux différents solveurs seront données dans la section 1.3).

Dans le cas des **méthodes directes** (par exemple, la méthode LU), l'idée est de trouver deux matrices L triangulaire inférieure à diagonale unitaire et U triangulaire supérieure telles que $A = LU$. Après cette étape de factorisation, le système $Ax = b$ revient à résoudre deux

systèmes linéaires sur des matrices triangulaires :

$$Ax = b \Leftrightarrow \begin{cases} Ly = b \\ Ux = y \end{cases}$$

Cette seconde étape est appelée *descente-remontée*.

Les principaux avantages des méthodes directes sont la précision de la solution (robustesse numérique) et le fait que la matrice factorisée peut servir pour résoudre le système avec différents seconds membres sans surcoût. L'inconvénient de ces méthodes est la quantité de mémoire et de calculs requis.

Les **méthodes itératives** consistent à partir d'une solution initiale x_0 et à construire une suite $(x_i)_{i\in\mathbb{N}}$ qui converge vers la solution du problème. Ces méthodes demandent beaucoup moins de mémoire et de calculs que les méthodes directes ; de plus, lorsque la précision souhaitée n'est pas trop importante, le nombre d'itérations nécessaire est petit. Malheureusement, il est difficile de trouver des méthodes qui convergent correctement dans tous les cas : beaucoup de méthodes itératives sont spécifiques à certains types de problèmes.

Une classe importante des méthodes itératives sont les méthodes multigrilles. Dans le cas où le système est issu d'une méthode d'approximation sur un maillage, ces méthodes consistent à alterner hiérarchiquement des itérations peu coûteuses sur un système simplifié (correspondant à un maillage grossier) et d'autres sur le maillage complet (le plus fin).

Les deux autres méthodes combinent des aspects directs et itératifs.

Dans le cas des **factorisations incomplètes**, l'idée consiste à calculer deux matrices $\widetilde{L}$ et $\widetilde{U}$ qui correspondent à une factorisation directe approchée de la matrice $A \approx \widetilde{L}\widetilde{U}$. Cette factorisation approchée demande moins de calculs et de mémoire qu'une factorisation exacte. On peut ensuite résoudre le système $\widetilde{U}^{-1}\widetilde{L}^{-1}Ax = \widetilde{U}^{-1}\widetilde{L}^{-1}b$ avec une méthode itérative ; la matrice $\widetilde{U}^{-1}\widetilde{L}^{-1}A$ ayant *a priori* un meilleur conditionnement que A, ce système peut converger plus rapidement que $Ax = b$, tout en ayant une solution équivalente. On dit qu'on *préconditionne* le système avec la matrice $M = \widetilde{L}\widetilde{U}$.

Enfin, les **méthodes hybrides** qui nous intéressent dans cette thèse s'appuient sur un découpage du graphe d'adjacence de A à l'aide d'une décomposition de domaine. Le graphe d'adjacence d'une matrice A est le graphe $G = (V, E)$ où V est l'ensemble des sommets associés aux inconnues et E l'ensemble des arêtes associées aux termes non-nuls ; ce graphe peut se déduire du maillage dans le cadre d'une simulation physique. L'idée des méthodes de décomposition de domaine est de découper ce graphe en de multiples parties appelées domaines. Comme les inconnues de l'intérieur de deux domaines distincts ne sont pas couplées, on peut résoudre les sous-systèmes associés à l'intérieur des domaines de manière indépendante. On peut par exemple utiliser une méthode directe pour cette étape. Ensuite, il faut résoudre le système sur l'interface entre les domaines ; on utilise pour cela une méthode itérative préconditionnée, par exemple par une factorisation incomplète.

À ce jour, les méthodes hybrides constituent une alternative particulièrement intéressante car elles permettent de tirer profit des avantages des autres méthodes ; en faisant varier le nombre de

domaines, on peut ajuster la part du système résolue en direct et en itératif, et ainsi augmenter la robustesse du solveur, ou au contraire diminuer celle-ci en faveur de davantage de parallélisme et d'une moindre quantité de calculs et de mémoire nécessaires. La présente thèse se focalisera donc uniquement sur les méthodes hybrides. Dans une première section, ce chapitre fera les rappels nécessaires sur les méthodes directes pour introduire les méthodes hybrides. Puis, nous présenterons les méthodes hybrides basées sur une décomposition de domaine et l'utilisation d'un complément de Schur. Enfin, nous donnerons les détails de mise en œuvre spécifiques aux différents solveurs existants.

1.1 Rappels sur les méthodes directes

Nous présentons ici succinctement les méthodes directes, dont la compréhension est utile à l'explication des méthodes hybrides. De nombreux solveurs de ce type existent, tels que PaStiX [HRR02], SuperLU [LD03, GDL07], PARDISO [SG04] ou MUMPS [ADL98, ADKL01]. Les trois premiers reposent sur une approche dite supernodale et le dernier sur une approche multifrontale ; nous décrirons ici uniquement la première méthode, et renvoyons le lecteur à [Liu92, GKK97, ADKL01, L'E12] pour la seconde.

Dans toute la suite de cette thèse, nous supposerons systématiquement que les matrices ont une structure de non-zéros symétrique (c'est-à-dire $a_{ij} = 0$ si et seulement si $a_{ji} = 0$). De plus, nous considérerons uniquement les factorisations de type LU, où L est triangulaire inférieure à diagonale unitaire et U triangulaire supérieure. Tout ce qui suit s'adapte naturellement aux autres types de factorisation existantes (Cholesky $A = LL^t$, Crout $A = LDL^t$).

L'algorithme 1 expose l'algorithme LU scalaire. La matrice A est écrasée au fur et à mesure par les facteurs L et U et la diagonale unitaire de L n'est pas stockée.

Algorithme 1 : `scalaireLU`

```
   /* Version dense et sur place */
   Entrées : Matrice A = (a_ij) régulière de dimension n
   Sorties : Matrices L et U telles que LU = A
1  pour k = 1 a n faire
2      pour i = k + 1 a n faire
3          a_ik ← a_ik / a_kk ;
4          pour j = k + 1 a n faire
               /* Report des contributions */
5              a_ij ← a_ij − a_ik a_kj ;
```

Une remarque importante est qu'une matrice peut être permutée symétriquement avant d'être factorisée. Si P est une matrice de permutation, le système $Ax = b$ peut ainsi être remplacé par $PAP^t y = Pb$. x s'obtient alors simplement en permutant le vecteur y : on a $y = Px$. Cette opération s'appelle *la renumérotation des inconnues* (ou des colonnes). Elle est équivalente à changer les numéros des sommets dans le graphe d'adjacence de la matrice.

Un fait remarquable est que le nombre de non-zéros de la matrice factorisée varie considérablement avec la matrice P choisie. Plus précisément, on appelle *terme de remplissage* un coefficient (i, j) qui est nul dans la matrice avant la factorisation et non-nul après [1]. Le graphe d'adjacence correspondant à la matrice factorisée LU, comprenant les termes de A auxquels s'ajoutent les termes de remplissage, est appelé *graphe d'élimination*, et est noté $G^\star = (V, E^\star)$. Il peut être calculé grâce au *théorème de caractérisation* [RTL76], qui affirme qu'un terme (i, j) appartient à $G^\star$ si et seulement il existe un chemin de i à j dans G ne passant que par des sommets intermédiaires de numéros plus petits que i et j. Nous verrons au chapitre 3 une heuristique permettant de trouver une permutation P minimisant le nombre de termes de remplissage sur la base de ce théorème. Une fois la permutation choisie, l'algorithme de *factorisation symbolique*, s'appuyant également sur cette propriété, permet de caractériser la structure de $G^\star$ avec une complexité en temps et en espace variant comme $\mathcal{O}(|E^\star|)$.

L'étude de l'indépendance des calculs, en vue de la parallélisation, amène également à la conclusion que la numérotation des inconnues doit être soigneusement choisie. Dans le cadre d'une distribution dite 1D (par colonne), on considère qu'une tâche est associée à la factorisation d'une colonne k, c'est-à-dire correspond à l'exécution des lignes 3-5 de l'algorithme 1. L'algorithme permet alors de déduire que les tâches $k' < k$ impactant sur une tâche k sont celles telles que $a_{k'k}$ est non-nul au moment de la factorisation de k. On peut définir un graphe orienté dont les sommets sont les colonnes de la matrice et dans lequel un arc de i à j signifie que la tâche i doit être réalisée avant la tâche j. En supprimant toutes les arêtes de transitivité, on obtient un arbre, appelé *arbre d'élimination* [Liu90]. La numérotation choisie doit alors maximiser la largeur de cet arbre (pour un maximum de parallélisme) tout en minimisant sa hauteur (afin de réduire le temps total nécessaire à la factorisation).

Enfin, un troisième intérêt de la permutation de la matrice est qu'elle peut donner la possibilité de regrouper les non-zéros dans des blocs denses [GL81]. Cela permet alors d'utiliser la spécification BLAS [LHKK79], une formalisation d'un ensemble de routines bas-niveau pour effectuer les opérations d'algèbre linéaire classiques telles que les multiplications matricielles ou la résolution de systèmes triangulaires. Plusieurs implémentations des BLAS existent, permettant d'utiliser efficacement le micro-parallélisme des architectures pour les opérations sur les blocs denses. Afin de pouvoir utiliser ces routines, le stockage de la partie triangulaire inférieure de la matrice A est effectué sous la forme d'une partition de blocs de colonnes consécutives ayant la même structure de non-zéros (un exemple est donné en figure 1.2 à droite). Au sein d'un bloc de colonnes, les non-zéros sont regroupés à l'intérieur de blocs traités en tant que blocs denses. Le premier bloc dense de chaque bloc-colonne est toujours le bloc diagonal, de dimension carrée. La partie supérieure de la matrice est stockée de la même manière en blocs de lignes ayant exactement les mêmes tailles et les mêmes structures de blocs que leurs homologues verticaux. Une version par blocs de l'algorithme de factorisation symbolique permet alors de construire les blocs-colonnes pour L et U [GN87, CR89]. De cette façon, l'algorithme 1 peut être remplacé par sa version creuse « bloquée » décrite dans l'algorithme 2 et dont les notations sont expliquées sur la figure 1.1 [2]. La division du terme diagonal devient une factorisation LU dense du bloc diagonal ligne 2. Le report des contributions d'un bloc-colonne dans lui-même se transforme

1. Plus rigoureusement, un terme (i, j) est comptabilisé comme non-nul après la factorisation si le a_{ij} initial était non-nul ou si pour au moins une des mises à jour $a_{ij} \leftarrow a_{ij} - a_{ik}a_{kj}$ les deux termes a_{ik} et a_{kj} étaient non-nuls. Ces termes sont comptés "non-nuls" y compris dans le cas (rare) où la somme des contributions s'annule.

2. Plusieurs variantes existent, celle présentée ici est la version *right-looking*. D'autres variantes seront évoquées dans le chapitre suivant.

en une résolution de systèmes triangulaires qui se fait à l'aide de l'opération BLAS TRSM [3] ligne 4 et il en va de même pour les blocs-lignes ligne 5. Les autres contributions, à destinations des blocs-colonnes/lignes à droite du bloc-colonne/ligne k, deviennent un ensemble de produits matriciels calculés avec l'opération BLAS GEMM [4].

Algorithme 2 : `rightLookingLU`

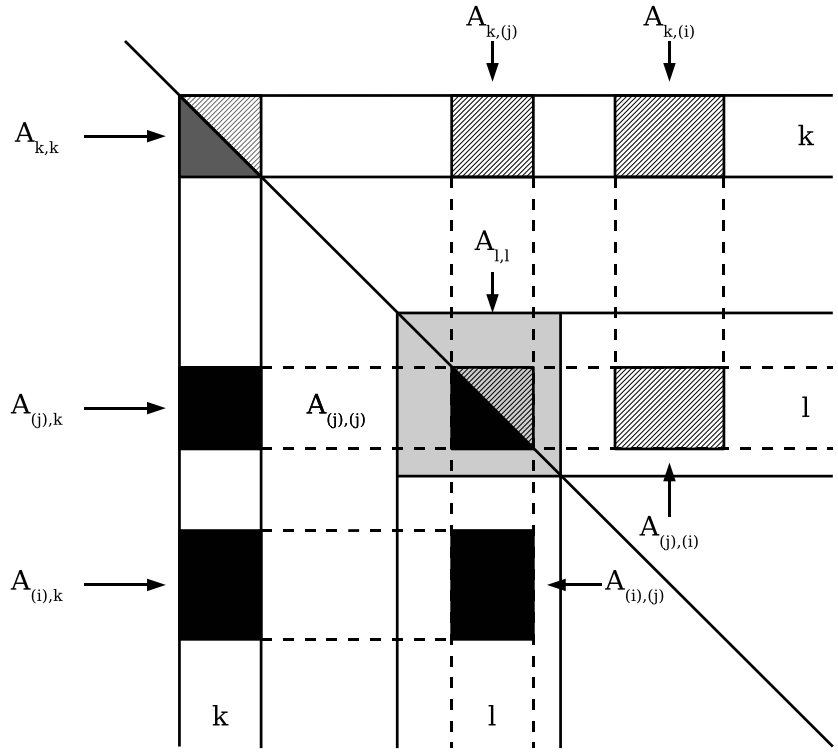
```
   /* Version creuse par blocs denses sur la structure bloc issue de la
      factorisation symbolique par blocs */
   Entrées : Matrice A = (A_ij) ayant N blocs-colonnes ; matrices L et U initialisées avec
             les coefficients non-nuls de A
   Sorties : Matrices L et U telles que LU = A
1  pour k = 1 a N faire
       /* Factorisation du bloc diagonal */
2      Factoriser A_{k,k} en L_{k,k} et U_{k,k};
       /* TRSM sur les blocs extradiagonaux du bloc-colonne et du bloc-ligne
          */
3      pour i ∈ [k] faire
4          L_{(i),k} ← A_{(i),k} U_{k,k}^{-1};
5          U_{k,(i)} ← L_{k,k}^{-1} A_{k,(i)};
       /* Calcul et report des contributions du bloc-colonne k vers les
          blocs-colonnes à sa droite, et du bloc-ligne k vers les blocs-lignes
          en-dessous de lui */
6      pour j ∈ [k] faire
7          pour i ∈ [k] tel que i ≥ j faire
8              L_{(i),(j)} ← L_{(i),(j)} − L_{(i),k} U_{k,(j)};
9              U_{(j),(i)} ← U_{(j),(i)} − L_{(j),k} U_{k,(i)};
```

Un bon algorithme de renumérotation [HK00] doit donc optimiser les trois critères évoqués ci-dessus : le remplissage doit être aussi faible que possible, l'arbre d'élimination large et peu haut et la matrice renumérotée doit être creuse par blocs denses. Le principal algorithme employé est la dissection emboîtée [Geo73, ADD96, PRA00], qui utilise un bipartitionnement récursif. Cet algorithme est notamment implémenté dans les logiciels MeTiS [KK98b] et Scotch [PR96, PR97]. Un exemple de matrice renumérotée par dissection emboîtée, ainsi que son graphe d'adjacence et son arbre d'élimination par bloc-colonne, sont représentés sur la figure 1.2.

Une fois cette phase de renumérotation et de factorisation symbolique par blocs effectuées, la matrice est distribuée sur plusieurs processus avec un ensemble de tâches attribué à chacun d'eux. La matrice est factorisée en parallèle en suivant l'algorithme 2 et en respectant les contraintes de séquentialité fournies par l'arbre d'élimination. Lorsque la factorisation d'un bloc-colonne/ligne provoque des contributions dans un bloc-colonne/ligne possédé par un autre processus, un envoi doit être effectué.

3. Le TRSM est l'opération consistant à résoudre $\alpha AX = B$ ou $\alpha XA = B$, où α est un scalaire, A et B sont des matrices de dimensions compatibles et B est triangulaire supérieure ou inférieure.

4. Le GEMM est l'opération consistant à calculer $\alpha AB + \beta C$, où α et β sont des scalaires et A, B et C des matrices quelconques de dimensions compatibles.



Notations pour les algorithmes de factorisation par blocs :

- Les blocs-colonnes (et les blocs-lignes) sont numérotés de 1 à N.
- Un bloc de la matrice est noté $A_{(i),(j)}$, où i et j sont des couples d'entiers, correspondants respectivement à l'intervalle de lignes et l'intervalle de colonnes sur lesquels s'étend le bloc. Si k est un indice bloc-colonne, on peut noter, par abus, $A_{k,(i)}$, $A_{(i),k}$ ou encore $A_{k,k}$. Les variables entières (qu'on reconnaît comme telles car elles ne sont *pas* entourées de parenthèses) doivent alors être remplacées implicitement par l'intervalle de colonnes du bloc-colonne correspondant.
- Si i et j sont deux couples, le test $i \leq j$ correspond à l'ordre lexicographique ; en pratique, ce test ne sera utilisé qu'avec des intervalles égaux ou disjoints.
- $[k]$ est l'ensemble des intervalles de lignes correspondants aux blocs extra-diagonaux du bloc-colonne k.
- $right(b)$ (où b est un bloc) est le numéro du bloc-colonne dont le bloc diagonal se situe à droite du bloc b. Sur le dessin, $right(A_{(j),k})$ vaut l.
- $< k, l >$ (k et l étant des blocs-colonnes avec $k < l$) est l'intervalle de lignes j tel que $face(A_{(j),k})$ vaut l. Sur le dessin, $< k, l >= j$.
- $Brow(l)$ (où l est un bloc-colonne) est l'ensemble des blocs-colonnes ayant un bloc dans le prolongement gauche de $A_{l,l}$. Sur le dessin, $k \in Brow(l)$.

FIGURE 1.1 – Notations pour les algorithmes de factorisation de matrices creuses par blocs denses.

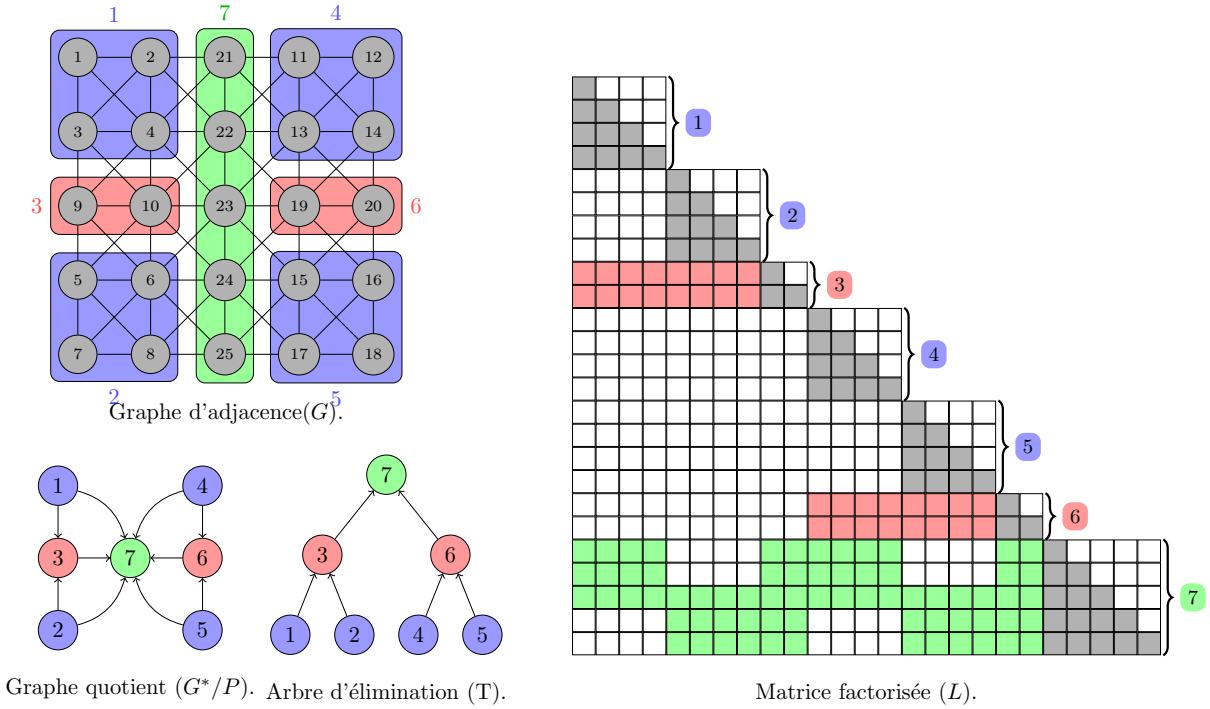


FIGURE 1.2 – Une matrice creuse déjà factorisée comprenant 7 blocs de colonnes (à droite), son graphe d'adjacence G (en haut à gauche), le graphe des dépendances entre les blocs-colonnes (en bas à gauche) et l'arbre d'élimination par bloc-colonne (en bas au milieu).

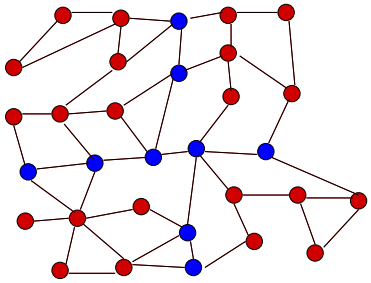
1.2 Méthode de décomposition de domaine avec complément de Schur

La section précédente a décrit rapidement les méthodes directes ; nous allons à présent présenter les méthodes hybrides basées sur une décomposition de domaine, qui s'appuient en partie sur celles-ci et partagent de nombreux points communs. Les grandes étapes de ces méthodes sont les suivantes :

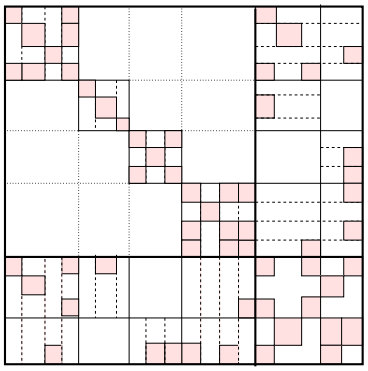
— décomposition de domaine et renumérotation ;
— factorisation de l'intérieur des domaines ;
— calcul d'un préconditionneur ;
— résolution itérative.

La présente section décrit chacun de ces trois points et leur articulation, en se focalisant uniquement sur l'approche appelée « complément de Schur » dans laquelle les domaines se recouvrent seulement sur un minimum de sommets. D'autres méthodes existent, comme les méthodes FETI [FMR94] ou les méthodes de Schwarz [Wid92]. Dans une méthode de Schwarz additive, par exemple, le graphe d'adjacence de la matrice peut être décomposé en domaines se superposant partiellement. Les sous-matrices associées aux différents domaines sont factorisées en parallèle, et les facteurs obtenus sont additionnés sans tenir compte des dépendances entre les domaines. Cette nouvelle matrice sert de préconditionneur pour la résolution itérative.

La phase de calcul du préconditionneur des méthodes hybrides met parfois en jeu une fac-



(a) Un exemple de décomposition en 4 domaines sur un graphe. Les intérieurs sont en rouge et les interfaces en bleu.



(b) Une matrice réordonnée après une décomposition en 4 domaines. Chaque bloc diagonal de la partie B est une matrice creuse par blocs denses (les carrés roses représentent des blocs denses).

FIGURE 1.3 – Décomposition de domaine (visions graphique et matricielle).

torisation approchée d'une sous-matrice. La dernière partie de cette section expliquera les deux principales méthodes utilisées à cette fin.

1.2.1 Décomposition de domaine et renumérotation

Comme il a été expliqué dans l'introduction du chapitre, le principe est de diviser le graphe d'adjacence en domaines, qui pourront par la suite être traités en parallèle. Un partitionneur de graphe (tel que SCOTCH ou METIS) est donc utilisé pour découper le graphe en domaines avec un minimum de recouvrement. Celui-ci s'appuie généralement soit sur un algorithme de bipartitionnement récursif évoqué précédemment, soit sur un algorithme *k-way* [KK98c]. Les inconnues appartenant à deux domaines ou plus sont appelées *interfaces* ; les autres constituent les *intérieurs*. Un exemple d'une telle décomposition est donnée sur la figure 1.3a.

La matrice est ensuite réordonnée de manière à placer d'abord les lignes et les colonnes correspondant aux intérieurs des domaines, regroupées domaine par domaine, puis celles des interfaces. Le système $Ax = b$ ainsi permuté a alors la forme suivante :

$$\begin{pmatrix} B & F \\ E & C \end{pmatrix} \begin{pmatrix} x_B \\ x_C \end{pmatrix} = \begin{pmatrix} b_B \\ b_C \end{pmatrix} \tag{1.1}$$

La matrice B correspond alors aux intérieurs des domaines, la matrice C aux interfaces et les matrices E et F, qui constituent les arêtes les reliant, sont appelées *matrices de couplage*.

Par définition, il n'existe pas d'arête entre deux intérieurs de deux domaines différents. Par conséquent, la matrice B est diagonale par domaine. Si le découpage est effectué en m domaines,

on peut donc écrire :

$$\begin{pmatrix} B_1 & & & & F_1 \\ & B_2 & & & F_2 \\ & & \ddots & & \vdots \\ & & & B_m & F_m \\ E_1 & E_2 & \dots & E_m & C \end{pmatrix} \begin{pmatrix} x_{B_1} \\ x_{B_2} \\ \vdots \\ x_{B_m} \\ x_C \end{pmatrix} = \begin{pmatrix} b_{B_1} \\ b_{B_2} \\ \vdots \\ b_{B_m} \\ b_C \end{pmatrix} \tag{1.2}$$

On peut remarquer que cette renumérotation n'impose qu'une numérotation des inconnues des domaines relativement aux autres domaines ; elle ne donne aucune contrainte pour la numérotation des inconnues au sein même d'un domaine. Ainsi, on peut appliquer de façon naturelle l'algorithme de renumérotation des solveurs directs sur chaque domaine. Chacune des matrices apparaissant dans l'équation ci-dessus est alors elle-même creuse par blocs denses, selon le format décrit dans la section précédente. Un exemple de matrice réordonnée après décomposition de domaine est donné en figure 1.3b.

1.2.2 Factorisation des intérieurs, calcul d'un préconditionneur et résolution itérative

Une fois la renumérotation effectuée, l'idée des méthodes de décomposition de domaine est de résoudre les sous-systèmes associés à chaque matrice B_i ci-dessus de manière indépendante, en utilisant par exemple une méthode directe. Une méthode itérative préconditionnée est ensuite utilisée pour résoudre le système global comprenant l'interface. Pour expliquer ce découpage en étapes, développons la première ligne de l'équation 1.1 :

$$x_B = B^{-1}(b_B - Fx_C) \tag{1.3}$$

La deuxième ligne donne quant à elle $Ex_B + Cx_C = b_C$. En remplaçant par l'expression de x_B et en réorganisant l'équation, on obtient $(C - EB^{-1}F)x_C = b_C - EB^{-1}b_B$. En posant :

$$x'_B = B^{-1}b_B \tag{1.4}$$

et :

$$S = C - EB^{-1}F \tag{1.5}$$

on peut réécrire cette deuxième équation sous la forme :

$$Sx_C = b_C - Ex'_B \tag{1.6}$$

La matrice S est appelée *complément de Schur*.

Dans un premier temps, les factorisations des matrices B_i en $L_{B_i}U_{B_i}$ sont réalisées, ce qui peut être fait indépendamment pour chaque B_i. Puis, les matrices définies par $G = L^{-1}F$ et $W = EU^{-1}$ sont calculées, ce qui permet d'en déduire le complément de Schur $S = C - WG$. Une factorisation approchée de S en $\widetilde{L_S}/\widetilde{U_S}$ est alors effectuée. De manière intéressante, on peut remarquer que ces étapes sont équivalentes à une factorisation complète de la matrice A : celle-ci est débutée (par exemple) de manière exacte avec l'algorithme right-looking présenté dans la section précédente ; elle est stoppée à la dernière colonne correspondant à un intérieur, ce qui provoque le calcul de L_B, U_B, G, W et S ; puis, elle est poursuivie de manière approchée sur le

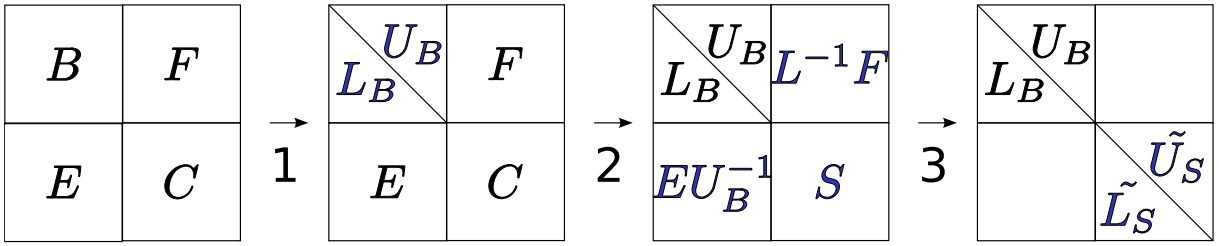


FIGURE 1.4 – Algorithme de factorisation des intérieurs (du début à l'étape 2 incluse) et de construction du préconditionneur (étape 3).

complément de Schur, permettant de calculer les facteurs $\widetilde{L_S}$ et $\widetilde{U_S}$. La première partie (jusqu'au calcul de S) est appelée *factorisation des intérieurs*, et la seconde *calcul du préconditionneur*. Ces deux phases successives sont illustrées sur la figure 1.4.

Les matrices L_B, U_B, G, W, $\widetilde{L_S}$ et $\widetilde{U_S}$ ainsi calculées peuvent ensuite être utilisées en tant que préconditionneur pour pré-calculer une solution approchée du système $Ax = b$ et obtenir une convergence plus rapide de la méthode itérative. Plus précisément, deux principales variantes existent.

- La première consiste à **itérer sur le système complet**. Les équations 1.3, 1.4 et 1.6, peuvent être utilisées pour définir le préconditionneur suivant :

$$x = M^{-1}y \Leftrightarrow \begin{cases} x_C \leftarrow \widetilde{U_S^{-1}}\widetilde{L_S^{-1}}(y_C - EU_B^{-1}L_B^{-1}y_B) \\ x_B \leftarrow U_B^{-1}L_B^{-1}(y_B - Fx_C) \end{cases} \tag{1.7}$$

  Ainsi, lorsque la méthode itérative demande de faire un produit matrice-vecteur avec la matrice du système, le calcul de $x \leftarrow M^{-1}Az$ est effectué en calculant d'abord $y = Az$ puis en utilisant les équations précédentes, mettant en jeu des produits matrice-vecteur et des résolutions triangulaires [5].

  On peut noter que le préconditionneur précédent ne fait pas intervenir les matrices de couplage W et G ; celles-ci sont utiles pour le calcul de S mais peuvent être libérées immédiatement après. Si la mémoire n'est pas un facteur critique, une variante consiste en revanche à garder en mémoire W et à le substituer au EU_B^{-1} de la première équation afin d'économiser un produit matrice-vecteur.

  On peut enfin remarquer que S n'est utilisée qu'afin de calculer la factorisation approchée $\widetilde{L_S}\widetilde{U_S}$, et peut donc être libérée dès que ces matrices ont été calculées. Au total, en plus de la mémoire nécessaire pour le stockage de la matrice A (*i.e* B, E, F et C), ce préconditionneur nécessite de garder en mémoire les matrices L_B, U_B, $\widetilde{L_S}$ et $\widetilde{U_S}$, auxquelles s'ajoute éventuellement W.

- La seconde possibilité pour effectuer la résolution consiste à **itérer** uniquement **sur le système réduit correspondant au complément de Schur**, de taille moindre que le système complet. En effet, si l'on suppose que la matrice B a été factorisée de manière

5. Rappelons en effet que L_B^{-1}, U_B^{-1}, $\widetilde{L_S^{-1}}$ et $\widetilde{U_S^{-1}}$ ne sont jamais explicitement calculées, car ce sont des matrices denses coûteuses en mémoire et en temps de calcul.

exacte, alors on peut libérer les matrices G et W dès que le complément de Schur est calculé puis effectuer les trois étapes suivantes :

1. calculer $x'_B = U_B^{-1} L_B^{-1} b_B$ (équation 1.4) ;
2. résoudre le système réduit $Sx_C = b_S$ de l'équation 1.6 (ou on a posé $b_S = b_C - Ex'_B$) en utilisant le préconditionneur $M^{-1} = \widetilde{U_S^{-1}} \widetilde{L_S^{-1}}$;
3. en déduire x_B avec l'équation 1.3 : $x_B = U_B^{-1} L_B^{-1} (b_B - F x_C)$.

Au total, la mémoire requise pour ce préconditionneur est, en plus de celle de la matrice A, celle des matrices L_B, U_B, $\widetilde{L_S}$, $\widetilde{U_S}$ et S. Contrairement à la première variante, on peut noter que la matrice S est cette fois nécessaire jusqu'à la fin de la résolution. Cependant, si son stockage est trop coûteux, on peut décider de la libérer malgré tout lorsque les facteurs $\widetilde{L_S}$ et $\widetilde{U_S}$ sont calculés ; S est alors remplacée par sa définition lors de l'étape de résolution sur le système réduit (équation 1.5). Chaque produit matrice-vecteur faisant intervenir S est alors remplacé par trois autres auxquels s'ajoutent deux descentes-remontées, mais la mémoire économisée est significative.

Il est montré dans [Saa03] que lorsque B est factorisée exactement et que la même méthode de Krylov est utilisée dans les deux cas, la séquence des solutions successives obtenue est la même que l'on itère sur le système complet ou sur le système réduit. De plus, à précision désirée fixée sur le système global, il est aisé de calculer la précision nécessaire correspondante sur le système réduit. Notons ainsi r le résidu du système global, c'est-à-dire la différence entre le vecteur obtenu et la solution exacte du système, et r_S le résidu sur le système réduit ; si l'on souhaite que r vérifie $\frac{\|r\|}{\|b\|} < tol$ (où tol est choisi), alors il faut et il suffit de satisfaire la contrainte équivalente $\frac{\|r_S\|}{\|b_S\|} < tol_S$ avec un nombre tol_S vérifiant $tol_S < tol \frac{\|b\|}{\|b_S\|}$. Étant donné cette équivalence, et sachant que le système réduit est de taille moindre, il est plus avantageux d'itérer sur celui-ci. Rappelons toutefois que ce résultat n'est vrai que lorsque B est factorisée de manière exacte ; dans le cas où L_B et U_B sont des approximations, il est nécessaire de faire au moins une partie des itérations avec le système complet pour converger vers la solution du système.

1.2.3 Calcul d'une factorisation approchée d'une matrice

Comme nous avons vu à la section précédente, la factorisation des intérieurs est souvent (mais pas toujours) effectuée de manière exacte, et celle du complément de Schur est la plupart du temps approchée. Nous voyons donc dans cette section les deux principales méthodes existantes pour réaliser une factorisation approchée d'une matrice T (qui est généralement S), à moindre coût mémoire et/ou calculatoire. Ces techniques sont appelées *factorisations incomplètes*. La première solution présentée utilise l'idée de supprimer de façon arbitraire certains termes (ou blocs) de remplissage, en se basant sur les chemins d'élimination pour déterminer l'importance des termes. La seconde famille de factorisation utilise quant à elle l'idée de supprimer les coefficients selon leur magnitude.

1.2.3.1 Factorisation incomplète $ILU(k)$

La première solution pour effectuer une factorisation incomplète d'une matrice T est la factorisation $ILU(k)$ [Man80]. Rappelons d'abord le théorème de caractérisation du remplissage : le

remplissage d'un terme (i, j) est conditionné par l'existence d'un chemin de i à j dans le graphe de la matrice ne passant que par des sommets d'indices inférieurs à $min(i, j)$. Un tel chemin est appelé chemin d'élimination. Le principe d'une factorisation $ILU(k)$ consiste à n'autoriser le remplissage que pour les coefficients correspondants à des chemins d'élimination de taille inférieure ou égale à $k + 1$.

Pour ce faire, notons $d_{ij}(p)$ la longueur du plus court chemin entre i et j n'empruntant que des sommets intermédiaires d'indices inférieurs ou égaux à p, $i - 1$ et $j - 1$. Notez qu'un tel chemin :

— soit ne passe que par des sommets inférieurs ou égaux à $\min(p - 1, i - 1, j - 1)$;
— soit passe une unique fois par le sommet p, en empruntant uniquement des sommets inférieurs ou égaux à $\min(p - 1, i - 1, j - 1)$ entre i et p et entre p et j.

Les $d_{ij}(p)$ peuvent ainsi être calculés avec la formule récursive suivante :

$$d_{ij}(p) = \begin{cases} 1 & \text{pour } p = 0 \text{ et } t_{ij} \neq 0 \\ \infty & \text{pour } p = 0 \text{ et } t_{ij} = 0 \\ d_{ij}(p-1) & \text{pour } \min(i,j) \leq p \\ \min(d_{ij}(p-1), d_{ip}(p-1) + d_{pj}(p-1) + 1) & \text{pour } 0 < p < \min(i,j) \end{cases}$$

Le terme (i, j) est alors autorisé par une factorisation incomplète de type $ILU(k)$ si et seulement si $d_{ij}(n) \leq k + 1$.

Dans la formule ci-dessus, on peut remarquer que les dépendances permettent de calculer les matrices $d_{ij}(p)$ par ordre de p croissant. De plus, lors de la construction de $d_{ij}(p)$, seule la sous-matrice carrée en bas à droite correspondant à $\min(i, j) > p$ diffère de la matrice précédente $d_{ij}(p-1)$. Les calculs dans ladite sous-matrice mettent en jeu les coefficients $d_{ip}(p-1)$ et $d_{pj}(p-1)$ qui ne sont justement pas modifiés par l'itération p. Ainsi, l'algorithme peut être implémenté en n'utilisant qu'une seule matrice d_{ij} valant successivement $d_{ij}(0)$, $d_{ij}(1)$, ... $d_{ij}(n)$ au cours des itérations successives. L'algorithme correspondant est donné en Algo. 3.

Algorithme 3 : `calculRemplissageILU(k)`

```
1  /* (i,j) != 0 dans ILU(k) <==> dij <= k+1 à la fin de l'algorithme */;
2  pour i de 1 a n faire
3      pour j de 1 a n faire
4          si t_ij ≠ 0 alors
5              d_ij ← 1;
6          sinon
7              d_ij ← ∞;
8  pour p de 1 a n faire
9      pour i de p+1 a n faire
10         pour j de p+1 a n faire
11             d_ij ← min(d_ij, d_ip + d_pj + 1);
```

Malheureusement, on peut constater que la quantité de calcul que nécessite cet algorithme est du même ordre de grandeur que celle de la factorisation numérique. Toutefois, les matrices sur

lesquelles on opère ont été renumérotées de sorte à avoir une structure creuse par blocs denses, comme dans l'exemple de la figure 1.2. Or, la formule présentée ci-dessus reste valable en utilisant le graphe d'adjacence de T cette fois-ci quotienté par la partition induite par le découpage en blocs. En réécrivant l'algorithme précédent, la boucle en p itérerait alors sur l'ensemble des blocs-colonnes, et i et j sur des blocs. Il faut noter que contrairement à l'algorithme de factorisation LU par bloc (algorithme 2), qui demandait une multiplication de matrices ($BLAS3\ GEMM$) pour chaque couple de blocs d'un bloc-colonne et du bloc-ligne correspondant, l'algorithme de calcul du remplissage ne demande lui qu'une seule opération pour chacun de ces couples. On peut en effet considérer, pour simplifier, qu'une contribution remplit toujours totalement le bloc dans lequel elle a lieu. Ainsi, la matrice d_{ij} peut se contenter de stocker un seul coefficient par bloc, et le coût calculatoire pour la prévision du remplissage est alors significativement moindre que celui de la factorisation.

Le principal atout de la méthode $ILU(k)$ est qu'on peut prévoir la structure des non-zéros de la matrice factorisée, et que celle-ci a une structure par blocs denses ; il est donc possible d'utiliser les opérations BLAS, particulièrement efficaces, lors de la factorisation numérique. Par ailleurs, la méthode peut être implémentée en parallèle [HP99].

1.2.3.2 Factorisation incomplète $ILUT(\tau)$

Le principal défaut des méthodes $ILU(k)$ est qu'elles se basent sur un critère algébrique pour limiter le remplissage. Si, dans le cas des matrices à diagonale dominante notamment, il y a une corrélation entre l'importance d'un terme (i, j) et la petitesse du plus court chemin de remplissage reliant i à j, il existe néanmoins d'autres matrices sur lesquelles un tel critère échoue à produire un préconditionneur amenant une convergence convenable. A l'inverse, sur certaines matrices, la factorisation $ILU(k)$ peut garder de nombreux termes peu importants et ainsi gaspiller des calculs et de la mémoire.

Une alternative aux factorisations $ILU(k)$ est donc de baser le choix des termes à éliminer sur leur valeur. Plus précisément, on choisit un seuil τ, et on garde uniquement les termes dont les valeurs absolues sont supérieures à τ dans la matrice factorisée. Cette technique est appelée $ILUT$ (pour « Incomplete LU Threshold ») [Saa94]. Comme les différentes colonnes de la matrice peuvent détenir des coefficients ayant plusieurs ordres de grandeur de différence, on utilise en pratique un seuillage relatif ; lors du seuillage d'une colonne, on commence par normer celle-ci, puis on supprime les coefficients en-deçà de τ.

A nombre égal de coefficients supprimés, un préconditionneur $ILUT(\tau)$ est généralement plus robuste que son homologue $ILU(k)$. Cependant, l'inconvénient majeur de la méthode $ILUT(\tau)$ est qu'il n'est pas possible de prévoir la structure des non zéros à l'avance. Pour ne pas allouer davantage que nécessaire, il faut donc réaliser la factorisation colonne par colonne (ou bloc de colonnes par bloc de colonnes), en utilisant un espace temporaire pour stocker intégralement la colonne en cours de factorisation, et un autre espace, alloué au fur et à mesure, pour stocker la matrice factorisée définitive comprenant uniquement les valeurs supérieures à τ.

Enfin, un autre désavantage des méthodes $ILUT(\tau)$ est qu'on ne peut pas connaître par avance le nombre de termes supérieurs à τ dans la matrice factorisée ; ainsi, si on se donne comme contrainte de ne pas dépasser une certaine limite mémoire m_{max}, il est nécessaire de faire plusieurs tentatives pour trouver le τ approprié : un τ trop petit risque de demander trop

de mémoire, et à l'inverse un τ grand petit pourrait aboutir à une approximation trop grossière de L_T et U_T ne permettant pas de faire converger le système. Une solution à ce problème est de fixer également une limite p de termes gardés sur une colonne : si plus de p termes sont supérieurs à τ au sein d'une colonne, on ne garde alors que les p plus grands. Cette variante a l'avantage de fournir une borne supérieure de la mémoire utilisée, mais elle donne des résultats variables ; certaines matrices peuvent en effet avoir de nombreux termes importants sur certaines colonnes, et les supprimer peut fortement compromettre la convergence du système.

L'algorithme 4 décrit cette dernière variante. Notez que les boucles sont légèrement modifiées par rapport à l'algorithme 1, puisqu'ici la matrice doit être calculée en scalaire, ligne par ligne (alors qu'auparavant, l'étape k faisait des modifications sur l'ensemble de la sous-matrice $(n-k-1)\times(n-k-1)$ du coin inférieur droit).

Algorithme 4 : `calculRemplissageILUT(`τ`)`

```
1  pour i = 1 a n faire
2      w ← t_{i*};
3      pour k = 1 a i − 1 faire
4          si t_{ik} ≠ 0 alors
5              w_k ← t_{ik}/t_{kk};
6              pour j = k + 1 a n faire
7                  w_j ← t_{ij} − t_{ik} t_{kj};
8      λ ← ‖w‖_2;
9      nnz ← 0;
10     pour k = 1 a n faire
11         si w_k < τλ alors
12             w ← 0;
13         sinon
14             nnz + +;
15     Allouer la ligne i de la matrice factorisée finale avec nnz éléments;
16     Recopier les termes non-nuls de w dans cette ligne;
```

1.3 Principaux solveurs de type complément de Schur

La section précédente a présenté, de façon générale, les grandes étapes d'une méthode hybride par décomposition de domaine et complément de Schur. Nous étudions à présent plus précisément les détails de mise en œuvre et la parallélisation de cette méthode dans le cas des différents solveurs existants. Avant cela, nous commençons par décrire le tronc commun des différentes approches.

Comme il a été expliqué dans la section 1.2.2, une fois la décomposition de domaine et la renumérotation réalisées, la méthode hybride demande d'effectuer une factorisation right-looking des premières colonnes de la matrice afin de calculer L_B, U_B, G, W et S. Or, la matrice B étant diagonale par blocs, on peut remarquer que les contributions de la factorisation des différents domaines ne se recouvrent que sur l'interface. Il est donc possible d'effectuer les factorisations

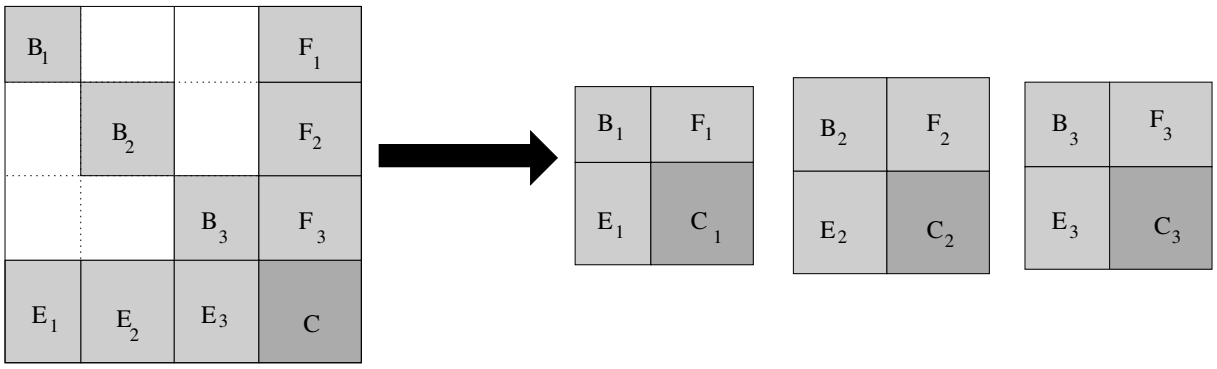


FIGURE 1.5 – Extraction des matrices de domaine. La factorisation de chaque matrice peut être réalisée en parallèle.

de façon indépendante pour les différents domaines. Plus précisément, pour un domaine i, on appelle E_i la portion de la matrice E en-dessous de B_i, F_i la portion de matrice F à droite de B_i, et C_i la portion de matrice C dans laquelle le domaine i apporte des contributions ; les C_i des différents domaines peuvent ne pas être disjoints. Enfin, on note A_i la sous-matrice du domaine i, constituée de B_i, E_i, F_i et C_i. Alors, la phase de factorisation right-looking des colonnes de B est équivalente à la factorisation des colonnes B_i de chaque matrice A_i, suivi d'une phase de reconstitution de la matrice S en additionnant les portions des C_i se recouvrant. Il en résulte que les différentes matrices A_i peuvent être extraites et les factorisations correspondantes effectuées en parallèle ; cette approche est illustrée sur la figure 1.5.

Une fois les intérieurs des différents domaines factorisés et les compléments de Schur locaux calculés, il reste à factoriser le complément de Schur global. Du fait de sa taille, celui-ci n'est jamais rassemblé sur un unique processus, mais reste à tout moment distribué. Comme les S_i locaux se recouvrent partiellement, il faut néanmoins effectuer une phase d'assemblage bloc par bloc avant de commencer la factorisation du complément de Schur global.

Nous exposons à présent les multiples stratégies mises en œuvre dans les différents solveurs hybrides.

1.3.1 Le solveur HIPS

Le solveur HIPS [GH08, Gai09], développé par Pascal Hénon et Jérémie Gaidamour, est conçu pour utiliser plusieurs domaines par processus MPI. Les domaines crées lors de la décomposition étant souvent de tailles inégales, cette stratégie a l'avantage de permettre un bon équilibrage mémoire en faisant une répartition judicieuse des domaines sur les processus. Par ailleurs, les résultats montrent que le préconditionneur construit est suffisamment robuste pour obtenir une bonne convergence même avec un grand nombre de domaines. L'augmentation du nombre de domaines ne pose donc pas de problème et le solveur a une bonne scalabilité lorsque le nombre de processus augmente.

Afin de limiter la mémoire nécessaire au stockage du préconditionneur, le solveur HIPS offre la possibilité de restreindre le remplissage en utilisant un seuillage numérique $ILUT(\tau)$ sur les matrices W, G, S et $\widetilde{L_S}$ / $\widetilde{U_S}$, comme il a été vu dans la section 1.2.3.2.

Dans la même optique, et également dans le but d'avoir un plus fort potentiel de parallélisation lors de la factorisation du complément de Schur, le solveur HIPS introduit deux variantes de restriction arbitraire du remplissage du complément de Schur. On peut observer que dans une factorisation directe sur place sans restriction, la matrice C passe par trois stades successifs : elle est d'abord C, puis devient le complément de Schur S après la factorisation des colonnes de B, et est enfin remplacée par la factorisation $\widetilde{L_S}\widetilde{U_S}$. Chaque étape ajoute de nouveaux blocs de contributions. HIPS décide de restreindre les blocs de remplissage de cette sous-matrice soit au premier stade dans la variante dite « R_C » (blocs de remplissage de C uniquement), soit au second stade dans la variante dite « R_S » (blocs de remplissage de S uniquement). La factorisation (exacte puis approchée) est toujours menée jusqu'à son terme, mais les contributions en-dehors du schéma de remplissage choisi ne sont pas prises en compte. La variante R_C correspond à une factorisation $ILU(k=0)$ de la matrice A sur les colonnes de C [6]. La variante R_S, plus robuste et plus coûteuse en mémoire, revient quant à elle à une factorisation $ILU(0)$ de la matrice S. A l'échelle de la matrice A en entier, on peut montrer que R_S préserve au moins tous les chemins d'élimination de longueur 4, et inclut donc au minimum tout le remplissage d'une factorisation $ILU(k=3)$ [Gai09].

Il résulte des règles de remplissage précédentes que deux compléments de Schur locaux ne peuvent se recouvrir que s'ils correspondent à des domaines adjacents. Le solveur HIPS prend en compte cette propriété en assignant préférentiellement un ensemble de domaines adjacents les uns aux autres à chaque processus, formant des super-domaines. En procédant ainsi, les seules communications nécessaires pour la factorisation parallèle du complément de Schur se produisent entre des processus responsables de super-domaines adjacents, et le volume des communications correspond à la taille de la super-interface correspondante.

Pour finir, une particularité du solveur HIPS est qu'il utilise une décomposition de domaine dite hiérarchique. L'interface des domaines est ainsi redécoupée en blocs appelés connecteurs, et chaque connecteur se voit assigner un entier positif tel qu'un connecteur de niveau i ne peut être adjacent qu'à des connecteurs de niveau $j < i$. Les connecteurs de niveau i jouent donc le rôle de séparateurs pour les niveaux inférieurs. De plus, la matrice C est numérotée de manière à placer les connecteurs par ordre croissant de niveau. Dans le cas du motif R_C, par exemple, l'avantage de cette méthode est que tous les connecteurs d'un même niveau peuvent être éliminés indépendamment les uns des autres. La factorisation de S présente donc un fort degré de parallélisme.

1.3.2 Le solveur PDSLIN

Nous présentons à présent les principales spécificités du solveur PDSLIN [YL11, YLE11]. Contrairement au solveur HIPS, ce solveur utilise plusieurs processus par domaine. Les factorisations des matrices locales A_i sont réalisées en mémoire distribuée par le solveur direct SuperLU_DIST, qui utilise une distribution 2D cyclique pour répartir le travail sur les processus.

Une fois les domaines factorisés, PDSLIN utilise un sous-ensemble des processus pour factoriser le complément de Schur. Le solveur fait en sorte que les phases de factorisation des domaines et de reconstitution du Schur global puissent se superposer partiellement. A cette fin,

6. La factorisation $ILU(k)$ a été décrite en section 1.2.3.1.

il assigne la factorisation du complément de Schur aux processus travaillant sur les domaines les plus petits ; comme ceux-ci terminent leur factorisation plus tôt que les autres, ils deviennent libres pour commencer à recevoir les blocs du complément de Schur, tandis que les processus à qui ont été assignés les domaines les plus gros terminent leur factorisation.

Tout comme HIPS, le solveur PDSLIN utilise un seuillage numérique pour limiter le nombre de termes non-nuls dans W, G puis S. De plus, le solveur évite d'allouer une trop grande quantité de mémoire sur les processus chargés de factoriser le complément de Schur en n'allouant à la fois que deux buffers de réception pour les Schur locaux.

Afin d'améliorer la stabilité numérique de la résolution du complément de Schur, PDSLIN applique une permutation à la matrice afin de maximiser les valeurs absolues des termes diagonaux, puis normalise chaque ligne. L'algorithme utilisé pour trouver une bonne permutation s'appelle MC64 [DK99]. Cependant, comme il s'agit d'un code séquentiel et que le complément de Schur reste distribué tout au long de la factorisation, le solveur se contente d'appliquer l'algorithme sur les blocs des compléments de Schur locaux, ce qui permet d'avoir une stabilité suffisante en pratique.

1.3.3 Le solveur ShyLU

Le solveur ShyLU [RBH12] se base sur l'observation que le nombre de cœurs de calcul par nœud augmente rapidement, et qu'une approche de type « un domaine par cœur » n'est pas viable à terme : en effet, une augmentation trop importante du nombre de domaines se traduit par un plus grand nombre d'itérations lors de la résolution. Le solveur propose donc à la place une approche de type « un domaine par nœud ». De plus, le bon compromis de l'utilisation de MPI se situe entre l'extrême « 1 processus MPI par nœud » et « 1 processus MPI par cœur », l'idéal se situant autour de « 1 processus MPI par région à accès mémoire uniforme ». Le solveur part de ce principe et envisage une utilisation de type « plusieurs processus MPI par domaine » (et donc par nœud). En outre, ShyLU est multithreadé, ce qui lui donne un second niveau de parallélisme permettant d'exploiter l'augmentation du nombre de cœurs de calcul.

Une autre particularité du solveur ShyLU est qu'il utilise des séparateurs dits « épais ». Plus précisément, un chemin dont les extrémités se situent dans deux domaines différents traverse au moins deux sommets de l'interface (ce qui revient également à dire que le solveur utilise des « séparateurs-arête » au lieu de « séparateurs-sommet »). Cette propriété a pour conséquence que chaque sommet de l'interface est relié à au plus un sommet de l'intérieur d'un domaine. Il en résulte que les matrices E et F, tout comme B, sont diagonales par bloc. La matrice A a donc la forme suivante :

$$\begin{pmatrix} B_1 & & & & F_1 & & & \\ & B_2 & & & & F_2 & & \\ & & \ddots & & & & \ddots & \\ & & & B_m & & & & F_m \\ E_1 & & & & C_{11} & C_{12} & \dots & C_{1m} \\ & E_2 & & & C_{21} & C_{22} & \dots & C_{2m} \\ & & \ddots & & \vdots & \vdots & \ddots & \vdots \\ & & & E_m & C_{m1} & C_{m2} & \dots & C_{mm} \end{pmatrix}$$

Cette manière de procéder a l'inconvénient qu'elle peut augmenter sensiblement la taille du complément de Schur, puisqu'un séparateur-arête peut posséder jusqu'à deux fois plus de sommets qu'un séparateur-sommet. En revanche, elle présente également un avantage intéressant : en faisant ainsi, un domaine A_i n'a de contribution que dans le bloc diagonal C_{ii}, et les compléments de Schur locaux sont donc disjoints. Contrairement aux autres solveurs, ShyLU ne nécessite donc pas de phase d'assemblage du complément de Schur global.

Tout comme HIPS et PDSLIN, le solveur ShyLU utilise diverses stratégies afin de limiter le remplissage du complément de Schur. Il offre tout d'abord la possibilité d'effectuer un seuillage numérique. Cependant, afin d'augmenter la robustesse du préconditionneur construit, ce seuillage n'a lieu que sur le complément de Schur S, et non sur W et G comme dans les précédents solveurs. Deuxièmement, ShyLU a également une variante lui permettant de limiter le remplissage du Schur à un pochoir prédéfini. Cette stratégie sera décrite plus précisément dans le chapitre 2 (section 2.1.2) qui présente les principales stratégies de réduction de la consommation mémoire.

1.3.4 Le solveur MaPHyS

Le solveur MaPHyS [Hai08, GH09, GHS10, AGGR11, AGG+13] est un autre solveur basé sur l'approche de décomposition de domaine et de complément de Schur, qui associe un ou plusieurs processus MPI par domaine. Le complément de Schur est assemblé d'une manière particulière en se basant sur une méthode de Schwarz. Plus précisément, notons $\overline{S_i}$ la restriction du complément de Schur S au domaine i et R_i l'opérateur de restriction associé. Les $\overline{S_i}$ représentent ainsi les compléments de Schur locaux déjà assemblés, alors que les S_i sont ceux avant assemblage. On a donc :

$$S = \sum_{i=0}^{m-1} {}^tR_i S_i R_i \tag{1.8}$$

et :

$$S = \max_{i=0}^{m-1} {}^tR_i \overline{S_i} R_i \tag{1.9}$$

où pour chaque coefficient, tous les opérandes non-nuls du max sont égaux [7]. Le solveur MaPHyS construit le préconditionneur suivant :

$$M = \sum_{i=0}^{m} {}^tR_i \overline{S_i}^{-1} R_i \tag{1.10}$$

7. *i.e.* $\forall i,j \in [\![0, m-1]\!], \forall k,l \in [\![0, n-1]\!], (R_i\overline{S_i}R_i)_{kl} = (R_j\overline{S_j}R_j)_{kl}$ ou $(R_i\overline{S_i}R_i)_{kl} = 0$ ou $(R_j\overline{S_j}R_j)_{kl} = 0$.

Ainsi, il inverse chaque Schur local assemblé, en parallèle, puis en fait la somme [8]. L'intérêt d'inverser les $\overline{S_i}$ plutôt que les S_i réside dans le fait que les premiers sont garantis d'être inversibles dans le cas d'une matrice de départ symétrique définie positive.

On peut remarquer que la complexité de la factorisation des compléments de Schur dépend de la taille de ceux-ci, qui elle-même correspond à la taille des interfaces de la décomposition de domaine. Pour avoir un bon équilibrage de la charge lors de leur factorisation parallèle, il faut donc que les interfaces locales des domaines soient équilibrés. Le chapitre 3 de cette thèse abordera ce problème.

Comme les autres solveurs, le solveur MaPHyS intègre en plus une stratégie de seuillage numérique, facultative, permettant de réduire la mémoire nécessaire au stockage du préconditionneur. Ce seuillage est effectué sur les matrices $\overline{S_i}$.

Enfin, MaPHyS offre également la possibilité de factoriser les intérieurs de manière approchée en utilisant une méthode $ILUT(\tau)$ (voir section 1.2.3.2). Le préconditionneur est alors calculé de la même manière, mais en remplaçant les $\overline{S_i}$ par leurs approximations. Cette méthode permet de calculer un préconditionneur à moindre coût.

1.4 Positionnement de la thèse

Dans ce chapitre, nous avons présenté la méthode de résolution hybride basée sur une décomposition de domaine et un complément de Schur, puis passé en revue les différents solveurs existants de ce type. De façon générale, les méthodes hybrides sont moins coûteuses en temps et en consommation mémoire que les méthodes directes. Cependant, lors de la résolution de systèmes linéaires très grands, les méthodes hybrides font elles aussi face à plusieurs limites.

Cela est vrai, notamment, en terme de consommation mémoire. Celle-ci peut facilement être réduite *via* une augmentation du nombre de domaines ce qui, de plus, accroît le degré de parallélisme de la méthode. Néanmoins, augmenter ce paramètre a également pour effet de faire croître la taille totale de l'interface, et donc du complément de Schur. Par conséquent, au-delà d'un certain seuil, effectuer un découpage en davantage de domaines est contre-productif et ne permet plus de diminuer la mémoire. En outre, un nombre de domaines accru s'accompagne généralement d'une détérioration de la convergence. Un compromis est donc à trouver et il est nécessaire d'utiliser des techniques complémentaires afin de faire baisser la quantité de mémoire utilisée.

Le chapitre 2 de cette thèse se concentre donc sur les aspects « mémoire » des solveurs hybrides. Nous y exposerons les différentes techniques déjà employées par les solveurs existants afin d'économiser celle-ci. Nous présenterons ensuite une première contribution de notre travail, pouvant être mise en œuvre dans le cadre d'une approche où plusieurs processus sont utilisés pour factoriser chaque domaine. Nos idées nouvelles se basent sur un ordonnancement spécifique des tâches, des allocations et des désallocations lors de la factorisation afin de masquer les surcoûts mémoire apparaissant lors de la factorisation des intérieurs et du calcul du préconditionneur.

8. En réalité, les inverses ne sont pas formés ; lorsque M est utilisé, des descentes-remontées sur les factorisations LU des $\overline{S_i}$ sont effectuées.

Elles seront validées dans la dernière partie du chapitre.

Une autre problématique importante liée aux solveurs hybrides est l'équilibrage des domaines dans le but de réaliser une bonne répartition de la charge de travail. Celui-ci est réalisé par un partitionneur de graphe tel que Scotch. Actuellement, ces logiciels permettent uniquement d'équilibrer le nombre de sommets dans chaque domaine. Cependant, la complexité de la factorisation d'un domaine dépend également de la taille de son interface locale. Il est donc souhaitable d'équilibrer aussi ces interfaces locales, ce qui n'était pas pris en compte jusque-là dans les partitionneurs. Nous proposerons donc dans le chapitre 3 plusieurs variantes des algorithmes existants afin de considérer ce critère supplémentaire, puis validerons ces nouvelles approches.

Chapitre 2

Réduction de la mémoire dans les méthodes hybrides

Sommaire

2.1 Méthodes de réduction mémoire préexistantes 25
2.1.1 Seuillage numérique des matrices de couplage (HIPS et PDSLIN) . . . 26
2.1.2 Restriction du remplissage du complément de Schur à un « pochoir » (ShyLU) 28
2.1.3 Allocation et libération des matrices de couplage par domaine (HIPS) . 29
2.2 Optimisation de la gestion des matrices de couplage 30
2.2.1 Variantes fan-in, fan-out, right-looking et left-looking 30
2.2.2 Gestion optimisée de la mémoire 32
2.2.2.1 Version right-looking pure avec allocation dynamique des blocs 34
2.2.2.2 Version mixte left-right-looking pour le traitement du couplage 34
2.2.3 Expériences numériques 36

Le précédent chapitre a détaillé le fonctionnement des méthodes hybrides basées sur une approche de décomposition de domaine avec complément de Schur [9]. Ces méthodes ont l'avantage de fournir un bon compromis entre les méthodes directes, robustes mais coûteuses en temps et en espace, et les méthodes itératives, qui parfois convergent difficilement. Cependant, les méthodes hybrides ne sont pas dépourvues de surcoûts mémoire importants, bien que dans une moindre mesure par rapport aux méthodes directes. La première section du présent chapitre passe en revue les multiples approches déjà mises en œuvre dans les différents solveurs pour traiter ce problème, et la seconde expose les idées nouvelles apportées dans cette thèse.

2.1 Méthodes de réduction mémoire préexistantes

Nous faisons dans cette section un tour d'horizon des différentes pistes pouvant être exploitées afin de réduire la mémoire utilisée par un solveur hybride. Nous supposerons que le nombre de domaines de la décomposition est fixé car, même s'il a un impact sur la quantité de mémoire

9. Dans la suite nous parlerons simplement de « méthodes hybrides » pour désigner ces méthodes-là en particulier.

utilisée, il est en pratique choisi en fonction de la robustesse du préconditionneur et du degré de parallélisme voulu.

Comme nous l'avons vu dans le chapitre précédent, la mémoire d'un solveur hybride est essentiellement constituée par la mémoire du préconditionneur utilisé, qui demande de stocker les facteurs L_B, U_B, $\widetilde{L_S}$ et $\widetilde{U_S}$, auquel s'ajoute S dans le cas où les itérations n'ont lieu que sur l'interface [10]. La construction de ces matrices demande elle-même une certaine quantité de mémoire temporaire, due d'une part à la mémoire utilisée par les solveurs directs, et d'autre part au stockage provisoire des matrices W et G, qui ne sont pas gardées pour l'étape de résolution. La mémoire nécessitée par L_B et U_B dépend de l'algorithme de renumérotation utilisé dans les domaines ; en pratique, l'algorithme principalement employé, la dissection emboîtée, permet d'obtenir un remplissage raisonnable tout en assurant que la factorisation présente un bon degré de parallélisme. Pour ce qui est de la matrice S du complément de Schur, nous avons vu au chapitre précédent qu'il pouvait éventuellement être libéré au fur et à mesure du calcul de $\widetilde{L_S}$ et $\widetilde{U_S}$, économisant ainsi une grande quantité de mémoire au prix d'un surcoût en temps acceptable lors de la résolution. Enfin, les matrices $\widetilde{L_S}$ et $\widetilde{U_S}$ peuvent être creusées plus ou moins en jouant sur le paramètre τ de la factorisation $ILUT(\tau)$ (section 1.2.3.2 page 17), k de la factorisation $ILU(k)$ (section 1.2.3.1 page 15), ou en utilisant les schémas de remplissage R_C et R_S de HIPS (section 1.3.1 page 19) [11]. Cependant, la diminution du nombre de termes dans $\widetilde{L_S}$ et $\widetilde{U_S}$ détériore la robustesse du préconditionneur, et ce biais ne peut donc être utilisé que dans une certaine mesure.

La présente section expose trois méthodes déjà existantes pour diminuer la consommation mémoire dans un solveur hybride. Les deux premières se basent sur le calcul approché de S, soit par creusement numérique, à la manière d'une méthode $ILUT(\tau)$, soit de manière algébrique, par une restriction du motif des non-zéros de S à un « pochoir » préétabli. La dernière section présente une possibilité spécifique au solveur HIPS qui peut affecter plusieurs domaines à chaque processus.

2.1.1 Seuillage numérique des matrices de couplage (HIPS et PDSLIN)

Comme nous l'avons vu au chapitre 1, de nombreux solveurs offrent la possibilité d'utiliser une factorisation $ILUT(\tau)$ pour éliminer des termes de remplissage peu importants numériquement dans L_S et U_S. Les solveurs HIPS [GH08, Gai09] et PDSLIN [YL11] explorent davantage cette idée de seuillage numérique en l'étendant aux matrices temporaires de couplage, désormais notées $\widetilde{W}$ et $\widetilde{G}$; contrairement à auparavant, la matrice S est alors calculée de manière approchée et notée $\widetilde{S}$, puis une factorisation $ILUT(\tau)$ de cette approximation est calculée pour obtenir $\widetilde{L_S}$ et $\widetilde{U_S}$. L'économie mémoire est alors plus forte, tout en gardant une qualité de préconditionneur acceptable.

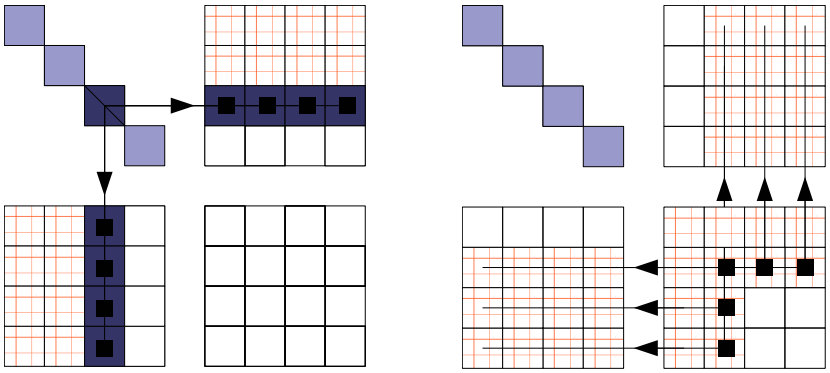
De plus, afin de minimiser le pic mémoire dû aux matrices provisoires W et G, l'ordonnancement des tâches de la factorisation des intérieurs est modifié. Au départ, la mémoire pour W, G et S n'est pas préallouée. Les calculs se déroulent ensuite en deux étapes, illustrées par les figures 2.1a et 2.1b respectivement.

10. Les notations L_B, U_B, $\widetilde{L_S}$, $\widetilde{U_S}$, W et G ont été introduites dans le chapitre 1.

11. Rappelons néanmoins que l'intérêt premier de cette dernière technique est la réduction des communications nécessaires lors de la factorisation parallèle du complément de Schur.

Dans un premier temps, la matrice B est factorisée et les matrices W et G sont formées. Ces dernières sont calculées bloc de colonnes par bloc de colonnes pour W, ou blocs de lignes par bloc de lignes pour G ; chaque nouveau bloc de colonnes (resp. lignes) calculé est placé dans un bloc temporaire, sur lequel le seuillage numérique est appliqué immédiatement. Le nombre de non-zéros du bloc étant alors connu, la structure scalaire définitive pour stocker ce morceau est allouée et les coefficients sont recopiés à l'intérieur. De cette façon, W et G sont seuillées au fur et à mesure de leur construction, et à aucun moment l'intégralité des versions non-seuillées de ces matrices n'est stockée en mémoire.

Dans un second temps, l'allocation, le calcul et la factorisation du complément de Schur sont effectuées. Le calcul de S implique de reporter les contributions apportées par W et G (opération BLAS GEMM des lignes 8-9 de l'algorithme 2 page 9). Dans une optique de limitation du pic mémoire, l'ensemble de ces opérations est entremêlée avec la factorisation incomplète de S, de façon à libérer les premiers blocs de W et G au fur et à mesure que les blocs de S sont alloués. Plus précisément, à chaque étape, les contributions apportées par un ensemble de lignes de G et un ensemble de colonnes symétriques de W sont reportées, provoquant l'allocation des blocs associés de S, puis les parties correspondantes de W et G sont libérées. Les blocs nouvellement créés de S sont alors factorisés et seuillés. Afin d'éviter que cette factorisation ne provoque l'allocation immédiate de davantage de blocs de S (à cause des contributions de S dans lui-même reportées à droite), un ordonnancement particulier appelé *left-looking* est employé (voir section 2.2.1). De manière intéressante, cet entrelacement des opérations (report des contributions de W et G + factorisation left-looking de S) est équivalent à une factorisation left-looking de l'ensemble de la matrice A qui serait débutée à partir des colonnes de l'interface.



(a) Factorisation de B, TRSM et seuillage de W et G.

(b) Factorisation left-looking d'un bloc-colonne de S.

FIGURE 2.1 – Illustration de la technique de seuillage numérique de W et G suivi d'une factorisation $ILUT(\tau)$ de S. Les carrés blancs représentent des blocs non-alloués ; le quadrillage rouge symbolise des parties creuses qui ne sont pas groupées au sein de blocs denses.

Comme expliqué dans la section 1.2.3.2, l'inconvénient du seuillage numérique est l'impossibilité d'utiliser les opérations BLAS, du fait que la structure finale de la matrice n'est pas connue à l'avance. Par conséquent, la section suivante étudie une autre méthode permettant de garder des blocs denses.

2.1.2 Restriction du remplissage du complément de Schur à un « pochoir » (ShyLU)

Nous étudions à présent une optimisation mémoire permettant de réduire le nombre de termes de S. Elle est utilisée dans le solveur ShyLU [RBH12], et consiste à choisir à l'avance le remplissage de S selon un « pochoir » prédéfini. ShyLU utilise comme pochoir une matrice bande, ce qui convient généralement bien dans le cas de problèmes issus d'équations aux dérivées partielles. Le remplissage n'est alors autorisé qu'au sein du motif de la matrice C et du pochoir utilisé. Dans la suite, nous noterons $\overline{S}$ la matrice S restreinte à ce remplissage. L'avantage de cette méthode est qu'elle permet de calculer $\overline{S}$ sous une forme compacte, en faisant en sorte qu'aucun terme en dehors du motif des non-zéros de $\overline{S}$ ne soit calculé.

Nous allons illustrer la technique dans le cas symétrique sur un exemple. Supposons que C ait le motif de non-zéros constitué d'une diagonale simple plus deux termes dans la seconde colonne et la seconde ligne. Si nous choisissons comme pochoir une matrice tridiagonale, la matrice $\overline{S}$ aura alors la forme présentée sur la figure 2.2, correspondant au motif de C auquel on a ajouté une matrice tridiagonale.

$$\overline{S} = \begin{pmatrix} a_1 & a_8 & & & & & \\ a_8 & a_2 & a_9 & a_{14} & & a_{15} & \\ & a_9 & a_3 & a_{10} & & & \\ & a_{14} & a_{10} & a_4 & a_{11} & & \\ & & & a_{11} & a_5 & a_{12} & \\ & a_{15} & & & a_{12} & a_6 & a_{13} \\ & & & & & a_{13} & a_7 \end{pmatrix} \quad V = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \quad \overline{S}V = \begin{pmatrix} a_1 & a_8 & 0 \\ a_8 + a_9 & a_2 & a_{14} + a_{15} \\ a_3 & a_9 & a_{10} \\ a_{10} + a_{11} & a_{14} & a_4 \\ a_5 & 0 & a_{11} + a_{12} \\ a_{12} + a_{13} & a_{15} & a_6 \\ a_7 & 0 & a_{13} \end{pmatrix}$$

FIGURE 2.2 – La matrice $\overline{S}$ correspondant à S avec un remplissage tronqué, la matrice V du coloriage et le produit $\overline{S}V$.

Nous allons à présent calculer une matrice V telle que le produit $\overline{S}V$ soit une représentation compacte de $\overline{S}$. Pour cela, on calcule un coloriage du graphe associé à $\overline{S}$, contenant aussi peu de couleurs que possible. Le problème étant NP-complet [12], on utilise pour cela une heuristique. Dans notre exemple, le graphe et un 3-coloriage de celui-ci sont représentés sur la figure 2.3. La matrice V est alors une matrice $n \times p$, où n est la taille de C et p le nombre de couleurs, dans laquelle chaque ligne i comprend un 1 dans la colonne correspondant à la couleur j du sommet i, et des zéros partout ailleurs. Dans notre cas, en prenant par exemple « rouge = 1 », « bleu = 2 » et « vert = 3 », on obtient la matrice V de la figure 2.2.

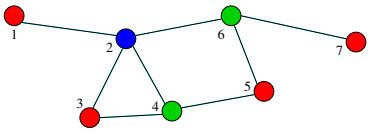


FIGURE 2.3 – Le graphe correspondant à $\overline{S}$ et un 3-coloriage de ses sommets (ce qui est ici le minimum).

On obtient alors le produit $\overline{S}V$ présenté dans la dernière matrice de la figure. On peut consta-

12. Voir par exemple `https://fr.wikipedia.org/wiki/Coloration_de_graphe`.

ter ici que tous les coefficients de $\overline{S}$ peuvent être retrouvés à partir de $\overline{S}V$: tous les coefficients sauf a_{11} et a_{12} apparaissent dans la matrice, et on peut déduire a_{11} de la somme $a_{10}+a_{11}$ et a_{12} de la somme $a_{12}+a_{13}$. Il est montré dans [CM92] que ce résultat est général et qu'il existe un algorithme permettant de reconstruire $\overline{S}$ à partir de $\overline{S}V$. A l'inverse, on constate que tous les termes de $\overline{S}V$ ne sont pas nécessaires pour retrouver $\overline{S}$: les trois zéros sont inutiles et les trois sommes non citées ci-dessus sont redondantes. Plus le nombre de couleurs utilisées sera réduit, plus l'économie en terme de mémoire sera grande. Ce nombre dépend à la fois de la structure du graphe, du nombre de non zéros de $\overline{S}$ et de la qualité de l'heuristique de coloriage (le problème étant NP-complet).

L'idée utilisée dans ShyLU est alors de procéder comme suit :

1. on choisit un pochoir (généralement, une matrice bande de largeur 5% de la dimension de la matrice), ce qui donne la structure des non zéros de $\overline{S}$;
2. on calcule un coloriage du graphe associé, ce qui donne la matrice V ;
3. on calcule $CV - W(GV)$, ce qui donne la matrice $\overline{S}V$;
4. on en déduit la matrice $\overline{S}$.

Chacune des étapes 2, 3 et 4 peuvent être effectuées en parallèle. L'inconvénient majeur de cette méthode est qu'il n'est pas possible *a priori* de « deviner » quel pochoir fonctionnera correctement pour une matrice donnée, en dehors du cas particulier où la matrice est issue d'un problème aux dérivées partielles. Cependant, dans le cas où un pochoir bande suffit pour faire converger le système, cette méthode permet d'économiser davantage de mémoire que les autres méthodes présentées dans cette section.

2.1.3 Allocation et libération des matrices de couplage par domaine (HIPS)

Il existe une troisième possibilité pour réduire la mémoire dans le cas où plusieurs domaines par processus sont utilisés. Elle est implémentée dans le solveur HIPS et constitue une alternative à la solution présentée en sous-section 2.1.1 (les deux optimisations ne pouvant être mises en œuvres simultanément). Cette variante consiste à remarquer que les matrices W_i et G_i des différents domaines sont indépendantes les unes des autres, car par construction de la décomposition, il n'existe aucune contribution d'une de ces matrices dans une autre. Comme ces matrices ne sont utiles que jusqu'au calcul du complément de Schur, on peut donc procéder de la manière suivante, pour chaque domaine i assigné à un même processus (en supposant que la mémoire pour S est préallouée) :

1. factoriser la matrice B_i en L_{B_i}/U_{B_i} ;
2. allouer la mémoire pour W_i et G_i et les calculer (opération BLAS TRSM) ;
3. reporter les contributions de W_i et G_i dans le complément de Schur (opération BLAS GEMM) ;
4. désallouer W_i et G_i avant de passer au domaine suivant.

En procédant de cette manière, le pic mémoire sur un processus p survient lorsque le domaine $imax_p$ ayant les plus grosses matrices G_{imax_p} et W_{imax_p} est factorisé. On a alors S, L_B , U_B , G_{imax_p} et W_{imax_p} allouées en même temps.

2.2 Optimisation de la gestion des matrices de couplage

La section précédente a présenté plusieurs méthodes déjà existantes afin de réduire la consommation mémoire dans les solveurs hybrides. Nous explicitons à présent de nouvelles idées introduites dans cette thèse. Avant de les exposer, nous avons besoin de préciser les différents ordonnancements des tâches et des communications envisageables pour la mise en œuvre de l'algorithme 2 page 9 du chapitre 1. Dans une première sous-section, nous parlons donc des variantes fan-in, fan-out, right-looking et left-looking des méthodes directes. Puis, nous présentons notre propre solution de gestion de la mémoire, qui est utilisable dans le cadre d'un solveur hybride avec plusieurs processus par domaine. Enfin, nous détaillons les résultats obtenus avec notre méthode.

2.2.1 Variantes fan-in, fan-out, right-looking et left-looking

Plusieurs variantes de l'algorithme 2 de factorisation d'une matrice creuse par blocs denses existent ; celles-ci diffèrent dans la manière de reporter les contributions des lignes 8-9. Pour ne pas alourdir le texte, ce qui suit ne parlera que des opérations effectuées sur les blocs-colonnes (c'est-à-dire dans toute la partie triangulaire inférieure de la matrice) ; dans tous les cas la même chose est faite, symétriquement, pour les blocs-lignes. De plus, nous parlerons uniquement de l'approche **supernodale**, le lecteur pouvant se référer à [Liu92, GKK97, ADKL01, L'E12] pour l'autre approche couramment employée, appelée multifrontale.

L'approche supernodale comporte trois variantes. La première, appelée **right-looking fan-out** [GHL86], groupe les contributions des lignes 8 à 9 de l'algorithme 2. Ainsi, lorsque des blocs de contributions $L_{(i),k}U_{k,(j)}$ et $L_{(j),k}U_{k,(i)}$ doivent être envoyés d'un bloc-colonne k à un bloc-colonne j détenu par un autre processus, l'envoi est effectué immédiatement. Cela a pour inconvénient qu'un grand nombre de messages est engendré entre les processus, puisque chaque bloc-colonne k peut demander jusqu'à $\mathcal{O}(|\,[k]\,|^2)$ envois.

C'est pour résoudre ce problème que la variante **right-looking fan-in** [AEL90] a été introduite. Lorsqu'un processus p a une contribution dans un bloc-colonne l détenu par le processus p', il agrège cette contribution dans un bloc mémoire local appelé « bloc d'agrégation » et noté $DL_{p,l}$ (pour la partie inférieure) [13]. Ce bloc représente la portion du bloc-colonne l dans laquelle le processus p a des contributions, compactée verticalement pour éviter le stockage de zéros inutiles. Lorsque la dernière contribution de p vers p' est reportée dans ce bloc, $DL_{p,l}$ est envoyé en une seule fois à p'. De cette manière, on retarde les communications et on limite leur quantité [14]. En revanche, cette économie a pour prix une consommation mémoire accrue [HNP91] : à tout moment, chaque processus p détient un ensemble de blocs $DL_{p,l}$ pour chaque processus p' à qui il a des contributions en attente d'envoi. La thèse [Hén01] montre que dans certains cas, ce surcoût mémoire peut devenir très important pour des matrices telles que `THREAD` et `CUBE47` de la collection [Dav] avec 32 processus. Pour ces problèmes, le stockage des seuls blocs de contributions agrégées peut dépasser le double de la mémoire nécessaire au stockage de la matrice.

13. Dans les sections suivantes, son symétrique pour les blocs-lignes sera noté $DU_{p,l}$.

14. On réduit également souvent le volume échangé, puisque les contributions se recouvrent. Ce n'est toutefois pas toujours le cas puisque le bloc $DL_{p,l}$, qui est le plus petit rectangle contenant toutes les contributions de p dans l, peut contenir des termes sans contribution et qui ne seraient donc pas envoyés dans le cas fan-out.

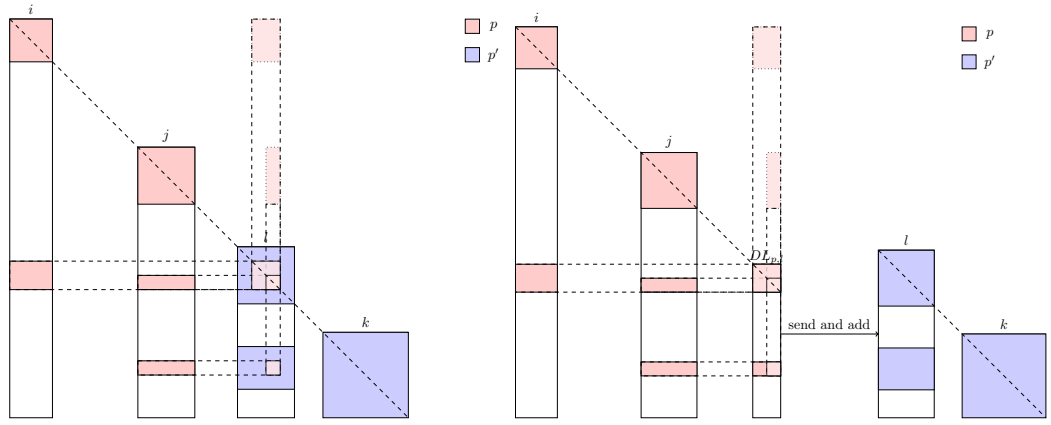


FIGURE 2.4 – Illustration des variantes fan-out et fan-in de l'algorithme LU right-looking. Dans la matrice représentée, les blocs-colonnes i et j détenus par le processus p ont chacun une contribution à apporter au bloc-colonne l appartenant à p'. Dans la version fan-out (à gauche), deux envois sont effectués : l'un au moment de la factorisation de i, l'autre au moment de celle de j. Au contraire, dans la version fan-in (à droite), le processus p agrège ses contributions vers l dans un bloc $DL_{p,l}$. Une fois la dernière contribution reportée (c'est-à-dire lorsque j est factorisé), le bloc est envoyé en une seule fois à p'.

La figure 2.4 illustre les variantes fan-in et fan-out.

Enfin, une troisième possibilité consiste à utiliser un algorithme **left-looking** [DER86, NP93]. Jusqu'à présent, un bloc-colonne k était factorisé puis ses contributions vers les blocs-colonnes $k' > k$ correspondant à ses ascendants dans l'arbre d'élimination par bloc étaient reportées. L'approche left-looking propose de faire le contraire : lors de la factorisation d'un bloc-colonne k, les contributions provenant des fils de k sont reportées, puis le bloc diagonal est factorisé et le TRSM effectué sur les blocs extra-diagonaux. Cette variante est présentée dans l'algorithme 5 (voir notations sur la figure 1.1 page 10). En pratique, elle est moins utilisée car elle implique un surcoût en temps à cause des communications. En revanche, à l'instar de la méthode right-looking fan-out, elle ne provoque aucun surcoût mémoire.

Nous détaillons à présent une variante des algorithmes précédents cherchant à faire un compromis entre le coût mémoire et le coût des communications. Elle a été implémentée dans le solveur PASTIX mais n'a pas été maintenue en raison de l'important surcoût en temps qu'elle induisait. L'idée était de fixer une limite mémoire m à ne pas dépasser sur chaque processus p, et à appliquer un algorithme fan-in systématiquement, excepté dans le cas où l'allocation d'un bloc était sur le point de provoquer le dépassement de la limite. Dans ce cas, un algorithme glouton consistant à envoyer et désallouer le ou les blocs de contributions partielles $DL_{p,l}$ les plus tardivement réaccédés était appliqué. De plus, toutes les situations de ce genre pouvant être prévues à l'avance (l'ordonnancement étant statique), l'envoi des blocs concernés était réalisé le plus tôt possible, c'est-à-dire lors de leur dernier accès précédant l'instant t de l'allocation problématique. Il était ainsi espéré que l'envoi du bloc serait terminé avant t. A la différence d'un algorithme fan-in « pur », toutefois, les blocs de contributions partielles envoyés prématurément devaient être réalloués et ré-envoyés par la suite.

Algorithme 5 : `leftLookingLU`

```
/* Version creuse par blocs denses et sur place */
Entrées : Matrice A = (A_ij) ayant N blocs-colonnes
Sorties : Matrices L et U telles que LU = A
1 pour k = 1 a N faire
    /* Calcul des contributions provenant des blocs-colonnes à gauche et
       blocs-lignes en haut */
2   pour l ∈ BRow(k) faire
3     j ←< l, k >;
4     pour i ∈ [l] tel que i ≥ j faire
5       L_(i),(j) ← L_(i),(j) − L_(i),l U_l,(j);
6       U_(j),(i) ← U_(j),(i) − L_(j),l U_l,(i);
    /* Factorisation du bloc diagonal */
7   Factoriser A_k,k en L_k,k et U_k,k;
    /* TRSM sur les blocs extra-diagonaux du bloc-colonne et du bloc-ligne
       */
8   pour i ∈ [k] faire
9     L_(i),k ← A_(i),k U_k,k^-1;
10    U_k,(i) ← L_k,k^-1 A_k,(i);
```

Pour finir, il est à noter que d'autres variantes s'efforçant de trouver un compromis mémoire/communications ont été proposées. Citons notamment Cleve Ashcraft dans [Ash93], qui conjugue les approches fan-in et fan-out dans une approche plus globale appelée *fan-both*.

2.2.2 Gestion optimisée de la mémoire

Nous exposons à présent une contribution de cette thèse, qui combine plusieurs des techniques évoquées dans le paragraphe précédent.

Notre cadre de travail est le suivant. Nous supposons qu'un solveur hybride utilisant plusieurs processus par domaine est employé. Chaque matrice de domaine A_i en entier, constituée des sous-matrices B_i, E_i, F_i et C_i (les C_i se recouvrant partiellement), est donc fournie à un solveur direct pour être factorisée avec plusieurs processus MPI. Par « factorisation d'un domaine », nous entendons ici les étapes consistant à factoriser B_i en $L_{B_i}U_{B_i}$, calculer les matrices de couplage $G_i = L_{B_i}^{-1}F_i$ et $W_i = E_iU_{B_i}^{-1}$ et en déduire le complément de Schur local $S_i = C_i - W_iG_i$. Comme nous l'avons expliqué au chapitre 1, ces trois étapes sont équivalentes à la factorisation right-looking de la matrice A_i en s'arrêtant à la dernière colonne de B_i.

Nous avons décidé de nous focaliser plus particulièrement sur le solveur direct PaStiX. En plus d'effectuer la factorisation avec plusieurs processus MPI, celui-ci offre la possibilité intéressante d'utiliser plusieurs threads par processus. Dans la version actuelle, PaStiX utilise un algorithme fan-in pur afin de gérer les contributions entre des processus différents et un algorithme fan-out pur pour régir celles d'un thread à un autre. PaStiX a en effet été conçu pour être lancé avec un processus MPI par nœud et un thread par cœur, se conformant ainsi à l'architecture sous-jacente ; les choix précédents sont donc justifiés par une volonté d'économiser

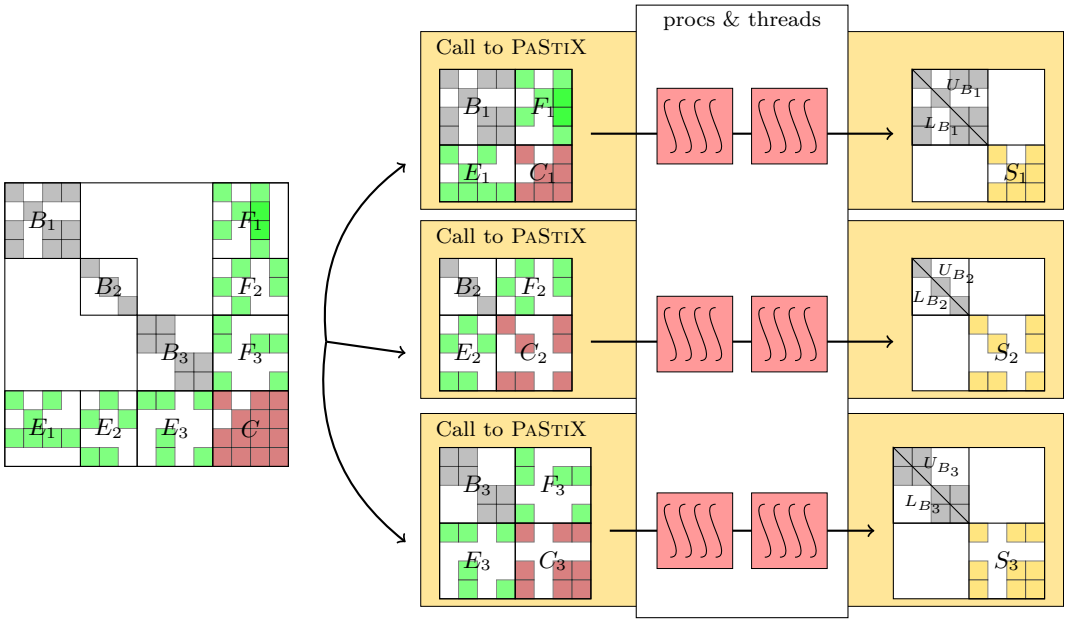


FIGURE 2.5 – Les trois niveaux de parallélisme exploitables par une approche hybride. Ici, on a 3 domaines, et chaque domaine fait l'objet d'un appel parallèle au solveur PaStiX. Ce dernier utilise lui-même 2 processus et 4 threads par processus par appel. Le nombre total de threads exécutés en parallèle est donc 24.

les communications d'un nœud à l'autre et de réduire la mémoire pour les contributions en mémoire partagée. Une limitation de cette analyse est que l'utilisation du fan-out pour toutes les contributions locales peut parfois entraîner des problèmes de contention mémoire ; il a ainsi été montré dans [HRR05] qu'on peut obtenir de meilleures performances temporelles en lançant PaStiX avec plusieurs processus MPI par nœud. Évidemment, le pic mémoire sur chaque processus augmente avec le nombre total de processus employés ; le paradigme « plusieurs processus par nœud » ne peut donc être utilisé que lorsque la mémoire disponible est suffisante.

Le cadre décrit précédemment est résumé par la figure 2.5. On y voit apparaître les trois niveaux de parallélisme de notre approche : les différents domaines sont indépendants les uns des autres, et chaque domaine peut être factorisé en parallèle avec plusieurs processus et plusieurs threads par processus. Les trois variables que sont nombre de domaines, le nombre de processus MPI par domaine et le nombre de threads par processus sont choisies par l'utilisateur.

Lors d'un appel à PaStiX effectué par un solveur hybride sur un domaine, deux types de surcoûts mémoire peuvent apparaître. Le premier, déjà évoqué, est le surcoût dû aux blocs $DL_{p,l}/DU_{p,l}$ de l'algorithme fan-in, d'autant plus important que le nombre de processus par domaine est élevé. Le second est lié au préconditionneur du solveur hybride : comme expliqué au chapitre 1, les matrices de couplage W_i et G_i ne sont finalement pas conservées. Pendant toute la factorisation, elles constituent donc une surconsommation mémoire, jusqu'à ce que toutes les contributions dans C_i aient été reportées et qu'elles puissent être libérées. Contrairement aux surcoûts des blocs $DL_{p,l}/DU_{p,l}$, le surcoût des matrices W_i et G_i diminue avec le nombre de processus par domaine, puisque le couplage est réparti entre les processus.

Dans toute la suite, nous nous concentrerons sur un unique appel à PaStiX sur la matrice d'un domaine, et nous allons voir comment réduire la mémoire lors de cet appel. Les indices i indiquant le numéro de domaine (B_i, E_i, etc.) seront omis. De plus, au sein d'une matrice de domaine, la partie haute des blocs-colonnes (correspondant à la matrice B), de même que la partie gauche des blocs-lignes, sera désignée par « la partie directe », puisque la matrice B est factorisée de manière directe et (la plupart du temps) exacte. Les parties basse des blocs-colonnes et droite des blocs-lignes, qui correspondent aux matrices E (ou W) et F (ou G), seront quant à elles appelées « la partie couplage ». La justification de ce nom tient au fait que dans le graphe d'adjacence, les arêtes de E et F sont celles qui relient les sommets de l'intérieur du domaine à ceux de l'interface. Les intervalles de lignes/colonnes correspondants à la partie directe et à la partie couplage seront notés respectivement I_{direct} et $I_{couplage}$ dans la suite.

Nous allons à présent exposer deux idées complémentaires afin de parvenir à masquer les surcoûts mémoires. La première sous-section ci-dessous propose un ordonnancement optimisé des allocations et désallocations. La seconde entreprend de profiter des avantages de l'algorithme left-looking, qui n'est actuellement pas utilisé par PaStiX, afin de pouvoir libérer le plus tôt possible les blocs du couplage.

2.2.2.1 Version right-looking pure avec allocation dynamique des blocs

Une première idée pour réduire la consommation mémoire est de retarder l'allocation de chaque bloc jusqu'au moment où il est réellement requis. Au cours de la factorisation, l'allocation d'un bloc q est requise dans deux cas :

— lorsqu'un bloc-colonne z auquel il appartient est factorisé ;
— lorsqu'une contribution provenant d'un bloc-colonne sur la gauche de z (qui peut être local ou non-local) doit être reportée dans q.

La seconde idée tend à réduire le surcoût mémoire dû aux matrices de couplage W et G. Si on considère les blocs-colonnes de la matrice W un par un, on peut noter qu'un bloc-colonne peut être libéré dès qu'il a été traité ; il en va de même pour les blocs-lignes de G. Cela est dû au fait qu'un algorithme right-looking est utilisé pour effectuer la factorisation. Ainsi, quand le bloc-colonne z a été factorisé, on peut libérer tous les blocs dans la partie couplage de z.

Ces idées sont illustrées sur la figure 2.6 dans le cas d'un unique processus MPI pour le domaine considéré. Au début, aucun bloc n'est alloué. Lorsque le bloc-colonne 0 est traité, tous les blocs de ce bloc-colonne sont alloués. A cause des contributions à droite, presque tous les blocs des blocs-colonnes 1 et 2 doivent être alloués également. Ensuite, la partie couplage du bloc-colonne 0, devenue inutile, est libérée, et la factorisation du bloc-colonne 1 est effectuée. Cela conduit à allouer le dernier bloc non-encore alloué du bloc-colonne 1. Comme on peut le voir, les contributions provenant du bloc-colonne 1 ne requièrent aucune nouvelle allocation. Ensuite, les blocs de couplage du bloc-colonne 1 sont libérés et la factorisation continue sur les blocs-colonnes suivants.

2.2.2.2 Version mixte left-right-looking pour le traitement du couplage

Le problème de l'algorithme précédent est qu'une grande quantité de blocs peuvent être alloués de manière très précoce. Cet état de fait est lié à l'utilisation d'un algorithme right-

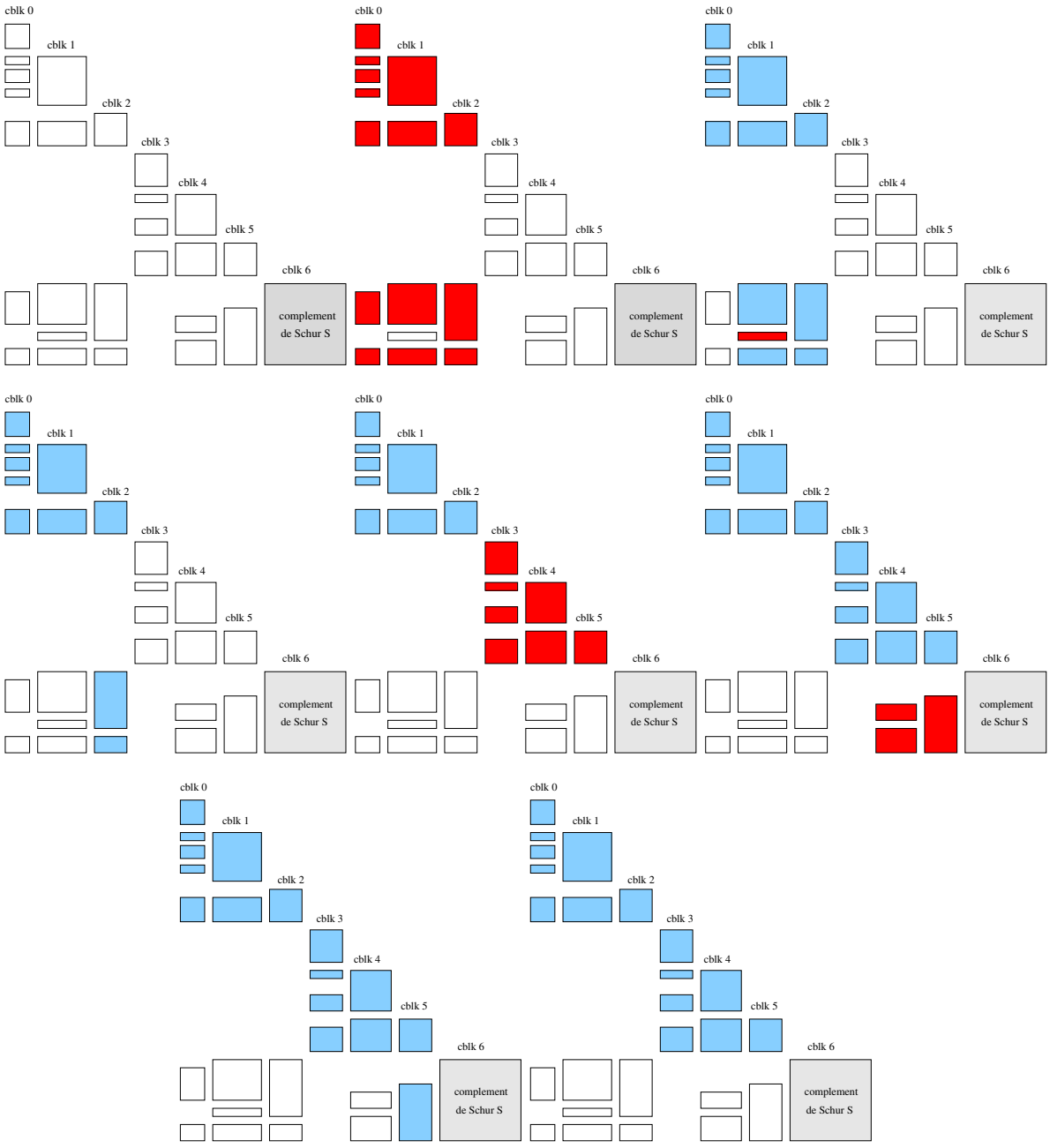


FIGURE 2.6 – Illustration de l'algorithme consistant à allouer la mémoire pour les blocs aussi tard que possible et à libérer les blocs du couplage aussitôt que possible. Un seul domaine est montré, factorisé avec un unique processus. Chaque étape correspond à la factorisation d'un bloc-colonne. Les blocs non encore alloués sont en blanc, les blocs nouvellement alloués en rouge et les autres blocs alloués en bleu.

looking : la factorisation d'un bloc-colonne est responsable de l'allocation de nombreux blocs à sa droite. Pour résoudre ce problème, une solution serait d'utiliser un schéma left-looking pour ce qui concerne les contributions locales en mémoire partagée ; pour les contributions non-locales, en mémoire distribuée, on conserverait un schéma right-looking fan-in permettant de limiter les communications. Cependant, l'approche left-looking empêche de libérer les blocs de couplage dès que le bloc-colonne correspondant a été factorisé, car ces blocs peuvent générer des contributions vers les blocs-colonnes suivants. C'est la raison pour laquelle nous avons décidé de faire une version mixte left-right-looking. Ainsi, dans le but d'optimiser les communications, nous employons toujours un algorithme de type right-looking fan-in pour toutes les contributions distantes en mémoire distribuée, aussi bien pour les celles de la partie directe que pour celles de la partie couplage. En revanche, pour les contributions locales en mémoire partagée, la stratégie suivie dépend de la partie dans laquelle se trouve le bloc : un schéma left-looking est utilisé dans la partie directe, ce qui permet d'éviter l'allocation de nouveaux blocs à droite ; et un schéma right-looking fan-out est employé pour la partie couplage, afin de pouvoir libérer les blocs de couplage au plus tôt.

Cette nouvelle stratégie est illustrée par la figure 2.7, sur la même matrice que la figure 2.6. Cette fois-ci, lorsque le bloc-colonne 0 est traité, les contributions de la partie directe ne sont pas reportées. Celles de la partie couplage en revanche le sont toujours, autorisant ainsi la libération des blocs du couplage. Ensuite, le bloc-colonne 1 est traité. Les blocs de la partie directe, ainsi que les blocs non-déjà alloués dans la partie couplage, sont alloués. Les contributions provenant des blocs-colonnes sur la gauche (*i.e.*, le bloc-colonne 0 seulement) sont reportées dans la partie directe du bloc-colonne 1. Les contributions du couplage du bloc-colonne 1 sont reportées à droite, ce qui ne requière pas de nouvelles allocations dans cet exemple. Après que le couplage du bloc-colonne 1 a été libéré, le bloc-colonne 2 est traité, et ainsi de suite.

Pour finir, la méthode est décrite dans l'algorithme 6. Pour chaque bloc-colonne/ligne local k, le processus p commence par allouer et initialiser les blocs qui ne le sont pas encore. Puis, il reporte de façon "left-looking" les contributions locales impactant la partie directe de k (lignes 3 à 7). Le bloc diagonal est ensuite factorisé et l'opération BLAS TRSM effectuée. Les contributions du bloc-colonne/ligne k devant être reportées le sont alors : d'une part celles qui impactent la partie couplage d'un bloc-colonne/ligne local (lignes 16 à 19), d'autre part celles qui arrivent dans un bloc-colonne/ligne distant (lignes 21 à 24). A chaque fois, les blocs sont préalablement alloués et initialisés si cela n'a pas déjà été fait. Dans le cas "distant", ce sont les blocs d'agrégation de contributions $DL_{p,l}/DU_{p,l}$ qui sont utilisés à la place. Ceux-ci sont envoyés et libérés (lignes 27-28) dès que la dernière contribution les modifiant a été reportée. Enfin, la partie couplage du bloc-colonne/ligne k est libérée.

Afin d'être exhaustif dans la présentation des allocations et désallocations effectuées, nous donnons dans l'algorithme 7 le code exécuté lors de la réception de blocs $DL_{p',l}/DU_{p',l}$ par un processus p : le ou les blocs destinataires de ces contributions sont alloués et initialisés si besoin, et les contributions sont ajoutées aux bons endroits.

2.2.3 Expériences numériques

Nous présentons à présent des expériences numériques afin de valider les idées introduites dans les paragraphes précédents. Pour les réaliser, nous avons choisi deux matrices challenge appelées `audi` et `haltere` présentées dans le tableau 2.1. Nous avons utilisé le solveur MaPHyS

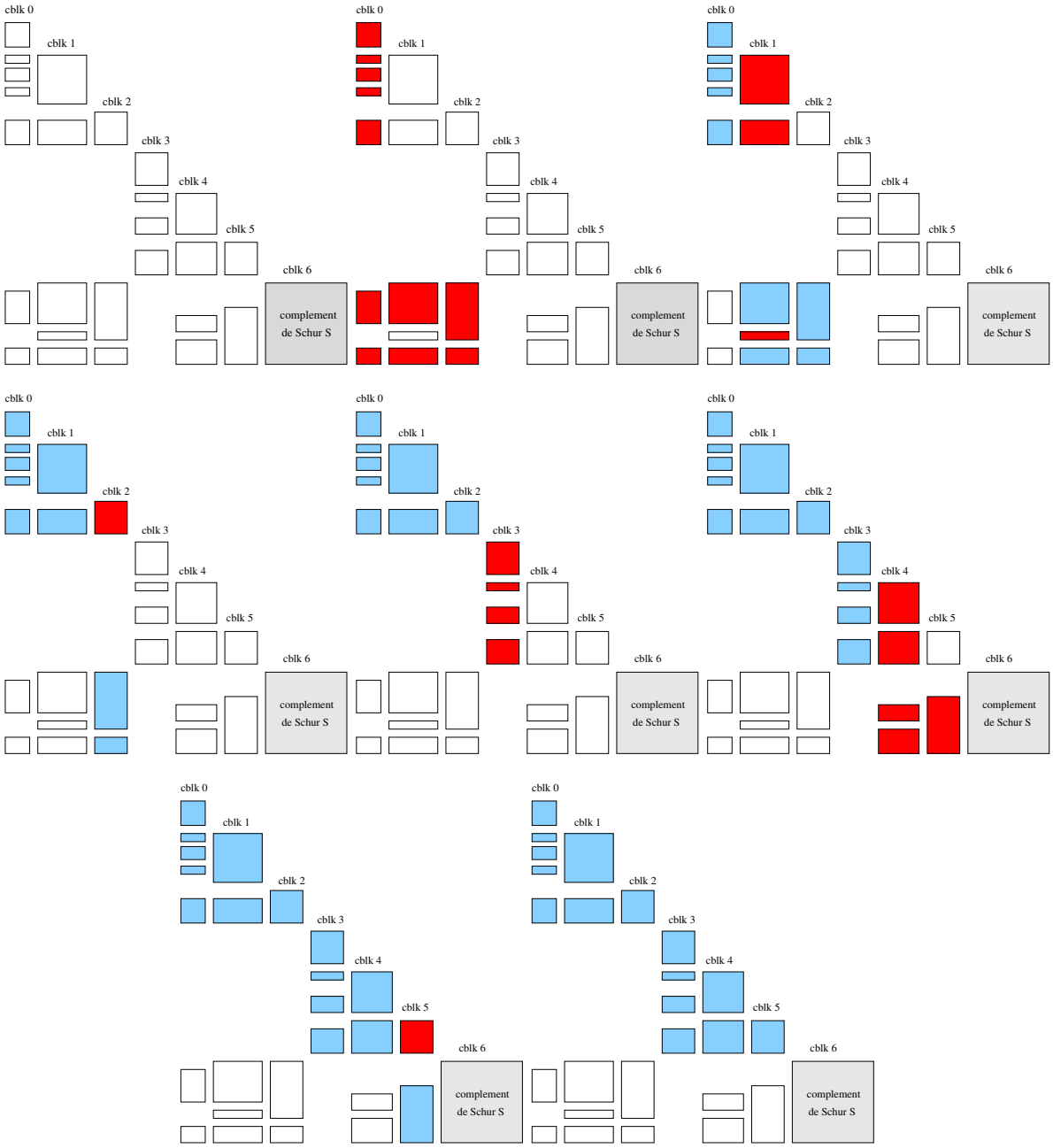


FIGURE 2.7 – Illustration de l'algorithme left-right-looking. Un seul domaine est montré, factorisé avec un unique processus. Les conventions pour les couleurs sont les mêmes que dans la figure 2.6.

Algorithme 6 : `leftRightLookingLU`

Entrées : Matrice A (représentant un domaine) ayant N_{direct} blocs-colonnes en-dehors du Schur
Entier p représentant le rang du processus
Sorties : Matrices L et U telles que $LU = A$

pour $k = 1$ **a** N_{direct} *tel que* k *est possédé localement par* p **faire**
  Allouer et initialiser les blocs non-encore alloués du bloc-colonne k;
  `/* Left-looking pour les contributions des blocs-colonnes locaux impactant la partie` $I_{directe}$ `du bloc-colonne` k `*/`
  pour $l \in BRow(k)$ *tel que* l *est possédé localement par* p **faire**
    $j \leftarrow < l, k >$;`/* Nécessairement,` $j \subset I_{direct}$ `*/`
    pour $i \in [l]$ *tel que* $i \geq j$ *et* $i \subset I_{direct}$ **faire**
      $L_{(i),(j)} \leftarrow L_{(i),(j)} - L_{(i),l}U_{l,(j)}$;
      $U_{(j),(i)} \leftarrow U_{(j),(i)} - L_{(j),l}U_{l,(i)}$;
  Vérifier que tous les blocs d'agrégation de contributions distants impactant le bloc-colonne k ont été reçus;
  `/* Factorisation du bloc diagonal */`
  Factoriser $A_{k,k}$ en $L_{k,k}$ et $U_{k,k}$;
  `/* TRSM sur les blocs extra-diagonaux du bloc-colonne` k `*/`
  pour $i \in [k]$ **faire**
    $L_{(i),k} \leftarrow A_{(i),k}U_{k,k}^{-1}$;
    $U_{k,(i)} \leftarrow L_{k,k}^{-1}A_{k,(i)}$;
  pour $j \in [k]$ **faire**
    $l \leftarrow right(A_{(j),k})$;
    si l *est possédé localement par* p **alors**
      `/* Right-looking pour les contributions du bloc-colonne` k `vers la partie` $I_{couplage}$ `de blocs-colonnes locaux */`
      pour $i \in [k]$ *tel que* $i \geq j$ *et* $i \subset I_{couplage}$ **faire**
        Si nécessaire, allouer et initialiser les blocs contenant $L_{(i),(j)}$ et $U_{(j),(i)}$ (dans le bloc-colonne/bloc-ligne l);
        $L_{(i),(j)} \leftarrow L_{(i),(j)} - L_{(i),k}U_{k,(j)}$;
        $U_{(j),(i)} \leftarrow U_{(j),(i)} - L_{(j),k}U_{k,(i)}$;
    sinon
      `/* Right-looking pour toutes les contributions du bloc-colonne` k `vers des blocs-colonnes distants */`
      Si nécessaire, allouer et initialiser les blocs $DL_{p,l}$ et $DU_{p,l}$;
      pour $i \in [k]$ *tel que* $i \geq j$ **faire**
        $DL_{p,l,(i),(j)} \leftarrow DL_{p,l,(i),(j)} - L_{(i),k}U_{k,(j)}$;
        $DU_{p,l,(j),(i)} \leftarrow DU_{p,l,(j),(i)} - L_{(j),k}U_{k,(i)}$;
      si *c'était les dernières contributions du processus* p *dans le bloc-colonne* l **alors**
        $p' \leftarrow process_owner(l)$;
        Envoyer à p' le bloc d'agrégation contenant les contributions vers l;
        Libérer le bloc d'agrégation contenant les contributions vers l;
  Libérer les blocs de la partie $I_{couplage}$ du bloc-colonne k;

Algorithme 7 : `fanInRecv`

```
/* Algorithme exécuté par un thread, parallèlement à leftRightLookingLU */
si on a reçu un bloc DL_{p',l} et DU_{p',l} alors
    Si nécessaire, allouer et initialiser le(s) bloc(s) correspondant(s) du bloc-colonne et du
    bloc-ligne l;
    Ajouter les blocs reçus aux bons endroits dans le(s) bloc(s) concerné(s);
```

afin d'effectuer une décomposition de domaine en 2 à 32 domaines. Puis, nous avons calculé la consommation mémoire lors de la factorisation d'un de ces domaines avec le solveur PaStiX, en utilisant les stratégies décrites précédemment.

matrice	**taille** n	**non-zéros** nnz	**type**
`audi`	943 695	39 297 711	réelle
`haltere`	1 288 825	10 476 775	complexe

Table 2.1 – Description des matrices utilisées pour les tests. Les deux sont symétriques ; le nombre de non-zéros concerne une moitié de la matrice.

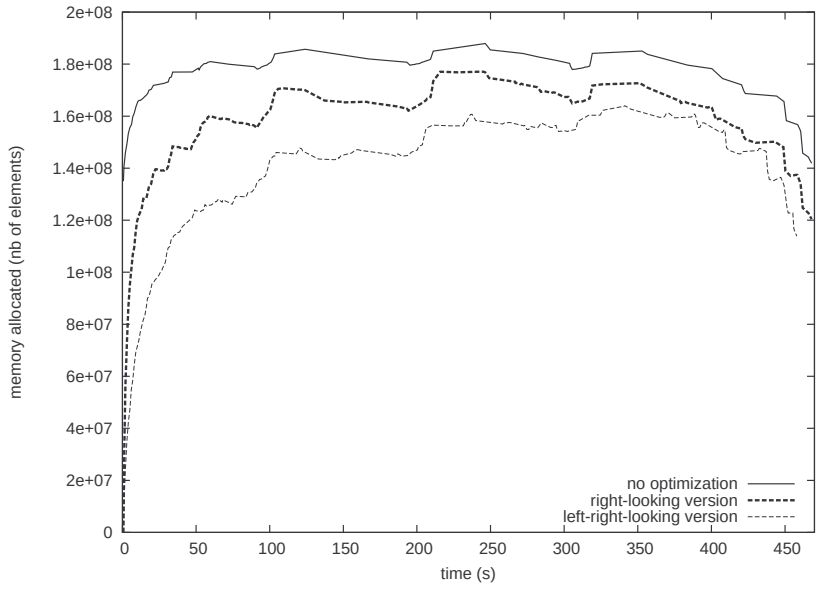


Figure 2.8 – Comparaison de la mémoire allouée durant la factorisation des intérieurs lorsque (a) aucune optimisation n'est faite, (b) la stratégie d'allocation est appliquée en utilisant un schéma right-looking et (c) la version mixte left-right-looking est employée. *Cas de test : domaine 1 de la matrice* ***audi*** *découpée en deux domaines ; 4 processus et 4 threads par processus ont été utilisés pour factoriser chaque domaine ; les courbes sont montrées pour le processus 1. Les temps sont basés sur des simulations.*

La figure 2.8 montre l'évolution de la consommation mémoire pendant la factorisation d'un domaine de la matrice `audi` découpée en deux domaines. Quatre processus et quatre threads par

processus ont été utilisé pour réaliser cette expérimentation. Les courbes montrent la consommation mémoire sur le processus 1 dans trois cas : lorsqu'aucune optimisation n'est faite (tous les blocs, ceux d'agrégation exceptés, sont alloués dès le début ; les libérations des parties couplages ont lieu en une fois, au dernier moment), lorsque la stratégie right-looking pure décrite au paragraphe 2.2.2.1 est utilisée et lorsque la stratégie mixte left-right-looking du paragraphe 2.2.2.2 est mise en œuvre. En comparant les deux courbes avec optimisations, on peut constater que la stratégie left-right-looking permet de faire augmenter plus graduellement la mémoire au début de la factorisation. Sur ce cas de test, la version mixte left-right-looking réduit le pic mémoire d'environ 10 % de la mémoire totale, et de près de 30 % si la mémoire finale est prise comme référence (*i.e.*, si nous prenons la mesure G_{pic} définie plus loin), ce qui est globalement satisfaisant. Les résultats sur la matrice `haltere` (table 2.3 présentée ultérieurement) sont quasiment du même ordre de grandeur.

Pour mesurer plus précisément la mémoire économisée avec notre méthode, nous avons introduit trois métriques comprises entre 0 et 1, définies comme suit :

$$G_{tot} = 1 - \frac{\sum_{p=0}^{nproc-1} m_p^*}{\sum_{p=0}^{nproc-1} m_p}\,; \quad G_{pic} = 1 - \frac{\sum_{p=0}^{nproc-1} m_p^* - r_p}{\sum_{p=0}^{nproc-1} m_p - r_p}\,; \quad G_{max} = 1 - \frac{\max_{p\in[\![0;nproc-1]\!]} m_p^*}{\max_{p\in[\![0;nproc-1]\!]} m_p}\,;$$

où $nproc$ est le nombre de processus, m_p le pic mémoire sur le processus p lorsqu'aucune optimisation n'est effectuée et m_p^* le pic mémoire sur le processus p lorsque les optimisations sont faites. G_{tot} représente la réduction moyenne de la mémoire sur tous les processus. Notez que le pic mémoire est au minimum égal à la mémoire finale, *i.e.* à la mémoire requise pour stocker L_B, U_B et S pour le domaine p considéré ; cette mémoire est notée r_p. Puisque la mémoire finale ne peut être réduite (à moins d'accepter de calculer un complément de Schur approché), G_{tot} est bornée par une constante strictement plus petite que 1. C'est pourquoi nous avons introduit G_{pic}, qui correspond à la même chose que G_{tot} mais prenant pour référence la mémoire finale au lieu de zéro. G_{pic} peut atteindre 1 si la mémoire utilisée sur chaque processus est en-deçà de r_p pendant toute la factorisation. Finalement, G_{max} donne le facteur de réduction mémoire sur le processus le pire. Si aucune optimisation n'est mise en œuvre, G_{tot}, G_{pic} et G_{max} sont tous nuls.

Les matrices `audi` et `haltere` ont été factorisées avec différents découpages en domaines et un nombre variable de processus et de threads ; les résultats sont présentés dans les tables 2.2 et 2.3. Lorsqu'on utilise un unique processus pour factoriser chacun des domaines de la matrice `audi` découpée en 8 domaines, le pic mémoire G_{pic} est réduit de 58% à 72% selon les domaines. Notez que tester sur un unique processus signifie l'absence de surcoût mémoire lié aux blocs d'agrégation ; ainsi, le pic mémoire dans ce cas est exclusivement causé par les matrices de couplage allouées temporairement. Puisque la quantité de mémoire potentiellement économisée par l'usage de notre nouvelle stratégie n'augmente pas lorsque $nproc$ augmente, la réduction du pic décroît lorsque $nproc$ augmente. La figure 2.9 reprend les résultats des tables et montre clairement cette baisse. Cependant, on observe toujours une réduction significative du pic lorsqu'on utilise jusqu'à 4 ou 8 processus par domaine.

La table 2.4 et la figure 2.10 montrent l'influence du nombre de threads utilisés par processus. On peut constater qu'il n'y a pas de dégradation significative de la réduction mémoire lorsque

ndom	num dom	taille domaine	taille Schur	*nproc*	G_{tot} (%)	G_{max} (%)	G_{pic} (%)
2	1	498 228	3 597	1	7,1	7,1	84,7
				4	9,9	11,0	29,0
				8	9,1	9,1	17,9
				16	4,0	3,6	6,2
2	2	449 064	3 597	1	7,1	7,1	83,3
				4	7,1	0,1	25,1
				8	8,6	8,6	20,3
				16	4,9	7,6	8,5
8	3	128 955	9 675	1	17,8	17,8	58,6
				4	9,0	8,6	13,6
				8	6,6	4,4	8,4
				16	2,3	0,6	2,7
8	5	112 656	3 585	1	13,7	13,7	72,3
				4	7,2	9,1	15,6
				8	4,7	0,1	7,8
				16	2,1	0,0	3,1
32	27	32 613	4 944	1	18,3	18,3	53,1
				4	9,9	6,3	13,8
				8	4,4	3,7	5,7
				16	2,3	0,1	2,8

TABLE 2.2 – Influence du nombre de processus par domaine sur la matrice `audi`. *Tous les tests ont été faits avec 4 threads par processus. « ndom » désigne le nombre total de domaines et « num dom » le numéro du domaine. Les tailles de domaine sont données en nombre de nœuds. Dans le cas de 8 domaines, les deux domaines choisis sont le plus petit (5) et le plus gros (3). Dans le cas de 32 domaines, il s'agit d'un domaine de taille moyenne (27).*

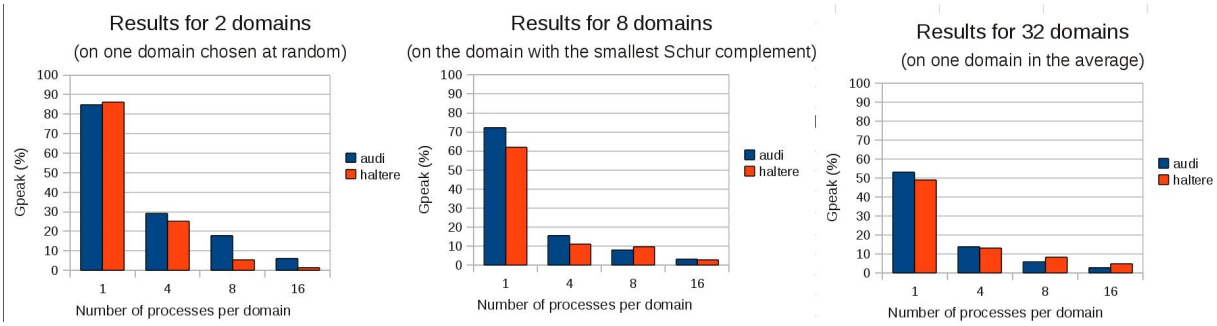


FIGURE 2.9 – Évolution de la mémoire économisée en fonction du nombre de processus par domaine, pour un découpage en 2, 8 ou 32 domaines. *Tous les tests ont été faits avec 4 threads par processus, sur les matrices* **`audi`** *(barres bleues) et* **`haltere`** *(barres oranges).*

ndom	num dom	taille domaine	taille Schur	*nproc*	G_{tot} (%)	G_{max} (%)	G_{pic} (%)
2	1	638 410	1 251	1	4,1	4,1	86,2
				4	6,3	6,5	25,0
				8	2,1	2,7	5,3
				16	0,8	0,0	1,5
2	2	651 666	1 251	1	4,0	4,0	86,0
				4	3,4	3,4	15,1
				8	1,6	0,6	4,3
				16	0,9	0,0	1,9
8	5	164 372	3 483	1	12,8	12,8	61,9
				4	5,8	4,5	11,0
				8	5,8	2,3	9,6
				16	1,9	1,5	2,8
8	6	165 586	4 038	1	12,6	12,6	59,7
				4	7,7	6,8	15,7
				8	5,4	3,2	8,8
				16	2,5	2,3	3,8
32	16	44 263	2 310	1	12,6	12,6	49,1
				4	7,3	6,1	13,1
				8	5,0	5,0	8,3
				16	3,1	4,7	4,8

TABLE 2.3 – Influence du nombre de processus par domaine sur la matrice `haltere`. *Tous les tests ont été faits avec 4 threads par processus. « ndom » désigne le nombre total de domaines et « num dom » le numéro du domaine. Les tailles des domaines sont données en nombre de nœuds. Dans le cas de 8 domaines, les deux domaines choisis sont le plus petit (5) et le plus gros (6). Dans le cas de 32 domaines, il s'agit d'un domaine de taille moyenne (16).*

	G_{tot} (%)			G_{max} (%)			G_{pic} (%)		
Nb de threads	1	4	8	1	4	8	1	4	8
1 processus	18,4	17,8	17,2	18,4	17,8	17,2	60,7	58,6	56,7
4 processus	8,3	9,0	7,3	8,1	8,6	6,3	12,2	13,6	11,0
8 processus	4,6	6,6	5,5	4,7	4,4	1,8	5,7	8,4	7,4
16 processus	3,2	2,3	2,8	2,4	0,6	0,8	3,6	2,7	3,6

TABLE 2.4 – Influence du nombre de threads par processus. *Cas de test : domaine 3 de la matrice* ***audi*** *découpée en 8 domaines.*

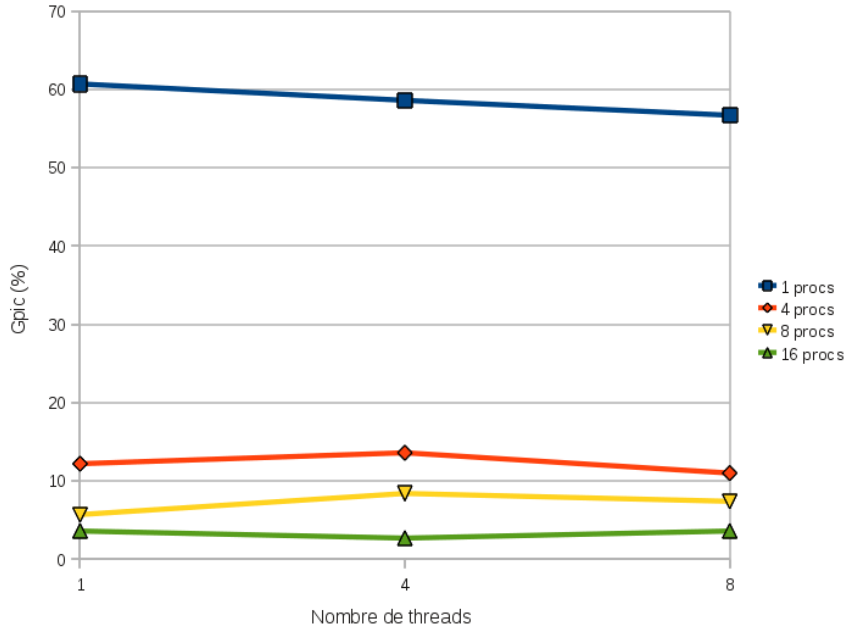


FIGURE 2.10 – Évolution de la mémoire économisée en fonction du nombre de threads par processus. *Cas de test : domaine 3 de la matrice* **audi** *découpée en 8 domaines.*

le nombre de threads augmente.

Comme expliqué auparavant, la méthode de décomposition de domaine autorise plusieurs niveaux de parallélisme. Pour les coupler, on peut faire varier 3 paramètres : le nombre de domaines dans la décomposition (ou, de manière équivalente, la taille moyenne des domaines), le nombre de processus MPI pour factoriser un domaine et le nombre de threads par processus MPI. Nos résultats suggèrent que nous devrions garder un nombre de processus par domaine relativement bas. On peut contrebalancer cela en multipliant le nombre de domaines, rendant chacun d'entre eux plus petit. De plus, les résultats montrent que le nombre de threads par processus peut être augmenté ; on a alors une meilleure scalabilité sans dégradation de la mémoire. Dans ces conditions, nos optimisations permettent une réduction significative de la consommation mémoire.

Conclusion

Dans ce chapitre, nous avons présenté de multiples optimisations pouvant être mises en œuvre afin de réduire la mémoire utilisée par les solveurs hybrides ; on peut par exemple s'appuyer sur un seuillage numérique, limiter les blocs de S à un pochoir prédéfini et, dans le cas où chaque processus est responsable de plusieurs domaines, gérer l'allocation et la libération des blocs de couplage domaine par domaine. De plus, la deuxième partie du chapitre a présenté une contribution de cette thèse, valable lorsqu'un ensemble de processus est employé pour chaque domaine ; elle consiste à utiliser un ordonnancement mixte left-right looking et à réaliser une gestion fine des allocations et libérations des blocs. Certaines des optimisations susmentionnées

peuvent bien sûr être combinées entre elles.

Nous avons par ailleurs validé la nouvelle approche dans le cas où un nombre réduit de processus par domaine est utilisé. Dans ces conditions, la scalabilité peut toujours être assurée par un nombre plus grand de domaines et/ou de threads par processus. Cela est particulièrement bien adapté pour les noeuds composés de puces multi-cœurs.

Pour conclure le chapitre, nous évoquons brièvement deux autres méthodes, qui sont moins spécifiques aux solveurs hybrides que les précédentes.

Tout d'abord, on peut remarquer que toutes les matrices utilisées par les solveurs hybrides sont creuses par blocs denses. Une possibilité intéressante de réduction mémoire consiste donc à **utiliser des matrices hiérarchiques** (dites H-matrices) pour stocker certains blocs denses sous forme compressée [Hac99]. Le principe est de remplacer une matrice dense T de dimension $n \times m$ par deux matrices P et Q de rang r vérifiant $T = PQ$, où P est de taille $n \times r$ et Q $r \times m$. Lorsque cela n'est pas possible, on utilise une approximation : $T \approx PQ$. r devient alors un paramètre réglant la qualité de l'approximation. Plus r est petit, moins la qualité est bonne, mais en contrepartie plus la mémoire économisée est grande. Les opérations de type Tz, où z est un vecteur, peuvent être remplacées par $P(Qz)$; une telle opération coûte $\mathcal{O}(r(m+n))$ au lieu de $\mathcal{O}(nm)$, ce qui permet également d'économiser des calculs. À ce stade, P et Q sont appelées des matrices *low-rank*. On peut alors enfin remplacer chacune d'elle récursivement par d'autres produits de matrices *low-rank* ; c'est ce qu'on appelle une H-matrice.

Une seconde possibilité, qui n'est pas spécifique aux solveurs linéaires creux, consiste à **faire de l'out-of-core**. Il s'agit d'une méthode très générale pouvant s'appliquer à n'importe quel programme soumis à une contrainte mémoire forte. Le principe est simplement de transférer une partie de la mémoire utilisée sur le disque dur pour libérer de la mémoire centrale. L'enregistrement et le chargement depuis un disque dur étant coûteux, il convient alors d'optimiser ces transferts. Cela requiert en pratique d'avoir une distribution statique ou au moins partiellement prévisible des tâches, de manière à toujours déplacer sur le disque les blocs mémoire réaccédés les plus tardivement. Comme le choix des blocs à déplacer est lié à l'ordonnancement des tâches, il est alors commode d'utiliser un ordonnanceur de tâches tel que StarPU [ATNW11], et de lui déléguer la gestion de l'out-of-core.

À ce jour, les solveurs hybrides n'implémentent pas directement la fonctionnalité out-of-core, mais certains des solveurs directs qu'ils utilisent le font (entre autres MUMPS [ADL98], PaStiX [HRR02] et [TU08]).

Chapitre 3

Amélioration du partitionnement pour un meilleur équilibrage de la charge

Sommaire

3.1 Algorithme de bipartitionnement récursif 47
3.2 Multiniveau 49
3.3 Raffinement d'un séparateur : l'algorithme de Fiduccia-Mattheyses 53
3.4 Algorithmes de partitionnement 59
3.4.1 Greedy graph growing 60
3.4.2 Double greedy graph growing 64
3.4.3 Halo-first greedy graph growing 69
3.5 Résultats 75
3.5.1 Etude de scalabilité 76
3.5.2 Résultats dans le solveur hybride MaPHyS 78

Le chapitre précédent s'intéressait à limiter la mémoire utilisée par les solveurs linéaires creux hybrides. Nous nous concentrons à présent sur l'aspect temporel et plus précisément sur l'équilibrage de la charge entre les processus.

Rappelons d'abord le fonctionnement des méthodes hybrides, en se focalisant plus particulièrement sur les choix faits pour le solveur hybride MaPHyS. Dans un premier temps, un partitionneur de graphe tel que Scotch [PR96] ou MeTiS [KK98b] est appelé afin de diviser le graphe en domaines de tailles équivalentes. Ensuite, chaque domaine i, constitué de son intérieur B_i de taille n_i, de son interface C_i de taille s_i et du couplage entre les deux est extrait et passé en entrée à un solveur direct (figure 1.5 page 19). Celui-ci effectue une factorisation right-looking des colonnes associées aux nœuds de la partie intérieure, formant ainsi les matrices locales L_{B_i}, U_{B_i} et S_i. Cette étape est appelée la factorisation des domaines. Le Schur local S_i (matrice dense de taille s_i) est alors assemblé en une matrice $\overline{S_i}$ puis factorisé, soit de manière exacte, soit de manière approchée avec un seuillage ; c'est la phase de calcul du préconditionneur. Ce dernier est enfin utilisé pour itérer sur le système réduit au complément de Schur (équations (1.6) page 13

et (1.10) page 22), en se ramenant à des itérations locales par domaine et en assemblant le vecteur solution jusqu'à convergence.

Plaçons-nous dans le cadre des graphes correspondant à des maillages d'éléments finis 3D qui constitue le domaine naturel applicatif de ce travail de thèse. Si les nœuds intérieurs de chaque domaine sont renumérotés par dissections emboîtées (voir section 1.1), le coût opératoire global associé au domaine i est asymptotiquement :

$$\underbrace{\mathcal{O}(n_i^2 + n_i^{4/3} s_i + n_i^{2/3} s_i^2)}_{\text{factorisation du domaine}} + \underbrace{\mathcal{O}(s_i^3)}_{\text{calcul du préconditionneur}} + \underbrace{\mathcal{O}(nb_iterations \times s_i^2)}_{\text{résolution itérative}} \tag{3.1}$$

si les $\overline{S_i}$ sont factorisés exactement (sans seuillage). On pourra consulter [LRT79] pour le premier terme de cette somme asymptotique. Pour les maillages 3D correspondant à des objets ayant un *bon aspect ratio* (voir [MV91]), ce qui est très souvent le cas pour les problèmes de simulation numérique dans lesquels une approximation par éléments finis est utilisée, s_i varie comme un $\mathcal{O}(n_i^{2/3})$. On voit donc clairement le poids important de la taille s_i du Schur local dans le coût opératoire qui est alors en $\mathcal{O}(n_i^2)$ pour les deux premiers termes, le coût opératoire du dernier dépendant plus spécifiquement du nombre d'itérations. Dans le cas où les $\overline{S_i}$ ont été seuillées, le coût opératoire du deuxième terme peut être plus faible avec cependant un nombre d'itérations généralement plus important dans le troisième terme, mais cela n'a pas d'impact dans le premier terme pour la factorisation du domaine.

On constate aussi que dans un cadre parallèle où les domaines sont distribués, l'équilibrage de la charge va donc dépendre tout autant de l'équilibrage des intérieurs de taille n_i que de l'équilibrage des interfaces de taille s_i. Cependant, jusqu'à aujourd'hui, les partitionneurs ne se sont intéressés exclusivement qu'à l'équilibrage des intérieurs.

L'objet de ce chapitre est donc d'étudier les algorithmes existants utilisés par les partitionneurs de graphes et de proposer des modifications afin de prendre en compte le critère supplémentaire d'équilibrage des interfaces. Deux algorithmes principaux sont utilisés par les partitionneurs de graphes. Le premier, appelé *k-way*, consiste à construire directement k parties dans le graphe [KK98c]. Nous allons nous concentrer ici sur le second, le *bipartitionnement récursif*, qui propose de découper le graphe en effectuant de multiples bipartitionnements imbriqués [15]. Ce choix est essentiellement dû au fait qu'il est plus aisé de satisfaire un critère multiple en découpant un graphe en $k = 2$ parties qu'en $k > 2$ parties.

Le premier paragraphe de ce chapitre explique plus en détail l'algorithme de bipartitionnement récursif. Il permettra, notamment, de définir la notion de *sommet halo*, et montrera que l'équilibrage de ces sommets particuliers suffit dans la majorité des cas à garantir l'équilibrage des interfaces des domaines construits. La section 2 présentera l'algorithme de multiniveau utilisé par le bipartitionnement récursif pour trouver un séparateur à chaque étape. Cet algorithme utilise lui-même deux sous-algorithmes afin de trouver et de raffiner un séparateur. Nous exposerons successivement ceux-ci dans les sections suivantes, ainsi que les variantes que nous proposons pour équilibrer les sommets halos. Enfin, nous présenterons les résultats obtenus avec nos algorithmes modifiés pour vérifier la validité et la qualité de ceux-ci.

15. Cette approche a été introduite pour obtenir des numérotations de type « dissection emboîtée » [Geo73, LRT79, HR98] pour les solveurs directs creux parallèles.

Afin de séparer clairement l'existant des contributions apportées par cette thèse dans les sections suivantes, nous précéderons ces dernières du marqueur ▶ tout au long de ce chapitre.

3.1 Algorithme de bipartitionnement récursif

Nous commençons ici par présenter l'algorithme de bipartitionnement récursif [Geo73, HR98] utilisé pour diviser un graphe en domaines et son adaptation afin d'équilibrer les interfaces des domaines créés.

Dans un graphe $G = (V, E)$, si (P_0, S, P_1) est une partition de V, on appelle S un *séparateur-sommet* (ou simplement un *séparateur*) si tous les chemins partant d'un sommet de P_0 et arrivant à un sommet de P_1 passent par un sommet de S. Le principe du bipartitionnement récursif consiste à trouver un séparateur S de taille aussi réduite que possible et qui sépare le graphe en deux parties P_0 et P_1 de tailles équivalentes. L'algorithme est ensuite appelé récursivement sur le graphe induit par P_0 puis sur le graphe induit par P_1, jusqu'à ce qu'un critère d'arrêt défini à l'avance soit atteint (par exemple, un nombre de sommets restants ou un nombre de niveaux de récursion).

Algorithme 8 : `bipartitionnementRécursif`

Entrée : Graphe $G = (V, E)$
Sortie : Arbre dont chaque nœud représente un ensemble de sommets. Chaque feuille correspond à l'intérieur d'un domaine, chaque nœud interne à un séparateur.

```
1 si critereArretAtteint(G) alors
2 |  retourner arbre(V, ∅, ∅);
3 sinon
4 |  (P0, S, P1)   ← trouverSéparateur (G);
5 |  filsGauche ← bipartitionnementRécursif (P0, E ∩ (P0 × P0));
6 |  filsDroit   ← bipartitionnementRécursif (P1, E ∩ (P1 × P1));
7 |  retourner arbre(S, filsGauche, filsDroit);
```

Cette méthode conduit à l'algorithme 8. Il construit petit à petit un arbre dont les nœuds internes contiennent les séparateurs et les feuilles les intérieurs des domaines. La figure 3.1 (à gauche) montre un exemple de bipartitionnement sur un graphe avec deux niveaux de récursivité, produisant ainsi 4 domaines. On peut remarquer que si le séparateur trouvé à chaque niveau équilibre correctement les parties, alors les intérieurs des domaines seront également bien équilibrés. En revanche, l'algorithme ne fournit aucune garantie sur l'équilibrage des interfaces. Ainsi, dans l'exemple montré, le domaine du milieu à droite est adjacent à l'intégralité du séparateur du 1^{er} niveau, alors que le domaine de droite n'est délimité que par le séparateur de 2^e niveau qui le jouxte, conduisant à un déséquilibre des interfaces dont les tailles vont de 4 à 8.

▶ On peut remarquer, pourtant, que ce déséquilibre aurait pu être évité. Si on reprend l'algorithme original de dissection emboîtée de Lipton, Rose et Tarjan [LRT79], les auteurs avaient initialement décidé de garder la trace des sommets des anciens séparateurs pendant toute la récursion. Nous proposons, tout au au long de ce chapitre, de reprendre la même idée dans le but d'équilibrer les interfaces. Plus précisément, nous définissons le *halo* d'un ensemble

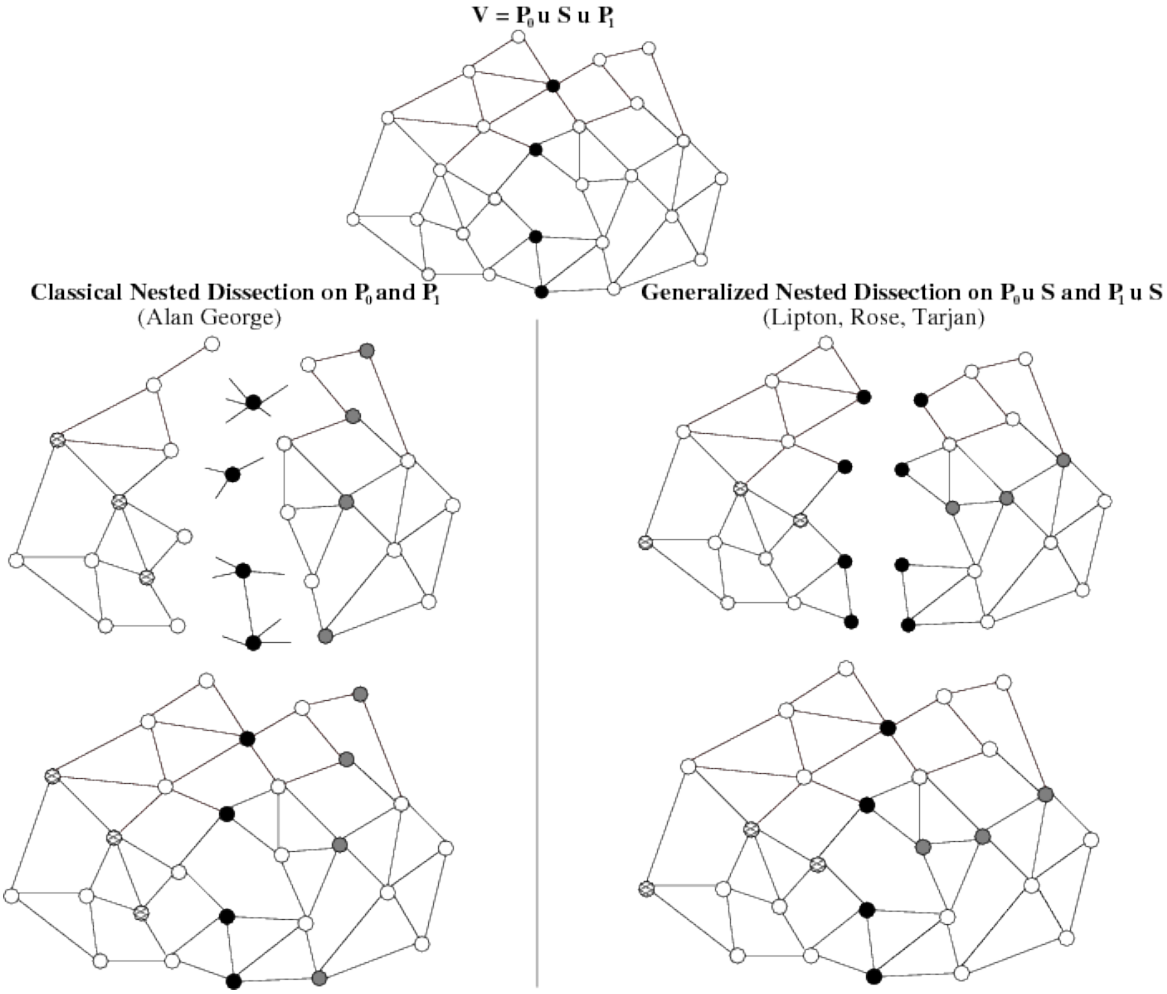


FIGURE 3.1 – *A gauche :* la récursion classique est effectuée sur P_0 et P_1. L'objectif est d'équilibrer les tailles des sous-graphes et de minimiser la taille des séparateurs. Cependant, les tailles des interfaces, représentées par les nœuds en noir, gris ou hachuré, peuvent être déséquilibrées : elles sont ici respectivement 4, 5, 6, et 8. *A droite :* la récursion est effectuée $P_0 \cup S$ et $P_1 \cup S$ et les sommets du halo sont équilibrés entre les parties, aboutissant à des tailles d'interface de 5 partout.

de sommets V' du graphe initial comme étant l'ensemble H des sommets dans $V \setminus V'$ qui sont adjacents à V'. Par ailleurs, nous supposons que la récursion est cette fois-ci effectuée sur chaque sous-graphe *avec* son halo. Le halo H d'une partie V' représente donc les sommets des séparateurs déjà construits jouxtant le sous-graphe. De plus, nous proposons de revisiter la fonction `trouverSéparateur` à la ligne 4 de l'algorithme 8 pour définir une nouvelle fonction `trouverSéparateurEquilibrantLeHalo` qui s'efforce d'équilibrer la répartition des sommets halo et non-halo dans les deux parties P_0 et P_1. De cette manière, chaque halo étant localement équilibré, les interfaces des domaines finalement produits par l'algorithme le sont aussi. Le nouveau bipartitionnement est donné par l'algorithme 9, dont la figure 3.1 (à droite) montre un exemple d'exécution : sur le même graphe que précédemment, les interfaces ont cette fois toutes une taille de 5.

Algorithme 9 : `bipartitionnementRécursifAvecHalo`

Entrée : Graphe avec halo $G = (V, H \subset V, E)$
Sortie : Arbre dont chaque nœud représente un ensemble de sommets. Chaque feuille correspond à l'intérieur d'un domaine, chaque nœud interne à un séparateur.

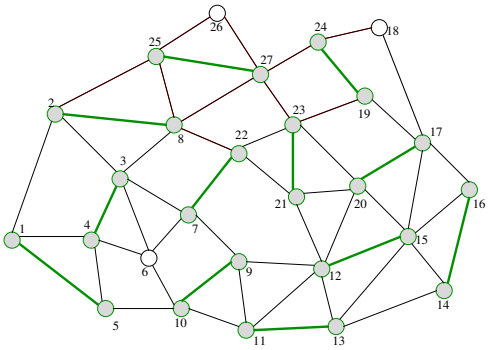
1 **si** $critereArretAtteint(G)$ **alors**
2 | **retourner** $arbre(V, \varnothing, \varnothing)$;
3 **sinon**
4 | $(P_0, S, P_1) \leftarrow$ `trouverSéparateurEquilibrantLeHalo` (G);
5 | $H_0 \leftarrow \{v \in H \cup S \mid \exists w \in P_0$ tel que $(v, w) \in E\}$;
6 | $H_1 \leftarrow \{v \in H \cup S \mid \exists w \in P_1$ tel que $(v, w) \in E\}$;
7 | $filsGauche \leftarrow$ `bipartitionnementRécursif` $(P_0, H_0, E \cap ((P_0 \cup H_0) \times (P_0 \cup H_0)))$;
8 | $filsDroit \leftarrow$ `bipartitionnementRécursif` $(P_1, H_1, E \cap ((P_1 \cup H_1) \times (P_1 \cup H_1)))$;
9 | **retourner** $arbre(S, filsGauche, filsDroit)$;

L'objet de la suite de ce chapitre est de mettre en œuvre la fonction `trouverSéparateurEquilibrantLeHalo`. Nous partirons pour cela de l'algorithme multiniveau utilisé par les partitionneurs pour réaliser la fonction `trouverSéparateur`.

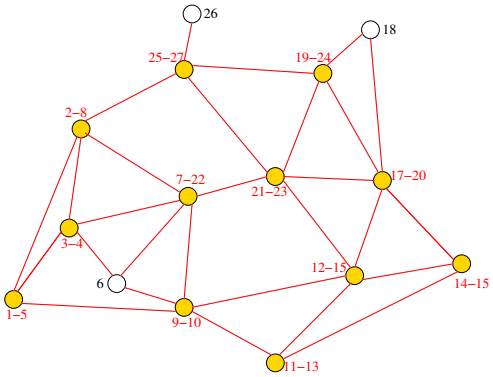
3.2 Multiniveau

Dans ce paragraphe, nous expliquons la méthode de multiniveau [HL95, KK98a] pour trouver un séparateur dans un graphe. Le multiniveau est en réalité une « méta-méthode » dans la mesure où il utilise plusieurs autres algorithmes. Le principe général est de compresser de multiples fois le graphe, d'utiliser un algorithme pour trouver un séparateur sur le graphe le plus grossier (le plus petit), puis de projeter et raffiner les séparateurs sur les graphes plus fins successifs. L'algorithme 10 décrit cette méthode. Comme nous le verrons plus loin, la fonction `trouverSéparateur` est réalisée par l'algorithme de *greedy graph growing* [KK98a] et la fonction de raffinement `raffinerSéparateur` par celui de *Fiduccia-Mattheyses* [FM82]. Un exemple d'exécution avec un seul niveau de compression est illustré en figure 3.2.

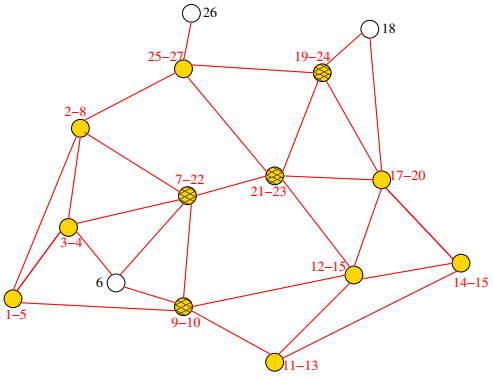
Avant la première compression, tous les sommets et arêtes ont un poids de 1. Pour réaliser une étape de compression d'un graphe (V_{fin}, E_{fin}) en un graphe $(V_{grossier}, E_{grossier})$, un couplage du graphe fin est calculé. Les sommets couplés sont alors fusionnés en un seul dans le graphe grossier et l'arête les reliant disparaît. Le graphe grossier est donc constitué des sommets « compressés » correspondant à deux sommets du graphe plus fin et de poids égal à la somme



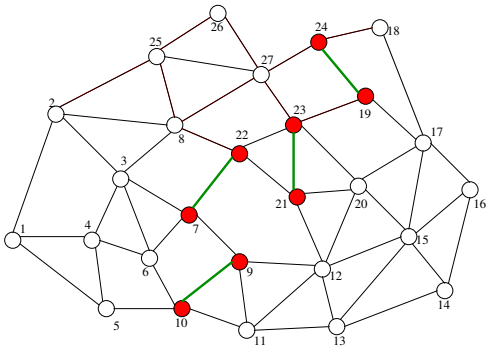
(a) Un couplage (pas forcément maximum) des sommets est effectué.



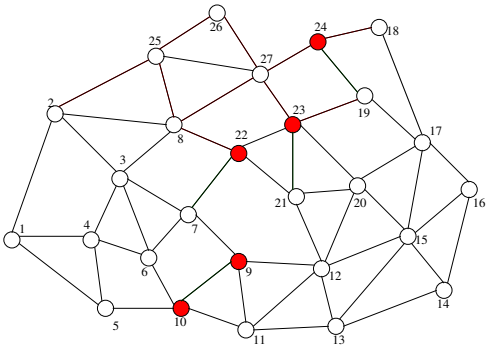
(b) Les sommets couplés sont compressés en un seul. Le poids d'un sommet compressé est la somme des poids des sommets couplés correspondant. Ainsi, les sommets jaunes ont ici un poids de 2 et les sommets blancs de 1. Les arêtes multiples sont fusionnées et prennent le poids correspondant (non montré sur la figure).



(c) Une fois le graphe compressé plusieurs fois jusqu'à atteindre un niveau de compression satisfaisant, un séparateur est calculé sur le graphe le plus grossier en utilisant l'algorithme *greedy graph growing*.



(d) Le séparateur du graphe le plus grossier est « projeté » sur le graphe « fin » à partir duquel il a été construit : un sommet appartient au séparateur si son homologue dans le graphe le plus grossier était dans le séparateur.



(e) Le séparateur du graphe fin est raffiné en utilisant l'algorithme de *Fiduccia-Mattheyses*. S'il y avait eu plusieurs niveaux de compression, les étapes (d) et (e) auraient été répétées sur les graphes « fins » successifs jusqu'à obtenir un séparateur du graphe original.

FIGURE 3.2 – Méthode de multiniveau

Algorithme 10 : `multiniveau`

Entrées : graphe $G = (V, E)$
Sortie : partition (P_0, S, P_1) de V telle que S est un (petit) séparateur, $|P_0| \approx |P_1|$

```
1  G_1 ← G;
2  i ← 1;
3  tant que G_i peut être compressé faire
4  |  G_{i+1} ← compresserGraphe (G_i);
5  |  i++;
6  n ← i;
7  (P_{0,n}, S_n, P_{1,n}) ← trouverSéparateur (G_n);
8  pour i de n − 1 a 1 faire
9  |  (P_{0,i}, S_i P_{1,i}) ← décompresserSéparateur (G_{i+1}, P_{0,i+1}, S_{i+1}, P_{1,i+1});
10 |  (P_{0,i}, S_i P_{1,i}) ← raffinerSéparateur (G_i, P_{0,i}, S_i, P_{1,i});
11 return (P_{0,1}, S_1, P_{1,1});
```

des poids des sommets fins correspondants, ainsi que des sommets qui n'ont pas pu être couplés, dont le poids ne change pas. Si v_1 et v_2 sont des sommets du graphe grossier, correspondant aux ensembles de sommets V_1 et V_2 dans le graphe fin ($|V_1|, |V_2| \in \{1, 2\}$), alors il existe une arête entre v_1 et v_2 dans le graphe grossier si et seulement si $E_{fin} \cap (V_1 \times V_2)$ n'est pas vide ; le cas échéant, le poids de l'arête grossière est la somme des poids des arêtes de cet ensemble.

Comme la compression du graphe se réalise sur plusieurs niveaux, il n'est pas nécessaire de calculer un couplage maximum. Ainsi, si par exemple l'algorithme d'Edmonds était utilisé[16], chaque niveau de compression coûterait $\mathcal{O}(\sqrt{|V|}|E|)$. A cela, Scotch préfère utiliser un algorithme simple en $\mathcal{O}(|V| + |E|)$, basé sur un parcours des sommets tentant de coupler « gloutonnement » chacun d'eux à un de ses voisins non-couplé, et ce quitte à compresser quelques niveaux de plus. Cependant, afin de s'assurer que la compression marche suffisamment bien et que la faible complexité de chaque niveau ne soit pas contrebalancée par un trop grand nombre de niveaux, le critère d'arrêt de la compression de Scotch comprend, en sus d'un nombre de sommets seuil, un facteur de compression seuil. On appelle *facteur de compression* d'un graphe grossier le rapport de son nombre de sommets par le nombre de sommets du graphe fin correspondant. Ce facteur varie de 0.5, pour un couplage parfait, à 1 pour un couplage où aucun sommet n'a été couplé. Scotch s'arrête par défaut juste avant que le nombre de sommets tombe en-deçà de 100 ou que le facteur de compression dépasse 0.8. De cette façon, le nombre de sommets du graphe au $i^{ème}$ niveau de compression ne peut pas excéder $0.8^i|V_{initial}|$; si l'on suppose que le nombre d'arêtes décroît de la même manière, la complexité de l'ensemble de la phase de compressions successives est bornée par $\mathcal{O}((|V_{initial}| + |E_{initial}|) \sum_{0 \leq i \leq nblevel} 0.8^i) = \mathcal{O}(5(|V_{initial}| + |E_{intial}|)) = \mathcal{O}(|V_{initial}| + |E_{intial}|)$.

Détaillons maintenant comment le couplage est effectué par Scotch. La réussite de l'étape de multiniveau repose sur le fait que le graphe soit compressé de manière homogène. Le graphe compressé doit ainsi simplifier le graphe initial tout en conservant ses caractéristiques, afin que l'algorithme de recherche d'un séparateur appliqué sur le graphe le plus grossier puisse donner une bonne estimation d'un séparateur sur le graphe fin initial. De bons indicateurs d'une com-

16. Voir par exemple `https://en.wikipedia.org/wiki/Blossom_algorithm`.

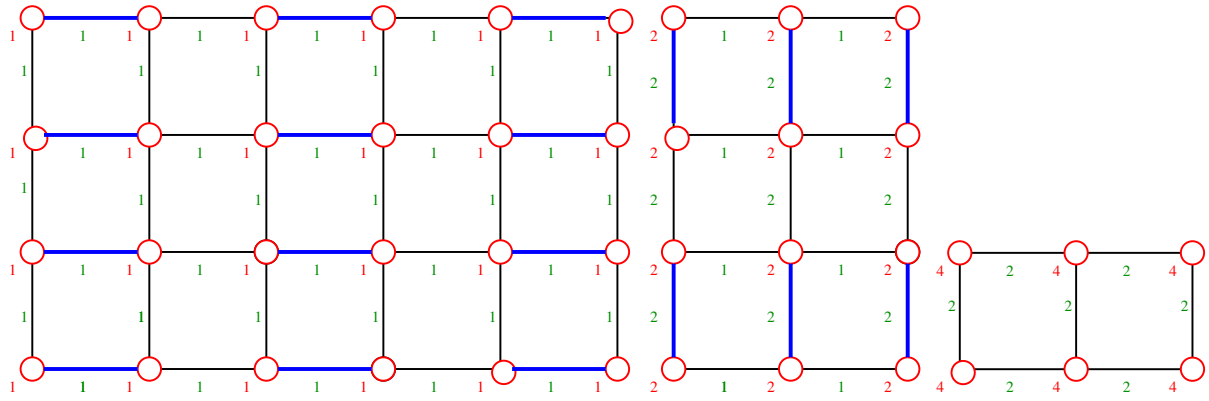


FIGURE 3.3 – Illustration de l'heuristique du *heavy-edge matching*. A gauche, tous les sommets et arêtes du graphe initial ont un poids de 1. Les arêtes en bleu sont les arêtes couplées ; elles sont toutes horizontales. La compression selon ces arêtes permet d'obtenir le graphe du milieu. Dans ce graphe, ce sont maintenant les arêtes verticales qui ont les poids les plus forts, encourageant ainsi une compression le long d'arêtes verticales pour obtenir le graphe de droite.

pression homogène sont les poids des arêtes et des sommets ; puisqu'on considère ici des graphes d'adjacence, ces poids sont égaux dans le graphe initial. SCOTCH effectue un premier parcours des sommets dans lequel il tente de coupler en priorité les sommets ayant un poids plus faible que la moyenne (très précisément, ceux dont le poids est inférieur à $\frac{3}{5}$ du poids moyen d'un sommet) ; puis dans un second parcours, il s'occupe des autres sommets. L'idée est ainsi de « remonter » les poids les plus faibles afin d'éviter que la différence entre le sommet le plus lourd et le sommet le plus léger n'augmente, créant ainsi un graphe hétérogène. Il veille également à ne pas créer de sommet grossier trop lourd : pour cela, il calcule une borne inférieure du poids moyen d'un sommet du graphe grossier (calculable à partir du facteur 0.8 de compression minimum) et convient qu'aucun sommet grossier ne devra avoir un poids excédant une fois et demi ce nombre. Lorsqu'il tente de coupler un sommet, SCOTCH examine la liste de ses voisins non-déjà couplés et en retire les sommets trop lourds. Parmi les choix restants, il choisit de coupler le sommet selon l'arête la plus lourde, et ce en utilisant l'heuristique du « heavy-edge matching » [KK95]. Comme les arêtes du couplage disparaissent dans le graphe grossier, cette stratégie permet de supprimer préférentiellement les plus grosses arêtes, espérant ainsi limiter les disparités dans les poids des arêtes. Une autre justification intuitive consiste à dire que si le graphe est localement compressé selon une direction, les arêtes restantes dans la même direction gardent leur poids, alors que les arêtes transversales fusionnent et donc grossissent. Lors de la compression suivante, le graphe sera donc compressé suivant une autre direction, assurant ainsi l'homogénéité de la compression sur plusieurs niveaux. Un exemple illustrant cette explication est donné en figure 3.3.

A présent, il reste à décrire comment décompresser un séparateur grossier dans un graphe plus fin (fonction `décompresserSéparateur`). Pour cela, SCOTCH décide qu'un sommet d'un graphe fin appartient à la même partie — P_0, P_1 ou S — que son homologue grossier. De cette façon, le séparateur fin est bel et bien un séparateur : en effet, tous les chemins partant d'un sommet de P_0 et arrivant à un sommet de P_1 dans le graphe fin correspondent à un chemin passant par un sommet du séparateur dans le graphe grossier. Par construction, le chemin dans le graphe fin passe donc lui aussi par un sommet du séparateur. En revanche, on peut noter que le séparateur ainsi formé du graphe fin pourra être « épais » et donc localement non-optimal ;

ceci sera corrigé par l'appel à la fonction `raffinerSéparateur`.

► Nous venons de décrire l'algorithme de multiniveau tel qu'il se déroule classiquement, réalisant la fonction `trouverSéparateur` de l'algorithme 8 sur un graphe dépourvu de halo. A présent, nous décrivons les modifications que nous lui avons apporté pour remplir le rôle de la fonction `trouverSéparateurEquilibrantLeHalo` de l'algorithme 9.

Afin de répercuter l'information sur les sommets halo jusque dans le graphe le plus grossier, nous avons enrichi la phase de compression. Désormais, chaque sommet possède un double poids, comportant une composante « non-halo » et une composante « halo ». Les poids dans le graphe initial sont établis à $(1, 0)$ pour chaque sommet non-halo et $(0, 1)$ pour chaque sommet halo. Lors de la construction d'un couplage pour compresser le graphe, nous décidons de prendre en compte uniquement la somme de ces deux composantes. En faisant de la sorte, notre modification n'influence pas le couplage choisi ; en particulier, les sommets halos ne sont pas préférentiellement couplés à d'autres sommets halos, ni l'inverse. Ajouter ce type de contraintes pourrait en effet se révéler délétère, car dans de nombreux cas, les sommets du halo sont proches les uns des autres mais pas forcément voisins. C'est ce qui se passe, par exemple, sur une grille 2D « 4 points » découpée en losange le long des diagonales. Si les sommets du halo sont les bords du losange, le halo est totalement déconnecté ; imposer un couplage des sommets halos entre eux signifierait que ceux-ci ne seraient jamais couplés, résultant en une compression non-homogène entre les sommets du halo et les sommets non-halo.

Pour finir, dans le contexte de ces deux types de poids, nous redéfinissons la notion *sommet halo*. Un sommet est maintenant considéré comme « sommet halo » dès lors que son poids halo est non-nul, c'est-à-dire si l'ensemble des sommets auxquels il correspond dans le graphe initial contient au moins un sommet halo. Nous convenons également que si C est un ensemble de sommets, $\overline{C}$ désigne le sous-ensemble de ses sommets halos. $|C|$ désignera la somme des poids non-halo des sommets de C et $|\overline{C}|$ la somme des poids halos [17]. Comme la phase de compression n'est pas modifiée, nous nous attendons à ce que la proportion de sommets halo reste à peu près la même de niveau en niveau.

3.3 Raffinement d'un séparateur : l'algorithme de Fiduccia-Mattheyses

L'algorithme de Fiduccia-Mattheyses (FM) [FM82] est un algorithme implémenté par Scotch pour raffiner un séparateur existant. Il est utilisé par l'algorithme 10 de multiniveau présenté dans la section précédente. Son fonctionnement est décrit par l'algorithme 11. Deux versions existent ; la première permet d'optimiser des séparateurs constitués d'arêtes, la seconde des séparateurs constitués de sommets. Comme ce sont les séparateurs sommet qui nous intéressent ici, nous nous concentrerons uniquement sur la seconde version.

17. Comme nous travaillons sur des graphes d'adjacence, ceux-ci n'ont que des poids halo ou non-halo de 1 avant la première compression (plus précisément, $(1, 0)$ ou $(0, 1)$). Quelque soit le niveau de compression, le poids non-halo (resp. halo) d'un sommet est le nombre de sommets non-halo (resp. halo) auquel il correspond dans le graphe initial. La notation $|C|$ désigne donc à la fois un poids dans le graphe compressé et un cardinal dans le graphe initial.

Algorithme 11 : `fiducciaMattheyses`

Entrées : graphe : $G = (V, E)$; nombre de passes : $nbPasses$; nombre de mouvements consécutifs sans amélioration autorisés au cours d'une passe (hill-climbing) : $nbMvts$; déséquilibre acceptable : Δ_{th} ; partition initiale : (P_0, S, P_1) avec $S \neq \emptyset$

Sortie : partition (P_0, S, P_1) de V telle que S est un (petit) séparateur et $|P_0| \approx |P_1|$

Notation : pour une partie $i \in \{0, 1\}$, $\neg i$ désigne l'autre partie (la partie $1 - i$).

1 $(P_0^\star, S^\star, P_1^\star) \leftarrow (P_0, S, P_1)$;
2 $numPasse \leftarrow 0$;
`/* Boucle sur les passes */`
3 **répéter**
4 $(P_0, S, P_1) \leftarrow (P_0^\star, S^\star, P_1^\star)$; `/* Restauration de la meilleure partition */`
5 $\Delta \leftarrow |P_0| - |P_1|$;
6 $tabou \leftarrow \emptyset$; `/* Sommets déjà déplacés au cours de la passe */`
7 $pref \leftarrow (\ mod)(numPasse, 2)$; `/* Partie préférée de la passe courante */`
8 $numMvt \leftarrow 0$;
9 $amélioré \leftarrow faux$;
`/* Boucle d'une passe */`
10 **tant que** $numMvt < nbMvts$ **faire**
`/* Choix d'un sommet` v `du séparateur et d'une partie` i `*/`
11 $(f, v, i) \leftarrow$ `choisirSéparateur` $(S \setminus tabou, \max(\Delta_{th}, |\Delta|), pref)$;
12 **si** $\neg f$ **alors** `/* Aucun sommet déplaçable */`
13 break;
`/* Déplacement de` v `du séparateur vers la partie` i `*/`
14 $R \leftarrow \{w | (v, w) \in E \text{ et } w \in P_{\neg i}\}$;
15 $S \leftarrow S \setminus \{v\} \cup R$;
16 $P_i \leftarrow P_i \cup \{v\}$;
17 $P_{\neg i} \leftarrow P_{\neg i} \setminus R$;
18 $\Delta \leftarrow |P_0| - |P_1|$;
19 $numMvt$++;
20 $tabou \leftarrow tabou \cup \{v\}$;
`/* Sélection du meilleur séparateur */`
21 **si** (P_0, S, P_1) *est meilleure que* $(P_0^\star, S^\star, P_1^\star)$ **alors**
22 $(P_0^\star, S^\star, P_1^\star) \leftarrow (P_0, S, P_1)$;
23 $numMvt \leftarrow 0$;
24 $amélioré \leftarrow vrai$;
25 $numPasse$++;
26 **jusqu'à** $\neg amélioré$ *ou* $(numPasse = nbPasses)$;
27 **retourner** $(P_0^\star, S^\star, P_1^\star)$;

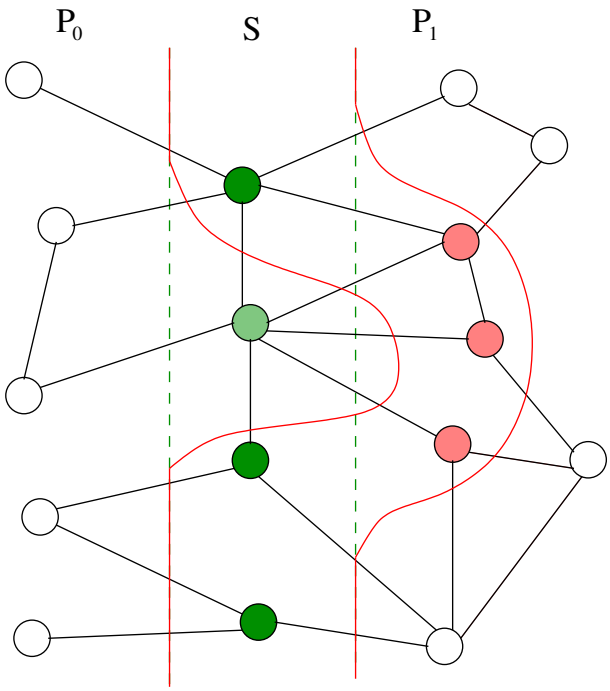


FIGURE 3.4 – Déplacement d'un sommet par l'algorithme FM. Au départ, le séparateur S contient tous les sommets verts. Le sommet en vert clair est choisi pour être mis dans la partie P_0. Les voisins dans P_1 du sommet vert clair (ensemble R dans l'algorithme 11, sommets en rose ici) sont ajoutés à S pour maintenir un séparateur valide.

L'algorithme FM se base sur une recherche locale autour du séparateur initial. Un *mouvement* de la recherche consiste à choisir un sommet du séparateur courant et à le mettre dans l'une des deux parties. Afin de préserver un séparateur correct, les voisins du sommet dans l'autre partie doivent également entrer dans le séparateur (voir figure 3.4). L'algorithme FM effectue plusieurs passes (ensemble de mouvements consécutifs) et continue tant que le nombre maximum de passes n'est pas atteint et que la dernière passe a apporté des améliorations. Chaque passe commence depuis le meilleur séparateur trouvé jusqu'alors (et donc forcément trouvé au cours de la passe précédente). De plus, les passes paires ont une légère préférence pour déplacer les sommets vers la partie 0, tandis qu'à l'inverse les passes impaires favorisent les mouvements vers la partie 1.

La fonction `choisirSommet` à la ligne 11 de l'algorithme 11 choisit le meilleur couple (s, i) à déplacer (où $s \in S$ et $i \in \{0, 1\}$ est une partie). Elle a trois objectifs : obtenir un déséquilibre $\Delta = |P_0| - |P_1|$ raisonnable, minimiser le séparateur S et déplacer un sommet vers la partie préférée de la passe courante [18]. Plus précisément, `choisirSommet` procède en trois temps :

— en premier lieu, elle assure qu'une fois le mouvement réalisé, le nouveau déséquilibre n'excède pas $\max(|\Delta|, \Delta_{th})$, où Δ est le déséquilibre courant (avec son signe), et Δ_{th} un déséquilibre absolu fixé ($\Delta_{th} \geq 0$). Cela signifie que si le déséquilibre courant est en dehors

18. FM a en réalité certaines exceptions pour les sommets isolés, *i.e.* les sommets du séparateur qui ne sont adjacents qu'à au plus une seule des deux parties. Les traitements particuliers qui sont appliqués à ces sommets, qui impliquent quelques difficultés supplémentaires et présentent peu d'intérêt, ne seront pas décrits précisément ici.

de l'intervalle $[-\Delta_{th}, \Delta_{th}]$, le déséquilibre ne sera pas dégradé, et s'il est à l'intérieur, il sera conservé. On peut ainsi distinguer deux phases successives lors de l'exécution de l'algorithme : dans un premier temps, les mouvements réduisent le déséquilibre jusqu'à atteindre l'intervalle $[-\Delta_{th}, \Delta_{th}]$, puis dans un deuxième temps les mouvements se font exclusivement avec un déséquilibre dans l'intervalle $[-\Delta_{th}, \Delta_{th}]$ [19]. Si aucun mouvement satisfaisant ces contraintes n'est possible, la fonction retourne $f = faux$, provoquant l'arrêt de la passe courante (lignes 12-13) ;
— parmi les choix restants, la fonction choisit le sommet menant à la plus petite taille de séparateur (et cela même si ce choix amène un séparateur plus grand que le précédent) ;
— si plusieurs possibilités demeurent, `choisirSommet` choisit un mouvement vers la partie préférée de la passe.

Après chaque mouvement, l'algorithme 11 vérifie si le séparateur courant est le meilleur séparateur trouvé jusqu'à présent. Le choix est effectué de la manière suivante (ligne 21) : si une partition dans laquelle $|\Delta| \leq \Delta_{th}$ n'a jamais été trouvée, la partition dont le $|\Delta|$ est le plus petit est gardée. Sinon, seules les partitions vérifiant $|\Delta| \leq \Delta_{th}$ sont gardées et, parmi elles, celle ayant le plus petit séparateur est sélectionnée. En cas d'égalité, celle avec le plus petit $|\Delta|$ est privilégiée.

Afin de garantir que la recherche locale ne fasse pas les mêmes choix à de multiples reprises, une recherche tabou [20] est implémentée. Un ensemble de sommets *tabou* est maintenu et vidé à chaque nouvelle passe (ligne 7). Lorsqu'un sommet est choisi, il est placé dans l'ensemble *tabou* (ligne 20) et n'est plus autorisé à être déplacé jusqu'à la passe suivante. De cette façon, au cours d'une passe, un sommet peut entrer dans le séparateur, être choisi pour le quitter puis y entrer à nouveau par le mouvement d'un de ses voisins ; mais, dans ce cas, il demeurera par la suite dans le séparateur [21].

Une passe s'arrête soit lorsqu'il n'y a plus aucun mouvement possible (lignes 12-13), soit lorsque les derniers $nbMvts$ mouvements n'ont pas apporté d'amélioration (variable $numMvt$ lignes 9, 10, 19, 24). C'est le principe du *hill-climbing* : donner la possibilité à une passe de faire $nbMvts$ mouvements infructueux consécutifs permet à l'algorithme de sortir de minima locaux et d'atteindre des minima plus petits.

▶ Nous exposons à présent les modifications que nous avons apporté à l'algorithme FM pour prendre en compte l'équilibrage du halo. Dans le cadre de la méthode de multiniveau, nous travaillons sur des sommets ayant un poids double non-halo/halo tels que définis à la fin de la section 3.2.

Le nouvel algorithme doit équilibrer à la fois les sommets de l'intérieur du graphe et les sommets du halo. On peut remarquer que cela peut être localement contradictoire ; pour équilibrer le halo, il est possible de devoir déplacer un sommet du séparateur vers une partie i, alors que pour équilibrer les parties, il serait nécessaire de déplacer un sommet vers l'autre partie $\neg i$. L'algorithme FM ne pouvant agir que localement, on peut aussi constater que l'équilibrage du halo

19. A cause de la gestion des sommets isolés, il est en réalité possible de repasser de la deuxième phase à la première.
20. Voir par exemple `https://fr.wikipedia.org/wiki/Recherche_tabou`.
21. Là encore, cela peut ne pas être vrai pour les sommets isolés.

n'est possible que lorsqu'il existe un sommet du halo « dans les environs » (plus précisément, soit dans le séparateur, soit parmi les voisins du séparateur se trouvant dans la partie excédentaire). Lorsque le halo est déséquilibré, il faut donc mettre à profit toutes les occasions qui se présentent pour le rééquilibrer. C'est pourquoi nous avons fait le choix d'équilibrer en priorité les sommets du halo. Dans la plupart des cas, si l'algorithme de partitionnement modifié fait correctement son travail sur le graphe le plus grossier, il est probable que le séparateur initial contienne des sommets du halo en son sein ou parmi ses voisins. Notre stratégie permet alors de maintenir cette situation (puisque les sommets du halo sont généralement proches les uns des autres), et d'éviter que le séparateur ne « glisse » le long du halo jusqu'à ne plus y être attenant, du fait de l'optimisation d'autres critères.

Notre nouvelle version est décrite par l'algorithme 12. Les choix susmentionnés se traduisent par les modifications suivantes. La décision du sommet v à déplacer du séparateur à une partie i est revisitée. Si la partition courante n'a pas un équilibrage absolu du halo $|\overline{\Delta}|$ en-deçà du seuil $\overline{\Delta}_{th} \geq 0$ (choisi par l'utilisateur), alors la fonction `choisirHalo` est appelée ligne 14. Cette fonction tente de réparer l'équilibrage du halo. Elle utilise le signe de $\overline{\Delta} = |\overline{P_0}| - |\overline{P_1}|$ pour connaître la partie dans laquelle le sommet doit être déplacé : un déplacement vers la partie 0 augmente $\overline{\Delta}$, un déplacement vers la partie 1 le décroît. Ensuite, elle sélectionne un sommet dont le déplacement vers la partie i minimise le nouveau déséquilibre du halo. S'il n'existe aucun sommet améliorant strictement le déséquilibre, l'appel à `choisirHalo` échoue. Dans ce cas, ou si la partition avait déjà un déséquilibre halo raisonnable (*i.e.* si on avait $|\overline{\Delta}| \leq \overline{\Delta}_{th}$), la fonction `choisirSéparateur` est appelée à la place (ligne 16) et fonctionne comme décrit dans le paragraphe sur la version originale [22].

En pratique, dans un cas typique, le séparateur initial est déjà attenant au halo, et les sommets du halo sont proches les uns des autres. Dans ce cas, le fait de donner la priorité au halo résulte en un déplacement du séparateur en deux phases successives ; au cours de la première, les mouvements permettent d'atteindre un équilibrage raisonnable du halo ; et au cours de la seconde, le séparateur s'adapte pour assurer l'équilibrage des parties en maintenant l'équilibre déjà construit du halo.

Enfin, nous avons adapté la stratégie pour choisir la *meilleure* partition en ligne 27 de l'algorithme 12 afin de satisfaire l'équilibrage des parties et du halo. Pour un ensemble de partitions à comparer, nous procédons comme suit (notez que dans l'algorithme, les comparaisons sont effectuées deux par deux) :

- si aucune partition vérifiant $|\Delta| \leq \Delta_{th}$ n'a été trouvée, la partition dont le $|\Delta|$ est le plus petit est sélectionnée ;
- si une partition vérifiant $|\Delta| \leq \Delta_{th}$ a été trouvée, mais aucune vérifiant à la fois $|\Delta| \leq \Delta_{th}$ et $|\overline{\Delta}| \leq \overline{\Delta}_{th}$, alors seules les partitions vérifiant $|\Delta| \leq \Delta_{th}$ sont pré-sélectionnées ; parmi elles, celle avec le meilleur équilibrage du halo est gardée (*i.e.* celle dont le $|\overline{\Delta}|$ est le plus petit) ;
- si une partition vérifiant à la fois $|\Delta| \leq \Delta_{th}$ et $|\overline{\Delta}| \leq \overline{\Delta}_{th}$ a été trouvée, alors seules les partitions vérifiant ces deux contraintes sont gardées ; parmi elles, celle ayant le plus petit séparateur est choisie. En cas d'égalité, le choix est fait en faveur de la partition ayant le plus petit $|\overline{\Delta}|$, puis le plus petit $|\Delta|$.

22. La version modifiée de *FM* comporte elle aussi des cas particuliers liés aux sommets isolés qui ne seront pas expliqués dans le détail.

Algorithme 12 : `fiducciaMattheysesModifié`

Entrées : graphe avec halo $G = (V, E, \overline{V})$; nombre de passes : $nbPasses$; nombre de mouvements consécutifs sans amélioration autorisés au cours d'une passe (hill-climbing) : $nbMvts$; déséquilibre acceptable : Δ_{th} et $\overline{\Delta}_{th}$; partition initiale : (P_0, S, P_1) avec $S \neq \emptyset$

Sortie : partition (P_0, S, P_1) de V telle que S est un (petit) séparateur et $|P_0| \approx |P_1|$

Notation : pour une partie $i \in \{0, 1\}$, $\neg i$ désigne l'autre partie (la partie $1 - i$).

1 $(P_0^\star, S^\star, P_1^\star) \leftarrow (P_0, S, P_1)$;
2 $numPasse \leftarrow 0$;
`/* Boucle sur les passes */`
3 **répéter**
4 $(P_0, S, P_1) \leftarrow (P_0^\star, S^\star, P_1^\star)$; `/* Restauration de la meilleure partition */`
5 $\Delta \leftarrow |P_0| - |P_1|$;
6 $\overline{\Delta} \leftarrow |\overline{P_0}| - |\overline{P_1}|$;
7 $tabou \leftarrow \emptyset$; `/* Sommets déjà déplacés au cours de la passe */`
8 $pref \leftarrow$ `mod` $(numPasse, 2)$; `/* Partie préférée de la passe courante */`
9 $numMvt \leftarrow 0$;
10 $amélioré \leftarrow faux$;
`/* Boucle d'une passe */`
11 **tant que** $numMvt < nbMvts$ **faire**
`/* Choix d'un sommet` v `du séparateur et d'une partie` i `*/`
12 $f \leftarrow faux$;
13 **si** $|\overline{\Delta}| > \overline{\Delta}_{th}$ **alors**
14 $(f, v, i) \leftarrow$ `choisirHalo` $(S \setminus tabou, \overline{\Delta})$;
15 **si** $\neg f$ **alors**
16 $(f, v, i) \leftarrow$ `choisirSéparateur` $(S \setminus tabou, \max(\Delta_{th}, |\Delta|), pref)$;
17 **si** $\neg f$ **alors** `/* Aucun sommet déplaçable */`
18 break;
`/* Déplacement de` v `du séparateur vers la partie` i `*/`
19 $R \leftarrow \{w | (v, w) \in E \text{ et } w \in P_{\neg i}\}$;
20 $S \leftarrow S \setminus \{v\} \cup R$;
21 $P_i \leftarrow P_i \cup \{v\}$;
22 $P_{\neg i} \leftarrow P_{\neg i} \setminus R$;
23 $\Delta \leftarrow |P_0| - |P_1|$;
24 $\overline{\Delta} \leftarrow |\overline{P_0}| - |\overline{P_1}|$;
25 $numMvt$++;
26 $tabou \leftarrow tabou \cup \{v\}$;
`/* Sélection du meilleur séparateur */`
27 **si** (P_0, S, P_1) *est meilleure que* $(P_0^\star, S^\star, P_1^\star)$ **alors**
28 $(P_0^\star, S^\star, P_1^\star) \leftarrow (P_0, S, P_1)$;
29 $numMvt \leftarrow 0$;
30 $amélioré \leftarrow vrai$;
31 $numPasse$++;
32 **jusqu'à** $\neg amélioré$ *ou* $(numPasse = nbPasses)$;
33 **retourner** $(P_0^\star, S^\star, P_1^\star)$;

3.4 Algorithmes de partitionnement

Dans la section précédente, nous avons présenté l'algorithme de raffinement de Fiduccia-Mattheyses, ainsi que les modifications que nous lui avons apporté afin de prendre en compte l'équilibrage des sommets du halo. Cet algorithme est utilisé dans le cadre de la méthode multiniveau à chaque étape de décompression du séparateur. Il reste maintenant à voir comment adapter l'algorithme qui construit le séparateur initial sur le graphe le plus grossier.

La présente section s'intéresse donc à cet algorithme, appelé *greedy graph growing* (*GG*) [KK98a]. La première sous-section détaillera l'algorithme *GG* original, suivi d'une adaptation relativement directe pour les graphes avec halo. Des résultats préliminaires insatisfaisants nous ont poussé à considérer d'autres pistes, que nous présenterons dans les deux sous-sections suivantes. La première, appelée *double greedy graph growing* (*DG*), s'inspire de l'idée de l'algorithme à « bulles » [DPSW00, MMS09] consistant à utiliser une graine par partie, mais est basée sur une approche différente. Dans la seconde, nommée *halo-first greedy graph growing* (*HF*), nous proposons de trouver d'abord un séparateur du graphe du halo, puis d'en déduire un séparateur pour le graphe dans son ensemble.

Afin de comparer les deux variantes *DG* et *HF* à l'algorithme initial non-modifié *GG*, nous avons décidé de présenter une partie de nos résultats tout au long de cette section. Des résultats plus approfondis avec une étude de scalabilité et une intégration dans le solveur MaPHyS seront présentés dans la section 3.5.1.

Avant de préciser les paramètres de nos tests, nous devons faire une remarque sur la réalisation d'un bon équilibrage des parties. Le code de bipartitionnement récursif que nous utiliserons est basé sur celui de la dissection emboîtée [Geo73, LRT79, HR98] du solveur Scotch. Celui-ci est employé habituellement dans un but de renumérotation de la matrice pour l'utilisation d'un solveur direct. Dans ce contexte, le principal objectif de Scotch est de minimiser la taille du séparateur, tout en maintenant le déséquilibre local des intérieurs en-deçà d'un pourcentage fixé, noté *bal*, dont la valeur par défaut est de 10%. La récursion est poursuivie jusqu'à ce qu'un nombre fixé de sommets soit atteint. C'est en fait ce critère d'arrêt qui garantit un bon équilibrage des feuilles et non réellement la valeur de *bal*. En effet, les déséquilibres locaux s'accumulent de niveau en niveau, de sorte qu'à un niveau i, les déséquilibres des parties à cette profondeur de l'arbre peuvent atteindre de l'ordre de $bal \times i$ pourcents. Malheureusement, nous nous focalisons ici non pas sur une renumérotation de matrice, mais sur le partitionnement d'un graphe en un nombre déterminé de parties. Cela nous impose de choisir un critère d'arrêt basé sur le nombre de niveaux de l'arbre [23]. Ainsi, nous pouvons être confrontés à ces importants déséquilibres. La solution consistant à réduire *bal* pour contrebalancer cet effet fait très rapidement face à sa limite : lorsque *bal* est trop petit, le degré de liberté pour trouver un bon séparateur est réduit. Nous avons donc décidé d'utiliser une contrainte *bal* dépendant du niveau : aux niveaux les plus élevés, les sous-graphes sont gros, ce qui nous permet d'utiliser une contrainte plus forte tout en laissant de la marge pour la minimisation du séparateur ; aux niveaux les plus bas, nous relâchons la contrainte. Plus précisément, si p niveaux sont requis, le niveau i s'efforce d'obtenir un déséquilibre local n'excédant pas $\max(\frac{bal}{2^{p-i+1}}, minbal)$ pourcents, où *minbal* est un

23. Pour être exact, nous pourrions envisager un critère d'arrêt basé sur un nombre de sommets seuil, puis gloutonnement fusionner les feuilles sœurs dans l'arbre qui forment la plus petite partie, cela jusqu'à obtenir le nombre désiré de feuilles. Cependant, rien ne garantit que notre équilibrage du halo, effectué à chaque niveau, soit conservé par un tel algorithme.

seuil assurant que la contrainte ne devienne pas trop forte.

Dans la suite, tous les tests seront effectués avec un nombre déterminé de niveaux de récursion, fixé à $p = 4$ dans toute la section, ce qui résultera en un découpage en 16 domaines. L'ensemble de la méthode multiniveau modifiée sera employée, en particulier l'algorithme FM décrit dans la précédente section. La colonne GG dans les résultats fera référence à la stratégie de Scotch non modifiée, avec $bal = 10\%$. La colonne $GG^\star$ et les autres colonnes (concernant les algorithmes DG et HF exposés dans les sections 3.4.2 et 3.4.3) utiliseront la contrainte dépendant du niveau décrite ci-dessus, avec $bal = \frac{10}{2^{p-i+1}}\%$ et $minbal = 1\%$. $GG^\star$ correspond donc à une version de Scotch avec l'algorithme GG par défaut mais auquel nous avons modifié la contrainte bal pour dépendre du niveau.

Nos algorithmes seront évalués sur deux critères. Pour chaque domaine, les tailles des intérieurs et des interfaces seront mesurées. Ensuite, la différence du maximum et du minimum sera effectuée pour chacun des deux, ce qui donnera deux métriques : le *déséquilibre de l'interface* d'une part, et le *déséquilibre des intérieurs* d'autre part. Par exemple, dans le graphe en bas à gauche de la figure 3.1 page 48, la taille de tous les intérieurs vaut 4, d'où un déséquilibre des intérieurs nul, et les tailles des halos sont de 4, 5, 6 et 8 respectivement, soit un déséquilibre des interfaces valant 4.

La table 3.1 présente l'ensemble des matrices qui seront utilisées pour nos tests, donnant leur taille n et leur nombre de non-zéros nnz de $A + A^t$. La colonne id sera utilisée pour identifier les matrices dans la suite de cette thèse. Les matrices 1-20 proviennent de la collection de matrices creuses de l'Université de Floride [Dav] et les dix dernières matrices nous ont été fournies par nos partenaires industriels.

3.4.1 Greedy graph growing

Comme mentionné précédemment, l'algorithme implémenté par le partitionneur Scotch pour trouver un bon séparateur sur le graphe le plus grossier $G = (V, E)$ créé par la technique de multiniveau est appelé *greedy graph growing* (ou GG). L'idée est de choisir aléatoirement un sommet graine dans le graphe et de faire grossir une partie depuis cette graine, jusqu'à ce que celle-ci atteigne la moitié du poids du graphe. L'algorithme 13 décrit cette méthode. A la ligne 3, la graine w est choisie. Le singleton $\{w\}$ forme le séparateur S initial entre la partie P_1, vide, et la partie P_0, contenant tous les autres sommets. Ensuite, à chaque étape, un sommet v du séparateur S courant est choisi (ligne 8) et transféré du séparateur S à la partie grossissante P_1. Le choix est orienté par la minimisation du séparateur courant. De plus, tous les voisins de v dans P_0 sont retirés de P_0 et ajoutés à S, de telle sorte que S demeure un séparateur des deux parties. Ce procédé est répété jusqu'à ce que le poids de la partie grossissante P_1 égale ou dépasse le poids de la partie P_0.

Le résultat de cet algorithme est très dépendant du sommet graine w choisi au départ. C'est pourquoi Scotch effectue plusieurs passes, tentant dans chacune d'elles de faire grossir P_1 depuis une graine différente. Finalement, Scotch restaure la meilleure partition (P_0, S, P_1) trouvée au cours de toutes les passes en se basant sur le poids des séparateurs trouvés.

▶ Dans une première tentative d'adapter cet algorithme avec l'objectif supplémentaire d'équi-

TABLE 3.1 – Matrices de test. Toutes sont issues de problèmes 2D ou 3D. 1-20 viennent de la collection de matrices creuses de l'Université de Floride, et 21-30 de partenaires industriels.

id	**Matrice**	**n**	**nnz**
1	Dubcova3	146689	3489960
2	wave	156317	2118662
3	dj_pretok	182730	1512512
4	turon_m	189924	1557062
5	stomach	213360	3236576
6	BenElechi1	245874	12904622
7	torso3	259156	4372658
8	mario002	389874	1867114
9	helm2d03	392257	2349678
10	kim2	456976	10905268
11	mc2depi	525825	3148800
12	tmt_unsym	917825	3666976
13	t2em	921632	3673536
14	ldoor	952203	45570272
15	bone010	986703	70679622
16	ecology1	1000000	39996000
17	dielFilterV3real	1102824	88203196
18	thermal2	1228045	7352268
19	StocF-1465	1465137	19540252
20	Hook_1498	1498023	59419422
21	**NICE-25**	140662	5547944
22	**MHD**	485597	23747544
23	**Inline**	503712	36312630
24	**ultrasound**	531441	32544720
25	**Audikw_1**	943695	76708152
26	**Haltere**	1288825	18375900
27	**NICE-5**	2233031	175971592
28	**Almond**	6994683	102965400
29	**NICE-7**	8159758	661012794
30	**10millions**	10423737	157298268

Algorithme 13 : `greedyGraphGrowing`

Entrées : graphe $G = (V, E)$; nombre de passes $nbPasses$
Sortie : partition (P_0, S, P_1) de V telle que S est un (petit) séparateur et $|P_0| \approx |P_1|$

```
1  (P0*, S*, P1*) ← (∅, V, ∅);
2  pour p = 1 a nbPasses faire
3      w ← graineAléatoire (V) ;
4      P0 ← V \ {w};
5      P1 ← ∅;
6      S ← {w};
7      tant que |P0| et |P1| ne sont pas équilibrés faire
8          v ← choisirSommet (S) ;
9          N ← {j|(v, j) ∈ E et j ∈ P0} ;                    /* voisins de v dans P0 */
10         S ← S \ {v} ∪ N;
11         P0 ← P0 \ N;
12         P1 ← P1 ∪ {v};
13     si (P0, S, P1) est meilleure que (P0*, S*, P1*) alors
14         (P0*, S*, P1*) ← (P0, S, P1);
15 retourner (P0*, S*, P1*);
```

librer les sommets du halo, nous avons effectué les modifications suivantes.

Premièrement, le choix d'un sommet à déplacer de S à P_1 se fait désormais en tenant compte de l'équilibrage du halo. Ainsi, si P_1 n'a pas autant de sommets que P_0, alors la fonction `choisirSommet` prend préférentiellement un sommet du halo. Afin d'éviter que les sommets halos soient systématiquement choisis au début (car $\overline{P_0} = \overline{V}$ et $\overline{P_1} = \emptyset$ à l'initialisation), nous ramenons la comparaison à la taille courante des parties. Plus précisément, si

$$\frac{|\overline{P_1}|}{|P_1|} < \frac{|\overline{P_0}|}{|P_0|}$$

et qu'il existe un sommet du halo dans S, alors ce sommet est choisi obligatoirement. Dans le cas contraire, un sommet de S (halo ou non-halo) est choisi en se basant uniquement sur la minimisation du séparateur courant.

Cependant, le nombre attendu de sommets du halo est faible par rapport au nombre de sommets du graphe. En particulier, pour les maillages à n nœuds correspondant à des cas en d dimensions, la taille attendue des séparateurs est de $\mathcal{O}(n^{(d-1)/d})$ [MV91, MTV91] et la distance d'un sommet quelconque à un sommet du séparateur de $\mathcal{O}(n^{1/d})$. Ainsi, un sommet graine w choisi aléatoirement sera en moyenne éloigné de $\mathcal{O}(n^{1/d})$ du plus proche sommet halo. De plus, tant qu'aucun sommet halo n'aura été atteint lors du parcours, le choix se fera exclusivement par rapport à la minimisation du séparateur, ce qui aura pour effet, dans un graphe de type « grille », de faire grossir un boule autour de la graine initiale w. Or, dans l'exemple d'une grille, la taille d'une boule de rayon r possède $\mathcal{O}(r^d)$ sommets. L'algorithme devrait ainsi parcourir $\mathcal{O}((n^{1/d})^d) = \mathcal{O}(n)$ sommets en moyenne avant de pouvoir faire un choix basé sur l'équilibrage du halo. Dans le pire cas, le critère d'arrêt pourrait même être atteint avant d'avoir rencontré un sommet du halo, résultant en une partie P_0 comprenant tous les sommets du halo.

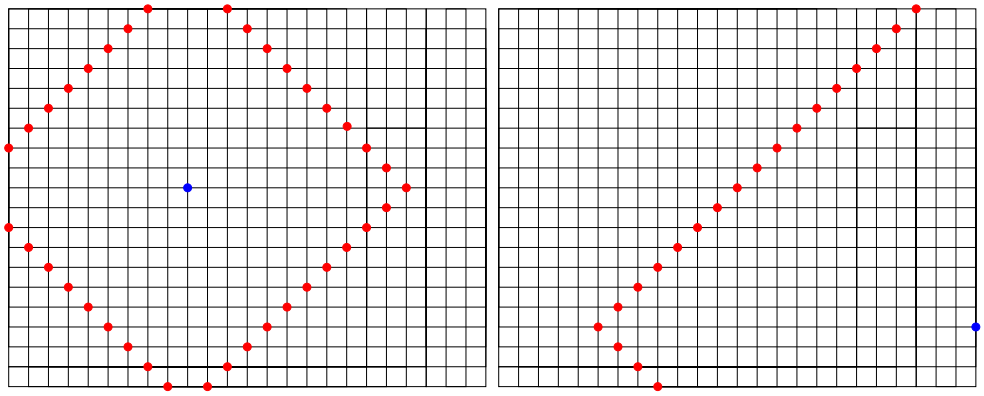


FIGURE 3.5 – Un exemple d'exécution du *greedy graph growing* avec deux graines différentes. *A gauche :* la graine est choisie loin des « bords ». *A droite :* la graine est choisie sur un « bord ».

Le fait de faire plusieurs passes peut compenser en partie ce problème, mais il est plus judicieux de choisir les graines aléatoires directement au sein du halo. Telle est ainsi notre seconde modification. De part la manière dont ils sont construits (ce sont des séparateurs), les sommets du halo ont une grande probabilité de se trouver proches les uns des autres. Ainsi, en faisant grossir P_1 à partir d'un sommet du halo, d'autres sommets du halo ont de grandes chances d'être rencontrés au fur et à mesure du parcours, laissant du choix à l'algorithme pour satisfaire les deux critères.

Une autre justification est la suivante. Comme illustré par la figure 3.5, dans le cas d'un maillage, les meilleures graines se trouvent le plus souvent sur le « bord » du graphe [24]. Intuitivement, lorsqu'un séparateur grossissant atteint un bord, les sommets correspondants disparaissent du séparateur, car ils ne sont adjacents qu'à une des deux parties séparées. Ainsi, les graines proches des « bords » les atteignent rapidement, d'où un séparateur plus petit. Or, les sommets du halo étant des sommets d'anciens séparateurs, ils se situent précisément sur le « bord » du graphe. Il paraît donc judicieux de choisir la graine au sein du halo.

Enfin, nous avons également modifié le choix de la « meilleure » passe. Celui-ci ne se porte plus sur la passe ayant obtenu le plus petit séparateur. Désormais, parmi les séparateurs n'excédant pas 20% de la taille du plus petit séparateur trouvé, le séparateur équilibrant le mieux le halo est sélectionné.

Nous avons implémenté et testé ce nouvel algorithme dans le partitionneur SCOTCH. Les résultats sont cependant mitigés. La figure 3.6 illustre un cas problématique typique. Dans le sous-graphe présenté, les sommets du halo sont en vert (bord gauche) et rouge (bord droit). La graine de départ se situe sur le premier tiers du bord rouge en partant du haut. La partie P_1, en violet, a grossi à partir de là. Le séparateur est représenté en bleu foncé. Cependant, lorsque que P_1 a atteint le bord vert, elle possédait déjà suffisamment de sommets halo par rapport à la

24. La notion de « bord » est floue : dans le cas d'une grille, par exemple, ce terme fait intuitivement référence à tous les sommets n'ayant pas un degré égal au degré maximum du graphe. Une tentative de définition plus générale pourrait y inclure tous les sommets dits pseudo-périphériques : un sommet est pseudo-périphérique s'il est à la plus grande distance de chacun de ses sommets les plus éloignés. Mais avec cette définition, seuls les quatre coins d'une grille 2D feraient partie des « bords ».

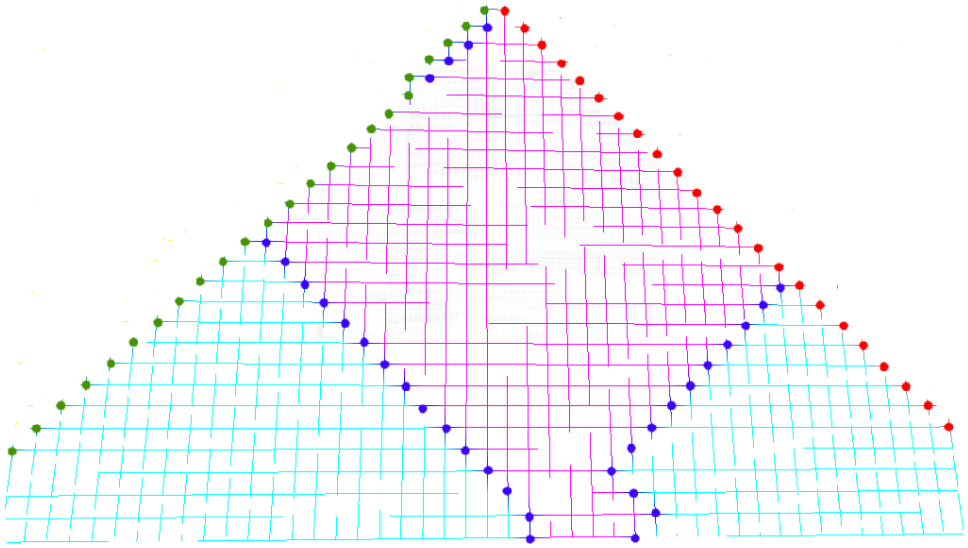


FIGURE 3.6 – Illustration d'un cas pathologique avec le *GG* modifié. La partie bleue est finalement scindée en trois parties déconnectées les unes des autres.

partie P_0 bleue clair. Elle a donc continué à s'étendre vers le bas sans prendre les 5 sommets verts en haut au centre (légèrement à gauche). Il en résulte un morceau de séparateur jouxtant le halo.

Cela a pour conséquence que la partie P_0 comporte trois composantes connexes : les deux parties bleu clair sur la gauche et la droite, ainsi que les 5 sommets verts déjà mentionnés en haut au centre. Il s'agit d'un problème inhérent au *GG* : seule la partie grossissante (P_1) est garantie d'être connexe si le graphe de départ l'est. Cependant, dans la version non-modifiée de *GG*, le cas d'une partie P_0 non-connexe se produit plus rarement, car la minimisation du séparateur, qui est le seul critère (aussi bien lors de la phase de grossissement que dans celle de sélection de la meilleure passe), va généralement de pair avec la connexité de la partie P_0 [25]. A présent que le halo est devenu une priorité, la situation de non-connexité se produit plus fréquemment.

3.4.2 Double greedy graph growing

L'algorithme précédent parvient à un bon équilibrage du halo en général mais, dans la plupart des cas, au prix d'une partie P_0 déconnectée. Nous avons donc décidé d'utiliser deux graines initiales, une pour chaque partie, et de faire grossir les deux parties simultanément. Cependant, cette nouvelle stratégie peut amener des blocages. En effet, tandis qu'elle grossit, une partie peut empêcher la progression de l'autre : cela se produit lorsque $V \setminus (P_0 \cup P_1)$ n'est pas vide mais n'a aucun sommet accessible depuis l'une des parties P_i. Afin d'éviter ce problème, il est nécessaire de retarder, dans la mesure du réalisable, le moment où les deux parties se rejoignent.

25. Il existe de nombreux contre-exemples à cette « règle générale ». Prenons un graphe en trois parties : celles de gauche et de droite contiennent chacune un quart des sommets et sont reliées par un unique sommet à la partie centrale, contenant l'autre moitié des sommets. Le plus petit séparateur valide est alors constitué des deux sommets assurant la liaison des deux parties gauche et droite à la partie centrale. La partie P_0, constituée des parties gauche et droite, n'est pas connexe.

L'algorithme 14 décrit cette nouvelle méthode. La ligne 3 sélectionne les deux graines w_i et $w_{\neg i}$ parmi le halo. Ces graines sont choisies aussi éloignées que possible de telle façon que les parties se rencontrent le plus tard possible. Les deux parties, initialement vides, sont agrandies en partant de leur sommet graine respectif. A chaque étape, la plus petite partie i est choisie (lignes 13-16), ainsi qu'un sommet v de sa frontière S_i (ligne 19). La priorité pour le choix est donnée à l'équilibrage du halo ; si la partie i possède moins de sommets halo que la partie $\neg i$, un sommet du halo est choisi préférentiellement, et dans le cas contraire, un sommet non-halo est privilégié. Si plusieurs choix de sommets demeurent, alors le sommet qui est le plus proche de w_i et le plus éloigné de $w_{\neg i}$ (*i.e.* le sommet v ayant la plus petite valeur de $dist(v, w_i) - dist(v, w_{\neg i})$) est sélectionné. Cela permet, encore une fois, de retarder le moment où les parties se rencontrent. Tout comme l'algorithme de *greedy graph growing* à une seule graine présenté dans la section précédente, v est ensuite ajouté à P_i (ligne 21), retiré de S_i, et S_i est mis à jour afin de rester une frontière de P_i (ligne 20). Ce procédé est répété jusqu'à ce que tous les sommets soient dans l'une ou l'autre des parties, ce qui se traduit par $V \setminus (P_0 \cup P_1) = \emptyset$ (terminaison de la boucle « tant que » ligne 12), ou que la progression d'une des parties soient bloquée, c'est-à-dire que $S_i = \emptyset$ (ligne 17).

Dans ce dernier cas, une solution consisterait à placer tous les sommets restants dans $V \setminus (P_0 \cup P_1)$ de la partie à laquelle ils sont adjacents. Si moins de 10% de sommets restent, c'est effectivement la solution que nous privilégions aux lignes 23-25. Sinon, P_0 et P_1 sont vidés (ligne 23) et nous réessayons de faire grossir les parties depuis w_0 et w_1 en utilisant des informations additionnelles afin d'éviter de revenir à nouveau dans une situation de blocage. Plus précisément, nous définissons un ensemble de *points de contrôle* pour chaque partie i, contenant uniquement leur graine respective à l'initialisation. Lors d'une situation de blocage, nous ajoutons un nouveau point de contrôle à la partie i qui ne peut plus s'étendre. Ce point de contrôle est défini par le sommet de P_i étant le plus proche des sommets de $V \setminus (P_0 \cup P_1)$. Ensuite, les parties sont vidées et agrandies à nouveau. Lorsqu'un sommet est choisi pour l'ajouter à une partie i (ligne 19), le premier critère est comme auparavant l'équilibrage du halo. Le second critère consiste désormais à choisir le sommet v ayant la plus petite valeur $min_j\{dist(v, pointContrôle_i[j])\} - min_j\{dist(v, pointControle_{\neg i}[j])\}$. En d'autre termes, la partie i sera « attirée » par ses propres points de contrôle et « repoussée » par les points de contrôle de l'autre partie. (On peut remarquer que cette règle est en fait la généralisation de la précédente).

La stratégie que nous venons de décrire est répétée jusqu'à ce qu'une partition (P_0, P_1) de V depuis (w_0, w_1) soit trouvée, ou que le nombre maximum d'essais soit atteint, signifiant l'échec du double *greedy graph growing* (DG). Dans le premier cas, un algorithme de couverture de sommets minimum est appliqué aux arêtes de la frontière entre P_0 et P_1 afin de construire un séparateur S constitué de sommets (ligne 29).

Comme pour les algorithmes précédents, plusieurs passes sont effectuées avec différents couples de sommets graines. Nous sélectionnons finalement la partition la meilleure parmi toutes les passes réalisées ligne 32.

Pour conclure cette section, précisons la manière dont nous choisissons les graines initiales. Nous avons mentionné que nous prenions des sommets « aussi éloignés que possible » et « dans le halo ». Cela peut vouloir dire deux choses. Premièrement, nous pouvons prendre deux sommets

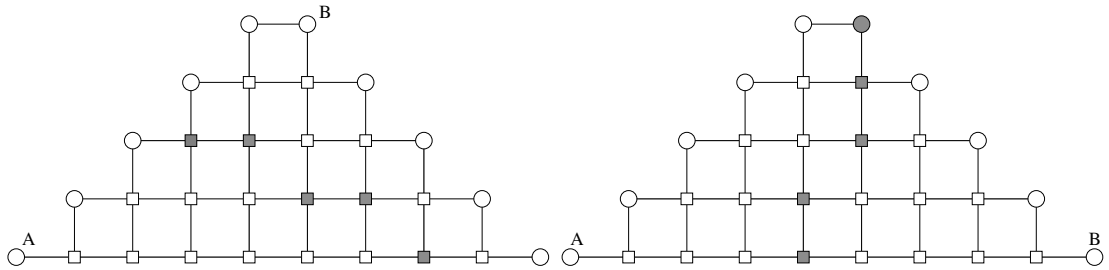


FIGURE 3.7 – Un exemple montrant que prendre les sommets graines A et B « aussi loin que possible le long du halo » (à droite) peut être meilleur que prendre les sommets graines dans le halo « aussi loin que possible dans le graphe entier » (à gauche). Les sommets halos sont identifiés par des cercles.

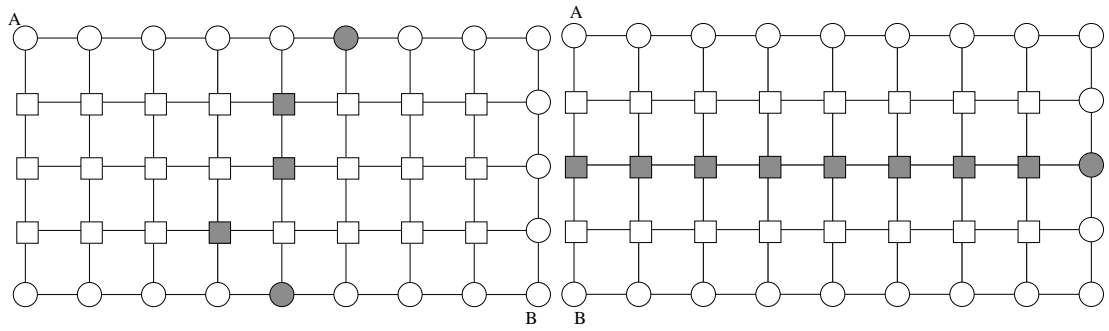


FIGURE 3.8 – Un exemple montrant que prendre les sommets graines A et B dans le halo « aussi loin que possible dans le graphe entier » (à gauche) peut être meilleur que prendre les sommets « aussi éloignés que possible le long du halo » (à droite). Les sommets halo sont identifiés par des cercles.

du halo aussi éloignés que possible dans l'ensemble du graphe. Deuxièmement, nous pouvons extraire le graphe du halo (la section suivante expliquera comment réaliser cela) et choisir les deux sommets les plus éloignés l'un de l'autre dans cette restriction du graphe entier. Les figures 3.7 et 3.8 montrent que selon le graphe, chacune des deux solutions peut être meilleure que l'autre. Dans le graphe de gauche de la figure 3.7, les sommets A et B sont à distance 9 dans le graphe entier, ce qui est le maximum possible. Le sommet le plus à droite est également à distance 9 de A et n'a pas de raison d'être préféré à B. Cependant, le séparateur trouvé avec ces graines ne parvient pas à équilibrer correctement les sommets halo (3 dans une partie, 7 dans l'autre). Sur le graphe de droite, le graphe du halo a été construit et les sommets respectivement le plus à gauche et le plus à droite ont été identifiés comme étant les plus éloignés. Cette fois, le séparateur qui en découle arrive à un équilibrage satisfaisant. Dans le second exemple de la figure 3.8, les deux possibilités de choix de graines parviennent à répartir correctement les sommets halo, mais avec des tailles de séparateur significativement différentes : la stratégie « aussi loin que possible le long du halo » donne un séparateur de taille 9, ce qui est beaucoup plus que la stratégie « loin dans le graphe entier », dont le séparateur compte 5 sommets seulement. Ainsi, les deux stratégies ont leurs avantages, et c'est pourquoi nous essayons chacune d'entre elles.

Nous avons testé le *double greedy graph growing* sur quatre niveaux de récursion (*i.e.* 16 do-

Algorithme 14 : `doubleGreedyGraphGrowing`

Entrées : graphe avec halo $G = (V, E, \overline{V})$; nombre de passes $nbPasses$; nombre d'essais $nbEssais$

Sortie : partition (P_0, S, P_1) de V telle que S est un (petit) séparateur, $|P_0| \approx |P_1|$ et $|\overline{P_0}| \approx |\overline{P_1}|$

```
1  (P_0^⋆, S^⋆, P_1^⋆) ← (∅, V, ∅);
2  pour p = 1 à nbPasses faire
3      (w_0, w_1) ← grainesAléatoires (V) ;
4      pointContrôle_0 ← {w_0};
5      pointContrôle_1 ← {w_1};
6      succès ← faux;
7      pour q = 1 à nbEssais faire
8          P_0, P_1 ← ∅;
9          S_0 ← {w_0};
10         S_1 ← {w_1};
11         distPointsContrôle ← calculerDistances (G, pointContrôle_0, pointContrôle_1);
12         tant que V \ (P_0 ∪ P_1) ≠ ∅ faire
13             si |P_0| < |P_1| alors
14                 i ← 0;
15             sinon
16                 i ← 1;
17             si S_i = ∅ alors
18                 break;
19             v ← choisirSommet (S_i, distPointsContrôle) ;
20             S_i ← S_i \ {v} ∪ {j|(v, j) ∈ E et j ∈ V \ (P_0 ∪ P_1)} ;
21             P_i ← P_i ∪ {v} ;
22         si |V \ (P_0 ∪ P_1)| ≤ 0.1|V| alors
23             P_¬i ← V ∪ P_i;
24             succès ← vrai;
25             break;
26         sinon
27             pointContrôle_i ← pointContrôle_i∪ trouverNouveauPointDeContrôle
               (G, P_i, P_¬i);
28     si succès alors
29         S ← couvertureSommetMinimale (E ∩ (P_0 × P_1));
30         P_0 ← P_0 \ S;
31         P_1 ← P_1 \ S;
32         si (P_0, S, P_1) est meilleur que (P_0^⋆, S^⋆, P_1^⋆) alors
33             (P_0^⋆, S^⋆, P_1^⋆) ← (P_0, S, P_1);
34 retourner (P_0^⋆, S^⋆, P_1^⋆);
```

	Déséquilibre des interfaces			Déséquilibre des intérieurs		
id	*GG*	% *GG**	% *DG*	*GG*	% *GG**	% *DG*
1	297	7,4	**-19,2**	1311	**-69,1**	-23,3
2	1112	1,6	**-40,0**	4678	**-66,5**	-78,9
3	522	-11,3	**-39,7**	1635	**-13,9**	-13,0
4	244	-9,0	**-11,5**	1568	-23,7	**-31,1**
5	475	**-17,9**	-3,8	4605	**-85,6**	-62,3
6	869	-21,3	**-41,3**	4107	**-78,5**	-49,6
7	**905**	49,4	26,4	4942	**-69,1**	-63,7
8	261	0,0	**-46,7**	420	**0,0**	79,8
9	365	-3,8	**-44,9**	8128	**-82,8**	-65,2
10	1002	14,0	**-32,1**	4852	**-73,2**	-44,6
11	509	-3,7	**-44,4**	2831	**-84,1**	27,5
12	569	-25,0	**-59,1**	22416	**-79,5**	-67,5
13	532	-11,8	**-58,5**	12049	**-69,0**	-49,5
14	756	-23,1	**-46,3**	17458	-61,9	**-66,6**
15	6678	6,6	**-29,8**	35466	**-73,9**	-68,4
16	564	-11,9	**-62,9**	15092	**-65,5**	-58,0
17	2130	18,6	**-50,4**	32202	**-72,9**	-67,7
18	335	17,0	**-19,7**	28057	**-70,1**	-64,9
19	2604	-16,4	**-42,0**	25853	**-66,5**	-59,5
20	9990	-14,2	**-25,8**	73635	**-80,7**	-16,5
21	927	0,6	**-40,9**	2377	-58,6	**-59,5**
22	3468	5,0	**-78,5**	**3336**	4,3	18,2
23	1869	7,1	**-3,2**	14424	**-73,3**	-64,4
24	2460	-9,1	**-73,4**	8940	-44,1	**-60,6**
25	6837	-11,6	**-69,7**	26877	-63,0	**-68,9**
26	780	-15,9	**-53,8**	24987	-58,3	**-65,2**
27	6168	-27,5	**-68,4**	67721	-68,7	**-76,2**
28	4344	-15,6	**-41,4**	240729	-76,9	**-77,1**
29	11539	8,6	**-51,3**	244959	-73,9	**-77,2**
30	9936	-6,9	**-52,0**	286992	**-72,6**	-8,5

TABLE 3.2 – Résultats du *double greedy graph growing* en comparaison avec le *greedy graph growing* de SCOTCH. *Résultats obtenus avec un découpage en 16 domaines.*

maines). Les résultats sont présentés dans la table 3.2. Les colonnes *GG* donnent le déséquilibre des intérieurs et de l'interface de la version non modifiée de Scotch avec $bal = 10\%$. Les autres colonnes fournissent uniquement un pourcentage relatif à la colonne *GG* correspondante. Par exemple, pour la matrice `ecology1` (16), le double greedy graph growing (colonne *DG*) parvient à un équilibrage du halo $62,9\%$ meilleur que la version non-modifiée *GG*, ce qui signifie que le déséquilibre halo vaut $(1 - 0.629) \times 564 \simeq 209$. La colonne $GG^\star$ fait référence à une version modifiée de Scotch dans laquelle le paramètre d'équilibrage dépend du niveau (voir l'introduction de cette section). Pour chaque critère, nous avons mis en valeur en gras le meilleur résultat.

On remarque premièrement qu'avec une simple modification du critère de *bal* par niveau ($GG^\star$), on améliore significativement le déséquilibre des intérieurs. Sans surprise, $GG^\star$ n'améliore cependant pas sensiblement l'équilibrage du halo ; en moyenne sur les 30 matrices, le gain est de 4%.

Nous commentons à présent les résultats de *DG*. Nous pouvons constater que l'équilibrage de l'interface effectué par celui-ci est meilleur que celui obtenu avec *GG* et $GG^\star$ sur toutes les matrices excepté deux. Les gains peuvent aller jusqu'à $78,5\%$ sur la matrice `MHD` (22), avec une moyenne de 40% sur l'ensemble des matrices. *DG* parvient également à un meilleur équilibrage des intérieurs que *GG* sur toutes les matrices sauf deux ; le gain moyen pour cette colonne est de 45%. Il est également meilleur que $GG^\star$ sur un tiers des cas de test, ce qui est honorable compte tenu du fait qu'il doit optimiser un critère supplémentaire. De plus, nous pouvons constater que sur toutes les matrices provenant de nos partenaires industriels sauf une, les gains sont très bons sur les deux critères.

3.4.3 Halo-first greedy graph growing

Dans le paragraphe précédent, nous avons présenté un algorithme construisant un séparateur des parties et du halo simultanément. Une autre idée consiste à partitionner d'abord le halo, puis à faire grossir les parties depuis deux ensembles de graines correspondant aux deux parties du halo.

On peut définir dans un premier temps le graphe du halo comme étant la restriction du graphe $G = (V, E)$ aux seuls sommets halos et aux arcs les reliant, c'est-à-dire au graphe $G_h = (\overline{V}, E \cap (\overline{V} \times \overline{V}))$. Malheureusement, G_h est alors très souvent non-connexe et même possiblement totalement déconnecté. Considérons par exemple un graphe de type grille 2D où chaque sommet a pour voisins les sommets au-dessus, en-dessous, à gauche et à droite. Si ce graphe est coupé selon une diagonale, alors les sommets du halo, le long de la diagonale, sont à distance deux les uns des autres.

Une idée serait de reconnecter le halo en considérant le voisinage proche de chacun des sommets. Cela n'est pas suffisant : comme le montre la figure 3.9, il peut être nécessaire de considérer des sommets arbitrairement éloignés du halo pour reconnecter celui-ci. La figure montre également qu'ignorer les positions respectives des sommets halos peut conduire à un très mauvais séparateur du halo.

Nous avons donc utilisé l'algorithme 15 pour reconnecter le halo. D'abord, l'algorithme créé une famille d'ensembles disjoints P contenant un singleton pour chaque sommet du halo. A chaque étape de l'algorithme, on trouve les deux ensembles Q et R dans P qui sont les plus

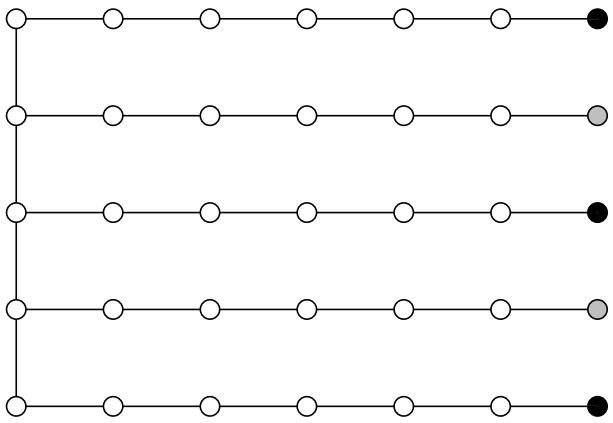


FIGURE 3.9 – Dans cet exemple, le halo (sommets à droite) est totalement déconnecté, et on constate qu'il est nécessaire d'explorer son voisinage éloigné pour le reconnecter. Cependant, ignorer les positions relatives des sommets du halo pour construire un séparateur conduirait en général à un très mauvais séparateur, si par exemple nous prenons les sommets noirs dans une partie et les gris dans l'autre.

proches. Plus précisément, si Q et R sont des ensembles de sommets, on définit la distance de Q à R comme étant $dist(Q,R) = \min_{q\in Q, r\in R}(dist(q,r))$; on sélectionne alors dans P deux ensembles Q et R à distance minimale. Les deux ensembles Q et R sont ensuite retirés de P, fusionnés en y ajoutant tous les sommets du (d'un) plus court chemin de l'un à l'autre et réinsérés dans P. La cardinalité de P décroît donc globalement de 1. Si le graphe G est connexe, alors l'algorithme continue ainsi pendant $|\overline{V}| - 1$ étapes, jusqu'à ce que P ne contienne plus qu'un unique élément, constitué d'un ensemble connexe comprenant tous les sommets du halo. Dans le cas contraire, l'algorithme s'arrête dès qu'il n'existe plus de chemin permettant de relier des parties non-déjà fusionnées. Le graphe reconnecté du halo, qui sera désormais noté G'_h, est constitué de tous les sommets dans les ensembles P et de toutes les arêtes de G les reliant.

Algorithme 15 : `construireHaloReconnecté`

Entrées : graphe avec halo $G = (V, E, \overline{V} = \{h_0, h_1, \dots h_{|\overline{V}|}\})$
Sortie : sous-graphe de G contenant tous les sommets halos (et peu d'autres sommets) ; si G est connexe alors ce sous-graphe l'est également

$P \leftarrow \{\{h_0\}, \{h_1\}, \dots, \{h_{|\overline{V}|}\}\}$;
tant que $|P| \neq 1$ **faire**
  $(Q, R) \leftarrow$ deux ensembles de P tels que $dist(Q,R) = \min_{\substack{S,T\in P\times P \\ S\neq T}} dist(S,T)$;
  si $dist(Q,R) = \infty$ **alors** `/* Si G n'est pas connexe */`
    break ;
  $Z \leftarrow$ ensemble de sommets d'un plus court chemin de Q à R dans G;
  $P \leftarrow P \setminus \{Q, R\}$;
  $P \leftarrow P \cup \{Q \cup R \cup Z\}$;
$W \leftarrow \bigcup_{P_i\in P} P_i$;
retourner $(W, E \cap (W \times W))$;

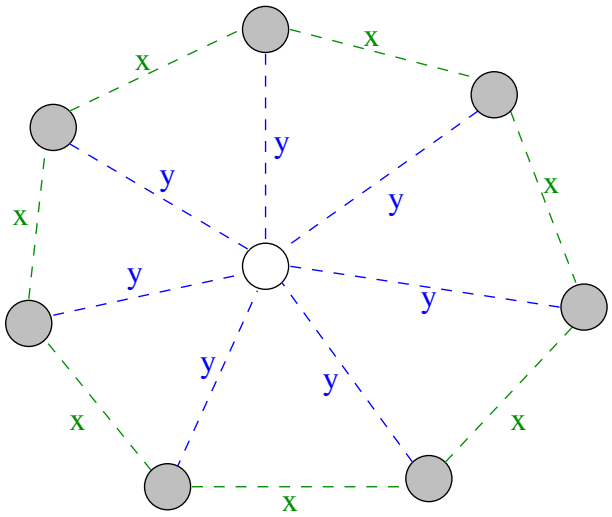


FIGURE 3.10 – Illustration du fait que l'algorithme de reconnexion du halo ne permet pas forcément de sélectionner un ensemble minimum de sommets. Les sommets halos sont représentés en gris. Les lignes pointillées représentent des chemins constitués de sommets non-halos ; x et y représentent les longueurs des chemins. Si $2y > x$, alors l'algorithme 15 sélectionnera tous les chemins verts sauf un, pour une taille totale de $1 + (|\overline{V}| - 1) \times x$ sommets. Pourtant, en sélectionnant l'ensemble des chemins bleus, on aurait pu arriver à un ensemble de $1+|\overline{V}|\times(y-1)$ sommets, ce qui peut être plus petit pour certaines valeurs de x et y.

En pratique, l'algorithme est implémenté avec un seul parcours en largeur global du graphe G, initialisé avec chaque sommet halo dans la file. Le parcours en largeur fait ainsi grossir des « bulles » autour de chaque sommet du halo ; lorsque deux bulles distinctes se touchent, un plus court chemin entre deux parties vient d'être trouvé. Les deux parties correspondantes sont alors fusionnées. Pour représenter la famille P d'ensembles disjoints, on utilise une forêt d'arbres disjoints et les heuristiques de compression de chemin et de l'union par rang [Tar75]. Chaque opération de fusion coûte ainsi un temps amorti de $\mathcal{O}(\alpha(|\overline{V}|))$ où α désigne l'inverse de la fonction d'Ackermann [26]. L'algorithme a donc une complexité globale de $\mathcal{O}((|V|+|E|)\alpha(|\overline{V}|))$.

On peut remarquer que cet algorithme ne permet pas de sélectionner un ensemble minimum de sommets reconnectant le halo : la figure 3.10 présente une famille de contre-exemples. Cependant, notre objectif n'est pas de minimiser le nombre de sommets, mais de reconstruire un graphe du halo qui rend compte au mieux des connexions entre les sommets du halo.

Appelons `construireHaloReconnecté` la fonction construisant le graphe G'_h. L'algorithme 16 donne les principales étapes pour trouver un séparateur. Ligne 4, un premier appel à `greedyGraphGrowing` est effectué pour calculer un découpage du halo (V'_{h0}, S'_h, V'_{h1}). Puis, une version adaptée de la méthode de *greedy graph growing* à deux graines est utilisée ligne 5, initialisée avec l'ensemble de graines V'_{h0} pour la partie 0 et V'_{h1} pour la partie 1 (cette version de DG n'a pas besoin de tenir compte du halo à proprement parler, dans la mesure où le halo lui est passé en tant que deux ensembles de départ). Cela permet d'obtenir une partition (P_0, S, P_1) de l'ensemble du graphe. De la même manière que pour les autres algorithmes, ces étapes peuvent

26. Voir par exemple `https://fr.wikipedia.org/wiki/Fonction_d%27Ackermann`.

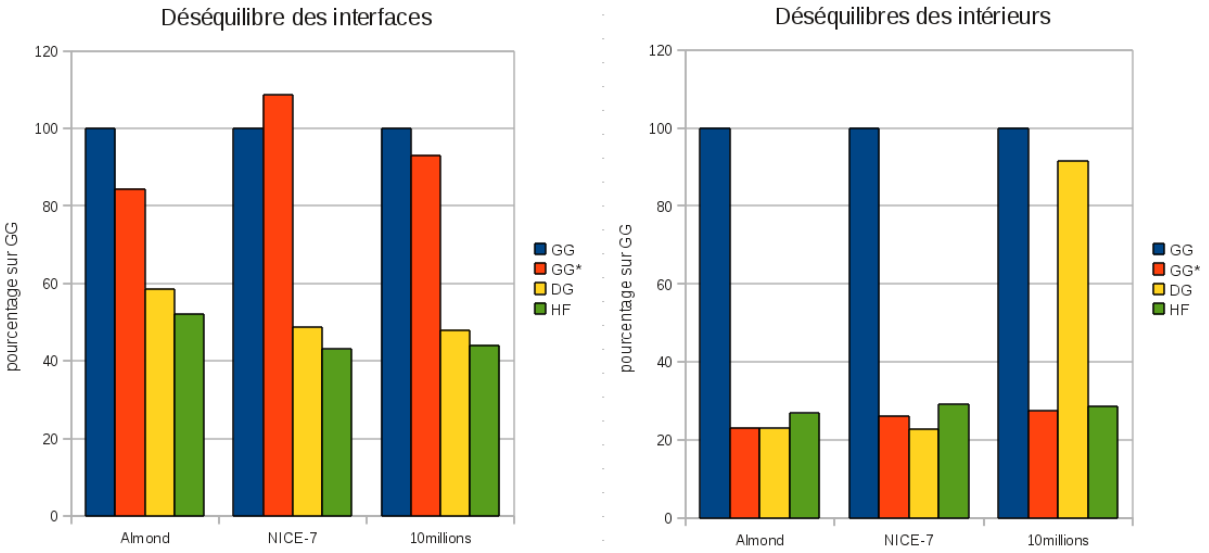


FIGURE 3.11 – Comparaison du *greedy graph growing GG* et $GG^\star$, *double greedy graph growing DG* et *halo-first greedy graph growing HF* sur les matrices `Almond`, `Nice` et `10Millions`. *Résultats obtenus avec un découpage en 16 domaines.*

êtres répétées, faisant ainsi plusieurs passes pour sélectionner la meilleure.

Algorithme 16 : `haloFirstGreedyGraphGrowing`

Entrées : graphe avec halo $G = (V, E, \overline{V})$; nombre de passes $nbPasses$
Sortie : partition (P_0, S, P_1) de V telle que S est un (petit) séparateur, $|P_0| \approx |P_1|$ et $|\overline{P_0}| \approx |\overline{P_1}|$

$(P_0^\star, S^\star, P_1^\star) \leftarrow (\emptyset, V, \emptyset)$;
$(V'_h, E'_h) \leftarrow$ `construireHaloReconnecté` $(V, E, \overline{V})$;
pour $p = 1$ **a** $nbPasses$ **faire**
    $(V'_{h0}, S'_h, V'_{h1}) \leftarrow$ `greedyGraphGrowing` (V'_h, E'_h);
    $(P_0, S, P_1) \leftarrow$ `doubleGreedyGraphGrowing` (V, E, V'_{h0}, V'_{h1});
    si (P_0, S, P_1) *est meilleure que* $(P_0^\star, S^\star, P_1^\star)$ **alors**
        $(P_0^\star, S^\star, P_1^\star) \leftarrow (P_0, S, P_1)$;
retourner $(P_0^\star, S^\star, P_1^\star)$;

Nous avons appliqué le même protocole de test que dans la sous-section 3.4.2 précédente. Les résultats sont montrés dans la table 3.3 où les colonnes HF font référence à l'algorithme du *halo-first greedy graph growing*. Nous avons également reporté les résultats de DG et HF dans une même table 3.4 pour les comparer l'un avec l'autre ; le meilleur est mis en valeur en gras.

Sur le critère de l'interface, l'approche HF est meilleure que le GG non-modifié et que le $GG^\star$ dans toutes les matrices sauf quatre. Le pire cas se présente sur la matrice `turon_m` (4), mais cet échec apparent est dû au fait que GG se comporte particulièrement bien sur cette matrice : le déséquilibre de l'interface est de seulement 244 pour un nombre de sommets égal à

	Déséquilibre des interfaces			Déséquilibre des intérieurs		
id	*GG*	% *GG**	% *HF*	*GG*	% *GG**	% *HF*
1	297	7,4	**-29,3**	1311	-69,1	**-73,5**
2	1112	1,6	**-34,9**	4678	-66,5	**-74,2**
3	522	-11,3	**-63,4**	1635	**-13,9**	-7,2
4	244	**-9,0**	103,3	1568	-23,7	**-24,5**
5	475	-17,9	**-26,7**	4605	**-85,6**	-74,3
6	869	-21,3	**-48,2**	4107	**-78,5**	-65,6
7	905	49,4	**-24,2**	4942	-69,1	**-72,2**
8	261	0,0	**-69,0**	420	0,0	**-11,4**
9	365	-3,8	**-41,4**	8128	**-82,8**	-69,2
10	1002	14,0	**-66,9**	4852	**-73,2**	-47,5
11	509	-3,7	**-50,3**	2831	**-84,1**	-19,1
12	569	-25,0	**-70,1**	22416	**-79,5**	-72,9
13	532	-11,8	**-37,0**	12049	**-69,0**	-52,3
14	756	**-23,1**	-20,4	17458	-61,9	**-67,2**
15	6678	6,6	**-22,4**	35466	**-73,9**	-68,2
16	564	-11,9	**-54,4**	15092	**-65,5**	-59,2
17	2130	18,6	**-44,8**	32202	-72,9	**-73,0**
18	**335**	17,0	14,6	28057	**-70,1**	-68,3
19	2604	**-16,4**	-15,8	25853	**-66,5**	-49,7
20	9990	-14,2	**-55,4**	73635	**-80,7**	-79,6
21	927	0,6	**-49,4**	2377	-58,6	**-64,5**
22	3468	5,0	**-75,4**	**3336**	4,3	16,1
23	1869	7,1	**-24,2**	14424	**-73,3**	-72,0
24	2460	-9,1	**-55,9**	8940	-44,1	**-51,5**
25	6837	-11,6	**-65,5**	26877	-63,0	**-80,3**
26	780	-15,9	**-33,6**	24987	-58,3	**-66,0**
27	6168	-27,5	**-73,6**	67721	-68,7	**-74,5**
28	4344	-15,6	**-47,9**	240729	**-76,9**	-73,2
29	11539	8,6	**-57,0**	244959	**-73,9**	-70,9
30	9936	-6,9	**-55,9**	286992	**-72,6**	-71,4

TABLE 3.3 – Résultats du *halo-first greedy graph growing* en comparaison avec le *greedy graph growing* de SCOTCH. *Résultats obtenus avec un découpage en 16 domaines.*

	Déséquilibre des interfaces			Déséquilibre des intérieurs		
id	*GG*	% *DG*	% *HF*	*GG*	% *DG*	% *HF*
1	297	-19,2	**-29,3**	1311	-23,3	**-73,5**
2	1112	**-40,0**	-34,9	4678	**-78,9**	-74,2
3	522	-39,7	**-63,4**	1635	**-13,0**	-7,2
4	244	**-11,5**	103,3	1568	**-31,1**	-24,5
5	475	-3,8	**-26,7**	4605	-62,3	**-74,3**
6	869	-41,3	**-48,2**	4107	-49,6	**-65,6**
7	905	26,4	**-24,2**	4942	-63,7	**-72,2**
8	261	-46,7	**-69,0**	420	79,8	**-11,4**
9	365	**-44,9**	-41,4	8128	-65,2	**-69,2**
10	1002	-32,1	**-66,9**	4852	-44,6	**-47,5**
11	509	-44,4	**-50,3**	2831	27,5	**-19,1**
12	569	-59,1	**-70,1**	22416	-67,5	**-72,9**
13	532	**-58,5**	-37,**0**	12049	-49,5	**-52,3**
14	756	**-46,3**	-20,4	17458	-66,6	**-67,2**
15	6678	**-29,8**	-22,4	35466	**-68,4**	-68,2
16	564	**-62,9**	-54,4	15092	-58,**0**	**-59,2**
17	2130	**-50,4**	-44,8	32202	-67,7	**-73,0**
18	335	**-19,7**	14,6	28057	-64,9	**-68,3**
19	2604	**-42,0**	-15,8	25853	**-59,5**	-49,7
20	9990	-25,8	**-55,4**	73635	-16,5	**-79,6**
21	927	-40,9	**-49,4**	2377	-59,5	**-64,5**
22	3468	**-78,5**	-75,4	3336	18,2	**16,1**
23	1869	-3,2	**-24,2**	14424	-64,4	**-72,0**
24	2460	**-73,4**	-55,9	8940	**-60,6**	-51,5
25	6837	**-69,7**	-65,5	26877	-68,9	**-80,3**
26	780	**-53,8**	-33,6	24987	-65,2	**-66,0**
27	6168	-68,4	**-73,6**	67721	**-76,2**	-74,5
28	4344	-41,4	**-47,9**	240729	**-77,1**	-73,2
29	11539	-51,3	**-57,0**	244959	**-77,2**	-70,9
30	9936	-52,0	**-55,9**	286992	-8,5	**-71,4**

TABLE 3.4 – Comparaison entre le *double greedy graph growing* et le *halo-first greedy graph growing.Résultats obtenus avec un découpage en 16 domaines.*

189 924. Globalement, le gain moyen de HF par rapport à GG non-modifié pour l'interface est de 38%, avec un maximum de $75,4\%$; cela est presque aussi bien que DG. Si on compare les gains de HF par rapport à DG sur ce critère, HF bat DG dans 16 cas sur 30, ce qui confirme cette tendance.

De plus, HF parvient à un gain sur le déséquilibre des intérieurs pour toutes les matrices excepté une. En moyenne, ce déséquilibre est amélioré de 56% par HF. Cela est meilleur que DG d'environ 10%, et si les gains sont comparés un à un, HF bat DG sur deux tiers de nos matrices de test. DG n'est pas obsolète cependant : par exemple, sur la matrice `ultrasound` (24), DG aboutit à un meilleur résultat sur les deux critères.

Pour finir, la figure 3.11 donne une vision graphique des résultats sur les trois plus grosses matrices de ce tableau. La barre correspondant à GG dans les deux graphiques est fixée à 100% et les résultats de $GG^\star$, DG et HF sont affichés relativement à lui. Le diagramme de gauche montre les déséquilibres des interfaces, celui de droite les déséquilibres des intérieurs.

On observe que DG et HF fournissent un équilibrage des interfaces significativement plus important que GG et $GG^\star$, légèrement meilleur dans le cas de HF que DG. En dehors de la matrice `10millions` pour laquelle DG fournit un déséquilibre des intérieurs relativement important (mais toutefois inférieur à celui de GG), les trois méthodes $GG^\star$, DG et HF donnent sensiblement les mêmes équilibrages d'intérieur.

3.5 Résultats

Dans cette section, nous présentons les résultats obtenus avec nos algorithmes modifiés. La section précédente en a déjà exposés quelques-uns, dans le but de comparer les différents algorithmes entre eux au fur et à mesure de leur introduction. Tous ces tests étaient effectués avec 16 domaines et en mesurant directement les tailles des domaines et des interfaces. À présent notre objectif est de répondre à deux nouvelles questions : d'une part, comment nos algorithmes se comportent-ils lorsqu'on divise en davantage de domaines ; et d'autre part, les meilleurs équilibrages que nous obtenons sur les graphes permettent-ils en pratique un meilleur équilibrage dans la chaîne d'exécution complète d'un solveur hybride tel que MaPHyS.

Les deux sous-sections suivantes ont pour but de répondre à chacune de ces questions. Nous rappelons que les matrices de tests que nous utilisons ont été présentées dans la table 3.1 page 61. En dehors du nombre de domaines, les paramètres de test sont les mêmes que dans la section précédente : GG désigne l'algorithme du *greedy graph growing* avec une valeur $bal = 10\%$ (correspondant à une version de Scotch non-modifiée), et $GG^\star$, DG et HF ont été utilisés avec une balance variable $bal = \frac{10}{2^{p-i+1}}\%$ et une balance-seuil $minbal = 1\%$.

Avant de commencer, nous présentons un résultat visuel avec 16 domaines, dont le but est de donner un aperçu des types d'équilibrage que peuvent fournir les différents algorithmes. Pour plus de clarté, nous avons désactivé le multiniveau pour cette expérience. Ainsi, les découpages ont été réalisés avec un bipartitionnement récursif utilisant exclusivement l'un des algorithmes GG, DG ou HF suivi d'un raffinement avec l'algorithme de Fiduccia-Mattheyses. Les résultats sont montrés sur la figure 3.12 sur le maillage `mario002` ($id = 8$).

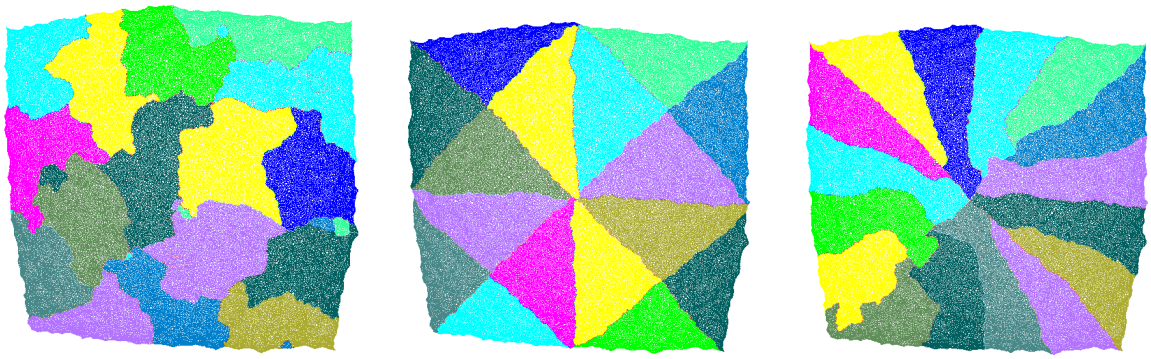


FIGURE 3.12 – Partitionnement d'un graphe du graphe de la matrice `mario002` en 16 domaines avec SCOTCH. De gauche à droite, la méthode appliquée est le *greedy graph growing*, le *double greedy graph growing* et le *halo-first greedy graph growing*.

On peut constater que l'algorithme *GG* créé des domaines aux formes irrégulières, menant ici à un déséquilibre des interfaces de 224. Au contraire, l'algorithme *DG* se comporte mieux sur cet exemple, obtenant 16 domaines de forme triangulaire. Le déséquilibre des interfaces est cette fois de 151, et provient du fait que les huit triangles au centre ont leurs trois côtés adjacents aux autres domaines, alors que les huit du bord ont un côté sur l'extérieur du graphe, ne touchant pas d'autre domaine. Enfin, l'algorithme *HF* est le meilleur des trois en parvenant à un déséquilibre de seulement 145. Pour obtenir cela, il a construit des domaines de forme allongée tout autour du « centre » du graphe.

3.5.1 Etude de scalabilité

Nous présentons maintenant une étude de scalabilité afin de montrer le comportement de nos algorithmes avec un nombre variable de domaines. Comme nous l'avons expliqué dans le chapitre 1, plus le nombre de domaines est élevé, plus la convergence de la méthode itérative est difficile. Nous visons donc un découpage en un nombre modéré de domaines, jusqu'à un demi millier, ce qui représente 9 niveaux de bipartitionnement récursif (512 domaines). Au-delà de ce découpage, davantage de parallélisme peut être exploité en employant plusieurs processus par domaine et plusieurs threads par processus.

Les résultats sont reportés dans les tables 3.5, 3.6 et 3.7. Pour cette étude, nous nous sommes focalisés sur les trois plus grosses matrices de notre ensemble de test : la `Almond` (28), la `NICE-7` (29) et la `10millions` (30). La colonne *dom* donne le nombre de domaines. Comme dans la section précédente, la colonne suivante donne le déséquilibre des interfaces de l'algorithme *GG*, suivi des pourcentages de gains de $GG^\star$, *DG* et *HF* relativement à *GG* pour cette métrique. Le groupe de quatre colonnes suivantes est similaire, mais concerne quant à lui le déséquilibre des intérieurs. Dans chaque groupe de colonnes, la valeur ou le gain du meilleur algorithme est mis en évidence par l'emploi d'une police grasse.

La figure 3.13 illustre les données présentées dans les tables. L'axe des abscisses représente le nombre de domaines, et son pas est logarithmique. L'axe des ordonnées représente le déséquilibre

	Déséquilibre des interfaces				Déséquilibre des intérieurs			
dom	*GG*	% $GG^\star$	% *DG*	% *HF*	*GG*	% $GG^\star$	% *DG*	% *HF*
16	4344	-15,6	-41,4	**-47,9**	240729	-76,9	**-77,1**	-73,2
32	3179	**-34,8**	-31,5	-0,5	133886	-73,8	**-80,1**	-76,9
64	2258	-5,8	**-47,6**	-17,8	83819	**-80,5**	-79,2	-79,1
128	1822	-29,5	**-42,9**	-32,6	48087	-78,4	-77,6	**-80,9**
256	1071	-2,2	**-44,4**	2,5	27695	-81,6	-77,9	**-83,0**
512	910	0,8	-17,1	**-22,3**	16243	-83,8	-80,0	**-84,7**

TABLE 3.5 – Résultats de *GG*, *DG* et *HF* avec différents nombres de domaines pour la matrice `Almond`.

	Déséquilibre des interfaces				Déséquilibre des intérieurs			
dom	*GG*	% $GG^\star$	% *DG*	% *HF*	*GG*	% $GG^\star$	% *DG*	% *HF*
16	11539	8,6	-51,3	**-57,0**	244959	-73,9	**-77,2**	-70,9
32	10991	-26,3	**-45,7**	-38,5	188834	-80,3	-81,0	**-81,6**
64	8997	-27,5	40,7	**-30,8**	101838	-75,9	-6,6	**-80,2**
128	5694	-14,4	11,7	**-26,9**	66792	-83,9	-1,9	**-84,3**
256	4554	-3,2	**-33,1**	-27,6	34314	-77,4	7,3	**-82,4**
512	3762	-16,7	-30,8	**-33,1**	19734	-79,1	-6,3	**-84,2**

TABLE 3.6 – Résultats de *GG*, *DG* et *HF* avec différents nombres de domaines pour la matrice `NICE-7`.

	Déséquilibre des interfaces				Déséquilibre des intérieurs			
dom	*GG*	% $GG^\star$	% *DG*	% *HF*	*GG*	% $GG^\star$	% *DG*	% *HF*
16	9936	-6,9	-52,0	**-55,9**	286992	**-72,6**	-8,5	-71,4
32	6666	-0,5	43,7	**-56,6**	188900	-69,5	13,0	**-77,7**
64	7036	-13,3	**-47,4**	-31,1	125444	-76,9	-18,8	**-78,4**
128	4564	-11,2	4,0	**-48,7**	79754	**-82,9**	-52,9	-78,8
256	3114	-6,2	163,5	**-32,5**	42931	-79,5	-30,2	**-80,8**
512	2336	-19,5	22,2	**-54,2**	25800	**-83,5**	49,3	-83,4

TABLE 3.7 – Résultats de *GG*, *DG* et *HF* avec différents nombres de domaines pour la matrice `10millions`.

des interfaces (pour les graphiques de la partie gauche) ou des intérieurs (pour les graphiques de la partie droite). Les résultats de chaque algorithme sont donnés par une courbe, en bleu pour l'algorithme de référence GG et en rouge, jaune et vert respectivement pour les algorithmes $GG^\star$, DG et HF.

On peut tout d'abord constater que, dans presque toutes les configurations, une de nos stratégies réalise un meilleur partitionnement que $GG^\star$. Lorsque tel n'est pas le cas, au moins une d'entre elle en est très proche. Nous remarquons également que sur plus de 16 domaines, DG ne fonctionne pas toujours très bien : sur la matrice `10millions`, dans le cas de 32 domaines, il détériore à la fois l'équilibrage de l'interface et des intérieurs en comparaison à GG original. DG est toutefois la meilleure stratégie (ou à quelques pourcents près de la meilleure) pour tous les nombres de domaines dans le cas de la matrice `Almond`.

En ce qui concerne l'algorithme HF, on constate qu'il permet d'effectuer de meilleurs équilibrages que DG en général. De plus, l'algorithme présente l'avantage d'être très stable ; dans tous les tests effectués, la courbe verte de HF est strictement en-dessous de la courbe bleue GG. Dans les cas de test `Nice-7` et `10millions`, quelque soit le nombre de domaines, HF effectue systématiquement un meilleur équilibrage des interfaces que $GG^\star$, et l'écart est significatif : pour la matrice `10millions` en particulier, le gain supplémentaire varie entre 18 et 56 %. Cette performance remarquable est réalisée sans compromis pour l'équilibrage des domaines ; pour les trois matrices, les courbes de $GG^\star$ et HF sont en effet quasiment confondues pour cette métrique.

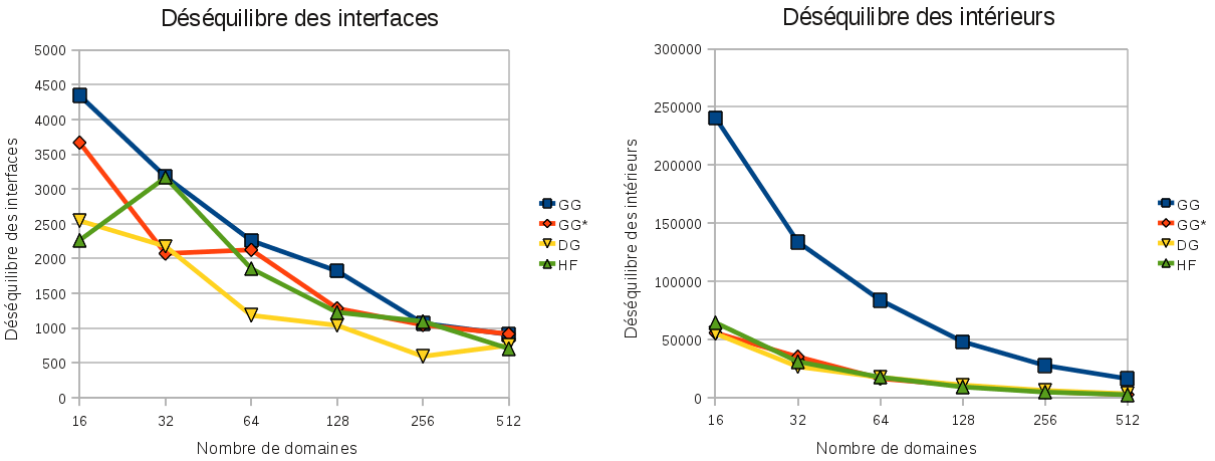
Compte tenu de ces résultats, nous pensons donc que l'algorithme HF doit être privilégié par rapport à DG lorsque le nombre de domaines devient important. Cependant, comme nous l'avons souligné pour la matrice `Almond`, il n'est pas toujours le meilleur. Dans un contexte de partitionnement parallèle, il pourrait être envisagé d'exécuter les deux approches et de prendre le meilleur des deux partitionnements obtenus.

Pour terminer, on peut remarquer que les gains d'équilibrage des intérieurs aussi bien pour DG que pour HF ne semblent pas fortement corrélés au nombre de domaines. En revanche, les gains d'équilibrage des interfaces semblent plutôt diminuer lorsque le nombre de domaines augmente, bien qu'il existe de nombreuses exceptions dans les trois tables. Néanmoins, les gains pour les deux métriques restent significatifs jusqu'à 512 domaines : en particulier, pour HF, les gains s'étendent de -22% à -54% pour l'interface et de -83 à -85% pour les intérieurs.

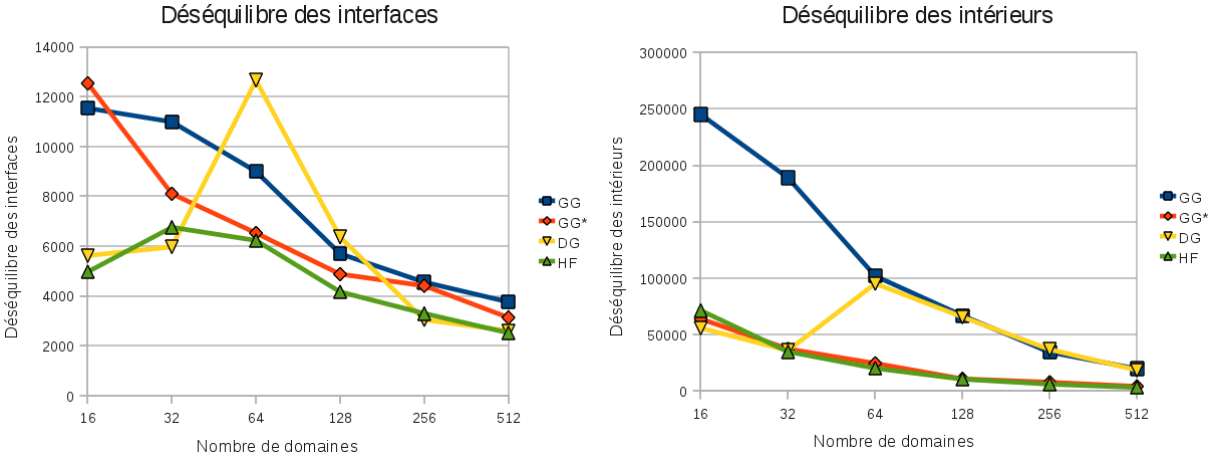
Nous venons de montrer la scalabilité de nos algorithmes au moins jusqu'à 512 domaines. Il reste à présent à vérifier les gains temporel qu'ils permettent dans la chaîne complète d'exécution d'un solveur hybride, ce qui constituait la motivation initiale de cette étude. Ce problème est l'objet de la section suivante.

3.5.2 Résultats dans le solveur hybride MaPHyS

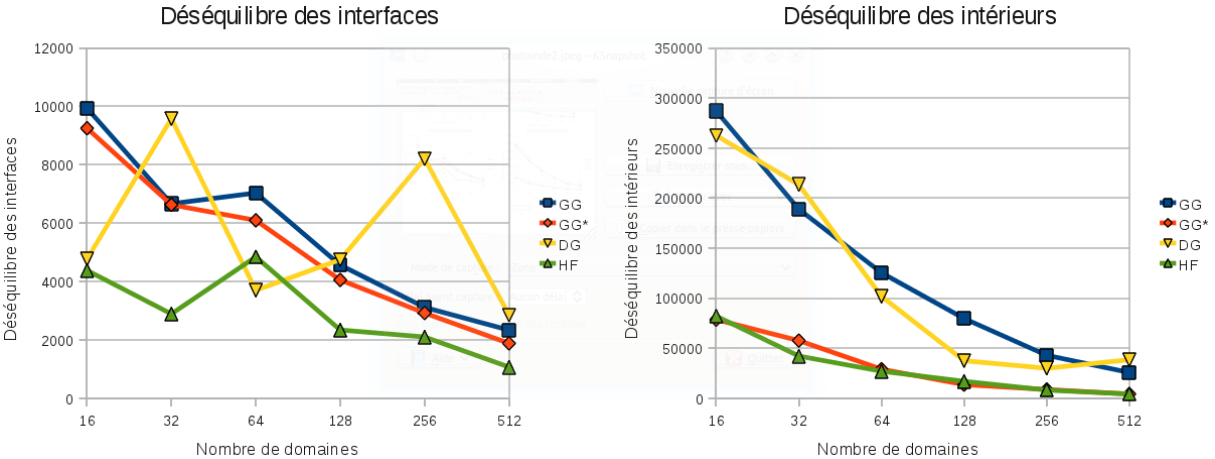
L'introduction de ce chapitre a rappelé le fonctionnement général des méthodes hybrides, et en particulier celui du solveur MaPHyS. Nous avons donné la formule de complexité de ce solveur pour les maillages d'éléments finis 3D (équation (3.1) page 46), qui constituent les cas qui nous intéressent. Cette formule faisait apparaître trois termes, correspondants respectivement à la complexité de la factorisation de l'intérieur d'un domaine, la complexité du calcul du



(a) Résultats pour la matrice `Almond`.



(b) Résultats pour la matrice `Nice-7`.



(c) Résultats pour la matrice `10Millions`.

FIGURE 3.13 – Scalabilité du déséquilibre de l'interface et des intérieurs en fonction du nombre de domaines.

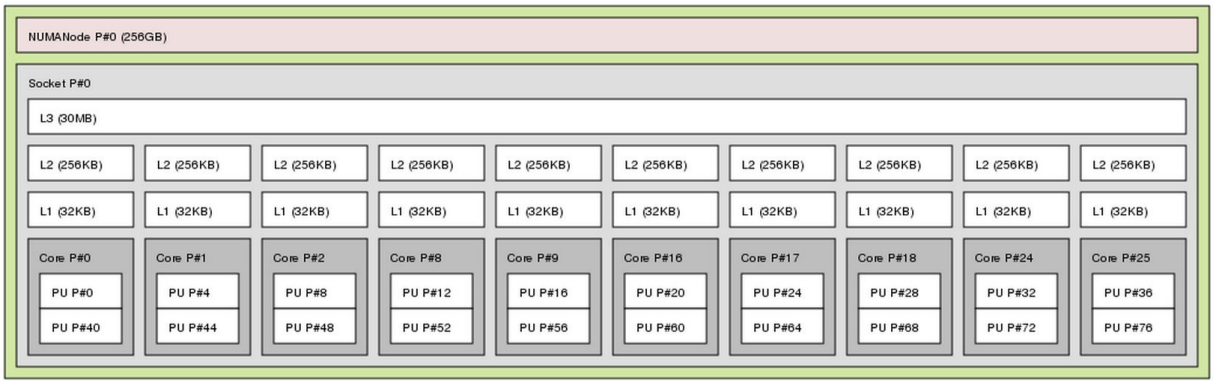


FIGURE 3.14 – Description d'un des quatre déca-cœurs de la machine *riri*.

préconditionneur local et la complexité de la résolution itérative. Nous avons souligné que dans chacun de ces trois termes, le rôle joué par la taille de l'interface locale était asymptotiquement au moins égal à celui de la taille du domaine. Dans le cas des deux dernières étapes, il était même prépondérant. Nous en avons déduit l'importance de l'équilibrage à la fois des nœuds des intérieurs et des nœuds des interfaces. Il en a alors découlé notre étude des différents algorithmes du bipartitionnement récursif de graphes présentée dans les sections précédentes. Pour chacun des algorithmes, nous avons proposé des adaptations afin de satisfaire les nouvelles contraintes. Nous avons ensuite implémenté ces variantes dans le partitionneur SCOTCH et vérifié que celles-ci étaient satisfaites.

Nous revenons à présent au point de départ de notre travail : les solveurs hybrides. Il nous reste à montrer que, comme l'équation (3.1) nous l'a laissé supposer au début, un meilleur équilibrage des intérieurs et des interfaces permet effectivement d'améliorer l'équilibrage de la charge lors des trois dernières étapes d'un solveur hybride.

Pour cette nouvelle série d'expérimentations, nous avons décidé de nous placer dans le cadre du solveur hybride MAPHYS. Celui-ci a été décrit à la section 1.3.4 page 22 du chapitre 1, et nous en avons également rappelé les grandes lignes dans l'introduction du présent chapitre. Nous avons choisi d'utiliser un seuillage numérique pour le préconditionneur, dont nous avons fixé la valeur à 10^{-2}. De plus, le critère pour déterminer la convergence de la solution a été fixé à 10^{-5}.

Tous les résultats qui vont suivre ont été obtenus en exécutant MAPHYS sur la machine *riri* de la plateforme d'expérimentation de Bordeaux PLAFRIM. Cette machine dispose de 1TB de RAM et est constituée de 4 déca-cœurs Intel® Xeon® E7-4870 (2,40GHz). Chacun d'eux possède la structure décrite par la figure 3.14 et ils sont reliés par un réseau Ethernet de vitesse 1Gb/s.

Nous avons sélectionné les matrices `Almond` (28) et `10millions` (30) pour cette étude. Nous avons renoncé à inclure la matrice `Nice-7` (27) car nous ne possédions que la structure de la matrice (c'est-à-dire son graphe d'adjacence) mais pas les valeurs associées. À la place, nous avons choisi deux autres matrices un peu plus petites, la `inline` (23) et la `MHD` (22). Notre étude a consisté à exécuter MAPHYS sur chacune de ces 4 matrices avec un nombre de domaines égal à 8, 16 et 32 successivement, pour les algorithmes *GG*, *DG* et *HF*.

Les résultats sont reportés dans les tables 3.8 (matrice `MHD`), 3.9 (matrice `inline`), 3.10 (matrice `Almond`) et 3.11 (matrice `10millions`). Pour chaque cas, nous donnons un ensemble d'informations relatives aux intérieurs : la taille moyenne des domaines, la taille du plus grand domaine et le déséquilibre entre le plus grand et le plus petit domaine. Nous donnons ensuite les mêmes informations pour les interfaces locales, et y ajoutons la taille de l'interface globale. Enfin, nous donnons les informations temporelles pour les trois étapes dont la complexité est susceptible d'être modifiée : la factorisation des intérieurs (factorisation de B_i, calcul de W_i et G_i puis du complément de Schur local S_i), le calcul du préconditionneur (assemblage du Schur local S_i en $\overline{S_i}$ et factorisation de $\overline{S_i}$) et la résolution itérative. Dans tous les cas, c'est le temps maximal pour tous les domaines qui est donné. Pour la résolution itérative, nous avons décidé de n'inclure que le temps moyen par itération, les temps pour calculer le second membre $b_S = B_C - Ex'_B$ (équation (1.6) page 13) et déduire x_B (équation (1.3) page 13) étant très faibles. Le nombre d'itérations est également fourni afin de vérifier la qualité de la convergence. Enfin, nous donnons le temps total, égal à la somme du temps de factorisation des intérieurs, du temps de calcul du préconditionneur et du temps par itération multiplié par le nombre d'itérations.

La politique de soumission de jobs sur la machine *riri* permet de lancer des exécutions pouvant durer jusqu'à 24 heures. Malheureusement, lorsque nous avons lancé l'expérimentation pour la matrice `10millions` avec l'algorithme GG, le temps limite a été dépassé avec 8 et 16 domaines. Les algorithmes DG et HF ont quant à eux terminé dans les temps mais les résultats ne servent à rien sans la comparaison avec GG, c'est pourquoi nous nous sommes contentés de donner les résultats complets pour 32 domaines pour cette matrice.

	8 domaines			16 domaines			32 domaines		
	GG	DG	HF	GG	DG	HF	GG	DG	HF
Intérieurs									
moy. domaine	64926	65041	65121	33979	34123	34086	18066	18146	18197
max. domaine	79575	66250	66072	48371	36127	36240	25585	20471	20684
déséq. domaines	35691	2275	1798	26878	3830	3972	16107	4221	4328
Interfaces									
moy. interface	7699	7847	7960	6322	6626	6574	4756	4916	4992
max. interface	9349	9062	8618	9516	8570	8805	7440	7322	7514
déséq. interfaces	3484	2271	1131	5684	3794	4000	5066	4357	4435
interface totale	27783	28045	28308	43086	45652	45415	59694	62248	63033
Temps									
temps facto.	132,5	**104,4**	106,3	66,5	37,0	**35,4**	15,0	**10,0**	11,2
temps précond.	4,9	4,0	**3,6**	**4,0**	**4,0**	4,3	3,3	**3,0**	3,1
temps par itér.	0,19	0,19	**0,16**	0,22	0,19	**0,16**	0,14	**0,13**	0,18
nombre d'itér.	**15**	**15**	16	**16**	**16**	17	18	**17**	**17**
temps total	140,2	**111,2**	112,5	74,0	44,0	**42,4**	20,8	**15,2**	17,4

TABLE 3.8 – Comparaison des différents algorithmes d'équilibrage dans MAPHYS avec la matrice `MHD`. *Les temps sont donnés en secondes.*

Nous passons à présent à l'analyse des résultats. Nous nous intéressons tout d'abord à la convergence des systèmes. A priori, nos algorithmes n'ont pas de raison particulière de modifier

	8 domaines			16 domaines			32 domaines		
	GG	*DG*	*HF*	*GG*	*DG*	*HF*	*GG*	*DG*	*HF*
Intérieurs									
moy. domaine	63553	63939	63895	32100	32311	32181	16298	16401	16427
max. domaine	95424	71577	71457	49710	36051	37131	28401	18870	18612
déséq. domaines	47646	13572	14418	31512	8358	9390	20457	4215	4470
Interfaces									
moy. interface	1179	1916	1812	1210	1602	1358	1050	1237	1293
max. interface	1833	3264	2727	2520	3225	2277	2064	2103	2145
déséq. interfaces	1368	2592	1785	2001	2484	1746	1818	1671	1569
interface totale	4719	7524	7041	9474	12378	10542	15768	18456	19452
Temps									
temps facto.	13,4	8,5	**8,2**	6,3	4,9	**4,6**	3,8	**2,1**	**2,1**
temps précond.	0,2	**0,5**	**0,5**	**0,3**	0,7	**0,3**	**0,3**	**0,3**	**0,3**
temps par itér.	**0,012**	0,025	0,019	0,018	0,027	**0,015**	0,020	**0,016**	0,017
nombre d'itér.	**67**	68	81	65	62	**51**	68	74	**66**
temps total	14,4	10,7	**10,2**	7,8	7,2	**5,7**	5,5	3,6	**3,5**

TABLE 3.9 – Comparaison des différents algorithmes d'équilibrage dans MAPHYS avec la matrice `inline`. *Les temps sont donnés en secondes.*

	8 domaines			16 domaines			32 domaines		
	GG	*DG*	*HF*	*GG*	*DG*	*HF*	*GG*	*DG*	*HF*
Intérieurs									
moy. domaine	88.10^4	88.10^4	88.10^4	44.10^4	44.10^4	44.10^4	22.10^4	22.10^4	22.10^4
max. domaine	12.10^5	10.10^5	10.10^5	74.10^4	51.10^4	47.10^4	33.10^4	25.10^4	25.10^4
déséq. domaines	73.10^4	24.10^4	24.10^4	49.10^4	12.10^4	8.10^4	21.10^4	6.10^4	5.10^4
Interfaces									
moy. interface	10466	10532	11254	7523	7709	7900	5492	5672	5749
max. interface	12958	12334	12171	10074	8842	9891	7099	6559	7010
déséq. interfaces	6546	3221	1779	5805	3455	3307	3118	2765	2603
interface totale	41604	41869	44768	59574	61057	62608	86551	89452	90664
Temps									
temps facto.	1069	748,9	**677,8**	486,7	268,0	**265,5**	151,9	**96,1**	104,6
temps précond.	481,4	**227,5**	276,0	701,3	**213,5**	284,0	364,6	168,9	**112,3**
temps par itér.	146,6	**84,6**	117,8	128,5	**91,1**	97,9	90,5	59,4	**50,0**
nombre d'itér.	500	500	500	500	500	500	500	500	500
temps total	7.10^4	$\mathbf{4.10^4}$	6.10^4	7.10^4	$\mathbf{5.10^4}$	5.10^4	5.10^4	3.10^4	$\mathbf{3.10^4}$

TABLE 3.10 – Comparaison des différents algorithmes d'équilibrage dans MAPHYS avec la matrice `Almond`. *Les temps sont donnés en secondes.*

	GG	*DG*	*HF*
Intérieurs			
moy. domaine	33.10^4	33.10^4	33.10^4
max. domaine	43.10^4	38.10^4	37.10^4
déséq. domaines	26.10^4	9.10^4	8.10^4
Interfaces			
moy. interface	8312	8498	8786
max. interface	10379	10377	10211
déséq. interfaces	9144	6335	6042
interface totale	130818	133749	138497
Temps			
temps facto.	472,3	356,4	**345,2**
temps précond.	373,1	**174,9**	229,3
temps par itér.	154,0	**109,6**	119,9
nombre d'itér.	500	500	500
temps total	79.10^3	$\mathbf{55.10^3}$	61.10^3

TABLE 3.11 – Comparaison des différents algorithmes d'équilibrage dans MAPHYS avec la matrice `10millions`. *Les temps sont donnés en secondes et le découpage est effectué en 32 domaines.*

celle-ci. Pour les matrices `Almond` et `10millions`, le système n'a pas convergé, et nous avons arrêté les expériences au bout de 500 itérations. Remarquons cependant que le système ne convergeait déjà pas avec la version native de SCOTCH (algorithme *GG*). Pour les deux autres matrices, le nombre d'itérations ne semble pas varier sensiblement ; le pire cas est une augmentation de 14 itérations entre *GG* et *HF* pour la matrice `inline` découpée en 8 domaines. Par ailleurs, pour un découpage en 16 domaines, la situation est inversée ; c'est au tour de *GG* de réaliser 14 itérations de plus que *HF*. Enfin, pour la matrice `MHD`, la variation du nombre d'itérations n'excède pas 1 dans tous les cas. Nous concluons de tout cela que nos nouvelles méthodes de partitionnement n'influencent pas significativement la robustesse du préconditionneur construit, ni à la hausse, ni à la baisse, ainsi qu'on pouvait s'y attendre.

En ce qui concerne les équilibrages, nous remarquons plusieurs tendances, valables aussi bien pour *DG* que pour *HF*. Tout d'abord, comme nous l'avons vu dans les sections précédentes, on observe une très forte réduction du déséquilibre des intérieurs. Celle-ci s'étend de 66 à 95 % sur l'ensemble des cas. En outre, elle est accompagnée par une baisse significative de 11 à 37% de la taille du plus grand domaine. Pour ce qui est des équilibrage des interfaces, nous mettons à part la matrice `inline` qui semble un peu problématique. Pour les 3 autres matrices, on constate que le déséquilibre baisse fortement, de 11 à 73% selon les cas. D'autre part, la taille de la plus grande interface locale est diminuée (ou, dans le pire cas, quasiment stable) ; cette réduction est modérée mais peut malgré tout atteindre jusqu'à 12% pour la matrice `Almond` divisée en 16 domaines avec *DG*. En contrepartie, on remarque une hausse de la taille moyenne des interfaces locales (1 à 7%) et de la taille totale de l'interface (1 à 8%). Cela est dû au fait que pour optimiser l'équilibre des interfaces, nous avons relâché la contrainte de minimisation des séparateurs dans nos algorithmes.

Sur le plan des données temporelles à présent, on observe une réduction systématique du

temps de factorisation des intérieurs. Nous rappelons que l'équation (3.1) prévoyait pour cette étape une complexité de $\mathcal{O}(n_i^2 + n_i^{4/3} s_i + n_i^{2/3} s_i^2)$, qui dépendait donc à la fois de la taille des intérieurs et de la taille des interfaces. Du fait que nous n'avons pas réalisé les expériences avec l'algorithme $GG^\star$, il n'est pas possible d'affirmer avec certitude que le meilleur équilibrage des interfaces a joué un rôle significatif dans la baisse de ce temps ; nous pensons néanmoins que c'est le cas. En revanche, il est raisonnable de penser que le meilleur équilibrage des intérieurs, à lui tout seul, a eu un impact sur cette baisse. Nous pouvons en effet constater que même dans les cas où le déséquilibre de l'interface augmente pour la matrice `inline`, la réduction du temps de factorisation des intérieurs se produit bel et bien.

Les deux autres temps qui nous intéressent sont le temps de calcul du préconditionneur et le temps par itération. La formule de l'équation (3.1) donnait une complexité respectivement de $\mathcal{O}(s_i^3)$ et $\mathcal{O}(s_i^2)$ pour ces termes. Cela signifie que, cette fois-ci, seule la taille des interfaces compte. C'est donc sans surprise que l'on constate que sur la matrice `inline`, le temps de calcul du préconditionneur et le temps par itération sont réduits (ou stables) lorsque le déséquilibre des interfaces a baissé et sont augmentés dans le cas contraire. À l'exception de deux anomalies marginales pour la matrice `MHD`, cela est également vrai pour les trois autres matrices, pour lesquelles le déséquilibre est systématiquement réduit, de même que les deux temps qui nous intéressent. On peut souligner les gains temporels particulièrement importants pour les deux plus grosses matrices `Almond` et `10millions` ; pour ces deux cas, le baisse moyenne sur le temps de calcul du préconditionneur avoisine 55%, et celle du temps par itération 31%.

Toujours à propos de ces deux mesures temporelles, nous avons cependant remarqué une chose curieuse que nous n'expliquons pas. Il semblerait en effet que la diminution de ces temps soit davantage corrélée au déséquilibre des interfaces (c'est-à-dire à la différence entre la plus grande et la plus petite interface locale) qu'à la taille de la plus grande interface locale. On le voit bien, notamment, pour la matrice `10millions`, dans le cas de l'algorithme *DG* : la taille de l'interface maximum est stable, et pourtant le temps de calcul du préconditionneur et le temps par itération baissent respectivement de 53 et 29%, de même que le déséquilibre des interfaces (31% de gain). Pourtant, les temps que nous avons fournis sont les temps pour le domaine le plus défavorable et donc, en principe, le plus gros. Une hypothèse est que peut être le nombre maximum de valeurs non-nulles dans les matrices d'interfaces locales a été réduit.

Pour terminer, on peut remarquer une baisse du temps total sur l'ensemble des cas, allant de 8 à 46% avec une moyenne de 29%.

Conclusion

Tout au long de ce chapitre, nous nous sommes intéressés au problème de l'équilibrage de la charge pour la phase de factorisation des domaines dans les méthodes hybrides. Nous avons montré la nécessite d'équilibrer à la fois le nombre de sommets dans les domaines et dans les interfaces locales entourant ceux-ci. Ce besoin se traduit tout au long du bipartitionnement récursif par l'équilibrage des sommets provenant des séparateurs déjà construits, appelés *sommets halo*. Nous avons montré de quelle manière reporter de niveau en niveau les informations concernant les sommets halos dans l'algorithme de multiniveau. Nous avons ensuite proposé des modifications dans l'algorithme de raffinement de séparateur de Fiduccia-Mattheyses pour en tenir compte. Enfin, nous avons donné deux variantes de l'algorithme du *greedy graph growing*

(*GG*), appelées respectivement *double greedy graph growing* (*DG*) et *halo-first greedy graph growing* (*HF*), afin d'équilibrer les sommets du halo lors de la recherche d'un séparateur sur le graphe le plus grossier du multiniveau.

Les principales conclusions de nos résultats sont les suivantes. Premièrement, nous avons implémenté nos algorithmes dans le partitionneur Scotch et montré nos deux algorithmes *DG* et *HF* améliorent significativement l'équilibrage des interfaces, les gains étant en moyenne de l'ordre de 40%. La taille de l'interface locale maximum est également réduite. Ces résultats se font sans compromis sur l'équilibrage des intérieurs, qui restent très bons. En contrepartie, l'interface totale et la moyenne des interfaces locales augmentent. Deuxièmement, l'algorithme *HF* semble se comporter généralement mieux que l'algorithme *DG*. La raison précise reste à définir et il existe malgré tout de nombreux cas où *DG* permet de meilleurs gains. Troisièmement, nous avons montré la scalabilité de nos algorithmes jusqu'à 512 domaines. Enfin, nous avons validé l'utilité de nos algorithmes en les intégrant au solveur hybride MaPHyS. On observe des gains temporels sur les trois principales étapes de la chaîne d'exécution du solveur, qui résultent en une baisse moyenne du temps total de près de 30%.

Conclusion et perspectives

Conclusion

Les simulations numériques revêtent aujourd'hui une grande importance, en particulier afin de prévoir le comportement de phénomènes relevant des lois de la physique. Afin de les effectuer, il est souvent nécessaire de pouvoir résoudre efficacement de très grands systèmes linéaires creux. La méthode hybride par décomposition de domaine et basée sur une approche « complément de Schur » permet de résoudre ce type de problème en offrant un compromis intéressant entre les méthodes directes, coûteuses en temps et en mémoire, et les méthodes itératives, qui ne convergent pas toujours.

L'objectif de cette thèse était d'explorer plusieurs aspects problématiques de ce type de méthode et de proposer des solutions. Premièrement, la consommation mémoire des méthodes hybrides reste assez importante, en dépit d'une significative amélioration si on les compare avec les méthodes directes. Deuxièmement, les étapes les plus coûteuses en temps de calcul et que l'on peut exécuter en parallèle sur les différents domaines exigent un bon équilibrage de la charge afin d'obtenir une bonne scalabilité d'ensemble du solveur.

Dans le chapitre 2, nous avons exposé plusieurs solutions déjà existantes pour réduire la mémoire dans les solveurs hybrides. La plupart de celles-ci reposent sur la suppression de certains termes jugés peu importants dans le couplage entre les domaines et leurs interfaces, dans le complément de Schur et/ou dans sa factorisation $\widetilde{L_S}\widetilde{U_S}$. Nous avons proposé une nouvelle technique reposant sur une gestion particulière de l'ordonnancement des tâches, notamment celles d'allocation et de libération de zones mémoire pour le calcul lors de la factorisation. Nos résultats ont montré que notre technique permettait de réduire de 50 à 85% le pic mémoire sur le processus le plus consommateur dans le cas d'une factorisation des domaines avec un seul processus par domaine. De plus, pour un nombre de processus par domaine modéré, les gains demeurent significatifs. Ainsi, pour tous les cas de test considérés et quelque soit le nombre de domaines (jusqu'à au moins 32), notre méthode réduit d'au moins 10% le pic mémoire lorsque 4 processus par domaine sont utilisés. En outre, nous avons montré que ces résultats variaient très peu avec le nombre de threads par processus utilisés, ce qui donne la possibilité de faire varier celui-ci à loisir dans un cadre de programmation MPI/threads.

Dans le chapitre 3, nous nous sommes intéressés au problème de l'équilibrage de la charge, qui se traduit par un problème de partitionnement de graphe. Nous avons mis en évidence le fait qu'il ne s'agissait pas uniquement d'équilibrer la taille des domaines, mais qu'il fallait aussi également équilibrer les interfaces locales les délimitant ; cet objectif peut être atteint en équilibrant les sommets du halo tout au long de l'algorithme de bipartitionnement récursif. Nous avons passé en revue les différents algorithmes mis en jeu par le partitionnement que sont le multiniveau, le

Fiduccia-Mattheyses et le *greedy graph growing*, et avons proposé des modifications pour chacun d'eux afin de tenir compte du double critère d'équilibrage. En particulier, nous avons introduit les algorithmes du *double greedy graph growing* et du *halo-first greedy graph growing*. Toutes ces modifications ont été implémentées dans le partitionneur Scotch. Nous avons ensuite montré que ces algorithmes permettaient de réaliser un gain moyen sur l'ensemble des matrices de test de l'ordre de 40% en terme d'équilibrage des interfaces locales, tout en préservant un très bon équilibrage des domaines. Ces gains demeurent importants au moins jusqu'à 512 domaines. Nous avons finalement testé notre méthode avec le solveur MaPHyS, ce qui nous a permis de valider ces gains de manière effective avec la chaîne complète d'exécution d'un solveur hybride. Nous avons ainsi montré que le temps total d'exécution était réduit de 30% en moyenne.

Perspectives

Cette thèse a proposé plusieurs solutions effectives pour améliorer la consommation mémoire et l'équilibrage de la charge au sein des méthodes hybrides. Nous exposons à présent plusieurs pistes qui pourraient être suivies pour aller plus loin.

En ce qui concerne la consommation mémoire, nos tests ont été effectués de manière symbolique. Il serait intéressant de les compléter avec des mesures temporelles afin de vérifier que ces modifications ne provoquent pas une augmentation significative des temps de calcul et de communication. En effet, nous avons changé l'ordonnancement des tâches, et les ordonnancements de type « left-looking » sont connus pour donner de moins bonnes performances. Nous avons cependant très peu employé cette manière d'organiser les tâches (et en particulier, uniquement pour des contributions locales) ; par conséquent, nous pensons que l'impact éventuel sur le temps de résolution total devrait être limité.

Concernant l'équilibrage de la charge, plusieurs axes peuvent être explorés afin d'améliorer encore le partitionnement. Premièrement, on peut remarquer dans nos résultats qu'aucune des deux méthodes *double greedy graph growing* et *halo-first greedy graph growing* n'est systématiquement meilleure que l'autre sur l'ensemble des cas de tests et sur les deux critères (équilibrage des intérieurs et des interfaces locales). Dans un cadre parallèle, on pourrait alors envisager d'utiliser les deux méthodes simultanément à chaque niveau du bipartitionnement récursif. La méthode ayant fourni le meilleur résultat serait alors sélectionnée à chaque appel. De plus, une étude pourrait également être menée afin de comprendre pourquoi l'une ou l'autre des méthodes fonctionne mieux dans certains cas, ce qui à terme pourrait permettre de choisir la meilleure méthode sans avoir à exécuter les deux.

Un deuxième point soulevé par notre étude est que, bien que nous parvenions à diminuer l'écart entre la plus grosse et la plus petite interface, l'interface totale entre les domaines a tendance à augmenter en comparaison avec le partitionnement natif de Scotch. Une piste serait donc d'étudier précisément les causes de cette augmentation. De plus, tous nos critères d'optimisation sont locaux dans l'algorithme de bipartitionnement récursif. Or, il pourrait dans certains cas être plus avantageux d'accepter un déséquilibre important des sommets du halo à un certain niveau du partitionnement, pour ensuite retrouver un équilibrage correct au niveau suivant. Une solution localement optimale peut en effet être non-optimale globalement. Il nous semble intéressant d'approfondir cette idée et d'autoriser éventuellement un déséquilibre important à

certains niveaux s'il est possible de corriger ce déséquilibre aux niveaux suivants.

Pour terminer, une dernière voie d'élargissement serait d'étudier la possibilité de mieux équilibrer les interfaces locales dans l'algorithme *k-way*, qui est la principale alternative existante au bipartitionnement récursif. Cette technique est entre autres implémentée dans le partitionneur MeTiS et dans la dernière version du partitionneur Scotch (Scotch 6.0). Dans cet algorithme, le graphe est tout d'abord compressé, puis les k domaines sont créés simultanéments sur le graphe le plus grossier. Le graphe est ensuite décompressé et raffiné avec une version de l'algorithme de Fiduccia-Mattheyses adaptée pour k parties. Une piste serait donc de modifier les différents algorithmes mis en jeu par le *k-way* de la même manière que dans le cas du bipartitionnement récursif étudié dans cette thèse.

Bibliographie

[ADD96] P. R. Amestoy, T. A. Davis, and I. S. Duff. An Approximate Minimum Degree Ordering Algorithm. *SIAM J. Matrix Anal. Appl.*, 17(4) :886–905, October 1996. 9

[ADKL01] P. R. Amestoy, I. S. Duff, J. Koster, and J.-Y. L'Excellent. A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1) :15–41, 2001. 7, 30

[ADL98] P. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal Parallel Distributed Symmetric and Unsymmetric Solvers. *Comput. Methods Appl. Mech. Eng*, 184 :501–520, 1998. 7, 44

[AEL90] C. Ashcraft, S. C. Eisenstat, and J. W. H. Liu. A Fan-in Algorithm for Distributed Sparse Numerical Factorization. *SIAM Journal on Scientific and Statistical Computing*, 11(3) :593–599, 1990. 30

[AGG+13] E. Agullo, L. Giraud, A. Guermouche, A. Haidar, and J. Roman. Parallel Algebraic Domain Decomposition Solver for the Solution of Augmented Systems. *Advances in Engineering Software*, 60–61(0) :23 – 30, 2013. CIVIL-COMP : Parallel, Distributed, Grid and Cloud Computing. 22

[AGGR11] E. Agullo, L. Giraud, A. Guermouche, and J. Roman. Parallel Hierarchical Hybrid Linear Solvers for Emerging Computing Platforms. *Compte Rendu de l'Académie des Sciences : Serie Mécanique*, 339(2-3) :96–103, 2011. 22

[Ash93] C. Ashcraft. The fan-both family of column-based distributed cholesky factorization algorithms. In Alan George, JohnR. Gilbert, and JosephW.H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, volume 56 of *The IMA Volumes in Mathematics and its Applications*, pages 159–190. Springer New York, 1993. 32

[ATNW11] C. Augonnet, S. Thibault, R. Namyst, and P. A. Wacrenier. StarPU : a Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation : Practice and Experience*, 23(2) :187–198, 2011. 44

[BHM00] W. Briggs, V. Henson, and S. McCormick. *A Multigrid Tutorial, Second Edition.* Society for Industrial and Applied Mathematics, second edition, 2000. 5

[Bra86] A. Brandt. Algebraic Multigrid Theory : the Symmetric Case. *Applied mathematics and computation*, 19(1) :23–56, 1986. 5

[CM69] E. Cuthill and J. McKee. Reducing the Bandwidth of Sparse Symmetric Matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, pages 157–172, New York, NY, USA, 1969. ACM.

[CM92] T. F. C. Chan and T. P. Mathew. The Interface Probing Technique in Domain Decomposition. *SIAM J. Matrix Anal. Appl.*, 13(1) :212–238, January 1992. 29

[CR89] P. Charrier and J. Roman. Algorithmique et calculs de complexité pour un solveur de type dissections emboîtées. *Numerische Mathematik*, 55(4) :463–476, 1989. 8

[Dav] Davis, Tim and Hu, Yifan. The University of Florida Sparse Matrix Collection. `http://www.cise.ufl.edu/research/sparse/matrices/`. 1, 30, 60

[Dav06] Timothy A. Davis. *Direct methods for sparse linear systems*, volume 2. Siam, 2006. 5

[DER86] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, Inc., New York, NY, USA, 1986. 5, 31

[DK99] I. S. Duff and J. Koster. The Design and Use of Algorithms for Permuting Large Entries to the Diagonal of Sparse Matrices. *SIAM J. Matrix Anal. Appl.*, 20(4) :889–901, July 1999. 21

[DPSW00] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Shape-optimized Mesh Partitioning and Load Balancing for Parallel Adaptive FEM. *Parallel Computing*, 26 :1555–1581, 2000. 59

[FM82] C. M. Fiduccia and R. M. Mattheyses. A Linear-time Heuristic for Improving Network Partitions. In *Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press. 49, 53

[FMR94] C. Farhat, J. Mandel, and F. X. Roux. Optimal Convergence Properties of the FETI Domain Decomposition Method. *Computer Methods in Applied Mechanics and Engineering*, 115 :365–385, 1994. 11

[Gai09] J. Gaidamour. *Design of a Parallel Hybrid Direct/Iterative Sparse Linear Solver*. Theses, Université Sciences et Technologies - Bordeaux I, December 2009. 19, 20, 26

[GDL07] L. Grigori, J. A. Demmel, and X. S. Li. Parallel Symbolic Factorization for Sparse LU with Static Pivoting. *SIAM J. Sci. Comput.*, 29(3) :1289–1314, 2007. 7

[Geo73] A. George. Nested Dissection of a Regular Finite Element Mesh. *SIAM Journal on Numerical Analysis*, 10(2) :345–363, 1973. 9, 46, 47, 59

[GH08] J. Gaidamour and P. Hénon. A Parallel Direct/Iterative Solver Based on a Schur Complement Approach. In *IEEE 11th International Conference on Computational Science and Engineering*, pages 98–105, Sao Paulo, Brésil, July 2008. 19, 26

[GH09] L. Giraud and A. Haidar. Parallel Algebraic Hybrid Solvers for Large 3D Convection-Diffusion Problems. *Numerical Algorithms*, 51(2) :151–177, 2009. 22

[GHL86] A. George, M. T. Heath, and J. Liu. Parallel Cholesky Factorization on a Shared-memory Multiprocessor. *Linear Algebra and its applications*, 77 :165–187, 1986. 30

[GHS10] L. Giraud, A. Haidar, and Y. Saad. Sparse Approximations of the Schur Complement for Parallel Algebraic Hybrid Linear Solvers in 3D. Rapport de recherche RR-7237, INRIA, March 2010. 22

[GKK97] A. Gupta, G. Karypis, and V. Kumar. Highly Scalable Parallel Algorithms for Sparse Matrix Factorization. *Parallel and Distributed Systems, IEEE Transactions on*, 8(5) :502–520, May 1997. 7, 30

[GL81] A. George and J. W. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice Hall Professional Technical Reference, 1981. 8

[GN87] Alan George and Esmond Ng. Symbolic factorization for sparse gaussian elimination with partial pivoting. *SIAM J. Sci. Stat. Comput.*, 8(6) :877–898, November 1987. 8

[Gup01] A. Gupta. Recent Progress in General Sparse Direct Solvers. In VassilN. Alexandrov, JackJ. Dongarra, BenjoeA. Juliano, RenéS. Renner, and C.J.Kenneth Tan, editors, *Computational Science — ICCS 2001*, volume 2073 of *Lecture Notes in Computer Science*, pages 823–831. Springer Berlin Heidelberg, 2001. 5

[Hac85] W. Hackbusch. *Multi-grid Methods and Applications*, volume 4. Springer-Verlag Berlin, 1985. 5

[Hac99] W. Hackbusch. A Sparse Matrix Arithmetic Based on H-Matrices. Part I : Introduction to H-Matrices. *Computing*, 62(2) :89–108, 1999. 44

[Hai08] A. Haidar. *On the Parallel Scalability of Hybrid Linear Solvers for Large 3D Problems*. Theses, Institut National Polytechnique de Toulouse - INPT, June 2008. 22

[Hén01] P. Hénon. *Distribution des Données et Régulation Statique des Calculs et des Communications pour la Résolution de Grands Systèmes Linéaires Creux par Méthode Directe*. PhD thesis, LaBRI, Université Bordeaux I, Talence, France, November 2001. 30

[HK00] B. Hendrickson and T. G. Kolda. Graph Partitioning Models for Parallel Computing. *Parallel Computing*, 26(12) :1519–1534, 2000. 9

[HKMR06] F. Hülsemann, M. Kowarschik, M. Mohr, and U. Rüde. Parallel Geometric Multigrid. In *Numerical Solution of Partial Differential Equations on Parallel Computers*, pages 165–208. Springer, 2006. 5

[HL95] B. Hendrickson and R. Leland. A Multi-Level Algorithm For Partitioning Graphs. In *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, pages 28–28, 1995. 49

[HNP91] M. T. Heath, E. Ng, and B. W. Peyton. Parallel Algorithms for Sparse Linear Systems. *SIAM Review*, 33(3) :pp. 420–460, 1991. 5, 30

[HP99] D. Hysom and A. Pothen. Efficient Parallel Computation of ILU(K) Preconditioners. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, New York, NY, USA, 1999. ACM. 17

[HR98] B. Hendrickson and E. Rothberg. Improving the Run Time and Quality of Nested Dissection Ordering. *SIAM Journal on Scientific Computing*, 20(2) :468–489, 1998. 46, 47, 59

[HRR02] P. Hénon, P. Ramet, and J. Roman. PaStiX : A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2) :301–321, January 2002. 7, 44

[HRR05] P. Hénon, P. Ramet, and J. Roman. On Using an Hybrid MPI-Thread Programming for the Implementation of a Parallel Sparse Direct Solver on a Network of SMP nodes. In *Proceedings of Sixth International Conference on Parallel Processing and Applied Mathematics, Workshop HPC Linear Algebra*, volume 3911 of *LNCS*, pages 1050–1057, Poznan, Poland, September 2005. Springer Verlag. 33

[HRR08] P. Hénon, P. Ramet, and J. Roman. On Finding Approximate Supernodes for an Efficient ILU(k) Factorization. *Parallel Computing*, 2008. 5

[KK95] G. Karypis and V. Kumar. Multilevel Graph Partitioning Schemes. In *Proc. 24th Intern. Conf. Par. Proc., III*, pages 113–122. CRC Press, 1995. 52

[KK98a] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1) :359–392, 1998. 49, 59

[KK98b] G. Karypis and V. Kumar. *A Software Package For Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*, 1998. 9, 45

[KK98c] G. Karypis and V. Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *Journal of Parallel and Distributed Computing*, 48 :96–129, 1998. 12, 46

[LD03] X. Li and J. W. Demmel. SuperLU DIST : A Scalable Distributed-memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Mathematical Software*, 29 :110–140, 2003. 7

[L'E12] J.-Y. L'Excellent. *Multifrontal Methods : Parallelism, Memory Usage and Numerical Aspects*. Habilitation à diriger des recherches, Ecole normale supérieure de lyon - ENS LYON, September 2012. 7, 30

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5(3) :308–323, September 1979. 8

[Liu90] J. W. H. Liu. The Role of Elimination Trees in Sparse Factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1) :134–172, 1990. 8

[Liu92] J. W. H. Liu. The Multifrontal Method for Sparse Matrix Solution : Theory and Practice. *SIAM Review*, 34(1) :pp. 82–109, 1992. 7, 30

[LRT79] R. J. Lipton, Donald J. Rose, and Robert Endre Tarjan. Generalized Nested Dissection. *SIAM Journal on Numerical Analysis*, 16(2), 1979. 46, 47, 59

[Man80] T. A. Manteuffel. An Incomplete Factorization Technique for Positive Definite Linear Systems. *Mathematics of Computation*, 34(150) :473–497, 1980. 5, 15

[MMS09] H. Meyerhenke, B. Monien, and T. Sauerwald. A New Diffusion-based Multilevel Algorithm for Computing Graph Partitions. *J. Parallel Distrib. Comput.*, 69(9) :750–761, September 2009. 59

[MTV91] G. L. Miller, S.-H. Teng, and S. A. Vavasis. A Unified Geometric Approach to Graph Separators. In *Proc. 31st Annual Symposium on Foundations of Computer Science*, pages 538–547, 1991. 62

[MV91] G. L. Miller and S. A. Vavasis. Density graphs and separators. In *Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 331–336, 1991. 46, 62

[NP93] E. G. Ng and B. W. Peyton. Block Sparse Cholesky Algorithms on Advanced Uniprocessor Computers. *SIAM Journal on Scientific Computing*, 14(5) :1034–1056, 1993. 31

[NR00] Esmond G. Ng and Padma Raghavan. Towards a scalable hybrid sparse solver. *Concurrency : Practice and Experience*, 12(2-3) :53–68, 2000. 5

[PR96] F. Pellegrini and J. Roman. Scotch : A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *High-Performance Computing and Networking*, volume 1067 of *Lecture Notes in Computer Science*, pages 493–498. Springer, 1996. 9, 45

[PR97] François Pellegrini and Jean Roman. Sparse matrix ordering with scotch. In B. Hertzberger and P. Sloot, editors, *High-Performance Computing and Networking*, volume 1225 of *Lecture Notes in Computer Science*, pages 370–378. Springer Berlin Heidelberg, 1997. 9

[PRA00] F. Pellegrini, J. Roman, and P. Amestoy. Hybridizing Nested Dissection and Halo Approximate Minimum Degree for Efficient Sparse Matrix Ordering. *Concurrency : Practice and Experience*, 12(2-3) :69–84, 2000. 9

[RBH12] S. Rajamanickam, E.G. Boman, and M.A. Heroux. ShyLU : A Hybrid-Hybrid Solver for Multicore Platforms. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 631–643, May 2012. 21, 28

[RTL76] D. J. Rose, R. E. Tarjan, and G. S. Lueker. Algorithmic Aspects of Vertex Elimination on Graphs. *SIAM Journal on Computing*, 5(2) :266–283, 1976. 8

[Saa94] Y. Saad. ILUT : A dual threshold incomplete LU factorization. *Numerical Linear Algebra with Applications*, 1(4) :387–402, 1994. 5, 17

[Saa03] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003. 5, 15

[SG04] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Journal of Future Generation Computer Systems*, 20 :475–487, 2004. 7

[SS86] Y. Saad and M. H. Schultz. GMRES : A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comput.*, 7(3) :856–869, July 1986. 5

[Tar75] Robert Endre Tarjan. Efficiency of a Good But Not Linear Set Union Algorithm. *J. ACM*, 22(2) :215–225, April 1975. 71

[TU08] Sivan Toledo and Anatoli Uchitel. A Supernodal Out-of-Core Sparse Gaussian-Elimination Method. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 4967 of *Lecture Notes in Computer Science*, pages 728–737. Springer Berlin Heidelberg, 2008. 44

[TW05] A. Toselli and O. B. Widlund. *Domain Decomposition Methods - Algorithms and Theory*. Springer Series in Computational Mathematics. Springer, 2005. 5

[Wid92] O. B. Widlund. Some Schwarz Methods For Symmetric And Nonsymmetric Elliptic Problems. In *Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations*. SIAM, 1992. 11

[YL11] I. Yamazaki and X. S. Li. On Techniques to Improve Robustness and Scalability of a Parallel Hybrid Linear Solver. In *High Performance Computing for Computational Science – VECPAR 2010*, volume 6449 of *Lecture Notes in Computer Science*, pages 421–434. Springer, 2011. 20, 26

[YLE11] I. Yamazaki, X. S. Li, and Ng E. PDSLin User Guide, March 2011. 20

Annexe A

Liste des publications

Congrès internationaux avec sélection et actes

[1] A. Casadei, P. Ramet, and J. Roman. *An improved recursive graph bipartitioning algorithm for well balanced domain decomposition.* 21st IEEE International Conference on High Performance Computing, Goa, India, décember 2014.

Congrès internationaux avec sélection

[2] A. Casadei and P. Ramet. *Memory Optimization to Build a Schur Complement.* In SIAM Conference on Applied Linear Algebra, Valence, Spain, June 2012.

Workshops internationaux avec sélection

[3] A. Casadei and P. Ramet. *Towards a recursive graph bipartitioning algorithm for well balanced domain decomposition.* In Mini-Symposium on Combinatorial Issues in Sparse Matrix Computation at ICIAM'15 conference, Pekin, China, August 2015.

[4] A. Casadei, P. Ramet, and J. Roman. *Towards a recursive graph bipartitioning algorithm for well balanced domain decomposition.* In Mini-Symposium on Partitioning for Complex Objectives at SIAM CSE'15 conference, Salt Lake City, USA, March 2015.

[5] A. Casadei, P. Ramet, and J. Roman. *Nested Dissection with Balanced Halo.* In SIAM Workshop on Combinatorial Scientific Computing, Lyon, France, July 2014.

[6] A. Casadei, L. Giraud, P. Ramet, and J. Roman. *Towards Domain Decomposition with Balanced Halo.* In Workshop Celebrating 40 Years of Nested Dissection, Waterloo, Canada, July 2013.

Rapports de recherche

[7] A. Casadei, P. Ramet, and J. Roman. *An improved recursive graph bipartitioning algorithm for well balanced domain decomposition.* Research Report RR-8582, August 2014.

[8] A. Casadei and P. Ramet. *Memory Optimization to Build a Schur Complement in an Hybrid Solver.* Research Report RR-7971, INRIA, 2012.