Titre: Title:	Stratégie de parallélisation hybride MPI/OPENMP pour EF, un programme d'analyse par éléments spectraux spécialisé pour la mécanique des fluides
Auteur: Author:	Guillaume Emond
Date:	2018
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Emond, G. (2018). Stratégie de parallélisation hybride MPI/OPENMP pour EF, un programme d'analyse par éléments spectraux spécialisé pour la mécanique des fluides [Mémoire de maîtrise, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/3703/

UNIVERSITÉ DE MONTRÉAL

STRATÉGIE DE PARALLÉLISATION HYBRIDE MPI/OPENMP POUR *EF*, UN PROGRAMME D'ANALYSE PAR ÉLÉMENTS SPECTRAUX SPÉCIALISÉ POUR LA MÉCANIQUE DES FLUIDES

GUILLAUME EMOND
DÉPARTEMENT DE GÉNIE MÉCANIQUE
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE MÉCANIQUE)
NOVEMBRE 2018

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

STRATÉGIE DE PARALLÉLISATION HYBRIDE MPI/OPENMP POUR *EF*, UN PROGRAMME D'ANALYSE PAR ÉLÉMENTS SPECTRAUX SPÉCIALISÉ POUR LA MÉCANIQUE DES FLUIDES

présenté par : EMOND Guillaume
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. VÉTEL Jérôme, Doctorat, président
M. GARON André, Ph. D., membre et directeur de recherche
M. PELLETIER Dominique, Ph. D., membre et codirecteur de recherche
M. ÉTIENNE Stéphane, Doctorat, membre

DÉDICACE

À tous mes amis et mes proches

REMERCIEMENTS

Avant tout, je tiens à remercier André Garon pour tout le temps et les efforts qu'il dévoue à ses étudiants. Il est toujours disponible pour discuter des embûches que l'on rencontre dans nos projets, pour partager sa vision d'expert, ou simplement pour une minute éducative autour d'un café.

Je remercie également mon co-directeur, Dominique Pelletier, pour ses conseils avisés et son sens de l'humour mémorable.

Merci aux membres du jury, Jérôme Vétel et Stéphane Étienne, d'avoir pris le temps de lire et d'analyser mon travail.

Je remcercie le programme *FONCER* pour leur soutien.

Merci à Alain Robidoux et à l'équipe de *Calcul Canada* pour leur aides précieuses avec les environnements informatiques des différentes plateformes de calculs.

Une mention spéciale pour Adrien Moulin, car sans lui, cette opportunité qui m'a été offerte n'aurait probablement jamais eu lieu.

J'aimerais saluer mes collèges, Pascal Boulos, Arthur Bawin, Danika Couture-Peck et Félix Bérard pour leur euthousiasme quotidien. Partager ce local avec vous fût un réel plaisir !

À l'aboutissement de ce projet, je pense à ma famille qui m'ont supporté et encouragé depuis le début de ma scolarité.

Finalement, je remercie chaleureusement Stéphanie Bessette pour avoir été à mes cotés pendant tous ce temps, pour s'être montrée compréhensive pour toutes les soirées que j'ai travaillé et pour son soutien moral dans les moments les plus déspérés.

RÉSUMÉ

Le programme d'analyse par éléments spectraux, *EF*, est développé au sein de Polytechnique Montréal afin de résoudre des problèmes de dynamique des fluides et de transfert de chaleur. Bénéficiant actuellement d'une parallélisation multi-threads avec l'interface *OpenMP*, il est optimisé pour le calcul sur des architectures à mémoire partagée uniquement. Typiquement, ces derniers sont munis de processeurs ayant tout au plus quelques dizaines de cœurs. Ce facteur devient limitant pour la réalisation de simulations d'envergure, notamment pour des domaines tridimensionnels.

Ce mémoire présente une stratégie de parallélisation hybride MPI/OMP adaptée aux structures de données d'*EF*, afin de porter le programme sur des environnements à mémoire distribuée disposant de ressources significativement plus grandes. La méthode implémentée utilise également un troisième niveau de parallélisme avec l'ajout d'instructions SIMD. Typiquement, ce dernier permet d'accélérer la phase d'assemblage par un facteur entre 2 et 4.

La résolution du système linéaire repose sur l'usage de solveurs directs distribués afin de conserver la robustesse du programme. Les solveurs *CPardiso* et *Mumps* sont alors incorporés au programme et leurs performances ont été évaluées sur un problème de diffusion thermique simple. Pour les deux solveurs, les besoins en mémoire sont redistribués équitablement et permettent donc de traîter des problèmes dont la taille excède la capacité maximale d'un seul noeud de calcul. En terme de temps d'exécution, *CPardiso* offre une certaine accélération pour la phase d'analyse et de factorisation mais aucun gain n'est observé lors de la résolution. Pour *Mumps*, des mesures préliminaires suggèrent des accélérations plus significatives, même pour la phase de résolution. Toutefois, plus de tests devront être effectués avant de se prononcer définitivement sur ce solveur.

ABSTRACT

EF is a spectral elements analysis software developped by Polytechnique Montréal to solve problems mainly related to fluid dynamics and thermal diffusion. Currently parallelized with OpenMP interface, the software is optimized for shared memory architectures with typically a few dozen cores. This limits the performance for analysis of large scale simulations, particularly for three dimensional domains.

This work develops a MPI/OMP hybrid parallelization strategy customized for *EF*'s data structures in order to run the program on significantly larger resources with distributed memory architectures. OpenMP SIMD constructs are also used in the matrix assembling phase as a third level of parallelization. Typical result shows a speed-up factor between 2 and 4.

This strategy uses distributed solvers based on direct methods to maintain the software's reliability. *Cluster Pardiso* and *Mumps* solvers are integrated and their performances evaluated using simple thermal diffusion problem. For both solvers, results show that memory is equitably distributed within each process. Therefore, they are useful for the treatment of large scale problems. Concerning execution time of *CPardiso*, the analysis phase as well as factorization benefit from hybrid parallelization but no gain is obtained with this method in the solving phase. With *Mumps*, preliminary results suggest more important speed-ups. However, it has not been tested on large scale problems yet.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

API	Application Programming Interface
BDF	Backward Differential Formula
CPU	Central Processing Unit
CRS	Compressed Row Storage
CSR	Compressed Sparse Row
DCSR	Distributed Compressed Row Storage
CFD	Computational Fluid Dynamics
DDL	Degré De liberté
E/S	Entrée / Sortie
HPC	high Performance Computing
MEF	Méthode des éléments finis
MES	Méthode des éléments Spectraux
MIMD	Multiple Instructions Multiple Data
OMP	Open MultiProcessing
SE	Système d'Exploitation
SIMD	Single Instruction Multiple Data

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

Bien que les avancées technologiques dans le domaine de l'informatique offrent des plateformes de calcul avec plus de mémoire et des processeurs toujours plus puissants que la génération précédente, la résolution de problèmes d'envergure requiert des algorithmes parallèles d'autant plus efficaces. En effet, la fréquence nominale des processeurs ayant atteint un plateau, il est plus avantageux d'investir dans la conception de processeurs avec la plus grande densité de cœurs possible. Ainsi, un programme ne peut exploiter pleinement la puissance d'une architecture moderne que s'il est en mesure de prendre avantage de chacune de ses unités de calculs.

Au sein de l'École Polytechnique de Montréal, le laboratoire de dynamique des fluides (LADYF) développe et utilise des logiciels, implémentant la méthode des éléments finis (MEF), pour résoudre des problèmes de transfert thermique et de mécanique des fluides. Jusqu'à présent, les ressources informatiques disponibles, dotées au maximum de quelques dizaines de cœurs, limitent la complexité des simulations, parallélisées avec OpenMP (OMP), à des modèles 2D ou de petits problèmes 3D [2]. Puisque l'achat de plateformes plus performantes s'avère très dispendieux, une alternative plus raisonnable consiste à employer les serveurs de *Calcul Québec* et *Calcul Canada*, disponibles gratuitement pour les institutions de recherche. Ces grappes de calculs possèdent typiquement plusieurs milliers de cœurs. Néanmoins, contrairement aux ordinateurs du LADYF, les serveurs ont une mémoire distribuée. En d'autres mots, les processeurs sont répartis sur différentes plateformes, appelées *noeuds*, connectées par réseau. Pour une telle architecture, un algorithme parallèle doit utiliser un protocole de communication afin d'échanger de l'information d'un noeud à l'autre. Le standard MPI (Message Passing Interface) représente l'état de l'art dans le domaine.

Nous proposons, dans le cadre de ce mémoire, une stratégie de parallélisation hybride MPI/OMP adaptée aux besoins et aux structures de *EF*, un programme d'analyse par éléments spectraux. Ces éléments, utilisant des polynômes d'interpolation de degré élevé afin d'accélérer la convergence des calculs, contiennent beaucoup plus d'information que les éléments finis. Par conséquent, le programme est doté d'un troisième niveau de parallélisme basé sur des instructions vectorielles. Cette stratégie repose sur l'utilisation de méthodes directes, par l'intermédiaire de solveurs distribués, afin de conserver la même robustesse que la version séquentielle d'*EF*.

Plan du mémoire

Avant d'entrer dans le vif du sujet, les chapitres 2, 3 et 4 introduisent quelques notions de base utiles à la compréhension des prochains chapitres. Ensuite, les chapitres 5 et 6 énoncent les caractéristiques de *EF* et des algorithmes implémentés. Enfin, les résultats sont présentés aux chapitres 7 et 8.

Au chapitre 2, la méthode des éléments finis est brièvement présentée d'un point de vue algorithmique, c'est-à-dire qu'on s'attarde principalement aux structures de données et à la mécanique pour les construire.

Le chapitre 3 introduit quelques notions de programmation parallèle. Premièrement, une description des types d'architecture informatique parallèle explicite un peu plus les notions de mémoire partagée et de mémoire distribuée. Ces deux architectures font naturellement apparaître les concepts de processus et de *threads*. À travers ces derniers, nous abordons les principes de mémoire virtuelle, de changement de contexte et de problèmes de concurrence. Les interfaces de programmation (API) MPI et OpenMP sont ensuite présentées. Puisqu'elles sont au cœur de ce projet, nous couvrirons plus en détails les fonctionnalités principales de chacune. Pour terminer, les différentes approches pour superposer les parallélisations MPI et OpenMP sont survolées.

Le 4^e chapitre reprend le contenu des deux précédents pour aborder plus spécifiquement la parallélisation de la méthode des éléments finis. On y définit les types de décomposition de domaine ainsi que les différents logiciels de partitionnement de graphes pour la résolution des systèmes linéaires.

La stratégie de parallélisation MPI est décrite au chapitre 5. Ce dernier commence par dresser un portrait du programme *EF*. Il présente ensuite les modifications apportées au logiciel, notamment par rapport au maillage, à la numérotation des degrés de liberté (DDLs) et à la construction d'un système d'équations global.

Au chapitre 6, on retrouve deux modifications n'étant pas directement reliées à la parallélisation MPI. La première aborde un troisième niveau de parallélisation, dans la phase d'assemblage, grâce à l'ajout d'instructions vectorielles. La seconde décrit un module gérant l'utilisation de plusieurs solveurs linéaires et des structures matricielles associées.

Le 7^e chapitre renferme les différentes études utilisées pour vérifier la bonne implémentation de la nouvelle version d'*EF*.

L'étude des performances du programme et des différents solveurs supportés est réalisée au chapitre 8.

CHAPITRE 2 CONCEPTS DE BASE EN ÉLÉMENTS-FINIS

Puisque l'implantation d'un paradigme de parallélisation hybride MPI/OMP requiert de modifier le cœur du logiciel *EF*, des notions de base en éléments finis sont requises à la compréhension des prochains chapitres. En fait, seulement une description des structures de données principales et de leurs mécaniques est nécessaire. Des approches plus formelles et plus complètes de la méthode des éléments finis (MEF) sont disponibles dans plusieurs ouvrages reconnus [3, 4]. Il en est de même pour la méthode des éléments spectraux (MES) [5, 6].

Brièvement, la méthode des éléments finis est une technique de résolution d'EDP (Équations aux Dérivées Partielles), particulièrement appropriée pour des problèmes définis sur des géométries complexes. En effet, elle consiste à diviser un domaine de forme quelconque, en de petits éléments de forme régulière, sur lesquels il est facile d'imposer les EDPs. Toutefois, les systèmes d'équations élémentaires, étant couplés avec les systèmes des éléments voisins, ne peuvent être résolus indépendamment. Ils sont alors combinés à l'intérieur d'un système linéaire global. La solution est finalement obtenue par la résolution de ce système.

2.1 Le maillage

Le maillage est à la base de la méthode des éléments finis. Il représente la discrétisation spatiale du domaine géométrique en cellules contiguës. Il existe plusieurs formats de structures de données permettant de décrire un maillage, mais, en général, on y retrouve les deux objets suivants :

Les nœuds : Un nœud est simplement un point dans l'espace, mais il représente également l'entité insécable avec laquelle les autres objets sont définis. Les coordonnées de tous les nœuds présents dans le maillage sont répertoriées dans un tableau.

Les éléments : Les éléments sont des sous-intervalles du domaine géométrique. Ils prennent généralement des formes simples telles que des triangles pour un domaine 2D ou des tétraèdres en 3D. Un élément K est généralement défini par une connectivé, soit un ensemble de n_g^K noeuds géométriques. Par exemple, pour un élément triangulaire ($n_g^K = 3$), les nœuds $\{\mathbf{x}_1^K, \mathbf{x}_2^K, \mathbf{x}_3^K\}$ représentent les sommets du triangle. Un maillage contient alors un tableau de connectivité des éléments où chaque ligne contient la connectivité d'un élément.

Un élément en D dimensions peut être défini par une hiérarchie de sous-éléments k^d,

où $0 \leq d < D$. Par exemple, un tétraèdre serait alors défini par ses faces, qui seraient à leur tour définies par leurs arêtes délimitées par deux points. Somme toute, la forme exacte de l'élément est dictée par la position des nœuds qui le composent.

2.2 La table de numérotation

Pour un élément K, on pose un nombre n_c^K de nœuds de calcul positionnés aux coordonnées $\tilde{x}_i^K$ pour $i = 1, 2, ..., n_c^K$. Ces nœuds sont associés aux degrés de libertés (DDL) du problème. Un DDL est un coefficient utilisé avec les polynômes d'interpolation afin de reconstruire la solution. Pour chaque nœud de calcul, il peut y avoir au plus un DDL par variable, pour un total de n_d^K DDLs par élément. Il est donc nécessaire de conserver en mémoire une table de numérotation, de dimension n_c par n_v (le nombre total de nœuds de calcul et le nombre de variables) indiquant pour chaque nœud, les numéros des DDLs associés.

Les degrés de liberté sont répartis en deux groupes : les DDLs inconnus et les DDLs connus. Un DDL est dit connu lorsque sa valeur est spécifiée par le problème, généralement par l'imposition d'une condition frontière essentielle. En prévision de l'étape d'assemblage du système global, il est pratique de numéroter les DDLs inconnus en premier et les connus ensuite. La table de numérotation est alors de la forme :

$$NUMER = \begin{bmatrix} \mathbf{U_I^1} & \mathbf{U_I^2} & \cdots & \mathbf{U_I^{n_v}} \\ \mathbf{U_C^1} & \mathbf{U_C^2} & \cdots & \mathbf{U_C^{n_v}} \end{bmatrix} = \begin{bmatrix} u_1^1 & u_1^2 & \cdots & u_1^{n_v} \\ u_2^1 & u_2^2 & \cdots & u_2^{n_v} \\ \vdots & \vdots & \ddots & \vdots \\ u_{n_c}^1 & u_{n_c}^2 & \cdots & u_{n_c}^{n_v} \end{bmatrix} \tag{2.1}$$

Une telle numérotation s'effectue en trois étapes :

1. Pour tous les nœuds, on compte les degrés de liberté associés à chaque variable.
2. Le tableau est parcouru, ligne par ligne, en numérotant les DDLs inconnus seulement.
3. On procède à une deuxième passe pour numéroter les DDLs connus.

2.3 Le système global

Sans perte de généralité, considérons un problème variationnel de la forme $a(u, w) = l(w)$, où a et l sont les formes bilinéaire et linéaire. u est la solution du problème et w est une fonction appartenant à l'ensemble des variations admissibles W (ou fonctions tests). La MEF procède à une discrétisation spatiale du domaine géométrique, afin de traiter le problème élément

par élément. La solution des équations différentielles est également discrétisée. En effet, la solution est approximée, sur chaque élément, par un ensemble de fonctions d'interpolation linéairement indépendantes ψ_j^K, $j = 1, 2, \ldots, n_d^K$ telle que

$$u^K(x) \simeq u^{n_d}(x)|_K = \sum_{j=1}^{n_d^K} u_j \psi_j^K(x). \tag{2.2}$$

Le problème peut donc se réécrire

$$\sum_{j=1}^{n_d^K} u_j a\left(\psi_j^K(x), w\right) = l\left(w\right). \tag{2.3}$$

Puisque cette équation est vraie pour tout $w \in W$, on peut former un système linéaire élémentaire en prenant un ensemble de fonctions ϕ_i^K, $i = 1, 2, \ldots, n_d^K$

$$\begin{bmatrix} a(\psi_1^K, \phi_1^K) & a(\psi_2^K, \phi_1^K) & \cdots & a(\psi_{n_d^K}^K, \phi_1^K) \\ a(\psi_1^K, \phi_2^K) & a(\psi_2^K, \phi_2^K) & \cdots & a(\psi_{n_d^K}^K, \phi_2^K) \\ \vdots & \vdots & \ddots & \vdots \\ a(\psi_1^K, \phi_{n_d^K}^K) & a(\psi_2^K, \phi_{n_d^K}^K) & \cdots & a(\psi_{n_d^K}^K), \phi_{n_d^K}^K) \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_{n_d^K} \end{bmatrix} = \begin{bmatrix} l(\phi_1^K) \\ l(\phi_2^K) \\ \vdots \\ l(\phi_{n_d^K}^K) \end{bmatrix} \tag{2.4}$$

Chacun des interpolants est construit avec la propriété de l'équation (2.5) de manière à ce que le support compact des ϕ_i^K est restreint à K si $\tilde{x}_i^K$ est à l'intérieur de l'élément. Dans le cas où x_i^K est situé à la frontière ∂K, le support s'étend également aux éléments adjacents à x_i^K.

$$\phi_i^K(\tilde{x}_j^K) = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases} \tag{2.5}$$

Dans le second cas, l'équation globale associée à ϕ_i est composée de termes provenant de tous les éléments de son support. On doit donc procéder à l'assemblage du système linéaire global.

Tout d'abord, on doit définir les vecteurs d'adressage qui indiquent, pour un élément a_{ij} de la matrice élémentaire, à quelle ligne et à quelle colonne l'insertion dans la matrice globale doit se faire. Ces vecteurs sont créés à partir de la table de connectivité et de numérotation des DDLs. La première nous donne les nœuds qui composent l'élément en question et la deuxième retourne, pour chaque nœud, le numéro du DDL associé, soit l'indice de la ligne ou de la colonne dans la matrice globale. On peut alors boucler sur les éléments du problème et

additionner leurs contributions. La figure 2.1 schématise l'insertion d'un système élémentaire dans le système global.

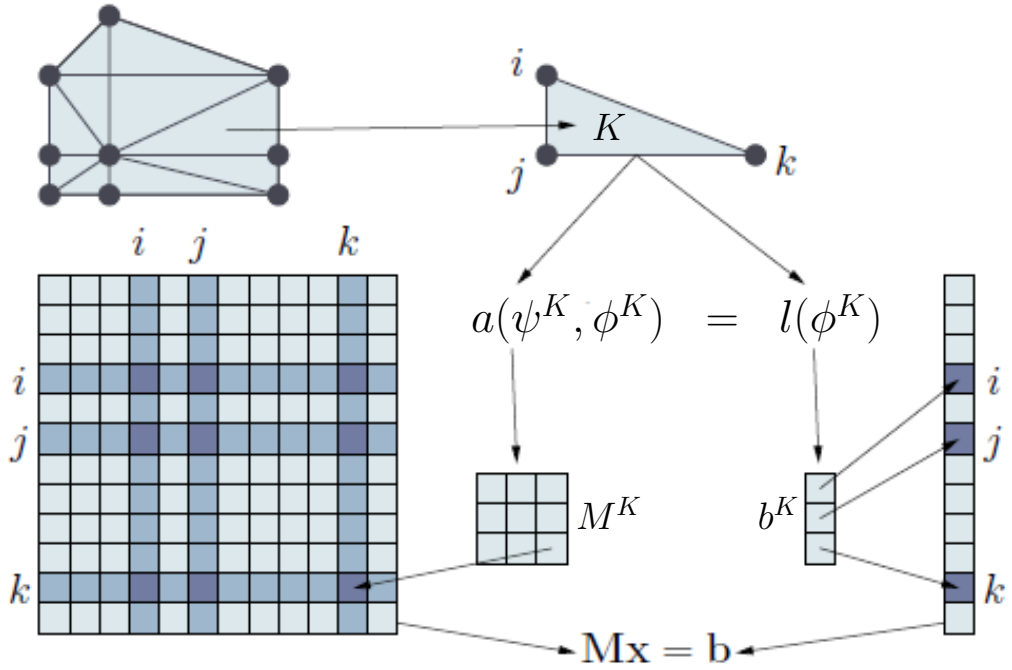


Figure 2.1 Assemblage du système linéaire global

Avec une numérotation des DDLs telle que présentée plus haut, l'assemblage produit un système par blocs (voir équation (2.6)). En effet, on peut identifier un système pour les DDLs inconnus uniquement et un autre pour les variables secondaires B_I. Finalement, la solution est obtenue par la résolution du système (2.7).

$$\begin{bmatrix} A_{II} & A_{IC} \\ A_{CI} & A_{CC} \end{bmatrix} \begin{bmatrix} U_I \\ U_C \end{bmatrix} = \begin{bmatrix} B_C \\ B_I \end{bmatrix} \tag{2.6}$$

$$\begin{aligned} A_{II}U_I &= B_C - A_{IC}U_C \\ B_I &= A_{CI}U_I + A_{CC}U_C \end{aligned} \tag{2.7}$$

Au final, la gestion du maillage, la numérotation des DDLs et la construction du système linéaire constituent les fondations d'un programme d'analyse par éléments finis. Elles seront donc des sujets récurrents dans la parallélisation d'*EF*.

CHAPITRE 3 CONCEPTS DE BASE EN PROGRAMMATION PARALLÈLE

La programmation parallèle consiste à écrire et concevoir des programmes pouvant traiter plusieurs données simultanément. Cette parallélisation peut se faire par le biais de plusieurs mécanismes et ce chapitre consiste à résumer brièvement certains d'entre eux. Les prochaines sections feront plus particulièrement référence aux interfaces de programmation (API) telles que OpenMP et MPI.

3.1 Les architectures parallèles

La programmation parallèle permet de tirer avantage des architectures informatiques parallèles. Cela signifie qu'il doit y avoir une relation très étroite entre le matériel et le logiciel afin d'obtenir des accélérations optimales. Cette courte section s'attarde donc à définir quelques modèles d'architecture.

La taxonomie de Flynn

La taxonomie de Flynn [7], énoncée pour la première fois par l'auteur du même nom, permet de catégoriser les architectures parallèles en 4 classes selon le nombre de flux d'instructions et de jeux de données utilisés. Ces dernières sont :

— SISD (Single Instruction Single Data) :
Cette catégorie correspond à une opération purement séquentielle où une seule donnée est traitée à la fois par une seule unité d'exécution. Les premiers ordinateurs domestiques étaient de type SISD.

— SIMD (Single Instruction Multiple Data) :
Ce type de mécanisme exploite le parallélisme au niveau des données, c'est-à-dire que la même opération (instruction) est appliquée sur plusieurs données simultanément. Le meilleur exemple de SIMD est le calcul sur un processeur vectoriel. Bien que les machines vectorielles ne soient plus populaires aujourd'hui, tous les processeurs actuels conservent des petites unités de calcul SIMD.

— MISD (Multiple Instruction Single Data) :
Les architectures MISD sont généralement utilisées pour la détection de défaillances d'un système. Elles sont plutôt rares, car le SIMD et MIMD se sont avérés plus adéquats pour la majorité des applications parallèles.

— MIMD (Multiple Instruction Multiple Data) :
Le parallélisme MIMD est effectué sur des architectures possédant des processeurs indépendants. Chaque processeur, ayant une file d'instructions, peut effectuer des opérations sur ses données respectives.

Deux modèles de répartition de la mémoire

Les architectures MIMD sont, et de loin, les plus répandues. La grande variété d'applications a mené à l'élaboration de plusieurs sous-types d'architecture. Ces derniers sont principalement départagés par leur répartition de la mémoire centrale. En fait, les ordinateurs parallèles sont à mémoire partagée ou à mémoire distribuée [8].

La mémoire est dite partagée lorsque tous les processeurs sont connectés, généralement par un bus, à une mémoire centrale globale. Ce modèle permet de programmer facilement une application parallèle avec un espace d'adressage unique. Toutefois, le nombre de processeurs disponibles sur des ordinateurs à mémoire partagée est limité à quelques dizaines dû à la bande passante du bus. De plus, les processeurs ont également un petit espace mémoire privé appelé cache. L'accès à cette cache ne nécessite pas l'usage du bus et est, par conséquent, beaucoup plus rapide qu'un accès à la mémoire centrale. Toutefois, lorsqu'une ligne de données se retrouve sur plus d'une cache en même temps et se voit modifiée sur l'une des caches, le système d'exploitation doit invalider la ligne sur les autres processeurs et la remplacer par sa nouvelle valeur. Pour ce faire, un protocole de cohérence de cache doit procéder à des communications inter-processeurs. Sans ce protocole, le programme n'aurait pas une vision cohérente de sa mémoire et mènerait à un résultat erroné. Ces communications peuvent devenir coûteuses si beaucoup de processeurs modifient des données très localisées. Ce type d'architecture n'est donc pas bien adapté pour les applications massivement parallèles.

Les architectures à mémoire distribuée sont généralement référencées sous les termes de serveurs, grappes de calculs ou encore de super-ordinateur dans le cas où les composantes sont optimisées pour le calcul haute performance (HPC). Elles sont constituées de plusieurs ordinateurs, ou nœuds, connectés par un réseau où chacun possède une mémoire centrale. Dans cette configuration, la bande passante du bus est réservée pour les processeurs présents sur ce nœud puisque les communications inter-nodales se transmettent via le réseau. Le faible couplage de ce type d'architecture permet d'assembler des serveurs à plusieurs centaines de nœuds. Toutefois, la latence du réseau est plus grande que celle du bus [8]. Il y a donc plus de pression sur l'application à minimiser ses communications inter-nodales.

3.2 Les Processus

D'un point de vue logiciel, la parallélisation MIMD se traduit par l'utilisation de plusieurs processus. Un processus est une abstraction du système d'exploitation (SE) permettant d'isoler un programme en exécution. L'objectif est de donner l'illusion que ce programme est le seul à être exécuté sur le système. Il aura alors l'impression d'avoir un accès exclusif aux ressources comme les processeurs, la mémoire centrale, le disque dur, etc. Un programmeur n'a donc pas besoin de se soucier de l'interaction de son programme avec les autres processus ; c'est le système d'exploitation qui s'en chargera à l'aide des concepts de mémoire virtuelle et de commutation de contextes.

La mémoire virtuelle

La mémoire, au niveau logiciel, est organisée en un vecteur continu de cellules de la taille d'un octet. Chaque cellule est associée à une adresse (un entier positif). L'ensemble des adresses, permettant d'accéder à n'importe quelle cellule de la mémoire, est appelé espace d'adressage. Une mémoire physique de M octets possède donc un espace de M adresses.

Dans les systèmes d'exploitation actuels, les processus utilisent d'autres espaces d'adressage dits virtuels. Chaque processus possède son propre espace et ces derniers ne contiennent que les informations qui définissent le programme. Ce mécanisme permet donc d'empêcher un processus de corrompre la mémoire utilisée par d'autres processus ou pire encore, par le système d'exploitation. Lorsqu'un programme accède à une donnée en mémoire centrale, une unité de gestion de mémoire (MMU) traduit l'adresse virtuelle en adresse physique.

Un autre avantage de cette mémoire virtuelle est qu'elle n'est pas limitée par la taille de mémoire vive, mais seulement par la longueur de l'adresse, c'est-à-dire que, selon l'architecture (32 ou 64 bits), l'espace d'adressage virtuel a une taille de 2^{32} ou 2^{64}, ce qui est largement supérieur à la mémoire physique. Si le programme est plus grand que la mémoire disponible physiquement, l'information est sauvegardée sur le disque et seulement les parties utilisées sont chargées en mémoire. Cette utilisation de la mémoire comme cache pour le disque permet une gestion efficace de l'espace limité en mémoire.

Les espaces virtuels sont organisés de façon à donner aux processus une vision uniforme et exclusive de la mémoire. La figure 3.1 illustre les différents éléments en mémoire d'un processus.

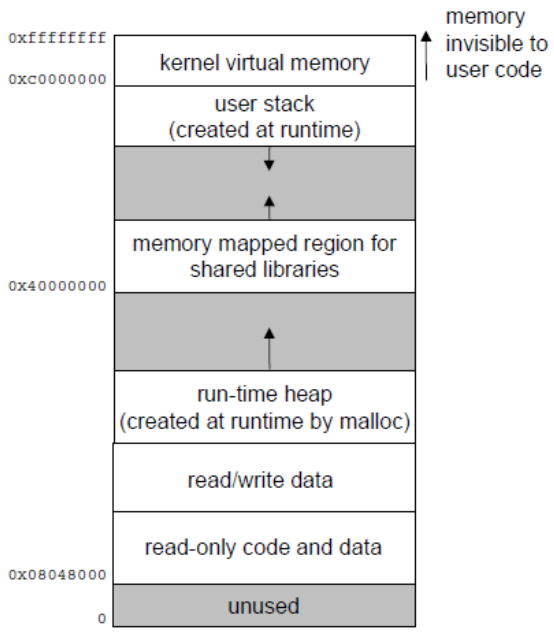


Figure 3.1 Division de la mémoire virtuelle (tiré de l'ouvrage de Bryant et O'Hallaron [9])

Commutation de contexte

Le nombre de processus en exécution sur une machine est généralement plus grand que le nombre de processeurs disponibles. Le système d'exploitation possède un mécanisme, appelé changement de contexte, qui permet à différents processus de s'exécuter concurremment, c'est-à-dire que leurs instructions sont entrelacées et s'exécutent à tour de rôle sur un même CPU (Central Processing Unit).

En effet, seulement un processus peut avoir le contrôle d'un CPU à la fois. Lorsque ce contrôle passe à un autre processus, le système d'exploitation (Kernel) enregistre préalablement les informations nécessaires pour relancer ultérieurement ce processus dans le même état, comme si ce dernier n'avait jamais été interrompu. Ces informations forment le contexte du processus et sont enregistrées dans le bloc de contrôle de processus (PCB). La figure 3.2 permet de visualiser le déroulement d'une commutation de contexte entre un programme A et B.

Encore une fois, le système d'exploitation possède un mécanisme d'ordonnancement qui gère l'ordre dans lequel les processus sont exécutés ainsi que le temps CPU alloué à chacun. Le fait que ces interruptions ne sont pas opérées au niveau utilisateur a d'importantes répercussions sur le développement d'un programme MIMD :

— Le temps d'exécution d'un programme n'est pas fixe ni reproductible.

— On ne peut supposer que les processus sont synchronisés. La synchronisation doit être

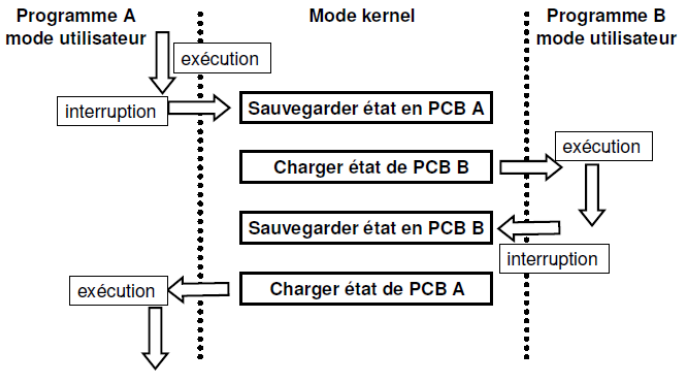


Figure 3.2 Changement de contexte (tiré de l'ouvrage de Boucheneb et Tores [10])

assurée par des moyens externes.

— Les changements de contexte, lorsque nombreux, peuvent mener à une baisse de performance considérable.

La dernière considération est particulièrement importante. En analyse numérique, où le nombre de processus est contrôlé par l'utilisateur, on choisit généralement d'imposer un seul processus par CPU (ou parfois deux si le processeur est muni de technologie *hyperthreads*). Il peut également s'avérer utile d'attacher un processus à un CPU en particulier. Cette technique de *CPU pinning* permet à un processus, après un changement de contexte, de réutiliser des informations en mémoire qui lui appartiennent et qui n'auraient pas été écrasées par les autres processus depuis son dernier temps CPU.

3.2.1 Synchronisation et communication

Si les processus sont asynchrones et ont une mémoire complètement indépendante les unes des autres, comment fait-on pour paralléliser un programme avec ces derniers ? La réponse se trouve dans les communications inter-processus. Au même titre que des communications étaient nécessaires pour maintenir une cohérence de la cache dans les architectures à mémoire partagée, un programme distribué communique entre ses processus pour obtenir une certaine cohérence entre les mémoires virtuelles. La seule différence est que ces communications ne sont pas gérées par le système d'exploitation, mais plutôt par le programmeur. Heureusement, des bibliothèques de messagerie sont disponibles en accès libre et les plus répandues suivent le standard MPI (Message Passing Interface).

Les éléments importants de ce standard, utilisé dans le programme *EF*, sont présentés ci-dessous. Toutes les informations subséquentes sont tirées de la documentation MPI officielle [1].

Communication "Point-to-Point"

Les informations sont transférées d'un processus à l'autre par l'intermédiaire d'un message. Tout comme une lettre postale, un message est composé d'une enveloppe ainsi que des informations qu'elle contient. Afin que le message soit valide, l'enveloppe doit définir l'identificateur (rang) de la source et du destinataire, un communicateur et optionnellement une étiquette. Les données sont généralement envoyées sous la forme d'un vecteur contigu en mémoire. Ce dernier est donc défini par l'adresse du premier élément, le nombre d'éléments ainsi que leur taille.

Le communicateur est un concept important de l'environnement MPI. Il représente, pour un groupe de processus, le contexte de communication des messages échangés entre ces processus. Un processus peut avoir accès à plusieurs communicateurs différents et les communications effectuées sur l'un n'interfèrent pas avec les autres. Dans le cadre de notre comparaison avec le courrier, le communicateur prendrait la forme de la compagnie de transport. Ce dernier est au courant de tous les messages échangés dans une certaine région, mais ne possède pas d'information sur les colis transportés par d'autres compagnies. Toutefois, cette analogie n'est pas parfaite, car le rang (adresses) des processus est défini à l'intérieur d'un communicateur. Un processus peut donc avoir plusieurs rangs avec des valeurs différentes.

Concrètement, un message est échangé par l'appel à deux fonctions ; le processus émetteur doit appeler la fonction *envoyer (send)* et le destinataire, la fonction *recevoir (receive)*.

```
MPI_SEND(buf, count, datatype, dest, tag, comm)
MPI_RECV(buf, count, datatype, source, tag, comm, status)
```

L'argument supplémentaire de MPI_RECV, status, contient des informations sur l'état de la communication lors de la réception du message. Pour qu'un message soit capté par le receveur, il est nécessaire que les enveloppes à l'envoi et à la réception soient identiques.

Sémantiques et modes de communication

Il existe deux sémantiques de communication : bloquante et non bloquante. Pour chacune, il est possible de choisir l'un des trois modes de communication : *synchronous*, *ready* ou *buffered*. À des fins de simplicité, nous ne considérerons ici que le mode synchrone.

L'envoi d'un message bloquant, avec un appel à MPI_SEND par exemple, ne se termine qu'après que les données soient copiées sur le processus destinataire. Une communication bloquante entraîne donc une synchronisation des processus. Dans un échange complexe de messages entre plusieurs processus, il est possible que ce type de communication mène à un interblocage. Reprenons l'exemple 3.8 de la documentation MPI où deux processus tentent

de s'envoyer un message chacun.

```
IF (rank == 0) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 1, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 1, tag, comm, ierr)
ELSE IF (rank == 1) THEN
    CALL MPI_RECV(recvbuf, count, MPI_REAL, 0, tag, comm, status, ierr)
    CALL MPI_SEND(sendbuf, count, MPI_REAL, 0, tag, comm, ierr)
END IF
```

Cette implémentation échouera, car les opérations de réception ne peuvent se terminer tant que leur message respectif n'est pas envoyé depuis l'autre processus. Or, les messages ne peuvent être envoyés tant que les opérations de réception ne sont pas terminées. Les deux processus se retrouvent donc dans une situation d'interblocage et resteront en attente jusqu'à ce qu'ils reçoivent un signal de terminaison (SIGKILL).

Les communications non bloquantes permettent de se libérer de la synchronisation des processus implicite lors des appels aux fonctions MPI. Pour cela, l'envoi et la réception d'un message se font chacun en deux appels de fonction. Le premier sert à débuter l'opération, mais il peut se terminer avant que les données soient copiées. Ces fonctions sont dites non bloquantes. Le deuxième appel sert à compléter l'opération et est par définition bloquant. Un processus peut donc effectuer des calculs entre ces deux fonctions et élimine donc les coûts de synchronisation.

Les appels pour amorcer les communications portent les mêmes noms que leurs équivalents bloquants avec le préfixe I (MPI_Isend). La complétion se fait avec la fonction MPI_Wait.

Communication collective

Selon la relation entre les processus, il peut rapidement devenir fastidieux de programmer tous les envois de message un par un. Le standard MPI met en place un moyen pour que tout le groupe de processus d'un communicateur puisse se transmettre des messages. Ce sont les communications collectives. Ces routines sont généralement optimisées pour minimiser le nombre de messages utilisés pour accomplir une certaine opération. Ces communications sont donc le parfait mélange de performance et de simplicité.

Les opérations collectives les plus communes sont représentées à la figure 3.3 tirée encore une fois de la documentation officielle MPI.

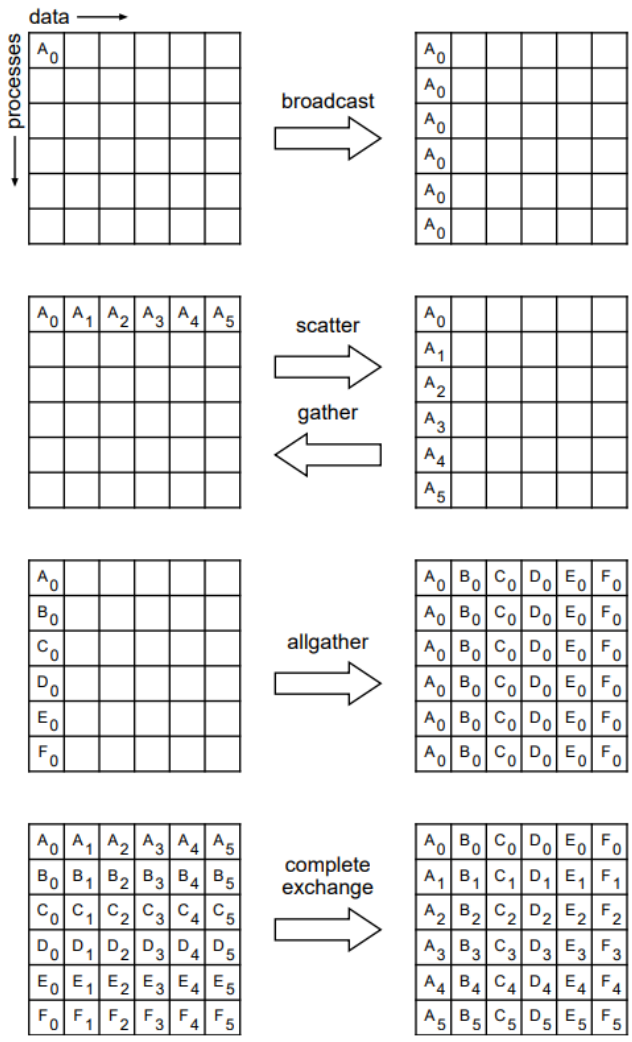


Figure 3.3 Schématisation des opérations collectives MPI [1]

Les colonnes de gauche et droite représentent respectivement l'état de la zone mémoire à l'envoi et à la réception.

On y retrouve les fonctions : MPI_Bcast, MPI_Scatter, MPI_Gather, MPI_Allgather et MPI_AlltoAll (complete exchange). Par défaut, les segments de données sont tous de la même taille ce qui restreint leur utilité. Toutefois, il existe des versions vectorielles de ces fonctions, portant le suffixe v, ayant des messages de longueurs variables. De plus, ces communications sont également offertes dans une sémantique non bloquante.

L'optimisation de ces routines n'est pas dictée par le standard et est laissée à la discrétion du distributeur de la bibliothèque MPI. Toutefois, on peut s'attendre à ce qu'une bonne implémentation utilise des arbres de communications équilibrés.

Opération de réduction

Le standard offre des fonctions globales de réduction qui permettent de mélanger communications et opérations arithmétiques. Par exemple, calculer la somme des éléments d'un vecteur distribué nécessiterait de calculer la somme des éléments locaux sur chaque processus, de rassembler ces sous-sommes sur un seul processus et ensuite, de les additionner pour obtenir le véritable total. Tout ceci peut être remplacé par une opération de réduction où l'addition des sommes est optimisée selon le schéma de communication utilisé.

Des opérations de réduction partielle, appelées *Scan* sont également disponibles. Dans l'exemple précédent, une opération de *Scan* renverrait, à chaque processus i la somme partielle des éléments détenus par les processus de rang i et inférieur. Une opération *Exscan*, pour *Exclude*, produit le même résultat, mais sans comptabiliser les éléments du processus courant i.

Topologie virtuelle

Néanmoins, il y a un point négatif aux communications collectives présentées jusqu'à présent ; un processus doit partager de l'information avec tout le groupe alors que, dans plusieurs applications, il est suffisant de communiquer qu'avec une petite fraction d'entre eux. Lorsque le groupe de processus devient imposant, un algorithme peut souffrir du nombre de communications superflues. Pour remédier à ce problème, il est possible d'imposer une topologie virtuelle sur le communicateur. Cette topologie, sous la forme d'un graphe orienté, définit les relations entre les processus. Deux processus, représentés par des nœuds du graphe, sont voisins si une arête les relit. Le sens de l'arc définit le sens des communications possibles entre les processus, c'est-à-dire qu'un processus peut recevoir des messages de son voisin sans pouvoir lui répondre ou vice versa. Cette topologie ne correspond pas à la répartition des processus sur les processeurs, mais une implémentation MPI peut s'en servir pour optimiser cette répartition afin de réduire le coût de communication. Les fonctions permettant une communication entre voisins ne sont définies que pour les opérations *AlltoAll* et *Allgather*. Ces dernières ont le préfixe *_Neighbor_*.

Entrée-Sortie E/S

La parallélisation d'une application avec MPI permet de traiter une quantité de données importante. Il s'ensuit que les phases d'écriture et de lecture de fichiers seront d'autant plus sollicitées. MPI fournit donc une interface pour partitionner les données des fichiers et des routines E/S parallèles optimisées.

Partitionnnement des fichiers

Rappelons-nous que les processus, pour ne pas être en conflits les uns avec les autres, utilisent

une vision de la mémoire indépendante nommée *mémoire virtuelle*. Le principe derrière les opérations E/S parallèles est identique. L'objectif est de donner, à un processus, une vue exclusive d'un fichier qui est, en réalité, partagée par tout le groupe. Cette vue est définie par des types de données, au sens MPI, représentant un agencement de blocs mémoires où certains de ces blocs peuvent être vides. Pour définir la vue d'un fichier, il est nécessaire de définir deux types de donnée :

— *etype* est l'élément de base des opérations E/S. Les indexes, déplacements et quantités sont donnés en nombre de *etype*.

— *filetype* est lui-même un agencement de *etypes*. Contrairement au premier type, *filetype* peut varier d'un processus à l'autre. Il permet donc de contrôler le partitionnement des données parmi les processus.

La vue d'un fichier est directement construite par la répétition contiguë du filetype. De plus, il est possible de décaler la vue par rapport au début du fichier afin d'omettre facilement les en-têtes.

Si les vues se chevauchent, l'accès aux données du fichier peut être concurrentiel et mener à des résultats erronés. La figure 3.4 illustre comment un choix judicieux de filetype produit des vues exclusives et complémentaires.

Figure 3.4 Partitionnement d'un fichier [1]

Manipulation des fichiers

Les opérations E/S MPI manipulent les fichiers à travers une interface. C'est dans cette dernière que la vue du fichier est configurée. De plus, cette interface définit un pointeur de fichier individuel pour chaque processus ainsi qu'un pointeur de fichier partagé. Les fonctions E/S peuvent donc accéder aux données à partir de l'un ou l'autre de ces pointeurs ou en mentionnant explicitement le décalage (en etype) depuis le début de la vue. Les différents modes d'accès n'interfèrent pas ensemble, c'est-à-dire qu'une méthode utilisant le pointeur de fichier individuel ne modifie pas le pointeur partagé et vice versa.

Les routines, pour chaque mode d'accès, possèdent des équivalents bloquants/non bloquants et individuels/collectifs. Les opérations collectives peuvent bénéficier de plus d'optimisations automatiques que leurs contreparties.

Lorsqu'un fichier est bien partitionné, les méthodes avec décalage explicite et pointeur individuel s'utilisent de façon similaire aux fonctions E/S *POSIX*. Toutefois, les méthodes utilisant le pointeur partagé nécessitent quelques précisions.

— Ce mode d'accès n'est valide que si tous les processus ont la même vue du fichier.

— L'ordre de lecture ou écriture, pour des opérations individuelles concurrentes, n'est pas déterminé. D'autres moyens doivent être employés pour assurer une bonne synchronisation.

— La version collective attribue les données par ordre de rang des processus. L'opération est donc équivalente à une lecture séquentielle où les processus lisent, l'un après l'autre, un segment contigu du fichier.

Notation

La représentation d'un programme distribué n'est pas toujours évidente, particulièrement lorsqu'il y a beaucoup d'interactions entre les processus. Une notation est alors instaurée afin de simplifier les algorithmes à venir. Les définitions suivantes expriment les concepts principaux du chapitre.

Définition 3.1 (Les groupes)
Un groupe de processus est déterminé par l'ensemble des rangs de ces derniers[1]*. Le symbole* $\mathcal{G}$ *est fréquement utilisé pour désigner un groupe. Certains groupes usuels sont spécialement identifiés :*

— $\mathcal{G}_{world}$ *: Tous les processus*[2]*.*

— $\mathcal{G}_{master}$ *: Uniquement le processus 0.*

— $\mathcal{G}_{slaves}$ *: Tous les processus à l'exception de 0.*

— $\mathcal{G}_{self}$ *: Uniquement le processus local.*

Définition 3.2 (Les communicateurs)
Soient les groupes $\mathcal{G}_1$ *et* $\mathcal{G}_2$*. Posons A comme un ensemble d'arcs définis par des couples*

1. En réalité, le rang d'un processus est défini selon le communicateur. Pour alléger la notation, sans perte de généralité, nous utilisons les rangs définis par le communicateur `MPI_COMM_WORLD`.
2. Nous utiliserons l'abréviation $\mathcal{G}_w$

$(i, j) \in \mathcal{G}_1 \mapsto \mathcal{G}_2$. *Un communicateur* $\mathcal{C}^A_{\mathcal{G}_1 \to \mathcal{G}_2}$ *représente le graphe orienté d'une topologie virtuelle où* A *est l'ensemble des communications possibles à l'intérieur de ce communicateur (voir figure 3.5).*

On définit le communicateur $\mathcal{C}_{world} \equiv \mathcal{C}^A_{\mathcal{G}_w \to \mathcal{G}_w}$ *tel que* $A = \mathcal{G}_w \times \mathcal{G}_w$

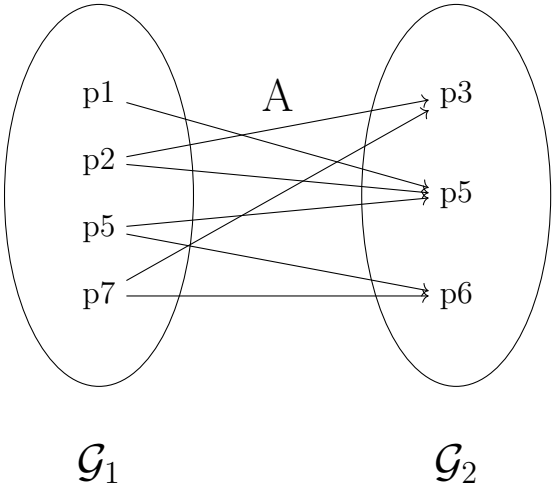


Figure 3.5 Exemple de topologie virtuelle définissant l'ensemble des communications d'un communicateur.

Définition 3.3 (Les espaces mémoires locaux)
La mémoire virtuelle d'un processus k *est représentée par l'espace* $\mathcal{M}^{(k)}$. *Ce dernier est dit* local *au processus* k. *Un objet quelconque* $x \in \mathcal{M}^{(k)}$ *est écrit* $x^{(k)}$.

Par défaut[3]*, les espaces mémoires sont toujours disjoints.*

$$\mathcal{M}^{(k)} \cap \mathcal{M}^{(p)} = \emptyset, \quad k \neq p$$

Définition 3.4 (Les espaces mémoires globaux)
Soient les processus i *et* j. *La combinaison de leurs espaces mémoire forment un espace global, ou espace distribué, tel que*

$$\mathcal{M}^{(i,j)} = \mathcal{M}^{(i)} \cup \mathcal{M}^{(j)}$$

Un objet x *distribué sur ces deux processus est noté* $x^{(i,j)}$ *et est défini par la relation suivante :*

$$x^{(i,j)} \equiv \left(x_i^{(i)}, x_j^{(j)}\right) \in \mathcal{M}^{(i)} \times \mathcal{M}^{(j)}$$

3. Sans prendre en compte les accès mémoire à distance (RMA)

Remarque 1. Dans un cadre général, les objets définis dans l'espace global d'un groupe de processus $\mathcal{G}$ sont notés :

$$\mathcal{M}^{(\mathcal{G})} = \bigcup_{i\in\mathcal{G}} \mathcal{M}^{(i)}$$

$$x^{(\mathcal{G})} \equiv \left(x_i^{(i)}\right)_{i\in\mathcal{G}} \in \bigtimes_{i\in\mathcal{G}} \mathcal{M}^{(i)}$$

Définition 3.5 (Les communications *point-to-point*)
Soit un objet $x^{(k)}$, l'effet d'une communication MPI de type point-to-point *peut être interprété comme une transformation bijective $T^{\mathcal{C}}_{k\to p} : \mathcal{M}^{(k)} \mapsto \mathcal{M}^{(p)}$ tel que*

$$x^{(p)} = T^{\mathcal{C}^A_{\mathcal{G}_1\to\mathcal{G}_2}}_{k\to p}\left(x^{(k)}\right), \quad k\in\mathcal{G}_1,\ p\in\mathcal{G}_2,\ (k,p)\in A$$

Remarque 2. Le contenu de x n'est pas modifié par la transformation.

Remarque 3. Puisque nous utilisons uniquement le communicateur $\mathcal{C}_{world}$ dans le cadre de ce mémoire, nous prélivégions la notation réduite T^p_k.

Définition 3.6 (Les communications collectives)
Soient les groupes $\mathcal{G}_1$ et $\mathcal{G}_2$. Les communications collectives sont des transformations qui distribuent un objet x résidant sur $\mathcal{M}^{(\mathcal{G}_1)}$ vers $\mathcal{M}^{(\mathcal{G}_2)}$.

$$T^{\mathcal{C}^A_{\mathcal{G}_1\to\mathcal{G}_2}}_{\mathcal{G}_1\to\mathcal{G}_2} : \mathcal{M}^{(\mathcal{G}_1)} \mapsto \mathcal{M}^{(\mathcal{G}_2)}$$

Remarque 4. Encore une fois, nous omettons d'écrire le communicateur afin d'alléger la notation. Ainsi, on note :

$$x^{(\mathcal{G}_2)} = T^{\mathcal{G}_2}_{\mathcal{G}_1}\left(x^{(\mathcal{G}_1)}\right)$$

Les transformations collectives les plus courrantes sont définies ici[4]

- $T^p_{scatter} = T^{\mathcal{G}}_p = \left[T^j_p\right]_{\forall j\in\mathcal{G}}$: $\left(\boldsymbol{x}_i^{(i)}\right)_{\forall i\in\mathcal{G}} = T^p_{scatter}\left(\left[\boldsymbol{x}_i\right]^{(p)}_{\forall i\in\mathcal{G}}\right)$
- $T^p_{gather} = T^p_{\mathcal{G}} = (T^p_i)_{\forall i\in\mathcal{G}}$: $\left[\boldsymbol{x}_i\right]^{(p)}_{\forall i\in\mathcal{G}} = T^p_{gather}\left(\left(\boldsymbol{x}_i^{(i)}\right)_{\forall i\in\mathcal{G}}\right)$
- $T_{all} = T^{\mathcal{G}}_{\mathcal{G}} = \left(\left[T^j_i\right]_{\forall j\in\mathcal{G}}\right)$: $\left(\left[\boldsymbol{x}^i_j\right]^{(i)}_{\forall j\in\mathcal{G}}\right)_{\forall i\in\mathcal{G}} = T_{all}\left(\left(\left[\boldsymbol{x}^j_i\right]^{(i)}_{\forall j\in\mathcal{G}}\right)_{\forall i\in\mathcal{G}}\right)$

Remarque 5. Puisque, dans ce mémoire, la grande majorité des structures sont distribuées sur $\mathcal{G}_w$ et que les algorithmes sont symétriques d'un processus à l'autre, on évite de décrire tous les objets comme un n-uplet $\left(\mathbf{x}_i^{(i)}\right)_{\forall i\in\mathcal{G}} \in \mathcal{M}^{(\mathcal{G}_w)}$. On utilise plutôt une notation exprimant le

4. voir l'annexe C pour obtenir une définition de la notation $[\cdot]$

point de vue d'un seul processus p. Par exemple, une communication de type *AllToAll* s'écrit de la façon suivante :

$$\left[\mathbf{x}_j^i\right]_{\forall j \in \mathcal{G}} = T_{all}\left(\left[\mathbf{x}_i^j\right]_{\forall j \in \mathcal{G}}\right)$$

Il est implicite que tous les objets sont locaux et que tous les autres processus exécutent exactement les mêmes opérations. Au final, on obtient une notation épurée et sans ambiguïté.

3.3 Les threads

La section 3.2 introduit la notion de processus et de son contexte. Bien que le contenu exact puisse varier légèrement d'un système d'exploitation (SE) à l'autre, un contexte est divisible en deux sous-catégories : le contexte noyau (ou kernel) et le contexte programme (figure 3.6a).

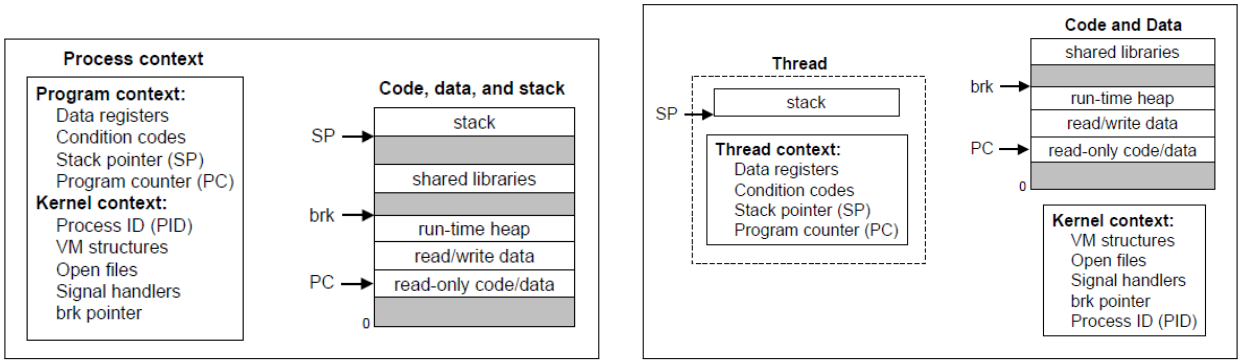
— Le contexte noyau décrit l'ensemble des ressources disponibles, soit des informations sur son espace d'adressage, les fichiers ouverts, les communications interprocessus, etc. Certaines informations nécessaires au SE s'y retrouvent également (PID, statut, priorité).

— Le contexte programme permet de spécifier où est rendu un programme dans son exécution. Il contient alors le contenu des registres, le compteur ordinal (pointeur vers les instructions) et un pointeur vers la pile d'exécution.

Il est possible de réorganiser les éléments de la figure 3.6a pour faire apparaître le concept de fil d'exécution, plus souvent appelé *thread*. La figure 3.6b montre qu'un processus est composé du contexte kernel, de l'espace d'adressage et d'un thread, lui-même constitué du contexte du programme et d'une pile d'exécution.

En fait, plusieurs threads peuvent être associés au même processus, utilisant ainsi un contexte kernel et un espace d'adressage commun. Ces threads, comme les processus, permettent d'accomplir différentes tâches simultanément.

Les threads sont une alternative très intéressante par rapport aux processus, car le partage des données ne nécessite pas de communication interprocessus. De plus, le changement de contexte entre threads est beaucoup plus rapide que pour les processus, car seulement le contexte du thread est modifié. Toutefois, l'accès au même espace mémoire peut mener à des problèmes de concurrence.



(a) contexte d'un processus (b) contexte d'un thread

Figure 3.6 Représentation d'un thread dans le contexte d'un processus

Ces schémas sont tirés de l'ouvrage de Bryant et O'Hallaron[9]

3.3.1 OpenMP

La popularité grandissante des programmes multithreads a encouragé le développement de plusieurs APIs afin d'obtenir une gestion de threads facile et efficace. Les plus répandues sont PThread(POSIX Thread), OMP (Open MultiProcessing) et TBB (Threading Building Blocks). De par sa simplicité et sa variété de fonctionnalités, l'interface OpenMP a été choisie pour la parallélisation du programme *EF* et sera donc couvert plus en détails dans cette section.

Modèle d'exécution

OpenMP utilise un modèle d'exécution *fork-join*, c'est-à-dire que des blocs parallèles sont insérés dans un programme séquentiel. Lorsque le thread initial, ou maître, rencontre une région parallèle, ce dernier crée une équipe de threads dont il fait lui-même partie. Le code à l'intérieur de la section parallèle définit les tâches qui seront attribuées à chacun d'eux. Lorsque tous les membres de l'équipe ont terminé leurs tâches, les threads sont supprimés à l'exception du thread maître qui poursuivra l'exécution du reste du programme. L'expression *fork-join* prend tout son sens lorsque le modèle est schématisé comme à la figure 3.7

Le modèle accepte également plusieurs niveaux de région parallèle imbriqués, accordant ainsi un contrôle plus fin de la charge de travail attribuée à chaque thread. Le nombre de threads créés dans un bloc parallèle est dicté par une variable de contrôle interne (ICV) [5]. Cette variable peut être dynamiquement modifiée par la routine `omp_set_num_threads()`.

5. *Internal Control Variable*

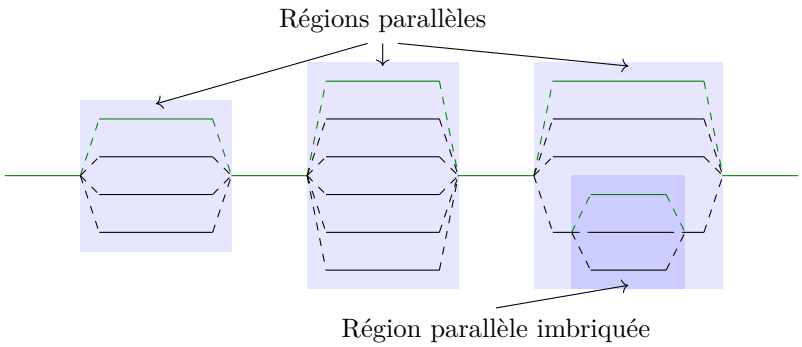


Figure 3.7 Modèle d'exécution *fork-join*

Concrètement, la création d'un bloc parallèle se fait avec la directive *parallel*. Cette dernière est appelée, en *C++*, de la manière suivante :

```
#pragma omp parallel [clause[ [, ]clause] ...] new-line
    structured-block
```

Certaines clauses permettent de contrôler l'environnement des sections parallèles. Par exemple, la clause `num_threads()` spécifie la taille de l'équipe pour cette région. L'affinité des threads est également contrôlée par la clause `proc_bind()`.

Le modèle d'exécution d'OpenMP ne requiert pas de restructuration majeure du programme. Seulement les sections critiques ont besoin d'être modifiées. Donc, cela en fait un moyen peu coûteux de paralléliser un logiciel.

Modèle de mémoire

OpenMP est basé sur un modèle de mémoire partagée. Toutefois, l'API offre également, à chaque thread, une interface de mémoire privée. Par défaut, un objet, existant en dehors d'un bloc associé à une directive OMP, a un accès, une fois à l'intérieur de ce bloc, partagé entre les membres de l'équipe. Il est possible de changer le type d'accès mémoire par défaut avec la clause `default()`.

Pour qu'une variable soit placée en mémoire privée, il est généralement nécessaire d'utiliser l'une des clauses d'attribut de partage de données. Ces dernières sont énumérées ci-dessous.

shared : Les variables sont explicitement déclarées comme partagées.

private : Lorsqu'un objet est déclaré privé, chaque thread crée un nouvel objet (du même type que l'original) et fait référence à ce dernier jusqu'à la fin de la directive OMP.

firstprivate : Les arguments sont privés et initialisés avec la valeur des objets originaux.

lastprivate : À la fin d'une directive OMP `for` ou `sections` (section 3.3.1), l'objet original reprend la valeur de la variable locale dans la dernière itération ou section.

Directives de partage des tâches

Dans une région parallèle, chaque thread de l'équipe est assigné à une tâche implicite correspondant au code à l'intérieur de cette région. Les threads exécutent donc les mêmes instructions. Afin d'implémenter une véritable parallélisation MIMD, d'autres directives sont nécessaires à l'intérieur d'un bloc parallèle pour introduire des variations entre les instructions des tâches implicites. Les directives de partage de tâches [6] permettent de distribuer les itérations d'une boucle ou d'assigner un segment de code à un thread en particulier.

Partage de boucle

La directive de boucle est l'une des plus utilisées. Elle consiste à distribuer les itérations d'une boucle de forme canonique parmi les threads existant dans la région parallèle. L'exécution de la boucle peut alors se faire en parallèle.

La définition d'une boucle canonique peut être résumée par les deux propriétés suivantes :

— Le nombre d'itérations est déterminé avant l'exécution de la boucle.

— Les itérations de la boucle sont indépendantes.

La syntaxe de la directive est la suivante :

```
#pragma omp for [clause[[,] clause] ... ] new-line
    for-loops
```

Cette directive possède quelques clauses particulières :

Avec la clause `collapse`, il est possible de paralléliser une imbrication de boucles canoniques. L'ensemble des itérations est réuni au sein d'un même espace d'itéré, comme s'il n'y avait qu'une seule boucle. Toutefois, les boucles doivent être parfaitement imbriquées, c'est-à-dire qu'il ne peut y avoir d'instructions dans les boucles de niveaux supérieurs, mais seulement dans la dernière boucle.

En réalité, les itérations d'une boucle sont distribuées par morceaux, soit un nombre non nul d'itérations contiguës. La clause `schedule` permet à l'usager de choisir parmi certains algorithmes contrôlant la taille des morceaux et le moyen de distribution. Ces derniers sont présentés ci-dessous.

6. *Worksharing construct*

static : L'option *static* répartit uniformément le nombre d'itérations parmi les processus. La distribution se fait en une seule étape au début du calcul.

dynamic : L'algorithme *dynamic* distribue un morceau par thread. Lorsqu'un thread finit d'exécuter son morceau, il fait une demande pour en obtenir un autre jusqu'à ce qu'il n'en reste plus.

guided : L'algorithme *guided* distribue des morceaux d'itérations au fur et à mesure que les threads en font la requête. Toutefois, la granularité des morceaux est proportionnelle à la quantité de morceaux n'ayant pas encore été distribués.

Puisque les threads peuvent accomplir leur tâche à des vitesses différentes, il est possible, avec l'algorithme *static*, qu'un thread plus rapide ait besoin d'attendre ses homologues menant ainsi à une perte de performance. L'option *dynamic* vise à corriger ce problème en diminuant la granularité et en attribuant dynamiquement les morceaux. Ainsi un thread plus rapide exécutera plus d'itérations. Il y a une notion de partage des tâches. Toutefois, lorsque la granularité est fine, l'allocation dynamique entraîne un surcoût. L'option *guided* tente alors de faire un compromis entre les deux options précédentes en distribuant de gros morceaux au début et en raffinant la granularité vers la fin.

Les sections

La directive `Sections` Distribue des blocs de code aux différents threads. Chaque bloc est encadré par la sous-directive `Section`. Les *sections* ne sont exécutées qu'une seule fois et par un seul thread.

Un appel à la directive `Sections` est donc de la forme :

```
#pragma omp sections [clause[[,] clause] ...] new-line
    {
    [#pragma omp section new-line]
        structured-block
    [#pragma omp section new-line
        structured-block ]
    ...
    }
```

Single et Master

Le code situé sous la directive `Single` est exécuté par un seul thread. Pendant ce temps, le reste de l'équipe est en attente. La tâche peut être attribuée à n'importe quel thread. La directive `Master` est identique à `Single` à l'exception que seul le thread maître peut exécuter le bloc.

```
#pragma omp single [clause[[,] clause] ...] new-line
    structured-block
```

Cette directive est utilisée pour accomplir une tâche séquentielle sans sortir du bloc parallèle. Bien que la création de threads soit moins coûteuse que la création de processus, des entrées et sorties de blocs parallèles trop fréquentes peuvent affecter la performance du programme.

Synchronisation

L'API OpenMP fournit quelques mécanismes afin de gérer les accès concurrentiels aux données partagées. Les plus utiles sont décrites ici.

Barrier

La directive `barrier` est une fonction bloquante qui se termine seulement lorsque tous les threads d'une équipe ont rencontré la directive. Elle agit donc telle une barrière, car les instructions se situant au-dessus de la directive sont nécessairement exécutées par tous les threads avant le code en dessous de la barrière.

```
#pragma omp master new-line
    structured-block
```

Des barrières implicites sont ajoutées à la fin des directives `parrallel`, `for`, `sections` et `single`.

Critical

Dans l'éventualité qu'une partie des tâches implicites soit séquentielle, il est possible de définir une section critique. En effet, la directive `critical` encapsule un segment de code afin qu'un seul thread à la fois puisse l'exécuter. Lorsqu'un thread rencontre une section critique et qu'un autre membre de l'équipe est déjà à l'intérieur, il se met à la suite d'une file d'attente jusqu'à ce que la section soit libre. Au final, la section critique est exécutée une fois par toute l'équipe de threads.

```
#pragma omp critical [(name)] new-line
    structured-block
```

locks

Pour des problèmes de concurrence plus complexes, le mécanisme de synchronisation le plus polyvalent est l'utilisation de verrous. Ces derniers peuvent se retrouver dans deux états possibles : verrouillé ou déverrouillé. Pour chaque verrou, il existe une clé unique, ne donnant ainsi le contrôle qu'à un seul thread.

En C++, un verrou est encapsulé dans la structure `omp_lock_t`. Cet objet se manipule par l'entremise des routines suivantes :

omp_init_lock : Initialise les attributs de `omp_lock_t`

omp_destroy_lock : Efface les attributs de `omp_lock_t`

omp_set_lock : Attend jusqu'à ce que la clé soit disponible pour verrouiller le cadenas

omp_test_lock : Prend possession de la clé seulement si le verrou est débarré.

omp_unset_lock : Déverrouille le cadenas

L'objectif est donc de restreindre l'accès, à une variable ou un segment de code, au processus qui possède la clé. Ce dernier, lorsqu'il n'a plus besoin de la clé, peut la donner à un autre thread qui en fait la demande.

3.4 Paradigme de parallélisation hybride

À la section 3.1, nous avons introduit les architectures informatiques à mémoire partagée et à mémoire distribuée. Nous avons ensuite discuté, aux sections 3.2 et 3.3, des APIs de parallélisation adaptées à chacune d'entre elles. En réalité, ces catégories d'architectures ne sont pas mutuellement exclusives, car les plateformes de calcul haute performance sont généralement des grilles de nœuds SMP (Symmetric MultiProcessors) [7] à mémoire partagée. Sur ces systèmes hybrides, il n'est pas aussi évident de choisir un modèle de parallélisation optimal. Il est possible d'utiliser seulement MPI et d'obtenir de bonnes performances, mais il paraît généralement plus naturel d'utiliser un paradigme de parallélisation *hybride* pour gérer indépendamment les communications inter-nodales et intra-nodales.

À partir des APIs OpenMP et MPI, il existe tout un spectre de modèles de parallélisation hybride. Dans une étude de Rabenseifner [11], une classification de ces modèles est proposée. Cette dernière répartit les modèles parmi six classes exposées ci-dessous :

Pure MPI : Ce modèle fait abstraction des nœuds SMP et impose des processus sur tous les CPU (cœurs). Seuls des messages MPI sont utilisés, autant pour les communications inter-nodales qu'intra-nodales, engendrant ainsi un modèle de parallélisation à un seul niveau. Jusqu'à présent, ce modèle a démontré d'excellentes performances et est habituellement le choix de prédilection de nombreuses applications scientifiques. Toutefois, avec le nombre de CPU par nœud SMP qui croît continuellement et les performances réseau qui ne peuvent suivre la tendance, quelques problèmes apparaissent. En effet, même si l'implémentation de la bibliothèque MPI utilise la mémoire partagée pour optimiser l'envoi de messages à l'intérieur d'un nœud, la sémantique des communications MPI nécessite des copies mémoires, consommant la bande passante des connexions intra-nodales [12]. De plus, il suffit généralement de quelques threads pour saturer les connexions inter-nodales. La performance du modèle *pure MPI* dépend directement de

7. Multiprocesseur symétrique. On désigne par symétrique le fait que tous les processeurs soient identiques et que l'accès à la mémoire globale soit la même pour tous les processeurs

la répartition des processus sur les CPUs. Une mauvaise topologie virtuelle fera en sorte qu'une majorité des CPU ne communiquent pas avec d'autres CPU situés sur le même nœud SMP. Afin de minimiser les échanges entre nœuds, il faut que la topologie des processus MPI corresponde exactement avec la topologie des CPUs, ce qui est rarement le cas.

Hybride sans superposition : Le modèle hybride utilise un processus MPI par nœud SMP, dont chaque CPU est occupé par un thread OMP. Il n'y a donc pas de communications intra-nodales, car l'accès aux données est partagé dans le même espace mémoire. On peut donc s'attendre à ce que la compatibilité entre la topologie virtuelle des processus et celle des CPU ait un impact moins critique que pour le modèle *pure MPI*.

Dans cette catégorie, les phases de calculs parallèles sont séparées des phases de communication. Ici, on fait la distinction entre deux modèles dépendamment du nombre de CPU participant à l'échange de messages.

Master only : Le modèle hybride le plus simple consiste à effectuer les communications MPI seulement avec le thread maître. Toutefois, sur la majorité des plateformes de calculs, un thread ne suffit pas à utiliser pleinement la bande passante des connexions inter-nodales. La phase de communication est donc prolongée inutilement.

Multiple masters : Afin de remédier au problème de bande passante du modèle *maître unique*, le modèle *maîtres multiples* propose d'utiliser un petit groupe de threads pendant la phase de communication. La taille optimale du groupe varie selon les caractéristiques de l'architecture informatique et de l'application. Dans ce cas, la bibliothèque MPI doit être compatible avec une parallélisation multithreads.

L'inconvénient majeur de cette catégorie est qu'elle réduit la fraction parallèle du programme, puisque les threads ne participant pas aux appels MPI sont soit en attente, soit détruits. De plus, une barrière OMP est nécessaire avant chaque phase de communication causant d'autres délais de synchronisation.

Hybride avec superposition : Une deuxième catégorie de modèles hybrides a comme objectif d'introduire les communications à l'intérieur des sections parallèles où certains threads s'occupent des communications et les autres, du traitement de données. Cette formule est probablement optimale, mais elle demande généralement un effort de programmation considérable. En effet, chaque opération doit être implémentée de façon à séparer les tâches dépendantes des communications des tâches qui n'en ont pas besoin.

Encore une fois, selon le nombre de CPU participant aux communications, cette catégorie est subdivisée en deux modèles :

Hybrid funneled : Toutes les données périphériques [8] sont acheminées au thread maître qui se chargera de les envoyer.

Hybrid multiple : Le groupe dédié aux communications est constitué du thread maître et de quelques autres.

L'inconvénient du modèle avec superposition est qu'il est difficile, ou parfois même impossible, d'obtenir une répartition équitable de la charge de travail entre les groupes de threads communicateurs et travailleurs.

Pure OMP Une autre alternative est de recourir à l'API OpenMP uniquement. En effet, la version d'Intel, *Cluster OpenMP*, permet d'émuler un environnement à mémoire partagée alors que l'architecture est distribuée [9]. Les communications inter-nodales nécessaires au maintien de la cohérence mémoire sont gérées par l'API plutôt que par le programme. Malheureusement, le produit d'Intel est aujourd'hui discontinué [13].

Rabenseifner [11] fait également mention d'un modèle **mixte** entre *pure MPI* et *hybride*. Ce paradigme utilise plusieurs processus MPI par nœuds SMP, chacun avec un groupe de threads. Cela permet de combiner les avantages et inconvénients des deux modèles.

Avec l'arrivée de la version 3 du standard MPI en 2015, il est désormais possible de programmer un paradigme de parallélisation hybride uniquement avec la bibliothèque MPI [12]. Le mécanisme sous-jacent consiste à créer une fenêtre qui donne une vue sur un espace mémoire partagée. Par le biais de fonctions d'accès à distance [10], les processus peuvent écrire ou lire des données dans cette fenêtre. La différence avec les communications MPI traditionnelles est qu'un sous-système de MPI s'occupe de maintenir la cohérence de l'espace mémoire partagé.

En rétrospective, il existe une panoplie de modèles de parallélisation possible pour les architectures hybrides, et le choix d'un modèle optimal pour une application n'est pas du tout évident. Pour des raisons de simplicité, nous avons choisi d'utiliser, pour la parallélisation du programme *EF7* un modèle mixte, de style *master only*, où un processus MPI est imposé par processeur (socket).

8. Données, situées à la frontière d'un sous-domaine, devant être partagées par plusieurs processus.
9. On parle souvent de *virtual Distributed Shared Memory (DSM)*
10. remote memory access (RMA)

CHAPITRE 4 PARALLÉLISATION DE LA MÉTHODE DES ÉLÉMENTS FINIS

Que ce soit en industrie ou en milieu académique, la taille des problèmes étudiés en analyse par éléments finis ne cesse de croître. Les besoins en mémoire et en puissance de calcul semblent toujours autant d'actualité. Pour pallier la situation, les ordinateurs incorporent de plus en plus de CPUs dans leur architecture. La parallélisation d'une application est donc primordiale pour tirer profit des processeurs actuels. En éléments finis, la parallélisation va généralement de pair avec la notion de décomposition de domaine.

4.1 Décomposition de domaine

En éléments finis, la décomposition de domaine consiste à scinder le maillage en sous-domaines de façon à ce que chacun puisse être traité individuellement. Une partition est généralement associée à un processus ou à un thread. Pour l'assemblage, par exemple, les processus peuvent insérer les contributions élémentaires provenant uniquement des éléments de leur partition.

4.1.1 Deux types de décomposition de domaine

Selon la topologie du partitionnement du maillage, on distingue les deux modèles de décomposition de domaine définis ci-dessous :

Définition 4.1
La décomposition d'un domaine Ω en un ensemble de P sous-domaines $\Omega_i \subset \Omega$ est dite **sans recouvrement** *si les conditions suivantes sont respectées :*

$$\begin{cases} \bigcup_{i=0}^{P} \Omega_i & = \Omega \\ \Omega_i \cap \Omega_j & = \emptyset \qquad\qquad \textit{pour } i \neq j \end{cases} \tag{4.1}$$

Définition 4.2
La décomposition d'un domaine Ω en un ensemble de P sous-domaines $\Omega_i \subset \Omega$ est dite **avec recouvrement** *si la condition suivante est respectée :*

$$\bigcup_{i=1}^{P} \Omega_i = \Omega \tag{4.2}$$

Pour de tels domaines, il est pratique de définir $\Omega_i^* = \Omega_i \cap (\cup \Omega_j)$ *pour* $j \neq i$ *comme l'intersection d'une région avec son entourage et* $\bar{\Omega}_i = \Omega_i \backslash \Omega_i^*$.

Afin d'éviter certaines ambiguïtés, on utilise la notation $B^{(i)} = \partial\Omega_i \cap \Omega$ et $B_{[i]} = \partial\Omega_i \cap \partial\Omega$ pour identifier les sections intérieures et extérieures de la frontière $B_i = \partial\Omega_i$ d'une partition. L'ensemble des frontières intérieures est dénoté par $B = \cup B^{(i)}$. On définit également la frontière $B_{ij} = B_i \cap B_j$ entre les partitions i et j.

En pratique, une décomposition avec recouvrement est souvent obtenue par extension des partitions résultant d'une décomposition sans recouvrement.

4.1.2 Logiciels de partitionnement de graphes

Une décomposition permet d'obtenir un bon taux de parallélisme, si deux critères sont respectés :

— La charge de travail associée aux partitions est équilibrée. Si ce n'est pas le cas, certains processus accompliront leur tâche et resteront en attente jusqu'à ce que tous les processus aient terminé.

— Les interactions entre les partitions sont minimales. En effet, les opérations sur une partition sont généralement séquentielles. Le traitement des données aux frontières des partitions requiert donc une synchronisation soit par l'intermédiaire de communications MPI ou de sections critiques OMP. On cherche alors à minimiser les frontières B.

Le partitionnement du maillage relève d'un problème d'optimisation NP-complet. Heureusement, plusieurs bibliothèques, issues de la théorie des graphes, proposent des heuristiques généralement efficaces. Les plus répandues sont *Métis* (*ParMétis*) [14], *Scotch* (*PT-Scotch*) [15, 16], *PARTY* [17] et *Chaco* [18].

En fait, ces programmes ne partitionnent pas des maillages mais plutôt des graphes. Heureusement, il est possible de générer, à partir d'un maillage, deux types de graphe : le graphe dual et le graphe des nœuds. Le premier représente l'adjacence entre les éléments, alors que le second forme l'adjacence entre les nœuds. Ces graphes engendrent respectivement des partitionnements appelés *node-cut* et *element-cut*, car la ligne de séparation des partitions est située sur les nœuds ou sur les éléments, tel qu'illustré à la figure 4.1. On remarque également que le partitionnement *node-cut* est adapté pour une décomposition sans recouvrement, alors que l'autre est adapté pour une décomposition avec recouvrement.

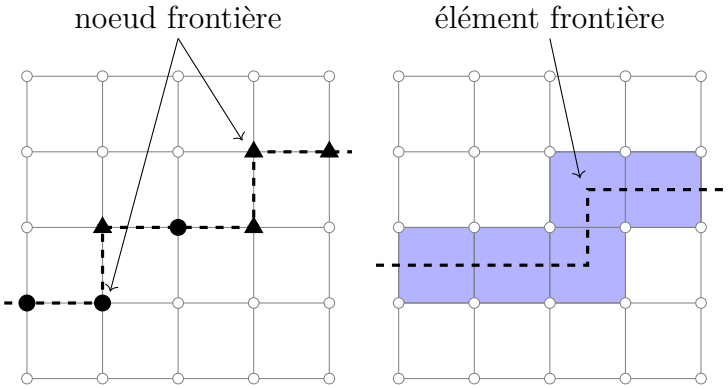


Figure 4.1 Partitionnement de graphes

Les partitions, produites depuis une division *node-cut*, ne partagent que les nœuds aux frontières alors qu'un partitionnement *element-cut* doit dupliquer toutes les informations des éléments aux frontières. il y a donc un impact direct sur le volume d'informations à communiquer entre les processus. En fait, le choix d'un partitionnement *node-cut* ou *element-cut* implique respectivement un assemblage global ou local [19].

Assemblage local : Les contributions élémentaires sont assemblées uniquement dans la partie locale de la matrice globale. Pour les éléments frontières, les lignes correspondant aux DDLs non locaux sont simplement ignorées.

Assemblage global : Les éléments sont assemblées et les contributions élémentaires non locales sont communiquées à leur propriétaire pour que ce dernier les assemble.

Un assemblage local a donc le désavantage d'avoir des calculs redondants alors qu'un assemblage global engendre un plus grand volume de communications. L'écart de performance entre ces deux méthodes est négligeable lorsque moins de quelques centaines de processus sont utilisés [20] [1]. Après ce seuil, la latence de communication devient trop importante et l'assemblage local est recommandé.

Pour des décomposition sans recouvrement, les objets situés sur une frontière appartiennent à l'un des processus adjacents. Ce processus est donc appelé *processus propriétaire* alors que les autres sont des *processus locataires*. Pour un propriétaire, l'objet en question est local et considéré au même titre que les autres objets au centre de la partition. Du point de vue des locataires, ce même objet est identifié comme *fantôme*, car les processus le voient, mais ne

1. Ces résultats sont valides pour un paradigme MPI mais n'ont pas été testés dans un paradigme hybride où la proportion frontière/domaine est beaucoup plus petite.

peuvent interagir directement avec lui. L'objet fantôme n'est qu'une projection d'un objet réel situé sur une autre partition. Cette notion de fantôme permet d'éviter des conflits entre un processus propriétaire et locataire. Il est parfois utile de sélectionner uniquement les objets réels situés aux frontières. Tout au long de ce document, nous utilisons le terme de *maître* pour désigner ce groupe. Ainsi la frontière d'une partition est constituée d'objets maîtres et fantômes.

4.1.3 Décomposition à plusieurs niveaux

Avec un paradigme de parallélisation hybride, il est naturel d'avoir au moins deux niveaux de décomposition de domaine. Le premier, plus grossier, est associé aux processus MPI alors que le deuxième est pour les threads OMP. En fait, on peut s'attendre à ce qu'il y ait autant de décompositions que de régions parallèles imbriquées.

Toutefois, dans un contexte de mémoire partagée, le partitionnement peut prendre une forme alternative : le coloriage. En fait, le concept du coloriage est entièrement opposé à celui du partitionnement traditionnel qui consiste à trouver des sous-ensembles d'éléments contigus. Le coloriage identifie des ensembles d'éléments avec des couleurs tel que tous les éléments d'une même couleur sont disjoints. Les éléments sont alors coloriés tels les pays d'une carte du monde. Cette particularité garantit qu'aucun DDL n'est partagé par deux éléments d'une même couleur et donc, que ces éléments peuvent être assemblés simultanément sans problème de concurrence. Il existe plusieurs algorithmes pour ce type d'application et ils peuvent produire des colorations différentes [21]. Puisqu'une synchronisation des threads (*omp barrier*) est nécessaire pour chaque couleur, l'algorithme optimal est celui produisant un minimum de couleur. Soulignons que cette valeur est toujours bornée inférieurement par le degré maximal $\Delta(G)$ du graphe dual G. Un maillage hautement irrégulier peut donc affecter la performance de ces algorithmes.

Cette approche est particulièrement intéressante, car elle est plus facile à incorporer qu'un partitionnement traditionnel. En effet, il n'y a pas de gestion particulière pour les nœuds ou DDLs aux frontières. Il est seulement nécessaire de regrouper les éléments par couleur. Le coloriage est tout à fait approprié pour la directive de boucle OpenMP. On peut alors bénéficier facilement de répartition dynamique de la charge de travail.

Une autre approche consiste à utiliser récursivement l'algorithme de bissection de *Métis* pour partitionner le maillage dans un style *Diviser pour régner* [22]. Cette méthode bénéficie des données plus localisées et les coefficients de la matrice formée sont, en moyenne, plus près de la diagonale.

4.2 Résolution parallèle d'un système linéaire

La section précédente aborde la décomposition de domaine principalement dans le contexte de l'assemblage, mais elle a également un impact direct sur la résolution du système linéaire global. Il existe deux approches pour résoudre un tel système distribué : utiliser un solveur distribué ou une *formulation hybride*.

4.2.1 Solveurs distribués

L'usage de solveurs distribués permet de traiter un problème, dont le domaine est décomposé, sensiblement de la même manière que le cas séquentiel. Autrement dit, un système algébrique global est construit et fourni au solveur pour être résolu. Puisque toutes les communications sous-jacentes à la résolution sont encapsulées dans le solveur, cette approche requiert normalement un effort de programmation moindre de la part de l'utilisateur. Toutefois, ce dernier doit gérer la construction d'un système global mais distribué sur les processus. Des indices locaux et globaux doivent alors être définis pour les DDLs ainsi qu'une méthode pour traduire les indices d'un ensemble à l'autre [23].

Plusieurs solveurs distribués sont disponibles en libre accès. Dans le domaine des solveurs itératifs, la bibliothèque d'algèbre linéaire *PETSc* [24] représente l'état de l'art dans l'implémentation des méthodes de Krylov. *Trilinos* [25] et *Hypre* [26] sont des bibliothèques, parmi tant d'autres, offrant des méthodes de résolutions itératives. Ces solveurs bénéficient d'un parallélisme MPI mais ne supportent généralement pas, ou dans un cadre limité, l'usage de threads OpenMP. En ce qui concerne les méthodes directes, les solveurs *Mumps* [27], *Cluster Pardiso* de Intel [28] [2] et la version disribuée de *SuperLU* [29] sont les options les plus courantes [30, 31]. Les performances de ces derniers sont généralement optimales lorsqu'ils sont utilisés dans un context hybride MPI/OMP.

Nous ne nous attarderons pas sur la mécanique interne de ces solveurs. Notre objectif est plutôt de présenter leur modes d'utilisation et plus spécifiquement, les formats des matrices requises par certains d'entre eux. Nous nous attarderons plus spécifiquement aux solveurs *Cluster Pardiso* de Intel, *Mumps* ainsi que l'interface offerte par *PETSc*.

Cluster Pardiso

Comme son nom l'indique, le solveur direct *CPardiso* est une version de *Pardiso*, incorporant le protocole MPI, pouvant être exécuté depuis une plateforme à mémoire distribuée. Il

2. où *CPardiso*

conserve également le parallélisme multithread de sa version originale, lui conférant typiquement un maximum de performance lorsqu'il est utilisé dans un paradigme hybride de style *funneled* [32].

Son mode d'emploi est, à toute fin pratique, identique à celui de *Pardiso*. En effet, les appels au solveur, pour les différentes phases de calcul, se font tous à partir de la méthode `cluster_sparse_solver`[3]. Cette dernière prend en entrée les arguments suivants :

— `Mtype` : Définit le type de matrice (symétrique, définie positive, complexe, etc) ;

— `Phase` : Définit le type de calcul (analyse, factorisation, résolution) à exécuter ;

— `Ai, Aj, Ax, B` : Les structures décrivant le système d'équations ;

— `Iparm` : Contrôle des paramètres ;

— `Comm` : Communicateur MPI ;

Toutes les options fournies au solveur sont définies dans le vecteur `Iparm`. Ces dernières sont les mêmes que pour *Pardiso* à l'exception des cases 1 et 39-41.

`Iparm[39]` définit le format de matrice en entrée. Il est possible de fournir une matrice CSR (Compressed Sparse Row) depuis un processus unique ou une matrice DCSR (Distributed Compressed Sparse Row) depuis tous les processus. Ces dernières sont présentées ci-dessous.

Le format CSR décrit la structure et le contenu d'une matrice creuse, à l'aide des trois vecteurs $\mathbf{A}_p$, $\mathbf{A}_j$ et $\mathbf{A}_x$, de manière à ce que les coefficients d'une même ligne soit contigus en mémoire et classés, en ordre croissant, par l'index de leur colonne. Les vecteurs $\mathbf{A}_j$ et $\mathbf{A}_x$, de longueur n_{nz}, le nombre de coefficient non nuls, renferment respectivement les indices des colonnes et les valeurs des coefficients. Le vecteur $\mathbf{A}_p$, d'une taille n_L équivalente au nombre de lignes de la matrice, contient la position du premier coefficient de chaque ligne.

La version distribuée de ce format consiste simplement à répartir les lignes de la matrice parmi les processus, en sous-ensembles contigus. Chaque processus possède donc une matrice locale CSR $A^{(i)} \equiv \left\{\mathbf{A}_p^{(i)},\ \mathbf{A}_j^{(i)},\ \mathbf{A}_x^{(i)}\right\}$ où $\mathbf{A}_j^{(i)}$ contient les indices globaux des colonnes. Lorsque ce format est choisi, *CPardiso* requiert que les indices de la première et dernière lignes des matrices locales soient donnés en `Iparm[40]` et `Iparm[41]`.

`Iparm[1]` permet de sélectionner une méthode de réordonnancement parmi un algorithme séquentiel, multithread ou distribué. Pour les deux premiers choix, l'analyse est exécutée uniquement sur le processus maître. Si la matrice est de type DCSR, elle est également centralisée sur ce processus. Dans cette situation, rien n'empêche les matrices locales de se chevaucher, car *CPardiso* se charge d'additionner les contributions des lignes partagées par

3. Un équivalent, avec le suffixe `_64`, est défini pour l'usage d'entier à 64 bits.

plusieurs processus. En ce qui concerne l'algorithme distribué, la matrice n'est pas centralisée et diminue considérablement le coût en mémoire.

Mumps

Mumps est également un solveur direct profitant d'une parallélisation MPI et OpenMP. Par rapport aux fonctionnalités et aux arguments d'entrée, ce solveur est très similaire à *CPardiso*.

Toutefois, il existe des différences majeures en ce qui concerne le format de la matrice ainsi que la distribution des vecteurs solution et RHS. En effet, *Mumps* requiert une matrice dans un format COO (Coordinates). Cette dernière décrit les coefficients d'une matrice A à l'aide de trois vecteur $\mathbf{A}_i, \mathbf{A}_j, \mathbf{A}_x$, de longueur n_{nz}, où $\mathbf{A}_i$ renferme les indices de ligne pour tous les coefficients. Comparativement au format CSR, ce format est moins compact et plus difficile à manipuler. Néanmoins, les coefficients peuvent être incorporés dans la matrice dans n'importe quel ordre. Pour une matrice distribuée, ce format possède la propriété qu'une matrice locale $A^{(i)}$ peut inclure n'importe quel coefficient, peu importe sa position dans la matrice globale.

Bien que le format de matrice confère un maximum de flexibilité, il n'en est pas de même pour le membre de droite. Ce dernier doit être centralisé sur le processus maître. En ce qui concerne le vecteur solution, *Mumps* a la possibilité de le stocker entièrement sur le processus maître ou de manière distribuée. Dans le second cas, la répartition du vecteur est définie par les structures internes de *Mumps* et ne correspond généralement pas à l'indexation propre à l'application.

PETSc

PETSc est une bibliothèque mutli-fonctionnelle. Son module *KSP* (Krylov SubSpace) permet d'utiliser des solveurs itératifs aussi bien que certains solveurs directs. Dans les deux cas, le système linéaire est construit à partir des objets distribués *mat* et *vec*.

En réalité, *PETSc* offre plusieurs formats possibles de matrice distribuée, mais nous ne considérons ici que le type *mpiaij*. Ce format est similaire au format DCSR. Cependant, les différents processus ne peuvent pas partager les mêmes lignes comme c'est le cas avec *CPardiso*. De plus, les colonnes sont également partitionnées en sous-ensembles continus pour former des blocs diagonaux et hors diagonaux tel qu'illustré à la figure 4.2. Concrètement, chaque processus conserve les coefficients de son bloc diagonal dans une matrice CSR et ceux des blocs hors diagonaux dans une autre matrice CSR.

Les structures de *PETSc* permettent à un processus d'insérer un élément situé sur une ligne appartenant à un autre processus. À l'interne, un processus possède une liste dynamique de

Processus 0	D	HD	HD
Processus 1	HD	D	HD
Processus 2	HD	HD	D

Figure 4.2 Structure par blocs diagonaux et hors diagonaux des matrices *PETSc*

coefficients à envoyer aux autres processus. Dès qu'un processus reconnaît qu'un coefficient n'est pas local, il le place à la fin de la liste. Les communications se font depuis les appels aux fonctions `MatAssemblyBegin` et `MatAssemblyEnd`. Ensuite, la liste est réinitialisée.

4.2.2 Formulations hybrides

Les méthodes de résolution distribuée les plus performantes sont généralement issues de formulations dites *hybrides*. Ces dernières consistent à reformuler un système d'équations en un ensemble équivalent de sous-problèmes locaux pouvant être résolus indépendamment. La solution globale $\mathbf{u}$ est ensuite obtenue par combinaison linéaire des solutions locales $\mathbf{w}_i$

$$\mathbf{u} = \sum_{i=1}^{P} \chi_i \mathbf{w}_i \tag{4.3}$$

où les χ_i sont un ensemble de fonctions continues appelées *partition d'unité* et défini tel que

$$\begin{cases} \chi_i \geq 0, & \text{dans } \Omega_i \\ \chi_i = 0, & \text{dans } \Omega/\Omega_i \\ \sum \chi_i = 1, & \text{dans } \Omega \end{cases} \tag{4.4}$$

Dans l'ouvrage de T. Mathiew [33] quatre formulations sont définies :

— formulation de Schwarz ;

— formulation de Steklov-Poincaré ;

— formulation par multiplicateur de Lagrange ;

— formulation de contrôle par moindres carrés [4].

Puisqu'il existe, pour chaque formulation, une grande variété de méthodes de résolution, il est impossible de dresser une liste exhaustive dans le cadre de ce mémoire. Afin d'introduire les

4. Traduction de *least squares-control formulation*

concepts principaux, nous aborderons deux algorithmes typiques : l'algorithme de Schwarz classique alterné et l'algorithme de sous-structuration.

Algorithme de Schwarz classique alterné

L'algorithme de Schwarz est l'une des premières méthodes de décomposition de domaine développées pour des domaines avec recouvrement. Sa formulation hybride consiste à placer des conditions de Dirichlet $\mathcal{B}_\mathcal{D}$ sur les frontières intérieures $B^{(i)}$ et d'y imposer la solution courante $\mathbf{w}_v$ des processus voisins.

Pour une équation elliptique (équation (4.5)), la formulation de Schwarz prend la forme donnée à l'équation (4.6).

$$\begin{cases} Lu \equiv -\nabla \cdot (a(x)\nabla)\, u + c(x)u &= f, \quad \text{dans } \Omega \\ \quad Mu \equiv \mathbf{n} \cdot (a(x)\nabla u) + \gamma u &= g_N, \quad \text{sur } \mathcal{B}_\mathcal{N} \\ \qquad\qquad\qquad\qquad\qquad u &= 0, \quad \text{sur } \mathcal{B}_\mathcal{D} \end{cases} \tag{4.5}$$

$$\begin{cases} Lw_i &= f, & \text{dans } \Omega_i^* \\ Mw_i &= g_N, & \text{sur } B_{[i]} \cap \mathcal{B}_\mathcal{N} \\ w_i &= w_v, & \text{sur } B^{(i)} \\ w_i &= 0, & \text{sur } B_{[i]} \cap \mathcal{B}_\mathcal{D} \end{cases} \tag{4.6}$$

L'algorithme classique de Schwarz est un procédé itératif qui résout les différents sous-systèmes à tour de rôle, de manière à converger vers la solution du problème (4.5). Entre chaque résolution d'un domaine Ω_i, les conditions de Dirichlet imposées sur $B^{(i+1)}$ sont mises à jour.

On remarque que l'algorithme 1 est séquentiel et que, par conséquent, il n'est jamais utilisé en pratique. Évidemment, plusieurs variantes permettent de profiter de plus de parallélisme. Un exemple consiste à regrouper les partitions par couleur $\mathcal{C}_k$ tel que

$$\Omega_i \cap \Omega_j = \emptyset, \quad \forall\ \Omega_i,\ \Omega_j \in \mathcal{C}_k.$$

Les partitions d'une même couleur peuvent donc être exécutées en parallèle. D'autres méthodes comme l'algorithme de Schwarz additif ou multiplicatif permettent d'obtenir des taux de parallélisme encore plus élevés.

Algorithme 1 Algorithme de Schwarz classique alterné

Input: initial solution $w^{(0)}$

1: **for** k←0,1,... until convergences, **do**
2: **for** i←1 to P **do**
3: Solve :

$$\begin{cases} Lv^{k+\frac{i}{P}} &= f, & \text{dans } \Omega_i^* \\ Mv^{k+\frac{i}{P}} &= g_N, & \text{sur } B_{[i]} \cap \mathcal{B}_{\mathcal{N}} \\ v^{k+\frac{i}{P}} &= w^{k+\frac{i-1}{P}}, & \text{sur } B^{(i)} \\ v^{k+\frac{i}{P}} &= 0, & \text{sur } B_{[i]} \cap \mathcal{B}_{\mathcal{D}} \end{cases}$$

4: **end for**
5: Update :

$$w^{k+\frac{i}{P}} \leftarrow \begin{cases} v^{k+\frac{i}{P}}, & \text{dans } \bar{\Omega}_i \\ w^{k+\frac{i-1}{P}}, & \text{dans } \Omega / \ \bar{\Omega}_i \end{cases}$$

6: **end for**

Algorithme de sous-structuration

À l'opposé des méthodes de Schwarz, l'algorithme de sous-structuration, basé sur le complément de Schur, adopte une approche de résolution plus directe. En effet, l'objectif de cette méthode est de retrouver, dans un premier temps, la solution aux frontières internes B et ensuite de résoudre les systèmes locaux en parallèle.

En fait, l'algorithme tire avantage d'une décomposition sans recouvrement pour construire une matrice bloc quasi-diagonale. Pour ce faire, la numérotation des DDLs doit correspondre à un ordre spécifique. Pour $n_I^{(i)}$ et $n_B^{(i)}$, le nombre de DDLs intérieurs et aux bords de Ω_i, la numérotation globale est telle que

$$\begin{cases} DDL_j \in \Omega_i, \text{ pour } \sum_{k=1}^{i-1} n_I^{(i)} \leq j \leq \sum_{k=1}^{i} n_I^{(i)}, \quad 1 \leq i \leq P \\ DDL_j \in B, \quad \text{pour } n_I \leq j \leq n_I + n_B \end{cases} \tag{4.7}$$

Cet ordonnancement a la propriété de former, à l'assemblage de la matrice globale, un système de la forme :

$$\begin{bmatrix} A_{II}^{(1)} & & 0 & A_{IB}^{(1)} \\ & \ddots & & \vdots \\ 0 & & A_{II}^{(P)} & A_{IB}^{(P)} \\ A_{BI}^{(1)} & \cdots & A_{BI}^{(P)} & A_{BB} \end{bmatrix} \begin{bmatrix} \mathbf{u}_I^{(1)} \\ \vdots \\ \mathbf{u}_I^{(P)} \\ \mathbf{u}_B \end{bmatrix} = \begin{bmatrix} \mathbf{f}_I^{(1)} \\ \vdots \\ \mathbf{f}_I^{(P)} \\ \mathbf{f}_B \end{bmatrix} \tag{4.8}$$

On observe, à l'équation (4.8), que la sous matrice A_{II} possède une structure bloc diagonale. Cela implique que la résolution des $\mathbf{u}_I^{(1)}$ peut s'exécuter en parallèle, à la condition que $\mathbf{u}_B$ soit préalablement déterminé. On cherche donc à découpler les équations de manière à former un système linéaire uniquement pour $\mathbf{u}_B$. Pour y arriver, il faut se départir du bloc A_{BI} ; c'est là que la notion de complément de Schur intervient.

Tout d'abord, il s'avère pratique de numéroter les DDLs aux bords en dernier de manière à obtenir le système linéaire (4.9).

$$A^{(i)}\mathbf{u}^{(i)} = \mathbf{f}^{(i)} \Longrightarrow \begin{bmatrix} A_{II}^{(i)} & A_{IB}^{(i)} \\ A_{BI}^{(i)} & A_{BB}^{(i)} \end{bmatrix} \begin{bmatrix} \mathbf{u}_I^{(i)} \\ \mathbf{u}_B^{(i)} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_I^{(i)} \\ \mathbf{f}_B^{(i)} \end{bmatrix} \tag{4.9}$$

On procède ensuite à une élimination de Gauss par bloc pour obtenir le système du complément de Schur

$$\begin{bmatrix} A_{II}^{(i)} & A_{IB}^{(i)} \\ 0 & A_{BB}^{(i)} - A_{BI}^{(i)} \left(A_{II}^{(i)}\right)^{-1} A_{IB}^{(i)} \end{bmatrix} \begin{bmatrix} \mathbf{u}_I^{(i)} \\ \mathbf{u}_B^{(i)} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_I^{(i)} \\ \mathbf{f}_B^{(i)} - A_{BI}^{(i)} \left(A_{II}^{(i)}\right)^{-1} \mathbf{f}_I^{(i)} \end{bmatrix} \tag{4.10}$$

$$\begin{bmatrix} A_{II}^{(i)} & A_{IB}^{(i)} \\ 0 & S^{(i)} \end{bmatrix} \begin{bmatrix} \mathbf{u}_I^{(i)} \\ \mathbf{u}_B^{(i)} \end{bmatrix} = \begin{bmatrix} \mathbf{f}_I^{(i)} \\ \mathbf{f}_S^{(i)} \end{bmatrix} \tag{4.11}$$

où $S^{(i)} = A_{BB}^{(i)} - A_{BI}^{(i)} \left(A_{II}^{(i)}\right)^{-1} A_{IB}^{(i)}$ et $\mathbf{f}_S^{(i)} = \mathbf{f}_B^{(i)} - A_{BI}^{(i)} \left(A_{II}^{(i)}\right)^{-1} \mathbf{f}_I^{(i)}$ représentent le complément de Schur et le membre de droite associé. Remarquons qu'à l'équation (4.11), la matrice $A^{(i)}$ est triangulaire supérieure par bloc et que par construction, la matrice globale hérite de cette propriété. En effet, l'assemblage génère ensuite un système de la forme [5] :

$$\begin{bmatrix} A_{II}^{(1)} & & 0 & A_{IB}^{(1)} \\ & \ddots & & \vdots \\ & & A_{II}^{(P)} & A_{IB}^{(P)} \\ & & & S \end{bmatrix} \begin{bmatrix} \mathbf{u}_I^{(1)} \\ \vdots \\ \mathbf{u}_I^{(P)} \\ \mathbf{u}_B \end{bmatrix} = \begin{bmatrix} \mathbf{f}_I^{(1)} \\ \vdots \\ \mathbf{f}_I^{(P)} \\ \mathbf{f}_S \end{bmatrix} \tag{4.12}$$

où

$$S = \sum_{i=1}^{P} S^{(i)} \qquad\qquad \mathbf{f}_S = \sum_{i=1}^{P} \mathbf{f}_S^{(i)}$$

De (4.12), on peut résoudre $S\mathbf{u}_B = \mathbf{f}_S$ indépendamment du reste du système. La solution

5. Il est important de souligner ici que le symbole de somme est interprété au sens d'assemblage éléments finis.

locale $\mathbf{u}_B^{(i)}$ peut ensuite être extraite afin de résoudre en parallèle les systèmes $A_{II}^{(i)}\mathbf{u}_I^{(i)} = A_{IB}^{(i)}\mathbf{u}_B^{(i)}$.

Finalement, la méthode de sous-structuration peut être résumée par l'algorithme 2.

Algorithme 2 Algorithme de sous-structuration

for i←0 to p, in parallel, **do**
    Assemble : $A_{II}^{(i)},\ A_{IB}^{(i)},\ A_{BB}^{(i)},\ \mathbf{f}_I^{(i)},\ \mathbf{f}_B^{(i)}$
    Compute Cholesky factorisation of $A_{II}^{(i)} = L^{(i)}L^{(i)^T}$
    Assemble $S^{(i)} = A_{BB}^{(i)} - A_{IB}^{(i)^T} L^{(i)^{-T}} L^{(i)^{-1}} A_{IB}^{(i)}$
    Assemble $\mathbf{f}_S^{(i)} = \mathbf{f}_B^{(i)} - A_{IB}^{(i)^T} L^{(i)^{-T}} L^{(i)^{-1}} \mathbf{f}_I^{(i)}$
end for

Compute $S = \sum_{i=0}^{P} S^{(i)}$.
Compute $\mathbf{f}_S = \sum_{i=0}^{P} \mathbf{f}_S^{(i)}$.
Solve $S\mathbf{u}_B = \mathbf{f}_S$

for i←0 to p, in parallel, **do**
    Solve $A_{II}^{(i)}\mathbf{u}_I^{(i)} = A_{IB}^{(i)}\mathbf{u}_B^{(i)}$
end for

Output: solution $\left(\mathbf{u}_I^{(1)^T}, \cdots, \mathbf{u}_I^{(P)^T}, \mathbf{u}_B^T\right)^T$

CHAPITRE 5 UNE VERSION DISTRIBUÉE DE *EF*

Dans ce chapitre, nous présenterons le compte rendu des principales modifications apportées au programme d'éléments finis (*EF*) afin d'en faire une version massivement parallèle. L'objectif est de fournir une vision d'ensemble pour les utilisateurs d'*EF* tout en donnant suffisamment de détails pour reproduire les algorithmes. En premier lieu, le programme EF est brièvement présenté. Ensuite, les modules responsables du maillage, de la numérotation des DDLs ainsi que de l'assemblage sont détaillés individuellement.

5.1 Introduction à EF

EF est un programme d'analyse par éléments finis développé au sein de la section aérothermique de Polythechnique Montréal, plus spécifiquement par le groupe de recherche dirigé par le professeur André Garon. Le logiciel est principalement utilisé pour résoudre des problèmes de dynamique des fluides et d'échange de chaleur, mais permet à l'usager de créer ses propres équations. Le programme peut résoudre autant les problèmes stationnaires qu'instationnaires, multidimensionnels ou même multi-physiques. Toutefois, la grande particularité d'*EF* est son implémentation de la méthode des éléments spectraux. En effet, *EF* admet l'usage de polynômes d'interpolation de degré élevé pour une précision de calcul accrue.

Le programme *EF*, pour s'exécuter, a besoin de deux fichiers. Le premier est un fichier de configuration renfermant la description du problème ainsi que toutes les options et paramètres reliés au logiciel. Le deuxième, lui, contient le maillage. Ce dernier est habituellement généré avec le logiciel libre Gmsh en format mesh (INRIA). Il est ensuite converti dans un format maison afin de regrouper les éléments par objets physiques. Ces objets sont une abstraction de Gmsh pour identifier un groupe d'éléments, mais ils sont également utilisés dans *EF*. La possibilité d'utiliser des interpolants de degré élevé contraint *EF*, dans son implémentation actuelle, à utiliser des éléments bien spécifiques. Pour des raisons techniques, il est primordial qu'il y ait un nœud par entité géométrique. Par exemple, un élément triangulaire à six nœuds, tel que présenté à la figure 5.1a, ne conviendrait pas, car aucun nœud n'est attribué à la surface de l'élément. En revanche, un triangle à sept nœuds possède un nœud sur tous ses sommets, arêtes et faces. La conversion au format *maillage* permet donc l'ajout de nœuds. Cette caractéristique provient du fait que les nœuds géométriques agissent en tant que supports pour les DDLs. Ce sujet sera traité plus en détails à la section 5.3.1.

À partir de ces fichiers de configuration et de maillage, le programme s'exécute et retourne

(a) Triangle à 6 nœuds

(b) Triangle à 7 nœuds

Figure 5.1 Éléments admissibles pour EF7

les champs des solutions aux temps désirés ainsi que des résultats secondaires calculés lors du post-traitement. L'exécution du programme se déroule en 3 étapes :

Étape 1 - Initialisation : Dans cette phase, les fichiers de configuration et de maillage sont d'abord traités afin de construire le domaine géométrique et les équations qui s'y rattachent. On procède ensuite à la numérotation des DDLs, à l'allocation mémoire du système matriciel et, pour terminer, à une analyse de la structure de non-zéros du système. Cette analyse réordonne les lignes et les colonnes de la matrice, réduisant ainsi sa largeur de bande, afin de minimiser le phénomène de *fill-in* lors de la factorisation matricielle.

Étape 2 - Résolution : Cette étape consiste à construire le système matriciel et à le résoudre. Jusqu'à présent, la résolution se fait à l'aide du solveur direct *Pardiso* de la *Math Kernel Library* (MKL) d'*Intel*. Ce dernier procède en premier à une factorisation incomplète LU pour ensuite facilement résoudre le problème. Ces opérations sont généralement effectuées de nombreuses fois lors d'une simulation. En fait, l'appel au solveur est situé à l'intérieur de deux boucles imbriquées. La première provient de l'algorithme de Newton utilisé pour résoudre les équations non linéaires. La deuxième, elle, provient de la résolution temporelle. Il est important de souligner que le système linéaire n'est pas reconstruit à chaque itération mais uniquement si le taux de convergence est trop faible.

Étape 3 - Finalisation : Avant la fin du programme, cette phase sert à l'écriture des champs de solutions et aux calculs de certains post-traitements.

Le programme *EF* est codé en C++ dans un paradigme orienté objet. Sous sa carrosserie, *EF* est divisé en quatre grands modules :

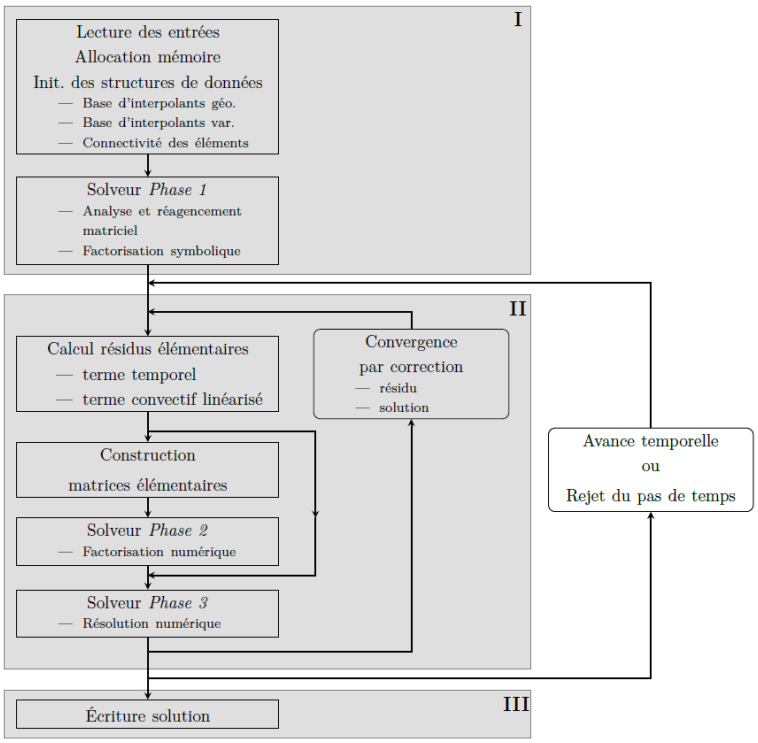


Figure 5.2 Organigramme des opérations du programme *EF* [2]

PRBSOL : Ce module encapsule toutes les informations qui définissent la solution en éléments finis. Elle ne contient pas seulement les vecteurs solutions, mais également la numérotation des DDLs, le maillage et l'ensemble des variables du problème. Cela permet de transmettre facilement ces informations aux autres modules.

PRBEQU : Les équations du problème sont définies dans cette section ainsi que les conditions aux frontières. De plus, toute la machinerie pour calculer les systèmes élémentaires s'y retrouve : éléments de référence, polynômes d'interpolation et quadrature d'intégration.

PRBSTA / PRBBDF : Ce module s'occupe du système d'équations ainsi que de sa résolution. L'algorithme d'assemblage s'y retrouve ainsi que les appels au solveur. La résolution du système se fait par correction du résidu avec un algorithme de Newton. Pour les problèmes instationnaires, la version PRBBDF utilise les formules de différenciations arrières (BDF) pour faire l'avance temporelle.

PRBPST : PRBPST est, dans son architecture, très similaire à PRBEQU. Il contient les équations à résoudre en post-traitement.

5.1.1 EF6

La 6^e version de *EF*, communément appelée *EF6*, tire avantage d'une parallélisation OpenMP dans les phases critiques. Cette API a été choisie car elle est facile à implémenter et permet une gestion dynamique de la charge de travail entre les threads. Néanmoins, l'usage d'OpenMP est limité aux architectures à mémoire partagée, généralement constituées de quelques dizaines de coeurs tout au plus. L'accélération obtenue est donc rapidement limitée par le nombre de processeurs disponibles. Il a été conclu, dans le mémoire de M. Moulin [2], que la transition vers une grappe de calcul serait nécessaire afin de mener à bien des études tridimensionnelles d'envergure. Ce passage nécessite cependant un changement de paradigme : la parallélisation MPI. Cette dernière requiert des modifications au programme plus invasives qu'avec OpenMP, où seulement certaines parties spécifiques du code sont parallélisées. Pour cette première tentative de version distribuée, notre stratégie repose sur l'utilisation de solveurs directs distribués, conservant ainsi la robustesse du programme. Les prochaines sections décriront donc les modifications apportées à *EF* pour y arriver, plus particulièrement vis-à-vis de la gestion du maillage, de la numérotation des DDLs et de la construction de la matrice globale.

5.2 La distribution du maillage

Le chapitre 4 a déjà démontré que la décomposition de domaines est à la base de la parallélisation de la méthode des éléments finis sur des architectures à mémoire distribuée. Cette section explicite plutôt les méthodes et les principales opérations utilisées dans *EF7* pour l'implémentation de cette décomposition. L'algorithme développé permet de distribuer un maillage en sous-domaines contigus et de renuméroter les nœuds de façon à obtenir des structures ordonnées et à faciliter les communications MPI. Dans un premier temps, les différentes classes développées seront exposées ainsi que leurs tâches et responsabilités respectives. Ensuite, un exemple simple permettra de visualiser l'algorithme de distribution du maillage étape par étape.

5.2.1 Le module de maillage

Dans *EF6*, le maillage est géré par trois classes : *Domaine*, *PartitionGéométrique*, *TopologieÉlémentaire*. *Domaine* est la classe de plus haut niveau et possède toutes les informations et structures de données concernant les coordonnées des nœuds géométriques ainsi que la connectivité des éléments. Les éléments appartenant à un même objet physique (au sens donné par Gmsh) sont regroupés dans une instance de *PartitionGéométrique*. Cette dernière

n'est qu'un "handle" [1] que l'on donne à un autre objet afin qu'il voit et manipule uniquement les éléments de ce groupe. La troisième classe, *TopologieÉlémentaire*, contient des informations sur le type d'élément employé ainsi que les polynômes d'interpolation et les quadratures pouvant être utilisées.

Deux nouvelles classes ont été ajoutées dans *EF7* pour implémenter la distribution du maillage. La classe abstraite *GraphManager* permet d'encapsuler les logiciels de partitionnement de graphes. Elle est responsable de convertir la connectivité du maillage en un graphe d'adjacence des éléments et de partitionner ce graphe. *EF7*, pour le moment, ne possède qu'une interface avec Parmetis mais pourra supporter d'autre logiciels dans les versions ultérieures. La classe *PartitionManager* est responsable de la majorité des opérations liées au partitionnement. Dans un premier temps, elle estime la charge de travail et les coûts de communications, arguments essentiels pour une redistribution optimale des éléments. Dans un second temps, elle est responsable de trouver les nœuds aux frontières des partitions virtuelles et de choisir un propriétaire unique pour ces nœuds. Conceptuellement, ces deux classes pourraient être intégrées dans la classe *Domaine*. Toutefois, en raison de leurs tailles imposantes, il était préférable de les implémenter séparément.

Finalement, quelques méthodes ont été ajoutées à la classe *Domaine* pour effectuer des opérations sur les structures de données telles que les opérations de communications, de lecture et d'écriture du maillage ainsi que la renumérotation des nœuds.

5.2.2 L'algorithme

La séquence d'opérations, depuis la lecture du maillage jusqu'à la renumérotation des nœuds, est décrite en s'appuyant sur l'exemple d'un maillage carré, constitué de quadrangles à 9 nœuds, que l'on partitionnera en trois sous-domaines. Les éléments du maillage sont identifiés à la figure 5.3a. Le domaine Ω est recouvert par les éléments 12 à 20, tandis que les numéros à l'extérieur du carré représentent les éléments de bords situés à la frontière Γ. La numérotation des nœuds est illustrée à la figure 5.4a.

La construction du domaine géométrique débute par la lecture du fichier de maillage. Présentement, puisque cette étape n'est effectuée qu'une seule fois au début du programme, EF6 peut se permettre une lecture séquentielle d'un fichier maillage en format ASCII. Dans un environnement distribué, il serait possible de conserver cette méthodologie mais plusieurs inconvénients pourraient survenir pour des problèmes d'envergure.

1. En informatique, un *handle* est un objet abstrait donnant accès à des ressources. Il s'agit d'une généralisation du concept de pointeur.

— Une phase de communication distincte est nécessaire, afin de distribuer les informations sur tous les processus.

— Si plusieurs unités de lecture sont disponibles sur la grappe de calcul, il n'y en a qu'une seule qui sera utilisée.

— La mémoire d'un nœud de calcul n'est peut-être pas suffisante pour stocker le maillage en entier.

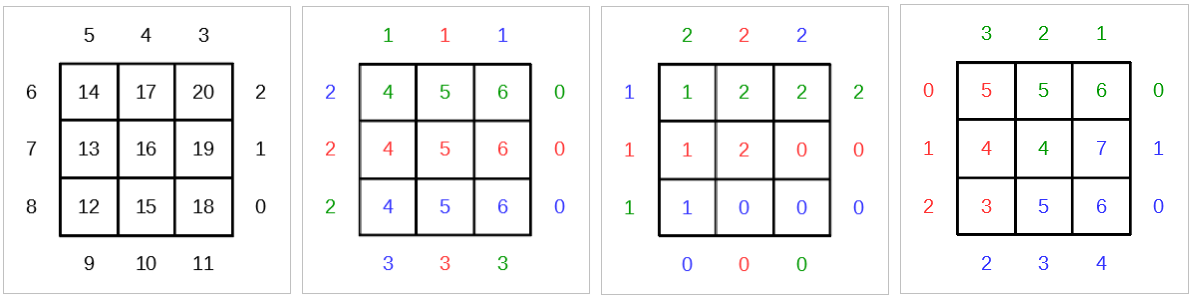
Une lecture séquentielle pourrait donc devenir un goulot d'étranglement pour EF7, tout particulièrement si l'opération est répétée plusieurs fois dans un contexte d'adaptation de maillage. Les méthodes de lecture ont donc été modifiées afin d'utiliser les routines des bibliothèques MPI présentées à la section 3.2.1. Pour ce faire, certaines modifications ont été apportées aux fichiers de maillage. Ces derniers sont désormais encryptés en binaire ; cela a pour avantage de réduire la taille du fichier et d'accélérer sa lecture. De plus, le format du fichier requiert maintenant un en-tête contenant le nombre de nœuds et d'éléments dans chaque partition. Ces informations permettent d'allouer, au préalable, la mémoire des structures de données.

Distribution des éléments

Après l'en-tête, la connectivité du maillage est lue dans un ordre arbitraire. Dans le cas de notre exemple, les processus lisent les éléments à tour de rôle [2]. La distribution des éléments est illustrée à la figure 5.3b, où les éléments d'une même couleur ont été lus par un même processus. Notons que sur un processus, nous utilisons une numérotation locale des éléments. Ainsi, on voit qu'ils ont sept éléments chacun.

Préalablement au partitionnement, le graphe dual est généré par une fonction de Parmétis. Ensuite, des poids liés aux nœuds et aux arêtes peuvent être attribués pour un partionnement optimal. En pratique, il est difficile de déterminer avec précision la charge de travail et les coûts de communications qu'un élément engendrera dans le reste du programme. En première approximation, la taille du système élémentaire, étant proportionnelle au nombre d'opérations arithmétiques effectuées lors de l'assemblage, est représentative de la charge de travail. Pour chaque équation, la taille des systèmes élémentaires est additionnée et assignée aux poids situés aux nœuds. En ce qui concerne les arêtes, le coût de communication entre deux éléments peut être approximé par le nombre de degrés de liberté en commun. Puisque cette recherche est légèrement plus complexe et que la qualité du partitionnement n'est pas un facteur limitatif jusqu'à présent, l'attribution de poids aux arêtes n'a pas été implémentée.

2. Un ordonnancement de style "Round-Robin".



(a) Maillage global initial

(b) Maillage local initial

(c) Identification des nouveaux processus propriétaires

(d) Maillage local partitionné

Figure 5.3 Résultats intermédiaires du partitionnement des éléments du maillage.

Les couleurs indiquent sur quel processus l'information réside :
0 - Bleu , 1 - Rouge , 2 - Vert

Une fois tous ces paramètres calculés, la routine de partitionnement est appelée. Cette dernière retourne un vecteur indiquant, pour chaque élément, le numéro du processus sur lequel il doit être envoyé (figure 5.3c). La connectivité de ces éléments est alors transmise, via des routines de communications collectives MPI *AllToAllv*, aux processus respectifs [3]. On obtient alors, comme en témoigne la figure 5.3d, une décomposition en sous-domaines contigus. Il est important de noter qu'il n'y a aucune garantie, pour les éléments de bord, de résider sur le même processus que l'élément du domaine auquel il se rattache. Par exemple, les éléments de bord 8 et 9 du maillage global initial (figure 5.3a) sont situés sur les côtés de l'élément 12. Toutefois, après partitionnement (figure 5.3d), le numéro 9 est placé sur le processus 0 (bleu), alors que les deux autres se retrouvent sur le processus 1 (rouge). Ce comportement complique l'imposition des conditions frontières, et les prochaines sections discuteront des méthodes appropriées pour gérer cette situation.

Distribution des nœuds

Maintenant que les éléments sont bien distribués, les nœuds doivent être attribués sur leur processus respectif. De plus, les nœuds aux frontières des partitions étant partagés par plusieurs processus, il est nécessaire de déterminer lequel sera propriétaire du nœud et lesquels seront locataires.

L'algorithme débute avec la lecture des coordonnées des nœuds [4]. Chaque processus lit une

3. Algorithmes 18 et 19 à l'annexe C.

4. Si des routines de lecture MPI non bloquantes sont utilisées (voir section 3.2.1), cette étape peut se faire simultanément aux manipulations précédentes sur les éléments.

suite de $n_g^{(p)} \approx N/P$ nœuds géométriques[5], où N et P sont le nombre de nœuds et de processus, de manière à ce que l'ensemble J des indices des nœuds d'un processus p est défini par (5.1).

$$J^{(p)} \equiv \left\{ j : \sum_{i=1}^{p-1} n_g^{(i)} \leq j \leq \sum_{k=1}^{p} n_g^{(i)} \right\}, \quad \text{pour } 1 \leq p \leq P \tag{5.1}$$

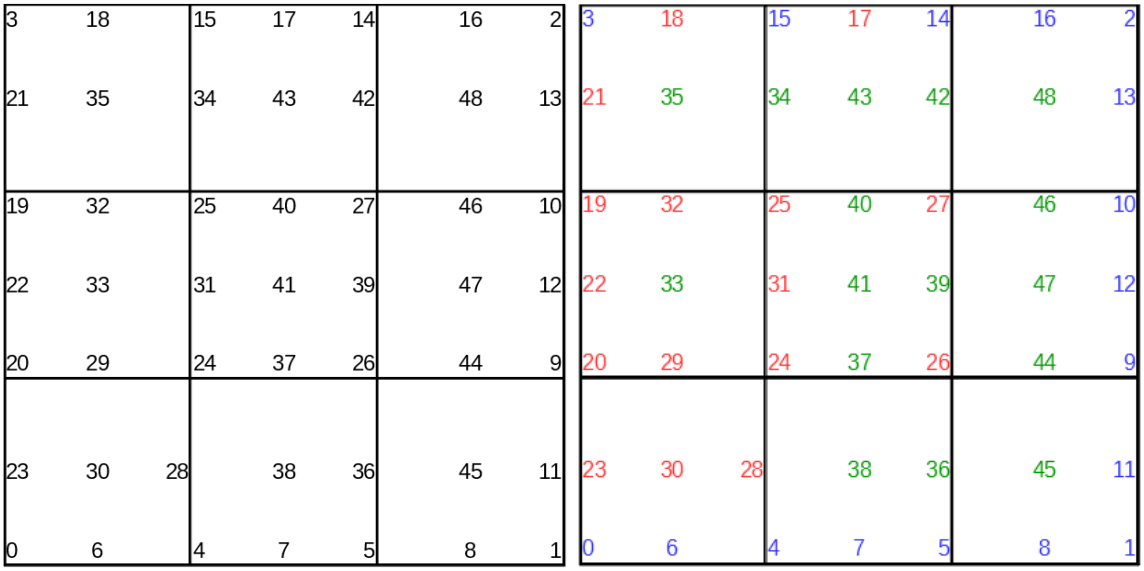
Dans le cas de notre exemple, on obtient la distribution de nœuds illustrée à la figure 5.4b. Bien que cette méthode n'attribue pas les nœuds aux bons processus, elle permet une lecture efficace et facilite la recherche des nœuds aux frontières des partitions virtuelles.

Afin de trouver les nœuds dupliqués, la liste ordonnée des nœuds locaux est créée sur chaque processus à partir de la connectivité (figure 5.4c). Chaque numéro de nœud est envoyé, par une communication *AllToAllv*, sur le processus qui possède ses coordonnées. À la réception des messages, les nœuds sont associés avec le numéro du processus émetteur (figure 5.5c), et sont alors triés à nouveau[6]. Comme il est mis en évidence dans la figure 5.5d, cette manière de procéder permet d'identifier rapidement les nœuds dupliqués, soit les nœuds aux frontières. L'heuristique pour choisir le processus propriétaire est arbitraire, puisque son impact sur la distribution de la charge de travail est estimé négligeable. Nous choisissons alors le processus du premier nœud dans la séquence de duplicatas. La répartition des nœuds maîtres et fantômes est explicitée à la figure 5.4d.

La liste des nœuds maîtres est transférée au processus propriétaire alors que les nœuds fantômes et le rang des propriétaires associés sont renvoyés vers les processus locataires. L'algorithme 3 résume toutes les opérations effectuées sur les nœuds jusqu'à présent. Notons que les coordonnées ne sont pas encore distribuées, car il faut d'abord procéder à la renumérotation des nœuds.

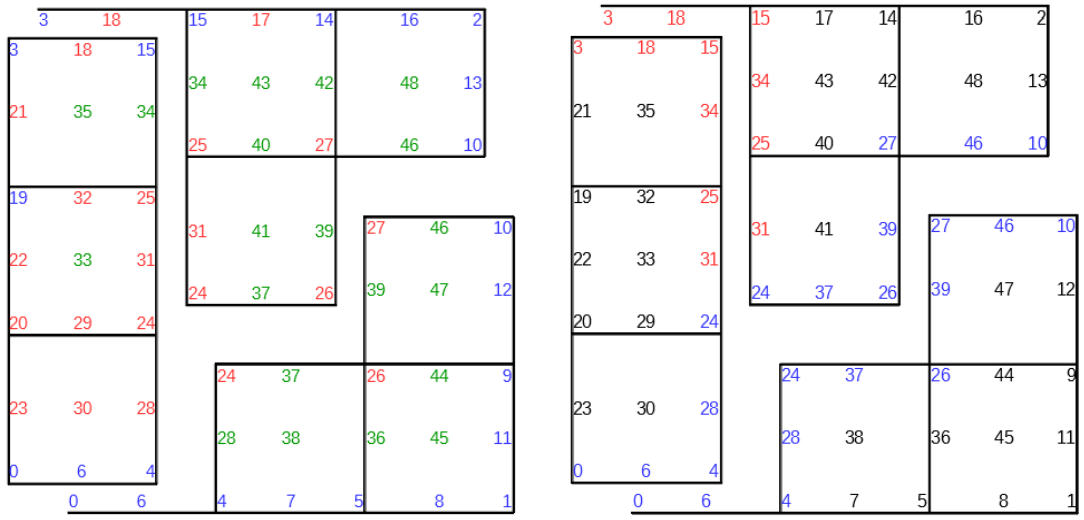
5. Si le nombre de nœuds n'est pas un multiple de P, alors les $R = NmoduloP$ premiers processus lisent un nœud supplémentaire.

6. Algorithmes 20 et 21 à l'annexe C.



(a) Maillage global initial

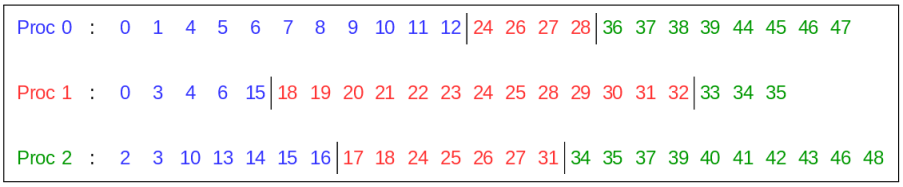
(b) Distribution des coordonnées à la lecture



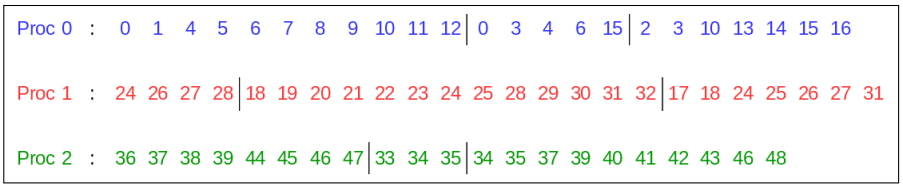
(c) Représentation distribuée des coordonnées selon le partitionnement de la figure 5.3d

(d) Répartition des nœuds maîtres et fantômes. Les couleurs permettent d'identifier les processus propriétaires.

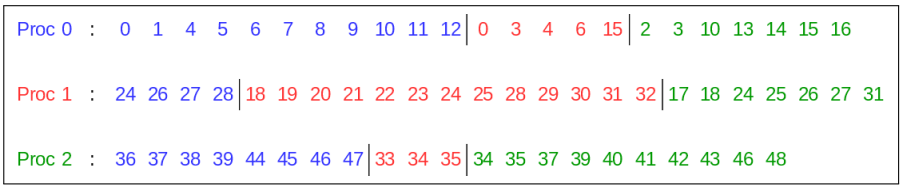
Figure 5.4 Partitionnement des nœuds à la lecture du fichier maillage



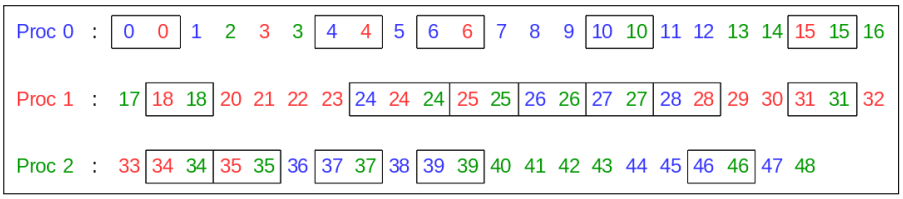
(a) Structure des nœuds à l'envoi des message MPI



(b) Structure des nœuds à la réception des message MPI



(c) Association des nœuds aux processus expéditeurs



(d) Liste des nœuds une fois triée

Figure 5.5 Représentation pas-à-pas des structures de données utilisées dans la recherche des nœuds frontières.

Algorithme 3 Partitionnement des nœuds géométriques

1: $\mathbf{c}$:= Connectivity table
2: $\mathbf{c} \leftarrow \texttt{std::sort}(\mathbf{c})$
3: $\mathbf{c} \leftarrow \texttt{std::unique}(\mathbf{c})$
4: $\mathbf{c}_i \leftarrow \left\{ c_j \mid c_j \in J^{(i)} \right\} \quad \forall i \in \mathcal{G}_w$
5: $[\mathbf{c}_i]_{i \in \mathcal{G}} \leftarrow T_{all}\left([\mathbf{c}_i]_{i \in \mathcal{G}}\right)$
6: $\mathbf{p}_i \leftarrow \{(c_j, i) \mid c_j \in \mathbf{c}_i\} \quad \forall i \in \mathcal{G}_w$
7: $\mathbf{p} = [\mathbf{p}_i]_{i \in \mathcal{G}}$
8: $\mathbf{p} \leftarrow \texttt{std::sort}(\mathbf{p})$ ▷ $\mathbf{p}$ is sorted by c_j first then by i
9: $\texttt{chooseMasterAndGhost}(\mathbf{p}) \rightarrow \begin{cases} \mathbf{m} := \textit{master nodes} \\ \mathbf{mo} := \textit{master nodes owners} \\ \mathbf{g} := \textit{ghost nodes} \\ \mathbf{go} := \textit{ghost nodes owners} \\ \mathbf{gt} := \textit{ghost nodes tenants} \end{cases}$
10: $\mathbf{m}_i \leftarrow \{m_j \mid mo_j = i\} \quad ; \mathbf{g}_i \leftarrow \{g_j \mid gt_j = i\} \quad ; \mathbf{go}_i \leftarrow \{go_j \mid gt_j = i\} \quad \forall i \in \mathcal{G}_w$
11: $\mathbf{m} \leftarrow T_{all}\left([\mathbf{m}_i]_{i \in \mathcal{G}}\right) \quad ; \mathbf{g} \leftarrow T_{all}\left([\mathbf{g}_i]_{i \in \mathcal{G}}\right) \quad ; \mathbf{go} \leftarrow T_{all}\left([\mathbf{go}_i]_{i \in \mathcal{G}}\right)$

Renumérotation des nœuds

Une étape du partitionnement consiste à établir une numérotation globale et locale des nœuds de manière à faciliter le passage d'une à l'autre ainsi que de simplifier les communications inter-processus. Pour ce faire, on choisit une numérotation ayant les propriétés suivantes :

$$\begin{cases} \mathcal{G}^{(p)} \equiv \left\{ g : \sum_{k=1}^{p-1} n^{(k)} \leq g \leq \sum_{k=1}^{p} n^{(k)} \right\} \\ \mathcal{L}_m^{(p)} \equiv \left\{ l_m : \ 0 \leq l_m \leq n_m^{(p)} \right\} \\ \mathcal{L}_r^{(p)} \equiv \left\{ l_r : \ 0 \leq l_r \leq n_r^{(p)} \right\} \\ \mathcal{L}_{f_k}^{(p)} \equiv \left\{ l_{f_k} : n_r^{(p)} + \sum_{k=1}^{p-1} n_{f_k}^{(p)} \leq l_{f_k} \leq n_r^{(p)} + \sum_{k=1}^{p} n_{f_k}^{(p)} \right\} \end{cases} , \ \forall k, p \in \mathcal{G}_w \tag{5.2}$$

où $\mathcal{G}^{(p)}$ et $\mathcal{L}^{(p)}$ sont les ensembles ordonnés des indices globaux et locaux du processus p. $\mathcal{L}_m^{(p)}$, $\mathcal{L}_r^{(p)}$ et $\mathcal{L}_{f_k}^{(p)}$ sont les ensembles des nœuds maîtres, réels et fantômes (dont le propriétaire est de rang k) et possèdent respectivement $n_m^{(p)}$, $n_r^{(p)}$ et $n_{f_k}^{(p)}$ indices chacun.

$$\begin{gathered} \mathcal{L}_f^{(p)} = \bigcup \mathcal{L}_{f_k}^{(p)}, \\ \mathcal{L}^{(p)} = \mathcal{L}_r^{(p)} \cup \mathcal{L}_f^{(p)}, \qquad \forall k \in \mathcal{G}_w \\ n_m = \sum n_m^{(k)}, \quad n_r = \sum n_r^{(k)}, \quad n_f = \sum n_f^{(k)}, \end{gathered} \tag{5.3}$$

Algorithme 4 Renumérotation des nœuds

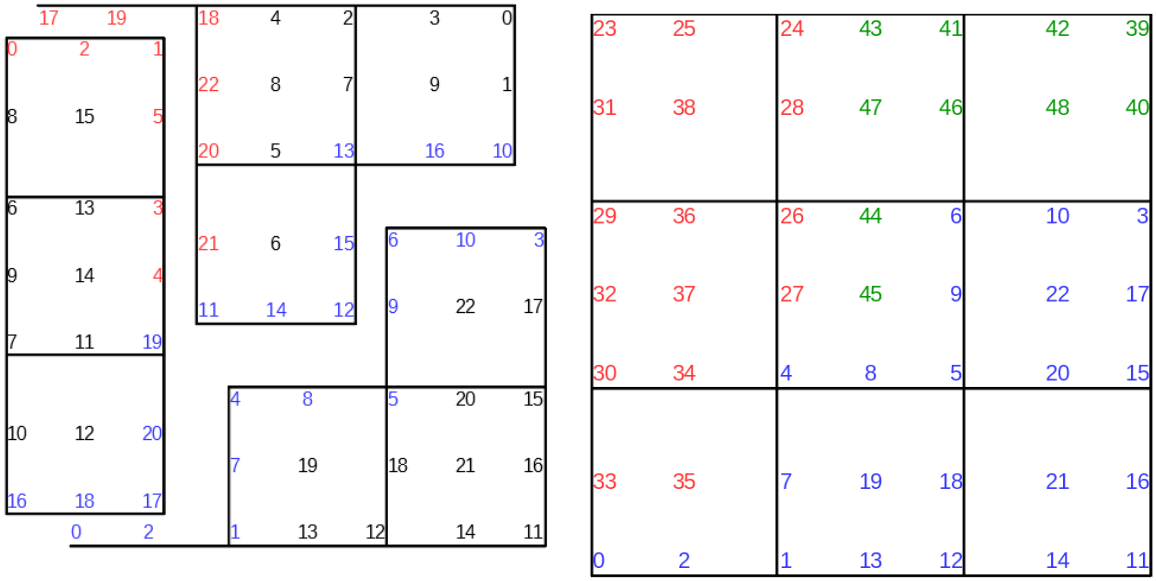
1: $\mathbf{m}$, $\mathbf{g}$:= master and ghost nodes from algorithm 3
2: $\mathbf{cnc}$:= connectivity table
3: $\mathbf{c}$:= list of all nodes in virtual partition
4: $p \in \mathcal{G}_{self}$

5: `masterFirst` := a unary predicate returning `true` if argument $x \in \mathbf{m}$
6: `ghostLast` := a unary predicate returning `false` if argument $x \in \mathbf{g}$
7: $\mathbf{c} \leftarrow$ `std::stable_partition`($\mathbf{c}$, `masterFirst`)
8: $\mathbf{c} \leftarrow$ `std::stable_partition`($\mathbf{c}$, `ghostLast`)
9: $\mathbf{c} \leftarrow [\mathbf{c}(\mathbf{s}), \mathbf{g}]$ ▷ $\mathbf{s} := \left\{0, \ldots, n_r^{(p)}\right\}$

10: $\mathbf{cnc} \leftarrow \mathbf{c}^{-1}(\mathbf{cnc})$

L'algorithme 4 présente une méthode pour effectuer cette renumérotation à partir des structures définies précédemment. La figure 5.6 permet de comparer les ensembles d'indices globaux et locaux.

Cette configuration permet aux nœuds réels de passer d'un repère local à global, et vice versa,



(a) Numérotation locale (b) Numérotation globale

Figure 5.6 Représentation des nœuds après renumérotation.

simplement avec l'ajout ou le retrait d'un index de décalage [7].

$$g^{(p)} = l_r^{(p)} + \sum_{i=1}^{p-1} n^{(i)} \tag{5.4}$$

Toutefois, il n'en est pas de même pour les nœuds fantômes, qui requièrent de conserver un tableau avec leurs indices globaux.

De plus, la disposition des nœuds fantômes décrite par l'équation (5.2) est parfaitement propice aux communications MPI, notamment celles de type *AllToAllv* (voir section 3.2.1) qui sont fréquement utilisées dans *EF7*. En effet, l'ensemble des nœuds fantômes $\mathcal{L}_f$ est construit sous la forme $[\mathcal{L}_{f_k}]_{k\in\mathcal{G}}$. Par conséquent, $\mathcal{L}_f$ est un argument valide pour une communication de type T_{all}. Toutefois, puisque $n_f \geq n_m$, les échanges d'information entre nœuds frontières ne sont pas bijectifs. On cherche alors à trouver une correspondance entre les nœuds maîtres et fantômes, qui une fois combinée à une communication T, engendre une transformation injective ou surjective.

Définition 5.1 (Le vecteur de correspondance **c**)
Soit un indice $l_{f_p}^{(k)} \in \left\{0, 1, \ldots, n_{f_p}\right\}$. *Le vecteur* $\boldsymbol{c}_k^{(p)}$ *représente la correspondance entre les*

7. Cet indice est obtenu facilement à l'aide d'une somme partielle (`MPI_Exscan`) du nombre de nœuds réels sur chaque processus.

nœuds maîtres du processus p et les nœuds fantômes du processus k tel que

$$l_m^{(p)} = \boldsymbol{c}_k^{(p)} \left(T_k^p(l_{f_p}^{(k)}) \right), \quad \forall p, k \in \mathcal{G}, \quad l_m^{(p)} \in \mathcal{L}_m^{(p)} \tag{5.5}$$

Pour les communications globales, on définit la structure distribuée $\boldsymbol{c} \equiv \left(\left[\boldsymbol{c}_k^{(p)} \right]_{k \in \mathcal{G}} \right)_{p \in \mathcal{G}}$

Cet opérateur est facilement construit par l'algorithme 5. Ceci nous amène à définir les concepts suivants :

Algorithme 5 Construction de **c**

1: **m**, **g**, **go** := master nodes, ghost nodes and ghost owners from algorithm 3
2: $\mathbf{g}_i \leftarrow \{g_j \mid go_j = i\} \ \forall i \in \mathcal{G}_w$
3: $[\mathbf{g}_i]_{i \in \mathcal{G}_w} \leftarrow T_{all} \left([\mathbf{g}_i]_{i \in \mathcal{G}_w} \right)$
4: $\mathbf{c}_i \leftarrow \mathbf{m}^{-1}(\mathbf{g}_i)$
5: $\mathbf{c} \leftarrow [\mathbf{g}_i]_{i \in \mathcal{G}_w}$

Définition 5.2 (Sous-ensemble $\mathcal{L}_{f_p}^{l_m}$)
Soit un indice $l_{f_p}^{(k)} \in \left\{0, 1, \ldots, n_{f_p}\right\}$. *Le sous-ensemble des nœuds fantômes* $\mathcal{L}_{f_p}^{l_m}$ *associés au même nœud maître est tel que*

$$\mathcal{L}_{f_p}^{l_m} \equiv \left\{ l_{f_p}^{(k)} \mid \boldsymbol{c}_k^{(p)} \left(T_k^p(l_{f_p}^{(k)}) \right) = l_m^{(p)}, \ \forall k \in \mathcal{G} \right\} \tag{5.6}$$

Définition 5.3 (Opérateur de réduction R)
Soit un ensemble de données $\boldsymbol{x}$ *et un sous-ensemble* $\mathcal{S} \subseteq \left\{0, 1, \ldots, n_{f_p}\right\}$. *Les échanges d'informations aux nœuds se font par l'intermédiaire d'un opérateur de réduction* $R_f^{(p)}$ *défini par*

$$R_f^{(p)} \left(\boldsymbol{x}(\mathcal{S}) \right) \equiv \left\{ \boldsymbol{x}(\mathcal{L}_m^S) \mid \forall \ l_m \in \mathcal{L}_m^S : \boldsymbol{x}(l_m) = f \left(\boldsymbol{x}(\mathcal{L}_{f_p}^{l_m}) \right) \right\} \tag{5.7}$$

où $f : \mathcal{X}^{\mathrm{card}(\mathcal{L}_{f_p}^{l_m})} \mapsto \mathcal{X}$. *Les informations des nœuds maîtres peuvent alors être actualisées par une opération de la forme :*

$$\boldsymbol{x}(\mathcal{L}_m^{(p)}) = R_f^{(p)} \left(\boldsymbol{x} \left(T_{all}(\mathcal{L}_{f_p}) \right) \right), \quad \forall \ p \in \mathcal{G} \tag{5.8}$$

La méthodologie introduite jusqu'à présent est à la base de la parallélisation MPI du programme. Elle permet de redistribuer équitablement un maillage dans un style *node-cut* et prépare les structures de données nécessaires pour le reste du programme, notamment pour la numérotation des DDLs.

Le programme poursuit avec un algorithme de coloration des éléments pour permettre une parallélisation multithread sans concurrence. Cette partie est déjà présente dans la version *EF6* ; la seule modification apportée est l'utilisation de *Parmétis* afin de former le graphe d'adjacence des éléments.

Dépendamment de la stratégie utilisée pour l'écriture du champ de solution et des résultats du post-traitement, des opérations supplémentaires sont requises. Si la nouvelle numérotation est utilisée, le fichier de maillage doit être réécrit, dans le cas contraire, on doit conserver en mémoire la numérotation initiale.

5.3 La numérotation des DDLs

En fait, la numérotation est une étape fondamentalement séquentielle et tenter d'en faire un algorithme distribué fait apparaître certaines problématiques :

— Il est nécessaire, pour numéroter un DDL, de connaître le nombre de DDLs déjà indexés. O r, a priori, un processus n'a pas d'information sur la numération des DDLs situés sur les autres processus.

— *EF* fait l'usage de DDLs globaux, soit des degrés de liberté qui ne sont pas strictement définis sur un nœud, mais sur l'ensemble du domaine. Ces DDLs doivent donc être visibles depuis tous les processus, mais comptabilisés qu'une seule fois.

— Un processus n'est plus garanti d'avoir toutes les équations afin de bien numéroter les DDLs locaux situés à la frontière des partitions virtuelles.

Un algorithme naïf consiste à numéroter les DDLs un processus à la fois. Lorsqu'un processus termine sa phase de numérotation, il communique son résultat final au prochain. La solution proposée dans cette section évite une numérotation séquentielle des DDLs. Pour ce faire, la numérotation est répétée deux fois ; la première pour numéroter avec des indices locaux et la deuxième pour corriger l'effet des autres processus.

Définition du problème test

Encore une fois, un problème manufacturé sera utilisé afin d'expliciter le fonctionnement de l'algorithme. Le cas de figure choisi est basé sur le maillage de l'exemple de la section précédente. Afin de démontrer que l'algorithme est robuste et général, le système d'équations suivant est imposé :

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u}\cdot\nabla)\,\mathbf{u}\right) = -\nabla p + \mu\nabla\cdot\left(\nabla\mathbf{u} + \nabla^t\mathbf{u}\right) \qquad \forall \mathbf{x}\in\Omega \tag{5.9}$$

$$\int_\Omega p\; d\Omega = 0 \qquad \forall \mathbf{x}\in\Omega \tag{5.10}$$

$$u_x = u = 0 \qquad \forall \mathbf{x}\in\Gamma_{droite}\cup\Gamma_{gauche} \tag{5.11}$$

$$u_y = v = 0 \qquad \forall \mathbf{x}\in\Gamma_{haut}\cup\Gamma_{gauche}\cup\Gamma_{bas} \tag{5.12}$$

Les équations imposées sur Ω sont imposées sur tous les quadrangles du maillage alors que celles imposées sur Γ s'appliquent aux éléments de bords.

La résolution de l'équation de Navier-Stokes fait apparaître les variables u, v et p. Ces dernières seront identifiées par les couleurs bleu, vert et rouge respectivement. La contrainte sur p nécessite l'usage d'un multiplicateur de Lagrange, λ, qui doit être traité comme une variable globale. Avec ces équations, cette distribution de conditions de Dirichlet et le partitionnement du maillage actuel, cet exemple démontrera que l'algorithme est apte à surmonter toutes les situations pathologiques discutées plus haut.

Des interpolants P2 en vitesse et P1 en pression seront utilisés. Bien qu'il serait plus pertinent de tester des interpolants d'ordres plus élevés, car plusieurs DDLs résideraient sur un même super-nœud, l'exemple deviendrait trop lourd pour être d'une quelconque aide visuelle.

5.3.1 L'algorithme

Pour la méthode des éléments spectraux (MES), où le nombre de DDLs est largement supérieur au nombre de nœuds géométriques, il est avantageux de ne pas conserver la liste des nœuds de calculs en mémoire. Ces derniers sont plutôt générés sur un élément lorsque nécessaire et immédiatement effacés ensuite. Afin de numéroter les DDLs, on introduit la notion de *super-nœud*. Un super-nœud est l'ensemble des nœuds de calculs $\tilde{\mathbf{x}}$ situés sur un sous-élément k^d tel que

$$\tilde{\mathbf{x}} \in \tilde{k}^d \equiv k^d \backslash K^{d-1} \tag{5.13}$$

où K^{d-1} est l'ensemble des sous-éléments de k^d. Dans *EF*, les super-nœuds sont associés aux nœuds géométriques[8]. Pour chacune des n_v variables j du problème, on définit, à un super-nœud i, un ensemble de DDLs $D_{i,j}$.

Les prochains paragraphes détaillent les étapes du décompte, de qualification et d'indexage qui forment l'algorithme de numérotation des DDLs. Chacune de ces étapes consiste à ini-

8. Il est donc nécessaire d'utiliser un maillage avec un seul nœud par sous-éléments k^d.

tialiser une des tables de numérotation de dimension $n_g \times n_v$. Tout au long de ces phases, l'algorithme sera appliqué sur un problème test permettant de visualiser les effets de chaque opération importante.

Le décompte

Le décompte a donc pour objectif d'identifier le nombre $n_{i,j}$ de DDLs se situant dans $D_{i,j}$. En séquentiel, puisque $n_{i,j}$ est défini par les interpolants des différentes équations, il suffit de parcourir ces dernières et d'enregistrer le décompte dans une table. En parallèle, il peut exister un couple (i,j) pour lequel $n_{i,j}^{(p)} \neq n_{i,j}$, car le processus p ne possède pas les éléments nécessaires pour attribuer une valeur exacte.

Ce phénomène est exposé à la figure 5.7a où le compte $n_{19,0}^{(2)} = 0$ est incorrect [9]. Du point de vue de ce processus, seule l'équation (5.12) est imposée sur le nœud puisque le quadrangle adjacent, où est appliquée l'équation de Navier-Stokes (5.9), réside sur le processus 1. Par conséquent, aucun DDL n'est attribué localement pour la variable u.

Cette problématique survient à l'interface des partitions géométriques B^g coïncidant avec l'interface des partitions virtuelles B^v lorsque que les équations ne définissent pas les mêmes variables pour les différents cotés de $B \equiv B^g \cap B^v$.

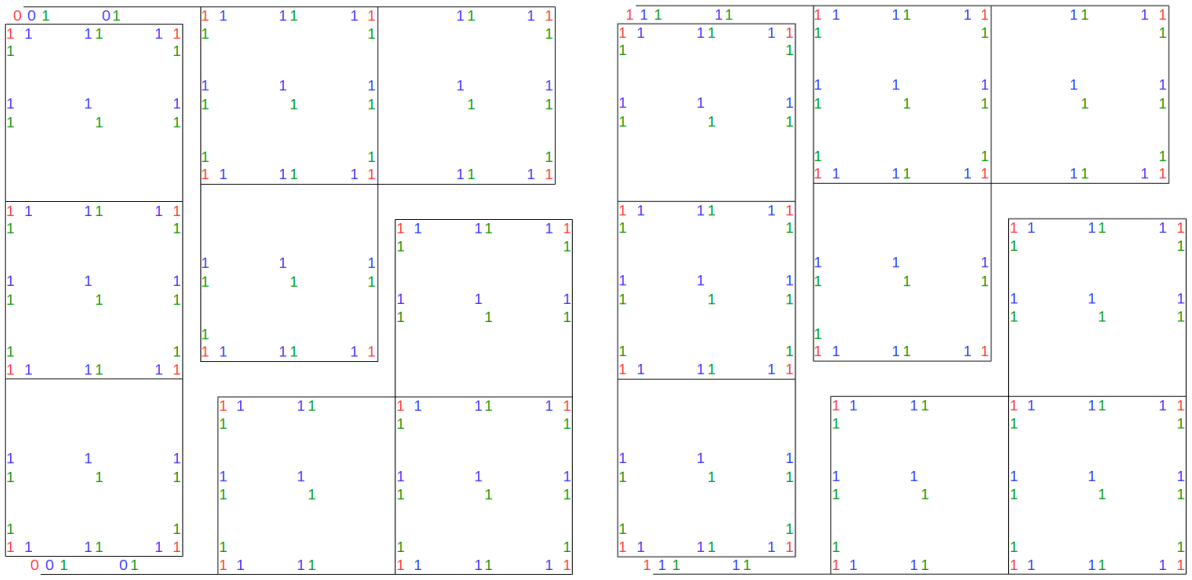
Pour corriger ce désaccord entre les différents processus, il faut d'abord déterminer quel est le véritable décompte et quel processus le détient. Ensuite, les données erronées doivent être actualisées.

Pour ce faire, tous les comptes associés à un même nœud sont rassemblés sur le processus propriétaire. Le véritable compte est nécessairement celui possédant la valeur la plus élevée. Finalement, le propriétaire peut renvoyer la valeur finale à ses locataires. Le résultat prévu (figure 5.7b) est alors obtenu par l'algorithme 6.

Algorithme 6 Le décompte

1: $\mathbf{n}_i := [n_{i,j}]_{j=1}^{n_v}$ ▷ Local numbers of DoF on super-node i for all variable j
2: $\mathbf{N} := [\mathbf{n}_i]_{i=1}^{n_g}$ ▷ Counts table
3: let $\mathbf{X}$ be a set of vectors $\mathbf{x}_i$. Define a function max that returns maximum values of column j such that $\max\{\mathbf{X}\} \rightarrow [\max\{\mathbf{x}_i(j)\}]_{j=1}^{n_v}$
4: $\mathbf{N}_g \leftarrow R_{\max}(T_{all}(\mathbf{N}(\mathcal{L}_f)))$
5: $\mathbf{N}(\mathcal{L}_m) \leftarrow \{\mathbf{n}_i \mid \forall i \in \mathcal{L}_m,\ \mathbf{n}_i = \max\{\mathbf{N}_g(i), \mathbf{N}(i)\}\}$
6: $\mathbf{N}(\mathcal{L}_f) \leftarrow T_{all}(\mathbf{c}^{-1}(\mathbf{N}(\mathcal{L}_m)))$

9. La position du nœud 19 est affichée à la figure 5.6a



(a) Nombre local de DDL par variable et par nœud

(b) Nombre global de DDL par variable et par nœud

Figure 5.7 Représentation des décomptes locaux et globaux

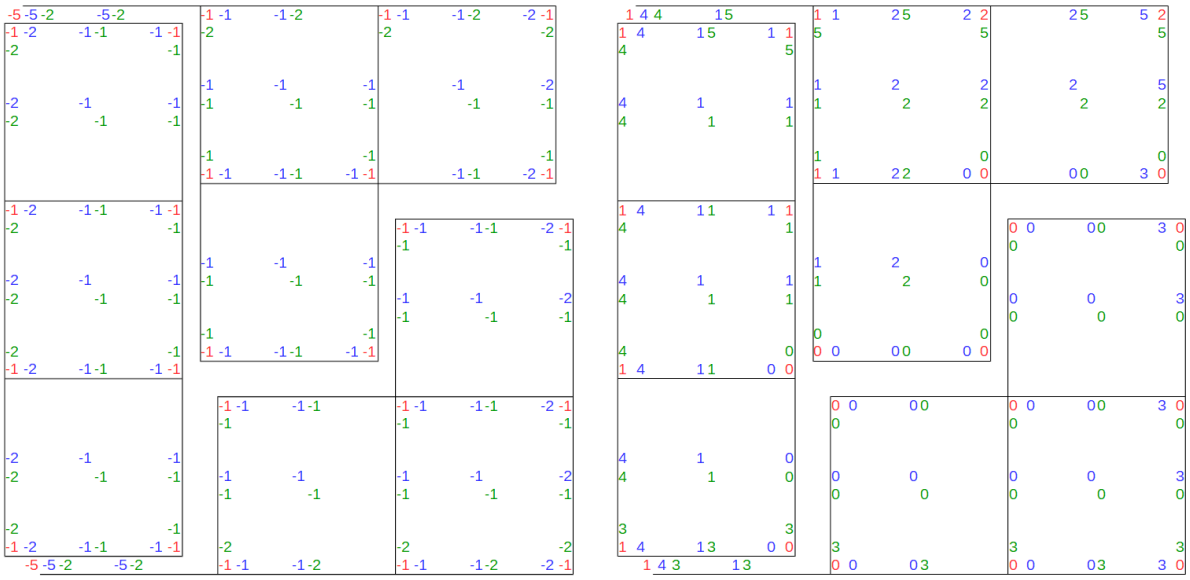
La qualification

La qualification consiste à étiqueter les DDLs à l'une ou l'autre des deux catégories : les inconnus ou les connus. Tel que présenté dans le chapitre 2.2, numéroter tous les DDLs inconnus en premier permet d'assembler le système matriciel plus facilement. L'implémentation utilisée dans *EF6* identifie les DDLs inconnus et connus avec des qualificatifs locaux de valence -1 et -2 respectivement.

$$\mathcal{Q}_L \equiv \{-1, -2\} \tag{5.14}$$

Pour l'algorithme distribué, une qualification locale n'est pas suffisante pour trois raisons. Premièrement, le problème d'incohérence entre processus de la section précédente survient de nouveau (figure 5.8a). Deuxièmement, il est nécessaire que l'ensemble des qualificatifs identifie les processus propriétaires des DDLs afin de numéroter ces derniers dans l'ordre défini par l'équation (5.16). Troisièment, le processus propriétaire d'un super-nœud doit être l'un de ceux qui possèdent la condition de Dirichlet. Dans le cas contraire, l'imposition des conditions essentielles nécessite deux communications collectives *AllToAll* au lieu d'une seule.

Un ensemble plus vaste de qualificatifs globaux $\mathcal{Q}_G$ est alors utilisé tel que



(a) Qualification locale (b) Qualification globale

Figure 5.8 Attribution locale et globale des qualificatifs

$$\begin{aligned} \mathcal{Q}_G &\equiv \mathcal{Q}_{G_i} \cup \mathcal{Q}_{G_c} \\ \mathcal{Q}_{G_i} &\equiv \{1, 2, \ldots, P-1, P\} \\ \mathcal{Q}_{G_c} &\equiv \{P+1, P+2, \ldots, 2P-1, 2P\} \end{aligned} \tag{5.15}$$

où $P = card(\mathcal{G}_{world})$. Un DDL inconnu est associé au qualificatif $q \in \mathcal{Q}_{G_i}$ représentant le rang du processus propriétaire. Un DDL connu est associé au qualificatif $q \in \mathcal{Q}_{G_c}$ de manière à ce que $q - P$ soit le rang du processus propriétaire.

Concrètement, la qualification locale est calculée dans un premier temps. Dans un second, les qualificatifs globaux sont définis à l'aide d'une fonction de qualification. Un exemple d'utilisation de cette fonction est donné à l'algorithme 7.

Définition 5.4 (Fonction de qualification)
Soit un tableau $\boldsymbol{Q}_{\mathcal{G}} = [\boldsymbol{q}_i]_{i \in \mathcal{G}}$ *de dimension* $card\{\mathcal{G}\} \times n_v$ *tel que* $q_{i,j} \in \mathcal{Q}_L$. *Pour chaque colonne* j, *on définit les ensembles* $Q^j_{-1} \equiv \{i \mid \boldsymbol{q}_i(j) = -1\}$ *et* $Q^j_{-2} \equiv \{i \mid \boldsymbol{q}_i(j) = -2\}$.

La fonction de qualification f_q *permet, pour chaque variable d'un super-nœud, d'assigner un qualificatif exprimant le rang du processus propriétaire et la nature des DDLs (connus ou*

inconnus) tel que

$$f_q\left\{\boldsymbol{Q}\right\} \rightarrow [q_j]_{j=1}^{n_v}, \quad q_j = \begin{cases} p_j, & si\ Q_{-2}^j = \emptyset,\ p_j \in Q_{-1}^j \\ p_j + card(\mathcal{G}_{world}), & si\ Q_{-2}^j \neq \emptyset,\ p_j \in Q_{-2}^j \end{cases}$$

Algorithme 7 La qualification

1: $\mathbf{q}_i := [q_{i,j}]_{j=1}^{n_v}$ ▷ $q_{i,j} \in \mathcal{Q}_L$
2: $\mathbf{Q} := [\mathbf{q}_i]_{i=1}^{n_g}$ ▷ Qualification table

3: $\mathbf{Q}_g \leftarrow R_{f_q}\left(T_{all}\left(\mathbf{Q}(\mathcal{L}_f)\right)\right)$
4: $\mathbf{Q}(\mathcal{L}_m) \leftarrow \left\{\mathbf{q}_i \mid \forall i \in \mathcal{L}_m,\ \mathbf{q}_i = f_q\left\{\mathbf{Q}_g(i), \mathbf{Q}(i)\right\}\right\}$
5: $\mathbf{Q}(\mathcal{L}_f) \leftarrow T_{all}\left(\mathbf{c}^{-1}\left(\mathbf{Q}(\mathcal{L}_m)\right)\right)$
6: $\mathbf{Q}(\mathcal{L}_{in}) \leftarrow \{\mathbf{q}_i \mid \forall i \in \mathcal{L}_{in},\ \mathbf{q}_i = f_q\left\{\mathbf{Q}(\mathcal{L}_{in})(i)\right\}\}$

L'indexage

Cette étape finale assigne à chaque degré de liberté, un indice spécifique afin d'obtenir une numérotation locale et globale telle que décrite à l'équation (5.16). $\mathcal{U}$ et $\mathcal{I}$ sont les ensembles d'indices globaux et locaux des DDLs inconnus alors que $\mathcal{K}$ et $\mathcal{C}$ sont les ensembles corrrespondant pour les DDLs inconnus.

$$\begin{cases} \mathcal{U}^{(p)} \equiv \left\{ g :\ \sum_{k=1}^{p-1} n^{i(k)} \leq g \leq \sum_{k=1}^{p} n^{i(k)} \right\} \\ \mathcal{I}_m^{(p)} \equiv \left\{ l :\ 0 \leq l \leq n_m^{i(p)} \right\} \\ \mathcal{I}_r^{(p)} \equiv \left\{ l :\ 0 \leq l \leq n_r^{i(p)} \right\} \\ \mathcal{I}_{f_k}^{(p)} \equiv \left\{ l : n_r^{i(p)} + \sum_{k=1}^{k-1} n_{f_k}^{i(p)} \leq l \leq n_r^{i(p)} + \sum_{k=1}^{p} n_{f_k}^{i(p)} \right\}, \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , \forall k, p \in \mathcal{G}_w \\ \mathcal{K}^{(p)} \equiv \left\{ g :\ n^i + \sum_{k=1}^{p-1} n^{c(k)} \leq g \leq n^i + \sum_{k=1}^{p} n^{c(k)} \right\} \\ \mathcal{C}_m^{(p)} \equiv \left\{ l :\ n^{i(p)} \leq l \leq n^{i(p)} + n_m^{c(p)} \right\} \\ \mathcal{C}_r^{(p)} \equiv \left\{ l :\ n^{i(p)} \leq l \leq n^{i(p)} + n_r^{c(p)} \right\} \\ \mathcal{C}_{f_k}^{(p)} \equiv \left\{ l : n^{i(p)} + n_r^{c(p)} + \sum_{k=1}^{k-1} n_{f_k}^{c(p)} \leq l \leq n^{i(p)} + n_r^{c(p)} + \sum_{k=1}^{p} n_{f_k}^{c(p)} \right\} \end{cases} \tag{5.16}$$

Tout comme la méthode séquentielle, les ensembles de DDLs inconnus et connus sont séparés, tant pour la numérotation globale que locale, de manière à ce que $l^i < l^c,\ \forall\ (l^i, l^c) \in \mathcal{I} \times \mathcal{C}$ et $g^i < g^c,\ \forall\ (g^i, g^c) \in \mathcal{U} \times \mathcal{K}$. Les DDLs des deux sous-ensembles sont ensuite indexés en fonction du qualificatif global pour finalement obtenir une structure parfaitement analogue à celle des nœuds géométriques. L'algorithme 8 résume les opérations nécessaires pour parvenir à une telle numérotation.

Algorithme 8 L'indexage

$\mathbf{N}$:= Counts table from algorithm 6
$\mathbf{Q}$:= Qualification table from algorithm 7
$\mathbf{I}$:= Indices table
$p \in \mathcal{G}_{self}$; $P \leftarrow \max\{\mathcal{G}_{world}\}$

$\mathbf{s}_j := \{i \mid \mathbf{Q}_i = j\}\ \ \forall j \in \mathcal{Q}_G$
$\mathbf{r}_{inc} \leftarrow [2, 3, \ldots, p-1, 1, p+1, \ldots, P]$
$\mathbf{r}_{con} \leftarrow [P+2, P+3, P+p-1, P+1, P+p+1, \ldots, 2P]$
$\mathbf{n}_q := \{n_j \mid n_j = \sum(\mathbf{N}(\mathbf{s}_j)),\ \ \forall j \in \mathcal{Q}_G\}$
$\mathbf{n}_q \leftarrow \mathbf{n}_q([\mathbf{r}_{inc}, \mathbf{r}_{con}])$
$\mathbf{d}_q := \left\{d_j \mid d_j = \sum_{i=1}^{j-1}(\mathbf{n}_q(i)),\ \ \forall j \in \mathcal{Q}_G\right\}$
$\mathbf{n}_q \leftarrow \vec{0}$

IdentifyDoF($\mathcal{L}_m$)
IdentifyDoF($\mathcal{L}_f$)
IdentifyDoF($\mathcal{L}_{in}$)

procedure IdentifyDoF($\mathcal{S}$)
    $\mathbf{n} = \mathbf{N}(\mathcal{S})$; $\mathbf{q} = \mathbf{Q}(\mathcal{S})$; $\mathbf{i} = \mathbf{I}(\mathcal{S})$;
    $\mathbf{s}_i \leftarrow \{s_j \mid \mathbf{q}(s_j) = i\}\ \ \forall i \in \mathcal{Q}_G$
    $\mathbf{n} \leftarrow \{n_i \mid n_i = \sum \mathbf{n}(\mathbf{s}_i),\ \ \forall i \in \mathcal{Q}_G\}$
    $\mathbf{i}(\mathbf{s}_i) \leftarrow \left\{u_j \mid u_j = \mathbf{d}_q(i) + \mathbf{n}_q(i) + \sum_{k=1}^{j} \mathbf{n}(\mathbf{s}_i)(j)\right\}\ \ \forall i \in \mathcal{Q}_G$
    $\mathbf{n}_q \leftarrow \mathbf{n}_q + \mathbf{n}$
    $\mathbf{N}(\mathcal{S}) = \mathbf{n}$; $\mathbf{Q}(\mathcal{S}) = \mathbf{q}$; $\mathbf{I}(\mathcal{S}) = \mathbf{i}$;
end procedure

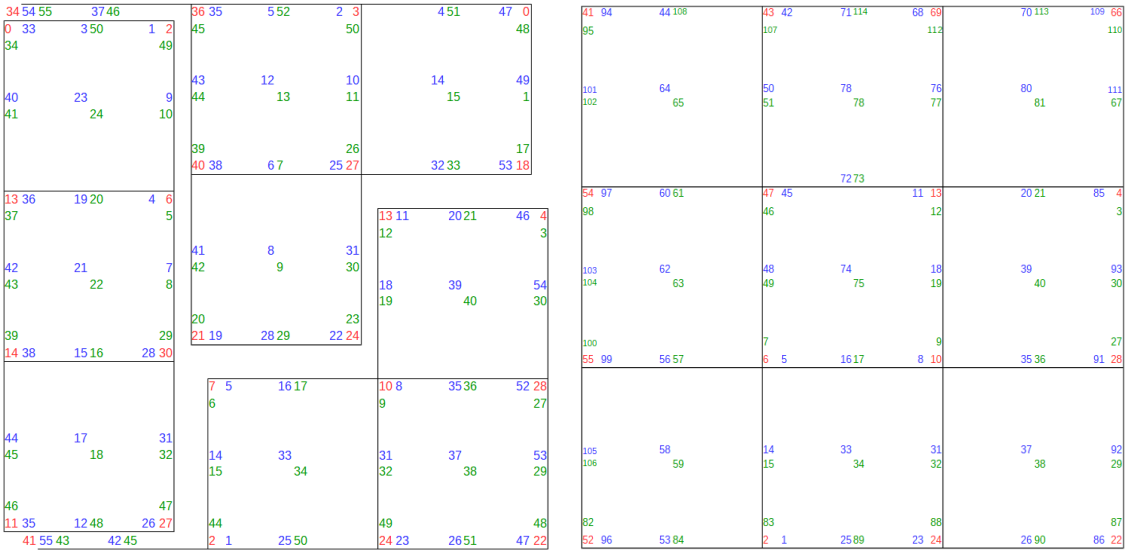
Il est important de souligner que la correspondance entre les DDLs maîtres et fantômes n'est pas la même que pour les nœuds géométriques. De plus, l'algorithme 5 ne peut être réutilisé, car, contrairement à la distribution du maillage, les indices globaux des DDLs fantômes sont encore indéterminés. Les algorithmes 9 et 10 présentent une ébauche de la séquence d'opération à effectuer pour retrouver des vecteurs de correspondances.

La numérotation locale de notre cas test est donnée à la figure 5.9a.

Algorithme 9 Correspondance entre DDLs maîtres et fantômes

N := Counts table from algorithm 6
Q := Qualification table from algorithm 7
I := Indices table
$p \in \mathcal{G}_{self}$; $P \leftarrow \max\{\mathcal{G}_{world}\}$

J := a table of size $n_m \times n_v$
for all $(i, j) \in \mathcal{L}_m \times \mathcal{V}$ **do**
  if $\mathbf{Q}(i)(j) = p$ or $P + p$ **then**
    $\mathbf{J}(i)(j) \leftarrow \mathbf{I}(i)(j)$
  end if
end for
$[\mathbf{I}_{f_k}]_{\forall k \in \mathcal{G}_w} \leftarrow T_{all}(\mathbf{I}(\mathcal{L}_f))$
for all $k \in \mathcal{G}_w$ **do**
  for all $(i, j) \in \left\{1 \dots n_{f_p}^{(k)}\right\} \times \mathcal{V}$ **do**
    if $\mathbf{Q}(i)(j) = k$ or $P + k$ **then**
      $\mathbf{J}(\mathbf{c}_k(i))(j) \leftarrow \mathbf{I}_{f_k}(i)(j)$
    end if
  end for
end for



(a) Numérotation locale (b) Numérotation globale

Figure 5.9 Représentation locale et globale de la numérotation des DDLs

Algorithme 10 Correspondance entre DDLs maîtres et fantômes (Suite)

$\mathbf{T}_m :=$ a 3D table of size $n_m \times n_v \times 3$
$\mathbf{T}_f :=$ a 3D table of size $n_{f_p} \times n_v \times 3$
for all $(i, j) \in \mathcal{L}_m \times \mathcal{V}$ **do**
    if $\mathbf{Q}(i)(j) \in \mathcal{Q}_{G_i}$ and $\mathbf{Q}(i)(j) \neq p$ **then**
        $\mathbf{T}_m(i)(j) \leftarrow [p,\ \mathbf{I}(i)(j),\ \mathbf{J}(i)(j)]$
    else if $\mathbf{Q}(i)(j) \in \mathcal{Q}_{G_c}$ and $\mathbf{Q}(i)(j) \neq P + p$ **then**
        $\mathbf{T}_m(i)(j) \leftarrow [p + P,\ \mathbf{I}(i)(j),\ \mathbf{J}(i)(j)]$
    end if
end for
for all $k \in \mathcal{G}_w$ **do**
    for all $(i, j) \in \left\{1 \dots n_{f_p}^{(k)}\right\} \times \mathcal{V}$ **do**
        if $\mathbf{Q}(i)(j) \in \mathcal{Q}_{G_i}$ and $\mathbf{Q}(i)(j) \neq p$ **then**
            $\mathbf{T}_f(i)(j) \leftarrow [k,\ \mathbf{I}(i)(j),\ \mathbf{J}(\mathbf{c}_k(i))(j)]$
        else if $\mathbf{Q}(i)(j) \in \mathcal{Q}_{G_c}$ and $\mathbf{Q}(i)(j) \neq P + p$ **then**
            $\mathbf{T}_f(i)(j) \leftarrow [k + P,\ \mathbf{I}_{f_k}(i)(j),\ \mathbf{J}(\mathbf{c}_k(i))(j)]$
        end if
    end for
end for
$\mathbf{T} \leftarrow [\mathbf{T}_m, \mathbf{T}_f]$
$\mathbf{T}_q \leftarrow \{\mathbf{t}_{i,j} \mid \mathbf{T}(i)(j)(1) = q\}$
$\mathbf{t}_{inc} \leftarrow T_{all}([\mathbf{T}_q]_{\forall q \in \mathcal{Q}_{G_i}})$
$\mathbf{t}_{con} \leftarrow T_{all}([\mathbf{T}_q]_{\forall q \in \mathcal{Q}_{G_c}})$
$\mathbf{t}_{inc} \leftarrow$ `std::sort`$(\mathbf{t}_{inc})$
$\mathbf{t}_{con} \leftarrow$ `std::sort`$(\mathbf{t}_{con})$
$\mathbf{c}_{inc} \leftarrow \{c_j \mid c_j = \mathbf{T}_{inc}(j)(3)\}$
$\mathbf{c}_{con} \leftarrow \{c_j \mid c_j = \mathbf{T}_{con}(j)(3)\}$

Les variables globales

À l'examen de la figure 5.9a, on s'aperçoit que les indices 0, 16, 25, sur les processus 0, 1 et 2 respectivement, sont manquants. Il ne s'agit pas d'une erreur ; ces DDLs ont bel et bien été numérotés. Ils correspondent au DDL global du multiplicateur de Lagrange. Ce dernier a été omis intentionnellement dans la figure 5.9, car il n'est pas localisé sur un nœud géométrique. Il est plutôt défini sur tout le domaine.

Pour *EF6*, il n'y a pas vraiment de distinction entre les variables globales et locales. L'unique différence est que les DDLs globaux sont arbitrairement imposés sur le nœud 0 afin de faciliter la construction des systèmes élémentaires. Ce procédé est simple, mais a le désavantage d'ajouter des colonnes vides dans les tables de numérotations.

Cette astuce n'est plus aussi simple avec la décomposition de domaine, car un nœud géométrique n'est pas défini sur tous les processus. Utiliser un nœud local par processus n'est pas non plus une option puisque les DDLs correspondants ne seront partagés que par les processus voisins. Il est donc nécessaire d'utiliser un autre support. C'est ici qu'entre en jeux le concept de nœud virtuel. Ces nœuds sont définis sur tous les processus et communiquent exclusivement entre eux. En effet, les nœuds virtuels doivent échanger avec tous les processus et non pas seulement leurs voisins. Jumeler les communications des deux types de nœuds priverait les DDLs locaux d'utiliser une topologie virtuelle (voir section 3.3).

Concrètement, ces nœuds virtuels représentent des entrées de plus au début des tables de numérotations. Le nombre de colonnes dans les tables correspond au nombre de variables locales. Ainsi, le nombre de nœuds virtuels nécessaires est donné par la division entière G/L où G et L sont le nombre de variables globales et locales respectivement. A priori, le fait d'avoir plusieurs nœuds virtuels est problématique, car on ne sait pas sur quel nœud réside les variables. Toutefois, la solution consiste à positionner les nœuds virtuels avant le début des tableaux. En réalité, des pointeurs vers le premier nœud géométrique sont conservés et tous les accès aux tableaux se font à partir de ces pointeurs. Ensuite, on identifie les variables globales avec des indices négatifs. L'accès à un DDL global se fait donc à partir du nœud local 0, avec un indice négatif, donnant l'impression de reculer avant le début du tableau (figure 5.10).

En définitive, la numérotation des DDLs, par les trois étapes proposées plus haut, constitue un algorithme distribué, générique et robuste.

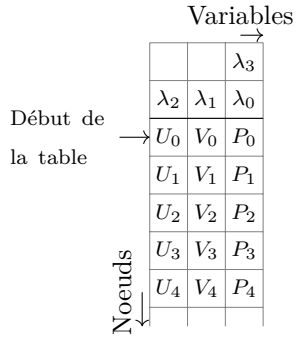


Figure 5.10 Format des tables de numérotation

5.4 Allocation et assemblage du système matriciel

Pour les programmes d'éléments finis, la phase d'assemblage du système matriciel est généralement une opération coûteuse. Des performances acceptables peuvent être obtenues uniquement si les structures de données de la matrice sont pré-allouées. Dans le cas contraire, une gestion dynamique de la mémoire requiert de recopier le contenu du système à chaque fois que l'espace mémoire est agrandi.

Pour un assemblage local, le nombre de coefficients présents dans chaque ligne de la matrice est facilement dénombrable. L'approche utilisé dans *EF6* consiste, dans un premier temps, à identifier pour chaque ligne, les éléments qui contribuent aux calculs de leurs coefficients. Dans un second temps, on parcourt les éléments d'une ligne pour compter le nombre de coefficients non nuls et on conserve également leur position dans la matrice. La structure creuse de la matrice est donc entièrement déterminée avant l'étape d'assemblage.

Cette méthode occasionne quelques difficultés lorsque le maillage est partitionné dans un style *node-cut* puisque les contributions élémentaires ne proviennent pas nécessairement d'éléments locaux. On cherche alors à mettre au point une démarche pour compter le nombre de coefficients d'une ligne avec un minimum de communications.

La difficulté, pour un processus locataire, est de déterminer si le coefficent d'une ligne fantôme est déjà comptabilisé par un autre processus. Quelques heuristiques ont été tentées mais il semble qu'aucune d'entre elles ne soient à l'abris d'un cas pathologique faussant le résultat.

Finalement, une approche plus naïve, mais robuste, a été adoptée. Elle consiste, pour les processus locataires, à transférer tous les coefficients fantômes (les indices des colonnes) au processus maître. Ce dernier peut ensuite calculer le nombre exact de coefficients non-nuls et pré-allouer la structure matricielle.

CHAPITRE 6 MODIFICATIONS ET OPTIMISATIONS DIVERSES

Considérant l'ampleur des modifications abordées au chapitre 5, visant à déployer *EF* sur des architectures à mémoire distribuée, la longue phase de développement s'est révélée être une bonne opportunité pour incorporer d'autres fonctionnalités.

En effet, *EF* peut désormais bénéficier d'instructions vectorielles pour accélérer la phase d'assemblage. De plus, un module a été implémenté afin de supporter l'usage de plusieurs solveurs. Les sections suivantes couvriront plus en détail ces deux composantes.

6.1 Construction des systèmes élémentaires par SIMD

Mise à part la résolution du système linéaire, le calcul des contributions élémentaires est l'opération la plus coûteuse du programme, particulièrement pour la MES. Les efforts de parallélisation sont donc concentrés à ce niveau. Jusqu'à présent, *EF* bénéficie d'une parallélisation OpenMP. Cette dernière, par une directive de boucle, redistribue les éléments parmi les threads. Ainsi, chaque thread peut construire simultanément le système élémentaire d'un élément. Pour *EF7*, cette parallélisation est conservée à l'intérieur de chaque processus.

Pour la MES, où la taille des systèmes élémentaires est considérable et le nombre d'éléments parfois restreint, une simple parallélisation au niveau des éléments n'est pas toujours optimale. La granularité des tâches est nécessairement plus grande que pour la MEF, et peut donc nuire au partage dynamique d'OpenMP. En revanche, il est possible d'exploiter, à l'intérieur d'un élément, un parallélisme au niveau des coefficients. En effet, le calcul d'une valeur dans la matrice élémentaire est entièrement indépendant des autres.

Une première tentative, présentée dans le mémoire de M. Moulin [2], consiste à introduire un second bloc parallèle imbriqué. Ainsi pour chaque élément, une sous-équipe de threads construit le système linéaire. Bien que cette méthode permet un contrôle plus fin de la distribution de la charge de travail, elle offre des gains notables uniquement lorsque le nombre d'éléments par couleur est inférieur ou du même ordre de grandeur que le nombre de CPUs disponibles. Or, en pratique, cette situation n'est que très rarement rencontrée.

Une alternative consiste à utiliser une parallélisation de type SIMD afin d'accroître les performances de la construction des systèmes élémentaires. Cette approche est utilisée en éléments finis classique pour traîter plusieurs éléments simultanément [22, 34]. Pour des interpolants d'ordre élevé, nous posons l'hypothèse que la taille des systèmes élémentaires est assez grande pour bénéficier d'instructions vectorielles afin de calculer plusieurs coefficients de la matrice.

En général, les compilateurs actuels vectorisent automatiquement les boucles et les segments de code propices à de telles optimisations. Toutefois, plusieurs facteurs peuvent nuire à cette autovectorisation :

— Les données d'une instruction ne sont pas indépendantes.
— Les tableaux de données ne sont pas alignés en mémoire.
— La boucle contient un branchement conditionnel ou un appel de fonction.
— Le nombre d'itérations d'une boucle n'est pas prédéterminé.
— Le segment de code est trop complexe pour le nombre de registres vectoriels.
— Les opérations contiennent des données de types différents.

Il n'est donc pas suggéré de faire aveuglément confiance au compilateur. En fait, à la compilation, il est possible de produire un rapport d'optimisations [1] afin de vérifier si le code désiré a bel et bien été vectorisé. Dans notre cas, nous avons constaté que la construction n'était pas optimisée. Il existe alors deux approches afin d'introduire manuellement une parallélisation SIMD. La première consiste à programmer directement en assembleur avec des instructions vectorielles. Certes, cette méthode est la plus efficace, mais requiert des programmeurs chevronnés. La deuxième, plus accessible, consiste à utiliser certaines instructions afin de guider le compilateur lors de l'optimisations. À l'aide de l'API OpenMP, nous empruntons la seconde approche.

En effet, depuis la version 4.0, OpenMP offre une interface pour facilement vectoriser un bloc de code. La directive `simd`, dont la syntaxe est présentée ci-dessous, et quelques clauses particulières sont à la base de cette interface.

```
#pragma omp simd [clause[ [ , ] clause] ... ] new-line
    for-loops
```

Cette directive, au même titre que `omp for`, permet d'exécuter plusieurs itérations de la boucle simultanément. Le nombre d'itérations concurrentes correspond à la longueur du vecteur utilisé (chaque itération correspond à un élément du vecteur). Si l'architecture informatique supporte plusieurs types d'instructions vectorielles, la clause `simdlen` permet d'indiquer la longueur de vecteur à utiliser. La clause `aligned` spécifie au compilateur l'alignement des tableaux de données. Tout comme avec les threads, le modèle de mémoire d'OpenMP offre un espace privé avec la clause `private`. Dans un paradigme SIMD, une variable déclarée privée est dupliquée pour chaque élément du vecteur. Cela est particulièrement utile pour traiter, vectoriellement, des opérations avec des scalaires.

1. Avec les compilateurs Intel, il suffit de rajouter l'option de compilation `-qopt-report=[0:5]` ou `-vec-report`.

Puisque l'instruction est de taille fixe, si le nombre d'itérations n'est pas un multiple de la longueur du vecteur, le compilateur doit traiter les itérations restantes différemment ; soit elles ne sont tout simplement pas vectorisées, soit un masque est appliqué faisant en sorte que les itérations superflues sont considérées comme des données factices. Le même phénomène survient si les tableaux de données ne sont pas alignés en mémoire ; les premières itérations sont traitées séparément.

Une ébauche d'une fonction calculant les coefficients d'une matrice élémentaire est présentée à l'algorithme[2] 11. Remarquons que, dans l'implémentation actuelle, seule la dernière boucle imbriquée est vectorisée. Même si la clause `collapse` est disponible, la vectorisation de plusieurs boucles implique, dans notre situation, que certains registres vectoriels soient constitués d'objets non contigus en mémoire, atténuant ainsi les gains de performance.

Algorithme 11 Vectorisation OMP

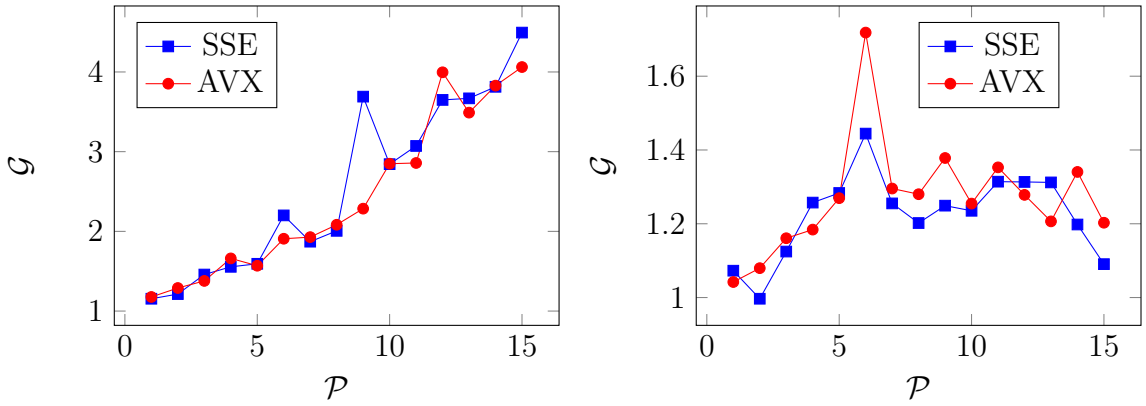
```
 1: procedure BUILDMATRIX(interpolants, parameters, elementary matrix)

 2:     Initialize variables' interpolating polynomial
 3:     Initialize parameters' function

 4:     for each quadrature's point do
 5:         Interpolate variables' value with their derivatives
 6:         Evaluate parameters' function

 7:         for each elementary equation do
 8:             Initialize equation dependent constants

 9:             #pragma omp simd private(constants) aligned(arrays:alignment)
10:             for each elementary variable do
11:                 Compute matrix coefficients
12:             end for
13:         end for
14:     end for
15: end procedure
```

Dans *EF7*, deux approches sont possibles pour construire les matrices élémentaires : évaluer l'expression exacte comme à l'algorithme 11 ou calculer le jacobien par différences finies, où chaque colonne de la matrice est obtenue par perturbation du résidu. Les deux méthodes seront prises en compte pour cette analyse.

Les performances de la vectorisation sont évaluées, dans un premier cas, avec un problème

2. Le calcul du membre de droite possède la même forme, mais sans la dernière boucle sur les variables. La directive `simd` est alors au dessus de la boucle sur les équations.

de diffusion appliqué sur un maillage de 4096 éléments. Des temps d'exécution sont mesurés pour une plage de polynômes d'interpolation de degré 1 à 15. Pour chacun, une quadrature d'ordre $2\mathcal{P}$-2 est utilisée pour obtenir une intégration exacte, reflétant les conditions d'utilisations usuelles du programme. La figure 6.1 présente l'accélération obtenue sur le temps total d'assemblage pour des instructions de type SSE (128 bits) et AVX (256 bits), correspondant à des vecteurs de longueur 2 et 4.



(a) Expression exacte

(b) Jacobienne par différences finies

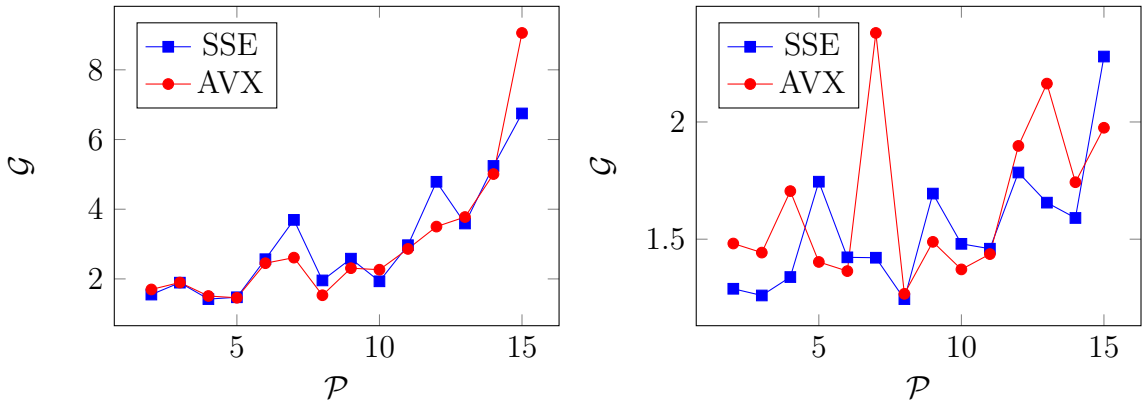
Figure 6.1 Performances de la vectorisation des méthodes de construction des systèmes élémentaires pour l'équation de diffusion. Les graphiques présentent l'accélération $\mathcal{G}$ en fonction des interpolants d'ordre $\mathcal{P}$.

La comparaison des figures 6.1a et 6.1b indique clairement que l'efficacité d'une vectorisation dépend de la méthode de construction des systèmes élémentaires. Pour l'évaluation de l'expression exacte, le gain semble suivre une relation linéaire avec le degré du polynôme. Ce comportement est attendu, car la fraction parallélisable augmente avec le degré. Toutefois, cette tendance n'est pas observée avec l'autre méthode, où le gain peine à dépasser les 40%. Les faibles performances du dernier cas de figure sont sans doute limitées par la proportion importante des opérations n'étant pas liées à l'évaluation du résidu. En effet, les temps d'exécution donnés à l'annexe D démontrent que, sans SIMD, le calcul par différences finies est typiquement deux à trois fois plus lent que le calcul de l'expression exacte. On peut donc conclure que la seconde méthode n'est pas adaptée à une telle parallélisation.

Le second cas d'étude (figure 6.2), faisant intervenir l'équation de Navier-Stokes, permet d'observer l'impact d'une formulation constituée de plusieurs variables couplées. Pour une telle formulation, la matrice élémentaire est définie par blocs (voir équation 6.1).

$$A = \begin{bmatrix} A_{UU} & A_{UV} & A_{UP} \\ A_{VU} & A_{VV} & A_{PP} \\ A_{PU} & A_{PV} & A_{PP} \end{bmatrix} \qquad B = \begin{bmatrix} B_U \\ B_V \\ B_P \end{bmatrix} \tag{6.1}$$

Une directive `simd` est alors associée à chaque bloc. La taille d'une sous-matrice n'est pas forcément un multiple de la taille du vecteur ; la fraction d'opération sur des données n'étant pas alignées en mémoire peut alors devenir considérable. On cherche donc à déterminer si le fait de scinder le système peut affecter les performances de l'assemblage. Les résultats illustrés à la figure 6.2a démontrent qu'un système par bloc bénéficie également de la parallélisation SIMD, mais le comportement n'est pas aussi linéaire que pour le problème de diffusion. L'accélération devient plus importante pour des polynômes d'ordre supérieur à 10.



(a) Expression exacte

(b) Jacobienne par différences finies

Figure 6.2 Performances de la vectorisation des méthodes de constructions des systèmes élémentaires pour l'équation de Navier-Stokes. Les graphiques présentent l'accélération $\mathcal{G}$ en fonction des interpolants d'ordre $\mathcal{P} - \mathcal{P}$-1.

Il faut également souligner que les accélérations observées sont parfois supérieures au facteur d'accélération théorique, soit la longueur des instructions SIMD utilisées. Sans analyse plus rigoureuse, il est difficile d'en identifier la cause avec certitude, mais nous supposons qu'une partie du gain provient d'une meilleure gestion des registres et des accès en mémoire centrale minimisant les fautes de cache.

Finalement, pour les exemples actuels, aucun des deux types d'instructions vectorielles ne semble prévaloir sur l'autre. Mais à la suite de ces résultats, il est évident qu'une vectorisation de la construction des systèmes élémentaires mène à des gains considérables en MES. L'API OpenMP s'est montré être un outil efficace et facile à utiliser. Il serait toutefois intéressant

de poursuivre cette étude avec d'autres architectures comme les Xéon Phi d'Intel, dotés d'un jeu d'instructions AVX512 (512 bits) ou même avec des GPGPUs.

6.2 Interface de système linéaire

Puisque le solveur Pardiso, utilisé par *EF6*, n'est pas distribué, l'ajout de la parallélisation MPI nécessite également l'ajout d'un nouveau solveur. Nous en avons donc profité pour mettre en place une interface afin de supporter plus d'un solveur. Il va sans dire que la possibilité de choisir dynamiquement un solveur adapté au problème donné est une caractéristique attrayante. Il y a toutefois un inconvénient : différents solveurs requièrent souvent des structures différentes en entrées. Cette interface permet donc de gérer tous ces branchements.

Initialement, dans *EF6*, le système matriciel et le solveur sont traités dans le même objet, indissociables l'un de l'autre. La fragmentation de ce module a fait apparaître une hiérarchie d'interfaces. Cette dernière est illustrée à la figure 6.3.

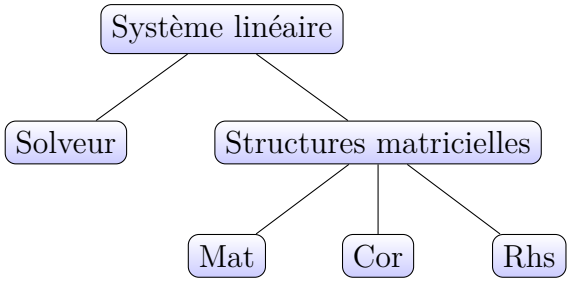


Figure 6.3 Module du système linéaire

Système linéaire : Le système linéaire englobe l'ensemble des interactions entre le solveur et le système matriciel. Il s'assure que les deux sont compatibles et toujours bien synchronisés. L'algorithme de Newton, pour traiter les équations non linéaires, s'effectue à l'intérieur de cette classe.

Solveur : Cette interface encapsule les appels aux différents solveurs. Pour qu'une implémentation soit valide, elle doit fournir trois appels pour l'initialisation, le préconditionnement et la résolution. Dans le cas de solveurs directs, on considère la factorisation matricielle comme un préconditionneur.

Structures matricielles : Le système matriciel s'assure de la cohérence entre les trois structures du système. Sa fonction principale est d'amorcer la construction des systèmes élémentaires et de procéder à l'assemblage. L'insertion des coefficients se fait par l'intermédiaire des sous-interfaces :

mat : Les différents formats de matrices distribuées y sont encapsulés.

rhs : Cette interface est responsable de l'assemblage du membre de droite (RHS) [3] ainsi que du vecteur des variables secondaires.

cor : La solution de la résolution du système matriciel y est stockée. Il ne s'agit pas du vecteur solution, mais d'une correction qui, à chaque itération, est ajoutée au vecteur solution.

Non seulement ces interfaces rendent le développement de nouvelles structures ou de solveurs plus facile, elles sont assez génériques pour interfacer avec d'autres bibliothèques. Par exemple, une version de ce module est actuellement implémentée pour accueillir les structures *PETSc*. Cela nous permet d'obtenir rapidement une grande variété de solveurs.

6.2.1 Insertion

Les différentes structures matricielles nous amènent à considérer plusieurs algorithmes d'insertion de coefficients. Les formats de matrice creuse sont généralement conçus pour minimiser l'espace mémoire requis. Néanmoins, l'accès à un coefficient arbitraire nécessite d'effectuer une recherche et est, par conséquent, plus dispendieux que pour une matrice dense. De plus, dans un paradigme distribué, où le maillage est partitionné dans un style *node-cut*, l'insertion doit gérer l'échange de contributions d'un processus à l'autre.

Insertion par recherche binaire

Une matrice de format *mpiaij* de *PETSc*, introduite à la section 4.2.1, est construite par l'insertion de blocs de coefficients avec la méthode `MatSetValues`. Cette dernière, ayant comme paramètres les indices globaux des lignes et colonnes, effectue, pour chaque coefficient, une recherche binaire afin de trouver la position correspondante dans la matrice du bloc diagonal ou du bloc hors-diagonal. L'algorithme 12 présente une ébauche de cette méthode.

Notons qu'une valeur, située sur une ligne n'appartenant pas au processus, n'est pas assemblée dans les matrices locales. Elle est plutôt ajoutée, temporairement, dans une pile avec tous les autres coefficients fantômes. À la fin de l'assemblage, la fonction `MatAssemblyBegin` parcourt cette pile et forme des messages non bloquant de type *point-to-point*. Ce mode de communication permet, à la réception, d'insérer les éléments d'un message sans attendre que toutes les communications soient complétées. Ensuite, le contenu de la pile est effacé.

Cette implémentation a été développée dans un environnement purement MPI et n'est pas

3. Nous utiliserons l'acronyme anglophone *Right Hand Side*

compatible avec une parallélisation OpenMP. En effet, même si le maillage est préalablement colorié de manière à éviter les situations de compétition, l'accès au pointeur du dessus de la pile est concurrentiel. Dans un paradigme hybride, il serait pertinent de rajouter un verrou sur ce pointeur.

En ce qui concerne la recherche binaire présentée à algorithme 13, cette dernière possède une complexité asymptotique de $\mathcal{O}(\log n)$ où n est le nombre de coefficients présents dans une rangée de la matrice. Ainsi, l'insertion d'une ligne de matrice élémentaire de taille m est de l'ordre de $\mathcal{O}(m \log n)$.

Algorithme 12 Insertion PETSc

```
1:  procedure addElementaryMatrix(nDoFI, nDoFJ, VADI, VADJ, elementMatrix)
2:      mat A_ ;                                          ▷ CSR matrix for diagonal bloc
3:      mat B_ ;                                  ▷ CSR matrix for off diagonal blocs
4:      stack S_ ;                                 ▷ Stack for off process coefficients

5:      for i ← 1 to nDoFI do
6:          row ← VADI[i] ;

7:          if row is locally owned then
8:              for j ← 1 to nDOFJ do
9:                  col ← VADJ[j] ;
10:                 if col is block diagonal then
11:                     Insert elementMatrix[i][j] in A_ with binary search ;
12:                 else
13:                     Insert elementMatrix[i][j] in B_ with binary search ;
14:                 end if
15:             end for
16:         else
17:             for j ← 1 to nDOFJ do
18:                 Insert elementMatrix[i][j] on top of S_ ;
19:             end for
20:         end if

21:     end for
22: end procedure
```

Compte tenu de l'impossibilité d'utiliser la bibliothèque *PETSc* dans un paradygme hybride, une version de matrice distribuée, basée sur le format DCSR, à été implémentée. La méthode d'insertion utilise une recherche binaire calquée sur celle de *PETSc*. De plus, les lignes fantômes sont également assemblées dans la matrice locale au même titre que les lignes réelles. Cette approche, conjointement avec un algorithme de coloration, permet d'assembler

Algorithme 13 Recherche binaire

```
1: procedure BINARYINSERT(row, nDoFJ, VADJ, values)
2:     rowPtr ← Aj_ + Ap_[row] ;
3:     valPtr ← Ax_ + Ap_[row] ;
4:     nNZRow ← Ap_[row+1] - Ap_[row] ;
5:     low ← 0; high ← nNZRow;
6:     lastcol ← -1 ;

7:     for j ← 1 to nDoFJ do
8:         if VADJ[j] is unknown then
9:             col ← VADJ[j] ;
10:            val ← values[j] ;

11:            if col <= lastcol then                          ▷ Reduce range
12:                low ← 0 ;
13:            else
14:                high ← nNZRow;
15:            end if
16:            lastcol ← col ;

17:            while high - low > 5 do                         ▷ Binary search
18:                t ← (high+low)/2;
19:                if rowPtr[t]>col then
20:                    high ← t ;
21:                else
22:                    low ← t ;
23:                end if
24:            end while

25:            for iNZ ← low,high do                           ▷ Linear search & insert
26:                if rowPtr[iNZ] == col then
27:                    valPtr[iNZ] += val ;
28:                    low ← iNZ+1 ;
29:                end if
30:            end for
31:        end if
32:    end for
33: end procedure
```

la matrice globale sans concurrence. Les lignes fantômes, étant déjà en ordre de processus propriétaire, sont prêtes à être envoyées par une communication de type *AlltoAllv*, c'est-à-dire qu'aucune copie mémoire n'est nécessaire pour former les messages. En revanche, à la réception, une deuxième recherche binaire est nécessaire.

Insertion groupée par recherche linéaire

En MES, les matrices sont généralement plus denses et ont, par conséquent, plus de coefficients non nuls par ligne. Il est alors justifié de se demander si la complexité logarithmique peut être néfaste sur les performances de l'assemblage. On propose alors un algorithme d'insertion de complexité quasi-constante.

Tel que présenté à l'algorithme 14, cette méthode requiert un vecteur, `posNZ_`, de taille N_{local}, soit la taille du système local, pour conserver en mémoire les positions exactes, dans la matrice, des non-zéros d'une ligne. La position d'un coefficient situé à la colonne j [4] est enregistré à la j^e case du vecteur `posNZ_`, offrant ainsi, avec le vecteur d'adressage, un accès direct au coefficient. L'algorithme consiste donc à parcourir les non-zéros d'une ligne pour actualiser `posNZ_` et ensuite, à ajouter, un à un, les contributions élémentaires.

Algorithme 14 Recherche linéaire

```
1:  procedure addElementaryMatrix(nDoFI, nDoFJ, VADI, VADJ, elementMatrix)
2:      Ap_, Aj_, Ax_ ;            ▷ CSR matrix : line pointers, column indices and values
3:      posNZ_ ;                                  ▷ Array of length N, the local system size

4:      for i ← 1,nDoFI do
5:          if VADI[i] is unknown then
6:              row ← VADI[i] ;
7:              for j ← Ap_[row],Ap_[row+1] do
8:                  posNZ_[Aj_[j]] ← j ;
9:              end for

10:             for j ← 1,nDoFJ do
11:                 if VADJ[j] is unknown then
12:                     Ax_[posNZ_[VADJ[j]]] += elementMatrix[i][j] ;
13:                 end if
14:             end for
15:         end if
16:     end for
17: end procedure
```

L'algorithme permet donc d'insérer m valeurs avec une complexité linéaire $\mathcal{O}(m+n)$. Pour

4. Indice local

une seule valeur, la complexité $\mathcal{O}(1+n/m)$ est alors dictée par le rapport n/m. En MES, où le support de la majorité des fonctions tests réside sur un seul élément, cette propriété est intéressante puisque le rapport n/m tend vers 1.

En contrepartie, cette approche n'est pas toujours applicable pour une matrice distribuée. En effet, l'utilisation d'indices globaux impliquerait que le vecteur `posNZ_` ait une taille de N_{global}, affectant la capacité de l'algorithme d'être appliqué à grande échelle. De plus, la conversion d'un indice global à local peut s'avérer coûteux [5]. Il est donc préférable de construire la matrice de format CSR avec les indices locaux et les convertir dans leurs équivalents globaux seulement à la fin de l'assemblage.

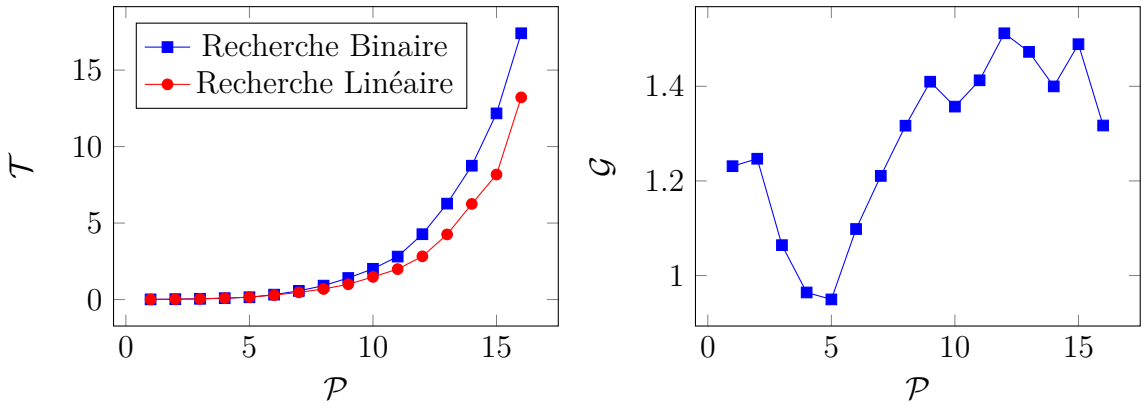
Notons que les coefficients d'une rangée d'une matrice CSR tel que demandé par *Pardiso*, doivent être en ordre croissant, or la conversion d'indices peut affecter cet ordre.

Performances des méthodes d'insertion

Selon les considérations précédentes, la recherche linéaire, bien que plus difficile à implémenter dans un environnement distribué, promet des performances accrues pour des systèmes élémentaires de grandes tailles. Nous tentons alors de déterminer si le gain est marginal ou s'il est pertinent de conserver cette méthode pour la MES.

Pour ce faire, nous mesurons le temps d'exécution de l'assemblage avec les deux méthodes, pour une gamme de polynômes d'interpolation. Les résultats affichés à la figure 6.4 démontre qu'il y a bel et bien un gain significatif pour des polynômes de degrée élevé.

5. Ce type de conversion est généralement accompli à l'aide d'une *map*, ayant un accès en $\mathcal{O}(\log(n))$ ou d'une table de hachage. Bien que cette dernière possède un temps d'accès constant, elle peut nécessiter plusieurs accès mémoire.



(a) Temps d'exécution $\mathcal{T}$ de l'assemblage pour les deux méthodes d'insertion

(b) Accélération $\mathcal{G}$ de la recherche linéaire par rapport à la recherche binaire

Figure 6.4 Comparaison des algorithmes d'insertion des contribution élémentaires pour des polynômes d'ordre $\mathcal{P}$. Les mesures proviennent d'un problème de diffusion 2D imposé sur un maillage de 4096 éléments

6.2.2 L'assemblage

Comme il a été mentionné précédemment, la classe *Structures matricielles* encapsule l'algorithme d'assemblage. Lors de la réécriture de l'algorithme pour la version *EF7*, quelques modifications ont été apportées.

Premièrement, la procédure d'assemblage de la matrice et du membre de droite ne requiert, désormais, que de boucler une seule fois sur les éléments. En effet, *EF6* implémente l'assemblage de la matrice et du RHS en deux fonctions distinctes. Le pseudo-code de l'une de ces fonctions est présenté à l'algorithme 15. Cette séparation est nécessaire, car la matrice n'est pas assemblée à chaque itération de Newton, contrairement au RHS. Toutefois, lorsque les deux structures sont assemblées, les fonctions sont appelées l'une après l'autre, faisant en sorte que les éléments soient parcourus à deux reprises. EF7 évite cette répétition en assemblant simultanément les deux structures.

Deuxièment, certaines équations, notamment celles faisant intervenir des variables globales, ne sont pas compatibles avec un assemblage parallèle, c'est-à-dire que l'insertion de contributions élémentaires occasionne des problèmes de concurrence. En effet, même si les éléments d'une même couleur sont disjoints, ils peuvent faire référence au même DDL global, donc à la même ligne du système global. Jusqu'à présent, *EF6* traite ces équations séquentiellement. La modification proposée pour EF7 consiste à séparer l'insertion des contributions associées

aux DDLs locaux de l'insertion pour les DDLs globaux. En fait, la solution est de conserver une variable privée par thread où les contributions globales sont accumulées. Une fois que tous les systèmes élémentaires ont été construits, on additionne ces variables avec une opération de réduction. Le thread maître, dans un bloc `single`, peut alors insérer la valeur finale dans le système linéaire sans concurrence.

Algorithme 15 Assemblage EF6

```
 1: procedure AssemblyMatrix
 2:     for ieq ← 1, nequation do                          ▷ Loop on equations
 3:         if equation is thread safe then
 4:             for icol ← 1, ncolor do                        ▷ Loop on colors
 5:                 #pragma omp parallel for iel ← 1, nelement  ▷ Loop on elements
 6:                     Initialize element
 7:                     Compute elementary matrix
 8:                     Insert elementary contributions into global matrix
 9:                 end omp parallel for
10:             end for
11:         else
12:             for icolor ← 1, ncolor do
13:                 for iel ← 1, nelement do                   ▷ Loop on elements
14:                     Initialize element
15:                     Compute elementary matrix
16:                     Insert elementary contributions into global matrix
17:                 end for
18:             end for
19:         end if
20:     end for
21: end procedure
```

Troisièment, l'assemblage se fait à l'intérieur d'une seule région parallèle. L'algorithme 15, en utilisant la directive *parallel* dans la dernière boucle imbriquée, doit créer et détruire un groupe de threads à répétition. Il est facile d'éviter ces opérations superflues en créant une région parallèle à l'extérieur des boucles sur les équations.

Quatrièmement, l'ordre des boucles de l'algorithme 15 n'est pas forcément optimal, puisqu'une barrière omp est nécessaire entre chaque couleur. Boucler sur les équations en premier et sur les couleurs ensuite engendre alors $nequation \times ncolor$ barrières. Cet ordre est imposé par le fait que l'algorithme de coloriage est appliqué sur chaque partition géométrique. Donc d'une équation à l'autre, les couleurs sont indépendantes. En réalité, rien n'empêche de colorier tout le maillage d'une partition virtuelle. Ainsi, les équations utilisent les mêmes couleurs et l'on peut alors interchanger les boucles dans l'algorithme d'assemblage. On peut donc diminuer le nombre de synchronisations d'un facteur *nequation*

Algorithme 16 Assemblage EF7

```
1: procedure AssemblyAll
2:     #pragma omp parallel
3:         [...]
4:         for icol ← 1, ncolor do                          ▷ Loop on colors
5:             [...]
6:             for ieq ← 1, nequation do                    ▷ Loop on equations
7:                 [...]
8:                 #pragma omp for iel ← 1, nelement        ▷ Loop on elements
9:                     Initialize element
10:                    Compute elementary matrix
11:                    Compute elementary RHS
12:                    Insert nodal elementary contributions into global matrix
13:                    Insert nodal elementary contributions into global RHS
14:                end omp for
15:                #pragma omp single
16:                    Insert global variable contributions into global matrix
17:                    Insert global variable contributions into global RHS
18:                end omp single
19:            end for
20:        end for
21:    end omp Parallel
22: end procedure
```

CHAPITRE 7 VÉRIFICATIONS

Puisque les deux derniers chapitres font mention d'une multitude de modifications apportées au coeur même du programme, il est essentiel d'ajouter, dans le cadre de ce mémoire, une phase de vérification. Ce chapitre introduit alors une suite de tests, basés sur des problèmes de diffusion et de mécanique des fluides, prouvant le bon fonctionnement de *EF7*.

7.1 Problème de diffusion

La première étape de cette vérification consiste à résoudre un problème simple pour confirmer le bon comportement des fonctionnalités de base du programme. Dans un premier temps, une analyse de convergence de l'erreur est réalisée pour valider la précision des solutions obtenues. Dans un second temps, l'influence du partitionnement est étudiée.

Inspiré des travaux de vérification et de validation de *EF6* [2], le problème choisi pour ces analyses est un problème de diffusion sur un domaine carré. Plus précisément, l'équation de Poisson (7.1) avec un terme source sinusoïdal est résolue.

$$-\Delta(u) = f \tag{7.1}$$

$$f(x,y) = -4\pi^2\left(2\sin(2\pi x)\sin(2\pi y) + 10\sin(20\pi y)\right) \tag{7.2}$$

La fonction arbitraire f (7.2) est construite, avec la méthode des solutions manufacturées, de manière à obtenir la solution u, illustrée à la figure 7.1. Cette dernière possède des termes sinusoïdaux avec des amplitudes et des fréquences éloignées afin que la fonction soit difficile à interpoler.

Cette analyse de convergence comporte deux volets : un pour la convergence H et l'autre pour la convergence P. Dans le premier cas, le problème est résolu à répétition avec une discrétisation spatiale de plus en plus fine. En utilisant une méthode de bissection pour le raffinement, les maillages ont entre 2^2 et 1024^2 éléments. Le comportement de l'erreur est alors comparé à celui dicté par l'équation (7.3).

$$\|U - U_{ex}\|_{L^2} = \epsilon_{L^2} \propto h^{P+1} \tag{7.3}$$

Des courbes d'erreurs (figure 7.2a) ont été mesurées pour des polynômes d'ordre 1 à 5. La

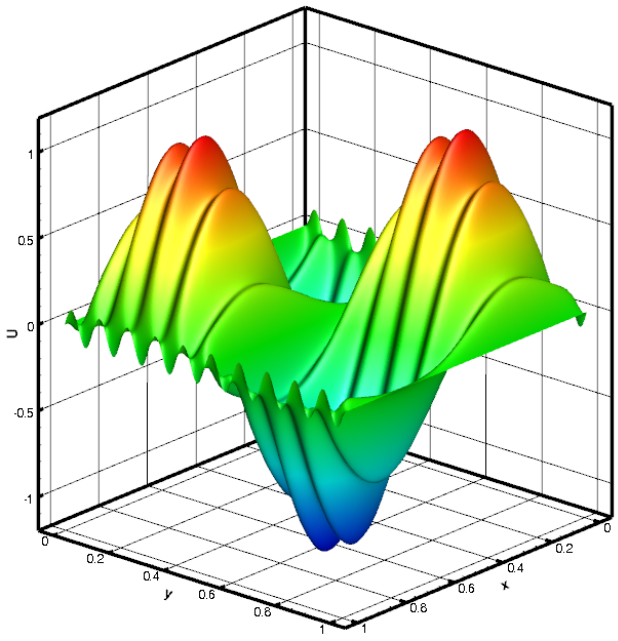


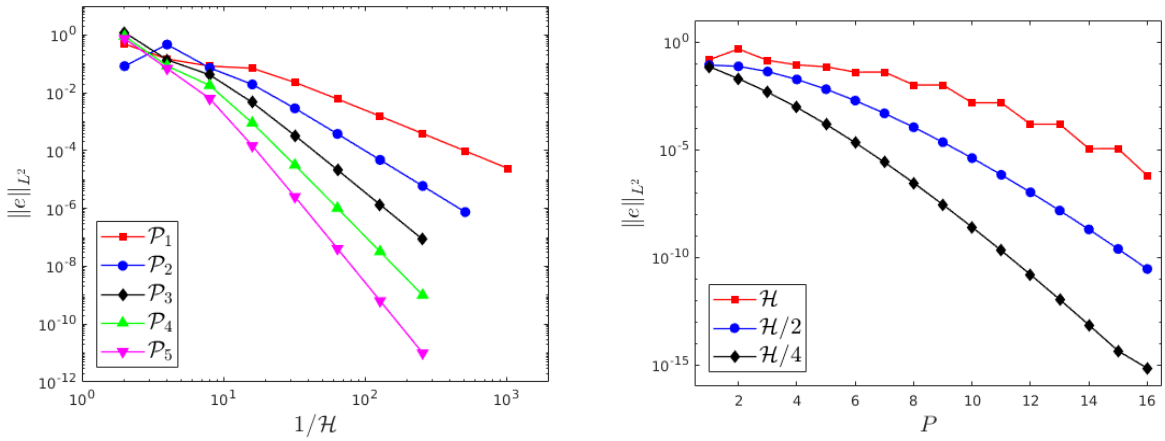
Figure 7.1 Solution Manufacturée $u(x, y) = \sin(2\pi x)\sin(2\pi y) + 0.1\sin(20\pi y)$

pente de chacune des droites est répertoriée dans le tableau 7.1. On constate que les taux de convergence s'apparentent bel et bien aux résultats attendus de $P + 1$. Dans le second cas, la convergence P est observée en utilisant un maillage constant mais en variant l'ordre des polynômes d'interpolation.

Les résultats de la figure 7.2 sont tirés de simulations exécutées avec trois processus MPI. Le second objectif de ce test est de vérifier que la précision des calculs est indépendante du nombre de partitions utilisées. Pour ce faire, l'analyse de convergence est répétée pour différentes tailles de groupe MPI. Sans illustrer toutes les données, l'écart relatif de l'erreur, pour des maillages à 64^2 ($h = 1/64$) et 256^2 ($h = 1/256$) éléments, est enregistré dans le tableau [1] 7.2. On s'aperçoit rapidement que le nombre de partitions n'a aucun impact sur la précision des calculs. Ce même comportement a été observé sur toutes les simulations comprises dans cette analyse de convergence.

Finalement, ce premier test met en évidence l'accord entre la théorie et les taux de convergence du tableau 7.1. De plus, il a été prouvé que le partitionnement du maillage et l'algorithme de résolution parallèle sont fonctionnels et robustes. Néanmoins, la nature du problème omet de vérifier certains cas plus *difficiles* à résoudre qui seront abordés aux sections suivantes.

1. L'erreur obtenue avec 1 seul processus MPI est considérée comme la valeur de référence



(a) Convergence H (b) Convergence P

Figure 7.2 Analyse de convergence

Tableau 7.1 Taux de Convergence

P	$\|\epsilon_h\|/\|\epsilon_{h/2}\|$
1	1.936681
2	2.94468
3	3.940916
4	4.950859
5	5.957799

Tableau 7.2 Écart relatif de l'erreur pour différentes tailles de groupe MPI

G	$h = 1/64$	$h = 1/256$
ϵ	2.1562e-05	8.5130e-08
2	3.1112e-14	6.4114e-13
3	6.2854e-15	3.1093e-14
4	2.1056e-14	7.4531e-13
5	1.0214e-14	9.8550e-13
10	3.7712e-15	7.9568e-13
100	1.0528e-14	1.9999e-12

7.1.1 Écoulement de Poiseuille

Comme il a été mentionné à la section 5.3.1, l'algorithme de numérotation des DDLs est largement plus complexe lorsque des problèmes multivariables sont impliqués. Le prochain cas test fait donc intervenir l'équation de Navier-Stokes incompressible pour résoudre un écoulement de Poiseuille 2D.

L'écoulement est simulé sur une conduite avec des conditions de non-glissement aux parois et un écoulement parabolique ($\mathbf{u}_{ex} = [y(1-y), 0]^T$) imposé en entrée. La conduite possède une longueur de $L = 20$ et une hauteur de $H = 1$. Encore une fois, le domaine est décomposé en trois partitions pour réaliser le calcul.

Le champ de vitesse simulé, illustré à la figure 7.3, conserve parfaitement le profil de vitesse d'entrée tout le long de la conduite. En fait, puisque l'approximation élémentaire Q_3Q_2 est utilisée, la solution est reproduite à la précision machine.

7.2 Écoulement statique autour d'un cylindre

Afin de nous convaincre du bon fonctionnement du code pour des problèmes plus réalistes et plus complexes, nous délaissons la méthode des solutions manufacturées pour une différente approche. En effet, les prochains tests utilisent les résultats obtenus avec *EF6* comme solutions de référence.

Cette méthode nous permet donc de vérifier le comportement du code pour des problèmes sans solution analytique connue tel que l'écoulement d'un fluide autour d'un cylindre fixe (illustré à la figure 7.4a).

Cette étude est réalisée pour un écoulement stationnaire et sur un domaine rectangulaire. Ce dernier possède une longueur de $40D$ et une largeur de $25D$ où D est le diamètre du cylindre. Le cylindre est positionné à une distance de $10D$ de l'entrée. Des conditions de Dirichlet sont

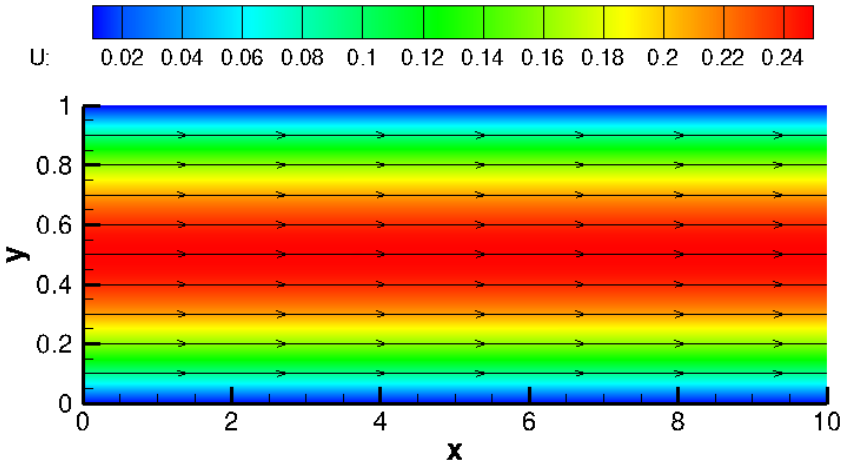


Figure 7.3 Écoulement de Poiseuille 2D

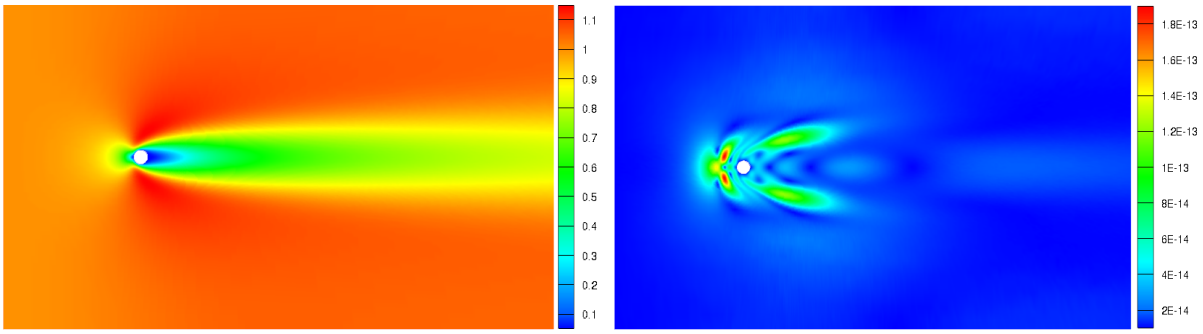
appliquées :

— à l'entrée, pour forcer une vitesse normale unitaire ($U = 1$)

— sur le cylindre, pour imposer des conditions de parois sans glissement ($U = 0, V = 0$)

— sur les cotés, pour simuler des conditions de symétrie ($V = 0$)

Les paramètres $D = 1$, $\rho = 1$ et $\mu = 0.1$, où ρ et μ sont la masse volumique et la viscosité du fluide, sont choisis de manière à obtenir un nombre de Reynolds de 10.



(a) Champ de vitesse $\|\mathbf{U}\|$ (b) Erreur $E = \|\mathbf{U}_{EF7} - \mathbf{U}_{EF6}\|$

Figure 7.4 Écoulement 2D autour d'un cylindre pour un $Re = 10$

L'étude est réalisée avec *EF6* et *EF7* (quatre processus MPI). Les profils de vitesse obtenus (figure 7.4) sont indiscernables et on remarque, en calculant l'erreur $E = \|\mathbf{U}_{EF7} - \mathbf{U}_{EF6}\|$

(figure 7.4b), qu'ils sont bel et bien identiques. En effet, l'erreur E est, en tout point du domaine, inférieure ou du même ordre de grandeur que la précision machine.

Cavité entraînée

Jusqu'à présent, aucun des cas tests ne fait intervenir de variables globales. Comme cela a été discuté à la section 5.3.1, ces dernières nécessitent une gestion légèrement différente des variables locales. Afin de vérifier l'implémentation de cette branche, le problème de la cavité entraînée est résolu.

Ce problème simule un écoulement à l'intérieur d'une cavité fermée qui, dans notre situation, est de forme carrée. Des conditions de non-glissement sont imposées sur les parois de la cavité à l'exception d'une seule, où une vitesse tangentielle est imposée. Le déplacement à cette paroi entraîne le reste de la cavité dans un mouvement circulaire. Ce problème présente la particularité d'avoir un champ de pression défini à une constante près. Il est alors nécessaire de définir une pression de référence, typiquement en imposant la pression en un point. Dans le cadre de cette vérification, la pression moyenne est imposée à l'aide d'un multiplicateur de Lagrange, introduisant ainsi une variable globale.

La cavité carrée de taille $L = 1$ est entraînée par la paroi supérieure, où une vitesse $U = x(1 - x)$ est appliquée. En ce qui concerne les propriétés du fluide ρ et μ, les valeurs 5.0 et 0.1 sont respectivement assignées. Finalement, la pression moyenne est imposée à 0.

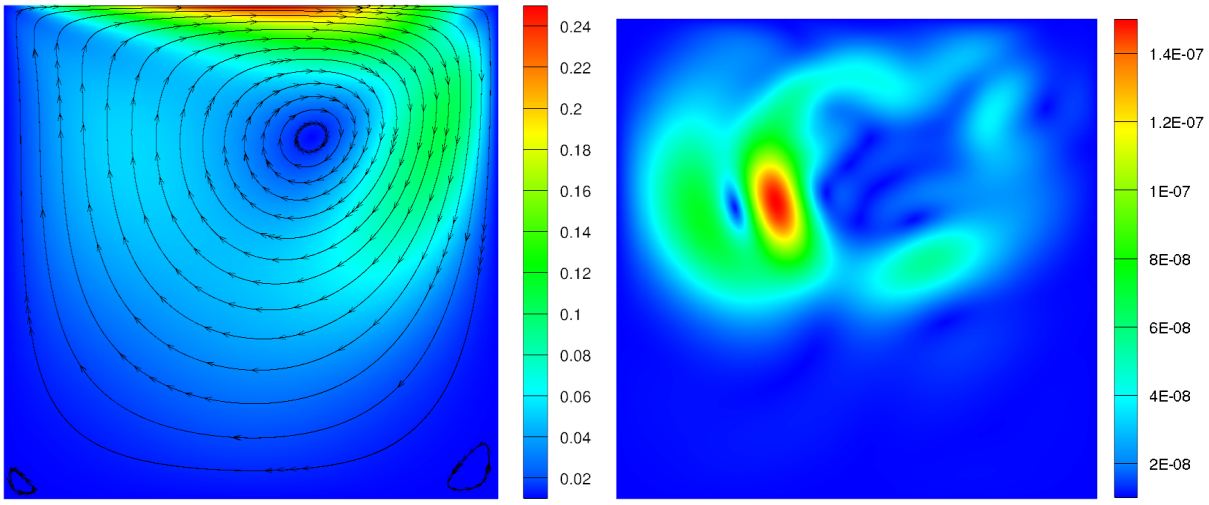
Les figures 7.5b et 7.5d illustrent l'écart en vitesse et en pression entre les solutions produite par *EF6* et *EF7*. Encore une fois, ces dernières sont, à toute fin pratique, identiques. On peut alors conclure que la gestion des variables globales est opérationnelle.

7.3 Allées de Von Karman

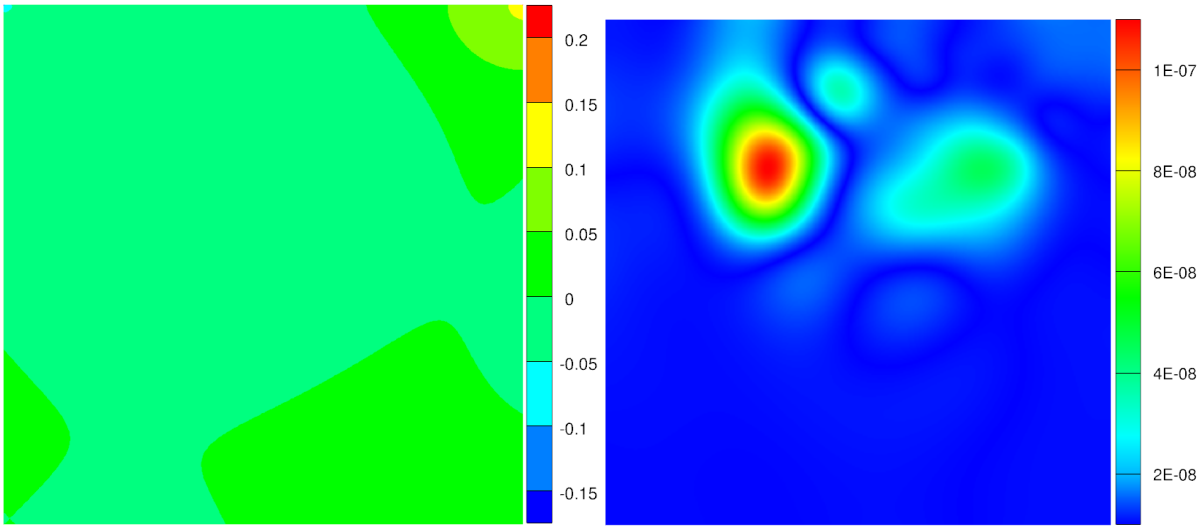
La dernière étape de cette vérification consiste à valider le module d'avance temporelle. Ce dernier n'a jamais été mentionné jusqu'à présent, car il n'a subi aucun changement majeur lors de la parallélisation MPI du programme. Cependant, quelques modifications ont dû être apportées afin de l'adapter aux nouvelles structures de données.

Le problème d'écoulement autour d'un cylindre est réutilisé dans cette section, mais cette fois, l'étude est réalisée dans un régime transitoire. L'erreur est alors calculée à partir de la fréquence des relâchers tourbillonaires générés en aval du cylindre. Pour ce faire, la vitesse est mesurée en un point aux coordonnées $\mathbf{x}_0 = \{0, 7.5\}$.

Notons que la condition frontière à l'entrée est modifiée afin de démarrer la simulation avec



(a) Champ de vitesse $\|\mathbf{U}\|$ (b) Erreur $E_U = \|\mathbf{U}_{EF7} - \mathbf{U}_{EF6}\|$



(c) Champ de pression P (d) Erreur $E_P = \|\mathbf{P}_{EF7} - \mathbf{P}_{EF6}\|$

Figure 7.5 Écoulement dans une cavité carrée pour un $Re \approx 50$

un profil de vitesse nul et d'accélérer graduellement l'écoulement pour obtenie $u = 1$. La forme exacte de cette dernière est donnée à l'équation (7.4) où $t_1 = 5$ et $t_2 = 7$. Les termes fonctions de la position y, brisant la symétrie du problème, sont ajoutés pour introduire une perturbation dans l'écoulement et ainsi générer plus rapidement les relâchers tourbillonnaires.

$$u(y,t) = \begin{cases} \left\{-2\left(\frac{t}{t_1}\right)^3 + 3\left(\frac{t}{t_1}\right)^2\right\}(1+0.01y) - \left\{-2\left(\frac{t}{t_2}\right)^3 + 3\left(\frac{t}{t_2}\right)^2\right\}0.01y, & t < t_1 \\ (1+0.01y) - \left\{-2\left(\frac{t}{t_2}\right)^3 + 3\left(\frac{t}{t_2}\right)^2\right\}0.01y, & t_1 \leq t < t_2 \\ 1, & t \geq t_2 \end{cases} \tag{7.4}$$

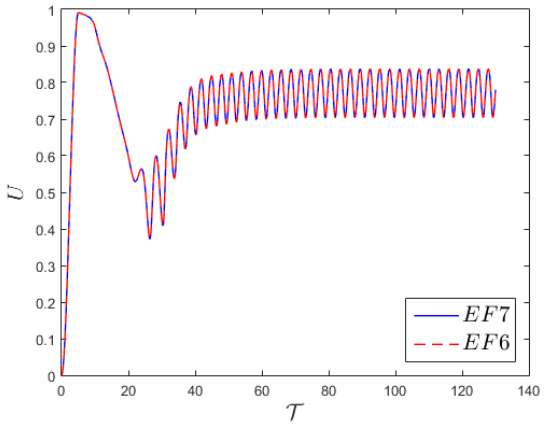
La figure 7.6a exprime la vitesse au point $\mathbf{x}_0$ pour les 130 premières secondes de simulation. Lorsque la vitesse à l'entrée atteint sa valeur nominale, on observe une zone transitoire ($\mathcal{T} \approx [5, 40]$) avant que l'écoulement se stabilise dans un régime périodique correspondant aux passages des tourbillons sur le point de mesure.

Avant de discuter de l'écart entre les solutions de *EF6* et *EF7*, il faut d'abord mentionner que le module d'avance temporelle, basé sur les formules de différences finies arrières (BDF [2]) utilise un pas de temps variable. Ce dernier est choisi de manière à ce que l'estimation de l'erreur temporelle soit borné par une tolérance τ. Dans le cas présent, ce paramètre est fixé à $\tau = 10^{-4}$. L'erreur de la vitesse mesurée en fonction du temps, présentée à la figure 7.6b, est inférieure à cette tolérance et est donc jugée acceptable.

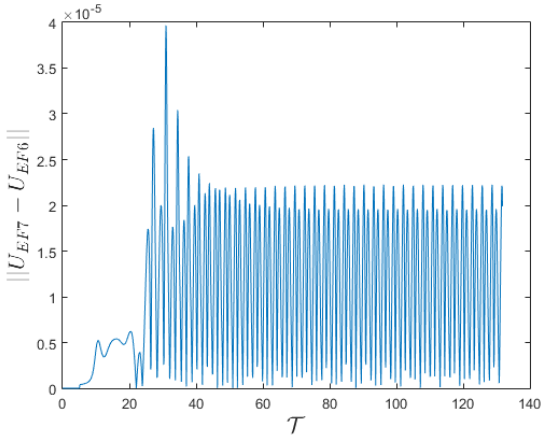
Pour éviter toutes ambiguïtés, le contenu fréquentiel du signal périodique est calculé avec une transformation de Fourier. La superposition des pics de la figure 7.6c, à une fréquence de 0.3433 Hz, confirme que la résolution temporelle est valide.

L'ensemble des tests confirme le bon fonctionnnement de *EF7*. Toutefois, avant de remplacer complètement *EF6*, la performance de la nouvelle parallélisation doit être caractérisée.

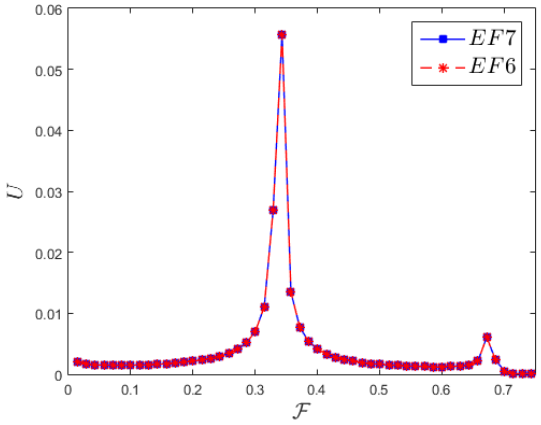
2. Backward differential formulas



(a) Vitesse $u(\mathbf{x}_0, t)$



(b) Erreur $E_u = \|\mathbf{u}_{EF7} - \mathbf{u}_{EF6}\|$



(c) Contenu en fréquence du signal

Figure 7.6 Mesure de la vitesse u au point $\mathbf{x}_0$

CHAPITRE 8 ANALYSE DE PERFORMANCES

Suite à la phase de vérification abordée au chapitre précédent, nous arrivons finalement au point culminant de ce mémoire : l'analyse des performances de *EF7*. Puisque la résolution du système matriciel peut s'exécuter avec différents solveurs, cette analyse tente de faire une distinction entre les performances des solveurs et celles de *EF7*.

Afin de comparer les perfomances de certains solveurs, les données présentées dans ce chapitre sont toutes issues du même problème de diffusion qu'à la section 7.1 résolu sur le serveur de calcul *Graham* de *Compute Canada*.

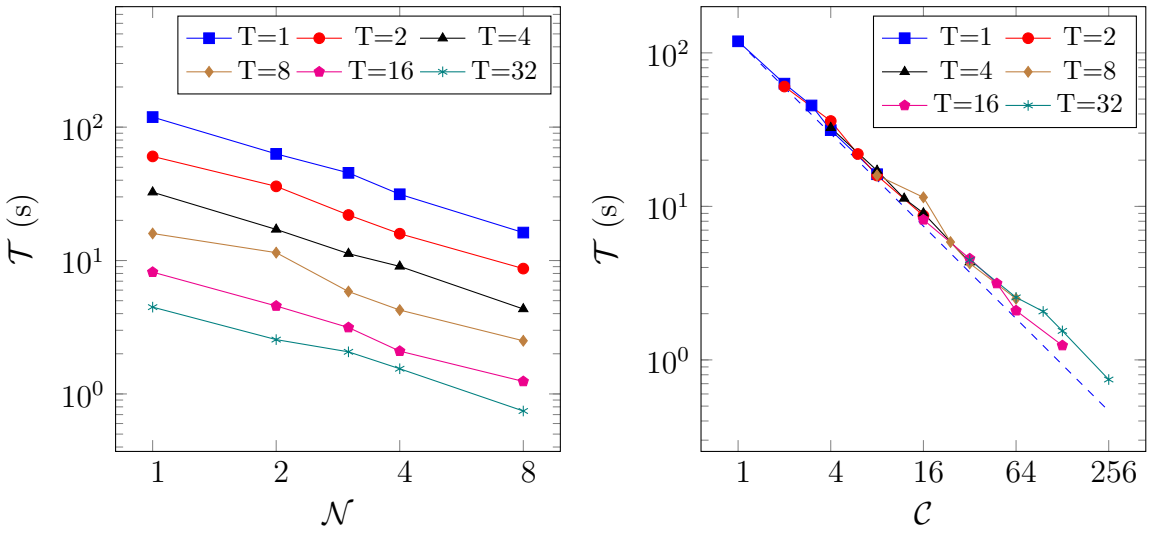
8.1 Performance de *EF7*

Cette première étude consiste à évaluer les gains d'une parallélisation hybride MPI/OMP. Afin de mesurer les performances de *EF7*, nonobstant le choix du solveur, nous utilisons l'étape d'assemblage comme indice de performance. En effet, l'étape d'initialisation du programme est généralement négligeable, en terme de temps d'exécution, comparativement à l'assemblage. Donc, nous évaluerons l'accélération de *EF7* pour différentes combinaisons de processus et de threads.

Le premier cas d'étude est réalisé avec des interpolants de degré 5, sur un maillage de 512×512 éléments, générant un problème à 6.55 M de DDLs et une matrice de 321 M de coefficients non nuls.

Les résultats affichés à la figure 8.1a montrent que le temps d'exécution diminue avec l'ajout de processus ou de thread. À la figure 8.1, les mêmes résultats sont présentés en fonction du nombre de cœurs utilisés de manière à constater que l'accélération provenant de MPI est sensiblement la même que celle procurée par OpenMP. En fait, en comparaison à la droite $1/P$ représentant une parallélisation parfaite [1] (trait en pointillé de la figure 8.1), les performances de *EF7* sont excellentes même lorsque des centaines de cœurs sont utilisés. On conclut que la parallélisation hybride est adaptée aux besoins de *EF7*.

1. Accélération obtenue selon la loi d'Amdhal si la fraction parallélisable est de 100%.



(a) Temps d'exécution $\mathcal{T}$ en fonction du nombre de noeuds de calculs $\mathcal{N}$

(b) Temps d'exécution $\mathcal{T}$ en fonction du nombre de coeurs $\mathcal{C}$ total

Figure 8.1 Accélération de l'assemblage par une parallélisation hybride

Les simulations sont réalisés avec 1 processus MPI par noeud et 1 thread par coeur $\mathcal{C}$. Les données sont regroupées selon le nombre de threads T utilisé par processus MPI.

8.2 Performances des solveurs

Le temps d'exécution du programme est majoritairement écoulé dans les appels aux solveurs. L'efficacité de ces derniers est donc primordiale pour la bonne performance du programme.

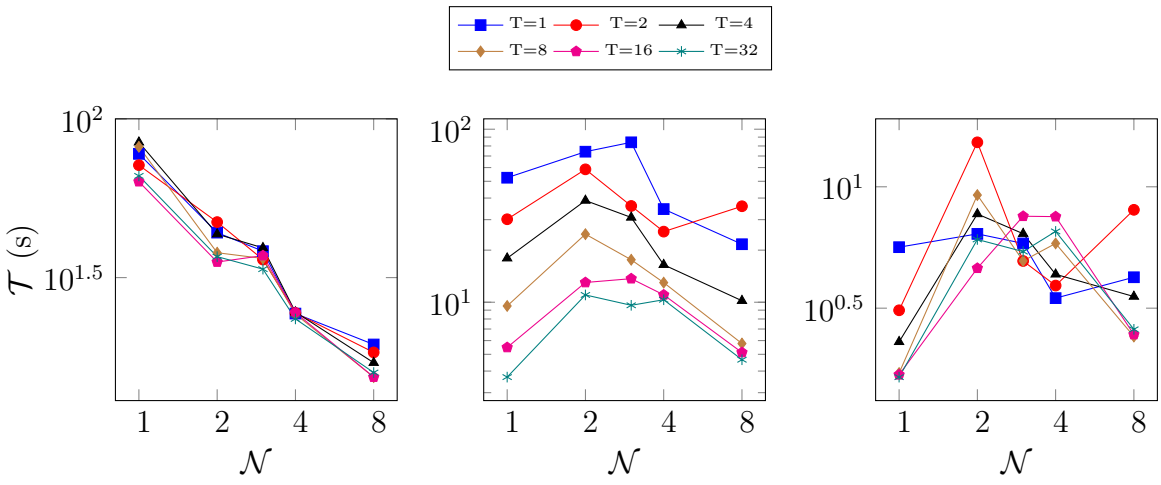
8.2.1 CPardiso

Accélération

Dans le cas des solveurs directs comme *CPardiso*, le système est résolu en trois phases : l'analyse, la factorisation et la résolution. La fréquence à laquelle chacune de ces phases sont exécutées peut varier grandement selon la nature du problème, il est donc important de les évaluer séparément. En effet, l'analyse de la matrice, présente dans l'étape d'initialisation (section 5.1) n'est exécutée qu'une seule fois alors que les deux autres, dans les itérations de Newton ou temporelles, peuvent survenir à maintes reprises [2]. Ainsi, même une petite accélération de la phase de résolution peut s'avérer être un gain considérable sur le temps d'exécution total d'un problème transitoire.

Les temps d'exécution $\mathcal{T}$ de chacune des phases sont exposés à la figure 8.2.

2. Notons que la matrice n'est pas factorisée à chaque itération, mais uniquement lorsque la convergence ralenti.



(a) Analyse (b) Factorisation (c) Résolution

Figure 8.2 Performances de Cluster Pardiso

Tout d'abord, on voit que l'analyse (figure 8.2a) bénéficie de la parallélisation MPI, mais ne retire que très peu de gain de la part d'OpenMP. Pour la factorisation, on observe le comportement inverse. Selon la figure 8.2b, l'ajout de threads par processus MPI diminue considérablement le temps d'exécution. Toutefois, l'usage d'un plus grand nombre de nœuds de calcul ne semble pas particulièrement avantageux. En effet, on remarque que le temps d'exécution augmente considérablement lors du passage de 1 à 2 nœuds et diminue légèrement par la suite. Même avec 8 processus, les résultats peinent à surpasser les performances obtenues avec 1 seul processus. En ce qui concerne la phase de résolution, elle ne semble pas systématiquement profiter d'aucune des deux parallélisations. Encore une fois, les performances sont généralement pires lorsqu'exécutées sur plus d'un processus.

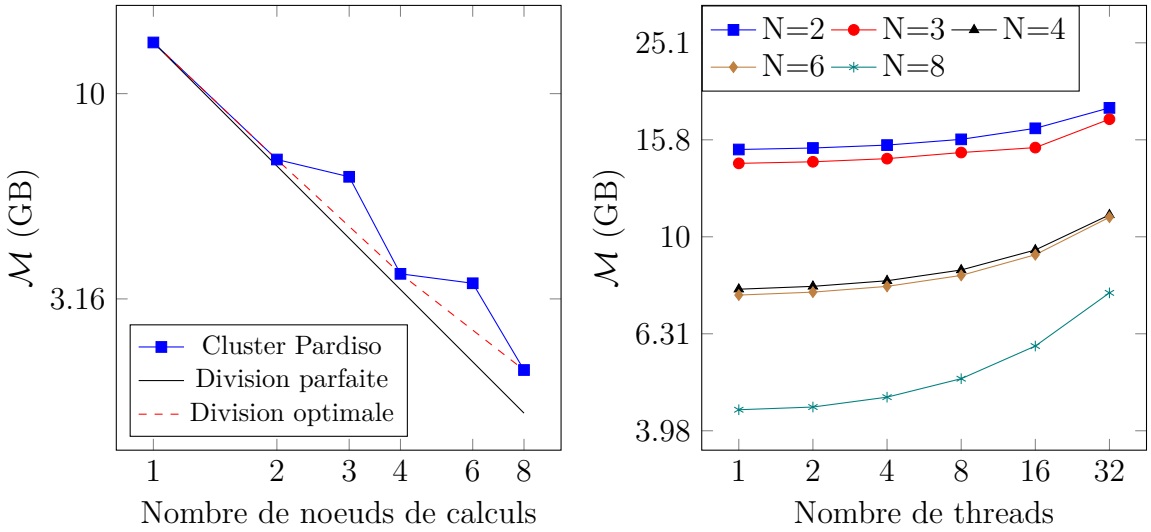
Ces résultats ne sont malheureusement pas à la hauteur des performances décrites dans la littérature [35, 31]. En effet, même si les phases d'analyse et de factorisation obtiennent un gain par l'augmentation du nombre de processus ou de threads utilisés, l'accélération totale est limitée par la phase de résolution.

Actuellement, nous sommes encore à la recherche d'une erreur dans la configuration de MKL qui pourrait expliquer ces faibles performances. Entre-temps, ces résultats laissent présager que le solveur direct *CPardiso* n'est pas une solution viable pour accélérer le programme.

Consommation mémoire

La performance de *CPardiso* s'exprime également par la distribution de ses besoins en mémoire. Puisque la mémoire d'un nœud de calcul est généralement limitative, la taille maximale d'un problème pouvant être traité est dictée par la capacité de *CPardiso* à diviser uniformément ses structures internes.

Pour cette analyse, la mémoire $\mathcal{M}$ allouée par processus, à la phase d'analyse et de factorisation, est illustrée à la figure 8.3. Notons que les valeurs affichées correspondent aux processus ayant alloué le maximum de mémoire.



(a) Analyse

(b) Factorisation

Figure 8.3 Consommation en mémoire de Cluster Pardiso

Tout d'abord, portons notre attention sur la phase d'analyse (figure 8.3a). On remarque que la consommation de mémoire décroît plus pour certaines tailles de groupe MPI. Ce comportement résulte de la transformation de la matrice en arbre binaire [35]. Ainsi, la distribution est optimale lorsque l'arbre est complet, soit lorsque le nombre de nœuds est une puissance de deux. Dans cette configuration, le coût en mémoire s'approche d'une distribution parfaite (trait plein de pente $1/p$). Également, la figure 8.3b présente, pour la factorisation, le même comportement discontinu qu'à l'analyse. Toutefois, on souligne ici que la mémoire allouée augmente légèrement avec le nombre de threads. Somme toute, la distribution des besoins en mémoire de *CPardiso* est bien équilibrée et permet de traîter des problèmes qui normalement, seraient trop gros pour un seul nœud de calcul.

Étude spectrale

Puisque l'usage de polynômes de degré variable affecte la densité de la matrice à résoudre, l'analyse de *CPardiso* se termine par une étude visant à observer les performances du solveur en MES.

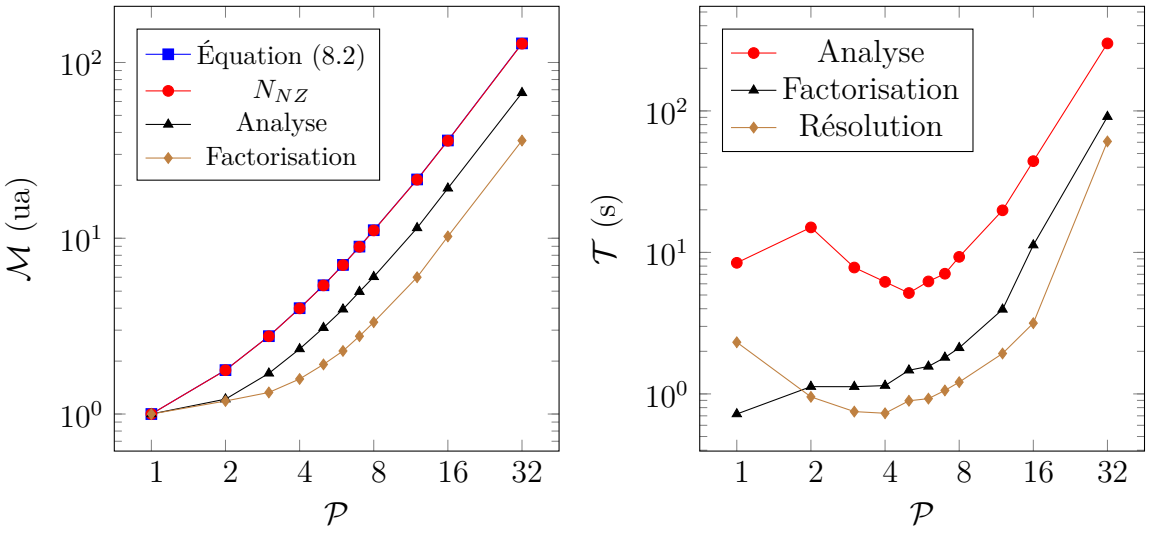
À cette fin, les maillages ont été ajustés pour obtenir le même nombre de DDLs en augmentant le degré $\mathcal{P}$ des polynômes.

Différents maillages ont été créés tel que, une fois jumelés avec l'interpolant d'ordre $\mathcal{P}$ correspondant, la taille du système N_{DDL} soit toujours la même. Sur un maillage carré structuré avec $\mathcal{N}$ éléments par coté, on obtient le nombre de DDLs et le nombre de coefficients non nuls par les relations [3] (8.1) et (8.2)

$$N_{DDL} \approx \mathcal{N}^2\mathcal{P}^2 \tag{8.1}$$

$$N_{NZ} \approx \mathcal{N}^2\left(\mathcal{P}^2+2\mathcal{P}\right)^2 \tag{8.2}$$

La figure 8.4 résume les résultats pour $N_{DDL} \approx 1M$. Sur le graphique de gauche (8.4a), la quantité de mémoire normalisée par rapport à $\mathcal{P}=1$ est affichée.



(a) Mémoire normalisée par rapport à $\mathcal{P}=1$

(b) Temps d'exécutions

Figure 8.4 Performances de Cluster Pardiso avec des éléments spectraux

Expérience réalisée avec 2 processus MPI et 4 thread OpenMP

3. Ces équations ne prennent pas en compte les conditions de Dirichlet.

On observe, à la figure 8.4a, que la consommation de mémoire est proportionnelle au nombre de coefficients non nuls de la matrice. On remarque également que, pour $\mathcal{P}$ assez élévé, la distance entre les courbes est pratiquement constante, laissant croire que le besoin en mémoire n'augmente pas exactement au même rythme que N_{DDL}. En effet, l'espace alloué à la phase d'analyse et de factorisation s'élargit respectivement 2 et 4 fois plus lentement que N_{DDL}.

Les temps d'exécutions pour les trois phases sont cumulés à la figure 8.4b. Il se trouve que *CPardiso* performe très bien pour des interpolants allant jusqu'à $\mathcal{P} \approx 8$. En fait, on observe que le cadre d'utilisation optimale n'est pas forcément avec des éléments linéaires, mais parfois avec des interpolants d'ordre plus élevé. Toutefois, pour des ordres très élevés ($\mathcal{P} > 12$), les performances semblent se détériorer rapidement.

8.2.2 Mumps

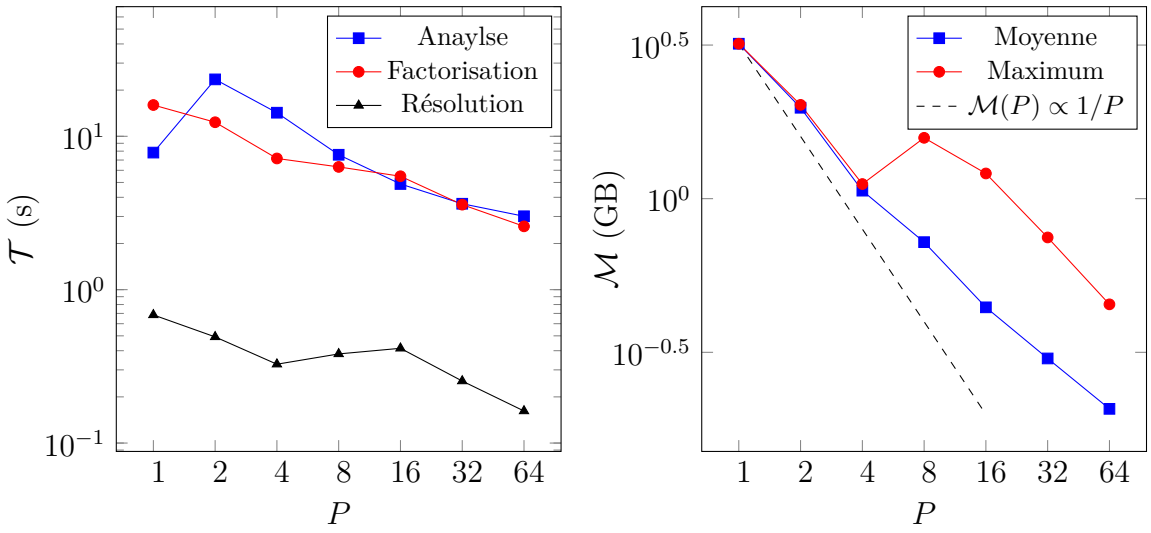
Les performances d'un deuxième solveur direct, *Mumps*, ont également été mesurées dans l'espoir de trouver une alternative à *CPardiso*. Contrairement aux analyses précédentes, uniquement la parallélisation MPI a été évaluée. Puisque *Mumps* a été installé par l'intermédiaire de *Petsc*, les options de compilation activant les threads OpenMP n'ont pas été fournies.

Le même problème de diffusion est résolu avec des interpolants d'ordre 5, mais le maillage utilisé est une grille de 256×256 éléments engendrant sensiblement $1.6M$ de degrés de liberté. Le système est délibérément plus petit que le précédent, car une erreur apparaît pour des problèmes plus grands. Ils est possible que cette erreur soit causé par le nombre de coefficient de la factorisation qui ne peut être représenté par un entier à 32 bits. Le solveur devra alors être recompilé avec des entiers de 64 bits.

Les temps d'exécution (figure 8.5a) de l'analyse, de la factorisation et de la résolution décroient avec l'ajout de processus. Pour l'analyse, le saut important entre $P = 1$ et $P = 2$ résulte du changement d'algorithme de réordonnancement [4]. Jusqu'à $P = 16$, l'accélération est proportionnnelle aux nombres de processus utilisés. Ensuite, le gain commence à diminuer. Ceci est dû à la taille du problème qui est trop petite.

À la figure 8.5b, la quantité maximale et moyenne de mémoire allouée par processus est affichée pour différentes tailles de groupes MPI. Les résultats montrent que la consommation moyenne décroit de façon significative. Toutefois l'écart entre la valeur maximale et moyenne pour $P > 4$ indique que les coefficients de la factorisation ne sont pas équitablement redistribués. Il est possible que ce déséquilibre survienne seulement lorsque beaucoup de processus sont exécutés pour un problème de petite taille.

4. Pour $P = 1$, l'analyse est séquentielle alors que pour $P > 2$, elle est parallèle.



(a) Temps d'exécution

(b) Consomation en mémoire de mumps à la factorisation

Figure 8.5 Performances de Mumps

Dans l'éventualité où la reconfiguration du solveur permettrait une parallélisation OpenMP et une meilleure mise à l'échelle, les résultats actuels suggèrent que *Mumps* est une alternative adéquate à *CPardiso*.

8.2.3 PETSc

Afin de mettre en perspective les performances des méthodes directes, une étude très succincte est réalisée pour observer le potentiel d'accélération des solveurs itératifs de la bibliothèque *PETSc*. Encore une fois, les performances sont mesurées sur le cas test de la section 8.2.2. Puisque le système est symétrique défini positif, nous utilisons l'algorithme du gradient conjugé sans aucun préconditionnement. Les résultats sont affichés à la figure 8.6.

Les temps de résolution sont significativement plus élevés que pour les solveurs directs, car plus de 5000 itérations sont nécessaires avant d'atteindre la convergence [5]. Il est évident que l'ajout d'un préconditionneur réduirait considérablement le nombre d'itérations calculées. De plus, l'accélération observée, avec l'ajout de processus, est largement plus grande que pour *Mumps* et *CPardiso*.

5. Le critère de convergence par défault de PETSc est défini par $\|\mathbf{r}\|_2 \leq \max(10^{-5} \|\mathbf{b}\|_2, 10^{-50})$, où $\mathbf{r}$ est le résidu et $\mathbf{b}$ le membre de droite.

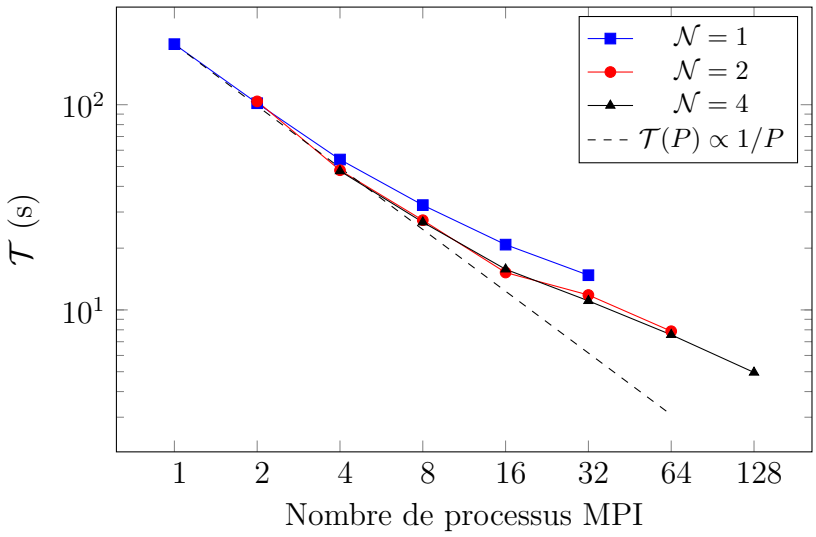


Figure 8.6 Performances de *PETSc* avec l'algorithme du gradient conjugé

CHAPITRE 9 CONCLUSION

Ce mémoire présente l'implémentation d'une parallélisation hybride MPI/OMP pour un logiciel d'analyse par éléments spectraux. La stratégie employée possède trois niveaux de parallélisme. Le premier consiste à diviser le domaine en partitions pouvant être traitées simultanément par plusieurs processus MPI. Pour le deuxième, un algorithme de coloration est appliqué sur les éléments d'une partition afin que leurs systèmes élémentaires puissent être construits sans concurrence par des threads OpenMP. Finalement, des instructions vectorielles sont utilisées pour calculer les coefficients des systèmes élémentaires.

Un effort de programmation majeur a été consacré au développement du premier niveau. En effet, une interface pour le logiciel *ParMétis* a été implémentée afin de créer un partitionnement sans recouvrement, de type *node-cut*. Les noeuds sont ensuite renumérotés afin de former des structures faciles à communiquer. Un algorithme parallèle de numérotation des degrés de liberté gérant indépendamment les variables nodales et globales a été développé. Les communications MPI nécessaires ont également été instaurées pour l'assemblage du système linéaire global.

Pour le troisième niveau de parallélisme, l'ajout de directives OMP*simd* s'est révélé être une approche simple et efficace pour implémenter des instructions vectorielles. La phase d'assemblage peut alors bénéficier d'un facteur d'accélération typique entre 2 et 4 selon la nature des équations et l'ordre des interpolants utilisés.

Toutes les nouvelles fonctionnalités ont été vérifiées par la méthode des solutions manufacturées ou par comparaison avec les résultats *EF6*. Une série de cas tests comprenant des problèmes de diffusion et d'écoulement incompressible est résolue en régime stationnaire ainsi que transitoire.

Puisque la stratégie développée se base sur l'utilisation de solveurs distribués, une interface à été développée afin de facilement incorporer les solveurs *CPardiso*, *Mumps* et certaines méthodes de Krylov implémentées dans *PETSc*. Les performances de ces derniers ont été évaluées sur un problème de diffusion thermique. Il a été observé que *CPardiso* et *Mumps* permettent d'aborder des problèmes de plus grande taille, mais que ces derniers offrent des accélérations modiques relativement aux solveurs itératifs de *Petsc*.

L'objectif primaire de ce projet n'est donc pas complété à 100% puisque les solveurs directs testés ne permettent pas encore d'obtenir des accélérations significatives lorsqu'un grand nombre de coeurs est utilisé.

9.1 Améliorations futures

Ce projet n'est donc que le premier pas dans le développement d'une version massivement parallèle du logiciel *EF*. Dans l'immédiat, une méthode de résolution doit être implémentée de manière à montrer des accélérations significatives même lorsque plusieurs centaines de coeurs sont en exécution. Sans abandonner totalement *CPardiso* et *Mumps*, nous suggérons, comme axe de recherche principal, de développer l'algorithme de sous-structuration présenté au chapitre 4.2.2. Cet algorithme, implémenté avec des solveurs directs, permettrait de conserver la robustesse du programme tout en améliorant le taux de parallélisation.

Dans un second temps, certains points peuvent être le sujet d'études ou d'améliorations à venir :

— Jusqu'à présent, *EF7* ne possède pas de moyen de visualiser les résultats. Toutes les figures présentées dans ce mémoire sont générées à partir de fichiers convertis pour être lisibles depuis *EF6*. Il serait donc intéressant de développer un module interagissant avec *Paraview* afin de visualiser les données directement.

— Les prochaines versions d'*EF* prévoient implanter une méthode d'adaptation de maillage. La phase d'initialisation du programme sera alors exécutée fréquemment et devra être le sujet d'un profilage détaillé afin de déterminer si une parallélisation OpenMP plus invasive est recommandée.

RÉFÉRENCES

[1] MPI Forum, *MPI : A Message-Passing Interface Standard*, 2015.

[2] A. Moulin, "Simulation numérique par la méthode des éléments spectraux des vibrations induites par relâcher tourbillonnaire," Master's thesis, École Polytechnique de Montréal, 2016.

[3] A. Ern and J.-L. Guermond, *Theory and practice of finite elements*, vol. 159. Springer Science & Business Media, 2013.

[4] S. Brenner and R. Scott, *The mathematical theory of finite element methods*, vol. 15. Springer Science & Business Media, 2007.

[5] C. Pozrikidis, *Introduction to finite and spectral element methods using MATLAB*. CRC Press, 2005.

[6] A. Q. Claudio Canuto, Yousuff Hussaini, *Spectral Methods Fundamentals in Single Domains*. Springer, 2006.

[7] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, pp. 948–960, Sept 1972.

[8] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design, Fifth Edition : The Hardware/Software Interface*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 5th ed., 2013.

[9] R. E. Bryant and D. R. O'Hallaron, *Computer Systems : A Programmer's Perspective*. USA : Addison-Wesley Publishing Company, 2nd ed., 2010.

[10] H. Bouchened and J. M. Tores, "Notes de cours." 2001.

[11] R. Rabenseifner, G. Hager, and G. Jost, "Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes," in *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, PDP '09, (Washington, DC, USA), pp. 427–436, IEEE Computer Society, 2009.

[12] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur, "Mpi + mpi : A new hybrid approach to parallel programming with mpi plus shared memory," vol. 95, 12 2013.

[13] J. Hoeflinger, "Cluster openmp for intel compiler." 2010.

[14] G. Karypis and V. Kumar, "MeTis : Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 4.0." `http://www.cs.umn.edu/~metis`, 2009.

[15] F. Pellegrini, "PT-Scotch and libScotch 5.1 User's Guide," Aug. 2008. User's manual.

[16] C. Chevalier and F. Pellegrini, "Pt-scotch : A tool for efficient parallel graph ordering," *Parallel Comput.*, vol. 34, pp. 318–331, July 2008.

[17] R. Preis and R. Diekmann, "Party - a software library for graph partitioning," in *Advances in Computational Mechanics with Parallel and Distributed Processing*, pp. 63–71, Civil-Comp Press, 1997.

[18] B. Hendrickson and R. Leland, "The Chaco User's Guide : Version 2.0," Tech. Rep. SAND94–2692, Sandia National Lab, 1994.

[19] X. Guo, M. Lange, G. Gorman, L. Mitchell, and M. Weiland, "Developing a scalable hybrid mpi/openmp unstructured finite element model," *Computers & Fluids*, vol. 110, pp. 227–234, 2015.

[20] X. Guo, G. Gorman, A. Sunderland, and M. Ashworth, "Developing hybrid openmp/mpi parallelism for fluidity-icom-next generation geophysical fluid modelling technology," *Cray user group*, 2012.

[21] T. Husfeldt, "Graph colouring algorithms," *CoRR*, vol. abs/1505.05825, 2015.

[22] L. Thébault and E. Petit, "Asynchronous and multithreaded communications on irregular applications using vectorized divide and conquer approach," *Journal of Parallel and Distributed Computing*, vol. 114, pp. 16 – 27, 2018.

[23] T. Heister, M. Kronbichler, and W. Bangerth, "Massively parallel finite element programming," in *European MPI Users' Group Meeting*, pp. 122–131, Springer, 2010.

[24] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, "PETSc Web page." `http://www.mcs.anl.gov/petsc`, 2018.

[25] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An overview of the trilinos project," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 397–423, 2005.

[26] R. D. Falgout and U. M. Yang, "hypre : a library of high performance preconditioners," in *Preconditioners," Lecture Notes in Computer Science*, pp. 632–641, 2002.

[27] 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*, vol. 23, no. 1, pp. 15–41, 2001.

[28] "Intel Math Kernel Library." http ://software.intel.com/en-us/articles/intel-mkl/.

[29] X. S. Li and J. W. Demmel, "Superlu-dist : A scalable distributed-memory sparse direct solver for unsymmetric linear systems," *ACM Trans. Math. Softw.*, vol. 29, pp. 110–140, June 2003.

[30] D. Garcia-Donoro, A. Amor-Martin, and L. E. Garcia-Castillo, "Higher-order finite element electromagnetics code for hpc environments," *Procedia Computer Science*, vol. 108, pp. 818–827, 2017.

[31] Intel Xeon Phi User Group, *WARP3D Implementation of MKL Cluster PARDISO Solver*.

[32] "Parallel Direct Sparse Solver for Clusters Interface." https ://software.intel.com/en-us/mkl-developer-reference-c-parallel-direct-sparse-solver-for-clusters-interface.

[33] T. Mathew, *Domain Decomposition Methods for the Numerical Solution of Partial Differential Equations*. Springer-Verlag Berlin Heidelberg, 2008.

[34] "Vectorization Becomes Important—Again." https ://www.codeproject.com/Articles/1203955/Vectorization-Becomes-Important-Again, 2017.

[35] A. Kalinkin and K. Arturov, "Asynchronous approach to memory management in sparse multifrontal methods on multiprocessors," *Applied Mathematics*, vol. 4, pp. 33–39, July 2008.

ANNEXE A NOTATION ET SYMBOLISME INFORMATIQUE

Définition A.1 (Indirection)
Soit les vecteurs $\boldsymbol{a} \in \mathcal{X}^m$ [1], $\boldsymbol{b} \in \mathcal{X}^n$ *et* $\boldsymbol{s} \in \mathbb{N}^n$ *tel que* $max\{s_1, \ldots, s_n\} \leq m$. *On peut définir l'opérateur d'indirection*

$$I : \mathcal{X}^m \longmapsto \mathcal{X}^n$$

ou par composante

$$I : a_{s_i} \longmapsto b_i$$

tel que l'opération $\boldsymbol{b} = \boldsymbol{a}(\boldsymbol{s})$ *est parfaitement équivalente au code suivant :*

```
for(int i=0; i<n; i++){
        b[i] = a[s[i]];
}
```

Définition A.2 (Structure inverse)
Soit les vecteurs $\boldsymbol{s} \in \mathbb{N}^n$ *et* $\boldsymbol{a} \equiv \{1, 2, \ldots, n\}$. *L'inverse de* $\boldsymbol{s}$, *notée* $\boldsymbol{s}^{-1} \in \mathbb{N}^{\max\{s\}}$, *est défini telle que*

$$\boldsymbol{s}^{-1}(\boldsymbol{s}) = \boldsymbol{a}$$

Remarque 6. Si $\max\{\mathbf{s}\} >> n$, $\mathbf{s}^{-1}$ est généralement implémentée sous la forme d'un arbre binaire ou d'une table de hachage.

Définition A.3 (Concaténation)
Soit les k *vecteurs* $\boldsymbol{x}_i \in \mathcal{X}^{n_i}$ *et le vecteur* $y \in \mathcal{X}^m$ *pour* $1 \leq i \leq k$ *et* $m = \sum n_i$. *L'opérateur de concaténation* $C : \mathcal{X}^{n_1} \times \mathcal{X}^{n_2} \times \ldots \times \mathcal{X}^{n_k} \longmapsto \mathcal{X}^m$ *permet de regrouper les éléments des* $\boldsymbol{x}_i$ *au sein du vecteur* $\boldsymbol{y}$ *tel que*

$$\boldsymbol{y} = [\boldsymbol{x}_i]_{i=1}^k$$

où

$$\boldsymbol{x}_i = \boldsymbol{y}(\boldsymbol{s}_i) \ , \qquad \boldsymbol{s}_i \equiv \left\{ \sum_{j=1}^{i-1} n_j + 1, \ldots, \sum_{j=1}^{i} n_j \right\}$$

---

1. **a** est un vecteur de dimension m dont les composantes sont des objets $x \in \mathcal{X}$

Définition A.4 (Vecteur de vecteur)
Soit les n vecteurs $\boldsymbol{x}_i \in \mathcal{X}^k$*. On définit le vecteur* $\boldsymbol{X}$ *de longueur n tel que* $\boldsymbol{X}(i) = \boldsymbol{x}_i$*.* $\boldsymbol{X}$ *est donc un tableau 2D et doit être déférencé deux fois pour obtenir l'une des composantes scalaires.*

$$\boldsymbol{X}(i)(j) = x_{i,j}$$

Remarque 7. Dans la situation où les vecteurs $\mathbf{x}_i$ sont continus en mémoire ($\mathbf{x} = [\mathbf{x}_i]_{i=1}^n$), on obtient les deux propriétés suivantes :

1. $\mathbf{X} = \mathbf{x}$
2. $\mathbf{X}(i)(j) = \mathbf{x}\left((i-1)n + j\right)$

ANNEXE B MODULE DE COMUNICATION MPI

Le développement de *EF7*, concernant les modifications abordées au chapitre 5, fait apparaître plusieurs communications collectives. Ces dernières sont accompagnées de segments de codes servant à gérer l'allocation et la construction des messages. Un module de communication a donc été mis au point afin d'alléger le programme et de fournir une interface plus concise pour la manipulation des messages. Il cible principalement les communications collectives vectorielles (`MPI_Xxxxxxv`), car elles sont plus fréquentes et sont plus complexes à implémenter.

L'interface comprend des méthodes pour automatiser les opérations de communication récurrentes de *EF7*, notamment la construction d'un message à partir d'un ensemble d'objets désordonnés. Un autre aspect intéressant de ce module est qu'il définit, pour chaque type de communication, une opération inverse afin de répondre à un message antérieur. Un processus peut donc traiter des données reçues par une opération `MPI_Scatterv` et retourner les résultats à l'expéditeur avec la fonction `MPI_Gatherv` en utilisant le même objet.

Le module est constitué de trois classes abstraites :

`Msg` : C'est l'interface responsable du contenu du message et des appels MPI. Dans un souci de généralité, l'interface doit traiter des messages avec des types de données MPI différents. Présentement, l'interface ne supporte que des objets formés d'un même type de base (char, int, double, etc). Par exemple, les coordonnées des noeuds, en 3D, sont transférées par blocs contigus composés de trois doubles. Pour ce faire, `Msg` est implémenté comme une classe patron [1]. Cette caractéristique permet de réutiliser les mêmes formats de communications plus souvent.

`MsgTemplate` : La classe `MsgTemplate` représente le gabarit ou l'enveloppe du message. Elle dispose donc d'information sur la quantité et la destination des éléments à transférer. Elle ne possède aucune information sur la nature du contenu du message. Plusieurs messages peuvent alors réutiliser le même gabarit, comme c'est souvent le cas pour le programme *EF7*.

`Msgcommunicator` : Encapsulant le communicateur MPI, son objectif est de rendre le contexte de communication le plus opaque possible pour les autres classes du module. En effet, cette classe permet de construire des messages utilisant une topologie virtuelle sans se soucier de la renumérotation du rang des processus. Elle définit également un

1. *Template class*

second communicateur correspondant à la topologie réciproque, soit la topologie dont le sens des arêtes est inversé.

Les objets de type `MsgTemplate` sont généralement créés par la méthode `build(indices, n, func)`. L'argument `indices` est un vecteur d'entiers de longueur `n`, correspondant aux éléments à envoyer. L'argument `func` est une fonction telle que `func(indices[i]) = dest`, où `dest` est le rang du processus auquel sera envoyé l'élément `i` du message.

Similairement, les messages sont construits à partir de la méthode `build(items, indices, n, func)` où le vecteur `items` de longueur `n` contient les objets à communiquer. L'appel à cette méthode permet de construire un message tel que l'objet `item[i]` est alors envoyé au processus `func(indices[i])`.

ANNEXE C ALGORITHMES DE DISTRIBUTION DU MAILLAGE

Algorithme 17 Distribution des éléments (Domaine)

Input:

1: `nElm` := number of local element ; `nPart` := number of geometric partition ;
2: `partGeoPtr` := indices of first element of each partitionGeo ;
3: `elmPtr` := Pointer to first node of each element ;

4: `owners` := array of length `nElm`, `owners[i]` return process' rank where element i should reside
5: **procedure** DOMAIN : :DISTRIBUTEELEMENT(`owners`)
6: `MsgAllToAllv<int>** msgs` ← `new MsgAllToAllv<int>*[nPart];`
7: **for all** partitions $0 <$ `iPart` $<$ `nPart` **do**
8: `partitions[iPart]->BuildMsgElm(msgs[ipart],&owner[partGeoPtr[iPart]])`
9: **end for**
10: deallocate graphe and connectivity table ;
11: `allocateConnec()` ;
12: **for all** partitions $0 <$ `iPart` $<$ `nPart` **do**
13: `partitions[iPart]->DistributeElement(msgs[iPart]);`
14: **end for**
15: delete msgs
16: **end procedure**

Algorithme 18 Distribution des éléments (Domaine) (suite)

```
1: procedure Domain : :AllocateConnec( )
2:    nElm ← 0; connecSize ← 0;
3:    for all partitions 0 < iPart < nPart do
4:       partGeoPtr[iPart] ← nElm;
5:       nElmPart ← partitions[iPart] -> getNElm();
6:       nNodePerElement ← partitions[iPart]->getNNodePerElm();
7:       nElm ← nElm + nElmPart;
8:       connecSize ← connecSize + nElmPart * nNodePerElement;
9:    end for
10:   elmPtr ← new int[nElm+1];
11:   connec ← new int[connecSize];
12:   nElm ← 0; connecSize ← 0;
13:   for all partitions 0 < iPart < nPart do
14:      partitions[iPart] -> setConnec(&connec[connecSize]);
15:      nElmPart ← partitions[iPart] -> getNElm();
16:      nNodePerElement ← partitions[iPart]->getNNodePerElm();
17:      for all nodes in partition do
18:         connecSize ← connecSize + nNodePerElement;
19:         elmPtr[++nElm] ← connecSize;
20:      end for
21:   end for
22: end procedure
```

Algorithme 19 Distribution des éléments (Partition)

Input:

```
1: nElm := number of local element in geometric partition
2: connec := connectivity table of partition's elements
3: EFelm := MPI datatype that contains connectivity of 1 element

4: procedure PartitionGeo : :buildMsgElm(msg,owners)
5:    msg ← new MsgAllToAllv<int>(nullptr,nullptr,EFelm);
6:    msg -> buildTemplate(owners,nElm);
7:    msg -> allocateBuffer();
8:    msg -> BuildBuffer(connec,owner,nElm);
9:    nElm ← msg -> getMsgTemplate() -> getNItemRecv();
10: end procedure

11: procedure PartitionGeo : :DistributeElement(msg)
12:    msg -> BuildBuffer(connec,owner,nElm);
13:    msg -> Communicate();
14: end procedure
```

Algorithme 20 Nomination des noeuds maîtres et fantômes

Input:

```
1: nodes := list of nodes indices
2: getOwner := function that takes node index and return owner's rank

3: procedure PartitionManager : :DistributeNode(nodes ;)
4:    /* Communication */
5:    EF7_MsgAllToAllv<EFint>* msgNode = new EF7_MsgAllToAllv<EFint>();
6:    msgNode->build(nodes,nodes,nNode_,getOwner,initialNodeDisposition);
7:    msgNode->communicate();

8:    /* Attach nodes with their owners' rank */
9:    nNodeRecv = msgNode->getMsgTemplate()->getNItemRecv();
10:   nodesPair = new pair<EFint,int>[nNodeRecv];
11:   buffer ← msgNode -> getRecvBuffer();
12:   for all processes iProc in MPI group do
13:      nNode ← msgNode -> getMsgTemplate() -> getRecvCounts(iProc);
14:      index ← msgNode -> getMsgTemplate() -> getRecvDispls(iProc);
15:      for all nodes receive from process iProc do
16:         nodesPair[index+iNode] ← std::make_pair(buffer[index+iNode],iProc);
17:      end for
18:   end for
19:   std::sort(nodesPair,nodesPair+nNodeRecv);
20:   ChooseMasterAndGhostNodes(nodesPair,nodesPair+nNodeRecv);

21:   /* Communication */
22:   MsgAllToAllv<int>* msgMaster = new MsgAllToAllv<int>();
23:   MsgAllToAllv<int>* msgGhost = new MsgAllToAllv<int>();
24:   MsgAllToAllv<int>* msgOwner = new MsgAllToAllv<int>();

25:   msgMaster->build(masterNodes, masterNodeOwner, nNodeMaster);
26:   msgGhost->build(ghostNodes, ghostNodeTenant, nNodeGhost);
27:   msgOwner->build(ghostNodeOwner, ghostNodeTenant, nNodeGhost);

28:   msgMaster->communicate();
29:   msgGhost->communicate();
30:   msgOwner->communicate();

31:   /* Modify domain structures */
32:   domain->partitionning(msgMaster, msgGhost, msgOwner);
33: end procedure
```

Algorithme 21 Nomination des noeuds maîtres et fantômes (suite)

Input:

`begin , end` := pointer to the beginning and the end of nodesPair array
`func` := function that chooses one element in an array delimeted by 2 pointers
`compare1` := binary predicate that return `true` if first elements of 2 pair are equal
`compare2` := binary predicate that return `true` if first elements of 2 pair are not equal

Output:

`masterNodes` := list of master nodes
`masterNodeOwner` := list of master nodes
`ghostNodes` := list of ghost nodes
`ghostNodeTenant` := list of ghost nodes' tenant processes
`ghostNodeOwner` := ranks of ghost nodes' owner processes

```
procedure PartitionManager : :ChooseMasterAndGhostNodes(begin, end)
    nNodeMaster ← 0;
    nNodeGhost ← 0;
    it1 ← std:adjacent_find(it1,end,compare1); ▷ find first of consecutive nodes

    while it1 < end do
        it2 ← std:adjacent_find(it1,end,compare2);   ▷ last of consecutive nodes
        if it2 ≠ end then
            ++it2;
        end if
        itMaster ← func(it1,it2);
        masterNodes[nMasterNode] ← itMaster->first;
        masterNodeOwner[nMasterNode] ← itMaster->second;
        ++nNodeMaster;
        while it1 ≠ it2 do
            if it1 neq itMaster then
                ghostNodes[nNodeGhost] ← it1->first;
                ghostNodeTenant[nNodeGhost] ← it1->second
                ghostNodeOwner[nNodeGhost] ← itMaster->second;
                ++nNodeGhost;
            end if
            ++it1;
        end while
        it1 = it2;
        it1 ← std:adjacent_find(it1,end,compare1);
    end while
end procedure
```

Algorithme 22 Partitionnement des noeuds géométriques

```
1: procedure DOMAIN : :PARTITIONNING(msgMaster, msgGhost, msgOwner)
2:     Getghost2Master(msgMaster, msgGhost, msgOwner);

3:     from connectivity, define nodes, an array of nodes in virtual partition
4:     Renumbering(msgMaster, msgGhost, nodes);

5:     DistributeNode(nodes)
6: end procedure
```

Algorithme 23 Association des noeuds maitres avec les noeuds fantômes correspondant

Input:

1: masters := list of master nodes
2: ghosts := list of ghost nodes
3: owners := ranks of owner processes

Output:

4: list of ghost nodes sorted by owner's rank
5: direct association between master and ghost nodes

```
6: procedure GETGHOST2MASTER(masters, ghosts, owners)
7:     define a map masterGlobal2Local ;          ▷ Or a hash table for a direct access
8:     for all nodes in masters do
9:         insert node in masterGlobal2Local with node index as the key and loop index
           as the value
10:    end for

11:    sort ghosts by owner's rank
12:    send ghosts to rigthful owner with MPI_Alltoallv

13:    create a ghost2master array of same length as the receive buffer of MPI message
14:    for all ghost nodes in receive buffer do
15:        insert at ghost2master[loop index] the value of masterGlobal2Local[ghost
           node index]
16:    end for
17: end procedure
```

Algorithme 24 Renumérotation des noeuds

Input:

1: `masters` := list of master nodes
2: `ghosts` := list of ghost nodes sorted by owner
3: `allnodes` := list of all nodes in current virtual partition

4: **procedure** RENUMBERING(`masters, ghosts, allnodes`)
5: define the unary predicate `setFirst` returning `true` if the argument is a master node
6: define the unary predicate `setLast` returning `false` if the argument is a ghost node

7: /* Place master nodes first and ghost nodes last */
8: `ptr` ← `std::stable_partition(allnodesBeginPtr,allnodesEndPtr,setFirst)`
9: `ptr` ← `std::stable_partition(ptr,allnodesEndPtr,setLast)`

10: Overwrite ghost nodes with the list of ghost sorted by owner

11: /* rewrite connectivity */
12: define a map with old nodes' indices as keys and new indices as values
13: for all nodes in the connectivity table, switch their indices for the new ones
14: **end procedure**

Algorithme 25 Distribution des coordonnées des noeuds géométriques

Input:

1: `nodes` := list of nodes indices
2: `getOwner` := function that takes node index and return owner's rank
3: `LocalNodeOffset` := index of first node read by actual partition

4: **procedure** DISTRIBUTENODE(`nodes`)
5: `EF7_MsgAllToAllv<EFint>* msgNode = new EF7_MsgAllToAllv<EFint>()`
6: `msgNode->build(nodes,nodes,nNode_,getOwner,initialNodeDisposition)`
7: `msgNode->communicate()`

8: `function<int(EFint)> distribution1 = [&](EFint n)->intreturn msgNode->getRecvBuffer()[n]-localNodeOffset;;`
9: function<int(EFint)> distribution2 = [&](EFint n)->intreturn initialNodeDisposition[n];;
10: `MsgAllToAllv<double>* msgCoord = new MsgAllToAllv<double> (nullptr, nullptr, EFnode, *(msgNode->getMsgTemplate()));`
11: `msgCoord->inverse();`
12: `msgCoord->allocateBuffer();`
13: `msgCoord->buildBuffer(coords, distribution1);`
14: `msgCoord->communicate();`

15: reallocate coordinates table(`coords`)
16: `msgCoord->unwrapBuffer(coords, distribution2);`
17: **end procedure**

ANNEXE D DONNÉES ET RÉSULTATS SUPPLÉMENTAIRES

P	Normal	SSE	AVX
1	1.2824E-02	1.1107E-02	1.0897E-02
2	3.0245E-02	2.4929E-02	2.3458E-02
3	6.6926E-02	4.5893E-02	4.8560E-02
4	1.3753E-01	8.8438E-02	8.2810E-02
5	2.3701E-01	1.4878E-01	1.5108E-01
6	5.4763E-01	2.4888E-01	2.8711E-01
7	1.0461E+00	5.5924E-01	5.4269E-01
8	1.8874E+00	9.4066E-01	9.0615E-01
9	3.1689E+00	8.5870E-01	1.3869E+00
10	5.5084E+00	1.9373E+00	1.9337E+00
11	8.4448E+00	2.7494E+00	2.9548E+00
12	1.3965E+01	3.8270E+00	3.4949E+00
13	1.9076E+01	5.1992E+00	5.4663E+00
14	2.9410E+01	7.7115E+00	7.6821E+00
15	4.2758E+01	9.5144E+00	1.0527E+01

(a) Expression exacte

P	Normal	SSE	AVX
1	1.2824E-02	1.1107E-02	1.0897E-02
2	3.0245E-02	2.4929E-02	2.3458E-02
3	6.6926E-02	4.5893E-02	4.8560E-02
4	1.3753E-01	8.8438E-02	8.2810E-02
5	2.3701E-01	1.4878E-01	1.5108E-01
6	5.4763E-01	2.4888E-01	2.8711E-01
7	1.0461E+00	5.5924E-01	5.4269E-01
8	1.8874E+00	9.4066E-01	9.0615E-01
9	3.1689E+00	8.5870E-01	1.3869E+00
10	5.5084E+00	1.9373E+00	1.9337E+00
11	8.4448E+00	2.7494E+00	2.9548E+00
12	1.3965E+01	3.8270E+00	3.4949E+00
13	1.9076E+01	5.1992E+00	5.4663E+00
14	2.9410E+01	7.7115E+00	7.6821E+00
15	4.2758E+01	9.5144E+00	1.0527E+01

(b) Différence fini

Tableau D.1 Temps d'assemblage avec vectorisation pour l'équation de diffusion 2D

P	Normal	SSE	AVX
2	2.2210E-02	1.7237E-02	1.4992E-02
3	8.9472E-02	7.1017E-02	6.2014E-02
4	2.1747E-01	1.6247E-01	1.2754E-01
5	5.8632E-01	3.3596E-01	4.1792E-01
6	1.3750E+00	9.6652E-01	1.0081E+00
7	3.3242E+00	2.3394E+00	1.3968E+00
8	5.2686E+00	4.2335E+00	4.1585E+00
9	1.1352E+01	6.6996E+00	7.6261E+00
10	1.8885E+01	1.2758E+01	1.3774E+01
11	5.5175E+01	3.7808E+01	3.8412E+01
12	1.1948E+02	6.6954E+01	6.2955E+01
13	1.7896E+02	1.0807E+02	8.2707E+01
14	2.5279E+02	1.5894E+02	1.4503E+02
15	3.9166E+02	1.7185E+02	1.9828E+02

(a) Expression exacte

P	Normal	SSE	AVX
2	1.1702E-02	7.5460E-03	6.8940E-03
3	4.2086E-02	2.2241E-02	2.2193E-02
4	6.7839E-02	4.7738E-02	4.4936E-02
5	1.3166E-01	8.9376E-02	9.0270E-02
6	3.5922E-01	1.4002E-01	1.4658E-01
7	9.5330E-01	2.5808E-01	3.6519E-01
8	1.0692E+00	5.4641E-01	6.9837E-01
9	2.9727E+00	1.1519E+00	1.2881E+00
10	3.9351E+00	2.0372E+00	1.7378E+00
11	7.6547E+00	2.5795E+00	2.6761E+00
12	1.0904E+01	2.2779E+00	3.1147E+00
13	1.9464E+01	5.4218E+00	5.1552E+00
14	3.0399E+01	5.8001E+00	6.0689E+00
15	7.5454E+01	1.1181E+01	8.3294E+00

(b) Différence fini

Tableau D.2 Temps d'assemblage avec vectorisation pour l'équation de Navier-Stokes 2D