Conception et validation d'algorithmes de remaillage parallèles à mémoire distribuée basés sur un remailleur séquentiel

Cédric Lachat

UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS

ÉCOLE DOCTORALE STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE

pour obtenir le grade de

Docteur en Sciences

de l'Université de Nice-Sophia Antipolis

Spécialité : Informatique

présentée par

Cédric LACHAT

Conception et validation d'algorithmes de remaillage parallèles à mémoire distribuée basés sur un remailleur séquentiel.

Thèse dirigée par **Hervé GUILLARD & Laurent HASCOET**

Soutenue le 13 décembre 2013

devant le jury composé de

M.	Édouard AUDIT	Chercheur au CEA & Directeur de la MdS	Rapporteur
M^{mme}	Cécile DOBRZYNSKI	Maître de conférence, Bordeaux I	Examinateur
M.	Mike GILES	University of Oxford	Rapporteur
M.	Hervé GUILLARD	Directeur de recherche, INRIA, Sophia-Antipolis	Examinateur
M.	Laurent HASCOET	Directeur de recherche, INRIA, Sophia-Antipolis	Examinateur
M.	François PELLEGRINI	Professeur, Bordeaux I	Examinateur

Table des matières

Liste des tableaux

Table des figures

Liste des définitions

Introduction générale

Les simulations numériques utilisées dans de nombreux domaines, tels que l'aéronautique, l'automobile, la météorologie, la mécanique des fluides, l'astrophysique, ou la physique nucléaire, pour ne citer que ces domaines, s'appuient sur des discrétisations du domaine d'étude, que nous appellerons « maillages » dans la suite de ce manuscrit. Ces maillages sont un formalisme nécessaire pour évaluer, sur un nombre fini de points du domaine, des valeurs physiques du problème, appelées « solution ». Ces valeurs peuvent être par exemple une température, une pression, une vitesse ou encore un champ magnétique.

Cette solution est calculée en certains points correspondant aux nœuds du maillage. Or, la précision de la solution dépend fortement de la qualité du maillage. C'est pourquoi des techniques de remaillage sont utilisées dans le but d'améliorer la qualité des maillages. Or, les maillages considérés sont de tailles de plus en plus grandes, et représentent des géométries de plus en plus complexes. Il est maintenant courant que ces maillages dépassent plusieurs dizaines de millions de mailles. D'ores et déjà, de tels maillages ne peuvent être remaillés sur une seule unité de traitement, appelé processeur. Il est nécessaire, pour traiter de tels cas, de paralléliser les méthodes de remaillage, dans le cadre d'architectures à mémoire distribuée, les seules capables de passer à l'échelle. Ceci revient à utiliser simultanément plusieurs processeurs, chacun d'entre eux opérant sur une mémoire qui lui est propre. Le maillage sera ainsi subdivisé en plusieurs parties que nous appellerons « sous-domaines ». Le maillage devra être distribué de telle sorte que la charge de calcul portée par chaque processeur soit équilibrée, ce qui reviendra par exemple à équilibrer les sous-domaines en termes de nombre de mailles.

Nous constatons dans la littérature l'existence de deux grandes familles de méthode de remaillage parallèle. La première, dont l'étude a commencé à la fin des années 1990, a consisté à paralléliser les techniques élémentaires de remaillage existantes, telles que la subdivision ou le basculement d'arête, l'insertion ou la suppression de nœuds. Or, les techniques qui modifient la topologie du maillage sont difficiles à paralléliser, du fait de la synchronisation devant être mise en œuvre pour traiter les éléments situés à l'interface entre plusieurs sous-domaines. La deuxième famille de méthodes, pour laquelle les travaux ont débuté dans les années 2000, a cherché à contourner les difficultés rencontrées. Au lieu de paralléliser les techniques fortement séquentielles, elles privilégient leur réutilisation dans un environnement parallèle. Le problème consiste dès lors à rechercher et à extraire des zones du maillage distribué afin de les remailler concurremment, mais indépendamment les unes des autres, sur des processeurs individuels.

Nous pensons que la deuxième famille de remaillage parallèle permet un meilleur passage à l'échelle, autrement dit de traiter des maillages de plus en plus grand sans que l'efficacité de la méthode ne soit remise en cause. L'objectif de notre thèse a été, dans un premier temps, de valider les travaux commencés lors de notre stage de fin d'études d'ingénieur. Ceux-ci ont porté sur l'amélioration de la vitesse d'exécution des programmes séquentiels de remaillage, grâce à la renumérotation judicieuse des constituants des maillages. Dans un deuxième temps, nous avons conçu des structures de données suffisamment abstraites afin qu'elles puissent représenter

n'importe quel type de maillage. Ceci a nécessité d'identifier les besoins des numériciens, afin de concevoir des structures adaptées à l'écriture de multiples classes de solveurs numériques. Dans un troisième temps, nous avons élaboré des algorithmes dédiés au remaillage parallèle, tels que l'identification en parallèle des zones à remailler. Nous avons porté une très grande attention à ce que ces algorithmes puissent passer à l'échelle, afin qu'ils puissent être utilisés dans le cadre de simulations frontières dont l'objectif est de traiter des problèmes représentés par des maillages comprenant plus d'un milliard de mailles.

Notre exposé commencera par un état de l'art, qui porte à la fois sur le remaillage parallèle et l'équilibrage de charge, parce que ces deux notions ne peuvent être découplées, ainsi que sur la renumérotation des données en mémoire.

Au chapitre 2, nous introduirons les différentes notions que nous utiliserons au cours de cette thèse.

Les structures de données, et les bases nécessaire à la compréhension des algorithmes de remaillage parallèle seront présentées dans le chapitre 3.

La question de la renumérotation sera traitée dans le chapitre 4, ce qui nous donnera l'occasion de comparer nos méthodes avec l'état de l'art.

Le cœur de la thèse, autrement dit les algorithmes dédiés au remaillage parallèle, seront exposés dans le chapitre 5. Nous y présenterons les résultats de remaillage que nous obtenons pour générer des maillages de taille dépassant les 300 millions de mailles.

Notre conclusion nous permettra de récapituler les différentes contributions que nous avons apportées au cours de cette thèse. Nous exposerons pour finir les différentes perspectives de nos travaux, à court et à long termes.

Chapitre 1

Présentation du problème, état de l'art et positionnement de notre approche

Sommaire

1.1 Présentation du problème

En mécanique des fluides numérique (*Computational fluid dynamics*, ou CFD, en anglais), l'adaptation de maillages est un outil très puissant qui permet d'améliorer la qualité des résultats des simulations, tout en réduisant le coût de calcul grâce à un choix et un positionnement optimaux des inconnues. Cette technique devient particulièrement avantageuse quand elle est couplée avec des méthodes numériques à haute précision et fait l'objet d'une implémentation parallèle.

La simulation d'écoulement dans des géométries complexes nécessite la génération de maillages comportant au moins plusieurs dizaines de millions de nœuds, dont la construction et le stockage requièrent énormément de mémoire et qui ne peuvent donc, pour les plus gros, être intégralement stockés dans la mémoire d'un unique ordinateur. C'est pour cette raison que de nombreux travaux récents se sont intéressés à la génération de maillages en parallèle, par exemple [1, 34, 49, 101]. De plus, la plupart des solveurs étant aujourd'hui parallélisés, savoir mailler en parallèle permettrait de coupler les techniques d'adaptation de maillages avec les solveurs sans migrer le maillage sur un unique processeur pour y effectuer un maillage séquentiel. La principale difficulté de la génération de maillage en parallèle réside dans la gestion des interfaces, c'est-à-dire des éléments communs à plusieurs processeurs. En effet, un processeur ayant besoin d'insérer un point sur une arête interface devra communiquer avec les autres processeurs contenant cette arête, lesquels pourront, à l'inverse, avoir décidé de ne pas couper cette arête.

1.2 État de l'art

La génération de maillages de grandes tailles nécessite de paralléliser l'étape de remaillage à partir d'un maillage initial grossier. Afin que la méthode puisse passer à l'échelle, seul le parallélisme à mémoire distribuée est envisageable. Cela implique de distribuer les données sur les différents nœuds de calcul constituant la machine parallèle, mais aussi de communiquer entre les différents nœuds. En particulier, les méthodes de remaillage nécessitent des synchronisations entre les processeurs.

Le remaillage parallèle combine plusieurs techniques. Avant le remaillage proprement dit, le maillage doit avoir été distribué au moyen d'un partitionneur parallèle. Comme, après le remaillage, les sous-domaines remaillés peuvent être déséquilibrés en terme de taille, il est nécessaire de procéder à un rééquilibrage de la charge du nouveau maillage sur les nœuds de calcul. Enfin, la hiérarchie mémoire peut influencer la vitesse d'exécution d'un programme si elle n'est pas correctement exploitée. En effet, la vitesse d'exécution des programmes de remaillage dépend de la numérotation des différentes entités qui composent le maillage. Par conséquent, nous développerons notre état de l'art autour de quatre axes, qui sont : le partitionnement parallèle, l'équilibrage de la charge, le remaillage parallèle et la renumérotation.

1.2.1 Partitionnement parallèle

La problématique de partitionnement est plus connue dans le domaine des graphes, sachant qu'un maillage peut être représenté sous forme d'un graphe. De nombreux algorithmes existent : spectraux [90], combinatoires [52, 70], évolutionnaires [51], mais seuls les algorithmes multi-niveaux [24, 26] permettent d'obtenir des résultats de bonne qualité sur des graphes de grandes tailles. Nous pouvons citer comme logiciels parallèles implémentant ces schémas multi-niveaux : PT-Scotch [89], ParJostle [120], ou encore ParMeTiS [69] pour ce qui concerne

les partitionneurs de graphes ; Zoltan [10] ou encore Par*k*way [103] pour le partitionnement d'hypergraphes (voir section 2.4, page 20).

1.2.2 Équilibrage de charge parallèle

L'équilibrage de la charge est souvent résolu avec un partitionnement de graphe. Différentes représentations de graphes existent ; le graphe peut être construit à partir des éléments du maillage (on parle de graphe dual), à partir des nœuds (on parle de graphe nodal) ou encore à partir des éléments et des nœuds (nous parlerons alors, dans notre approche, de graphe enrichi).

Plusieurs travaux, comme [43, 57, 79], s'appuient sur des algorithmes de partitionnement de graphe afin de réaliser l'équilibrage de la charge de maillages.

En 1999, la bibliothèque DRAMA [79] (Dynamic Re-Allocation of Meshes for parallel finite element Applications) a été développée afin de mettre en œuvre des algorithmes de repartitionnement dynamique de maillages pour des applications parallèles sur des éléments finis non structurés. Au moins deux codes applicatifs ont été interfacés avec DRAMA : PAM-CRASH/PAM-STAMP et FORGE3. FORGE3 met en œuvre le raffinement et le déraffinement de maillages 3D. PAM-CRASH calcule différents coûts de calcul et de communication en fonction des interfaces entre les différents sous-domaines du maillage. PAM-STAMP met en œuvre le raffinement et le déraffinement, mais sans remaillage. Le repartitionnement dynamique doit d'abord répondre aux exigences du partitionnement statique, mais aussi à celles-là : prendre le partitionnement actuel en compte, et interagir avec l'application.

Dans DRAMA, deux techniques de repartitionnement ont été introduites : la migration de maillage combiné à un remaillage parallèle, et le repartitionnement de maillage global avec un repartionnement parallèle en utilisant des graphes et des algorithmes multi-niveaux, au moyen de bibliothèques externes comme ParMeTiS et Jostle. DRAMA contient également un partitionneur géométrique. Des fonctions de coût ont été mises en place pour prendre en compte les déséquilibres de la charge, les besoins de communications et les architectures de machines hétérogènes. Les coûts des noyaux de calcul du code appelant DRAMA peuvent être portés soit par les éléments soit par les nœuds. Les coûts de communication ne s'appliquent qu'entre éléments.

K. D. Devine et coll. [43], en 2005 ont proposé différents algorithmes d'équilibrage de la charge, implémentés dans le logiciel Zoltan [40, 41]. Ce logiciel permet de partitionner des données, indépendamment de leur nature, par le biais de partitionneurs de graphes sous forme de bibliothèques externes. Il permet également de les redistribuer.

S. Gepner et coll. [57], en 2010 ont montré que le remaillage impacte directement l'équilibrage de charge, et donc la qualité du partitionnement. C'est pourquoi, il faut rééquilibrer la charge dynamiquement. ParMeTiS est utilisé dans l'algorithme d'équilibrage de charge pour rechercher des distributions de nœuds. L'équilibrage de charge permet ainsi d'obtenir une scalabilité linéaire. Les tests ont été réalisé sur un solveur parallélisé, basé sur une méthode de distribution du résidu [39], avec un maximum de 80 processeurs.

D'autres travaux, comme [84], présentent des algorithmes d'équilibrage de la charge basés sur le calcul de la charge par processeur et en résolvant des équations représentant les coûts de migration.

PLUM [84] permet de rééquilibrer la charge tout en minimisant les échanges de données. Le seul inconvénient a priori est que les algorithmes utilisés construisent des matrices de similarité dont la taille croît quadratiquement au nombre de processeurs.

Enfin, nous pouvons citer une dernière famille d'équilibrage de la charge, alliant différentes techniques dont certaines sont basées sur des graphes.

ParFUM [74], fournit des outils d'équilibrage de la charge. Différentes stratégies ont été implémentées afin de résoudre le problème du déséquilibrage de charge. La première affecte aux processeurs les objets triés de manière décroissante vis-à-vis de leur coût de calcul. Autrement dit, l'objet de coût le plus élévé est attribué au processeur le moins chargé, et ce itérativement. La deuxième stratégie redistribue les objets des processeurs les plus chargés. La troisième repose sur l'utilisation du partitionneur de graphe METIS. ParFUM permet aussi de transférer des données d'un maillage à un autre, par le biais de l'outil appelé Rocrem.

1.2.3 Remaillage parallèle

Les remailleurs séquentiels sont confrontés à des limites physiques comme la mémoire par processeur et nécessitent de plus en plus de temps de calculs pour remailler des maillages de plus en plus gros. C'est pourquoi nous allons nous focaliser sur le remaillage parallèle, et plus particulièrement dans le cas de maillages composés de tétraèdres isotropes et anisotropes.

Afin de remailler des maillages de très grandes tailles, il est nécessaire que les algorithmes de remaillage opèrent sur une architecture à mémoire distribuée, à l'aide de processus MPI. Ce parallélisme peut être couplé à un parallélisme à mémoire partagée, sachant que sur un *cluster*, chaque nœud physique possède plusieurs cœurs.

Comme nous le verrons, le remaillage soit répond au problème de l'erreur numérique induite par le maillage, soit consiste à bouger des corps en mouvement au sein d'un maillage.

On peut distinguer deux grandes familles de remaillage parallèle. La première, qui semble le plus naturelle, consiste à paralléliser les algorithmes de remaillage séquentiel existants. Une parallélisation efficace pouvant traiter des maillages de grandes tailles n'est pas évidente, à cause de la complexité des algorithmes de remaillage et des coûts de synchronisation entre processeurs. C'est pourquoi, une deuxième famille de remaillage parallèle a été proposée, en utilisant les remailleurs séquentiels afin de fournir un algorithme de remaillage parallèle.

1.2.3.1 Parallélisation des algorithmes séquentiels

La première famille correspond à la parallélisation des techniques de remaillages, comme la triangulation de Delaunay [56], les méthodes frontales [86], et la subdivision d'arêtes [81].

Les premières recherches sur le remaillage parallèle ont été menées dans les années 1990 sur les maillages 2D, par José G. Castaños et L. Laemmer.

En 1996, José G. Castaños et John E. Savage [17] ont commencé à introduire la notion de remaillage parallèle à mémoire distribuée sur des maillages 2D. Le remaillage parallèle ne peut être dissocié du repartitionnement et de la migration de données, sachant qu'il peut entraîner des déséquilibres de la charge. La technique utilisée permet d'adapter le maillage plusieurs fois tout en conservant la correspondance entre les éléments créés entre deux remaillages. Elle s'applique sur des triangles en coupant la plus longue arête pour générer deux triangles. Dès lors, la problématique du remaillage des frontières entre sous-domaines, tout en gardant un maillage global consistant, a été émise. Chaque processeur peut modifier la frontière avec ses voisins. Afin que les maillages générés ne soient conformes, un algorithme de synchronisation est appliqué afin que deux processeurs P_i et P_j ne créent chacun un nœud au même endroit. Sont utilisées à la fois la correspondance avec le maillage original et des requêtes entre les deux processeurs P_i et P_j, afin de faire pointer l'un des deux nœuds créés sur l'autre.

La méthode de remaillage proposée par Lutz Laemmer [71] en 1997 concerne des maillages 2D définis par des primitives géométriques (points, lignes et polygones). Une fois la géométrie du domaine déterminée, celle-ci est bi-partitionnée récursivement. La coupe passe par le centre de gravité de la zone et l'axe de la coupe dépend d'une fonction calculant le coût induit par le remaillage. Chaque sous-domaine construit les nœuds sur la frontière de son sous-domaine en utilisant le même algorithme sur chaque processeur. Les nœuds ont ainsi les mêmes coordonnées d'un processeur à l'autre. Seuls les numéros des nœuds sont communiqués au sein d'un groupe de processeurs partageant une frontière.

Chaque sous-domaine est ensuite remaillé indépendamment des sous-domaines voisins. Aucun nœud n'est ajouté sur les frontières entre sous-domaines. Cette méthode passe fort bien à l'échelle, vu que les communications ne dépendent que du nombre de processeurs, mais sa limite réside dans l'apparition très nette, sur le maillage, de la coupe entre sous-domaines, ce qui peut impacter directement la convergence des simulations.

Le remaillage parallèle sur des maillages homogènes 3D a été abordé en 2000, par L. Oliker. Des recherches ont suivis menées par N. Chrisochoides, A. Casagrande et A. Cavallo.

En 2000, Leonid Oliker et coll. [84] ont présenté à la fois des techniques de remaillage parallèle implémentées dans 3D_TAG et une bibliothèque d'équilibrage de charge, appelée PLUM [83]. Trois techniques de subdivision d'arêtes sont utilisées pour le remaillage parallèle sur des tétraèdres, dont deux techniques pour des tétraèdres anisotropes et la dernière pour des tétraèdres isotropes.

On part du principe que le maillage est centralisé et partitionné à l'aide de ParMeTiS. Les structures de données sont basées sur les sommets, arêtes et tétraèdres du maillage. Chaque sommet et arête sur les frontières entre sous-domaines, possède la liste des processeurs qui les partagent. Cela permet de savoir avec qui communiquer lors de la subdivision d'une arête sur la frontière. L'inconvénient de la méthode de remaillage présentée est qu'elle peut être directement affectée par les zones à remailler. Comme le montrent certains tests, leur accélération est de 9.2 et de 19.2 sur 64 processeurs, parce que les zones à remailler ne sont pas réparties sur tout le domaine.

Nikos Chrisochoides [29] a proposé en 2003 une version parallèle de l'algorithme de la triangulation de Delaunay. Cette méthode repose sur la demande d'informations aux processeurs voisins contenant la face frontière pour laquelle la cavité de Delaunay [115] doit être étendue. La demande d'informations peut être faite de proche en proche si la zone souhaitée est partagée par plusieurs sous-domaines. Ceci entraîne d'une part beaucoup de communications et, d'autre part, d'attendre le retour de l'information pour construire la cavité. Les maillages générés sont de l'ordre de 16 millions d'éléments sur 32 nœuds de calcul.

Dans la méthode proposée par A. Casagrande [16] en 2005, il n'est pas nécessaire de garder la correspondance avec le maillage original. Ceci permet une meilleure flexibilité et une meilleure qualité des différentes étapes d'adaptation, comme il n'existe pas de lien entre les maillages avant et après remaillage. Le maillage est raffiné et déraffiné en fonction d'une métrique. Le raffinement permet d'insérer un nœud sur une arête trop longue, alors que le déraffinement supprime un nœud en fusionnant deux nœuds. Cette méthode, couplée à des techniques d'optimisation comme la permutation d'arêtes en 2D ou de faces en 3D, produit des maillages similaires à ceux obtenus par remaillage séquentiel. L'adaptation parallèle de maillage nécessite des techniques de partitionnement pour l'équilibrage de charge afin d'obtenir des sous-domaines équilibrés. Par ailleurs, des tris et des renumérotations parallèles sont requis pour maintenir la cohérence entre la numérotation par processeur et la numérotation globale des différentes entités du maillage. L'adaptation parallèle est appliquée globalement sur un maillage pré-partitionné, et nécessite

une numérotation locale et un réagencement à l'intérieur de chaque processeur, mais aussi une numérotation globale pour que l'adaptation aboutisse sur un maillage qui sera repartitionné.

La parallélisation des changements structurels tels que l'ajout ou la suppression de nœuds, est le point le plus difficile, particulièrement pour les parties du recouvrement. Le basculement d'arêtes et le basculement de faces peut être fait sur les interfaces. La suppression est souvent le plus dur à réaliser. Les procédures de lissage modifient peu la topologie interne du maillage, la parallélisation est par conséquent plus facile et se restreint à conserver la cohérence sur la numérotation des nœuds, faces, arêtes et éléments. Le remaillage ayant entrainé des déséquilibres suite à la suppression ou à l'ajout de nœuds, un repartitionnement est nécessaire en utilisant des algorithmes de partitionnement parallèle de graphes. Pour compléter l'adaptation parallèle, des techniques de renumérotation sont appliquées sur les entités : éléments, arêtes, faces et nœuds, tels que des algorithmes d'arbres de recherche.

Enfin, le remaillage parallèle a été adapté à des maillages mixtes 3D par O. S. Lawlor [74]. ParFUM [74], basé sur Charm++ [67,112], est une bibliothèque pour les maillages non structurés. Elle décompose le maillage en sous-parties, chacune étant attribuée à un processeur virtuel. Chaque processeur physique contient un ou plusieurs processeurs virtuels. Chaque processeur virtuel met en œuvre des techniques de remaillage, tout en préservant la conformité du maillage. Des opérations basiques sont disponibles, comme l'ajout ou la suppression d'un élément ou d'un nœud. ParFUM échange entre processeurs virtuels une partie suffisante du maillage si l'élément ou le nœud est sur la frontière. D'autres opérations modifient le maillage, comme le basculement, l'insertion ou la suppression d'arêtes. Des primitives plus complexes permettent de passer de deux éléments partageant une face à trois éléments partageant une face. Enfin, le raffinement de maillage implémenté est basé sur un algorithme de bisection de la plus longue arête [94].

Le problème des techniques de remaillage mises en avant dans ParFUM est qu'elles nécessitent un nombre non négligeable de communications entre les processeurs virtuels partageant une frontière. Ces communications peuvent être diminuées dans le cas où le nombre de processeurs physiques est inférieur à celui des processeurs virtuels, sachant que des structures de données partagées entre processus légers sont utilisées. ParFUM permet ainsi de combiner l'usage de la mémoire distribuée avec de la mémoire partagée dans le parallélisme.

1.2.3.2 Réutilisation des algorithmes séquentiels

Étant donné les difficultés rencontrées pour paralléliser les algorithmes de remaillage, une seconde famille de méthodes de remaillage parallèle a été développée peu de temps après le début de la première.

La réutilisation d'algorithmes séquentiels au sein de remaillage parallèle a commencé en 2000 avec les travaux de T. Coupez [34]. Ces travaux ont été suivis par ceux de C. Dobrzynski [49] et U. Tremel [101].

Thierry Coupez [34] a proposé en 2000 une méthode de remaillage parallèle basée sur un processus itératif. Lors de chaque itération, des techniques de remaillage local sont appliquées sans modifier les frontières entre sous-domaines. Entre chaque itération, les frontières entre sous-domaines migrent en repartitionnant le maillage, cette étape étant précédée d'un changement de poids sur les éléments proches de la frontière, afin que les éléments qui étaient sur la frontière puissent être remaillés lors de la prochaine itération.

Le repartitionnement est basé sur le partitionnement de graphe. Le graphe est calculé à

partir des connectivités du maillage. La bibliothèque utilisée, DRAMA [79], procure une interface robuste pour les partitionneurs de graphe comme Jostle [109] ou ParMeTiS [68], permettant de convertir le maillage en graphe pour que ce dernier soit partitionné, puis le graphe est reconverti en maillage. Selon eux, un maillage ne peut être complètement représenté par un graphe. Les connectivités du maillage ne peuvent être retrouvées par les relations nœud à nœud, ni par les relations élément à élément. La migration des éléments du maillage s'effectue directement dans DRAMA sans passer par un graphe intermédiaire [99]. Utilisant uniquement la connectivité du maillage, le repartitionnement est calculé par optimisation locale par paires de processeurs. L'algorithme de repartitionnement est basé sur deux étapes qui sont exécutées itérativement :

— appariement des processeurs [87] permettant de regrouper deux par deux les processeurs ;
— repartitionnement au sein d'une paire. Un processeur ne peut échanger qu'avec le processeur de sa paire.

Un repartitionnement contraint, autrement dit avec des poids biaisés, est nécessaire après avoir remaillé chaque sous-domaine afin de déplacer les interfaces et de pouvoir les remailler. Il est nécessaire de modifier la fonction de coût de communication, afin que les éléments aux frontières entre sous-domaines soient placés à l'intérieur d'un sous-domaine à la prochaine itération. Pour cette raison, les techniques de remaillage local sont plus facilement applicables, comme dans le code FORGE3 [23] par exemple. Cette approche est itérative et essentiellement basée sur des échanges locaux, en tirant parti du partitionnement.

Cette technique de remaillage parallèle a été reprise par Cécile Dobrzynski [49] et testée en utilisant MMG3D [50], aussi bien sur des maillages isotropes qu'anisotropes, en 2D et en 3D. Des maillages de plus de 2 millions de tétraèdres ont ainsi pu être remaillés sur 32 processeurs.

L'algorithme de remaillage parallèle proposé par U. Tremel et coll. [101], pour les maillages non structurés formé de tétraèdres, est également un processus itératif. Tout d'abord, toutes les régions à remailler en parallèle sont déterminées en analysant les éléments locaux selon différents critères de qualité. À cela s'ajoutent des vérifications d'intersection pour détecter les éléments se croisant entre eux à cause de mouvements locaux de surfaces ou de volumes du maillage, introduits par les objets en mouvement.

Toutes les régions continues sont identifiées [1] comme étant à remailler au moyen d'un algorithme de coloration parallèle. Le maillage distribué est repartitionné et redistribué pour que chaque zone soit entièrement comprise sur un sous-domaine, de telle sorte que les domaines soient équilibrés en termes de charge de remaillage mais aussi en termes de nombre d'éléments et de nœuds. Chaque domaine extrait les régions marquées et les remaille séparément en parallèle. Le remaillage est basé sur une triangulation de Delaunay contrainte. Le maillage est rééquilibré s'il se produit un déséquilibre de charge en terme de nombre d'éléments ou de nœuds par domaine, vis-à-vis d'un seuil défini par l'utilisateur.

Si les régions à remailler sont trop grandes au point qu'elles ne tiennent pas sur un processeur, le processus échoue. Des algorithmes alternatifs doivent être développés pour remailler en parallèle en utilisant des techniques locales de remaillage, appliqués à des volumes d'un maillage de tétraèdres non structurés. Une option envisagée serait un algorithme de génération de volume de maillages qui remplirait les zones après que les surfaces aient été remaillées. Les candidats possibles pour le remaillage de volume sont des méthodes de type octree, frontale ou de Delaunay. Des maillages au sein de l'application SimServer, représentant des corps en mouvement, ont ainsi pu être remaillés. Les tailles de ces maillages varient entre 2,1 à 2,3 millions de nœuds et 12,1 à 13,4 millions d'éléments.

En 2005, Peter A. Cavallo et coll. [19] ont présenté une méthode d'adaptation, mise en œuvre

1. Deux éléments à remailler partageant une face appartiennent à la même région.

dans le code CRUNCH CFD [61, 62]. Le remaillage s'applique sur des maillages mixtes non structurés et utilise les suppressions d'arêtes, la triangulation de Delaunay et des techniques de lissage [4] pour les tétraèdres de mauvaise qualité [2]. Chaque sous-domaine effectue le remaillage sans toucher aux interfaces. Ensuite, les interfaces sont migrées par paires de processeurs. La migration des frontières entre processeurs, de l'ordre de 2 à 5 bandes d'éléments, peut avoir pour conséquence de créer ou de supprimer un voisinage entre deux processeurs. C'est pourquoi, une recherche à l'aide d'octree [3] a été utilisée afin d'accéder rapidement au voisinage entre deux processeurs. Par ailleurs, l'adaptation du maillage entraîne un déséquilibre de charge résolu par ParMeTiS en utilisant une méthode de diffusion [96]. Les cas tests utilisés pour valider l'adaptation parallèle sont des corps en mouvement avec des maillages pouvant atteindre 2,7 millions d'éléments sur 16 cœurs.

La limite de cette méthode réside dans la migration des éléments, qui est dépendante de la décomposition du domaine.

Un autre aspect du remaillage parallèle consiste à pouvoir transporter les variables relatives au maillage original sur le maillage obtenu par remaillage.

H. Digonnet et coll. [45] proposent, en 2007, en plus d'une méthode de remaillage parallèle basée sur la réutilisation de code séquentiel et de bougé d'interfaces, une méthode de transport hiérarchique parallèle afin de copier des valeurs d'un maillage à un autre. La parallélisation du remaillage a été introduite dans le mailleur MTC [33] qui génère des maillages non structurés et non hiérarchiques. Quant au repartitionneur parallèle utilisé, il s'agit de DRAMA [6] qui permet de déplacer les interfaces. L'utilisation du mailleur en parallèle améliore l'encapsulation du code, avec la possibilité d'enrichir le remailleur séquentiel sans compétences particulières en programmation parallèle. Pour l'interpolation, il est nécessaire de trouver à quel élément de l'ancien maillage appartient chaque nœud du nouveau maillage. Un algorithme naïf en $\mathcal{O}(n^2)$ peut être ramené en $\mathcal{O}(nlog(n))$ si l'on utilise une structure arborescente de boîtes pour localiser plus rapidement à quel élément de l'ancien maillage appartient un nœud. En parallèle, le premier niveau de hiérarchie est donné par les sous-domaines, si on suppose que le partitionnement avant et après remaillage soient le plus proche possible.

Certaines simulations numériques nécessitent plusieurs maillages corrélés, chacun possédant cependant ses caractéristiques propres. M. Ramadan [92] étend la méthode de remaillage parallèle à ces maillages.

Afin de réduire le temps de calcul des simulations portant sur des processus incrémentaux tel que le tréfilage ou le laminage, une méthode avec deux maillages est proposée par M. Ramadan et coll. [92] en 2009 : un maillage fin pour stocker les résultats et réaliser les calculs thermiques, et un maillage grossier pour les calculs de mécanique. Ceci permet de tirer avantage des déformations locales qu'entraînent les processus incrémentaux en déraffinant uniquement le maillage mécanique et par conséquent diminuer le temps de calcul.

Cette méthode a été développée au sein de la bibliothèque Cimlib [46], code parallèle d'éléments finis dédié aux simulations de formation de matériaux et développée par le groupe CIM des Mines de ParisTech, CEMEF. Le maillage est partitionné en plusieurs sous-domaines, autant que de processeurs disponibles, et chaque processeur ne contient qu'une partie du maillage. Les opérations de remaillage sont réalisées en utilisant le générateur de maillage MTC [35]. La procédure est itérative. Le maillage est partitionné en plusieurs sous-domaines, séparés par des interfaces. Pendant la première itération de remaillage, effectuée en parallèle sur chaque proces-

2. On appelle ainsi les tétraèdres dont un angle est supérieur à 120 degrés.

seur, les interfaces entre sous-domaines restent inchangées, afin d'éviter toute communication entre sous-domaines. Ensuite, un nouveau partitionnement est calculé afin de déplacer les interfaces et de pouvoir les remailler à l'itération suivante. L'algorithme itère jusqu'à que tous les éléments aient été remaillés. Les transferts entre le maillage thermique et le maillage mécanique sont également effectués en parallèle. Les maillages thermique et mécanique étant partitionnés indépendamment, il n'y a donc aucune raison pour que les zones de déformations du maillage mécanique appartiennent au même processeur que celui qui possède la zone correspondante du maillage thermique.

Nous allons voir, au travers des travaux de D. G. Larwood et Y. Yilmaz, des méthodes qui génèrent un maillage 3D à partir de la géométrie.

En 2003, D. G. Larwood et coll. ont exposé une méthode [73] de génération parallèle utilisant un schéma de « diviser pour résoudre » [31, 95]. Son but est d'utiliser des algorithmes séquentiels dans un logiciel parallèle. Cela permet l'interchangeabilité de l'algorithme séquentiel de remaillage volumique. Le processeur maître découpe le maillage en produisant un plan 2D, qui partage le maillage en deux sous-domaines. Ensuite, les esclaves divisent les parties que leur envoie le maître. Le maître controle la décomposition du domaine en enregistrant les découpes des sous-parties. Le critère de coupe d'une partie est le nombre de faces par sous-domaine. Une fois le nombre de sous-domaines atteint, les différentes parties du maillage peuvent être remaillées indépendamment. Le modèle du logiciel est MPMD (*multiple program multiple data*). Les données de bord des sous-domaines sont passés du maître aux différents processus par MPI.

Si le domaine décomposé est uniformément composé d'éléments, alors la décomposition géométrique peut donner des sous-domaines ayant une charge à peu près égale en terme de faces de frontières. En revanche, si les problèmes nécessitent plus d'éléments sur certaines régions, comme pour les calculs de mécanique des fluides, le résultat pourrait alors être déséquilibré, tout comme le remaillage volumique. En revanche, le nombre de faces frontières sur un sous-domaine n'est pas directement corrélé au nombre de tétraèdres qui seront créés. Dans la stratégie proposée, il n'y a pas de communications entre les processeurs lors du remaillage volumique, ce qui permet de partager le domaine en un nombre de parties plus grand que le nombre de processeurs. Afin de diminuer les déséquilibres, une surdécomposition du domaine est effectuée pour le maillage volumique afin d'avoir un meilleur équilibrage de charge [28, 107]. Un graphe quotient (voir section 2.4, page 20) est construit, représentant les différentes parties du maillages et leurs dépendances. Un poids est calculé en chaque partie en fonction du coût induit par le remaillage volumique. Les parties peuvent être ensuite partitionnées puis distribuées. En utilisant MeTiS, un repartitionnement est effectué en un nombre de parties égal au nombre de processeurs. Sur le cas test d'un avion F16 constitué après remaillage de 76 millions de tétraèdres et une surdécomposition de 800 sous-domaines sur 8 processeurs, le déséquilibre de charge obtenu est de l'ordre de 100 000 éléments, grâce à la surdécomposition. Un autre cas test d'un maillage pour une simulation électromagnétique sur F16 remaillé à 500 millions d'éléments sur 32 processeurs, met en évidence que la méthode est valide pour des maillages de grandes tailles.

L'inconvénient de cette méthode est cependant l'apparition trop nette de coupes de partitionnement sur le maillage.

Y. Yilmaz et coll. [129] ont présenté un générateur parallèle de maillages tétraédriques, en se basant sur un générateur séquentiel. Le remailleur séquentiel est Netgen [122], un logiciel sous licence LGPL qui a un large domaine d'utilisation. La génération parallèle débute par une décomposition séquentielle sur un nœud, de la géométrie en un ensemble d'espaces. Le volume de chaque partie est ensuite maillé en parallèle. Trois méthodes ont été implantées. La première décompose la géométrie CAD et produit des sous-maillages de surface conformes qui sont ensuite

envoyés aux processeurs pour générer les volumes. Les deuxième et troisième méthodes sont des méthodes de raffinement qui utilisent les informations de la géométrie CAD. Sur Curie, un maillage à un milliard d'éléments a été obtenu en moins d'une minute. Elmer est un code *open-source* de simulation multi-physiques développé par CSC-IT Center for Science. Elmer utilise une méthode d'éléments finis pour résoudre les équations différentielles partielles. Elmer contient un simple générateur de maillage non structuré qui permet de générer des maillages possédant des géométries simples. Les développeurs d'Elmer ont besoin de la génération parallèle de maillage et du partitionnement de maillage. L'objectif du projet est de développer un logiciel permettant de générer des maillages non structurés possédant plusieurs centaines de millions d'éléments et des géométries complexes, qui ne peuvent être induits par les générateurs séquentiels. Un autre objectif de la génération parallèle est de diminuer grandement le temps de génération.

1.2.4 Renumérotation pour augmenter la vitesse d'exécution des programmes.

Différents travaux concernent la renumérotation. Elle est abordée sur les maillages comme pré-traitement avant d'exécuter un code simulation numérique, par R. Löhner, puis directement dans la génération de maillages, par N. Amenta et M. Isenburg. En revanche, d'autres techniques possèdent un champ d'application plus large, comme celle mise en place par A. Loseille.

En 1993, R. Löhner [75] a introduit des stratégies de renumérotation, en se basant sur l'algorithme Cuthill-McKee inverse (RCM) [110], comme pré-traitement aux programmes de simulation numérique. Les stratégies utilisées ont permis de diviser le temps d'exécution d'un code avec des éléments finis. Ces techniques ont été reprises en 1997 par D.A Burgess et M.B. Giles [14,76]. Selon eux, il est recommandé que les générateurs de maillages incorporent un algorithme RCM qui optimise les performances de la mémoire hiérarchique. En 1998, R. Löhner [76] a présenté deux techniques de renumérotation destinées à des machines à mémoire partagée. La première donne des gains sur un faible nombre de processeurs, alors que la deuxième, qui consiste à modifier le programme afin d'intégrer la renumérotation, passe mieux à l'échelle.

En 2003, Nina Amenta et coll. [2] ont proposé et validé une méthode d'insertion de points au sein d'un triangulation de Delaunay. En règle général, lorsqu'un cercle de Delaunay est formé, l'insertion des points se fait aléatoirement. Or, cela n'est pas optimal en terme de localité de la mémoire au sein des ordinateurs. Ils utilisent un BRIO (*Biased randomized insertion order*) afin d'insérer dans un certain ordre les nœuds par la méthode de triangulation de Delaunay, sans l'altérer. Deux principaux atouts peuvent être mis en évidence : le temps gagné (divisé par dix dans certains cas), ainsi que la taille des maillages qui peuvent être traités (2,5 million de nœuds au lieu de 250 000).

Dans leur article de 2006 [64], Martin Isenburg et coll. utilisent une méthode à flux continu, permettant de générer de gros maillages à partir de machines avec de faibles ressources en mémoire. La particularité, outre le fait qu'ils utilisent entre autre un algorithme BRIO comme auparavant, est que leur algorithme de remaillage, permet d'optimiser l'efficacité du cache de par le peu de mémoire nécessaire.

Shrimp [127], développé par A. Loseille et F. Alauzet [77], utilise des courbes de Hilbert afin de renuméroter le maillage. Les courbes de Hilbert [118] sont un exemple de courbes de remplissage de l'espace introduites au siècle dernier. Elles ont permis de prouver qu'une courbe peut remplir une surface. Une géométrie en 3D est représentée dans la mémoire d'un ordinateur

comme un domaine en 1D. Les courbes de Hilbert constituent une représentation 1D de la géométrie 3D. En utilisant les courbes de Hilbert, Shrimp permet de réduire les défauts de cache (voir section 4.2, page 58) et d'accroître la vitesse d'exécution du remailleur. Une accélération de 10 a été observée avec MMG3D.

1.3 Positionnement, mise en œuvre et protocole expérimental

Nous allons commencer par discuter des structures de données nécessaires à la représentation des maillages distribués. Nous détaillerons ensuite comment nous nous positionnons sur le remaillage parallèle, et sur la renumérotation.

1.3.1 Structure de données pour la représentation des maillages distribués

Nous regrettons de ne pas avoir eu accès à la bibliothèque DRAMA, qui présente de nombreuses similarités avec le sujet de notre thèse. Notre regret est d'autant plus fort qu'il s'agit d'un logiciel libre financé par un projet européen.

Concernant notre positionnement face aux logiciels ou bibliothèques existants, nous avons décidé de développer une autre bibliothèque pour les raisons suivantes. D'une part, les structures de données des outils actuels ne sont pas assez généralistes ou ne sont pas adaptés à la manipulation efficace de maillages de grandes tailles. D'autre part, elles ne conviennent pas non plus à la mise en œuvre du remaillage parallèle.

Nous testerons nos structures de données au moyen d'une implémentation parallèle de méthodes numériques telles que la diffusion isotrope sur des maillages formés de tétraèdes.

1.3.2 Remaillage parallèle

Comme nous l'avons vu, il existe deux familles sur le remaillage parallèle : la première qui parallélise les algorithmes de remaillage et la deuxième qui, à travers un processus itératif, utilise les remailleurs séquentiels. Nous pensons que seule la deuxième famille permet de remailler efficacement des maillages de grandes tailles, parce que la première famille nécessite un nombre de communications dépendant de la taille des frontières entre sous-domaines. C'est pourquoi nous avons choisi la deuxième famille qui consiste à réutiliser des remailleurs séquentiels. Nous voulons que cette technique puisse être utilisée dans le cadre de calculs haute performance sur des machines parallèles pétaflopiques.

Par ailleurs, nous souhaitons que notre méthode soit la plus générique possible afin qu'elle puisse être utilisée sur des maillages constitués de différents types d'éléments et qu'il soit facile de coupler notre technique avec n'importe quel programme séquentiel de remaillage.

Le remaillage parallèle sera testé sur des maillages tétraédriques isotropes et anisotropes, en couplant notre méthode avec le remailleur séquentiel MMG3D basé sur une triangulation de Delaunay.

1.3.3 Renumérotation pour augmenter la vitesse d'exécution des programmes.

La numérotation des différentes entités d'un maillage influence significativement la vitesse d'exécution du remailleur qui le modifie. Nous pensons que la renumérotation ne peut être dissociée du remaillage parallèle. Le problème des renumérotations que nous avons vu, nécessitent soit de modifier le remailleur, soit n'exploitent pas les machines actuelles bénéficiant de plusieurs cœurs, comme par exemple les courbes de Hilbert. Nous proposons une alternative qui consiste

à former des groupes d'éléments à l'aide d'un partitionneur de graphes, puis de renuméroter les entités par groupe. Ces méthodes de renumérotation seront testées au sein du remailleur MMG3D et comparé avec Shrimp.

L'objectif de notre travail est d'intégrer, sous forme d'une bibliothèque, des algorithmes robustes de remaillage parallèle et d'équilibrage de la charge dynamique. Ces algorithmes devront être applicables à n'importe quel type de maillages : tétraèdres, pentaèdres, hexaèdres, etc., donnant au numéricien des outils efficaces et adaptés à la manipulation de maillages non structurés de grandes tailles, indépendamment du type des éléments.

Chapitre 2

Définitions pour la gestion de maillages distribués

Sommaire

Ce chapitre introduit les notions de graphes et de partitionnement de graphes qui nous seront utiles tout au long de cette thèse. Commençons tout d'abord par préciser certains termes qui seront employés tout au long de ce document.

2.1 Vocabulaire et notations

Nous utiliserons les notations ensemblistes classiques, ainsi que le vocabulaire français qui leur est habituellement associé. Afin d'éviter toute ambiguïté, nous rappelons les principaux termes qui seront utilisés et leur signification.

- L'expression « supérieur à » (respectivement « inférieur à ») signifie en fait « supérieur ou égal à » (respectivement « inférieur ou égal à ») ; dans le cas contraire, on dit « strictement supérieur à ». De même, un nombre sera dit « positif » si et seulement si il est supérieur (donc éventuellement égal) à 0 (équivalent au terme anglais « *non-negative* »).
- $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{R}$ et $\mathbb{C}$ désigneront respectivement l'ensemble des nombres entiers naturels (positifs), entiers relatifs, des nombres réels et des nombres complexes.
- Pour tout nombre réel x, la notation $\lceil x \rceil$ (respectivement $\lfloor x \rfloor$) désignera la partie entière supérieure (respectivement inférieure de x).
- La notation E^*, par exemple $\mathbb{N}^*$, signifiera que l'ensemble E est privé de l'élément 0.
- $|E|$ désignera le cardinal de l'ensemble E, c'est-à-dire son nombre d'éléments s'il s'agit d'un ensemble fini, $+\infty$ sinon.
- Pour un ensemble $X = \{x_i\}_{i \in [\![1;N]\!]}$, et une fonction $f : X \to \mathbb{R}$, la notation $\bar{f}$ désigne la moyenne arithmétique de f sur X :

$$\bar{f} = \frac{1}{N} \sum_{i=1}^{N} f(x_i) = \underset{x \in X}{\text{moy}}\, f(x) \ . \tag{2.1}$$

Pour évaluer, ou indiquer la complexité des algorithmes présentés nous utiliserons la notation de Landau au voisinage de $+\infty$ avec $\mathcal{O}$, Θ ou Ω.

Définition 1 (Notation de Landau)
Soient f et g deux fonctions de $\mathbb{R}$ dans $\mathbb{R}$. Le fait que f soit dominée (respectivement minorée) par g au voisinage de $+\infty$ est noté $f = \mathcal{O}(g)$ (respectivement $f = \Omega(g)$), tandis que le fait que f soit asymptotiquement équivalente à g en $+\infty$ est noté $f = \Theta(g)$.

$$f = \mathcal{O}(g) \quad \Leftrightarrow \quad \exists k \in \mathbb{R}_+^*, \exists X \in \mathbb{R}, \forall x \in \mathbb{R}, (x > X \Rightarrow |f(x)| \leq k|g(x)|) \ , \tag{2.2}$$

$$f = \Omega(g) \quad \Leftrightarrow \quad \exists k \in \mathbb{R}_+^*, \exists X \in \mathbb{R}, \forall x \in \mathbb{R}, (x > X \Rightarrow |f(x)| \geq k|g(x)|) \ , \tag{2.3}$$

$$f = \Theta(g) \quad \Leftrightarrow \quad f = \mathcal{O}(g) \text{ et } f = \Omega(g) \ . \tag{2.4}$$

Concernant le vocabulaire informatique, nous utiliserons les conventions usuelles concernant les tailles des objets, c'est-à-dire, par exemple, 1 kio = 1024 octets. Lorsque nous parlerons de processeur, nous évoquerons une unité de traitement, qui pourra correspondre à un cœur ou à un processeur logique dans le cas de l'utilisation d'une architecture permettant le *multi-threading*.

2.2 Vocabulaire du parallélisme

Dans la suite, nous considérerons que l'on dispose de p processeurs.

Définition 2 (Temps effectif parallèle, accélération)
Le temps effectif parallèle $t_p(p)$ est la durée que l'utilisateur attend pour qu'un programme parallèle termine sur p processeurs. On notera $t_p(p)$ en t_p.

Le temps séquentiel t_s est le temps mis pour résoudre le même problème avec le meilleur algorithme séquentiel connu.

L'accélération (« speed-up » en anglais) a_p correspond au rapport t_s sur $t_p(p)$:

$$a_p(p) = \frac{t_s}{t_p(p)} \ . \tag{2.5}$$

Sur des machines classiques (non quantiques [3]) utilisant des algorithmes déterministes, l'accélération a_p est majorée par p. En effet, la capacité de calcul avec p processeurs étant p fois plus élevée que celle avec un seul processeur, le temps d'exécution peut être au plus divisé par p. Cette remarque nous permet de définir la notion d'efficacité.

Remarque : Nous appellerons $t_p(1)$ le temps séquentiel de référence correspondant au temps effectif parallèle sur un processeur. Le temps $t_p(1)$ est différent du temps t_s car le premier concerne le programme parallèle alors que le second se réfère au meilleur algorithme séquentiel connu.

Définition 3 (Efficacité, rendement)
L'efficacité (ou rendement) η_p est définie comme étant le quotient entre a_p et p. Elle est donc majorée par 1.

$$0 \leq \eta_p = \frac{a_p}{p} \leq 1 \ . \tag{2.6}$$

Nous dirons qu'un programme est *scalable* en temps quand son efficacité tend vers 1 lorsque le nombre de processeurs p tend vers $+\infty$. Par extension, nous dirons qu'un programme est scalable lorsque son temps d'exécution avec p processeurs t_p est divisé par un facteur p par rapport au temps séquentiel de référence $t_p(1)$ [4]. De même, nous dirons qu'un programme est scalable en mémoire lorsque l'occupation en mémoire par processeur lors d'une exécution avec p processeurs est p fois inférieure à celle obtenue en effectuant le calcul sur un seul processeur.

2.3 Définitions sur les graphes

Nous allons maintenant définir les objets sur lesquels nous allons principalement travailler, c'est-à-dire les graphes et les maillages.

Définition 4 (Graphe non-orienté)
Un graphe non-orienté $G = (V, E)$ est une structure composée d'un ensemble V de sommets, et d'une collection $E \in V \times V$ de paires de sommets, appelées arêtes. $V(G)$ et $E(G)$ désigneront respectivement l'ensemble des sommets et la collection des arêtes de G.

Dans la suite, nous noterons $n_V = |V|$ le nombre de sommets et $m_E = |E|$ le nombre d'arêtes de G.

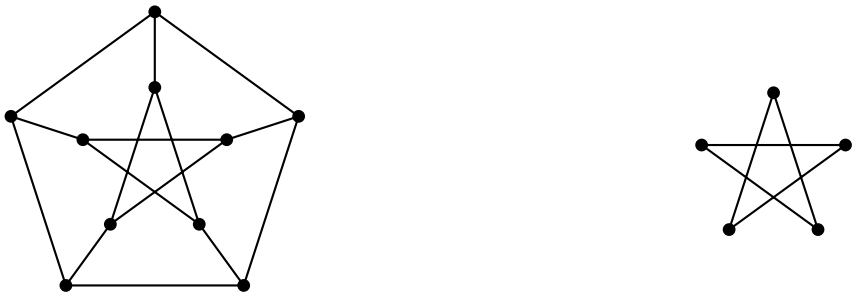
Une arête $\{u, v\}$ est dite incidente à u et à v. u et v sont les extrémités de $\{u, v\}$ et sont dits adjacents. Une arête de la forme $\{u, u\}$ est appelée boucle. Une arête existant en plusieurs exemplaires dans la collection des arêtes est une arête multiple.

3. Le calculateur quantique [126] ou ordinateur quantique repose sur des propriétés quantiques de la matière : superposition et intrication d'états quantiques.

4. Nous utilisons $t_p(1)$ et non t_s car nous devons comparer les temps du programme parallèle avec p processeurs et 1 processeur.

Définition 5 (Graphe simple)
Un graphe simple est un graphe sans boucles ni arêtes multiples.

Dans la suite, nous ne considérerons, sauf mention contraire, que des graphes simples non vides, c'est-à-dire comportant au moins un sommet, et nous parlerons donc d'ensemble d'arêtes pour $E(G)$. La figure 3.9, page 43 présente deux exemples de graphes simples.



(a) Un exemple de graphe simple : le graphe de Petersen.

(b) Un sous-graphe du graphe de Petersen.

FIGURE 2.1 – Exemples de graphes.

Définition 6 (Degré d'un sommet)
Soit u un sommet du graphe G. Le degré de u, noté $\delta(u)$, est le nombre d'arêtes de $E(G)$ incidentes à u.

Le degré minimal (respectivement maximal) de G, noté $\delta(G)$ (respectivement $\Delta(G)$) est le minimum (respectivement maximum) des degrés de tous les sommets de G. Le degré moyen de G, noté $\bar{\delta}(G)$, est la moyenne arithmétique des degrés de tous les sommets appartenant à $V(G)$.

$$\delta(G) \stackrel{\text{def}}{=} \min_{u \in V(G)} \delta(u) \ , \tag{2.7}$$

$$\Delta(G) \stackrel{\text{def}}{=} \max_{u \in V(G)} \delta(u) \ , \tag{2.8}$$

$$\bar{\delta}(G) \stackrel{\text{def}}{=} \frac{\sum_{u \in V(G)} \delta(u)}{|V(G)|} \ . \tag{2.9}$$

Définition 7 (Chemin)
Un chemin entre deux sommets u et v est une suite $\{\{w_1, w_2\}, \{w_2, w_3\}, \ldots, \{w_{k-1}, w_k\}\}$ d'arêtes de $E(G)$ telle que $u = w_1$ et $v = w_k$. Le nombre d'arêtes de la suite est appelé longueur du chemin.

Nous noterons $\mathcal{C}_G(u, v)$ l'ensemble des chemins de G entre u et v.

Définition 8 (Connexité)
Un graphe G est dit connexe lorsqu'il existe un chemin entre toute paire de sommets.

Un ensemble C de sommets tel qu'il soit un sous-ensemble maximal de $V(G)$ et tel que chaque paire de sommets soit relié par au moins un chemin est appelé composante connexe du graphe G.

Définition 9 (Arbre)
Un arbre est un graphe connexe muni de $|V| - 1$ arêtes.

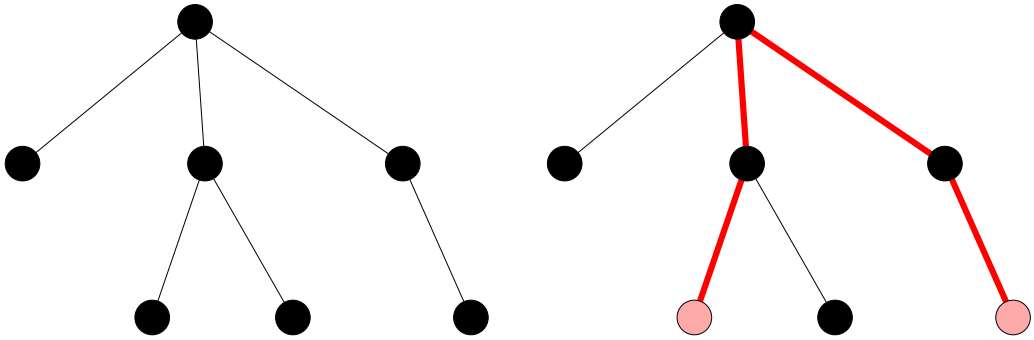
Un arbre à n sommets est donc le plus petit graphe, au sens du nombre d'arêtes, connexe à n sommets. La figure 2.2a représente un arbre.

Définition 10 (Distance dans un graphe)
La distance entre deux sommets u et v est la longueur du plus court chemin entre u et v, s'il en existe un, ou $+\infty$ sinon. Elle est notée $d(u, v)$.

Définition 11 (Diamètre d'un graphe)
Le diamètre d'un graphe G, noté $d(G)$ est égal au maximum de la distance entre deux sommets de G.

$$d(G) \stackrel{\text{def}}{=} \max_{(u,v)\in V_{(G)}^2} d(u, v) \ . \tag{2.10}$$

L'arbre de la figure 2.2b a donc un diamètre égal à 4.



(a) Un exemple d'arbre.

(b) Un chemin de longueur maximale (4) dans cet arbre. Ses extrémités sont en clair et les arêtes empruntées sont en rouge.

FIGURE 2.2 – Un exemple d'arbre de diamètre 4.

Définition 12 (Sous-graphe)
Un sous-graphe $H(U, F)$ de $G(V, E)$ est un graphe tel que :
- *U est un sous-ensemble de V : $U \subseteq V$;*
- *F est la restriction de E aux couples d'éléments de $U \times U$: $F = E \cap (U \times U)$.*

On dit que H est le sous-graphe de G induit par U et on le note $G|U$. Un exemple de sous-graphe est donné en figure 2.1b, page ci-contre.

Définition 13 (Graphe pondéré)
Un graphe $G(V, E)$ est dit pondéré lorsqu'il satisfait l'une au moins des deux conditions suivantes :
- *il existe une fonction $w_V : V \to \mathbb{R}$ associant à chaque sommet u son poids $w_V(u)$; on dit dans ce cas que les sommets de G sont valués ;*
- *il existe une fonction $w_E : E \to \mathbb{R}$ associant à chaque arête $\{u, v\}$ son poids $w_E(\{u, v\})$; on dit alors que les arêtes de G sont valuées.*

Dans la suite, nous ne considérerons que des graphes pondérés, en utilisant les fonctions constantes égales à 1 lorsque les graphes ne sont pas à sommets ou arêtes valués.

Définition 14 (Matrice d'adjacence)
La matrice d'adjacence d'un graphe G à n sommets est la matrice $A = (a_{uv})_{(u,v)\in[\![1;n]\!]^2}$ définie par :

$$\forall (u,v) \in [\![1;n]\!]^2, a_{uv} = \begin{cases} 0 & si\ \{u,v\} \notin E(G)\ , \\ 1 & si\ \{u,v\} \in E(G)\ . \end{cases} \tag{2.11}$$

Nous allons maintenant traiter de la coloration des graphes.

Définition 15 (Coloration d'un graphe)
La k-coloration d'un graphe $G(V,E)$ est la définition d'une fonction $f : V \to [\![1;k]\!]$ telle que :

$$\forall (u,v) \in V^2, \{u,v\} \in E \Rightarrow f(u) \neq f(v)\ . \tag{2.12}$$

La valeur $f(u)$ est appelée couleur du sommet u. Un graphe admettant une k-coloration est dit k-coloriable. On peut remarquer qu'un graphe de taille n est forcément n-coloriable.

2.4 Définitions du partitionnement de graphes

La notion de graphe quotient nous sera aussi utile dans la suite de cet ouvrage, plus particulièrement avec le partitionnement de graphes.

Définition 16 (Graphe quotient)
Soient $G(V,E)$ un graphe et $\mathcal{R}$ une relation d'équivalence pour les sommets de $V(G)$. On appelle graphe quotient le graphe $G_{|\mathcal{R}}$ défini de la façon suivante :

1. *les sommets de $G_{|\mathcal{R}}$ sont les classes d'équivalence sur $V(G)$. Si s est la surjection canonique de $V(G)$ dans $V(G)/\mathcal{R} = V(G_{|\mathcal{R}})$, alors on a :*

$$v' \in V(G_{|\mathcal{R}}) \iff \exists v \in V, s(v) = v'\ ; \tag{2.13}$$

2. *son ensemble d'arêtes est décrit par la relation :*

$$\forall \{u,v\} \in E(G), \{s(u), s(v)\} \in E(G_{|\mathcal{R}}) \Leftrightarrow \neg u \mathcal{R} v\ . \tag{2.14}$$

Un exemple de graphe quotient est visible en figure 2.3b. Pour ce dernier, la relation d'équivalence est « possède la même couleur », dans le cadre du coloriage présenté en figure 2.3a.

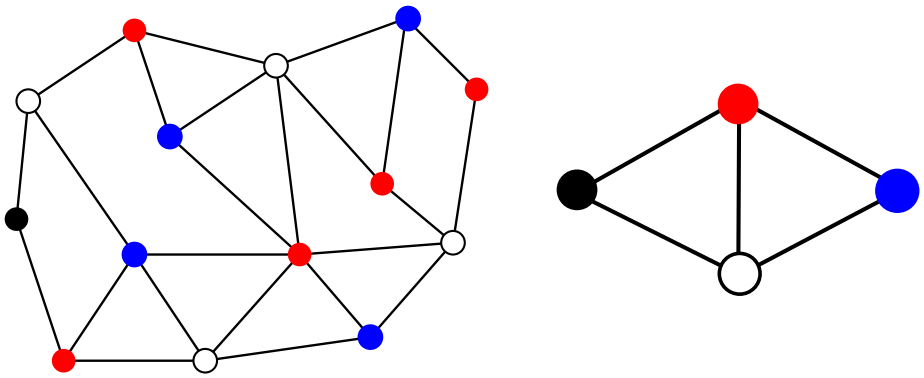
Comme nous évoquerons aussi quelquefois l'extension de notre problème de partitionnement de graphes aux hypergraphes, il est utile de préciser leur nature.

Définition 17 (Hypergraphe)
Un hypergraphe $H = (V, \mathcal{E})$ est une structure composée d'un ensemble V de sommets et d'un ensemble $\mathcal{E}$ de sous-ensembles de V appelés hyper-arêtes.

$V(G)$ et $\mathcal{E}(G)$ désigneront respectivement l'ensemble des sommets et l'ensemble des hyper-arêtes de G.

Un graphe est donc un hypergraphe dont chaque hyper-arête ne comporte que deux sommets.

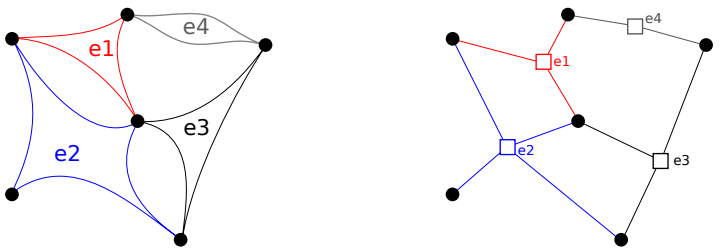
Définition 18 (Partition d'un ensemble)
Soit un ensemble E quelconque. Un ensemble Π de sous-ensembles de E est une partition de E si et seulement si :



(a) Exemple d'une 4-coloration d'un graphe.

(b) Graphe quotient du graphe précédent, muni de la relation d'équivalence « possède la même couleur ».

FIGURE 2.3 – Exemple de coloriage d'un graphe et du graphe quotient associé à cette coloration.



(a) Un exemple d'hypergraphe.

(b) Une autre représentation du même hypergraphe.

FIGURE 2.4 – Deux représentations d'un hypergraphe.

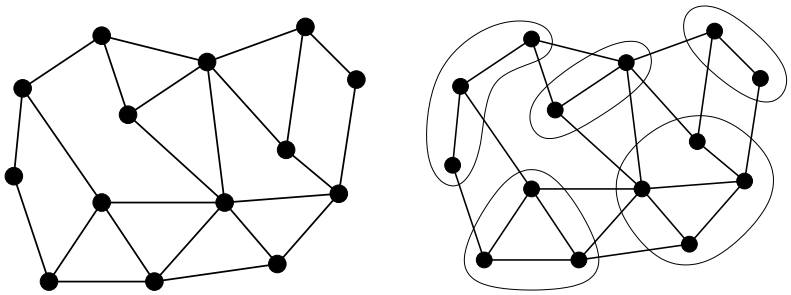
1. *Aucun élément de* Π *n'est vide ;*
2. *L'union des éléments de* Π *est égale à* E *;*
3. *Les éléments de* Π *sont deux à deux disjoints :* $\Pi_i \cap \Pi_j = \emptyset \Leftrightarrow i \neq j$.

Les éléments de Π *sont appelés les parties de la partition.*

Nous pouvons noter qu'une k-coloration du graphe G peut être vue comme une k-partition du graphe G avec des conditions particulières sur les relations entre les parties.

Nous noterons, pour tout sommet v de G, $\pi(v)$ la partie de Π_V contenant v et $\mathcal{P}(G)$ l'ensemble de toutes les partitions de $V(G)$. Un exemple de partition d'un graphe G est fourni en figure 2.5, page suivante. On peut remarquer que le nombre $|\mathcal{P}_k(G)|$ de k-partitions du graphe G est asymptotiquement minoré par $\Omega(k^n)$. Trouver la meilleure partition satisfaisant un certain critère en parcourant l'ensemble des partitions de G est donc matériellement impossible pour la plupart des graphes.

Le poids d'une partie π d'une partition Π du graphe $G(V, E)$ est égal à la somme des poids des sommets appartenant à cette partie π. On le note $w_V(\pi)$.



(a) Un graphe G. (b) Une partition Π de G.

FIGURE 2.5 – Partition d'un graphe.

Soit $\Pi \in \mathcal{P}(G) = (V_i)_{i \in [\![1;k]\!]}$ une partition de G. Parmi les arêtes de G, on distingue, relativement à une partie V_i, trois ensembles distincts :

1. $E_I(V_i)$, l'ensemble des arêtes internes associées à V_i, c'est-à-dire l'ensemble des arêtes ayant leurs deux extrémités dans V_i (*i.e.* l'ensemble des arêtes du sous-graphe $G|V_i$) ;
2. $E_F(V_i)$, l'ensemble des arêtes frontières associées à V_i, c'est-à-dire l'ensemble des arêtes ayant exactement une extrémité dans V_i ;
3. $E_E(V_i)$ l'ensemble des arêtes externes associées à V_i, c'est-à-dire l'ensemble des arêtes n'ayant aucune extrémité dans V_i.

Pour toute partie V_i, la famille $(E_I(V_i), E_F(V_i), E_E(V_i))$ est une partition arête Π_E du graphe G. Un exemple illustratif de ces ensembles est visible en figure 2.6a, page ci-contre.

Par extension, on définit :

$$E_I(\Pi) \stackrel{\text{def}}{=} \bigcup_{\pi \in \Pi} E_I(\pi) \,; \tag{2.15}$$

$$E_E(\Pi) \stackrel{\text{def}}{=} E(G) - E_I(\Pi) = \bigcup_{\pi \in \Pi} E_F(\pi) \,, \tag{2.16}$$

les ensembles qui représentent respectivement les arêtes internes et externes à l'ensemble des parties de la partition. L'ensemble des arêtes externes $E_E(\Pi)$ est très souvent noté $S(\Pi)$ et est appelé séparateur du graphe G pour la partition Π. Ces deux ensembles sont illustrés en figure 2.6b, page suivante.

Dans la pratique, un tel partitionnement est appelé partitionnement arête car les interfaces entre les différents ensembles de sommets correspondent à des ensembles d'arêtes.

Il existe aussi un k-partitionnement sommet du graphe G qui consiste à partitionner $V(G)$ en une famille $(V_i)_{i \in [\![1;k]\!]}$ d'ensembles de sommets d'une part, et un ensemble S de sommets d'autre part, tel qu'il n'existe pas d'arête $\{u, v\}$ reliant un sommet u de V_i à un sommet v de V_j lorsque $i \neq j$. Pour toute partie V_i, toute arête $\{u, v\}$ de $E_F(V_i)$ a donc une extrémité dans S. La partie S est appelée séparateur du graphe G pour le partitionnement Π.

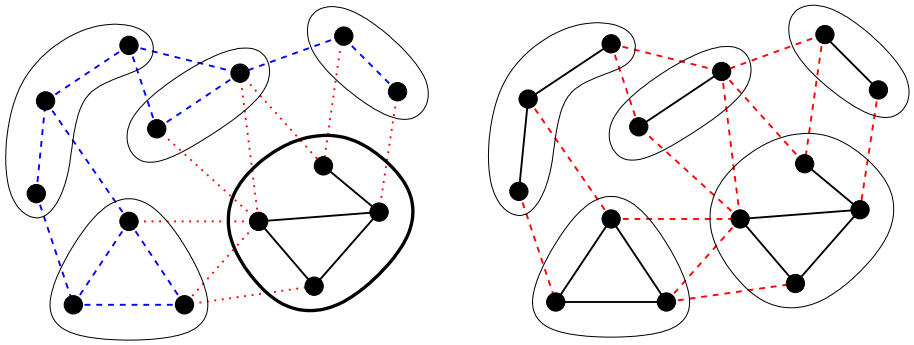
Le problème de k-partitionnement arête du graphe $G(V, E)$ correspond généralement à trouver la partition $\Pi \in P_V(G)$ telle que :

— le poids de chaque partie est identique, dans la mesure du possible ;

— le poids de l'interface entre les parties est le plus petit possible.

La première contrainte est une contrainte d'équilibrage, tandis que la seconde est une contrainte sur la taille des interfaces.

On obtient donc la définition suivante.



(a) L'ensemble des arêtes internes E_I (en traits pleins), frontières E_F (en pointillés) et externes E_E (en tirets) de la partie π (en trait épais) de G.

(b) Extension à la partition : le séparateur S apparaît en tirets, les arêtes internes en traits pleins.

FIGURE 2.6 – Les trois différents types d'arêtes induits par une partition.

Définition 19 (Problème du k-partitionnement de graphes)
Le problème de k-partitionnement d'un graphe non orienté $G = (V, E)$ à arêtes et sommets valués dans $\mathbb{R}$ où $w_V : V \to \mathbb{R}$ et $w_E : E \to \mathbb{R}$ désignent les applications qui à chaque sommet ou arête associent son poids, s'énonce maintenant de la manière suivante :

Trouver un couple $(S, (V_i)_{1\leq i\leq k})$ où S est un séparateur et $(V_i)_{1\leq i\leq k}$ une famille de sous-ensembles de V tels que :

1. *le poids de chaque partie V_i soit le même, c'est-à-dire qu'il faut minimiser la différence des poids entre chaque couple de parties :*

$$\forall (i,j) \in [\![1;k]\!]^2, |w_v(\pi_i) - w_v(\pi_j)| \text{ minimal ;}$$

2. *la coupe $\sum_{x\in S} w(x)$ soit minimale (avec $w = w_V$ dans le cas d'un séparateur sommet et $w = w_E$ dans le cas d'un séparateur arête).*

Un 2-partitionnement est appelé bipartitionnement.

2.5 Définitions sur les maillages

Commençons par définir un maillage de manière globale pour finir par une vue détaillée.

Définition 20 (Maillage)
Un maillage est une représentation discrète d'un domaine, composé de mailles, de faces, d'arêtes et de nœuds.

La figure 2.7, page suivante est un exemple de maillage.

Définition 21 (Maille ou Élément)
Une maille est un ensemble de nœuds reliés entre eux pour former un polyèdre ou un polygone, pouvant être caractérisé par sa dimension (2D ou 3D), son volume ou sa surface, sa géométrie : triangle, tétraèdre, etc...

Nous pouvons voir deux types de maille sur la figure 2.7, page suivante.

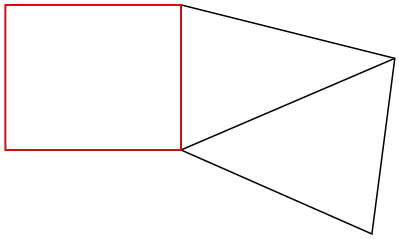


FIGURE 2.7 – Représentation d'une maille sous forme d'un rectangle (en rouge) et de deux mailles sous formes de triangles (en noir), le tout constituant un maillage.

Définition 22 (Face)
Une face est un secteur de plan qui compose la frontière d'un secteur polyèdre.

Définition 23 (Arête)
Le terme arête a déjà été défini dans le contexte des graphes. Dans les maillages, une arête fait référence à la topologie comme dans les graphes, mais également à la géométrie : une arête joint un nœud à un autre, et forme ainsi un segment.

La figure 2.8 nous donne un exemple d'arête.

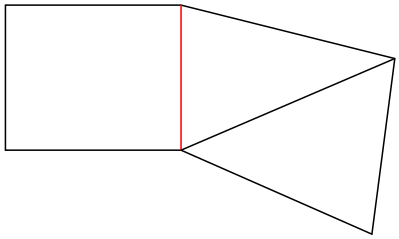


FIGURE 2.8 – Représentation en rouge d'une arête d'un maillage.

Définition 24 (Nœud)
Un nœud du maillage est un point de l'espace possédant des coordonnées cartésiennes.

La figure 2.9, page suivante illustre la notion de nœud.

Définition 25 (Face de peau)
Une face est dite de peau si elle n'appartient qu'à une seule maille.

Définition 26 (Peau du maillage)
Un nœud ou une arête fait partie de la peau du maillage s'il ou si elle appartient à une face de peau.

Une maille appartient à la peau du maillage si elle possède au moins une face de peau.

Pour parler de la peau du maillage, nous pouvons également donner le synonyme de "bord". Le reste du maillage est dit "interne".

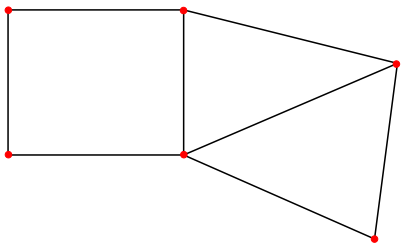
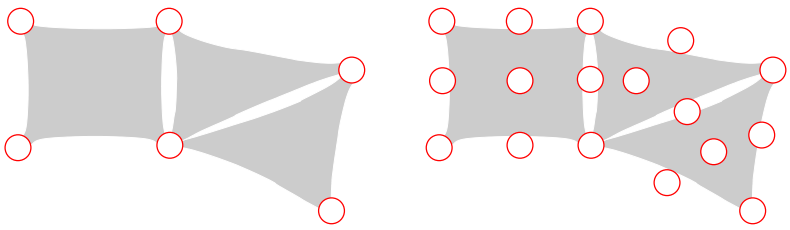


FIGURE 2.9 – Représentation en rouge des nœuds d'un maillage.

2.6 Définitions sur les graphes enrichis

Maintenant que nous avons défini ce qu'est un maillage, décrivons les besoins du numéricien utilisant des maillages, afin que notre structure de données soit le plus proche possible de ses attentes. Sa méthode numérique s'appuie sur une représentation discrète de son domaine de calcul, autrement dit un maillage. Sur celui-ci, le numéricien va placer des points sur lesquels il calcule sa solution (appelés degrés de liberté) en fonction de l'ordre du schéma et de la méthode numérique utilisée.

Définition 27 (Sommet)
Un sommet d'un graphe enrichi correspond à un ou plusieurs degrés de liberté nécessaires à la construction du modèle du numéricien. Un sommet peut être un nœud, une arête, une face, un élément, etc.



(a) Un exemple sur un maillage (en gris).

(b) Un autre exemple pour le même maillage (en gris).

FIGURE 2.10 – Les sommets (en rouge) d'un graphe enrichi.

Comme exemple de représentation discrète, il est possible de citer :
- le maillage simplicial formé de triangles ou de tétraèdres ;
- le maillage composé de quadrilatères ou d'hexaèdres ;
- le maillage mixte constitué en 2D des triangles et de quadrilatères, ou en 3D de prismes, tétraèdres et hexaèdres.

Nous pouvons ainsi avoir plusieurs types d'éléments, ce qui nous amène à distinguer des ensembles de sommets. En effet, comme sur la figure 2.10, le rectangle et les triangles ne sont pas distingués. Pour cela, nous allons utiliser la notion d'entité.

Définition 28 (Entité)
Une entité représente une catégorie de sommets ayant les mêmes caractéristiques. Un sommet

ne peut appartenir qu'à une seule entité. Il existe ainsi une surjection des sommets vers les entités.

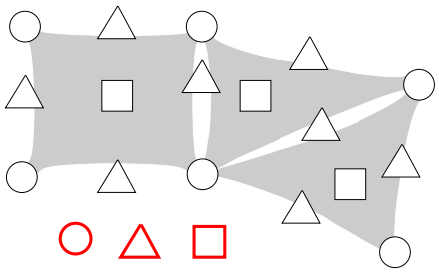


FIGURE 2.11 – Représentation en rouge des entités d'un maillage.

Dans les exemples précédents, chaque type de maille correspondra à une entité différente. Ainsi, dans le cas du maillage mixte 2D qui nous sert d'exemple, nous utilisons une pour les triangles et une pour les quadrilatères. Les entités vont nous permettre également de pouvoir distinguer les nœuds des éléments. La figure 2.11 est un exemple comprenant plusieurs entités. En effet, nous remarquons trois symboles différents (cercles, triangles et carrés) représentant chacun une entité (respectivement nœuds, arêtes et éléments).

Remarque : L'élément correspond à un sommet du graphe, comme le montre la figure 2.12. Il a la particularité d'être obligatoirement présent dans notre représentation de graphe.

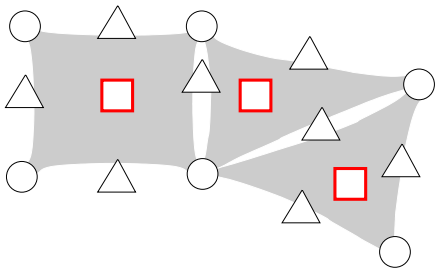


FIGURE 2.12 – Représentation en rouge des éléments d'un maillage.

Par ailleurs, il faut savoir que, suivant la méthode utilisée, le nombre d'informations à stocker par sommet est variable et que pour calculer la solution de la méthode aux degrés de liberté, les informations nécessaires proviennent de sommets voisins. Par exemple, à partir d'un élément, nous aurons besoin d'obtenir des informations portées par les nœuds composant la maille. Ceci peut être traduit par la notion de relation :

Définition 29 (Relation)
Une relation relie deux sommets s et t, si s dépend de t par rapport au modèle numérique utilisé ou bien si s est un élément et t est un sommet de la maille représentée par s.

En théorie des graphes, une relation est représentée par une arête. Comme ce terme est déjà utilisé pour les maillages et pour éviter toute ambiguité, nous avons préféré utiliser le terme de relation, illustré par la figure 2.13, page suivante.

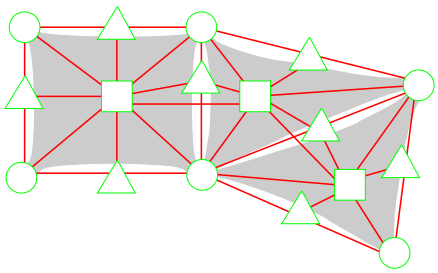


FIGURE 2.13 – Les sommets (en vert) et les relations (en rouge) forment un graphe enrichi correspondant à une des représentations du maillage (en gris).

Nous obtenons ainsi un graphe enrichi, illustré par la figure 2.13, défini par :

Définition 30 (Graphe enrichi)
Un graphe enrichi est un graphe formé de relations et de sommets classés en entités et sous-entités, dont certains doivent être des éléments.

Un maillage peut ainsi être vu comme un hypergraphe du point de vue topologique en nous restreignant aux entité nœuds et éléments : les nœuds étant l'ensemble de sommets et les éléments l'ensemble des hyper-arêtes de l'hypergraphe. Par simplification, nous assimilerons les graphes avec les entités sur les sommets par des graphes enrichis.

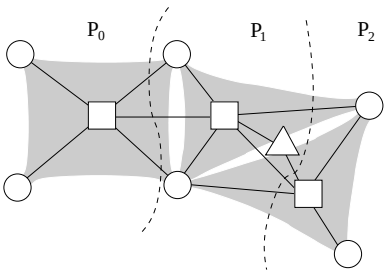


FIGURE 2.14 – Représentation du graphe enrichi correspondant au maillage distribué sur trois processeurs.

Étant donné que nous allons manipuler des maillages distribués, comme par exemple avec le maillage de la figure 2.14, nous avons besoin de caractériser un sommet par rapport au processus qui le détiendra.

Définition 31 (Sommet local)
Un sommet est dit local à, ou sur, un processeur p s'il appartient au processeur p, sachant qu'un sommet ne peut être local qu'à un processeur.

La figure 2.15, page suivante nous montre les sommets locaux sur un processeur.
Nous allons nous intéresser à l'adjacence sortante et entrante d'un sommet.

Définition 32 (Adjacence sortante d'un sommet)
L'adjacence sortante d'un sommet v du graphe G est la liste des voisins de v dans G.

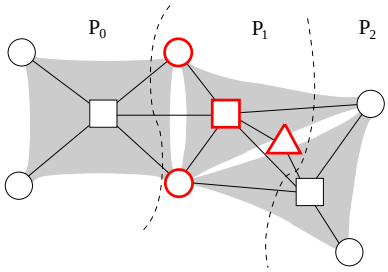


FIGURE 2.15 – Représentation en rouge des sommets locaux sur le processeur P_1.

Définition 33 (Adjacence entrante d'un sommet)
L'adjacence entrante d'un sommet v du graphe G est la liste des sommets u tel que u a pour voisin v dans G.

Remarque : L'adjacence sortante d'un sommet v local sur p est égale à son adjacence entrante si et seulement si les voisins de v sont locaux sur p.

Avant de continuer plus loin dans les définitions sur les types de sommets, nous avons besoin de revenir sur les besoins du numéricien. Suivant les méthodes numériques utilisés, le numéricien a, suivant les cas, besoin d'avoir l'adjacence sortante de tous les éléments qui sont sur un processeur. C'est pourquoi, nous allons commencer par introduire la notion de sommet fantôme englobant à la fois les sommets halo et les sommets du recouvrement.

Définition 34 (Sommet fantôme)
Un sommet est fantôme sur un processeur p s'il est local sur un processeur q, avec p voisin de q.

Les figures 2.16 et 2.17, page ci-contre illustrent la notion de sommet fantôme. Comme nous l'avons dit auparavant, suivant la méthode numérique, il n'est pas nécessaire d'avoir pour un sommet fantôme l'adjacence sortante, d'où la notion de sommet halo.

Définition 35 (Sommet halo)
Un sommet est halo sur un processeur p s'il est fantôme et si son adjacence sortante n'est pas dupliquée sur p.

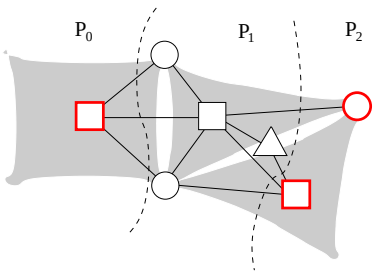


FIGURE 2.16 – Représentation en rouge des sommets halo sur le processeur P_1.

Nous pouvons voir sur la figure 2.16 les sommets halo ainsi que l'absence d'adjacence sortante pour ces sommets.

Définition 36 (Recouvrement)
Un recouvrement est une duplication de sommets ainsi que de leur adjacence sortante.

Définition 37 (Sommet du recouvrement)
Un sommet fait partie du recouvrement sur un processeur p s'il est fantôme et si son adjacence sortante est dupliquée sur p.

La figure 2.17 permet de distinguer les sommets du recouvrement des sommets locaux.

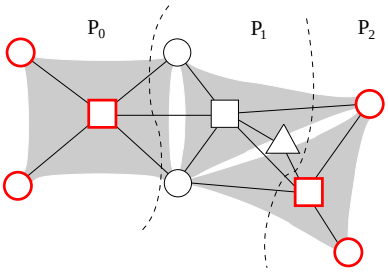


FIGURE 2.17 – Représentation en rouge des sommets du recouvrement sur le processeur P_1.

Définition 38 (Sommet interne)
Un sommet est interne s'il n'a aucune relation sortante vers un sommet fantôme.

Définition 39 (Sommet de bord)
Un sommet est de bord est un sommet qui est relié à au moins un sommet fantôme.

Nous pouvons en déduire qu'un sommet peut soit être interne soit de bord, mais ne peut pas être les deux à la fois, comme le montre la figure 2.18.

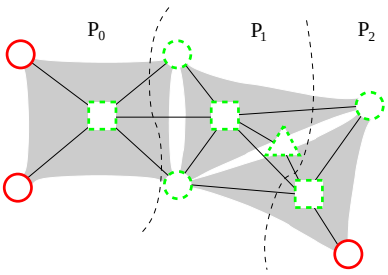


FIGURE 2.18 – Représentation des sommets internes (en rouge et traits pleins) et des sommets de bord (en vert et tirets) du graphe enrichi distribué.

Définition 40 (Sommet frontière)
Un sommet frontière est soit un sommet local p relié à un sommet fantôme q, soit un sommet fantôme p' relié à un sommet de bord q' avec p et p' n'appartenant pas au même processeur que q et q'.

La figure 2.19, page suivante illustre la notion de sommet frontière.

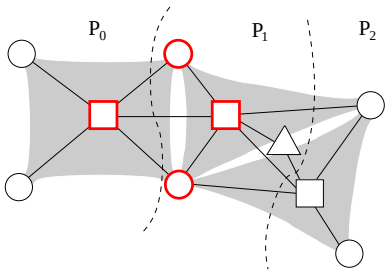


FIGURE 2.19 – Représentation des sommets frontières (en rouge) entre les processeurs P_0 et P_1.

Chapitre 3

Structures de données et algorithmique pour la gestion de maillages distribués

Sommaire

3.1 Modélisation de la mécanique des fluides

Pour illustrer les besoins du numéricien dans le cadre de la modélisation de la mécanique des fluides, nous allons nous intéresser à l'exemple de la résolution numérique de l'équation de Poisson.

3.1.1 Équation de Poisson

La résolution numérique de l'équation de Poisson est un problème stationnaire qui intervient fortement en physique (mécanique des fluides, électrostatique, *etc.*). L'équation de Poisson est :

$$-\Delta u(x,y) = f(x,y) \text{ sur un domaine } \Omega \subset \mathbb{R}^2, \text{ avec } f : \mathbb{R}^2 \to \mathbb{R} \text{ et } u : \mathbb{R}^2 \to \mathbb{R} \ . \tag{3.1}$$

L'opérateur Laplacien Δ est défini en 2D par :

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

Considérons alors, par exemple, une condition aux limites de type Dirichlet non homogène :

$$u(x,y) = g(x,y) \text{ sur la frontière du domaine } \Gamma = \partial\Omega \ ;$$

non homogène signifie que g peut être différent de 0. Utilisons la méthode des volumes finis, qui consiste à découper l'espace en des « volumes de contrôle » notés K_i, et à résoudre notre problème de manière discrète :

$$\Omega = \bigcup K_i \ . \tag{3.2}$$

Intégrons l'équation 3.1 sur K_i :

$$-\int_{K_i} \Delta u \ d\Omega \quad = \quad \int_{K_i} f \ d\Omega \ , \tag{3.3}$$

puis utilisons le théorème de Stokes :

$$-\int_{\partial K_i} \nabla u(u) \cdot \vec{n} \, d\delta \quad = \quad \int_{K_i} f \ d\Omega \ . \tag{3.4}$$

Si l'on considère que les K_i sont des polygônes délimités par leurs arêtes e, nous pouvons écrire :

$$-\int_{\partial K_i} \nabla u \cdot \vec{n} \, d\delta \quad = \quad -\sum_{e \in K_i} \int_e \nabla u \cdot \vec{n} \, d\delta \ . \tag{3.5}$$

L'approximation du flux numérique $\int_e \nabla u \cdot \vec{n}$ détermine le type de schéma aux volumes finis. Sur un maillage de contrôle formé de triangles, nous pouvons choisir d'approcher, le flux au moyen d'un schéma de type « *vertex centered* », c'est-à-dire homogène sur chaque arête e :

$$\int_e \nabla u \cdot \vec{n} \, d\delta = |e| \frac{u(x_{K_i}) - u(x_{K_j})}{d_{K_i K_j}} \ , \tag{3.6}$$

où les deux points x_{K_i} et x_{K_j} sont définis comme illustré en figure 3.1, page ci-contre.

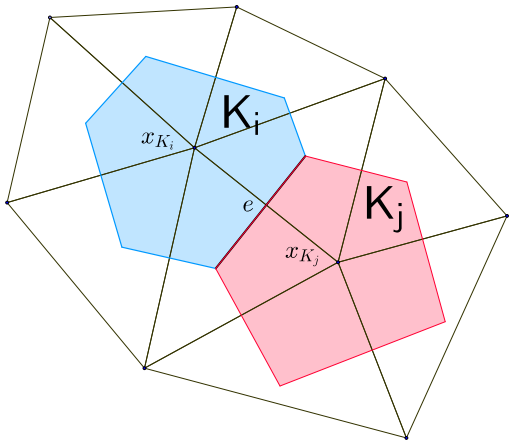


FIGURE 3.1 – Représentation des deux volumes de contrôle K_i et K_j, des deux points voisins x_{K_i} et x_{K_j}, et de l'arête e commune à K_i et K_j.

En approchant $\int_{K_i} f(x)\,dx$ par $|K_i| f(x_{K_i})$, nous obtenons alors pour chaque K_i :

$$\sum_{e \in K_i} |e| \frac{u(x_{K_i}) - u(x_{K_j})}{d_{K_i K_j}} \quad = \quad |K_i| f(x_{K_i}) \ , \tag{3.7}$$

En réécrivant les équations sous forme matricielle, nous obtenons :

$$\mathcal{A}\,\mathcal{U} = \mathcal{B} \text{ avec } \mathcal{U} = \begin{pmatrix} \cdots \\ u(x_{K_i}) \\ \cdots \end{pmatrix} \text{ et } \mathcal{B} = \begin{pmatrix} \cdots \\ |K_i| f(x_{K_i}) \\ \cdots \end{pmatrix} \ . \tag{3.8}$$

Nous pouvons alors résoudre le système linéaire (3.8) au moyen de la méthode de Jacobi. Pour cela, nous découpons la matrice $\mathcal{A}_k$ en $\mathcal{D} - \mathcal{E} - \mathcal{F}$, avec :

— $\mathcal{D}$ la matrice diagonale ;
— $\mathcal{E}$ la matrice triangulaire supérieure ;
— $\mathcal{F}$ la matrice triangulaire inférieure,

et nous résolvons de manière itérative le système suivant :

$$\mathcal{D}\mathcal{U}^{n+1} - (\mathcal{E} + \mathcal{F})\mathcal{U}^n = \mathcal{B} \ , \tag{3.9}$$

où $\mathcal{U}^n = \begin{pmatrix} u(x_{K_1})^{(n)} \\ \vdots \\ u(x_{K_N})^{(n)} \end{pmatrix}$ représente la $n^{ième}$ itération de Jacobi. La $n+1^{ième}$ itération s'écrit et se résoud ainsi :

$$\mathcal{U}^{n+1} = \mathcal{D}^{-1}(\mathcal{B} + (\mathcal{E} + \mathcal{F})\mathcal{U}) \ . \tag{3.10}$$

3.1.2 Formalisation

L'exemple de l'équation de Poisson présenté ci-dessus nous permet de recenser les besoins suivants nécessaires à la modélisation de la mécanique des fluides.

La solution étant portée par les x_{K_i}, qui correspondent aux nœuds du maillage, il est nécessaire de pouvoir les définir en tant que sommets, ainsi que de pouvoir associer des valeurs à ces sommets.

Une méthode pour calculer les contributions de chaque triangle au volume de contrôle K_i est de parcourir toutes les arêtes du maillage et, pour chacune d'entre elles, d'obtenir les coordonnées des nœuds qui la composent, dans le but de calculer la normale $\vec{n}$ et la distance $d_{K_iK_j}$. D'autre part, le calcul de $|e|$ nécessite d'explorer les éléments partageant chaque arête. Par conséquent, les éléments doivent également être considérés comme des sommets, ce qui nécessite de différencier des groupes de sommets en tant qu'entités distinctes. Dans notre exemple, nous choisissons comme entités les nœuds, les arêtes et les éléments.

Par ailleurs, sachant que e a pour extrémités le centre des cercles circonscrits aux triangles, les coordonnées de ces points peuvent être associées aux éléments. Dès à présent, nous voyons que la nature des valeurs qui sont associées est différente suivant l'entité à laquelle appartient le sommet. Sur les nœuds, nous stockons comme valeurs les coordonnées ainsi que la solution $\mathcal{U}$, alors que les éléments doivent porter les coordonnées du centre du cercle circonscrit. L'association des valeurs doit ainsi se faire par type d'entités.

Revenons sur le fait qu'à partir d'une arête, nous devons obtenir les nœuds qui la composent et les éléments qui la partagent. Cela implique en premier lieu d'associer entre eux certains sommets pour disposer de cette information ; nous utiliserons pour cela des relations entre les sommets. D'autre part, le besoin de parcourir tous les nœuds qui composent une arête fait apparaitre la nécessité de disposer d'un itérateur capable d'énumérer ces nœuds. Un itérateur global est également nécessaire afin de parcourir l'ensemble des arêtes du maillage.

En résumé, les entités nécessaires à la résolution de l'équation de Poisson sont : les éléments, les arêtes et les nœuds. Les relations nécessaires entre ces entités sont les couples arêtes - nœuds, arêtes - éléments et éléments - nœuds. Enfin, les valeurs associées aux différentes entités sont constituées des coordonnées pour les nœuds et les éléments, de la solution pour les nœuds et de l'aire des triangles pour les éléments. L'écriture du schéma nécessite également de pouvoir itérer globalement sur toutes les instances d'une entité du maillage, ainsi que d'itérer localement sur toutes les instances d'un certain type d'entité liées à une instance donnée.

3.1.3 Surcoût induit par le parallélisme

Afin de résoudre numériquement de tels problèmes, nous devons également prendre en compte la problématique induite par la taille croissante des maillages, tels que des maillages 3D possédant des géométries complexes. La modélisation de l'équation de Poisson sur une seule unité de traitement ne peut être réalisée sur des maillages de grandes tailles lorsque la taille de ces maillages excède la mémoire disponible.

Il faut par conséquent distribuer les données des maillages sur plusieurs unités de traitement, afin de pouvoir exécuter la simulation en parallèle. Chaque unité de traitement a la charge de calculer les incounnues du système qui lui ont été attribuées. Elle doit pour cela disposer des valeurs calculées à l'itération précédente par ses voisins aux bords de leurs domaines de calcul respectifs.

Si nous reformulons le système linéaire de l'équation (3.8) de la page 33 au niveau de chaque

unité de traitement, nous obtenons la représentation de $\mathcal{A}$,$\mathcal{U}$ et $\mathcal{B}$ suivante :

$$\left(\begin{array}{c|c} \cdots & \cdots \\ \cdots & \cdots \\ \textbf{Locale} & \textbf{Recouvrement} \\ \cdots & \cdots \\ \cdots & \cdots \\ \hline \cdots & \cdots \\ \textbf{Recouvrement} & \textit{Invalide} \\ \cdots & \cdots \end{array}\right)\left(\begin{array}{c} \cdots \\ \cdots \\ \textbf{Locale} \\ \cdots \\ \cdots \\ \hline \cdots \\ \textbf{Recouvrement} \\ \cdots \end{array}\right) = \left(\begin{array}{c} \cdots \\ \cdots \\ \textbf{Locale} \\ \cdots \\ \cdots \\ \hline \cdots \\ \textit{Non calculé} \\ \cdots \end{array}\right) \quad .(3.11)$$

Une des contraintes associées à la distribution des données est l'équilibrage de la charge des différents processeurs. Puisque les données sont distribuées, nous voulons donc minimiser, d'une part, la duplication de données sur plusieurs processeurs et, d'autre part, les communications inter-processeurs.

L'algorithme 1 décrit une itération de Jacobi en parallèle. Nous reviendrons plus en détails sur cet algorithme pour justifier nos choix lors de la manipulation de maillages, mais aussi sur les fonctionnalités qui doivent être mises à la disposition du numéricien.

Algorithme 1 Algorithme pour une itération de Jacobi en parallèle.

1: **Pour** chaque sommet v de bord **Faire**
2: $\quad \mathcal{U}_v^{n+1} = \mathcal{D}_v^{-1}(\mathcal{B}_v + \sum_{w=1}^{N}((\mathcal{E}_{v,w} + \mathcal{F}_{v,w})\,\mathcal{U}_w^n))$
3: **Fin de Pour**
4: Commencer la communication sur les sommets du recouvrement
5: **Pour** chaque sommet v interne **Faire**
6: $\quad \mathcal{U}_v^{n+1} = \mathcal{D}_v^{-1}(\mathcal{B}_v + \sum_{w=1}^{N}((\mathcal{E}_{v,w} + \mathcal{F}_{v,w})\,\mathcal{U}_w^n))$
7: **Fin de Pour**
8: Attendre la fin de la communication sur les sommets du recouvrement

3.2 Structure de données

Une étape indispensable de la conception des algorithmes parallèles est la définition des structures de données qui seront manipulées.

Nous allons présenter ici la façon dont sont distribuées, sur un ensemble de p processeurs, les données des maillages distribués, et l'impact de cette distribution sur les algorithmes de partitionnement, repartitionnement et adaptation parallèle de maillages distribués que nous allons mettre en œuvre.

En effet, de la nature et de la distribution des données dépendent grandement les performances des algorithmes que nous voulons développer.

3.2.1 Entités et sous-entités

Revenons tout d'abord sur les entités, dont le formalisme a déjà été utilisé [66]. Nous utilisons cette notion pour distinguer les nœuds, les arêtes, les faces, les éléments, etc. Dans l'exemple de l'équation de Poisson, nous avons mis en évidence trois entités : une pour les nœuds, une pour

les arêtes et une pour les éléments. En effet, comme nous l'avons vu, les nœuds définissent la géométrie et portent la solution, les arêtes du maillage définissent le flux des arêtes du volume de contrôle vers les nœuds et les éléments définissent l'aire ainsi que la topologie, via les relations entre les éléments et tous les sommets qui composent l'élément, à savoir les nœuds et les arêtes.

Les conditions aux limites de Dirichlet sont portées par les arêtes bornant le domaine de calcul. Nous pouvons voir ces arêtes « de bord » comme un sous-ensemble des arêtes. Nous appellerons « arêtes internes » les arêtes qui ne sont pas des arêtes de bord.

Introduisons la notion de sous-entité :

Définition 41 (Sous-entité)
Une sous-entité est un sous-ensemble d'une entité. Un sommet ne peut appartenir au plus qu'à une seule sous-entité. Une entité peut posséder zéro ou plusieurs sous-entités. Dans ce dernier cas, l'entité est appelée « entité mère ».

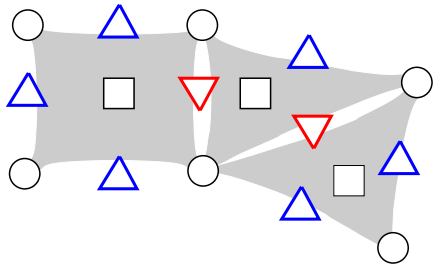


FIGURE 3.2 – Représentation en bleu et en rouge de deux sous-entités représentant respectivement les arêtes de bord et les arêtes internes d'un maillage.

Ainsi, l'entité arête peut posséder deux sous-entités, qui sont les arêtes internes et les arêtes de bord, comme l'illustre la figure 3.2.

3.2.2 Numérotation des sommets

Afin de pouvoir définir de façon univoque un maillage, nous devons pouvoir identifier de façon globale chaque constituant du maillage. Nous considérons pour cela une numérotation globale de chaque sommet, indépendamment du processeur et de l'entité auquel il appartient. Cette numérotation n'est pas nécessaire à la résolution des méthodes numériques, mais à la construction du graphe distribué représentant le maillage et des structures de communication (voir section 3.4.1, page 43). La figure 3.3, page ci-contre nous montre un exemple d'une telle numérotation. Le principe de distribution que nous avons choisi est le suivant : chaque processeur p_i possède un sous-ensemble $V_{|p_i}$ de sommets et un sous-ensemble $E_{|p_i}$ des relations incidentes aux sommets de $V_{|p_i}$. Les sommets $V_{|p_i}$ sont les sommets locaux au processeur p_i, et les sommets $\bigcup_{p_j \neq p_i} V_{|p_j}$ sont les sommets distants par rapport au processeur p_i. La famille $\{V_{|p_i}\}_{p_i \in [\![1;p]\!]}$ est une partition de l'ensemble des sommets V. Les relations reliant des sommets situés sur deux processeurs différents sont donc dupliquées sur ces deux processeurs. Afin de déterminer rapidement à quel processeur appartient un sommet, nous numérotons les sommets par ordre croissant de numéros de processeurs, comme le montre la figure 3.4, page suivante. Autrement dit, un processeur p_i contient toujours des sommets dont les numéro sont inférieurs à ceux du processeur p_{i+1}. Ainsi, à l'aide d'un tableau d'une taille égale au nombre de processeurs, dont

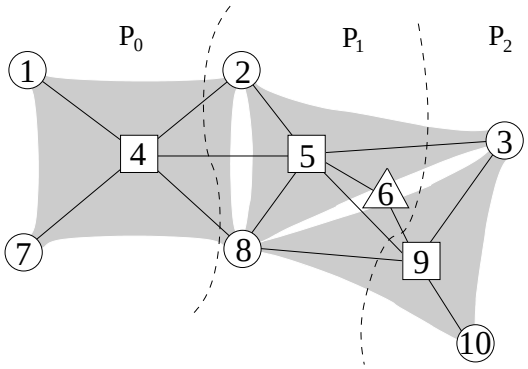


FIGURE 3.3 – Exemple de numérotation globale pour un maillage distribué sur 3 processeurs.

chaque case contient le plus petit numéro de sommet possédé par le processeur, nous pouvons déterminer par dichotomie sur quel processeur est stocké un sommet de numéro donné.

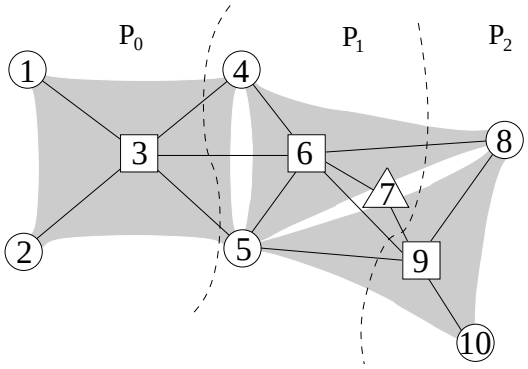


FIGURE 3.4 – Exemple d'un maillage distribué sur trois processeurs avec la numérotation globale choisie : les numéros des sommets sont triés par processeur.

Dans le paradigme du parallélisme à mémoire distribuée, chaque processeur effectue ses calculs sur les données contenues dans sa mémoire locale. Celle-ci, de taille restreinte, ne peut contenir de tableaux de données indexés globalement, d'autant que seules les cases correspondant aux données locales et voisines lui sont nécessaires.

Par conséquent, nous avons besoin d'une numérotation locale et compacte pour indexer les valeurs associées dans ces mémoires locales, ce qui permet de manipuler les données sur les sommets locaux. Afin de pouvoir effectuer tous les calculs sur le graphe enrichi au niveau local, on utilise la paire $(V_{|p_i}, E_{|p_i})$ comme un sous-graphe de G. Pour éviter des communications avec les sommets distants, ceux-ci sont dupliqués localement en tant que sommets « fantômes ». Par ailleurs, les données sont propres à chaque entité ; c'est pourquoi la numérotation locale doit être propre à chaque entité. Le numéro local d'un sommet qui permet d'accéder à ses valeurs, doit être compris entre un et le nombre de sommets locaux dans cette entité [5], comme le montre la figure 3.5, page suivante. Le processeur P_0 possède deux nœuds, dont les numéros sont compris entre 1 et 2, et un seul élément, à qui est attribué le numéro 1.

5. Ou entre zéro et le nombre de sommets locaux dans cette entité moins un, dans le cas où la numérotation des entités commence à partir de la valeur zéro.

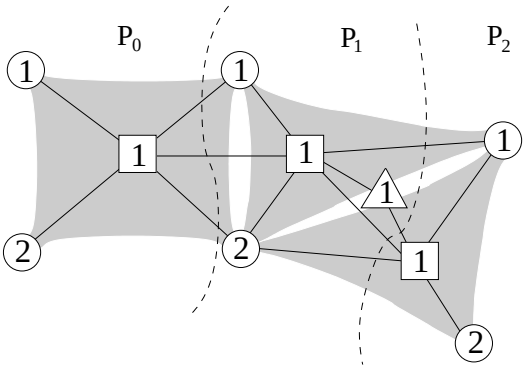


FIGURE 3.5 – Exemple d'un maillage distribué sur trois processeurs, illustrant la numérotation locale et par entité sur chaque processeur.

Cependant, cette numérotation est limitée aux sommets locaux, alors qu'il est nécessaire de lire les données des sommets distants, représentés localement sous forme de sommets fantômes. Afin de maintenir la compatibilité avec la numérotation locale, ces sommets sont numérotés sur chaque processeur à la suite des sommets locaux, en s'appuyant sur l'ordre défini par la numérotation globale. Autrement dit, les sommets fantômes d'un processeur P_i sont numérotés avant les sommets fantômes d'un processeur P_j si $i < j$. Sur la figure 3.6, nous voyons que l'élément fantôme du processeur P_0 est numéroté avant celui du processeur P_2, ceci étant une conséquence de la numérotation globale du graphe enrichi, définie en figure 3.4, page précédente.

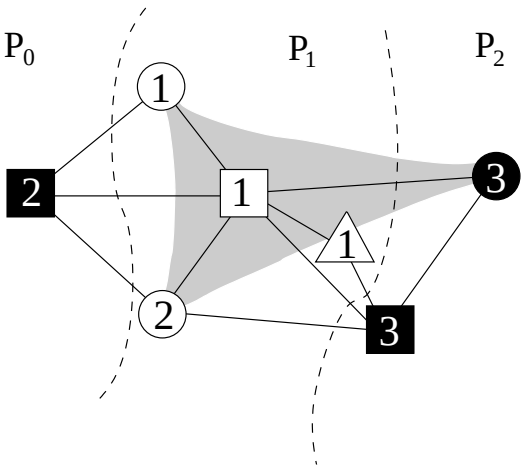


FIGURE 3.6 – Exemple d'un maillage distribué sur trois processeurs, illustrant la numérotation locale restreinte au processeur 1. Les sommets en noir représentent les sommets fantômes vis-à-vis de ce processeur.

Intéressons-nous maintenant à l'ordre des voisins pour chaque sommet. Cet ordre n'est important qu'au sein d'une même entité. Très souvent, les formats de fichiers de maillages s'appuient sur l'ordre des nœuds décrivant un élément afin de déterminer l'orientation de l'élément mais aussi des faces qui le composent. Si nous reprenons l'algorithme 1, page 35, se rapportant à une itération de Jacobi, chacune des deux boucles remplit le vecteur u. Chaque valeur du vecteur solution est mise à jour en parcourant, pour chaque élément v, ses éléments voisins. Or, plusieurs

itérations de Jacobi sont nécessaires afin de résoudre le système linéaire. Par conséquent, il est essentiel que les voisins d'un sommet dans une entité soient parcourus en temps constant. Nous ordonnons donc les voisins des sommets par entité en gardant leur ordre au sein de cette entité. Ceci permet de préserver l'information contenue dans le format de fichier, sur laquelle se basent les numériciens.

Si nous nous intéressons maintenant à la numérotation du point de vue des sous-entités, les sommets voisins vont être rangés par sous-entité. Par exemple, décomposons les nœuds en deux sous-entités : les nœuds internes et les nœuds de bord. Un élément qui possède des nœuds appartenant à ces deux sous-entités va stocker les nœuds voisins d'une sous-entité, puis de l'autre. Ainsi, les nœuds de l'élément peuvent être permutés, alors que, comme nous l'avons vu, l'orientation des éléments s'appuie sur l'ordre des nœuds.

Nous voulons donc que l'ordre des sommets puisse être garanti au sein d'une entité. Nous devons pour cela introduire la notion d'entité stable, qui s'opposera à la notion d'entité instable.

Définition 42 (Entité instable)
Une entité mère est dite instable quand l'ordre des voisins d'un sommet est déterminé au sein de chaque sous-entité et donc n'est potentiellement pas préservé au sein de l'entité mère.

Définition 43 (Entité stable)
Une entité mère est dite stable quand l'ordre des voisins d'un sommet est déterminé au sein de l'entité mère.

Remarquons que l'utilisation d'une entité stable augmente le temps d'accès aux voisins de chacune de ses sous-entités, qui devient linéaire vis-à-vis du nombre de voisins au niveau de l'entité mère. Nous pouvons l'expliquer de la manière suivante. Le voisin d'un sommet n'est stocké qu'une fois dans la liste d'adjacence du sommet, même si le voisin appartient à une entité stable. Le numéro stocké se réfère à l'entité et non aux sous-entités, afin d'éviter toute ambigüité. Par conséquent, si, pour un sommet, nous voulons le prochain voisin par rapport à une sous-entité, nous devons avancer dans la liste des voisins dans l'entité mère jusqu'à rencontrer le premier sommet appartenant à la sous-entité choisie.

Ainsi, dans l'exemple qui nous intéresse, nous pouvons utiliser une entité stable pour l'entité nœud, afin de garantir l'ordre des nœuds voisins d'un élément. Même si l'entité nœud possède plusieurs sous-entités, les nœuds qui composent l'élément seront ordonnés au sein de l'entité nœud et non par sous-entité. En permettant de classer les sommets dans l'entité nœud, l'entité stable conserve l'ordre des sommets au sein de cette entité, respectant en cela les pratiques habituelles des numériciens.

3.3 Manipulation des maillages distribués

Nous allons détailler dans ce chapitre les outils utilisés pour la simulation numérique, tels que le partitionnement et la redistribution de maillages.

3.3.1 Partitionnement de graphes et de maillages

Notre but est de résoudre le problème de partitionnement présenté dans la définition 19, page 23. Ce problème est un problème de recherche d'un optimum parmi l'ensemble des partitions

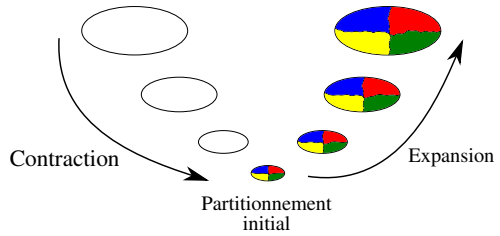


FIGURE 3.7 – Représentation schématique d'un 4-partitionnement à l'aide du schéma multi-niveaux.

du graphe G. Ce problème est NP-Difficile [55, 88]. Trouver la meilleure solution dans le cas d'un graphe G quelconque s'effectue donc selon une complexité proportionnelle au nombre de partitions de G, c'est-à-dire en $\Omega(p^n)$. De plus, on remarquera que vérifier que l'on dispose du meilleur partitionnement est généralement tout aussi difficile, c'est-à-dire NP-Complet. Par la suite, nous serons donc obligés de nous contenter de résultats non-optimaux, produits par des heuristiques.

La technique multi-niveaux, mise en œuvre par de nombreux logiciels de partitionnement, a été introduite par Barnard et Simon [5, 60, 105]. Son principe consiste à travailler sur un graphe réduit ayant les mêmes propriétés topologiques que le graphe initial, afin de réduire le coût de calcul des partitions et d'améliorer l'efficacité des algorithmes d'optimisation locale.

Le schéma multi-niveaux comporte trois étapes :

1. l'étape de contraction, durant laquelle une fonction de contraction est appliquée récursivement afin de construire une famille de graphes de tailles de plus en plus réduits mais topologiquement équivalents au graphe initial ;
2. l'étape de partitionnement initial, au cours de laquelle un partitionnement est calculé sur le plus petit graphe contracté ainsi obtenu. Parce qu'ils opèrent sur un graphe de taille réduite, les algorithmes d'optimisation globale sont bien moins coûteux que s'ils opéraient sur le graphe original ;
3. l'étape d'expansion, durant laquelle la partition initiale est prolongée et raffinée successivement sur les graphes de plus en plus fins jusqu'à obtenir une partition du graphe initial. Les algorithmes de raffinement employés dans cette phase sont des algorithmes d'optimisation locale, qui n'opèrent que pour lisser localement la frontière prolongée.

Cette procédure est illustrée par la figure 3.7.

Notre contribution en ce domaine a été d'adapter l'utilisation des outils de partitionnement de graphe existants au partitionnement des graphes enrichis que nous manipulons. Nous avons élaboré un premier partitionnement, basé sur le graphe des éléments, mais nous avons remarqué des déséquilibres de charge. C'est pourquoi, nous avons décidé d'élaborer une deuxième méthode : le partitionnement du graphe des entités du maillage.

3.3.2 Partitionnement du graphe des éléments

La première méthode que nous avons appliquée conséquente dans une première étape le sous-graphe induit du graphe enrichi restreint aux éléments. Une fois le partitionnement des éléments calculé, nous étendons le partitionnement aux autres sommets en nous basant sur l'algorithme de Luby [78], explicité par l'algorithme 2, page ci-contre. La méthode consiste à

distribuer aléatoirement un numéro de partie aux sommets n'appartenant pas encore à une partie, en fonction du numéro de partie des éléments voisins.

Algorithme 2 Algorithme de Luby pour partitionner les sommets autres que les éléments.

```
1: Pour chaque sommet u local Faire
2:     Si u n'est pas un élément alors
3:         v ← le premier élément voisin de u
4:         partition[u] ←partition[v]
5:     Fin de Si
6: Fin de Pour
```

L'avantage de cette méthode est de ne pas avoir à fournir l'ensemble du graphe enrichi au partitionneur, afin de ne pas augmenter de manière considérable l'espace mémoire. Par ailleurs, le temps de calcul est moindre, sachant que l'algorithme de Luby a une complexité en temps en $\Theta(n_v)$. Néanmoins, comme nous le verrons dans la partie dédiée à la validation expérimentale (section 3.5, page 52), cette méthode peut entrainer des déséquilibres de charge, parce qu'un processeur peut être privilégié si ses éléments sont toujours placés en premier dans les listes d'adjacence des sommets des autres entités.

3.3.3 Partitionnement du graphe des entités du maillage

Afin de remédier aux problèmes exposés ci-dessus, nous avons conçu une seconde méthode, qui fournit au partitionneur l'ensemble du graphe enrichi. Le partitionnement du graphe enrichi est alors plus coûteux en temps mais donne de meilleurs résultats.

Par ailleurs, un autre atout de cette méthode est la meilleure prise en considération des poids des sommets qui ne sont pas des éléments. En effet, les poids de ces sommets, qui sont maintenant communiqués au partitionneur, sont alors utilisés lors du partitionnement.

Cependant, cette méthode requiert un post-traitement pour éviter que des sommets d'une maille ne soient isolés par rapport au sommet représentant l'élément. Certaines méthodes, comme celles de Galerkin discontinues, ne tolèrent pas qu'un sommet d'une maille soit localisé sur un sous-domaine différent de ses éléments voisins.

3.3.4 Association des données

Comme nous l'avons vu dans la sous-section 3.1.2, page 34, relative aux fonctionnalités nécessaires à la résolution des problèmes numériques, la nature des données diffère d'une entité à l'autre. Par exemple, les coordonnées des nœuds sont portées par l'entité nœud, et celles des centres des cercles circonscrits par l'entité élément. Une autre donnée portée cette fois-ci par la sous-entité nœud de bord est celle des conditions aux limites. Ainsi, les données doivent pouvoir être associées aussi bien sur les entités que sur les sous-entités.

Nous distinguons trois besoins relatifs aux données associées. Leur point commun consiste à pouvoir redistribuer ces données lorsque la topologie du maillage change. Leur différence réside dans leur utilisation. Le premier cas, qui semble naturel, est d'associer une valeur aux sommets locaux et de pouvoir accéder à celle portée par les sommets distants. Ceci induit que la donnée est allouée depuis les sommets locaux et fantômes, mais aussi qu'il faille la diffuser depuis les sommets locaux, pour qu'elle soit dupliquée sur les sommets fantômes des autres processeurs. Ces données seront dites « publiques ».

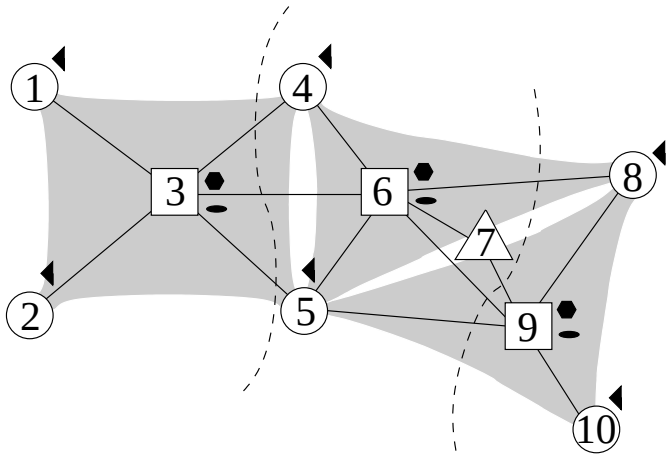


FIGURE 3.8 – Données associées sur différentes entités du maillage. Les nœuds stockent un type de donnée spécifique (cette donnée étant représentée par un triangle), alors que les éléments en possèdent deux (représentées par un hexagone et une ellipse).

En revanche, le numéricien peut aussi souhaiter stocker des données uniquement sur les sommets locaux, comme par exemple le second membre de l'équation 3.11, page 35 dans la résolution par la méthode de Jacobi. Ces données seront alors dites privées. Cela sous-entend que ces données ne seront jamais échangées entre les processeurs.

Un dernier cas consiste à affecter une valeur sur les sommets locaux et fantômes, pour y stocker des valeurs intermédiaires, tout en interdisant leur échange entre processeurs. Une telle donnée, dite « protégée », ne peut être valide sur les sommets fantômes après redistribution. Si une telle donnée est échangée entre processeurs, les valeurs anciennement portées par les sommets fantômes seront écrasées par celles portées par les processeurs où ces sommets sont locaux.

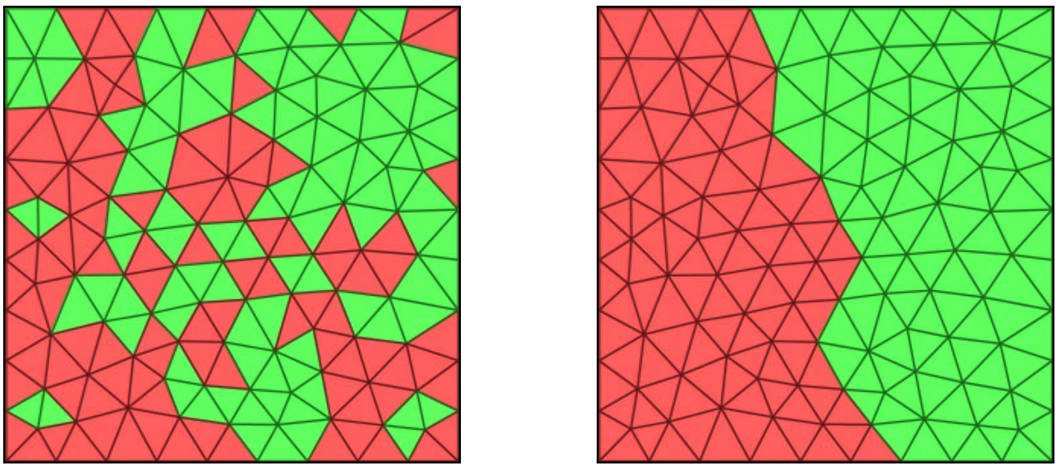
3.3.5 Redistribution

La redistribution d'un maillage s'effectue sur un maillage préalablement distribué. Une première distribution naïve d'un maillage peut être réalisée selon les numéros globaux des sommets, toutes entités confondues, en attribuant un intervalle à chaque processeur. Or, la volonté du numéricien est d'avoir un maillage distribué dont la charge soit équilibrée sur les processeurs. Cet équilibrage de la charge provient d'une partition donnée par le partitionneur. La redistribution se base sur cette partition pour déterminer un équilibrage de charge.

Une renumérotation (voir chapitre 4, page 57) peut également être appliquée lors de la redistribution, pour réorganiser les différentes entités du maillage afin de minimiser les défauts de cache, et ainsi d'augmenter la vitesse d'exécution du programme.

Les figures 3.9a, page suivante et 3.9b, page ci-contre représentent des maillages distribuées sur deux processeurs [6]. La première correspond au maillage distribué selon la numérotation globale initiale issue du fichier d'entrée, et la seconde illustre le maillage après redistribution basée sur un partitionnement.

6. En noir et blanc, les couleurs verte et rouge donnent des gris clair et foncé, respectivement.



(a) Après lecture et avant redistribution. (b) Après partitionnement et redistribution.

FIGURE 3.9 – Distribution des éléments d'un maillage sur deux processeurs. Le premier processeur est représenté par la couleur verte, alors que le deuxième est représenté par la couleur rouge.

3.4 Fonctionnalités pour la simulation numérique

3.4.1 Communications inter-processus

Dans le cas de données publiques, les sommets fantômes d'un processeur P_i portent une valeur qui provient de processeurs voisins possédant ces sommets en tant que sommets locaux. La donnée à échanger peut être commune à tous les sommets d'une sous-entité, d'une entité ou bien encore de toutes les entités.

La structure de communication que nous avons mise en œuvre a été reprise de PT-SCOTCH. Elle a été adaptée afin de pouvoir échanger des données aussi bien sur tous les sommets qu'entre sommets n'appartenant qu'à une entité ou à une sous-entité particulière.

L'algoritme 1, page 35 met en évidence des communications entre les processus pour mettre à jour les valeurs sur les éléments du recouvrement, sachant que le maillage est distribué en mémoire sur les différents processeurs. Nous devons remplir des tampons de communication afin de minimiser les accès mémoire. Ceci revient à lire séquentiellement et à extraire des valeurs du tableau correspondant à la donnée à transmettre.

Une valeur portée par un sommet local peut être envoyée à un ou plusieurs processeurs. Par ailleurs, les numéros des processeurs sont supérieur ou égaux à zéro et le saut entre deux sommets porteurs d'une valeur à envoyer est strictement supérieur à zéro. Nous pouvons ainsi combiner les numéros des processeurs destinataires et l'opposé des sauts entre sommets émetteurs. Le tableau contenant les sauts et les processeurs est appelé `procsidtab`[7]. Le sommet à partir duquel nous commençons est le premier sommet local, soit zéro ou un, selon le type de numérotation souhaitée par l'utilisateur. L'algorithme 3, page suivante est utilisé pour remplir ce tableau.

La structure de communication précédente est calculée pour chaque entité et sous-entité dont il existe au moins un sommet. Elle est utilisée par les communications collectives et point-à-point, de manière synchrone ou asynchrone. Les communications collectives mobilisent simultanément

7. Pour « *processor send index table* ». Les conventions de nommage que nous utilisons sont reprises de celles de PT-SCOTCH

Algorithme 3 Construction du tableau de remplissage des tableaux de communication : `procsidtab`.

1: $u \leftarrow baseval$
2: **Pour** chaque sommet v à envoyer **Faire**
3: **Si** $u \neq v$ **alors**
4: $procsidtab[i] \leftarrow u - v$
5: $u \leftarrow v$
6: $i \leftarrow i + 1$
7: **Fin de Si**
8: **Pour** chaque processeur p ayant v comme sommet fantôme **Faire**
9: $procsidtab[i] \leftarrow p$
10: $i \leftarrow i + 1$
11: **Fin de Pour**
12: **Fin de Pour**

tous les processeurs, alors que les communications point-à-point n'impliquent que deux processeurs. Ces dernières sont utilisées quand le nombre de processeurs voisins intervenant dans des échanges de données est inférieur à un pourcentage du nombre total de processeurs, seuil à partir duquel elles sont plus efficaces, comparées aux communications collectives. Les communications synchrones impliquent que l'échange de données soit terminé avant l'exécution du code qui suit la demande de communication. En revanche, les communications asynchrones décomposent l'échange en deux phases : l'initialisation et la propagation des données. Ceci permet d'exécuter la suite du programme pendant la propagation des données. Toutefois, la propagation des données doit être traitée par un *thread* pour qu'elle puisse avoir effectivement lieu pendant le calcul et non lors de l'attente explicite de la terminaison de la communication.

Cela implique également que les calculs doivent être efficacement séquencés afin que l'échange puisse commencer le plus tôt possible tout en ayant les données prêtes à l'envoi, comme dans l'algorithme 1, page 35 de l'itération du Jacobi. C'est pourquoi les itérateurs, décrits ci-après, doivent permettre de distinguer les sommets qui doivent échanger des données, des autres sommets, afin de pouvoir commencer la communication dès que les calculs sont terminés sur le premier ensemble de sommets.

3.4.2 Itérateurs

Reprenons l'équation de Poisson que nous avons décrite en section 3.1, page 32. Pour le calcul des flux sur chaque arête du volume de contrôle, comme le décrit l'équation 3.6, page 32, il faut une première boucle sur les arêtes et, pour chaque arête, une deuxième boucle pour parcourir soit les nœuds qui en sont les extrémités, soit les éléments qui la partagent. Les itérateurs sont un type abstrait permettant de lister un sous-ensemble de sommets, que ce soit au sein d'une entité ou d'un sommet. Nous devons donc concevoir deux sortes d'itérateurs que nous appellerons respectivement « itérateurs primaires » et « itérateurs secondaires ».

3.4.2.1 Itérateurs primaires

Les itérateurs primaires énumèrent tous les sommets au sein d'une entité ou d'une sous-entité localement à un processeur.

Utilisons ces itérateurs pour illustrer, au travers des figures qui suivent, les différents types de sommets que nous avons rencontrés.

Les sommets locaux recensent tous les sommets possédés par un processus. C'est pourquoi, sur la figure 3.10, chaque maille est soit en rouge, soit en vert. Autrement dit, chaque élément ne peut appartenir qu'à un unique processeur.

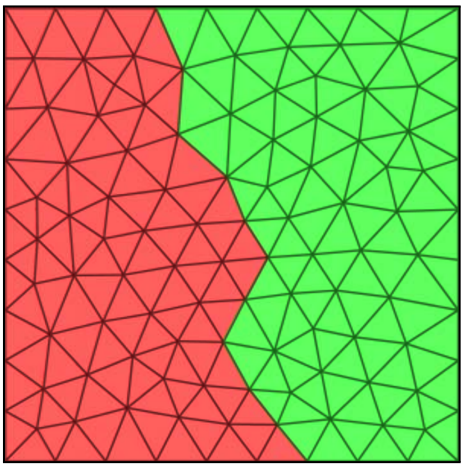


FIGURE 3.10 – Mailles locales des processeurs 0 et 1, en vert et en rouge respectivement.

Dans l'ensemble des sommets locaux, nous distinguons deux sous-ensembles. Le premier est celui des sommets internes, illustrés en figure 3.11. Ces sommets ne sont pas reliés à des sommets fantômes. Cette caractéristique explique la bande d'éléments avec un fond blanc entre les mailles rouges et vertes. Le deuxième sous-ensemble est constitué des sommets de bord, comme le montre la figure 3.12, page suivante. Ces sommets sont connectés à des sommets fantômes. Nous apercevons également que la plupart des mailles en rouge sont reliées par une face à une maille en vert. Au moins un des sommets de ces mailles doit donc appartenir au processeur voisin en tant que sommet fantôme.

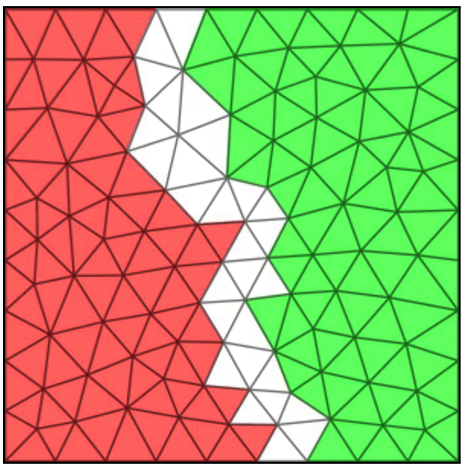


FIGURE 3.11 – Mailles internes des processeurs 0 et 1, en vert et en rouge respectivement.

Les sommets frontières sont soit des sommets locaux reliés à des sommets fantômes, soit des sommets fantômes reliés à des sommets locaux. Cette catégorie peut être couplée à une autre catégorie, si toutefois l'intersection des deux ensembles est non vide. La catégorie des sommets

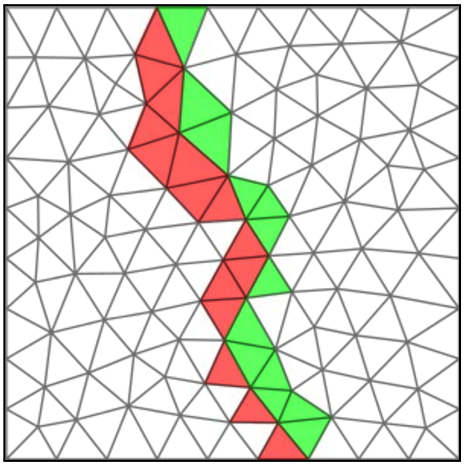


FIGURE 3.12 – Mailles de bord des processeurs 0 et 1, en vert et en rouge respectivement.

frontières ne peut pas être couplée avec celle des sommets internes.

Nous pouvons voir sur la figure 3.13, des points à l'intérieur de certains triangles, pour lesquels se sont superposées les deux couleurs. Prenons le cas du vert avec un point qui représente d'abord une coloration rouge à laquelle s'est superposée le vert, ce qui veut dire que le processeur 1 a parcouru cet élément avant le processeur 0. Ceci indique un parcours des sommets frontières de telle sorte que les sommets locaux sont listés avant les sommets distants.

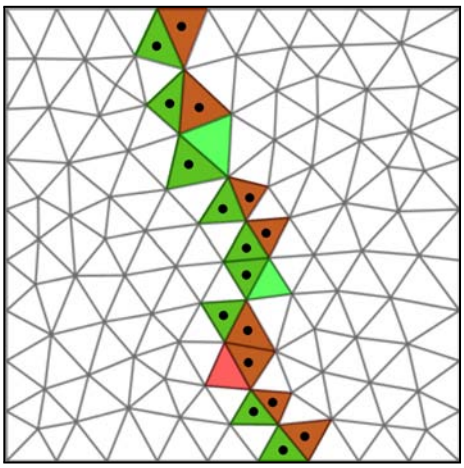


FIGURE 3.13 – Mailles frontières des processeurs 0 et 1, en vert et en rouge respectivement. Les triangles marqués d'un point représentent les mailles qui sont à la fois locales sur un processeur et distantes sur l'autre processeur.

Les sommets font partie du recouvrement s'ils sont possédés par un autre processus et qu'ils sont dupliqués, ainsi que leurs données associées, en lecture seule. Seul le processus auquel appartient le sommet peut modifier ses données sans que ces modifications soient écrasées lors de la prochaine diffusion. Une représentation d'un recouvrement de distance 1 est exposée en figure 3.14, page ci-contre.

Enfin, le dernier exemple d'itérateur de niveau primaire concerne aussi bien les sommets

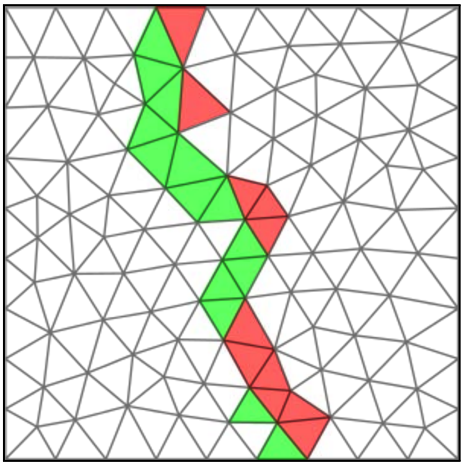


FIGURE 3.14 – Mailles du recouvrement des processeurs 0 et 1, en vert et en rouge respectivement.

locaux que les sommets du recouvrement, comme l'illustre la figure 3.15.

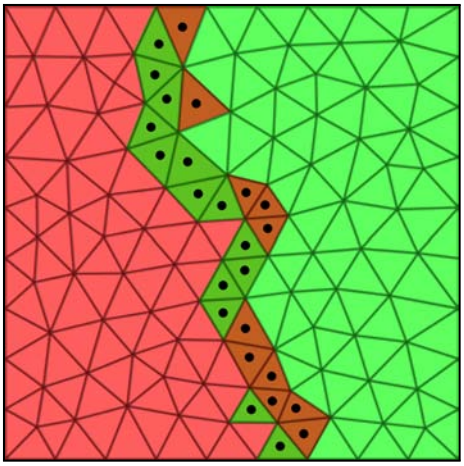


FIGURE 3.15 – Mailles locales et du recouvrement des processeurs 0 et 1, en vert et en rouge respectivement. Les triangles marqués d'un point représentent des mailles qui sont à la fois locales par un processeur et font partie du recouvrement sur l'autre processeur.

La parallélisation à grain fin en mémoire partagée du niveau primaire des itérateurs peut être aisément effectuée en découpant la liste des sommets en autant de sous-listes qu'il y a de processus légers. Chaque sous-liste contient un sous-ensemble de sommets dont les numéros sont contigus, n'altérant pas la localité spatiale.

3.4.2.2 Itérateurs secondaires

Les itérateurs de niveau secondaire énumèrent tous les sommets d'une entité ou sous-entité reliés à un sommet donné. Cela permet par exemple de calculer le volume d'un élément en parcourant les nœuds qui le constituent et, pour chacun d'entre eux, d'obtenir leurs coordonnées. Le sommet peut appartenir à une entité ou une sous-entité, tout comme les voisins énumérés.

Le sommet dont nous voulons les voisins peut être pris aussi bien dans son entité que dans sa sous-entité, sans que cela ait une conséquence sur la liste de ses voisins. En revanche, demander les voisins au sein d'une sous-entité sera plus restrictif que de demander les voisins au sein de l'entité mère.

Comme nous l'avons expliqué dans la section 3.2.2, page 36, les sommets obtenus au moyen des itérateurs peuvent être identifiés par plusieurs numéros différents, selon que l'on considère leur numéro au sein de leur entité ou de leur sous-entité.

Afin d'illustrer les différentes numérotations qui peuvent être manipulées par les itérateurs, nous allons prendre l'exemple d'un maillage distribué sur trois processeurs, illustré par la figure 3.16. Chaque sommet possède un numéro global unique quelle que soit l'entité à laquelle il appartient.

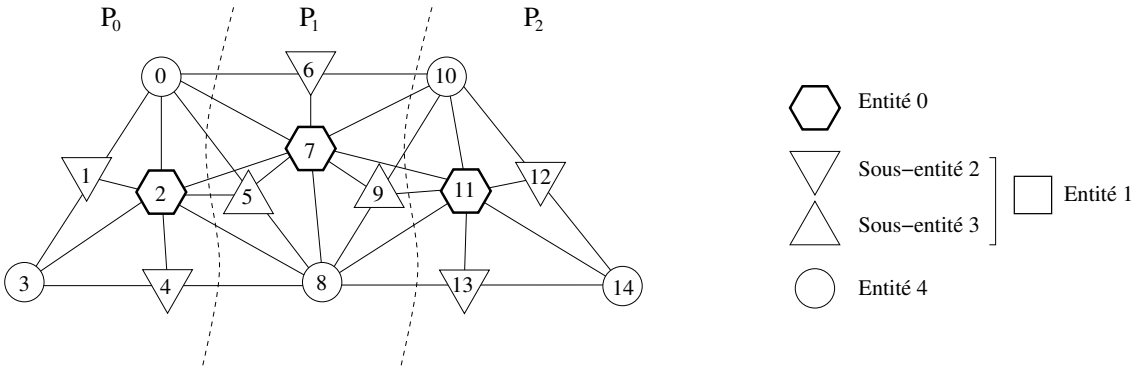


FIGURE 3.16 – Numérotation globale des sommets d'un maillage distribué sur trois processeurs. Chaque sommet appartient à une entité (cercles, hexagones, triangles) et éventuellement à une sous-entité (triangles pointant vers le haut ou vers le bas).

Afin de manipuler les données localement par processeur et de manière compacte par entité, il est possible d'obtenir la numérotation par entité locale à chaque processeur, comme illustré en figure 3.17, page ci-contre. À la différence du maillage montré en figure 3.4, page 37, la numérotation commence à partir de la valeur zéro.

Si nous distinguons, dans l'entité arête, les arêtes internes et les arêtes de bord, nous obtenons la numérotation de la figure 3.18, page ci-contre.

Dans le cas où un sommet appartient à une sous-entité en plus d'appartenir à son entité mère, un accesseur permet d'obtenir les deux informations. Par exemple, le nœud 0 du processeur P_1 de la figure 3.17, page suivante est relié aux arêtes $3, 0, 2, 4$, alors que dans la figure 3.18, page ci-contre, il est relié aux arêtes internes 0 et 1 et aux arêtes de bord 1 et 2. Ainsi, nous traitons différemment les arêtes voisines du nœud suivant selon que nous souhaitons accéder à celles qui sont sur le bord ou internes au maillage.

3.4.3 Recouvrement

Avant de parler de recouvrement, donnons une illustration d'un graphe enrichi distribué (voir section 2.6, page 25, du chapitre 2). Par défaut, les sommets qui sont dupliqués sont des sommets halos (voir définition 35, page 28), comme illustré en figure 3.19, page suivante.

Les sommets représentés en noir sont les sommets « halo », ce qui implique que ces sommets n'ont pas d'adjacence sortante. Or, cette information est parfois nécessaire, comme dans le cas de l'équation de Poisson où le parcours des arêtes se fait aussi bien sur les arêtes locales que sur

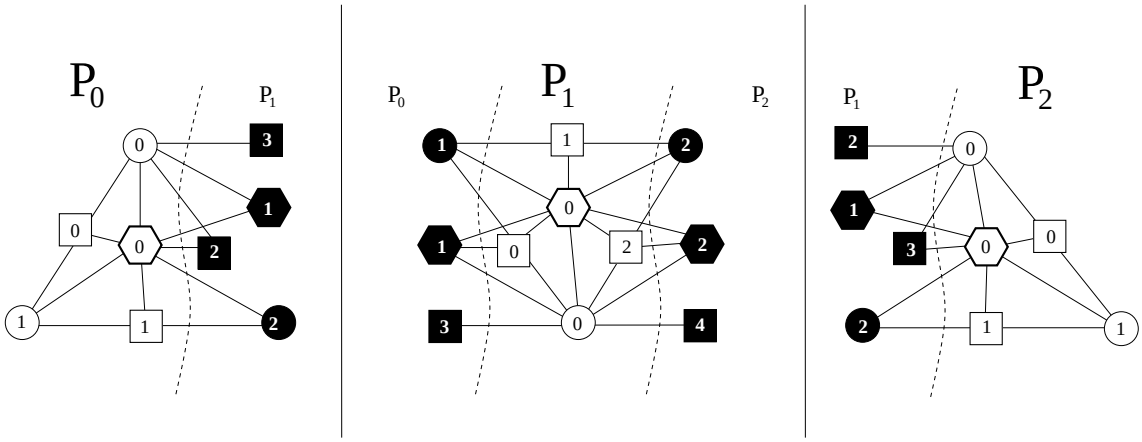


FIGURE 3.17 – Numérotation par entité, locale à chaque processeur des sommets (nœuds, éléments, arêtes) du maillage distribué de la figure 3.16, page précédente.

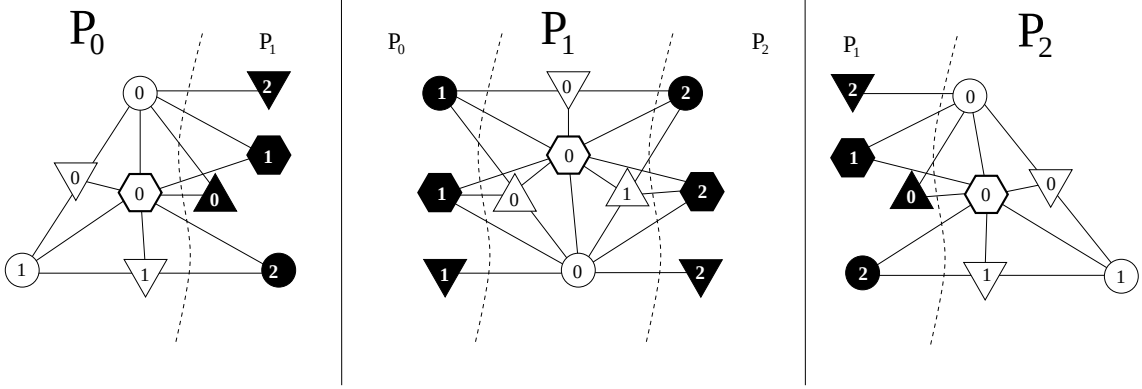


FIGURE 3.18 – Numérotation par sous-entité, locale à chaque processeur des sommets (nœuds, éléments, arêtes internes et arêtes de bord) du maillage distribué de la figure 3.16, page ci-contre.

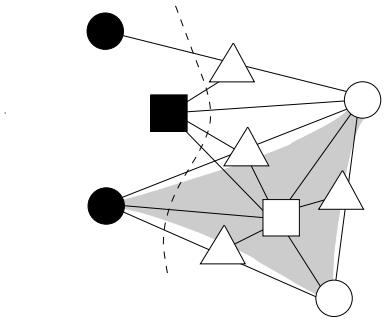


FIGURE 3.19 – Exemple de graphe enrichi distribué sans recouvrement restreint au processeur 2.

les arêtes fantômes, ceci afin de mettre à jour la solution portée par les nœuds locaux reliés à une arête fantôme.

En règle générale, les algorithmes numériques se basent sur des mailles complètes. Or, il est très difficile d'obtenir des mailles entières distantes à partir d'un «halo», sauf si le graphe forme une clique pour chaque maille. Une méthode pour cela serait d'étendre le «halo» suffisamment afin de construire les mailles distantes nécessaires, quitte à construire des sommets «halos» qui ne seront pas utilisés. Qui plus est, les données portées par ces sommets seraient échangées, alors qu'elles ne seraient pas utilisées. Par conséquent, il est nécessaire de construire autrement le recouvrement des mailles, par le biais des éléments. Deux types de relations sont nécessaires :

— les relations d'élément à élément ;
— les relations d'élément à tout autre sommet de la maille.

Ces relations sont illustrées par les figures 3.20 et 3.21 en page 50.

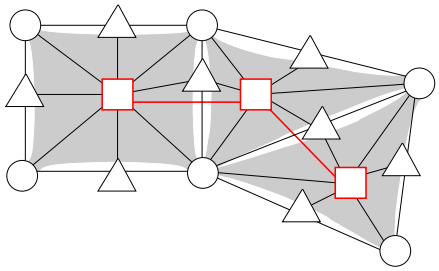


FIGURE 3.20 – Relation élément - élément pour un maillage.

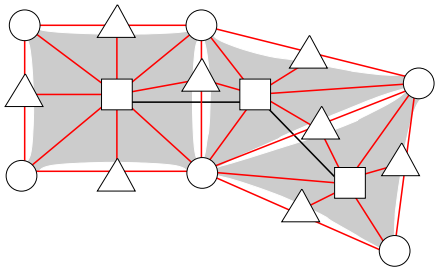


FIGURE 3.21 – Relation élément - maille pour un maillage.

Nous pouvons distinguer deux types de recouvrement. L'un est généraliste et se construit de proche en proche par diffusion à travers les sommets de toutes les entités. L'autre, plus spécifique, est extensible uniquement par le biais d'une seule entité. Le premier s'applique à notre algorithme 1, page 35, alors que le second convient pour des méthodes Galerkin discontinues, afin d'obtenir un recouvrement par le biais des faces.

Afin de clarifier ces deux notions, considérons l'exemple de la figure 3.22, page suivante. Les sommets en noir, qui font partie du recouvrement, peuvent être obtenus soit par un recouvrement généraliste de distance 1, soit par un recouvrement spécifique par le biais des arêtes. Ce deuxième choix s'explique par le fait que l'arête représentée sous forme d'un triangle partageant d'un côté un élément en fond blanc et un autre élément en fond grisé, est reliée à un élément fantôme, d'où le recouvrement de cet élément en fond blanc. Nous pouvons ajouter comme précision que le recouvrement généraliste peut être de distance n, alors que le recouvrement spécifique est obligatoirement de distance 1, s'il est utilisé. Nous avons fixé cette limite du recouvrement spécifique en fonction des besoins exprimés par les numériciens.

Le recouvrement généraliste de distance 1 est calculé en chaque maille dont les sommets sont répartis sur plusieurs processus. Chacun des processus qui la partage, recopie la partie manquante afin d'avoir une maille complète. Les sommets distants localisés sur un processus voisin sont recopiés sur le processus local, ainsi que leur adjacence et leurs données associées.

Le recouvrement de distance 1 est illustré par la figure 3.23, page ci-contre.

À cause de la relation élément à élément, les éléments fantômes mais qui sont reliés à une maille du recouvrement ont dû être traités de manière particulière. Ces éléments sont nécessaires pour calculer le recouvrement, mais ne sont pas utiles pour le numéricien. Par conséquent, nous avons choisi de traiter ces éléments comme des sommets “halo”.

Le recouvrement de distance n, avec $n > 1$, se définit récursivement. Si un élément v fait

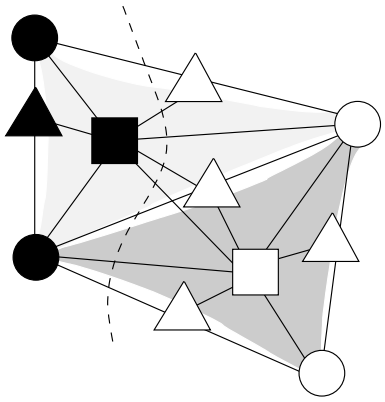


FIGURE 3.22 – Exemple de graphe enrichi distribué avec recouvrement de distance 1, restreint à un processeur. Les sommets en blanc sont locaux et ceux en noir sont des sommets fantômes. Le triangle noir n'est le voisin direct d'aucun sommet blanc mais est présent car il appartient à une maille dont certains sommets sont blancs.

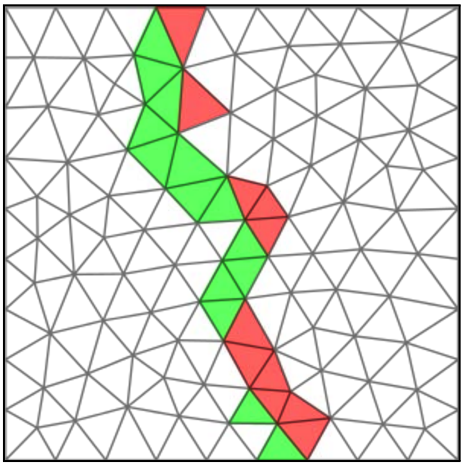
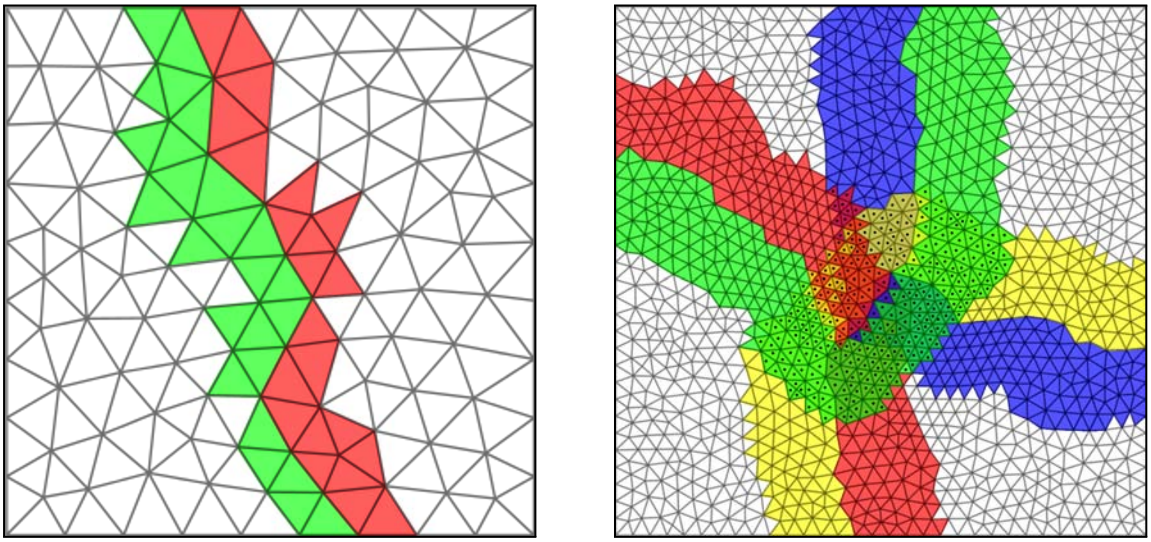


FIGURE 3.23 – Mailles du recouvrement de distance 1 des processeurs 0 et 1, en vert et en rouge respectivement.

partie du recouvrement de distance $n-1$, alors chaque élément v' voisin de v, v' n'appartenant pas au recouvrement de distance $n-1$, est ajouté par transitivité au recouvrement de distance n. La figure 3.24a, illustre un recouvrement de distance 2.



(a) Recouvrement de distance 2 des processeurs 0 et 1, en vert et en rouge respectivement.

(b) Recouvrement de distance 10 des procsseurs 0, 1, 2 et 3 en vert, en rouge, en bleu et en jaune respectivement.

FIGURE 3.24 – Mailles impliquées dans des recouvrements de différentes distances. Les triangles marqués d'un point représentent des mailles qui font partie du recouvrement sur plusieurs processeurs.

Un exemple de recouvrement spécifique, par le biais des arêtes, est présenté en figure 3.25, page ci-contre.

3.5 Validation expérimentale

3.5.1 Cas test

Nous proposons de résoudre l'équation de Poisson au moyen d'une méthode d'éléments finis [117]. Le système linéaire correspondant est ensuite résolu en utilisant la méthode de Jacobi. Les maillages sur lesquels nous allons résoudre l'équation de Poisson sont composés de tétraèdres. Chaque maillage représente un cube dont les coordonnées en x, y et z varient entre 0 et 1. Nous allons tester le passage à l'échelle faible en augmentant la taille du maillage et le nombre de processeurs, tout en conservant un nombre d'itérations de Jacobi fixé. Les tailles de maillages varient entre 6 et 62 millions d'éléments alors que le nombre de processeurs varie entre 3 et 31.

Nous allons également vérifier la scalabilité forte en utilisant un maillage suffisamment grand, de 62 millions d'éléments, seul le nombre de processeurs variant entre 5 et 65 processeurs.

3.5.2 Résultats

Nous allons analyser les résultats obtenus pour le cas test ci-dessus en commençant par comparer le partitionnement des éléments au partitionnement de tous les sommets. Nous vérifierons ensuite les scalabilités faible et forte du partitionnement.

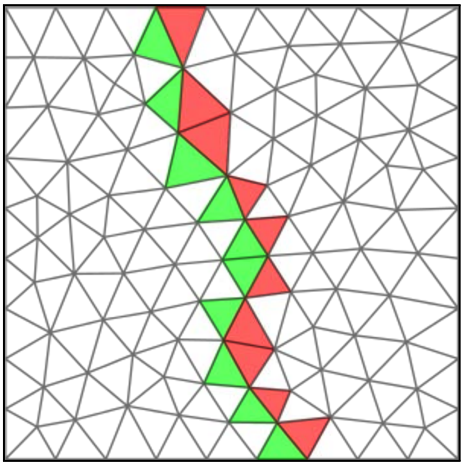


FIGURE 3.25 – Mailles du recouvrement spécifique des processeurs 0 et 1 en vert et en rouge respectivement.

3.5.2.1 Partitionnement des éléments versus partitionnement de tous les sommets

Nous avons comparé les deux méthodes de partitionnement sur un maillage à 62 millions d'éléments, partitionné sur 10 puis 40 processeurs.

La figure 3.26, page suivante représente les tailles des parties, en nombre de nœuds, dans le cas du partitionnement sur 10 processeurs. Comme nous pouvons le voir, l'écart de taille des sous-domaines est plus petit en utilisant le partitionnement sur tous les sommets qu'avec le partitionnement sur les éléments. La différence est plus prononcée lorsque nous partitionnons le maillage sur 40 processeurs, comme le montre la figure 3.27, page suivante.

3.5.2.2 Scalabilité faible

La figure 3.28, page 55 montre que les temps oscillent, alors que nous devrions avoir dans l'absolu une droite parallèle à l'axe des x. Ces écarts de valeurs sont difficilement interprétables, parce que le rapport entre le nombre d'éléments et le nombre de processeurs est relativement constant, aux alentours de deux millions d'éléments par processeur. Si nous calculons la différence entre les rapports maximum et minimum, nous obtenons moins de 100000 éléments d'écart. Cette différence ne justifie pas les différences observables sur la figure. Au vu des résultats que nous obtenons actuellement, nous ne pouvons pas conclure que notre implémentation de l'équation de Poisson est faiblement scalable.

Par ailleurs, l'exécution du programme a échoué sur des maillages de tailles plus grandes. La raison en est très probablement la limitation de représentation des entiers, qui sont pour le moment codés sur 32 bits. De fait, aucune numérotation globale des sommets ne peut dépasser 2 milliards d'items. Lors du codage de nos algorithmes, il a été prévu que la taille des entiers puisse être paramétrée à la compilation, mais cette fonctionnalité n'a pas encore été testée dans le cadre d'entiers à 64 bits.

3.5.2.3 Scalabilité forte

Contrairement à la scalabilité faible, les résultats sont plus exploitables, comme le montre la figure 3.29, page 56. Il n'est pas possible de lancer nos tests sur moins de 10 processeurs, parce

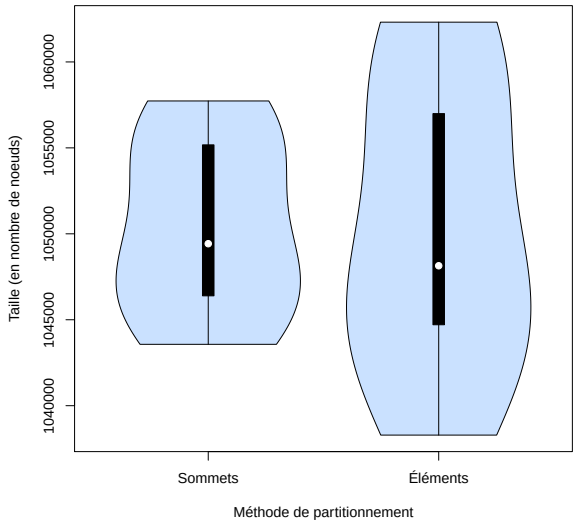


FIGURE 3.26 – Densité des tailles des sous-domaines, selon que le partitionneur opère sur graphe des éléments ou sur celui de tous les sommets du maillage, dans le cas d'une distribution sur 10 processeurs. La densité correspondant à la zone bleue est donnée avec une estimation par noyau [121]. Le trait épais noir représente l'écart interquartile [116], autrement dit les valeurs comprises entre 25 et 75 % de l'intervalle total. Le point blanc désigne la valeur moyenne.

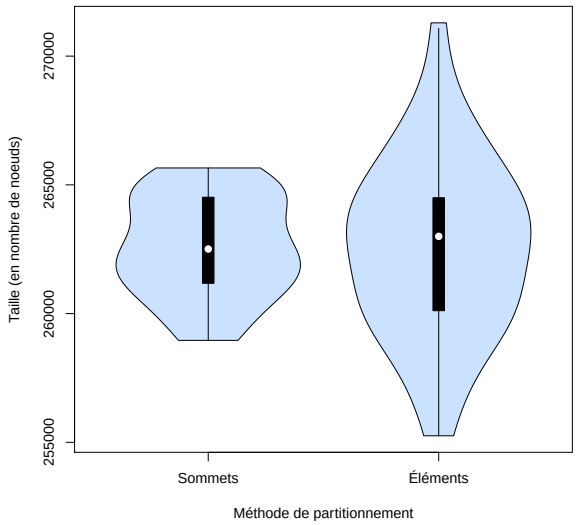


FIGURE 3.27 – Densité[8] des tailles des sous-domaines, selon que le partitionneur opère sur graphe des éléments ou sur celui de tous les sommets du maillage, dans le cas d'une distribution sur 40 processeurs. La densité correspondant à la zone bleue est donnée avec une estimation par noyau [121]. Le trait épais noir représente l'écart interquartile [116], autrement dit les valeurs comprises entre 25 et 75 % de l'intervalle total. Le point blanc désigne la valeur moyenne.

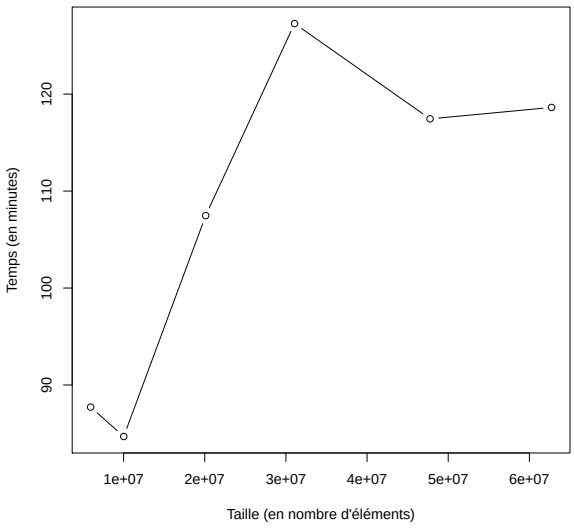


FIGURE 3.28 – Temps d'exécution de l'implémentation de l'équation de Poisson en utilisant Jacobi comme méthode de résolution, pour des maillages de tailles croissantes en gardant une charge moyenne de deux millions d'éléments par processeur.

que le maillage est trop volumineux pour être distribué sur un nombre plus petit de processeurs. Nous n'avons pas non plus pu générer de résultats sur plus de 65 processeurs, les exécutions sur un nombre plus important ayant échoué pour une raison encore inconnue.

Néanmoins, les valeurs dont nous disposons déjà témoignent de la bonne scalabilité de nos algorithmes et de leur mise en œuvre, cette scalabilité étant proche de l'accélération idéale.

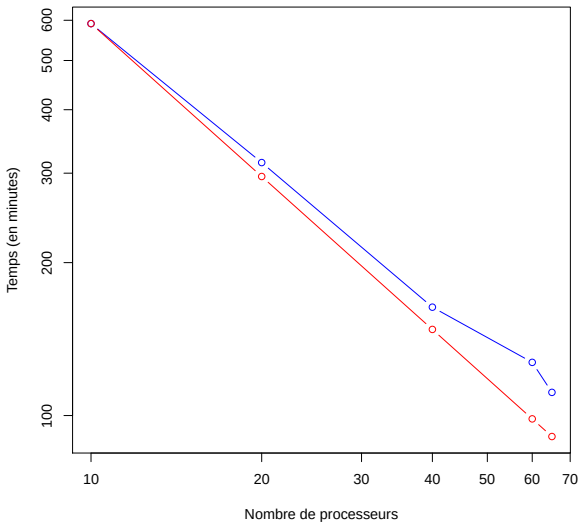


FIGURE 3.29 – Temps d'exécution (en bleu) de l'implémentation de l'équation de Poisson en utilisant Jacobi comme méthode de résolution, sur un maillage de 62 millions d'éléments. La courbe en rouge représente le temps mis par le programme sur 10 processeurs, rapporté au nombre de processeurs, pour chaque valeur de la courbe bleue.

Chapitre 4

Influence de la renumérotation sur la vitesse d'exécution

Sommaire

4.1 Introduction

Lorsque le temps d'exécution d'une application demeure trop important, l'optimisation du code source s'avère utile et nécessaire. Certains outils informatiques permettent de quantifier le temps passé dans chaque partie du code. Sachant cela, les zones à modifier sont identifiées et modifiées afin de diminuer le temps d'exécution.

Les « défauts de cache » sont une cause potentielle de mauvaise performance d'une application. Afin de comprendre à quoi cela correspond, nous nous intéresserons brièvement à l'architecture des ordinateurs [36], et en particulier au cache mémoire et à son interaction avec les autres parties de la machine. Ensuite, nous nous focaliserons sur une partie d'un algorithme de remaillage, pour laquelle le temps d'exécution dépend essentiellement de la latence de la mémoire, et donc de la performance du cache. Dans un troisième temps, nous présenterons notre solution, qui peut être parallélisée selon deux niveaux de granularité. Enfin, nous validerons notre méthode grâce à un ensemble d'expérimentations.

4.2 Exécution des programmes et principes de localité

Un ordinateur est schématiquement constitué d'un processeur, qui effectue les calculs, et d'une mémoire centrale, qui stocke les programmes ainsi que leurs données.

La mémoire centrale d'un ordinateur peut être vue comme un ensemble de cellules de taille identique, appelés « mots », dont chacune peut stocker une valeur numérique distincte. Ces cellules sont identifiées par une valeur numérique unique, appelée adresse. L'adresse peut donc être vue comme le « numéro » de la cellule.

Cependant, la mémoire centrale n'est pas aussi rapide que le processeur, pour de multiples raisons. La première est que, les mémoires étant de tailles de plus en plus importantes, les circuits mémoire ne peuvent être placés au plus près de l'unité de traitement du processeur. De fait, les transferts de données entre la mémoire et le processeur subissent une latence de plusieurs cycles, du fait de la distance que le signal doit parcourir. De plus, la commutation rapide des circuits génère une chaleur importante par effet Joule, et consomme beaucoup d'énergie. Si cette dépense est inévitable pour le processeur, elle n'est pas acceptable pour la mémoire, celle-ci représentant bien plus de transistors que le processeur. La technologie de stockage de l'information mise en œuvre au sein de la mémoire centrale est donc bien moins gourmande que la technologie utilisée au sein des processeurs, mais induit des temps de commutation bien moins rapides. Or, il est inutile de concevoir un processeur rapide s'il doit passer la presque totalité de son temps à attendre que lui soient fournies les données dont il a besoin.

Une première solution consiste à ce que le processeur ne calcule pas directement en mémoire, mais entre ses « registres ». Ceux-ci peuvent être vus comme les mots d'une mémoire spécifique et très rapide, hébergée au cœur du processeur. L'objectif des programmeurs et des compilateurs est alors d'utiliser le plus possible des données déjà contenues dans les registres, et de minimiser les transferts d'information entre les registres et la mémoire centrale. Or, les registres sont trop peu nombreux pour contenir toutes les données utiles à un instant donné, et la pénalité de chaque transfert en mémoire est encore bien trop coûteuse.

Ce problème peut néanmoins être partiellement résolu en tirant parti de deux principes découverts grâce à des études sur le comportement des programmes : les principes de localité.

— Le principe de localité temporelle stipule que, plus une donnée a été accédée récemment, et plus sa probabilité d'accès est élevée. C'est le cas par exemple du code des instructions contenues dans les boucles, ainsi que des compteurs de boucle que l'on incrémente.

— Le principe de localité spatiale stipule pour sa part que plus une donnée est proche de la dernière donnée accédée, et plus sa probabilité d'accès est élevée. C'est le cas par exemple des cases d'un tableau que l'on parcours, ou des différents champs d'une structure de données.

C'est ainsi qu'est intercalée, entre le processeur et la mémoire centrale, une mémoire appelée « mémoire cache » ou « antémémoire » [111]. Celle-ci est plus rapide que la mémoire centrale, tout en étant moins rapide que les registres. Son rôle est de conserver au plus près du processeur les données ayant été récemment utilisées, ou pouvant l'être dans un futur proche, en accord avec les principes de localité. Son fonctionnement est le suivant : lorsque l'unité de traitement demande une donnée à la mémoire centrale, sa requête est interceptée par le cache. Deux cas de figure se présentent alors :

1. si la donnée est déjà présente dans le cache, alors elle est retransmise à l'unité de traitement ;
2. si la donnée n'est pas ou plus présente dans le cache (on parle alors de « défaut de cache »), alors la demande est transférée à la mémoire centrale. Dans ce cas, le cache ne demande pas à la mémoire centrale que la donnée elle-même, mais également ses données voisines. Ainsi, si jamais le processeur les demande à son tour en vertu du principe de localité spatiale, le cache sera à même d'y répondre sans avoir à solliciter à nouveau la mémoire centrale. Pour cela :
   (a) le cache demande à la mémoire centrale un groupe de données comprenant la donnée voulue ; cet ensemble est appelé « ligne de cache »,
   (b) dans le même temps, le cache supprime de sa mémoire interne une ligne parmi celles ayant été le moins récemment demandées par le processeur (la probabilité que le processeur la demande à nouveau étant la plus faible, en vertu des principes de localité),
   (c) lorsque la ligne est fournie par la mémoire centrale, le cache transmet la donnée voulue au processeur, et recopie en son sein l'intégralité de la ligne, pour une probable réutilisation.

En fait, les principes de localité peuvent être appliqués de façon générale à l'ensemble des dispositifs mémoire de l'ordinateur. Il en découle une « hiérarchie mémoire », telle qu'au plus près de l'unité de traitement se trouvent des mémoires rapides et de petites tailles, et à plus grande distance des mémoires de grande capacité, très peu chères mais peu rapides :

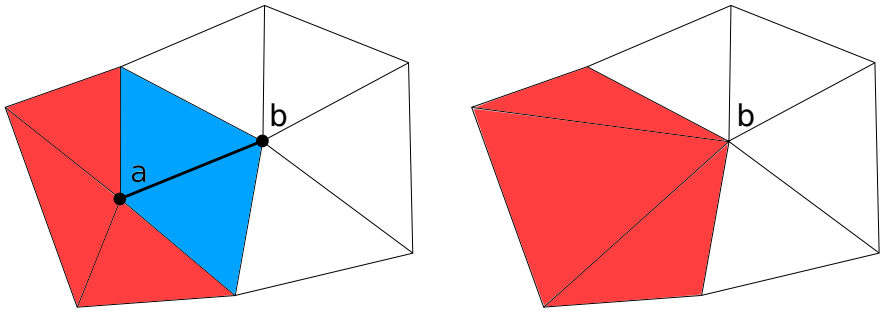
— registres ;
— mémoires caches (le pluriel est de mise, comme nous allons le voir) ;
— mémoire centrale ;
— disque dur ;
— bandes magnétiques.

Afin d'équilibrer la hiérarchie mémoire en termes de capacités et de latences, les constructeurs d'ordinateurs empilent habituellement plusieurs niveaux de cache, aux rôles différents :

— le premier niveau (L1, pour « *level 1* »), situé au plus près de l'unité de traitement est le plus rapide mais de petite taille, inférieure à 100 Kio. Son but est de stocker les instructions de la fonction en cours d'exécution (en particulier celles des boucles) et les données des champs des structures ;
— le deuxième niveau (L2) est moins rapide mais de taille plus importante que le cache L1, jusqu'à 1 Mio. Son rôle est de stocker des pans entiers de tableaux de données et le code de plusieurs fonctions ;
— le troisième niveau (L3) est facultatif. Sa présence dépend du différentiel de latences et de vitesse entre le cache L2 et la mémoire centrale. Sa taille est inférieure à 10 Mio.

4.3 Influence des caches sur les routines de remaillage

Afin d'illustrer l'influence de la prise en compte des effets de cache sur la vitesse d'exécution d'un remaillage, nous allons nous intéresser à l'une des techniques élémentaires de remaillage : la suppression d'une arête à l'intérieur d'un maillage. Cette technique consiste à modifier les éléments ayant pour sommet une extrémité de l'arête, et ainsi nécessite de connaitre les numéros de ces éléments, comme illustré en figure 4.1.



(a) Avant suppression. (b) Après suppression.

FIGURE 4.1 – Exemple de suppression d'une arête entre les nœuds a et b d'un maillage. Les éléments en bleu sont supprimés, alors que, dans le cas des éléments en rouge, le nœud a est fusionné au nœud b.

La suppression d'une arête consiste à fusionner les deux nœuds a et b, extrémités d'une arête. Sans perte de généralité, cela revient à remplacer a par b dans tous les éléments contenant a. Si un élément possède parmi ces sommets les deux extrémités de l'arête, alors il est supprimé. Algorithmiquement, cela se traduit par un parcours des éléments possédant a et, pour chacun de ces éléments, l'énumération des nœuds le constituant. En fonction des résultats de ce parcours, soit les deux nœuds sont fusionnés, soit l'élément est supprimé. En supposant que les éléments partageant un nœud soient accessibles en temps constant, il est nécessaire de stocker, pour chaque élément, la liste de ses nœuds.

Si les éléments ont des numéros trop éloignés pour qu'une ligne de cache les contienne tous, alors un ou plusieurs défaut de cache se produiront, ce qui affectera directement le temps d'exécution. Plus le nombre de défauts de cache sera important, et plus le temps d'exécution sera élevé.

4.4 Algorithme de renumérotation

La solution que nous avons mis en œuvre, consiste à renuméroter les différentes entités (éléments, faces, arêtes et nœuds) du maillage, afin de diminuer les défauts de cache, dans le but de diminuer le temps d'exécution.

Ce n'est pas le premier niveau de cache qui nous intéresse le plus, mais les niveaux suivants. En effet, le cache L1 est naturellement mobilisé lors du parcours des listes d'adjacence, qui représentent la majorité du volume des données. Notre objectif est donc de produire des renumérotations qui favoriseront la localité des données au sein des caches L2 et L3, c'est-à-dire sur des ensembles de listes d'adjacence. Il s'agit de regrouper des sommets voisins, susceptibles d'être accédés à peu de temps d'écart au sein du même algorithme.

Si nous reprenons l'exemple précédent, cela revient à vouloir numéroter les éléments de manière à ce que le plus grand écart entre le numéro d'un élément et ceux de ses voisins soit le plus petit possible, afin que les données des deux éléments aient la probabilité la plus grande possible d'être localisés au sein de la même ligne de cache ou page mémoire. Cette problématique ressemble aux numérotations de matrices de type Cuthill-McKee ou Gibbs-Poole-Stockmeyer [110] utilisées pour la renumérotation de matrices creuses. Une telle numérotation doit en fait être appliquée à toutes les entités.

Il est difficile d'obtenir une numérotation qui vérifie en chaque élément que l'écart maximal soit inférieur à la taille du deuxième niveau de cache. Il est toutefois possible de l'obtenir pour la majorité des éléments, si nous formons des groupes d'éléments, de telle sorte que les éléments soient connexes au sein de chaque groupe. Ainsi, pour chaque élément, ses voisins auront des numéros dont l'écart avec le numéro de l'élément sera inférieur à la taille du deuxième niveau de cache. Cependant, si l'élément possède un numéro supérieur à ses voisins, alors les autres éléments ne seront pas préchargés en mémoire. La probabilité que les éléments soient parcourus par ordre décroissant reste relativement faible, sachant que l'ordre de parcours est aléatoire. Par conséquent, nous bénéficierons de fait qu'un élément appartienne à un groupe probablement avant d'avoir fini de parcourir ses voisins. Nous pouvons appliquer le même raisonnement pour un nœud situé à la frontière entre deux groupes. Réaliser une première lecture d'un élément d'un groupe voisin permettra de précharger les autres éléments de ce groupe, voisins de cet élément.

Nous allons maintenant aborder plus en détail la constitution des groupes, puis comment nous numérotons les éléments groupe par groupe.

4.4.1 Constitution des groupes

Un moyen de grouper des données à consiste à utiliser un partitionneur de graphe. Ceci nécessite de construire un graphe à partir du maillage. Nous avons choisi de représenter le maillage sous la forme de son graphe des éléments. Autrement dit, les sommets du graphes correspondent aux éléments du maillages et une arête existe entre deux éléments s'ils partagent une face.

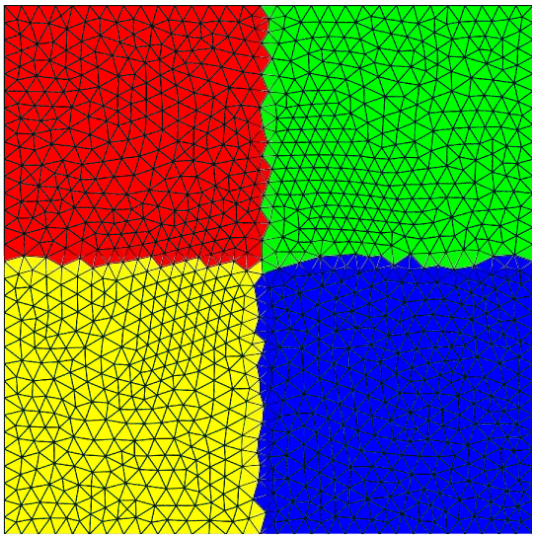


FIGURE 4.2 – Formation sur les éléments d'un maillage de 4 groupes (représentés en bleu, rouge, jaune et vert).

Ce graphe est ensuite partitionné. Par exemple, la figure 4.2, page précédente représente un maillage partitionné en quatre sous-ensembles (bleu, rouge, vert et jaune). Chaque sous-ensemble correspond à un groupe. Le nombre de groupes est calculé de telle sorte qu'un groupe puisse être contenu dans le cache de deuxième niveau. Au vu des tailles habituelles de ces caches au sein des architectures modernes, les groupes doivent donc contenir environ 500 éléments.

Le but du partitionnement est d'obtenir des groupes compacts de sommets. En revanche, il n'est pas nécessaire que tous les groupes soient exactement de la même taille, comme c'est le cas lorsque le partitionnement sert à répartir une charge de calcul. Ceci nous offre une certaine flexibilité dans le choix de la méthode de partitionnement. La rapidité étant pour nous le critère prépondérant, nous avons opté pour une méthode multi-niveaux k-aire, illustrée en figure 4.3. Les algorithmes d'optimisation locale que nous utilisons pour raffiner les partitions à la remontée du schéma multi-niveaux sont paramétrés pour être moins précis que lors d'un partitionnement classique, mais plus rapides.

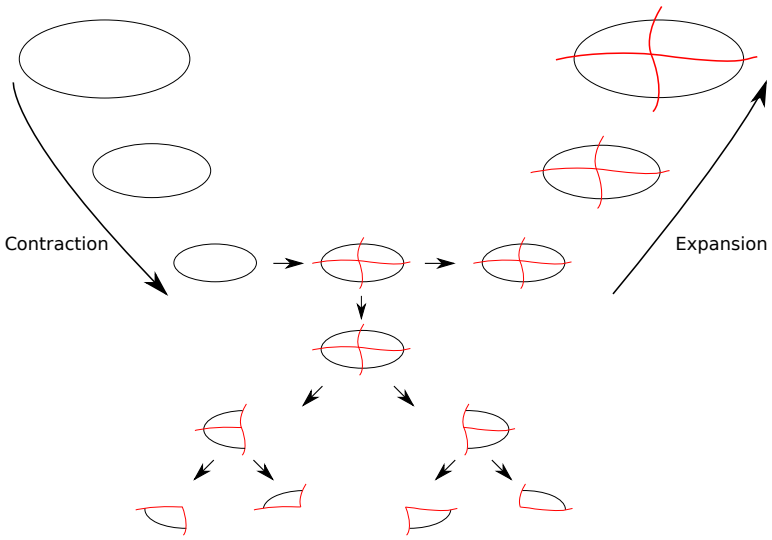


FIGURE 4.3 – k-Partitionnement séquentiel sur un graphe centralisé.

La renumérotation au sein du remailleur se dégrade naturellement au cours du temps. En effet, plusieurs des techniques appliquées sur le maillage modifient sa topologie, les nouveaux sommets créés (nœuds et éléments confondus) ayant nécessairement des numéros arbitraires, et donc arbitrairement éloignés de leurs voisins.

Deux types de méthodes peuvent être utilisées. Le premier type, qui recalcule intégralement une nouvelle renumérotation, peut être appelé « *scratch-reorder* » par analogie avec les méthodes de repartitionnement appelées « *scratch-remap* ». Le deuxième type est celui des méthodes itératives, qui cherchent à optimiser une renumérotation existante.

4.4.2 Renumérotation *scratch-remap* : au sein de chaque groupe

Puisque le maillage est stocké en mémoire, les différentes entités possèdent déjà une numérotation initiale. Pour créer une nouvelle numérotation en nous appuyant sur les groupes formés par le partitionneur, il nous faut également définir la numérotation individuelle des éléments au sein de chaque groupe. Autrement dit, un élément possède un numéro unique, indépendamment du groupe auquel il appartient. Nous souhaitons que la numérotation soit contigüe par groupe.

Une méthode possible pour les renuméroter consiste à remplir un tableau de paires, avec chaque paire contenant le numéro du groupe donné par le partitionneur et l'ancien numéro de l'élément. Ensuite, nous trions ce tableau uniquement en fonction de la valeur du groupe, afin de déterminer la nouvelle numérotation.

Il reste à renuméroter les autres entités. Le principe est le même pour chacune d'entre elles. Pour chaque entité du maillage, nous appliquons la méthode suivante. Nous commençons par définir un numéro courant, initialisé à 1. Nous parcourons les éléments dans l'ordre de la nouvelle numérotation. Pour chaque instance de l'entité en question qui compose l'élément en cours et qui n'a pas déjà été renuméroté, nous attribuons à celle-ci le numéro courant, qui est ensuite incrémenté.

Les structures de données qui accèdent aux éléments voisins d'un élément sont alors reconstruites en fonction de cette nouvelle numérotation.

4.4.3 Renumérotation itérative

Dans la section dédiée à la validation expérimentale, nous verrons que les temps pris par la renumérotation *scratch-reorder* décrite précédemment ne sont pas négligeables. C'est pourquoi nous avons également développé une technique de renumérotation itérative, qui s'appuie sur une renumérotation précédente.

La méthode de renumérotation que nous allons présenter est donc appelée lorsque la numérotation précédente a été altérée et que la localité des numéros de sommets doit être restaurée, pour bénéficier des effets de cache. Le remailleur a pu supprimer des éléments et en ajouter d'autres, en réutilisant les numéros ainsi libérés. Nous souhaitons nous appuyer sur l'ancienne numérotation pour calculer la nouvelle, sans passer par la constitution et la numérotation de groupes, qui requiert l'appel coûteux au partitionneur.

Pour cela, nous utilisons une simple file, destinée à stocker les sommets du maillage appartenant à une entité donnée. Nous commençons par parcourir par ordre croissant l'ensemble des sommets de l'entité en question, et par ajouter ces sommets à la file s'ils appartenaient à l'ancienne numérotation. Ensuite, tant que la file n'est pas vide, nous extrayons le premier sommet de la file, en lui affectant le premier nouveau numéro disponible, et en ajoutant dans la file ses voisins s'ils ont été créés depuis lors et qu'un nouveau numéro ne leur a pas encore été attribué. Si la file est vide, nous reprenons le premier parcours de sommets jusqu'à ce qu'elle ne le soit plus.

Cette renumérotation itérative, plus rapide à calculer, est bien évidemment moins efficace que celle avec des groupes, comme nous le verrons dans la validation expérimentale 4.6, page suivante.

4.5 Parallélisation de l'algorithme de renumérotation

La première méthode de renumérotation possède la particularité d'être parallélisable à grain fin, autrement dit au moyen de processus légers. Elle peut également bénéficier d'une parallélisation à gros grain, c'est-à-dire par le biais de processus MPI. Ces deux types de parallélisation ont été implémentées par François Pellegrini au sein du logiciel PT-Scotch en utilisant des threads POSIX.

4.5.1 Parallélisation à grain fin

Cette parallélisation a été réalisée dans l'algorithme de partitionnement que nous allons détailler maintenant. Elle permet de tirer parti des processeurs multi-cœurs qui tendent à être de plus en plus répandus.

Le partitionnement utilise des algorithmes de contraction de graphe et d'appariement. La contraction permet de fusionner des sommets afin d'obtenir un graphe plus petit sur lequel sera effectué un premier partitionnement. L'algorithme d'appariement est requis afin de trouver des paires de sommets qui vont être fusionnés.

4.5.2 Parallélisation à gros grain

La première renumérotation peut également être parallélisée à gros grain. Elle peut tout à fait être exploitée dans le cadre de notre renumérotation. Tout d'abord et comme nous l'avons vu dans le chapitre 1, la contraction de graphe utilisée dans le k-partitionnement peut se paralléliser. Concernant la numérotation par groupes, elle peut être appliquée dans le même temps sur une partie des groupes par chacun des processus. Afin de renuméroter les éléments au sein de chaque groupe et sur chaque processeur, il est nécessaire d'obtenir le premier numéro qui sera utilisé par le processeur, par le biais d'un *scan*, qui consiste pour un processeur p à sommer le nombre d'éléments des processus 0 à $p-1$. Enfin, chaque processeur peut numéroter les éléments au sein de chaque groupe.

En revanche, l'algorithme d'appariement parallélisé à grain fin serait plutôt incompatible avec la parallélisation à gros grain. Pour en expliquer la raison, il est nécessaire de détailler un peu la parallélisation à gros grain de l'algorithme d'appariement. Celui-ci nécessite des tampons de requêtes afin de synchroniser les appariements entre processus voisins et de déterminer quels sommets vont être fusionnés à la frontière. Si nous utilisons en plus une parallélisation à grain fin, les tampons utilisés entre processus MPI peuvent être remplis par n'importe quel processus léger. Afin que chacun puisse écrire sans écraser les données écrites par les autres processus légers, il est nécessaire de protéger ces tampons contre les écritures concurrentes. Supposons que les sommets soient distribués aléatoirement sur les différents processus MPI, autrement dit les voisins d'un sommet peuvent appartenir à différents processeurs. Par conséquent, lors de l'appariement du sommet, les processus légers risquent de rentrer souvent en conflit lors du remplissage du tampon.

4.6 Validation expérimentale : scalabilité et comparaison avec Shrimp

Nous allons dans un premier temps valider expérimentalement le passage à l'échelle de notre renumérotation, puis la comparer avec celle obtenue par le logiciel Shrimp [127] développé par l'équipe Gamma d'Inria Rocquencourt. Ce programme utilise des courbes de Hilbert afin de déterminer un parcours et dans le même temps de numéroter les éléments du maillage.

Tout d'abord, nous présentons les maillages utilisés lors de l'expérimentation et lors de la comparaison. Les tests ont porté aussi bien sur des maillages isotropes que sur des maillages anisotropes, représentant un cube, une sphère ou une biellette. Ainsi, par exemple, les maillages anisotropes de forme cubique seront notés $cube_a1$, $cube_a2$, etc.

Le tableau 4.1, page suivante recense les différents cas tests utilisés par la suite, classés par taille, avec leurs spécificités.

Nom	Métrique	Nombre d'éléments	Nombre de nœuds	Spécificités du maillage
$cube_a3$	anisotrope	66 608	24 989	déraffiné au maximum
$sphere_a1$	anisotrope	690 100	115 541	autour d'un avion
$cube_a2$	anisotrope	2 245 692	375 347	autour d'une statue
$sphere_a2$	anisotrope	7 886 967	135 5873	
$cube_i1$	isotrope	8 459 419	141 3115	

TABLE 4.1 – Maillages utilisés comme cas tests.

4.6.1 Scalabilité

Avant de présenter les tests de scalabilité que nous avons menés, il nous faut au préalable exposer comment fonctionne le remailleur séquentiel MMG3D que nous utilisons.

Le processus de remaillage est décomposé en deux phases. La première, notée P_1, réalise des insertions et des suppressions de nœuds, alors que la deuxième, notée P_2 effectue essentiellement des bougés de nœuds, ce qui modifie peu la topologie du maillage.

Afin de mettre en évidence l'influence de la renumérotation sur chacune de ces phases, nous avons mesuré les temps pris par le remailleur dans deux cas test différents. Le premier consiste à utiliser le remailleur comme une boîte noire, en lui fournissant en entrée un maillage qui aura soit été numéroté arbitrairement par l'outil qui l'a créé, soit renuméroté au préalable avec notre méthode. Le deuxième cas tes découple les deux phases du remailleur, en réalisant une renumérotation du maillage avant chacune des deux phases.

4.6.1.1 Renumérotation avant remaillage

La figure 4.4, page suivante présente les résultats des tests de scalabilité lorsque seule une unique étape de renumérotation est réalisée avant l'appel du remailleur. Au vu de cette courbe, la mise en œuvre de notre algorithme semble effectivement linéaire vis-à-vis de la taille du problème.

La différence entre les temps de remaillage mesurés, illustrés en figure 4.5, page suivante, dépend de la taille de ces maillages. Pour certains, le temps de remaillage peut être divisé par trois demis, voire quasiment par deux dans le meilleur des cas.

4.6.1.2 Renumérotation avant chaque phase de remaillage

Nous allons découpler les deux phases du remailleur et procéder à une renumérotation avant chaque phase. Nous n'exposons pas la première renumérotation, puisqu'elle est identique à celle que nous venons de voir dans la sous-section précédente. Commençons par les temps de la phase P_1, illustrés en figure 4.6, page 67. Si nous comparons cette figure avec la figure 4.5, page suivante, alors nous pouvons dire que la phase P_1 représente entre le dixième et le cinquième du temps global du remaillage.

La scalabilité de la renumérotation, montrée en figure 4.7, page 67, est moins bonne que précédemment, lorsqu'elle était utilisée avant la phase P_1 du remailleur. La raison en est selon nous que les maillages manipulés sont plus grands que précédemment, puisque de nombreux nœuds ont été insérés lors de la phase P_1. La localité des données manipulées est donc moins bonne.

La figure 4.8, page 68 nous permet de dire que la renumérotation est plus efficace avant la phase P_2 du remailleur, même si les gains qu'elle peut apporter à la phase P_2 ne sont pas non

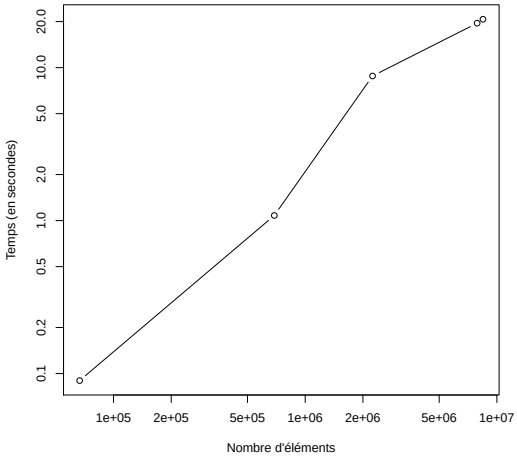


FIGURE 4.4 – Temps d'exécution de la renumérotation séquentielle avec SCOTCH sur des maillages de différentes tailles.

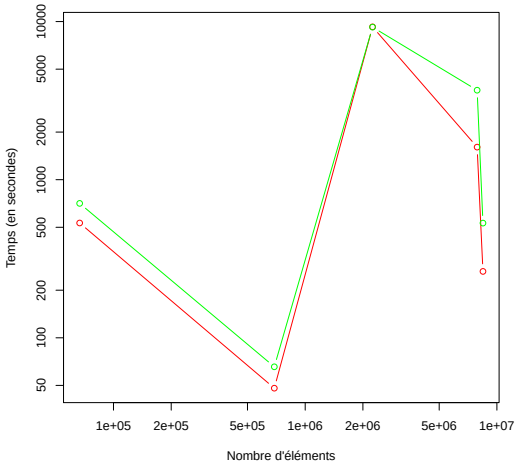


FIGURE 4.5 – Temps d'exécution du remailleur en fonction du nombre d'éléments après remaillage, sans renumérotation préalable (en vert) et avec renumérotation préalable au moyen de l'outil SCOTCH (en rouge).

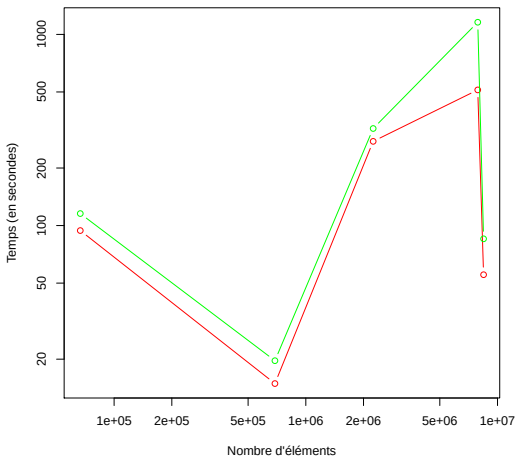


FIGURE 4.6 – Temps d'exécution de la phase P_1 du remailleur en fonction du nombre d'éléments obtenus après remaillage, sans renumérotation (en vert) et avec la renumérotation de SCOTCH (en rouge).

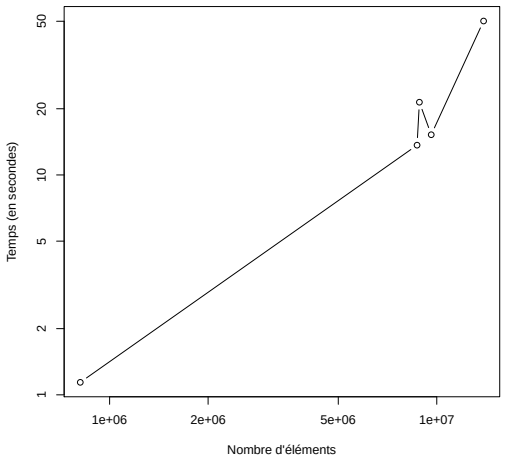


FIGURE 4.7 – Temps d'exécution de la renumérotation calculée entre les phases P_1 et P_2 du remailleur, en fonction du nombre d'éléments.

plus négligeables. Comme la phase P_2 modifie très peu la topologie du maillage, les avantages de disposer d'une bonne renumérotation au début de la phase P_2 seront préservés tout au long de cette dernière. Le gain de temps n'est pas visible lorsque nous effectuons une seule renumérotation avant le remaillage, parce que la phase P_1 ajoute et/ou supprime des nœuds de façon arbitraire, ce qui rend de moins en moins efficace notre la renumérotation initiale au fur et à mesure du remaillage.

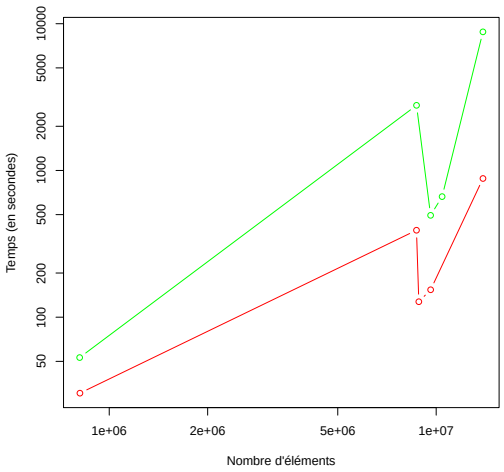


FIGURE 4.8 – Temps d'exécution de la phase P_2 du remailleur en fonction du nombre d'éléments obtenus après remaillage, sans renumérotation (en vert) et avec la renumérotation de SCOTCH (en rouge).

4.6.1.3 Comparaison des deux méthodes de renumérotation mises en œuvre

Comparons les deux méthodes de renumérotation décrites précédemment. De par les résultats présentés en figure 4.9, page ci-contre, nous pouvons dire, d'une part, que la deuxième renumérotation, indépendamment de la méthode, est plus coûteuse en temps, parce qu'entre temps la taille du maillage a évolué. Dans la moitié des cas, la renumérotation itérative met le même temps que la renumérotation dite « scratch-reorder », alors que dans l'autre moitié, elle s'exécute avec un tiers de temps en moins.

Le remaillage, illustré en figure 4.10, page suivante est plus rapide lorsque nous utilisons deux fois la renumérotation dite « scratch-reorder », alors que le gain de la renumérotation itérative par rapport à l'utilisation d'une seule renumérotation (courbe bleue) est presque nul. Néanmoins, sur un maillage, le temps de remaillage correspond aux deux cinquièmes du temps mis en utilisant qu'une seule renumérotation, mais reste toutefois plus long que si nous utilisons deux fois la technique « scractch-reorder ».

4.6.2 Comparaison avec SHRIMP

Plusieurs outils de renumérotation de maillages existent déjà. Nous allons comparer notre méthode avec le logiciel SHRIMP, développé par A. Loseille et F. Alauzet. Ce logiciel est basé sur le remplissage du maillage par des courbes de Hilbert, la renumérotation du maillage suivant le motif de la courbe calculée.

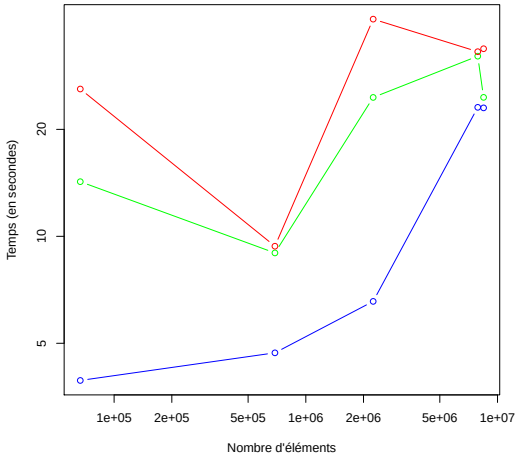


FIGURE 4.9 – Temps d'exécution total des renumérotations sur des maillages de différentes tailles, avec une seule renumérotaion avant la phase P_1 du remailleur (en bleu), avec deux renumérotations utilisant SCOTCH (en rouge) et avec deux renumérotations dont la deuxième est itérative (en vert).

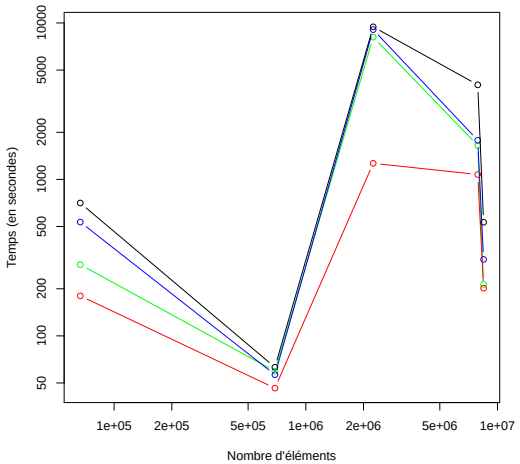


FIGURE 4.10 – Temps d'exécution du remailleur sur des maillages de différentes tailles, sans renumérotation (en noir), avec une seule renumérotaion avant la phase P_1 du remailleur (en bleu), avec deux renumérotations utilisant SCOTCH (en rouge) et avec deux renumérotations dont la deuxième est itérative (en vert).

En termes d'algorithmes, le nombre d'opérations effectuées par sommet semble à priori tendre en faveur de Shrimp, car Scotch s'appuie sur un ensemble de méthodes (multi-niveaux, bipartition récursive, etc) qui nécessitent de considérer de multiples fois de suite les mêmes sommets au cours du temps. Il est donc intéressant de disposer d'un étalonnage de nos méthodes avec celles de Shrimp, afin de mesurer la réalité de cet écart, et comment éventuellement le tourner en notre faveur, grâce au parallélisme.

Nous avons commencé par mesurer, sur notre jeu de test, les tailles des maillages considérés avant et après remaillage. Celles-ci sont donnés en figure 4.11. Elles vont nous permettre d'interpréter les résultats qui suivent.

Nous allons une fois de plus fournir un maillage renuméroté au remailleur, puis comparer les temps obtenus dans le cas de la renumérotation brute. Ensuite, nous séparerons les deux phases du remailleur. Nous appliquerons éventuellement une renumérotation entre ces deux phases, en plus de la renumérotation avant la phase P_1, afin de mesurer la dégradation de la performance induite par la phase P_1 de remaillage.

4.6.2.1 Renumérotation avant le remaillage

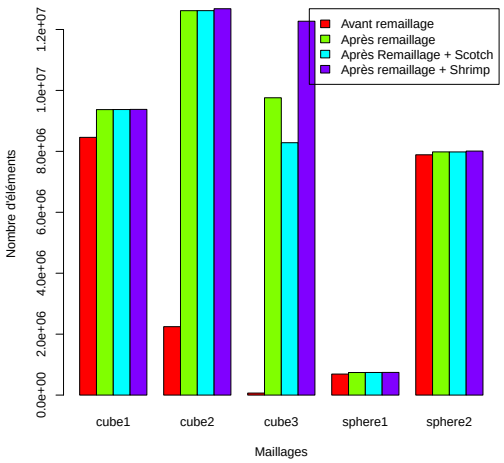


Figure 4.11 – Tailles des maillages avant et après remaillage, sur des maillages bruts, renumérotés avec Scotch et avec Shrimp

Nous pouvons comparer, grâce à la figure 4.12, page suivante, les temps de renumérotation obtenus avec notre méthode et avec Shrimp. Ce dernier est en moyenne deux fois plus rapide que Scotch sur ces maillages, ce qui est compatible avec nos considérations initiales sur le coût relatif des deux méthodes en termes d'accès aux données. Par ailleurs, les courbes de Hilbert dépendent de l'espace à parcourir et sont plus adaptées à des formes géométriques. Or, nos maillages sont soit des cubes, soit des sphères, ce qui explique également les différences de temps obtenus.

Comparons maintenant les temps de remaillage, illustrés en figure 4.13, page 72. Dans deux des cas, les renumérotations calculées par Shrimp et Scotch conduisent à des temps de remaillage équivalents. Sur un maillage, notre renumérotation est presque deux fois plus rapide. Cet écart peut s'expliquer par la figure 4.11, dans laquelle nous pouvons voir que le maillage

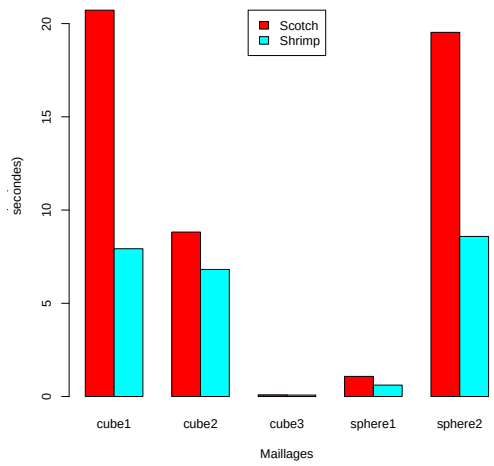


FIGURE 4.12 – Comparaison des temps d'exécution de notre renumérotation avec celle de SHRIMP sur différents maillages.

renuméroté avec SHRIMP possède pratiquement le double d'éléments par rapport au maillage remaillé préalablement renuméroté avec SCOTCH. Ces écarts très importants sont induits par la triangulation de Delaunay, qui dépend très fortement de l'ordre de parcours des éléments, et donc de la renumérotation. Pour tous les autres maillages testés, SHRIMP est plus rapide, jusqu'à environ $1,5$ fois dans le meilleur des cas.

4.6.2.2 Renumérotation avant chaque phase de remaillage

La figure 4.14, page suivante présente les tailles des différents maillages mesurées avant et après la phase P_1 du remaillage.

La renumérotation que nous appliquons avant la phase P_1 du remaillage, dont les résultats sont présentés en figure 4.12, est équivalente ce ce que nous avons déjà mis en œuvre auparavant, c'est-à-dire à renuméroter le maillage brut avant le début du remaillage. Les temps de la phase P_1 de remaillage sont donnés en figure 4.15, page 73. Pour l'un des cinq maillages, la renumérotation avec SCOTCH permet d'obtenir un meilleur temps, égal à trois cinquièmes de celui de SHRIMP, alors que dans les quatre autres cas, SHRIMP s'avère meilleur, pouvant générer des remaillages jusqu'à deux fois plus rapides.

La figure 4.14, page suivante montre les tailles des maillages de test, mesurées avant et après la phase P_2 du remaillage. Nous remarquons que les tailles varient peu par rapport à la figure 4.14, page suivante, qui indique les tailles obtenues après la phase P_1. En effet, peu d'insertions de nœuds sont effectuées lors de la phase P_2. Nous ne savons pas expliquer la différence de taille aussi importante observée sur le maillage « cube_a3 » entre les phases P_1 et P_2.

Sur la renumérotation entre les deux phases P_1 et P_2 du remailleur, SHRIMP est plus rapide, comme le montre la figure 4.17, page 74.

Si nous nous référons à la figure 4.18, page 74, il est difficile de savoir si la renumérotation avec SHRIMP permet de mieux accélérer la phase P_2 du remailleur, que celle utilisée avec SCOTCH.

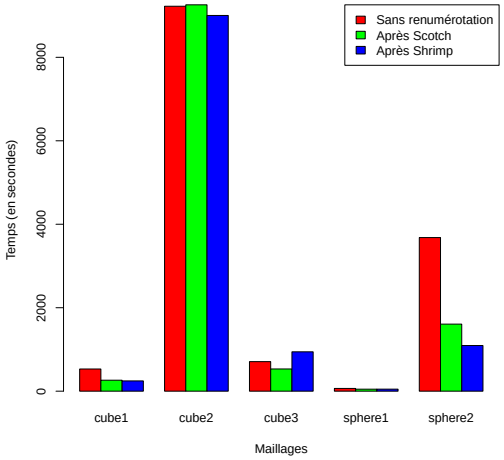


FIGURE 4.13 – Comparaison des temps d'exécution du remailleur sur des maillages bruts, renumérotés avec SCOTCH et avec SHRIMP.

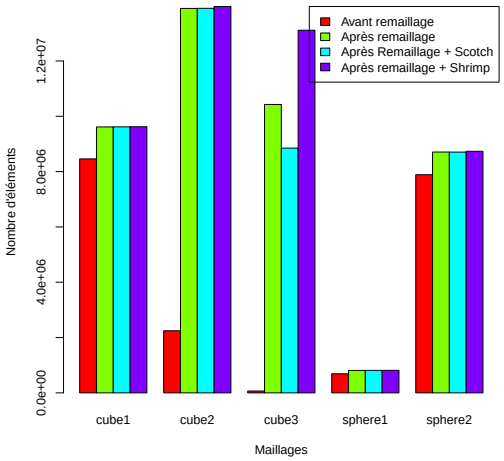


FIGURE 4.14 – Tailles des maillages mesurées avant et après la phase P_1 du remaillage, sur des maillages bruts, renumérotés avec SCOTCH et avec SHRIMP.

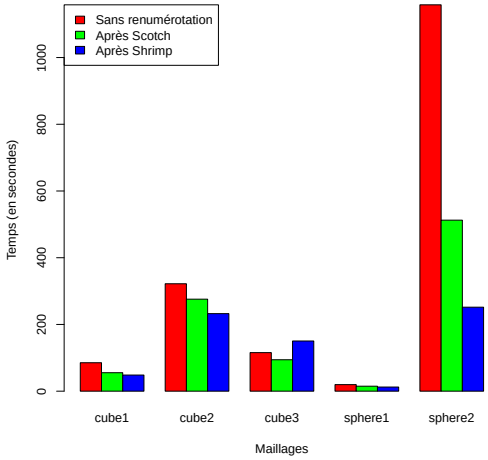


FIGURE 4.15 – Temps d'exécution de la phase P_1 du remailleur obtenus à partir de maillages bruts, renumérotés avec SCOTCH ou bien avec SHRIMP.

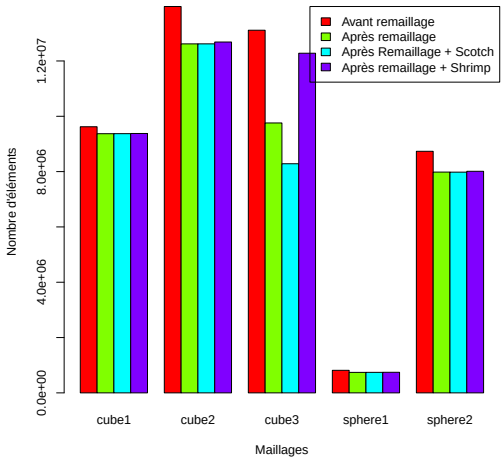


FIGURE 4.16 – Tailles des maillages obtenues avant et après la phase P_2 du remaillage, sur des maillages bruts, renumérotés avec SCOTCH et avec SHRIMP.

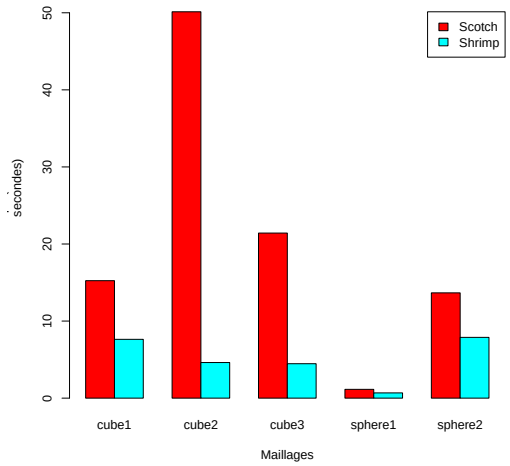


FIGURE 4.17 – Temps d'exécution de la renumérotation intermédiaire effectuée entre les deux phases P_1 et P_2 de remaillage, réalisée avec notre méthode et avec SHRIMP sur différents maillages.

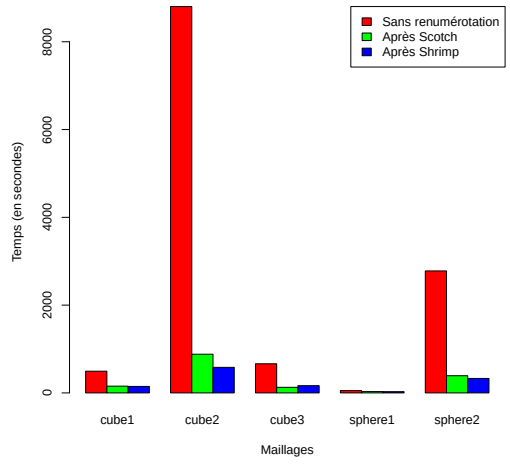


FIGURE 4.18 – Comparaison des temps d'exécution de la phase P_2 du remailleur sur des maillages bruts, renumérotés avec SCOTCH et avec SHRIMP.

4.7 Conclusion

Dans ce chapitre, nous avons mis en évidence l'influence de la renumérotation des maillages sur la vitesse d'exécution des remailleurs. Cette influence découle de l'augmentation des effets de cache induits par une numérotation adaptée.

La méthode que nous avons proposée, basée sur des techniques de partitionnement, est globalement moins rapide que la méthode mise en œuvre au sein du logiciel séquentiel Shrimp, et les renumérotations qu'elles produit ne sont pas de meilleure qualité que celles de ce dernier en termes d'accélération du remailleur séquentiel.

Cependant, l'avantage de nos méthodes est qu'elles peuvent être facilement parallélisées sur une architecture à mémoire distribuée, tant à grain fin qu'à gros grain, ce qui est plus difficile pour les méthodes basées sur les courbes de Hilbert. Nous regrettons, faute de temps suffisant, de ne pas avoir pu exposer les résultats relatifs à ces deux types de parallélisation. La parallélisation à grain fin aurait très probablement permis d'obtenir des temps de renumérotation meilleurs que ceux de Shrimp.

Chapitre 5

Remaillage parallèle

Sommaire

5.1 Problématique du remaillage parallèle

Les maillages que nous souhaitons remailler sont de plus en plus complexes et de tailles de plus en plus grandes. Leur remaillage ne peut donc être réalisé en séquentiel, faute de taille mémoire suffisante au sein d'un unique ordinateur. Le temps de remaillage serait également déraisonnable. Il est par conséquent nécessaire de recourir au parallélisme à mémoire distribuée afin de calculer ces remaillages.

La question du remaillage parallèle a été abordée selon deux approches différentes.

La première consiste à créer des algorithmes parallèles de remaillage [16, 17, 29, 38, 71, 74, 84, 93]. Le remaillage parallèle ne peut être résumé au remaillage des différents sous-domaines indépendamment les uns des autres, puisque les frontières situées à l'interface entre les sous-domaines portés par les différents processeurs doivent également être remaillées. Les différentes modifications effectuées sur une frontière doivent être réalisées conjointement par les processeurs.

Prenons l'exemple d'un élément situé à la frontière entre plusieurs sous-domaines, représenté en gris sur la figure 5.1. Afin qu'un nœud soit inséré, les différents processeurs partageant l'interface doivent mutuellement s'informer sur la modification ayant été appliquée, et par quel processeur. Dans notre exemple, un processeur a besoin des informations des trois autres. De plus, les techniques de remaillage local doivent être mises en œuvre uniquement par un processeur, ce qui nécessite des communications supplémentaires pour savoir quel processeur va modifier un élément donné, en accord avec les processeurs voisins. Si de tels mécanismes de synchronisation ne sont pas mis en place, la coupe entre les sous-domaines du maillage remaillé pourrait soit être erronée, dans le cas où plusieurs processeurs auraient agi sans se concerter, soit appareaître inchangée, dans le cas où aucun processeur ne la prendrait en charge.

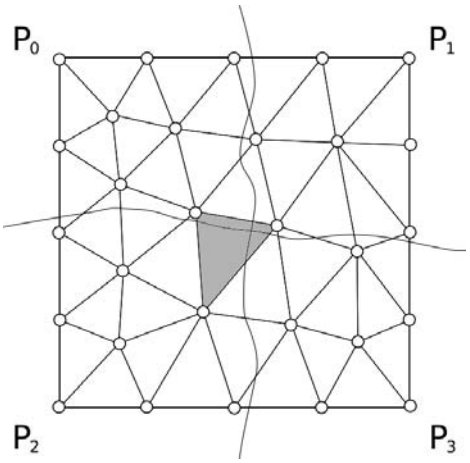


FIGURE 5.1 – Maillage distribué sur quatre processeurs, formé de triangles et de nœuds (cercles). Un nœud doit être inséré dans l'élément en gris.

La deuxième famille de remaillage parallèle [19, 34, 44, 45, 49, 92, 101], s'appuie sur des algorithmes de remaillage séquentiel pour obtenir le remaillage parallèle. Nous allons nous intéresser de façon plus détaillée à cette famille dans la suite de ce chapitre.

5.2 Algorithme de remaillage parallèle

Nous souhaitons généraliser cette dernière famille de remaillage, indépendamment de la taille du maillage et du nombre de processeurs utilisés. Autrement dit, la taille d'un sous-domaine peut excéder la taille maximale d'un maillage devant être fourni au remailleur séquentiel.

L'algorithme que nous utilisons est un processus itératif s'appliquant à des maillages distribués. Il extrait des zones à remailler, qui seront adaptées par un algorithme séquentiel. Le but de notre méthode est de pouvoir extraire et réintégrer efficacement des zones du maillage distribué tout en gardant un maillage distribué conforme et d'aussi bonne qualité que s'il avait été remaillé en séquentiel.

L'objectif de ce travail de recherche est de concevoir une méthode aussi performante qu'en séquentiel en termes de qualité, et également fortement scalable, afin de pouvoir traiter des maillages de plus d'un milliard d'éléments. La seule contrainte que nous imposons au remailleur séquentiel est de pouvoir lui spécifier un ensemble d'éléments qui ne seront pas modifiés. Cette condition est nécessaire pour que nous puissions réinsérer les zones remaillées dans le maillage distribué, comme nous le verrons en section 5.2.5, page 88. Les éléments à ne pas modifier seront en effet les éléments de la peau des zones fournies au remailleur.

5.2.1 Processus itératif à deux boucles

Le processus de remaillage est constitué de deux boucles imbriquées. La boucle dite « externe » exécute de facon itérative le processus de remaillage de l'ensemble du graphe distribué, tant que la qualité des éléments n'est pas satisfaisante. La deuxième boucle, dite « interne », s'applique à présenter au remailleur séquentiel l'ensemble des éléments marqués à remailler par la boucle externe. Nous allons maintenant voir en détails ces deux boucles.

5.2.1.1 Boucle externe

Afin de pouvoir opérer, notre algorithme de remaillage parallèle doit disposer des informations nécessaires au calcul de la qualité des éléments existants, afin de déterminer quels éléments doivent être remaillés. Il aura besoin pour cela des coordonnées des nœuds, qui seront fournies sous la forme d'une valeur associée aux entités nœuds du maillage distribué. Il sera également mécessaire de lui fournir une fonction d'évaluation de la qualité des éléments, indiquant si un élément doit être remaillé ou pas. Dans le cadre de cette thèse, notre fonction d'évaluation sera une métrique permettant de déterminer la longueur unité en chacun des points et dans chacune des dimensions du maillage. Tout élément dont les longueurs des arêtes s'écarteront trop de cette longueur idéale devront être remaillés. L'avantage d'utiliser une métrique comme évaluateur est qu'elle permet de prendre en compte l'anisotropie du remaillage.

Chaque itération de la boucle externe débute par le calcul en parallèle de la qualité de chaque élément, grâce aux informations ci-dessus. Les éléments dont la qualité n'est pas jugée satisfaisante sont alors marqués, au moyen d'une variable drapeau porté par les entités éléments. Le graphe ainsi étiqueté est fourni à la boucle interne, qui effectue le remaillage. Une fois cette étape de remaillage terminée, on effectue une nouvelle itération de la boucle externe, afin de tester la qualité du nouveau maillage distribué.

Le critère d'arrêt de la boucle externe est basé sur le ratio entre le nombre d'éléments marqués à remailler de l'itération précédente et celui de l'itération courante [9]. Ce ratio est comparé à un

9. Dans le cas de la première itération, la valeur correspondant à l'itération précédente est considérée comme étant égale à 0.

intervalle centré sur 1. Si le ratio est compris dans cet intervalle, autrement dit si le nombre d'éléments à remailler ne varie pas suffisamment d'une itération à l'autre, alors le processus itératif s'arrête.

On pourrait considérer que le remaillage ne devrait s'arrêter que lorsqu'il ne reste aucun élément jugé de mauvaise qualité. Cela n'est malheureusement pas toujours possible. Par exemple, il peut arriver que des éléments du bord du maillage soient jugés de mauvaise qualité, parce que très aplatis, alors que le remailleur est pour sa part incapable de les remailler, faute de savoir comment les subdiviser. En effet, subdiviser un élément de bord nécessite de savoir à quel endroit placer de nouveaux sommets sur la peau, ce qui nécessite de connaitre de façon analytique la géométrie de cette dernière. La version actuelle de l'outil MMG3D n'est encore pas capable de cela, ce qui a nécessité la mise en place du critère d'arrêt décrit ci-dessus.

5.2.1.2 Boucle interne

Pour effectuer le remaillage des éléments marqués à remailler par la boucle externe, nous effectuons la boucle interne décrite schématiquement en figure 5.2, page suivante.

Les troisième et cinquième étapes sont essentielles, mais ne constituent pas le cœur du problème, puisqu'elles effectuent simplement l'insertion et l'extraction de sous-maillages au sein du maillage distribué. Ces actions sont techniquement délicates, mais peu complexes.

La principale difficulté de notre algorithme se situe dans la deuxième étape, qui consiste à identifier les zones disjointes à fournir aux différentes instances du remailleur séquentiel. Nous avons élaboré cinq algorithmes différents pour résoudre ce problème, que nous comparerons en section 5.2.3, page 83.

Lors de la quatrième étape, chaque zone est remaillée par des instances indépendantes du remailleur séquentiel, qui s'exécuteront de façon indépendante sur les différents processeurs.

Nous allons maintenant détailler dans l'ordre les cinq étapes du processus, en nous appuyant pour cela sur les graphes et graphes enrichis distribués définis dans le chapitre 3, page 31.

5.2.2 Marquage des éléments à remailler

L'étape 1 a pour rôle de mettre à jour les drapeaux de remaillage. L'étape $1a$, au cours de la première itération, consiste juste à prendre en compte les drapeaux fournis par la boucle externe.

Lors des itérations suivantes, correspondant à l'étape $1b$, le marquage des éléments de l'itération précédente est modifié, en supprimant le marquage des éléments que le remailleur a eu la possibilité de remailler. Les éléments concernés sont donc ceux situés à l'intérieur de zones fournies au remailleur. Les éléments de la peau de ces zones restent pour leur part inchangés, à l'exception de ceux appartenant au bord du maillage distribué. En effet, les éléments de peau connectés au reste du maillage serviront à réaliser la « suture » avec celui-ci lors de la réintégration de la zone remaillée au sein du maillage distribué (voir section 5.2.5, page 88). Bien qu'ils soient fournis au remailleur, ce dernier n'a pas le droit de les modifier.

Qui plus est, si, lors du remaillage d'une zone, le remailleur séquentiel ne peut réaliser son travail (par exemple par manque d'espace mémoire), les éléments en question resteront marqués. Ils pourront ainsi être considérés à nouveau, au moins partiellement, lors du calcul du remaillage d'une zone connexe de celle ayant échoué. Notre algorithme possède ainsi une certaine robustesse vis-à-vis des remailleurs séquentiels qu'il utilise.

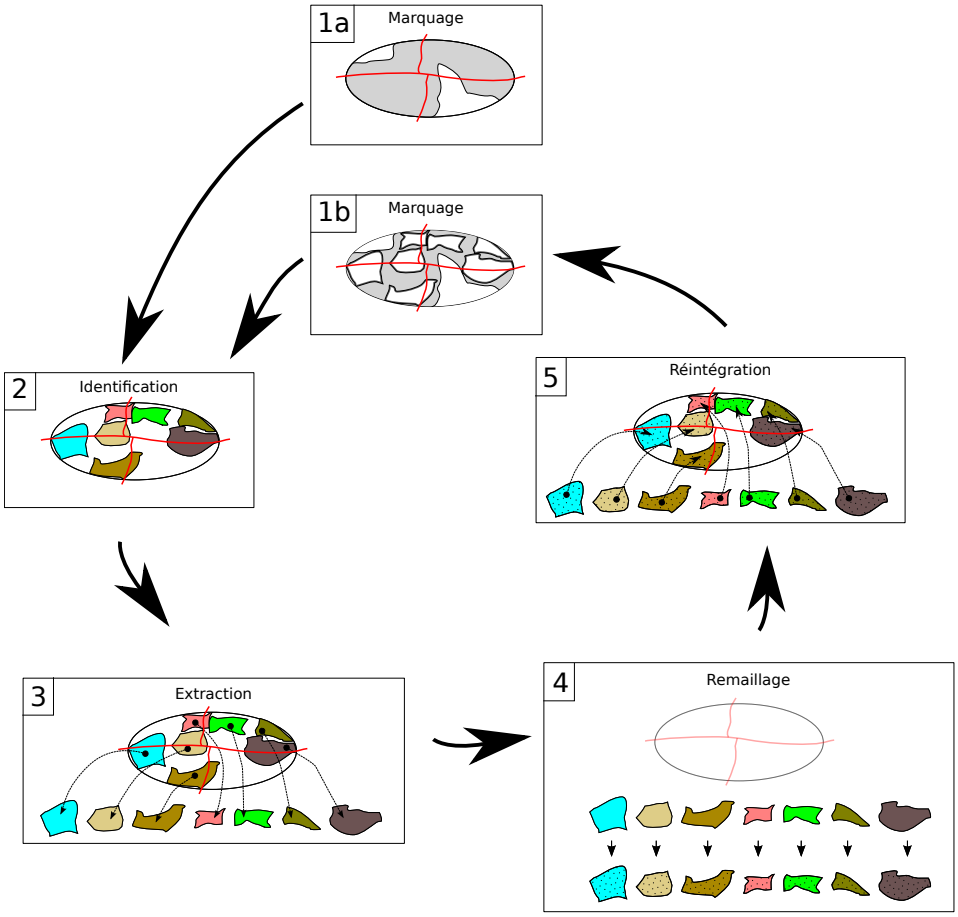


FIGURE 5.2 – Schéma de la boucle interne du processus de remaillage, composée de cinq étapes. La première étape gère le marquage des éléments à remailler (zone grisée) ; dans l'étape 1a, ce marquage est directement fourni par la boucle externe, alors que dans l'étape 1b, on enlève le marquage des éléments déjà vus par le remailleur. La deuxième étape est l'identification de zones indépendantes au sein de la partie à remailler. La troisième extrait ces zones et les répartit sur les processeurs. À la quatrième étape, les zones sont remaillées indépendamment les unes des autres. La cinquième étape réintègre les zones remaillées dans le maillage distribué. Tant que le maillage n'est pas considéré comme totalement remaillé, le processus est itéré sur les portions restantes.

5.2.3 Identification des zones à remailler

Cette étape vise à identifier un ensemble de zones indépendantes, pouvant être remaillées simultanément sans générer de conflits, en optimisant l'ensemble des critères suivants :

— maximiser la taille des zones, tout en ne dépassant pas cependant une taille maximale fixée. Cette taille peut correspondre par exemple à la taille à partir de laquelle la vitesse d'exécution du remailleur séquentiel diminue, du fait des effets de va-et-vient (« *swapping* ») en mémoire centrale ;
— minimiser la taille de la peau des zones, afin de contraindre au minimum le remailleur ;
— maximiser le nombre d'éléments envoyés au remailleur, par rapport aux éléments marqués à remailler.

Le deuxième critère, relatif à la minimisation de la taille de la peau, et dont nous justifions la nécessité en section 5.2.5, page 88, nous amène à introduire la notion de quotient isopérimétrique [65], qui permet de quantifier la surface par rapport au volume.

Définition 44 (Quotient isopérimétrique)
Le quotient isopérimétrique q est un facteur qui s'applique à une surface mesurable ayant une frontière rectifiable, c'est-à-dire de longueur finie et pour laquelle le mot « périmètre » a un sens. En 3D, il vaut : $q = 36\pi\frac{V^2}{S^3}$, où V est le volume et S est la surface.

Le quotient isopérimétrique est minoré par 0 et majoré par 1, cette valeur étant atteinte dans le cas de la sphère, qui minimise sa surface par rapport au volume qu'elle contient. Notre objectif est donc de fournir au remailleur séquentiel des zones d'une forme la plus sphérique possible, afin de minimiser le nombre d'éléments de peau, non remaillables, par rapport au nombre d'éléments contenus dans le volume fourni.

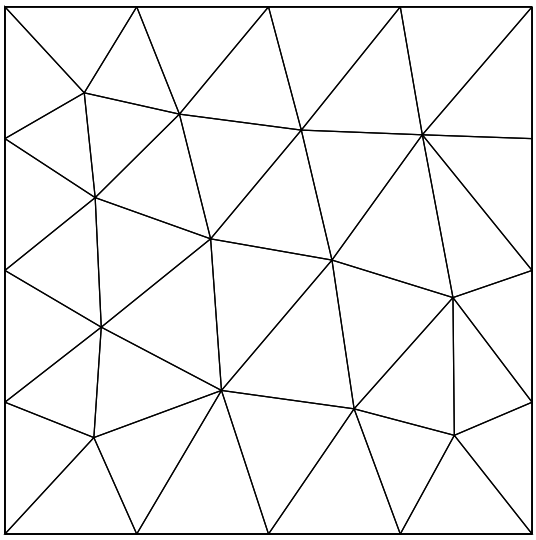
Les zones à fournir peuvent regrouper plusieurs parties à remailler indépendantes, ou bien être des portions d'un groupe connexe de sommets à remailler trop grand par rapport à la taille maximale de zone fixée. Il s'agit donc pour nous d'identifier des zones les plus connexes possibles, les plus sphériques possibles, et bornées par une taille maximale. Nous avons envisagé plusieurs méthodes afin de résoudre ce problème.

Tout d'abord, expliquons sur quelles structures de données opèrent ces différents algorithmes. Dans le graphe enrichi, les sommets et les relations dont nous avons besoin sont les éléments et les relations entre éléments, autrement dit le sous-graphe induit des éléments, tel qu'illustré en figure 5.3, page suivante. Ce graphe n'est plus considéré comme un graphe enrichi, puisqu'il ne contient qu'un seul type d'entité [10]. Par ailleurs, dans un premier temps, nous allons nous intéresser dans ce graphe aux éléments à remailler. Pour cela, nous extrayons le sous-graphe contenant uniquement les sommets marqués comme étant à remailler. Ce graphe correspond au sous-graphe induit des éléments à remailler, dont un exemple est représenté en figure 5.4, page suivante. Le graphe induit, qui possède par nature moins de sommets que le graphe original, possède donc une numérotation différente. Il faut donc maintenir une correspondance entre les numéros de sommets du graphe induit et les numéros de ces sommets dans le graphe original, qui sera utilisée au moment de construire les sous-maillages correspondant aux zones à remailler.

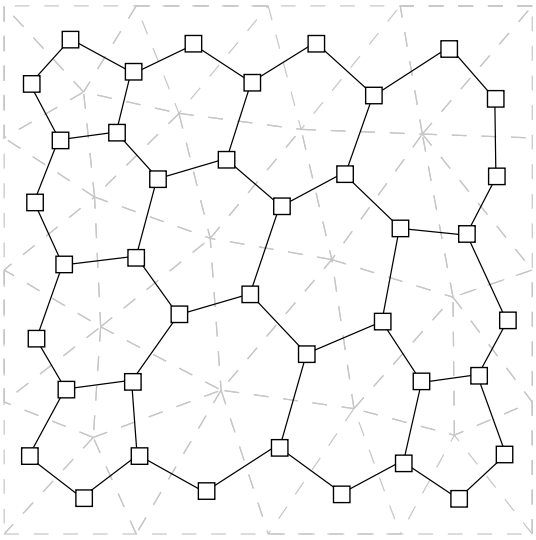
Nous avons successivement développé plusieurs algorithmes parallèles d'identification de zones. Chacun de ces algorithmes vise à répondre au problème de déterminer en parallèle les zones qui seront fournies aux différentes instances du remailleur séquentiel. Ces algorithmes sont les suivants :

1. contraction de graphe ;

10. Le cas spécifique où il y aurait plusieurs sous-entités est discuté en section 5.2.5, page 88.

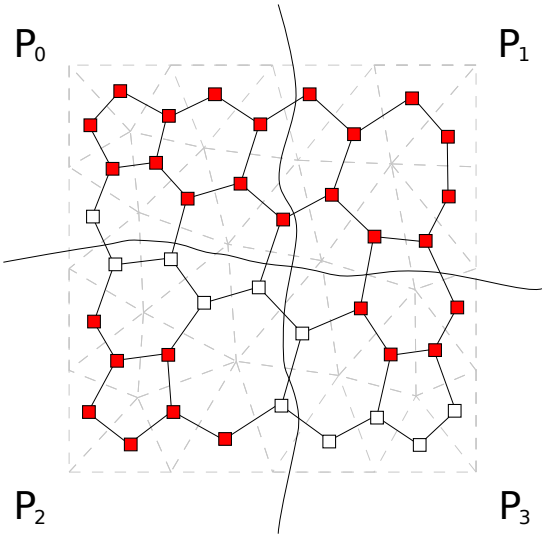


(a) Maillage 2D constitué de triangles.

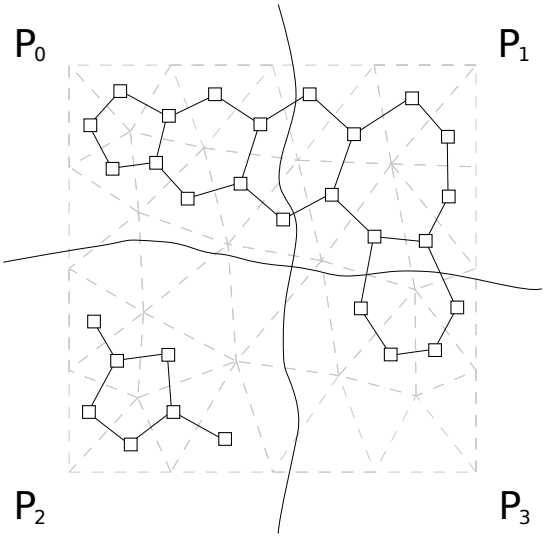


(b) Graphe des éléments (carrés et traits pleins) correspondant au maillage de gauche (en tirets).

FIGURE 5.3 – Exemple de maillage (a) et graphe des éléments correspondant (b).



(a) Graphe enrichi dont certains éléments sont à remailler (en rouge), distribué sur quatre processeurs.



(b) Sous-graphe induit des éléments à remailler correspondant au graphe de gauche.

FIGURE 5.4 – Exemple de sous-graphe induit des éléments à remailler.

2. expansion de graine sur le graphe ;
3. expansion de graine sur le graphe enrichi ;
4. contraction de graphe avec expansion de graine sur le graphe enrichi ;
5. partitionnement de graphe.

5.2.3.1 Contraction de graphe

Notre but étant de construire rapidement et en parallèle des ensembles compacts de sommets, il nous a semblé naturel d'utiliser à cet effet l'algorithme multi-niveaux de contraction de graphe utilisé par les partitionneurs. Outre que cet algorithme est déjà disponible et peut être facilement adapté à nos besoins, il répond en théorie aux critères que nous avons défini : il est fortement parallèle, et agglomère des ensembles de sommets voisins, donc potentiellement compacts.

Nous utilisons cet algorithme de la façon suivante. Tout d'abord, nous contractons le graphe induit sur $log(n)$ niveaux, où n est la taille de zone souhaitée. Chaque sommet du graphe le plus contracté est alors identifié par un numéro de zone unique représentant la « couleur » de chaque zone. Lors de la phase d'expansion, la couleur unique de chaque sommet contracté est diffusée aux sommets plus fins dont il est issu, et ainsi de suite de proche en proche jusqu'à disposer d'une coloration en zones de l'ensemble des sommets du graphe original. Ce processus est illustré en figure 5.5.

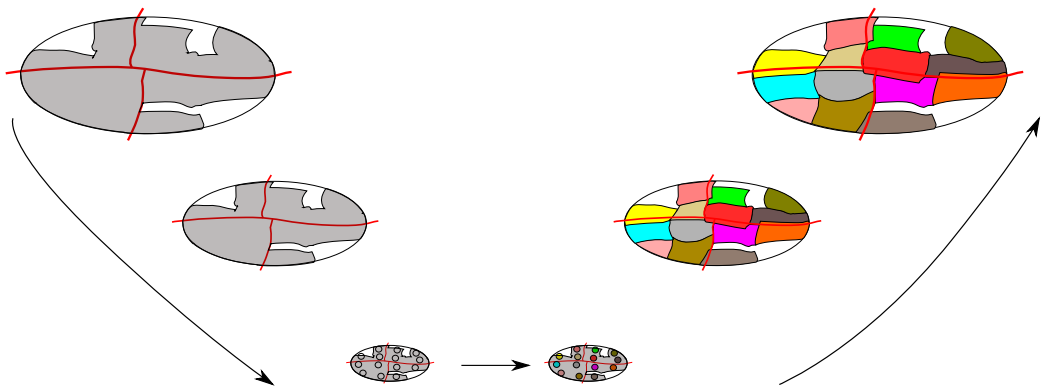


FIGURE 5.5 – Contraction du graphe induit distribué (en gris) sur 4 processeurs.

Cette méthode, bien que rapide, ne donne pas en pratique des zones de forme suffisamment compactes et sphériques. La cause en est le processus aléatoire d'appariement entre sommets voisins, qui peut conduire à choisir des sommets formant une chaîne plutôt qu'une boule compacte.

5.2.3.2 Expansion de graines sur le graphe

Afin d'obtenir des zones plus compactes, nous avons alors pensé nous appuyer explicitement sur la topologie du graphe original, en utilisant un processus de diffusion, illustré en figure 5.6, page suivante. Nous commençons par choisir, sur chaque sous-domaine, un élément à remailler de plus grand poids (voir sous-section 5.5.1.2, page 97). Nous attribuons une graine à ce sommet, chaque graine possédant une couleur unique. Nous procédons ensuite à une expansion de la zone issue de chacune des graines, au moyen d'un parcours en largeur des sommets du graphe induit des éléments à remailler. L'algorithme que nous avons conçu s'assure qu'un sommet ne puisse

pas être colorié simultanément de plusieurs couleurs différentes : si, lors d'une étape de diffusion, un sommet demande à être colorié d'une couleur différente par plusieurs de ses voisins, seule l'une des requêtes est acceptée. La croissance des boules est arrêtée dès que le poids des sommets contenus dans cette boule dépasse la valeur souhaitée.

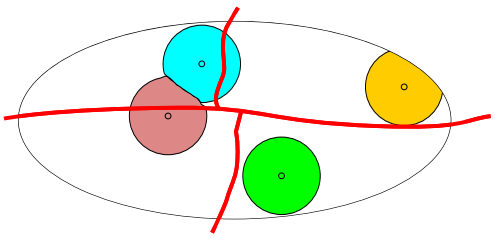


FIGURE 5.6 – Expansion d'une graine depuis les sous-domaines portés par chaque processeur, pour un maillage distribué sur quatre processeurs.

Contrairement à l'intuition que nous pourrions en avoir, le quotient isopérimétrique des zones construites par cette méthode n'est pas assez élevé, comme nous le verrons en section 5.6, page 100. La raison en est que la topologie du graphe des éléments ne définit pas une relation de voisinage topologiquement pertinente. En effet, dans ce graphe, il existe une arête entre deux éléments seulement si ces éléments sont voisins par une face, comme illustré en figure 5.7.

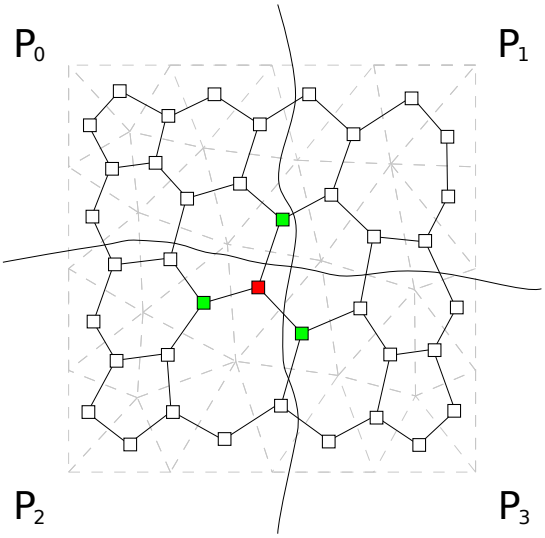


FIGURE 5.7 – Expansion d'une graine (en rouge) sur les éléments voisins (en vert) dans le graphe des éléments.

Afin d'avoir des zones dont le quotient isopérimétrique soit plus important, il faudrait que le voisinage des éléments soit défini par le fait qu'ils partagent au moins un nœud, et non pas seulement une face. Le problème d'un tel graphe serait sa taille trop importante en termes de nombre d'arêtes, puisque ce graphe serait constitué de cliques entre tous les éléments partageant un nœud donné. Cette taille dépasserait largement, en mémoire, celle déjà conséquente du graphe enrichi.

Puisqu'il n'est pas pertinent de construire un tel graphe, il semble en revanche plus pertinent

d'adapter notre algorithme de diffusion afin qu'il s'applique directement au graphe enrichi.

5.2.3.3 Expansion de graines sur le graphe enrichi

Comme nous venons de le dire, cette méthode s'apparente très fortement à la précédente, à la différence que le graphe utilisé n'est plus le graphe des éléments mais le graphe enrichi. La propagation d'une graine sur un élément revient à énumérer les nœuds voisins de cet élément puis, pour chacun d'entre eux, de considérer tous ses éléments voisins, comme illustré en figure 5.8.

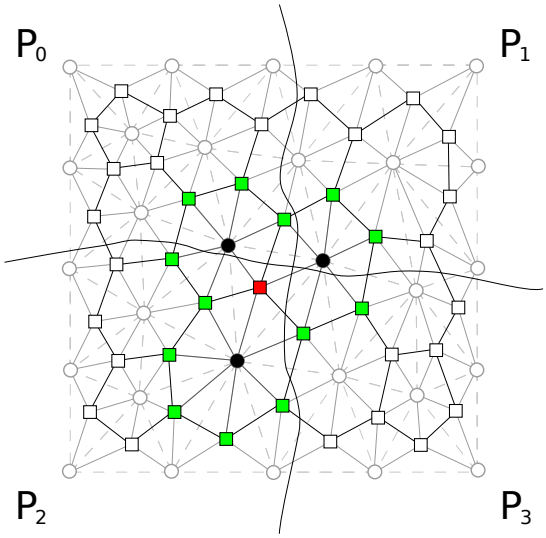


FIGURE 5.8 – Expansion d'une graine (en rouge) sur les éléments (en vert) via les nœuds voisins (en noir) dans le graphe enrichi.

L'identification des zones est plus simple que dans la première méthode, étant donné que l'expansion des graines est calculée directement sur le graphe enrichi, sans avoir besoin de construire le graphe des éléments.

Cependant, si le coefficient isopérimétrique des zones construites est satisfaisant, il persiste un problème quant au choix des graines. En ne sélectionnant qu'une graine par sous-domaine, c'est-à-dire par processeur, le nombre de zones à remailler à chaque itération est limité par le nombre de processeurs. Or, le choix des graines peut conduire à des zones de tailles très différentes, selon qu'une graine est encerclée par d'autres graines qui bloqueront la croissance de sa zone ou pas. Afin de remédier à ce possible déséquilibre de la charge, le nombre de zones doit être supérieur au nombre de processeurs.

5.2.3.4 Contraction de graphe et expansion de graines sur le graphe enrichi

Les deux dernières méthodes décrites précédemment sont limitées par le nombre de zones identifiées à chaque itération. Nous proposons par conséquent une quatrième méthode, qui combine les avantages de la première et de la troisième quant au nombre et à la forme des zones.

Dans un premier temps, nous utilisons notre premier algorithme afin de construire un ensemble de zones par contraction du graphe induit (voir section 5.2.3.1, page 85).

Cependant, les zones ainsi créés ne sont plus considérées pour elles-mêmes. Elles représentent maintenant un terroir au sein duquel une graine sera plantée, à raison d'une par couleur de

terroir. Pour éviter que plusieurs processeurs ne plantent chacun une graine sur un terroir dont ils possèdent une fraction, nous avons mis en œuvre la règle suivante : seul le processeur de numéro le plus petit en possession de sommets de la zone de couleur considérée aura le droit de planter la graine pour cette couleur.

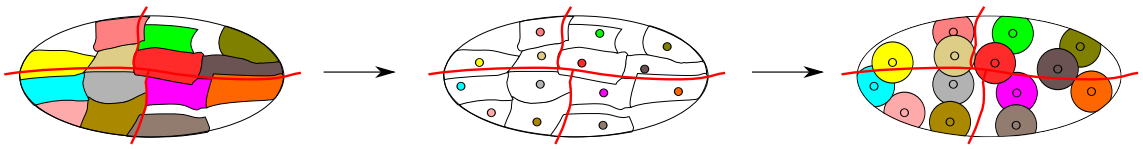


FIGURE 5.9 – Expansion d'une graine par zone identifiée lors de la contraction, pour un maillage distribué sur 4 processeurs.

5.2.3.5 Partitionnement de graphe

Dans la précédente méthode, le graphe induit des éléments à remailler est utilisé aussi bien pour la contraction du graphe que pour l'expansion des graines, ce qui revient à appliquer successivement deux algorithmes sur une même structure de données. Par conséquent, il nous a semblé intéressant de fusionner ces deux algorithmes en un seul.

Une méthode permettant de traiter une seule fois le graphe induit tout en gardant les deux critères que nous nous sommes fixés est de partitionner le graphe des éléments à remailler afin d'en extraire des zones plus lisses que dans le cadre de la contraction de graphe. En effet, le partitionnement, contrairement à la contraction de graphe, minimise la coupe et par conséquent affine la peau des zones à remailler. Les algorithmes de partitionnement multi-niveaux sont en réalité une contraction suivie d'un raffinement. Ainsi, l'algorithme de partitionnement est plus coûteux que l'algorithme de contraction mais il l'est moins que les algorithmes de contraction et d'expansion réunis.

5.2.4 Extraction des zones à remailler

Après que les zones à remailler ont été identifiées, chaque élément à remailler possède un numéro caractérisant la zone à laquelle appartient cet élément, comme illustré en figure 5.10, page ci-contre. Le but de l'extraction est de calculer un graphe enrichi centralisé pour chaque zone identifiée à l'étape précédente. Tout d'abord, les sommets liés aux élément d'une zone vont également appartenir à cette zone. Un sommet peut donc appartenir simultanément à plusieurs zones, comme nous pouvons le voir en figure 5.11, page suivante. Les informations relatives à ces sommets (topologie, données associées) sont dupliquées sur chacun des graphes enrichis induits, comme le montre la figure 5.12, page 90. Ces graphes enrichis induits possèdent une numérotation différente de celle du graphe enrichi original. Afin de faciliter la réintégration ultérieure des zones remaillées, il est donc nécessaire de créer une correspondance entre les numéros de sommets de chacun des graphes induits et ceux des sommets du graphe original.

5.2.5 Remaillage séquentiel

Dans le cas d'un maillage hybride comportant différents types de mailles (par exemple, des quadrangles et tétraèdres), nous aurons toujours une seule entité « élément », mais autant de sous-entités que de types de mailles au sein du graphe enrichi (deux sous-entités, pour l'exemple

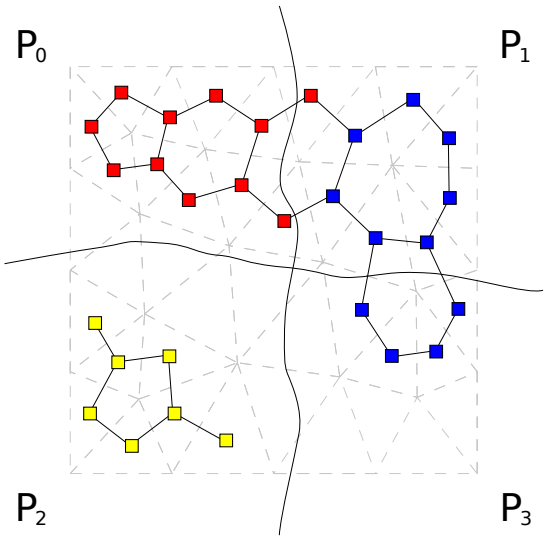


FIGURE 5.10 – Identification de trois zones (en rouge, en bleu et en jaune) des éléments à remailler.

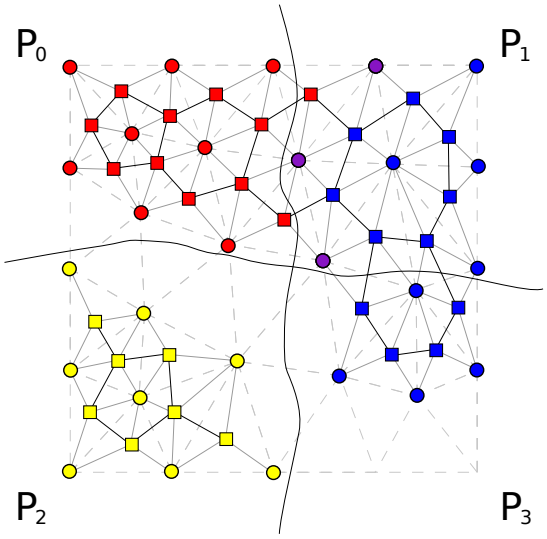


FIGURE 5.11 – Propagation aux nœuds de la couleur de zone à remailler (jaune, bleu et rouge). Les nœuds en violet appartiennent aux zones bleu et rouge.

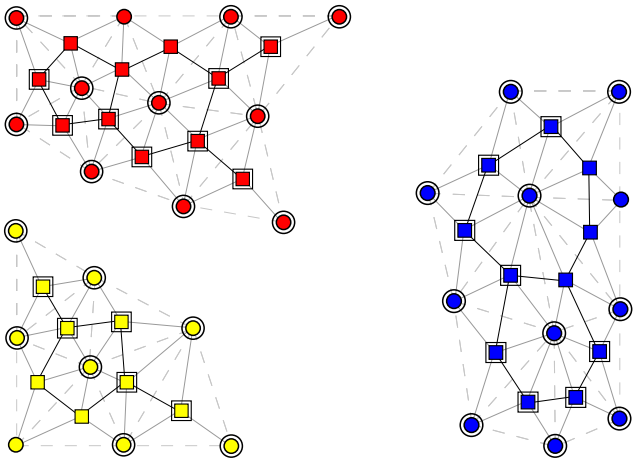


FIGURE 5.12 – Graphes enrichis centralisés (en rouge, bleu et jaune) induits par la partition des éléments du graphe enrichi distribué. Les éléments et nœuds qui ne doivent pas être modifiés par le remailleur sont respectivement représentés par un double carré et un double cercle.

des quadrangles et des tétraèdres). Chacune des zones identifiées pourrait donc être représentée par un seul graphe enrichi induit, composé des différents types de mailles, si le remailleur séquentiel peut les remailler. Sinon, il peut être possible de générer autant de graphes enrichis induits que de sous-entités d'éléments, par zone identifiée.

Chaque zone identifiée est extraite sous la forme d'un sous-graphe enrichi induit, puis remaillée au moyen d'un remailleur séquentiel. Pour que les zones puissent être réintégrées dans le maillage distribué, des éléments doivent servir de tampons entre le maillage distribué et les zones remaillés. Cela se traduit par le fait que les éléments de la peau des zones à remailler, hormis le bord du maillage distribué, ne soient pas modifiés par le remailleur. Ceci concerne tant les éléments de peau eux-mêmes, que tous les sommets appartenant à ces éléments, tels que leurs nœuds. Ainsi, dans l'exemple du maillage représenté par le sous-graphe enrichi jaune de la figure 5.16, page 93, nous pouvons voir que la peau de ce dernier n'a pas été remaillée, hormis celle correspondant au bord inférieur gauche du maillage distribué.

Après le remaillage séquentiel, tous les éléments qui étaient marqués à remailler avant ce remaillage (voir figure 5.4a, page 84) et qui faisaient partie de la peau de la zone à remailler conservent leur marquage, puisque le remailleur avait interdiction de les modifier. En revanche, tous les autres sommets de la zone perdent leur drapeau, comme nous pouvons le voir en figure 5.14, page suivante. Il apparaît également sur cette même figure que les nœuds de la peau du maillage distribué ne sont pas modifiés. Dans notre exemple, nous avons supposé que le remailleur séquentiel ne modifiait pas non plus les nœuds appartenant au bord du maillage distribué.

5.2.6 Réintégration des zones remaillées

Lors de la réintégration des zones, les sommets des graphes enrichis induits, ainsi que leur liste d'adjacence, doivent être réintégrés dans le graphe enrichi distribué. Le traitement des listes d'adjacence diffère cependant selon les cas.

Nous distinguons dans le graphe enrichi distribué trois types d'éléments : externes, de peau et internes. Un élément est dit « externe » s'il n'appartient à aucune zone à remailler. À l'opposé, un élément est dit « interne », ou « de peau », s'il a été recopié dans au moins un graphe enrichi

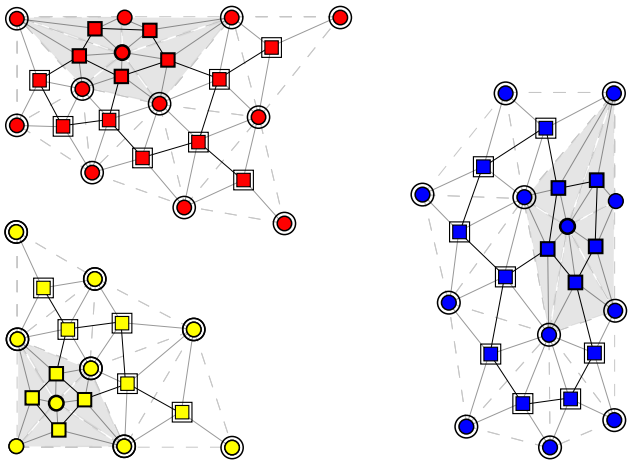


FIGURE 5.13 – Graphes enrichis centralisés (rouge, bleu et jaune) correspondant aux maillages centralisés remaillés. Les éléments et nœuds qui ne doivent pas être modifiés par le remailleur possèdent un double carré, respectivement ou un double cercle.

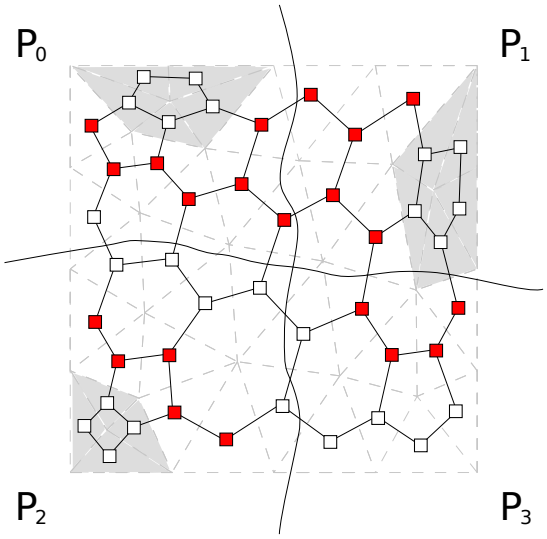


FIGURE 5.14 – Marquage des éléments à remailler après une itération de remaillage.

induit. L'élément de peau, à la différence de l'élément interne, possède au moins un sommet appartenant au bord de la zone remaillée. Seuls les éléments de peau et externes sont conservés dans le graphe enrichi distribué (voir section 5.5.2, page 98).

Nous allons maintenant détailler la méthode que nous employons pour réintégrer les sommets ainsi que leur liste d'adjacence. Il existe en fait trois cas possibles, illustrés par la figure 5.15 :

1. si un sommet (tel que le nœud 1 de la figure) appartient à un graphe enrichi induit et ne possède aucun voisin dans le graphe enrichi distribué, alors tous ses voisins au sein du graphe enrichi induit sont recopiés ;
2. si un sommet (nœud 2), qui n'est pas un élément, appartient à un graphe enrichi induit et que l'un au moins de ses voisins appartient au graphe enrichi distribué, alors tous ses voisins dans le graphe enrichi distribué sont recopiés ;
3. si un élément (élément A) appartient à un graphe enrichi induit et possède au moins un voisin dans le graphe enrichi distribué, alors tous ses voisins du graphe enrichi distribué puis de chacun des graphes enrichis induits sont recopiés, en supprimant les doublons.

Le deuxième cas peut également s'appliquer aux sommets du graphe enrichi distribué et qui n'appartiennent à aucun graphe enrichi induit. À chaque recopie, il est important de préserer l'ordre des voisins au sein de chaque entité, cet ordre étant en particulier important dans le cadre de la relation entre éléments et nœuds, comme nous l'avons vu en section 3.2.2, page 36. Afin de réintégrer les zones remaillées dans le maillage distribué, nous conservons le numéro global des sommets des mailles de peau de chaque zone remaillée. Cela nous permet de préserver la correspondance entre la numérotation globale du maillage distribué et la numérotation globale du maillage centralisé correspondant à chaque zone remaillée.

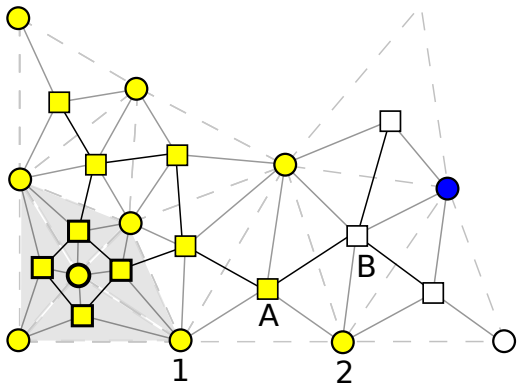


FIGURE 5.15 – Illustration des cas possibles de réintégration d'un sommet dans le graphe enrichi distribué, par les nœuds 1, 2 et les éléments A et B.

5.3 Rééquilibrage de la charge

Nous devons également garantir l'équilibrage de la charge après remaillage. Ce rééquilibrage est nécessaire tant lors du remaillage lui-même qu'après avoir obtenu le maillage distribué remaillé final.

5.3.1 Rééquilibrage lors de la réintégration des zones remaillées

Afin de minimiser le déséquilibre de charge entre sous-domaines, nous voulons répartir équitablement les sommets d'une zone remaillée sur les différents sous-domaines à partir desquels elle a été extraite. Pour les zones dont les sommets sont localisés sur plusieurs parties, nous souhaitons pouvoir conserver la même proportion de sommets par partie qu'il y avait avant remaillage, et distribuer équitablement les sommets restants. Par ailleurs, afin de préserver la localité des sommets par processeur et de ne pas permuter les processeurs lors de l'attribution des parties, les sommets appartenant à la peau des zones sont maintenus fixés sur le sous-domaine auquel ils appartenaient avant remaillage. Le but de cette répartition est de répartir le déséquilibre de la charge entre tous les processeurs partageant une zone. Cependant, cette méthode ne fonctionne pas si la zone en question provient d'un unique sous-domaine, puisque l'ensemble des nouveaux sommets créés sera affecté à ce sous-domaine, ce qui peut entraîner un important déséquilibre de charge.

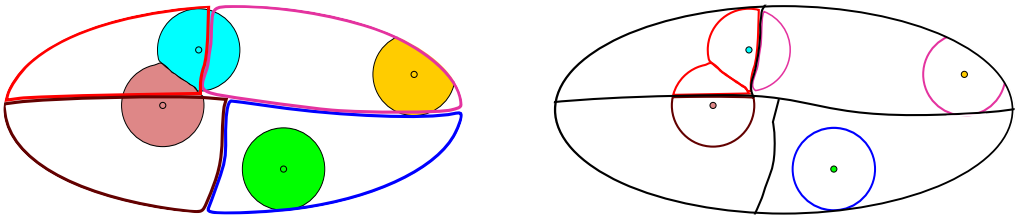


FIGURE 5.16 – Partitionnement sur points fixes au sein de chaque zone pour préserver la partie de l'enveloppe des zones présente sur chaque processeur.

5.3.2 Rééquilibrage à la fin du remaillage

Une fois qu'il ne reste plus d'éléments pouvant être remaillés, prend place une phase de redistribution globale. Son but est de remédier au déséquilibre de charge qui apparaît en dépit des techniques mises en œuvre pour le minimiser tout au long du processus de remaillage.

Nous employons pour cela une méthode de repartitionnement dynamique. Celle-ci permet de prendre en compte la répartition courante des sommets sur les processeurs, afin de minimiser leur coût de migration. Une fois le repartitionnement dynamique calculé, nous effectuons la redistribution du maillage distribué remaillé (voir section 3.3.5, page 42).

5.4 Qualité du remaillage parallèle

Notre approche de remaillage doit garantir une qualité de remaillage, équivalente à celle obtenue par un remaillage séquentiel. Sachant que le remaillage s'opère par zones, cela peut générer entre deux itérations un maillage distribué de mauvaise qualité, surtout si le rapport de raffinement ou de déraffinement est élevé. L'identification des zones peut en être altéré à l'itération suivante.

5.4.1 Contraintes sur la peau des zones à remailler

Comme nous l'avons vu, les éléments de la peau d'une zone fournie au remailleur séquentiel ne doivent pas être remaillés. Faute de cela, il ne serait pas toujours possible de réincorporer au

sein du maillage distribué initial la zone remaillée. Les éléments de peau qui étaient marqués le restent donc, pour être remaillés ultérieurement, au cours d'une itération suivante de la boucle interne.

Afin de permettre le travail du remailleur, il est donc nécessaire d'entourer les éléments que l'on souhaite remailler d'une « surcouche » d'éléments n'ayant pas vocation à être remaillés. Ceci nous amène à introduire la notion de « bande », déjà abordée implicitement à plusieurs reprises [11].

On définit la bande entourant un ensemble d'éléments comme l'ensemble des éléments voisins des sommets considérés, comme illustré en figure 5.17. Par transitivité, il est possible d'obtenir des bandes de largeur quelconque, en intégrant les éléments de bande déjà obtenus à l'ensemble des éléments considérés pour le calcul de la bande suivante.

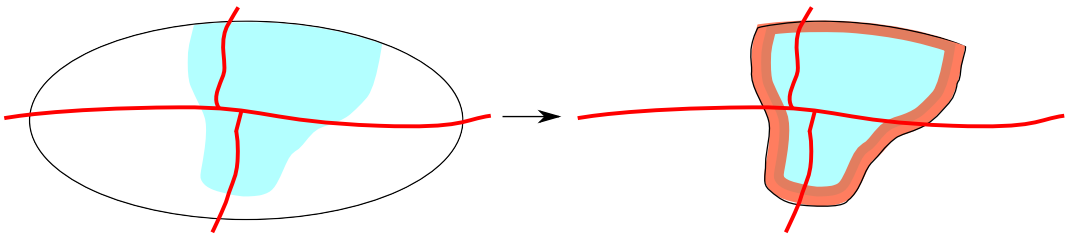


FIGURE 5.17 – Bande autour des éléments à remailler d'un graphe enrichi distribué sur quatre processeurs.

Parce que la construction des bandes fait intervenir la notion de voisinage, se pose également pour elles le fait de savoir comment est défini ce voisinage. Tout comme pour les algorithmes de croîssance de zones à partir de graines (voir sections 5.2.3.2, page 85 et 5.2.3.3, page 87), ce voisinage peut être défini à partir du graphe des éléments ou bien à partir du graphe enrichi définissant le maillage.

5.4.1.1 Graphe de bande

Le processus de construction de la bande à partir du graphe des éléments est similaire à celui de l'expansion de graine (voir section 5.2.3.2, page 85). La seule différence réside dans le fait que l'expansion ne concerne ici qu'une seule couleur, pour laquelle les multiples sommets graines sont les éléments à remailler.

Supposons, comme illustré en figure 5.18, page ci-contre, que certains éléments d'un maillage doivent être remaillés. Ces éléments ne peuvent effectivement être traités par le remailleur que si l'on fournit à celui-ci, en tant qu'éléments de peau, l'ensemble des éléments partageant au moins un nœud avec ces éléments.

Si l'on ne considère, pour définir le voisinage des éléments, que la topologie du graphe des éléments, alors une bande d'épaisseur 1 peut ne pas être toujours suffisante, comme illustré en figure 5.19a, page suivante. Dans ce cas précis, il faut calculer une bande de largeur 3 pour que tous les éléments voulus soient présents au sein de la bande, comme le montre la figure 5.19b, page ci-contre.

En fait, la largeur de bande nécessaire peut être arbitrairement grande, puisqu'elle dépend du nombre d'éléments partageant un unique nœud et devant être atteints de proche en proche

11. Lors de la construction des éléments de recouvrement et, plus récemment encore, lorsque nous avons discuté de la croîssance des zones à partir de graines.

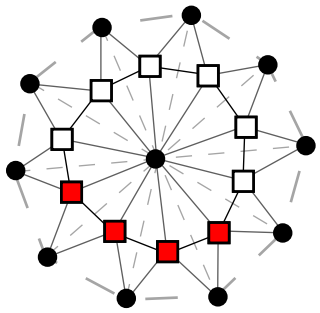
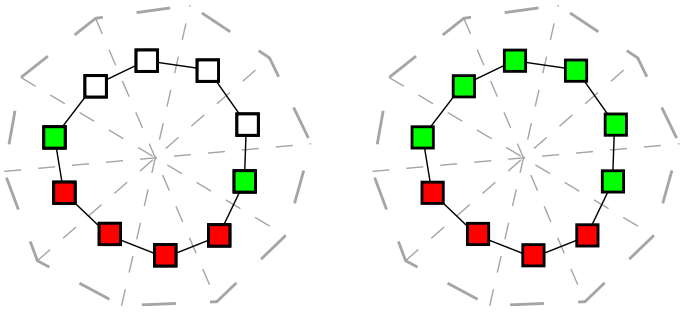


FIGURE 5.18 – Fragment de maillage dont les éléments en rouge doivent être remaillés. Seule la partie du maillage nécessaire à notre démonstration est représentée ici.



(a) Bande de largeur 1. (b) Bande de largeur 3.

FIGURE 5.19 – Construction, à partir des éléments à remailler (en rouge), de bandes d'éléments (en vert) au moyen de la relation de voisinage issue du graphe des éléments.

au travers du graphe des éléments. Le nombre de bandes nécessaire, qui peut être déjà élevé pour les maillages 2D, et est encore plus grand dans le cas des maillages 3D. La croîssance des bandes au moyen du graphe des éléments est donc particulièrement inefficace, puisqu'elle oblige, pour capturer des éléments voisins au sens des nœuds, a capturer des éléments très éloignés au sens des éléments.

5.4.1.2 Maillage de bande

C'est pour pallier au problème ci-dessus que, comme c'était déjà le cas pour la croissance des graines, nous avons décidé de fonder la relation de voisinage directement sur le graphe enrichi. La bande incorpore alors les éléments de proche en proche, par le biais des nœuds. Ainsi, l'exemple de la figure 5.20 montre comment la bande d'éléments en vert a pu être construite en seulement une itération, au travers du nœud partagé par tous les éléments.

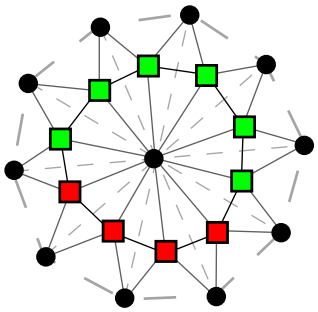


FIGURE 5.20 – Maillage de bande de distance 1 (en vert) autour des éléments marqués à remailler (en rouge) sur une partie extraite d'un graphe enrichi.

5.4.2 Formation des zones à remailler

Un autre critère essentiel à la qualité du remaillage est la surface des zones à remailler. Celle-ci doit être la plus petite possible par rapport à la taille de la zone, sinon le nombre d'itérations de la boucle interne devra être plus conséquent pour compenser ce fait. Nous avons évalué, dans la partie sur la validation expérimentale (voir section 5.6, page 100), le rapport de la taille de la peau sur la taille des zones à remailler, pour chacune des méthodes que nous avons précédemment décrites.

5.5 Optimisations en temps et en espace

Afin de rendre le remaillage parallèle aussi rapide que possible, nous avons mis en œuvre différentes méthodes d'optimisation, destinées à réduire le temps de calcul, mais aussi l'encombrement mémoire des structures de données manipulées.

Nous nous intéresserons tout d'abord à la manière de répartir sur les différents processeurs les zones à remailler. Nous mettrons ensuite en évidence l'intérêt d'utiliser un maillage distribué induit. Nous évoquerons à nouveau l'influence de la renumérotation sur le temps de remaillage. Nous discuterons alors de l'impact des méthodes d'extraction sur le nombre de zones produites à chaque itération de la boucle interne, puis nous aborderons la question de la pondération des maillages.

5.5.1 Distribution des zones à remailler

Comme nous l'avons évoqué au début de ce chapitre, le nombre de zones à remailler n'est pas corrélé au nombre de sous-domaines. Par conséquent, il est nécessaire, à chaque itération de la boucle interne de remaillage, de répartir au mieux les zones à remailler sur les différents processeurs.

Deux critères sont à prendre en considération. Le premier est le coût induit par le remaillage, qui peut différer d'une zone à l'autre. Le deuxième est le coût de centralisation d'une zone constituée de multiples fragments possédés par plusieurs processeurs. Par exemple, si une zone est à cheval entre deux processeurs, il est nécessaire d'envoyer l'un des fragments à l'autre processeur, voire d'envoyer les deux fragments à un autre processeur.

5.5.1.1 Formulation du problème

Afin de prendre en compte simultanément ces deux critères, nous reformulons ce problème comme un problème de partitionnement sous contrainte d'un graphe décrivant les tâches de remaillage. Les différents processeurs sont représentés sous la forme de sommets de poids nul, chacun fixé à la partie qu'il représente. Chacune des zones identifiées est également représentée par un sommet, dont le poids est égal au coût de calcul de remaillage estimé de cette zone. Il existe une arête entre un sommet de processeur et un sommet de zone si une fraction de la zone est déjà présente sur ce processeur. Le poids de cette arête représente le nombre d'éléments possédés par le processeur en question. La construction d'un tel graphe des tâches est illustrée en figure 5.22, page suivante.

Ainsi, calculer un partitionnement équilibré de ce graphe, et minimisant la coupe des arêtes, revient à équilibrer la charge du remaillage des zones sur les différents processeurs, en minimisant le coût de migration des fragments de zones. En effet, chaque arête coupée représente le fait qu'il faille déplacer autant d'éléments que le poids de l'arête l'indique, entre le processeur qui possède actuellement ce fragment, et le processeur sur lequel la zone comprenant ce fragment sera réassemblée et remaillée.

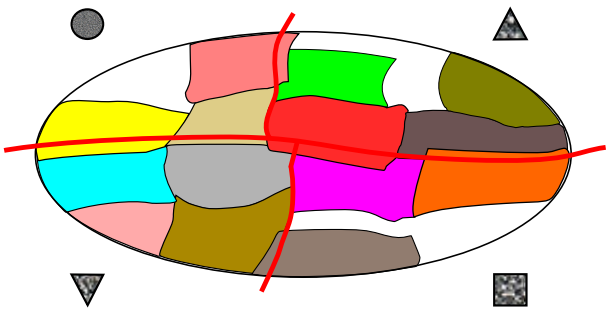


FIGURE 5.21 – Représentation des processeurs par un sommet de graphe (triangles, carré et cercle hachurés).

5.5.1.2 Estimation des coûts de calcul

La prédiction de la charge de travail que le remailleur doit consacrer à chaque élément est essentielle pour plusieurs raisons.

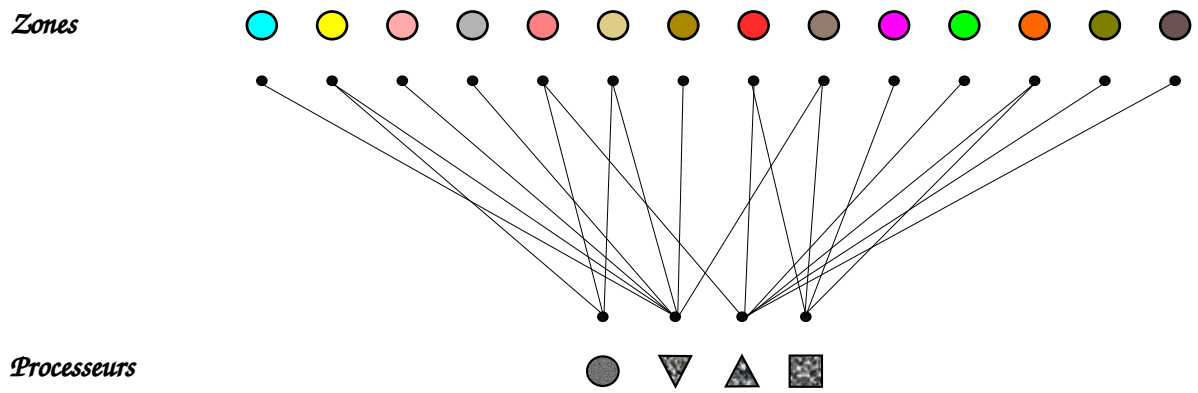


FIGURE 5.22 – Partitionnement sur points fixes des zones à remailler représentées chacune par un sommet (disque de couleur) et les processeurs par des sommets (triangles, carré et cercle hachurés).

Tout d'abord, elle est indispensable à la constitution de zones qui ne dépassent pas les capacités du remailleur séquentiel qui aura à les traiter. En effet, en fonction de la physique du cas à traiter, le remailleur peut avoir à rajouter un nombre conséquent de nœuds pour décomposer récursivement chacun des éléments d'une aire donnée. du maillage. Créer une zone unique regroupant tous ces éléments peut conduire à ce que le remailleur se trouve à court de ressources mémoire pour remailler correctement et totalement la zone.

Ensuite, elle est nécessaire à la bonne répartition des zones sur les processeurs qui auront à les remailler. Afin que cette répartition puisse être équitable en termes de charge de calcul, il faut pouvoir prédire le temps que le remailleur passera sur chaque zone, et donc sur chaque élément de ces zones.

L'estimateur de charge que nous avons conçu s'attache à prédire un coût de traitement proportionnel au nombre de nœuds qui seront insérés au sein de chaque élément, dans le cas d'un raffinement, ou de la quantité de travail nécessaire à ce que l'élément soit déraffiné. Pour le calculer, nous nous appuyons sur le tenseur de métrique en chaque nœud de l'élément (ce point sera détaillé en section 5.6.1, page 100).

5.5.1.3 Distribution des zones à remailler

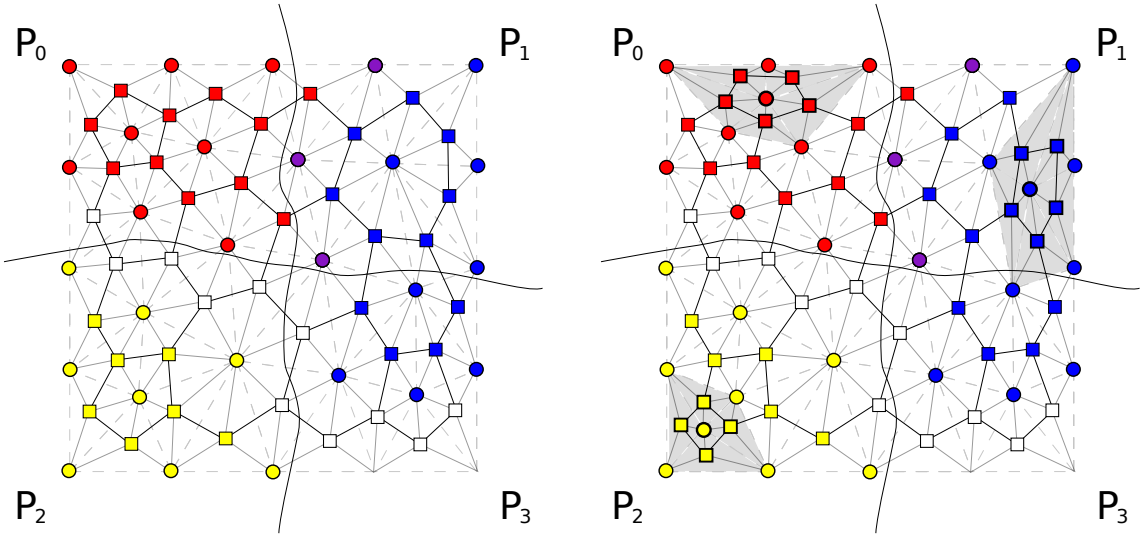
Une fois l'attribution des zones connue, chaque zone à remailler, ou ses fragments, doivent être envoyés sur le processeur qui en aura la charge. Cet envoi est réalisé à l'aide de communications collectives, en regroupant les zones destinées à chaque processeur. Ces communications collectives pourront éventuellement être remplacées par des communications point-à-point entre processeurs voisins partageant une frontière.

Un élément ne peut appartenir qu'à une seule zone, ce qui n'est pas forcément le cas pour les autres sommets. Une fois les sommets et leur liste d'adjacence échangés, les données associées nécessaires au remailleur sont elles aussi transmises. Cet envoi est séparé car les types de données des valeurs diffèrent de ceux utilisés pour décrire la topologie du maillage.

5.5.2 Maillage distribué induit

Intéressons-nous maintenant à la phase d'extraction des zones à remailler. Avant remaillage, les sommets qui sont à l'intérieur des zones vont être remplacés par des nouveaux sommets

fournis par le remailleur séquentiel, comme le montre la figure 5.23.



(a) Graphe enrichi distribué correspondant au maillage avant remaillage local.

(b) Graphe enrichi distribué correspondant au maillage après l'itération de remaillage local.

FIGURE 5.23 – Exemple de maillage dont trois zones (bleue, rouge et jaune) sont remaillées de manière indépendante.

Par conséquent, une fois que nous avons extrait les zones à remailler, nous pouvons supprimer du graphe enrichi distribué les sommets internes aux zones. Ainsi, nous ne dupliquons en mémoire, entre le graphe enrichi distribué et les graphes enrichis centralisés correspondant aux zones, que la peau de ces zones. Un exemple de maillage distribué privé de l'intérieur des zones en cours de remaillage est présenté en figure 5.24, page suivante. Le nouveau maillage distribué induit étant de taille inférieure au maillage distribué original, sa numérotation globale est bien évidemment différente. Afin de faciliter la réintégration ultérieure des zones remaillées au sein de ce maillage, Une table de correspondance est donc conservée sur le maillage induit, pour savoir à quel sommet du maillage original correspond un sommet du maillage induit. Cette table peut être vue comme une donnée associée à chaque sommet, indépendamment de son entité. Nous utilisons donc, comme support de cette valeur, l'entité virtuelle correspondant à tous les sommets (voir section 3.3.4, page 41).

5.5.3 Renumérotation

Comme nous l'avons vu dans le chapitre 4, page 57, la numérotation des sommets a un impact direct sur le temps mis par l'algorithme de remaillage. Une première renumérotation est effectuée lors de la distribution du maillage, afin de servir de base à la première itération du remaillage. Par la suite, au sein de chaque zone remaillée, le remailleur séquentiel a effectué une ou plusieurs renumérotations selon les algorithmes qu'il a pu mettre en œuvre. La dernière renumérotation ainsi effectuée est conservée et propagée pour numéroter l'intérieur des zones dans le maillage distribué. Ceci permet d'une part d'éviter de recalculer une renumérotation qui peut s'avérer coûteuse en temps et, d'autre part, de disposer d'une renumérotation garantissant une réduction des défauts de cache au sein de la zone remaillée, les zones non remaillées étant censées avoir bénéficié d'une telle renumérotation par le passé.

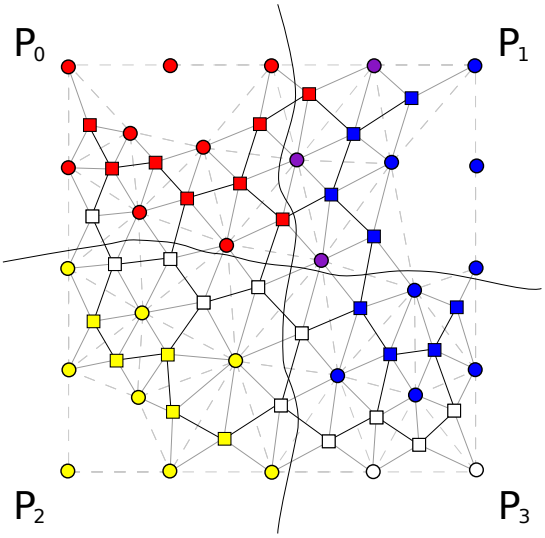


FIGURE 5.24 – Graphe enrichi distribué induit correspondant au maillage sans les parties internes des zones à remailler.

5.6 Validation expérimentale

Les tests que nous avons menés ont été effectués aussi bien avec des maillages isotropes qu'anisotropes. La différence entre les deux porte sur la définition de la métrique associée aux nœuds.

Dans le cas isotrope, cette métrique est un scalaire. Les métriques étant portées par les nœuds, cela signifie que toute arête ayant pour extrémité un nœud donné devra être d'une longueur proche de la valeur de la métrique scalaire portée par ce nœud.

Dans le cas anisotrope, la métrique est un tenseur. La taille des arêtes incidentes à un nœud donné est alors dépendante des directions de l'espace selon lesquelles ces arêtes sont orientées.

5.6.1 Rappels sur le remaillage anisotrope

Lors de la simulation numérique, la précision de la solution calculée ainsi que la convergence du schéma numérique dépendent grandement de la qualité du maillage, c'est-à-dire de l'adéquation des mailles avec l'évolution des grandeurs physiques de la solution.

L'adéquation du maillage aux valeurs de la solution courante peut être mise en œuvre en imposant une certaine taille, ainsi qu'une direction privilégiée, aux éléments du maillage, afin de contrôler l'erreur due aux calculs.

Nous allons brièvement présenter ci-dessous les notions que nous allons utiliser pour mesurer la qualité des remaillages que nous produisons. Une description plus détaillée est disponible dans [48].

5.6.1.1 Notion sur les métriques

Nous allons maintenant définir ce qu'est une métrique.

Définition 45 (Métrique)
Pour un point $P \in \mathbb{R}^3$, on définit un tenseur de métrique comme une matrice $\mathcal{M}(P)$ (3×3)

symétrique définie positive (c'est-à-dire non dégénérée) :

$$\mathcal{M}(P) = \begin{pmatrix} a & b & c \\ b & d & e \\ c & e & f \end{pmatrix},$$

avec $a > 0$, $d > 0$, $f > 0$ *et* $\det(\mathcal{M}(P)) > 0$.

Remarque : Le tenseur de métrique étant défini au point P, les différents termes de la matrice dépendent de P : $a = a(P)$, $b = b(P)$, etc.

Définition 46 (Produit scalaire)
Soient $\vec{u}$ *et* $\vec{v}$ *deux vecteurs de* $\mathbb{R}^3$. *Le produit scalaire de ces deux vecteurs dans l'espace euclidien classique pour une métrique* $\mathcal{M}$ *est défini comme suit :*

$$< \vec{u}, \vec{v} >_{\mathcal{M}} = {}^t\vec{u}\,\mathcal{M}\,\vec{v} =< {}^t\vec{u}, \mathcal{M}\,\vec{v} > \quad .$$

Définition 47 (Norme euclidienne)
À partir de la définition du produit scalaire, on peut définir la norme euclidienne d'un vecteur $\vec{u}$ *pour une métrique* $\mathcal{M}$ *:*

$$\| \vec{u} \|_{\mathcal{M}} = \sqrt{< \vec{u}, \vec{u} >_{\mathcal{M}}} = \sqrt{{}^t\vec{u}\,\mathcal{M}\,\vec{u}} \ .$$

Définition 48 (Distance)
Soient A *et* B *deux points de* $\mathbb{R}^3$. *On définit la distance entre* A *et* B *dans l'espace euclidien classique pour la métrique* $\mathcal{M}$, *constante, par :*

$$d_{\mathcal{M}}(A, B) = l_{\mathcal{M}}(\overrightarrow{AB}) = \|\overrightarrow{AB}\|_{\mathcal{M}} = \sqrt{{}^t\overrightarrow{AB}\,\mathcal{M}\,\overrightarrow{AB}} \ .$$

Définition 49 (Longueur d'un vecteur)
La longueur d'un vecteur $||\overrightarrow{v}||$ *est la distance mesurée entre les deux extrémités du vecteur* $\vec{u}$.

Définition 50 (Longueur unité)
Le vecteur $\overrightarrow{v}$ *est de longueur unité dans la métrique* $\mathcal{M}$ *si et seulement si* $||\overrightarrow{v}||_{\mathcal{M}} = 1$.

5.6.1.2 Notion de maillage unité

Nous souhaitons, par le biais du remaillage, contrôler l'erreur d'approximation sur le maillage. Pour cela, nous introduisons une métrique anisotrope en modifiant le produit scalaire qui sous-tend la notion de distance utilisée par l'adaptateur de maillages.

Nous voulons construire le maillage en générant des arêtes de longueurs unité. Cependant, la métrique peut être différente en chaque nœud du maillage. Nous devons par conséquent construire une longueur pour chaque arête. Lors de la construction du nouveau maillage, les nouveaux points sont insérés en créant des arêtes de longueur 1 : si P est un point du maillage auquel est associée la métrique $\mathcal{M}$, on veut insérer le point X tel que l'arête PX ait une longueur égale (ou proche) de 1 dans la métrique de $\mathcal{M}(P)$, c'est-à-dire :

$$l_{\mathcal{M}(P)}(PX) = \langle \overrightarrow{PX}, \overrightarrow{PX} \rangle^{\frac{1}{2}}_{\mathcal{M}(P)} = \sqrt{{}^t\overrightarrow{PX}\mathcal{M}(P)\overrightarrow{PX}} = 1 \ .$$

Géométriquement, ceci signifie que tout élément K contenant le point P doit être inclus dans l'ellipsoïde de centre P associé au tenseur de métrique $\mathcal{M}(P)$ et de telle manière que les arêtes issues de P soient de longueur 1 dans la métrique $\mathcal{M}(P)$.

En pratique, une métrique différente est prescrite en chaque sommet du maillage. Pour calculer la longueur de l'arête AB, il est nécessaire de prendre en compte à la fois de la métrique donnée en A et celle donnée en B, ainsi que des métriques de tous les points intermédiaires du segment AB.

Pour cela, nous définissons la longueur moyenne sur toutes les métriques de la manière suivante :

Définition 51 (Longueur moyenne)

$$l_{\mathcal{M}}(\overrightarrow{AB}) = \int_0^1 \sqrt{{}^t\overrightarrow{AB}\mathcal{M}(A + t\overrightarrow{AB})\overrightarrow{AB}}\, dt \ . \tag{5.1}$$

En utilisant la taille et la direction des éléments, un maillage est construit respectant au mieux ces critères. À l'aide de la majoration de l'erreur calculée en tout nœud du maillage, il est possible de générer des éléments dotés d'arêtes de longueur unité dans la métrique donnée. Nous pouvons dès lors définir le maillage unité, que nous allons utiliser par la suite.

Définition 52 (Maillage unité)
Un maillage unité est un maillage dont toutes les arêtes sont de longueur unité.

En pratique, est construit un maillage dont les arêtes ont une longueur proche de 1. Plus précisément chaque arête AB du maillage vérifie la relation suivante [54] :

$$\frac{1}{\sqrt{2}} < l_{AB} < \sqrt{2} \ .$$

5.6.1.3 Qualité du maillage

La taille des arêtes n'est pas une information suffisante pour savoir si un maillage est utilisable par des méthodes numériques. Il est nécessaire d'ajouter la notion de qualité. Celle-ci peut différer suivant l'utilisation du maillage. Dans notre cas, pour des calculs sur des éléments ou volumes finis, la forme ainsi que la répartion des, éléments sont des critères importants.

Il est possible de générer des éléments possédant des arêtes de longueur proche de 1, c'est-à-dire comprises entre $\frac{1}{\sqrt{2}}$ et $\sqrt{2}$, mais dont le volume est nul. Or, un maillage destiné à des calculs de type volumes ou éléments finis ne doit pas contenir de tels éléments.

La mesure de qualité fixée pour le tétraèdre K, dans le cas d'un maillage isotrope, est :

$$\mathcal{Q}_K = \frac{h_s^3}{V_K} \ ,$$

où $h_s = \sqrt{\sum_{i=1}^{6} h_i^2}$, h_i est la longueur de l'arête i du tétraèdre K et V_K son volume.

Dans le cas de maillages anisotropes, la qualité d'un tétraèdre est calculée comme étant celle mesurée dans l'espace euclidien relatif à une métrique moyenne définie sur le tétraèdre, comme suit :

$$\mathcal{M}_{moy} = \frac{1}{4}\left(\sum_{i=1}^{4} \mathcal{M}_i^{-1}\right)^{-1} \ .$$

La métrique moyenne est reprise pour formuler la qualité d'un tétraèdre K (le détail de ce calcul est décrit dans [48]) :

$$\mathcal{Q}_K = \alpha \frac{\left(\sum_{1 \leq i \leq j \leq 6} {}^t\overrightarrow{P_iP_j} \mathcal{M}_{moy} \overrightarrow{P_iP_j} \right)^{\frac{3}{2}}}{\sqrt{Det\left(\mathcal{M}_{moy}\right)} V_K} ,$$

où P_i et P_j sont les nœuds du tétraèdre K, et α est un coefficient de normalisation, afin que $\mathcal{Q}_K$ soit égal à 1 pour les tétraèdres équilatéraux.

5.6.2 Comparatif des méthodes testées

Afin d'évaluer l'efficacité de nos méthodes algorithmiques de remaillage parallèle, nous avons choisi comme cas test de référence un maillage de forme cubique, dont une coupe nous est présentée en figure 5.27, page 107. Le maillage est constitué de 125 000 nœuds et de 705 894 tétraèdres. La métrique isotrope varie entre $0,01$ et $0,025$, sachant que $0,025$ correspond à la taille des arêtes sur la peau du maillage et $0,01$ est la taille que nous avons choisie pour les arêtes internes. La métrique varie graduellement de $0,025$ à $0,01$.

Nous avons choisi de comparer les méthodes les plus pertinentes :

1. l'expansion de graines sur le graphe des éléments ;
2. l'expansion de graines sur le graphe enrichi ;
3. le partitionnement de graphe des éléments.

La méthode avec la contraction du graphe des éléments génère des zones difformes, avec un quotient isopérimétrique très faible. Quant à la méthode utilisant une contraction de graphe suivi par une expansion de graines sur le graphe des éléments, elle identifie quelques zones de très petites tailles, à cause de l'expansion des graines voisines. Dans les deux cas, le processus itératif peut converger difficilement à cause du nombre nécessaire d'itérations, ou ne pas converger.

La comparaison de nos méthodes a été réalisée en lançant nos cas tests sur le cluster Plafrim, en utilisant 4 processeurs.

5.6.2.1 Expansion de graines sur le graphe des éléments

Le tableau 5.1, page suivante nous donne des informations sur l'exécution et montre la qualité du maillage rapporté au maillage unité, après remaillage avec cette méthode.

5.6.2.2 Expansion de graines sur le graphe enrichi

Les résultats qui concernent l'expansion de graines sur le graphe enrichi sont recensés par le tableau 5.2, page 105.

5.6.2.3 Partitionnement de graphe

Le tableau 5.3, page 105 nous donne des informations sur l'exécution et montre la qualité du maillage rapporté au maillage unité, après remaillage avec le partitionnement de graphe.

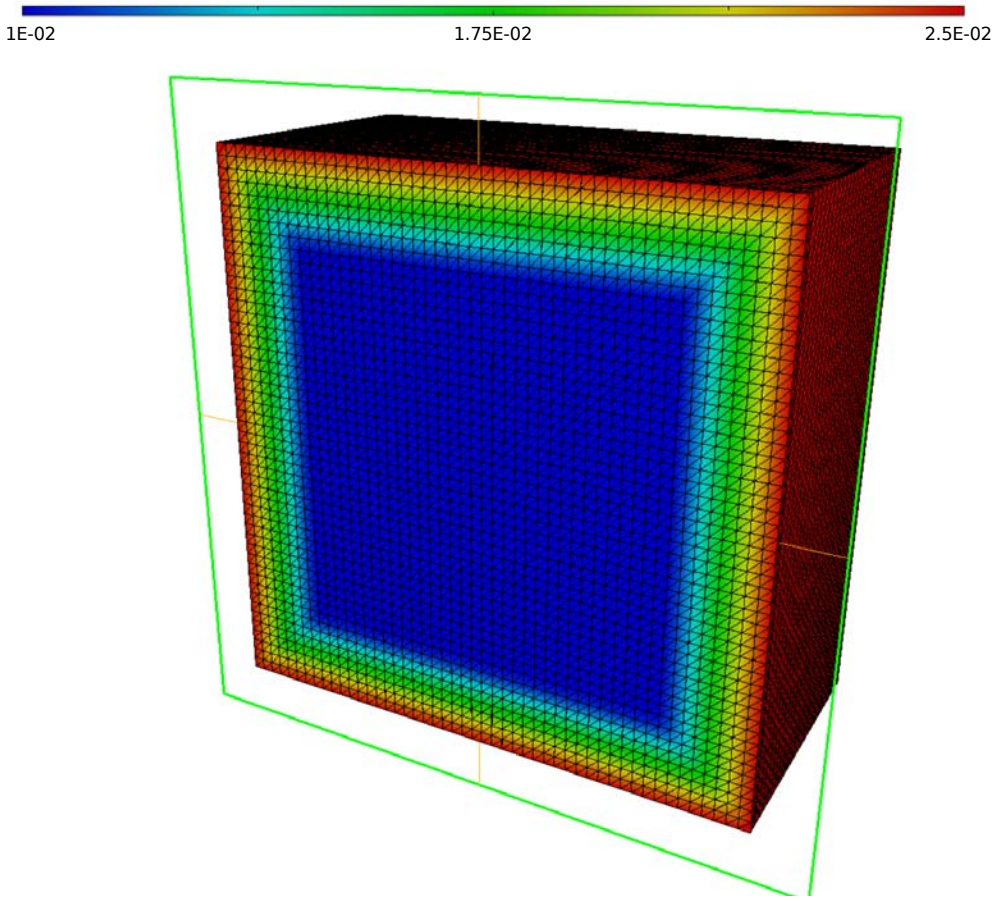


FIGURE 5.25 – Coupe d'un maillage sous forme de cube avec visualisation de sa métrique.

Temps d'exécution	8m 54s
Nombre de phases	4
Nombre d'itérations	9
Nombre d'éléments	4 629 962
Plus petite arête	0,5610
Plus longue arête	2,1937
Plus mauvais élément	7,0124
Qualité des éléments entre 1 et 2	99,79%
Arêtes entre 0,71 et 1,41	97,82%

TABLE 5.1 – Temps et caractéristiques d'un maillage isotrope sous forme d'un cube, après remaillage avec la méthode d'expansion de graines sur le graphe des éléments.

Temps d'exécution	11m 22s
Nombre de phases	4
Nombre d'itérations	44
Nombre d'éléments	4 685 527
Plus petite arête	0,5382
Plus longue arête	2,0767
Plus mauvais élément	4,97
Qualité des éléments entre 1 et 2	99,78%
Arêtes entre 0,71 et 1,41	97,82%

TABLE 5.2 – Temps et caractéristiques d'un maillage isotrope sous forme d'un cube, après remaillage avec la méthode d'expansion de graines sur le graphe enrichi.

Temps d'exécution	8m 31s
Nombre de phases	5
Nombre d'itérations	15
Nombre d'éléments	4 708 009
Plus petite arête	0,5528
Plus longue arête	2,1570
Plus mauvais élément	6,5564
Qualité des éléments entre 1 et 2	99,75%
Arêtes entre 0,71 et 1,41	97,80%

TABLE 5.3 – Temps et caractéristiques d'un maillage isotrope sous forme d'un cube, après remaillage avec la méthode de partitionnement de graphe.

5.6.2.4 Comparaison du quotient isopérimétrique

Nous avons évalué la valeur du quotient isopérimétrique pour les trois méthodes testées précédemment. Ces résultats sont présentés en figure 5.26, page suivante. Ces trois méthodes ont été couplées avec un maillage de bande, afin d'enlever les contraintes pesant sur les éléments de peau dez zones, l'interdiction d'être remaillés étant alors reportée sur les sommets de la bande.

En ce qui concerne la méthode d'expansion de graine sur le graphe enrichi, on constate une concentration des valeurs autour de $0,05$. La méthode d'expansion de graine sur le graphe produit pour sa part des zones dont le quotient isopérimétrique varie entre $0,05$ et $0,53$, alors que, pour la méthode de partitionnement, les valeurs fluctuent entre $0,03$ et $0,53$, à la différence près par rapport à la précédente que ces valeurs sont essentiellement regroupées aux alentours de $0,08$.

La valeur minimale du quotient isopérimétrique avec la méthode d'expansion de graines sur le graphe, est meilleure qu'avec les autres méthodes, et plus de zones sont formées avec un fort quotient isopérimétrique, comparé aux autres méthodes. Nous en déduisons donc qu'en termes de minimisation du la surface par rapport au volume contenu, il est plus avantageux d'utiliser la méthode d'expansion de graines sur le graphe des éléments. D'autres tests seraient nécessaires afin de déterminer si cette conclusion peut s'appliquer en général, en particulier dans le cas de maillages anisotropes pour lesquels la topologie diffère de celle des maillages isotropes, cette caractéristique impactant le graphe et par conséquent les algorithmes d'identification des zones.

5.6.2.5 Conclusion

Les trois méthodes dont nous avons reporté les résultats produisent des remaillages de qualité similaire. La principale différence concerne le temps d'exécution. La méthode d'expansion de graine sur le graphe enrichi nécessite beaucoup plus de temps que les autres. Ceci découle du nombre d'itérations effectuées, qui provient probablement d'une mauvaise taille donnée lors de l'identification des zones.

5.6.3 Solution retenue

Au vu des résultats précédents et de notre analyse, nous avons retenu comme solution d'identification des zones la méthode de partitionnement de graphes, parce qu'elle apparait plus robuste et plus rapide que les autres méthodes. Nous avons testé cette méthode aussi bien sur des maillages isotropes qu'anisotropes.

5.6.3.1 Maillages isotropes

Le maillage sur lequel nous avons réalisé nos tests est représenté en figure 5.27, page ci-contre. Il comporte 2 423 029 éléments et 1 071 626 nœuds.

Les résultats présentés dans le tableau 5.4, page 108 ont été générés à partir d'une métrique de $0,007$ appliquée au centre du maillage. Nous avons comparé les temps mis par notre méthode en parallèle, à ceux du remailleur séquentiel seul. La différence de temps rapporté au nombre de processeurs peut s'expliquer par l'omission volontaire de la renumérotation au sein de MMG3D seul, puisque nous ne l'avons pas encore implémenté dans notre version parallèle.

Les résultats qui figurent sur le tableau 5.5, page 108 ont été produits en utilisant une métrique de $0,005$. Si nous les comparons à ceux du tableau 5.4, page 108, nous remarquons que la qualité est meilleure, parce que ces tests ont utilisé la méthode multi-phases, alors que dans le précédent cas, elle n'était pas présente.

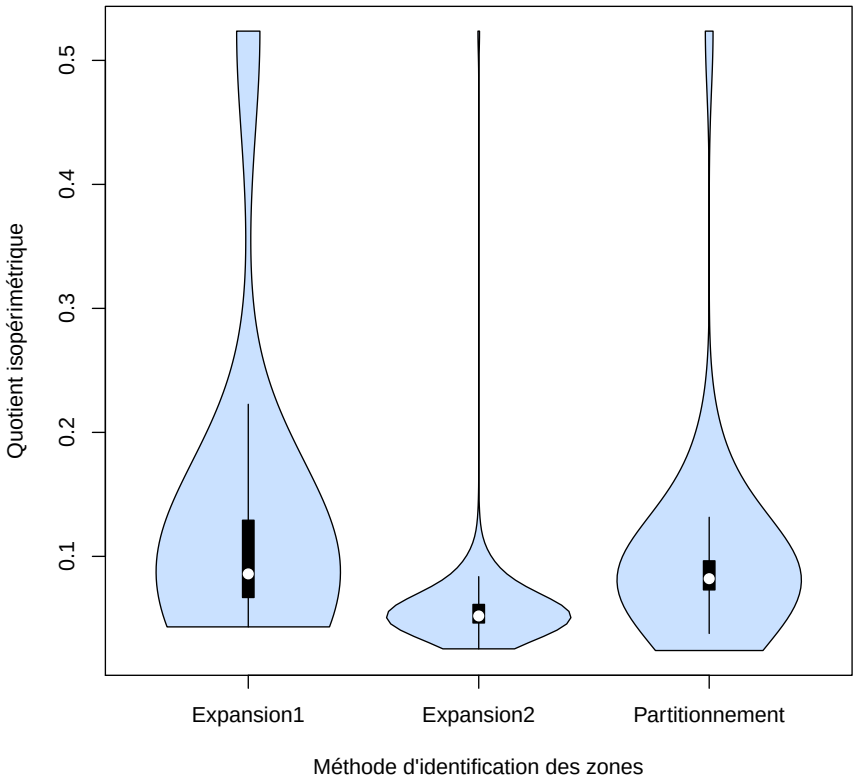


FIGURE 5.26 – Densités du quotient isopérimétrique calculé sur chaque zone remaillée, par les méthodes d'expansion de graines sur le graphe (Expansion1), d'expansion de graines sur le graphe enrichi (Expansion2) et de partitionnement de graphe.

FIGURE 5.27 – Coupe du maillage isotrope d'une biellette, sur lequel a été testé la méthode d'identification des zones par partitionnement.

	MMG3D sur 1 processeur	PaMPA-MMG3D sur 24 processeurs
Fréquence des processeurs (GHz)	2,40	3,06
Mémoire utilisée (kb)	27 588 940	51 116 044
Temps d'exécution	17h15m12s	00h21m14s
Nombre d'éléments	108 126 515	115 802 876
Plus petite arête	0,1470	0,1395
Plus longue arête	6,3309	11,2415
Qualité du plus mauvais élément	294,2669	294,2669
Qualité des éléments entre 1 et 2	99,65%	99,38%
Arêtes entre 0,71 et 1,41	97,25%	97,65%

TABLE 5.4 – Temps et caractéristiques du remaillage de la biellette avec une métrique de $0,007$ au cœur du maillage, en séquentiel et en parallèle.

	Sur 120 processeurs
Temps d'exécution	00h34m54s
Nombre d'éléments	318 027 812
Plus petite arête	0,2862
Plus longue arête	6,2161
Qualité du plus mauvais élément	235,6651
Qualité des éléments entre 1 et 2	99,58%
Arêtes entre 0,71 et 1,41	97,91%

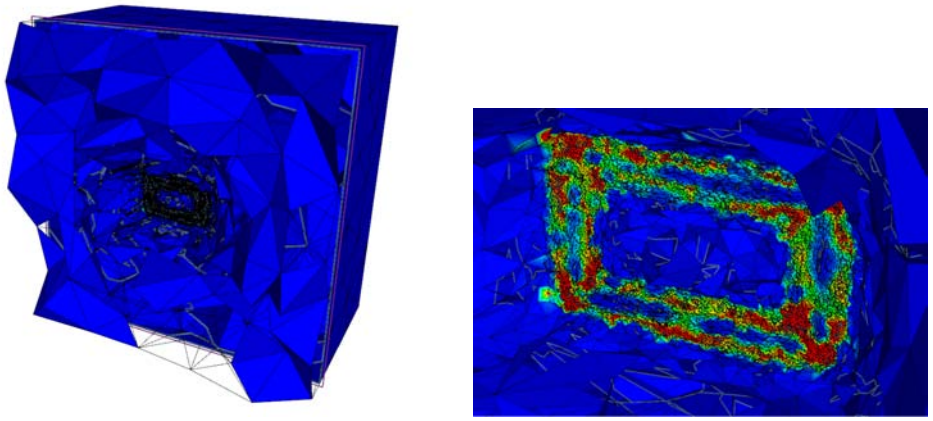
TABLE 5.5 – Temps et caractéristique du remaillage de la biellette avec une métrique de $0,005$ au cœur du maillage, en séquentiel et en parallèle.

	Sur 24 processeurs
Temps d'exécution	00h09m31s
Nombre d'éléments	10 419 623
Plus petite arête	0,1618
Plus longue arête	9,6597
Qualité du plus mauvais élément	22,2931
Qualité des éléments entre 1 et 2	98,61%
Arêtes entre 0,71 et 1,41	93,43%

TABLE 5.6 – Temps et caractéristique du remaillage du maillage sous forme d'un cube, composé de tétraèdres anisotropes.

5.6.3.2 Maillages anisotropes

Nous avons également appliquée notre méthode de remaillage parallèle avec le partitionnement de graphe, sur des maillages anisotropes. Le cas test utilisé est un maillage formé de tétraèdres anisotropes, représentant un cube. Deux coupes sont exposées en figure 5.28. Le tableau 5.6 montrent les spécificités du maillage obtenu après remaillage parallèle. La qualité du maillage est similaire à la qualité de maillages obtenus par remaillage séquentiel.



(a) Coupe globale. (b) Coupe du centre.

FIGURE 5.28 – Différentes vues du maillage anisotrope utilisé lors du test de remaillage.

Conclusion et perspectives

Conclusion

Ce mémoire nous a permis de présenter nos différentes contributions dans le domaine du remaillage parallèle destiné au calcul haute performance.

Notre exposé a commencé par un état de l'art sur le remaillage parallèle, l'équilibrage de charge et la renumérotation des données, trois domaines connexes et essentiels à la production d'outils logiciels efficaces de remaillage parallèle.

Nous avons également défini les différentes notions nécessaires à la compréhension de nos contributions.

En nous appuyant sur ce socle théorique, nous avons pu concevoir et mettre en œuvre plusieurs contributions relatives au remaillage parallèle. Ces contributions concernent trois problèmes, liés en un tout cohérent.

La première de ces contributions a consisté en la formalisation, au moyen d'un graphe enrichi, de la représentation des maillages utilisée par les numériciens. Ce formalisme permet de représenter des maillages mixtes, et d'effectuer les principales opérations mises en œuvre dans l'écriture d'un schéma numérique : définition de différents types d'entités et de sous-entités, de valeurs portées par ces entités et sous-entités, d'itérateurs opérant sur toutes les instances d'une entité ou bien toutes les instances voisines d'une instance donnée, etc.

Nous avons ensuite exposé comment cette structure de graphe enrichi pouvait être mise en œuvre de façon efficace sur une architecture parallèle à mémoire distribuée. Nous avons présenté comment numéroter les différents constituants d'un maillage distribué, au travers d'une numérotation globale et de numérotations locales. Ces numérotations permettent de mettre en œuvre les itérateurs sur les entités et sous-entités, ainsi que les recouvrements de taille variable entre sous-domaines.

Ces structures de données et les différents algorithmes pré-cités ont été implantés dans une première version de la bibliothèque logicielle PaMPA. L'utilisation de cette bibliothèque dans des cas concrets, tels que la résolution d'une équation de Poisson par une méthode d'éléments finis couplée à un Jacobi, nous a permis de résoudre différents problèmes rencontrés, comme par exemple la distinction entre recouvrement et halo.

La bibliothèque PaMPA a déjà été utilisée pour l'écriture de solveurs numériques tiers. Elle a en particulier servi de fondement à la réalisation du solveur Aerosol, réalisé en commun entre les équipes Bacchus et Cagire dans le cadre de la thèse de Damien Genêt.

Notre deuxième contribution concerne l'élaboration d'un algorithme de renumérotation en mémoire des constituants d'un maillage. La particularité de notre approche réside dans sa capacitée à être parallélisée tant à grain fin qu'à gros grain, cette parallélisation ayant été réalisée par François Pellegrini au sein du logiciel Scotch. Nous avons pu comparer notre méthode avec l'algorithme mis en œuvre au sein du programme Shrimp, et basé sur des courbes de Hilbert. L'étude expérimentale que nous avons menée montre que, bien que notre méthode fournisse en

général des résultats moins bons que celle de Shrimp, elle est en revanche facilement parallélisable, cette parallélisation permettant de compenser, par le moindre temps passé à calculer la renumérotation, notre moins bonne optimisation de celle-ci vis-à-vis du temps passé dans le remailleur.

Notre troisième contribution concerne le remaillage parallèle proprement dit, pour lequel nous avons conçu des algorithmes robustes. Nos algorithmes d'identification parallèle de zones à remailler ont été perfectionnés au fur et à mesure de notre travail. Des optimisations successives ont également permis soit d'améliorer la qualité des maillages générés, soit augmenter la vitesse d'exécution du programme. Nous avons finalement pu comparer certaines de nos méthodes sur un cas test, et avons testé plusieurs cas tests avec la méthode finalement retenue, c'est-à-dire le partitionnement du graphe des éléments. Un maillage isotrope constitué de plus de 300 millions d'éléments et un maillage anisotrope de plus de 10 millions d'éléments ont ainsi pu être remaillés en moins de 35 minutes.

Perspectives

Plusieurs perspectives de recherche sont apparues au cours de nos différents travaux.
Tout d'abord, les perspectives à court terme, qui nous semblent pertinentes, sont :

— ajouter et tester d'autres méthodes d'identification en parallèle des zones à remailler ;
— étudier comment incorporer la renumérotation au sein du processus itératif de remaillage, afin d'améliorer la vitesse de celui-ci ;
— remailler des maillages anisotropes de plus grandes tailles.

Nous pouvons ajouter, comme perspectives à moyen terme, qui nous paraissent intéressantes :

— repartitionner en parallèle en se basant sur partitionnement précédent, afin d'équilibrer la charge, tout en minimisant les migrations ;
— coupler l'interpolation des valeurs portées par le maillage, au remaillage parallèle proprement dit. Il s'agit d'un travail de recherche à part entière, commencé par Léo Nouveau, encadré par Cécile Dobrzynski et Héloïse Beaugendre. En effet, une interpolation appliquée plusieurs fois sur une zone peut induire une erreur numérique trop grande, et l'interpolation par sous-domaine est difficilement envisageable à cause de la redistribution des données lors de l'équilibrage de charge après le remaillage.

Annexe A

Code source de l'équation de Poisson

```
! Parallel program solving 2D Poisson equation on unstructured meshes :
!
!    U" = f, inside the domain
!    U  = g, on the boundary (Dirichlet condition)
!
! using PAMPA library to manage distributed mesh and MPI communications
!
! Implemented test case : f = -sin(y) and g = sin(y) => exact solution u = sin(y
)
! Modify DirichletValue, SourceTerm subroutines to change the test case
!
!
! Written by Laure Combe and extracted from Fluidbox software

PROGRAM helloPAMPA

#include "pampaf.h"
#include "pampaf-aerosol.h"

  IMPLICIT NONE
  INCLUDE 'mpif.h'

  TYPE cellule
        INTEGER                                  :: val
        TYPE(cellule), POINTER                   :: suiv=>Null()
  END TYPE cellule

  TYPE MatriceCSR
        INTEGER, DIMENSION(: ),        ALLOCATABLE :: Ia       !!-- [[NpointLoc
+ 1]], indice d'acces dans Vals et Ja au debut des elements non nuls de la ligne
        REAL,    DIMENSION(: ),        ALLOCATABLE :: Vals     !!-- [[Nnz]], val
eur des elements non nuls de la matrice
        REAL,    DIMENSION(: ),        ALLOCATABLE :: Diag     !!-- [[NpointLoc]
], valeur des elements diagonaux de la matrice
        INTEGER, DIMENSION(: ),        ALLOCATABLE :: Ja       !!-- [[Nnz]], col
onne ayant un element non nul
        INTEGER                                  :: Nnz      !!-- nombre total
 de termes non nuls dans la matrice, i.e. de voisins dans le graphe
        INTEGER                                  :: SizeMat  !!-- nombre de li
gnes dans la matrice = NpointLoc (= locaux + ghosts)
        INTEGER                                  :: Nin      !!-- nombre de li
gnes correspondant aux points locaux (ghosts exclus) dans la matrice
  END TYPE MatriceCSR

  INTEGER                                    :: rang, nproc, code, dummy
  CHARACTER(LEN = 100)                       :: InFileName, OutFileName
  LOGICAL                                    :: fileExist

  INTEGER,                                SAVE :: Ndim = 2
  INTEGER,                                SAVE :: Ndegre
  INTEGER,                                SAVE :: NpointGlob
  REAL,    DIMENSION(:, : ), POINTER,     SAVE :: CoorGlob,     Coor
  INTEGER,                                SAVE :: NelemtGlob
  INTEGER, DIMENSION(:, : ), ALLOCATABLE, SAVE :: NuGlob
  INTEGER,                                SAVE :: NsegFrGlob
  INTEGER, DIMENSION(:, : ), ALLOCATABLE, SAVE :: SegFrGlob
  INTEGER, DIMENSION(: ),    POINTER,     SAVE :: LogSegFrGlob, LogSegFr

  REAL,    DIMENSION(: ),    POINTER          ::               Ua
  INTEGER,                                SAVE ::               NpointFrLoc
  REAL,    DIMENSION(: ),    POINTER,     SAVE ::               VolEl
  TYPE(MatriceCSR),                       SAVE ::               MatCSR
  REAL,    DIMENSION(: ),    POINTER          ::               RHS

  TYPE(cellule), DIMENSION (: ), TARGET, ALLOCATABLE :: hashtableN2E, hashtableN
2SFr, hashtableN2N, hashtableE2E, hashtableE2SFr
  INTEGER,       DIMENSION(: ),          ALLOCATABLE :: NbVoisN2E, NbVoisN2SFr,
NbVoisN2N, NbVoisE2E, NbVoisE2SFr
  INTEGER,       DIMENSION(: ),          ALLOCATABLE :: BndEdge2Elemt
  INTEGER                                            :: cptN2E, cptN2SFr, cptN2N
, cptEdgeTab, cptE2E, cptE2SFr

  INTEGER(KIND=PAMPAF_NUM),           PARAMETER :: PAMPA_ENTITY_ELMT    =  1  !
Element      entity
  INTEGER(KIND=PAMPAF_NUM),           PARAMETER :: PAMPA_ENTITY_NODE    =  3  !
Node         entity
  INTEGER(KIND=PAMPAF_NUM),           PARAMETER :: PAMPA_ENTITY_BNDEDGE =  2  !
Boundary edge entity
  INTEGER(KIND=PAMPAF_NUM),           PARAMETER :: enttnbr              =  3  !
elements + nodes + boundary edges
  INTEGER(KIND=PAMPAF_NUM),           PARAMETER :: baseval              =  1  !
delta offset between C and Fortran beginning index arrays
  INTEGER(KIND=PAMPAF_NUM),           PARAMETER :: overlap              =  1
  INTEGER(KIND=PAMPAF_NUM),           PARAMETER :: PAMPA_TAG_BNDLOG     =  1
  INTEGER(KIND=PAMPAF_NUM),           PARAMETER :: PAMPA_TAG_SOL        =  2
  INTEGER(KIND=PAMPAF_NUM),           PARAMETER :: PAMPA_TAG_VOL        =  3
  INTEGER(KIND=PAMPAF_NUM),           PARAMETER :: PAMPA_TAG_RHS        =  4
  INTEGER(KIND=PAMPAF_NUM), DIMENSION(:), ALLOCATABLE :: datatab
  DOUBLE PRECISION                           :: strat(PAMPAF_STRATDIM)

  ! Global PAMPA mesh (before partitionning)
  DOUBLE PRECISION                           :: dm_tmp(PAMPAF_DMESHDIM)   ! D
istributed original mesh (Before partionning)
  INTEGER(KIND=PAMPAF_NUM)                   :: vertnbr,      &
       &                                               edgenbr,      &
       &                                               edgesiz,      &
       &                                               valumax
  INTEGER(KIND=PAMPAF_NUM), ALLOCATABLE, TARGET, DIMENSION(:) :: &
       &                                           venttab,      &
       &                                           verttab,      &
       &                                           edgetab,      &
       &                                           partloctab

  INTEGER(KIND=PAMPAF_NUM), DIMENSION(:), POINTER :: permgsttab

  ! Distributed PAMPA mesh
  DOUBLE PRECISION                           :: dm(PAMPAF_DMESHDIM) ! Distrib
uted mesh              (After partionning)
  INTEGER(KIND=PAMPAF_NUM)                   :: vertlocnbr,   &
       &                                           edgelocnbr,   &
       &                                           nodelocnbr,   &
       &                                           elemlocnbr,   &
       &                                           segfrlocnbr
  INTEGER(KIND=PAMPAF_NUM), ALLOCATABLE, TARGET, DIMENSION(:) :: &
       &                                           vertloctab,   &
       &                                           edgeloctab,   &
       &                                           ventloctab
  REAL,    DIMENSION(:,:), ALLOCATABLE       :: coorloctab         ! nodes coo
rdinates
  INTEGER, DIMENSION(:),   ALLOCATABLE       :: logloctab          ! nodes tag
 (0 for internal and >0 for boundary)
  INTEGER, DIMENSION(:),   ALLOCATABLE       :: logsegfrloctab     ! boundary
edges tag

  INTEGER                                    :: type_coor, type_real

  CHARACTER(LEN = 3)                         :: rangstr, nprocstr  ! increase
the string size if nbproc > 999
  INTEGER                                    :: len1, len2

  ! Residus output
  INTEGER,                                SAVE :: resdunit
  CHARACTER(LEN=15),                      SAVE :: resdfile

  CHARACTER(LEN=1024)                        :: exec
  INTEGER                                    :: nb_args
  DOUBLE PRECISION,                   PARAMETER :: PI    = 2.d0 * asin(1.d0)
  CHARACTER(LEN = 10),                PARAMETER :: iofmt = '(A,I9,X,A)'

  CHARACTER(LEN=3+8)                         :: outfile
  INTEGER                                    :: outunit, istat, ierr


  NULLIFY(Ua)
  NULLIFY(CoorGlob)
  NULLIFY(Coor)
  NULLIFY(VolEl)
  NULLIFY(RHS)

  ! MPI Initializations
  !--------------------
  CALL MPI_INIT_THREAD(MPI_THREAD_MULTIPLE, dummy, code)

  IF (code /= 0 .OR. dummy /= MPI_THREAD_MULTIPLE) THEN
        WRITE(6,*) "MPI_INIT_THREAD error", MPI_THREAD_MULTIPLE, dummy, code
        STOP
  END IF
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD,nproc,code)
  IF (nproc > 999) THEN
        WRITE(6,*) "ERROR : Number of precessors > 999"
        STOP
  END IF
  CALL MPI_COMM_RANK(MPI_COMM_WORLD,rang,code)

  ! Define real type to declare to mpi
  IF (Kind(1.0) == Selected_real_kind(8)) THEN
        type_real = MPI_REAL8
  ELSE IF (Kind(1.0) == Selected_real_kind(4)) THEN
        type_real = MPI_REAL4
  ELSE
        WRITE(6,*) rang, "ERROR : kind(1.0)==", Kind(1.0), " not taken into account"
        STOP
  END IF
  IF (rang == 0) THEN
        print *, "nb procs mpi:", nproc
        WRITE(6,*) rang, "kind(1.0) =", Kind(1.0)
  END IF

  Ndim = 3
  CALL MPI_TYPE_CONTIGUOUS(Ndim, type_real, type_coor, code)
  CALL MPI_TYPE_COMMIT(type_coor,code)

  ! Select mesh
  !------------
  CALL getarg(0, exec)

  nb_args = iargc()

  IF (nb_args /= 2) THEN
        IF (rang == 0) THEN
      WRITE(6,*) "ERROR : bad number of arguments :", nb_args
      WRITE(6,*) "Command line : helloPAMPA InInFileName OutInFileName"
        END IF
        STOP
  END IF

  CALL getarg(1, InFileName)
  CALL getarg(2, OutFileName)

  INQUIRE(FILE = InFileName, EXIST = fileExist)
  IF (.NOT. fileExist) THEN
        WRITE(6,*) "ERROR : File ", TRIM(InFileName), " not found"
        STOP 100
  END IF
  IF (rang == 0) THEN
        WRITE(6,*) rang, "Test case : ", InFilename
  END IF

  ! Compute a string associated to the current processor (used for output and de
bug files)
  rangstr = " "
  len1 = 1
  WRITE(rangstr(len1: len1 + 2), "(i3)") rang + 1
  rangstr(len1: len1 + 2) = Adjustl(rangstr(len1: len1 + 2))
  IF((rang+1)<100)then
        len2 = Index(rangstr(len1: len1 + 2), " ") -1
        rangstr(len1: len1 + 2) = Repeat("0", 3 - len2) // Trim(rangstr(len1: le
n1 + len2))
  END IF
  WRITE(rangstr, "(A)") Trim(Adjustl(rangstr))

  ! Compute a string associated to the number of processors (used for residu fil
e)
  nprocstr = " "
  len1 = 1
  WRITE(nproc, "(i3)") nproc
  nprocstr(len1: len1 + 2) = Adjustl(nprocstr(len1: len1 + 2))
  IF((nproc)<100)then
        len2 = Index(nprocstr(len1: len1 + 2), " ") -1
        nprocstr(len1: len1 + 2) = Repeat("0", 3 - len2) // Trim(nprocstr(len1:
len1 + len2))
  END IF
  WRITE(nprocstr, "(A)") Trim(Adjustl(nprocstr))

  ! open output file
  IF (rang == 0) THEN
        outunit = 10
        outfile = Trim(Adjustl("out_" // nprocstr // ".out"))
        OPEN( UNIT = outunit, FILE = Trim(outfile), IOSTAT = istat, STATUS = "R
EPLACE" )
        IF (istat /= 0) THEN
      WRITE(6,*) " ERROR : ouverture de ", Trim(outfile), " -> ", istat
      STOP
        END IF
        REWIND(outunit)
  END IF

  ! Parallel Mesh Computations
  !---------------------------
  CALL PAMPAF_dmeshinit(dm_tmp, MPI_comm_world, retval = ierr)
  IF (ierr .ne. 0) THEN
    WRITE(6,*) rang, "PAMPAF_dmeshinit error"
    STOP
  END IF

  allocate (datatab(2))
  datatab(1) = 1
  datatab(2) = 3
        CALL PAMPAF_AEROSOL_dmeshLoad(dm_tmp, Infilename, datatab, ior (PAMPAF
_AEROSOL_FACES, PAMPAF_AEROSOL_NODE2NODE) , ierr)
        IF (ierr .ne. 0) THEN
      WRITE(6,*) rang, "PAMPAF_AEROSOL_dmeshload error"
      STOP
        END IF
#ifdef DEBUG
        CALL PAMPAF_dmeshcheck(dm_tmp, ierr)
        IF (ierr .ne. 0) THEN
      WRITE(6,*) rang, "PAMPAF_dmeshcheck error"
      STOP
    else
      WRITE(6,*) rang, "PAMPAF_dmeshcheck ok"
        END IF
#endif /* DEBUG */
```

```
  ! On all processors:
  CALL PampaDistributedMesh() ! Build PAMPA distributed mesh:
  !                           1- Scatter centralized mesh on all processo
rs
  !                           2- Call PAMPA mesh partionner
  !                           3- Redistribute distribute mesh
  !CALL ElementVolume()
  CALL ElementVolume3d()
  CALL InitializeMatrixCSR()

  ! Solution computation
  !---------------------

  CALL InitSol()
  CALL FillMatrix()
  CALL SolveSystem()
  !CALL WriteDistributedMeshAndSolFiles()
  CALL WriteDistributedMeshAndSolFiles3d()
  ! CALL WriteDistributedMeshAndSolFiles3d_bis()
  CALL PAMPAF_dmeshexit(dm)


  ! Free memory
  !------------

  IF (ALLOCATED(MatCSR%Vals))   DEALLOCATE(MatCSR%Vals)

  IF (ALLOCATED(MatCSR%Diag))   DEALLOCATE(MatCSR%Diag)

  IF (ALLOCATED(MatCSR%Ia))     DEALLOCATE(MatCSR%Ia)

  IF (ALLOCATED(MatCSR%Ja))     DEALLOCATE(MatCSR%Ja)

  WRITE(6,*)
  WRITE(6,*) "----------------------------------------"
  WRITE(6,*) rang, "HelloPAMPA successfully ended"
  WRITE(6,*) "----------------------------------------"
  WRITE(6,*)

  ! MPI exit
  !---------
  CALL MPI_TYPE_FREE(type_coor, code)
  IF (rang == 0) CLOSE(outunit)
  CALL MPI_FINALIZE(code)


  STOP

  CONTAINS

  SUBROUTINE PampaDistributedMesh()
        CHARACTER(LEN = *), PARAMETER              :: mod_name = "PampaDistributedM
esh"
        INTEGER                                    :: statut, is, np
        REAL                                       :: temps_fin, temps_deb
        INTEGER(KIND=PAMPAF_NUM), DIMENSION(:), ALLOCATABLE, TARGET
:: entttab, tagtab
        INTEGER(KIND=PAMPAF_NUM) :: typenbr
        INTEGER, DIMENSION(:), ALLOCATABLE, TARGET             :: typetab

        ! number of associated values to PaMPA entities
        valumax = 6 ! Node coordinates, node log, boundary edge log, element vol
ume, physical solution and RHS
        WRITE(6,*)
        WRITE(6,*) "****************************************************************************
********"
        WRITE(6,*) rang, "Construction du maillage distribue PaMPA ... DEB"

        CALL Cpu_time(temps_deb)

        CALL PAMPAF_stratinit(strat, statut)
        IF (statut .ne. 0) THEN
      WRITE(6,*) rang, mod_name, "PAMPAF_stratinit error"
      STOP
        END IF

        CALL PAMPAF_dmeshSize(dm_tmp, vertlocnbr = vertlocnbr)
        ALLOCATE(partloctab(vertlocnbr))

        ! Partition distributed mesh
        CALL PAMPAF_dmeshpart(dm_tmp, nproc, strat, partloctab, statut)
        IF (statut .ne. 0) THEN
      WRITE(6,*) rang, mod_name, "PAMPAF_meshpart error"
      STOP
        END IF

        CALL PAMPAF_dmeshinit(dm, MPI_comm_world, retval = statut)
        IF (statut .ne. 0) THEN
      WRITE(6,*) rang, mod_name, "PAMPAF_dmeshinit error"
      STOP
        END IF

        ! Redistribute temporary distributed mesh according to the partitioning
        CALL PAMPAF_dmeshRedist(dm_tmp,partloctab, vertlocdlt = INT(-1, PAMPAF_N
UM), edgelocdlt = INT(-1, PAMPAF_NUM), ovlpglbval = INT(1, PAMPAF_NUM), dmshptr
= dm, retval = statut)
        IF (statut .ne. 0) THEN
      WRITE(6,*) rang, mod_name, "PAMPAF_dmeshinit error"
      STOP
        END IF
        CALL PAMPAF_dmeshexit(dm_tmp)
#ifdef DEBUG
        CALL PAMPAF_dmeshCheck(dm, statut)
        IF (statut .ne. 0) THEN
      WRITE(6,*) rang, mod_name, "PAMPAF_dmeshinit error"
      STOP
        END IF
#endif /* DEBUG */
        DEALLOCATE (partloctab)

        ! Associate data to PaMPA entities
        !---------------------------------
        ! - Read local data
        ! - Create MPI contiguous type if necessary and declare it to PaMPA
        ! - Allocate and Associate data to entity  (PAMPAF_dmeshvaluelink  )
        ! - Retrieve data of overlap entities      (PAMPAF_dmeshHaloPartial)
        ! Retrieve coordinates of ghost nodes
        CALL PAMPAF_dmeshValueData(dm,&
          &                        enttnum    = PAMPA_ENTITY_NODE,
    &
          &                        tagnum  = INT(PAMPAF_TAG_GEOM, PAMPAF_NUM),
  &
          &                        valuloctab = coor,
&
          &                        retval     = statut)

        CALL PAMPAF_dmeshHaloValue(dm,
&
          &                        enttnum = PAMPA_ENTITY_NODE,
  &
          &                        tagnum  = INT(PAMPAF_TAG_GEOM, PAMPAF_NUM),
 &
          &                        retval  = statut)

        ! Boundary edges Log data
        !------------------------
```

```
        ! FIXME on suppose avoir que 0 comme ref
        CALL PAMPAF_dmeshvaluelink(dm,                                    &
          &                        valuloctab = LogSegFr,
    &
          &                        flagval = PAMPAF_VALUE_PUBLIC,
&
          &                        typeval    = mpi_integer,
&
          &                        enttnum    = PAMPA_ENTITY_BNDEDGE,
 &
          &                          tagnum     = PAMPAF_TAG_REF,
    &
          &                        retval     = ierr)

        logsegfr = 0

        ! Retrieve log of ghost boundary edges

        CALL PAMPAF_dmeshHaloValue(dm,
&
          &                          enttnum = PAMPA_ENTITY_BNDEDGE,
   &
          &                          tagnum     = PAMPAF_TAG_REF,
    &
          &                          retval  = statut)

        CALL Cpu_time(temps_fin)
        temps_fin = temps_fin - temps_deb
        WRITE(6,*) rang, "Temps construction maillage distribue PaMPA :", temps_fin

        WRITE(6,*) rang, "Construction du maillage distribue PaMPA ... FIN"
        WRITE(6,*) "****************************************************************************
********"
        WRITE(6,*)
        flush(6)

  END SUBROUTINE PampaDistributedMesh

  SUBROUTINE ElementVolume()
    CHARACTER(LEN = *),  PARAMETER      :: mod_name = "ElementVolume"
    INTEGER,             PARAMETER      :: Nsmplx = 3
    INTEGER, DIMENSION(Nsmplx)          :: NuElem
    INTEGER                             :: jt, ierr, ngb, ingb
    TYPE(PAMPAF_Iterator)               :: it_vrt, it_ngb
    INTEGER(KIND=PAMPAF_NUM)            :: elmtnbr, elmtovnbr, elmtghstnbr

    WRITE(6,FMT=iofmt)       "-> ", rang, mod_name
    CALL PAMPAF_dmeshvaluelink(dm,                                        &
      &                        valuloctab = VolEl,                       &
      &                        flagval = PAMPAF_VALUE_PUBLIC,              &
      &                        typeval    = type_real,                   &
      &                        enttnum    = PAMPA_ENTITY_ELMT,           &
      &                        tagnum     = INT(PAMPA_TAG_VOL, PAMPAF_NUM), &
      &                        retval     = ierr)

    CALL PAMPAF_dmeshEnttSize(dm, PAMPA_ENTITY_ELMT, dummy,    elmtnbr,  elmtovnb
r,  elmtghstnbr,  retval = ierr)

    VolEL = 0.

    CALL PAMPAF_dmeshItInitStart(dm,  PAMPA_ENTITY_ELMT,  PAMPAF_VERT_LOCAL,  it_vr
t,  ierr)
    CALL PAMPAF_dmeshItInit     (dm,  PAMPA_ENTITY_ELMT,  PAMPA_ENTITY_NODE,  it_n
gb,  ierr)

    DO WHILE (PAMPAF_itHasMore(it_vrt))
      jt   = PAMPAF_itCurEnttVertNum(it_vrt)
      CALL PAMPAF_itStart(it_ngb, jt, ierr)

      ! Retrieve element connectivity
      ngb    = 0
      NuElem = 0
      DO WHILE (PAMPAF_itHasMore(it_ngb))
        ngb        = ngb + 1
        ingb       = PAMPAF_itCurEnttVertNum(it_ngb)
        NuElem(ngb) = ingb
        PAMPAF_itNext(it_ngb)
      END DO

      ! Compute element volume
      VolEl(jt) = 0.5 * abs(    (Coor(1,NuElem(2)) - Coor(1,NuElem(1))) * (Coor(2
,NuElem(3)) - Coor(2,NuElem(1))) &
        &                       - (Coor(2,NuElem(2)) - Coor(2,NuElem(1))) * (Coor
(1,NuElem(3)) - Coor(1,NuElem(1))))

      PAMPAF_itNext(it_vrt)
    END DO

    ! Retrieve volume of ghost elements
    CALL PAMPAF_dmeshHaloValue(dm,                                        &
      &                        enttnum = PAMPA_ENTITY_ELMT,             &
      &                        tagnum  = INT(PAMPA_TAG_VOL, PAMPAF_NUM),   &
      &                        retval  = ierr)
    WRITE(6,FMT=iofmt)       "<- ", rang, mod_name

  END SUBROUTINE ElementVolume

  SUBROUTINE ElementVolume3d()
        CHARACTER(LEN = *), PARAMETER       :: mod_name = "ElementVolume3d"
        INTEGER,            PARAMETER       :: Nsmplx = 4 ! Tetrahedra
    INTEGER, DIMENSION(Nsmplx)          :: NuElem
    REAL, DIMENSION(3,Nsmplx)           :: CoorElemt
        INTEGER                             :: jt, ierr, ngb, ingb, k
        TYPE(PAMPAF_Iterator)               :: it_vrt, it_ngb
        INTEGER(KIND=PAMPAF_NUM)            :: elmtnbr, elmtovnbr, elmtghstnbr

        WRITE(6,FMT=iofmt)       "-> ", rang, mod_name
        CALL PAMPAF_dmeshvaluelink(dm,                                    &
          &                        valuloctab = VolEl,
 &
          &                        flagval = PAMPAF_VALUE_PUBLIC,
   &
          &                        typeval    = type_real,
 &
          &                        enttnum    = PAMPA_ENTITY_ELMT,
 &
          &                        tagnum     = INT(PAMPA_TAG_VOL, PAMPAF_NUM)
, &
          &                        retval     = ierr)

        CALL PAMPAF_dmeshEnttSize(dm, PAMPA_ENTITY_ELMT, dummy,    elmtnbr,  elmt
ovnbr,  elmtghstnbr,  retval = ierr)

        VolEL = 0.

        CALL PAMPAF_dmeshItInitStart(dm,  PAMPA_ENTITY_ELMT,  PAMPAF_VERT_LOCAL,  i
t_vrt,  ierr)
        CALL PAMPAF_dmeshItInit     (dm,  PAMPA_ENTITY_ELMT,  PAMPA_ENTITY_NODE,
it_ngb,  ierr)

        DO WHILE (PAMPAF_itHasMore(it_vrt))
      jt   = PAMPAF_itCurEnttVertNum(it_vrt)
      CALL PAMPAF_itStart(it_ngb, jt, ierr)

      ! Retrieve element connectivity
      ngb    = 0
```

```
      NuElem = 0
      DO WHILE (PAMPAF_itHasMore(it_ngb))
        ngb         = ngb + 1
        ingb        = PAMPAF_itCurEnttVertNum(it_ngb)
        NuElem(ngb) = ingb
        PAMPAF_itNext(it_ngb)
      END DO

      DO k = 1, Nsmplx
        CoorElemt(:,k) = Coor(:,NuElem(k))
      END DO

      ! Compute tetrahedron volume
      VolEl(jt) =   (   CoorElemt(1,2) * (CoorElemt(2,3) * CoorElemt(3,4) - Coor
Elemt(3,3) * CoorElemt(2,4))   &
      &             - CoorElemt(1,3) * (CoorElemt(2,2) * CoorElemt(3,4) - Coor
Elemt(3,2) * CoorElemt(2,4))   &
      &             + CoorElemt(1,4) * (CoorElemt(2,2) * CoorElemt(3,3) - Coor
Elemt(3,2) * CoorElemt(2,3)) ) &
      &         - (   CoorElemt(1,1) * (CoorElemt(2,3) * CoorElemt(3,4) - Coor
Elemt(3,3) * CoorElemt(2,4))   &
      &             - CoorElemt(1,3) * (CoorElemt(2,1) * CoorElemt(3,4) - Coor
Elemt(3,1) * CoorElemt(2,4))   &
      &             + CoorElemt(1,4) * (CoorElemt(2,1) * CoorElemt(3,3) - Coor
Elemt(3,1) * CoorElemt(2,3)) ) &
      &         + (   CoorElemt(1,1) * (CoorElemt(2,2) * CoorElemt(3,4) - Coor
Elemt(3,2) * CoorElemt(2,4))   &
      &             - CoorElemt(1,2) * (CoorElemt(2,1) * CoorElemt(3,4) - Coor
Elemt(3,1) * CoorElemt(2,4))   &
      &             + CoorElemt(1,4) * (CoorElemt(2,1) * CoorElemt(3,2) - Coor
Elemt(3,1) * CoorElemt(2,2)) ) &
      &         - (   CoorElemt(1,1) * (CoorElemt(2,2) * CoorElemt(3,3) - Coor
Elemt(3,2) * CoorElemt(2,3))   &
      &             - CoorElemt(1,2) * (CoorElemt(2,1) * CoorElemt(3,3) - Coor
Elemt(3,1) * CoorElemt(2,3))   &
      &             + CoorElemt(1,3) * (CoorElemt(2,1) * CoorElemt(3,2) - Coor
Elemt(3,1) * CoorElemt(2,2)) )
      VolEl(jt) = abs(VolEl(jt)) / 6.

      PAMPAF_itNext(it_vrt)
        END DO
        CALL PAMPAF_dmeshItExit(it_vrt, ierr)
        CALL PAMPAF_dmeshItExit(it_ngb, ierr)


        ! Retrieve volume of ghost elements
        CALL PAMPAF_dmeshHaloValue(dm,                                        &
&
          &                        enttnum = PAMPA_ENTITY_ELMT,
&
          &                        tagnum  = INT(PAMPA_TAG_VOL, PAMPAF_NUM),
 &
          &                        retval  = ierr)
        WRITE(6,FMT=iofmt)       "<- ", rang, mod_name

  END SUBROUTINE ElementVolume3d

  SUBROUTINE InitializeMatrixCSR()
        CHARACTER(LEN = *),                 PARAMETER :: mod_name = "InitializeMatrix
CSR"
        INTEGER                                       :: inode, ingb, k, iref, i
proc, np, ierr
        INTEGER                                       :: nodenbr, nodeovnbr, nod
eghstnbr
        TYPE(PAMPAF_Iterator)                         :: it_vrt, it_ngb
        INTEGER                                       :: nbnghb, nbnghbsum, nnz
        REAL                                          :: temps_fin, temps_deb
        INTEGER, DIMENSION(:),            ALLOCATABLE :: matsize, matnin, matnnz

        WRITE(6,FMT=iofmt)       "-> ", rang, mod_name
        CALL Cpu_time(temps_deb)

        CALL PAMPAF_dmeshEnttSize(dm, PAMPA_ENTITY_NODE, dummy,    nodenbr,  node
ovnbr,  nodeghstnbr,  retval = ierr)

        CALL PAMPAF_dmeshMatInit(dm, PAMPA_ENTITY_NODE, ierr)
        CALL PAMPAF_dmeshMatSize(dm, PAMPA_ENTITY_NODE, nnz, ierr)
        WRITE(6,*) rang, "Numbers of non zero of the matrix computed by PAMPA", Nnz

        MatCSR%Nnz      = nnz
        MatCSR%Nin      = nodenbr
        MatCSR%SizeMat  = nodeovnbr ! = NpointLoc
        WRITE(6,*) rang, "matrix : size, Nnz, Nin", MatCSR%SizeMat, MatCSR%Nnz, MatCSR%
Nin

        ! Print cpu time and matrix size of each proc in output file
        ALLOCATE(matsize(0:nproc-1), matnin(0:nproc-1), matnnz(0:nproc-1))
        matsize = 0
        matnin  = 0
        matnnz  = 0
        CALL MPI_GATHER(MatCSR%SizeMat, 1, MPI_INTEGER, matsize, 1, MPI_INTEGER
, 0, MPI_COMM_WORLD, code)
        CALL MPI_GATHER(MatCSR%Nin,     1, MPI_INTEGER, matnin,  1, MPI_INTEGER
, 0, MPI_COMM_WORLD, code)
        CALL MPI_GATHER(MatCSR%Nnz,     1, MPI_INTEGER, matnnz,  1, MPI_INTEGER
, 0, MPI_COMM_WORLD, code)
        IF (RANG == 0) THEN
      WRITE(outunit,*)
      WRITE(outunit,*) "Collect Data on distributed CSR matrix"
      WRITE(outunit,*)
      WRITE(outunit,*) "   proc rank / size  /   Nin   /   Nnz"
      DO np = 0, nproc-1
        WRITE(outunit,*) np, matsize(np), matnin(np), matnnz(np)
      END DO
      WRITE(outunit,*)
      WRITE(outunit,*) "min =", minval(matsize), minval(matnin), minval(matnnz)
      WRITE(outunit,*) "max =", maxval(matsize), maxval(matnin), maxval(matnnz)
      WRITE(outunit,*) "mid =", sum(matsize)/nproc, sum(matnin)/nproc, sum(matnnz
)/nproc
      WRITE(outunit,*)
      FLUSH(outunit)
        END IF
        DEALLOCATE(matsize, matnin, matnnz)

        ALLOCATE(MatCSR%Vals(MatCSR%Nnz), MatCSR%Diag(MatCSR%SizeMat), STAT = ie
rr )
        IF (ierr /= 0) THEN
      WRITE(6,*) " ERROR : Unable to allocate CSR matrix", MatCSR%SizeMat, MatCSR%Nnz
      STOP
        END IF
        MatCSR%Vals(: ) = 0.
        MatCSR%Diag(: ) = 0.

        CALL Cpu_time(temps_fin)
        temps_fin = temps_fin - temps_deb
        WRITE(6,*) rang, "Temps construction matrice :", temps_fin
        flush(6)

        WRITE(6,FMT=iofmt)       "<- ", rang, mod_name

  END SUBROUTINE InitializeMatrixCSR

  SUBROUTINE InitSol()
        CHARACTER(LEN = *), PARAMETER             :: mod_name = "InitSol"
        INTEGER                                   :: ierr
        INTEGER(KIND=PAMPAF_NUM)                  :: nodenbr, segfrnbr
        INTEGER(KIND=PAMPAF_NUM)                  :: nodeovnbr, segfrovnbr
        INTEGER(KIND=PAMPAF_NUM)                  :: nodeghstnbr, segfrghstnbr
        INTEGER                                   :: ilog, ifr, is, nbnghb, ire
f, iproc
        TYPE(PAMPAF_Iterator)                     :: it_vrt, it_ngb

        WRITE(6,FMT=iofmt)       "-> ", rang, mod_name
        CALL PAMPAF_dmeshEnttSize(dm, PAMPA_ENTITY_NODE, dummy,    nodenbr,  node
ovnbr,  nodeghstnbr,  retval = ierr)
        CALL PAMPAF_dmeshEnttSize(dm, PAMPA_ENTITY_BNDEDGE, dummy, segfrnbr, segf
rovnbr, segfrghstnbr, retval = ierr)

        ! Ua : Initial solution inside the domain
        ! Associate Ua to PAMPA nodes entity
        CALL PAMPAF_dmeshvaluelink(dm,                                        &
          &                        valuloctab = Ua,
 &
          &                        flagval = PAMPAF_VALUE_PUBLIC,
   &
          &                        typeval    = type_real,
 &
          &                        enttnum    = PAMPA_ENTITY_NODE,
 &
          &                        tagnum     = INT(PAMPA_TAG_SOL, PAMPAF_NUM)
, &
          &                        retval     = ierr)

        Ua = 0.


        ! Retrieve solution of overlap nodes

        CALL PAMPAF_dmeshHaloValue(dm,                                        &
          &                        enttnum = PAMPA_ENTITY_NODE,
 &
          &                        tagnum  = INT(PAMPA_TAG_SOL, PAMPAF_NUM),
 &
          &                        retval  = ierr)
        WRITE(6,FMT=iofmt)       "<- ", rang, mod_name

  END SUBROUTINE InitSol

  ! Calcul des Gradients de Fonctions de Base
  SUBROUTINE GradPhi(s1, s2, s3, GrdPhi )
        CHARACTER(LEN = *),     PARAMETER   :: mod_name = "GradPhi"
        REAL, DIMENSION(: ),    INTENT(IN)  :: s1, s2, s3
        REAL, DIMENSION(:, : ), INTENT(OUT) :: GrdPhi
        REAL                                :: x12, y12, x13, y13, x23, y23
        REAL                                :: Airt
        x13         = s1(1) - s3(1)
        y13         = s1(2) - s3(2)
        x12         = s1(1) - s2(1)
        y12         = s1(2) - s2(2)
        x23         = s2(1) - s3(1)
        y23         = s2(2) - s3(2)

        GrdPhi(1, 1) = s2(2) - s3(2)
        GrdPhi(2, 1) = s3(1) - s2(1)
        GrdPhi(1, 2) = s3(2) - s1(2)
        GrdPhi(2, 2) = s1(1) - s3(1)
        GrdPhi(1, 3) = s1(2) - s2(2)
        GrdPhi(2, 3) = s2(1) - s1(1)
        Airt = Abs( x13 * y23 - x23 * y13 )
        IF (Airt == 0.) THEN
      WRITE(6,*) rang, mod_name, ": Airt=0 : x13=", x13," y23=", y23 ," x23=", x23, " y
13=", y13
      WRITE(6,*) rang, mod_name, ": s1=", s1," s2=", s2 ," s3=", s3
      STOP
        ELSE
      GrdPhi = GrdPhi / Airt
        END IF
  END SUBROUTINE GradPhi

  SUBROUTINE GradPhi3d(x, GrdPhi)
    CHARACTER(LEN = *),     PARAMETER           :: mod_name = "GradPhi3d"
    REAL, DIMENSION(:, : ), INTENT(IN)          :: x
    REAL, DIMENSION(:, : ), INTENT(OUT) :: GrdPhi
    REAL, DIMENSION(Size(x, dim = 1), Size(x, dim = 2)) :: GrdPhi
    REAL :: z12, z13, z14, z23, z24, z34, y12, y13, y14, y23, y24, y34, Volt

    z12 = x(3,2) - x(3,1)
    y12 = x(2,2) - x(2,1)
    z13 = x(3,3) - x(3,1)
    y13 = x(2,3) - x(2,1)
    z14 = x(3,4) - x(3,1)
    y14 = x(2,4) - x(2,1)
    z23 = x(3,3) - x(3,2)
    y23 = x(2,3) - x(2,2)
    z24 = x(3,4) - x(3,2)
    y24 = x(2,4) - x(2,2)
    z34 = x(3,4) - x(3,3)
    y34 = x(2,4) - x(2,3)

    GrdPhi(1,1) =   x(2,2) * z34 - x(2,3) * z24 + x(2,4) * z23
    GrdPhi(1,2) = - x(2,1) * z34 + x(2,3) * z14 - x(2,4) * z13
    GrdPhi(1,3) =   x(2,1) * z24 - x(2,2) * z14 + x(2,4) * z12
    GrdPhi(1,4) = - x(2,1) * z23 + x(2,2) * z13 - x(2,3) * z12

    GrdPhi(2,1) = - x(1,2) * z34 + x(1,3) * z24 - x(1,4) * z23
    GrdPhi(2,2) =   x(1,1) * z34 - x(1,3) * z14 + x(1,4) * z13
    GrdPhi(2,3) = - x(1,1) * z24 + x(1,2) * z14 - x(1,4) * z12
    GrdPhi(2,4) =   x(1,1) * z23 - x(1,2) * z13 + x(1,3) * z12

    GrdPhi(3,1) =   x(1,2) * y34 - x(1,3) * y24 + x(1,4) * y23
    GrdPhi(3,2) = - x(1,1) * y34 + x(1,3) * y14 - x(1,4) * y13
    GrdPhi(3,3) =   x(1,1) * y24 - x(1,2) * y14 + x(1,4) * y12
    GrdPhi(3,4) = - x(1,1) * y23 + x(1,2) * y13 - x(1,3) * y12

    Volt = Abs(x(1,1) * GrdPhi(1,1) + x(1,2) * GrdPhi(1,2) + x(1,3) * GrdPhi(1,3
) + x(1,4) * GrdPhi(1,4))

    IF (Volt == 0.) THEN
      WRITE(6,*) rang, mod_name, " ERROR : Volt=0"
      STOP
    ELSE
      GrdPhi = GrdPhi / Volt
    END IF

  END SUBROUTINE GradPhi3d

  SUBROUTINE FillMatrix()
        CHARACTER(LEN = *),             PARAMETER :: mod_name = "FillMatrix"
        INTEGER,                        PARAMETER :: Nsmplx = 4 !3
        INTEGER, DIMENSION(Nsmplx)                :: NuElemt
        REAL,    DIMENSION(Ndim, Nsmplx)          :: CoorElemt
        REAL   , DIMENSION(Ndim,Nsmplx),  TARGET  :: GrdPhi
        REAL                                      :: JJac, Volt, normale
        REAL                                      :: temps_fin, temps_deb
        INTEGER                                   :: i, j, is, jt, js, ifr, ilog
        INTEGER                                   :: iref, iproc, ierr, ngb
        INTEGER(KIND=PAMPAF_NUM)                  :: nodenbr, nodeovnbr, nodeghst
nbr
        TYPE(PAMPAF_Iterator)                     :: it_vrt, it_ngb
        WRITE(6,FMT=iofmt) "-> ", rang, mod_name
        CALL PAMPAF_dmeshEnttSize(dm, PAMPA_ENTITY_NODE, dummy,    nodenbr,  node
ovnbr,  nodeghstnbr,  retval = ierr)
        CALL Cpu_time(temps_deb)

        CALL PAMPAF_dmeshvaluelink(dm,
```

```
 &
         &                          valuloctab = RHS,
 &
         &                          flagval    = PAMPAF_VALUE_PUBLIC,
 &
         &                          typeval    = type_real,
 &
         &                          enttnum    = PAMPA_ENTITY_NODE,
 &
         &                          tagnum     = INT(PAMPA_TAG_RHS, PAMPAF_NUM)
, &
         &                          retval     = ierr)

        RHS = 0.

        CALL PAMPAF_dmeshItInitStart(dm,  PAMPA_ENTITY_ELMT,  PAMPAF_VERT_ANY,    i
t_vrt,  ierr)
        CALL PAMPAF_dmeshItInit     (dm,  PAMPA_ENTITY_ELMT,  PAMPA_ENTITY_NODE,
it_ngb,  ierr)

        DO WHILE (PAMPAF_itHasMore(it_vrt))
      jt   = PAMPAF_itCurEnttVertNum(it_vrt)
      Volt = VolEl(jt)

      ngb  = 0
      CALL PAMPAF_itStart(it_ngb, jt, ierr)
      DO WHILE (PAMPAF_itHasMore(it_ngb))
        ngb = ngb + 1
        is  = PAMPAF_itCurEnttVertNum(it_ngb)
        NuElemt(ngb)     = is
        CoorElemt(:,ngb) = Coor(:,is)
        PAMPAF_itNext(it_ngb)
      END DO
      IF (ngb /= Nsmplx) THEN
        CALL PAMPAF_dmeshvertdata(dm, jt, PAMPA_ENTITY_ELMT, PAMPA_ENTITY_NODE, n
gb, iref, iproc, ierr)
        WRITE(6,*) rang, mod_name, " PB with element connectivity :", jt, iref, ngb
        STOP
          END IF

          ! Gradient of basis functions on current triangle element
          !CALL GradPhi(CoorElemt(:,1), CoorElemt(:,2), CoorElemt(:,3), GrdPhi)
! No need to store GrdPhi
      CALL GradPhi3d(CoorElemt, GrdPhi) ! No need to store GrdPhi

          ! Fulfill local matrix
          ! -------------------
          DO i = 1, Nsmplx
                is = NuElemt(i)
                DO j = 1, Nsmplx
          js      = NuElemt(j)
          JJac    = Volt * Sum(GrdPhi(:,i)*GrdPhi(:,j))

          CALL assemblage_addCSR(JJac,is,js)
                END DO ! loop on j

       !RHS(is) = RHS(is) - Volt * SourceTerm(Coor(1,is), Coor(2,is)) / Nsmplx
                RHS(is) = RHS(is) - Volt * SourceTerm(Coor(1,is), Coor(2,is), Co
or(3,is)) / Nsmplx

          END DO ! loop on i

      PAMPAF_itNext(it_vrt)

        END DO
        CALL PAMPAF_dmeshItExit(it_vrt, ierr)
        CALL PAMPAF_dmeshItExit(it_ngb, ierr)

        ! Take into account boundary conditions
        !--------------------------------------
        ! Hyp : All boundaries are Dirichlet conditions

        CALL PAMPAF_dmeshItInitStart(dm,  PAMPA_ENTITY_BNDEDGE,  PAMPAF_VERT_ANY,
 it_vrt,  ierr)
        CALL PAMPAF_dmeshItInit     (dm,  PAMPA_ENTITY_BNDEDGE,  PAMPA_ENTITY_NODE
,  it_ngb,  ierr)

        DO WHILE (PAMPAF_itHasMore(it_vrt))
      ifr   = PAMPAF_itCurEnttVertNum(it_vrt)
      ilog  = LogSegFr(ifr)

      ngb  = 0
      CALL PAMPAF_itStart(it_ngb, ifr, ierr)
      DO WHILE (PAMPAF_itHasMore(it_ngb))
        ngb = ngb + 1
        is = PAMPAF_itCurEnttVertNum(it_ngb)
        SELECT CASE(ilog)
        CASE DEFAULT ! dirichlet anywhere
          CALL assemblage_setidentblokline(is)
          !RHS(is) = DirichletValue(Coor(1,is), Coor(2,is))
          RHS(is) = DirichletValue(Coor(1,is), Coor(2,is), Coor(3,is))
        END SELECT

        PAMPAF_itNext(it_ngb)
      END DO
      IF (ngb /= Nsmplx - 1) THEN
        CALL PAMPAF_dmeshvertdata(dm, ifr, PAMPA_ENTITY_BNDEDGE, PAMPA_ENTITY_NOD
E, ngb, iref, iproc, ierr)
        WRITE(6,*) mod_name, " PB with element connectivity :", jt, iref, ngb
        STOP
          END IF

      PAMPAF_itNext(it_vrt)
        END DO
        CALL PAMPAF_dmeshItExit(it_vrt, ierr)
        CALL PAMPAF_dmeshItExit(it_ngb, ierr)

        ! Retrieve RHS of overlap nodes

        CALL PAMPAF_dmeshHaloValue(dm,                                          &
          &                        enttnum = PAMPA_ENTITY_NODE,                 &
          &                        tagnum  = INT(PAMPA_TAG_RHS, PAMPAF_NUM), &
          &                        retval  = ierr)

        CALL Cpu_time(temps_fin)
        temps_fin = temps_fin - temps_deb
        WRITE(6,*) rang, "Temps remplissage matrice :", temps_fin
        flush(6)
        WRITE(6,FMT=iofmt)       "<- ", rang, mod_name

  END SUBROUTINE FillMatrix

  REAL FUNCTION SourceTerm(x, y, z)
        REAL :: x, y
    REAL, OPTIONAL :: z
        SourceTerm = -sin(z)
        !SourceTerm = -2 * cos(x) * cos(y)
  END FUNCTION SourceTerm


  REAL FUNCTION DirichletValue(x, y, z)
        REAL :: x, y
    REAL, OPTIONAL :: z
        DirichletValue = sin(z)
        !DirichletValue = cos(x) * cos(y)
  END FUNCTION DirichletValue
```

```
  !!-- Addition d'une matrice elementaire, avec une recherche lineaire d'indice
qui pourrait etre factorisee (%%%%????)
  !!-- (si le maillage, meme deformable, ne change pas de structure).
  SUBROUTINE assemblage_addCSR(Matelem, Iin, Jin)
        CHARACTER(LEN = *),              PARAMETER       :: mod_name = "assemblage
_addCSR"
        INTEGER,                         INTENT( IN)     :: Iin, Jin
        REAL,                            INTENT( IN)     :: Matelem
        INTEGER                                          :: imat,ierr

        IF (Iin == Jin) THEN
      MatCSR%Diag(Iin) = MatCSR%Diag(Iin) + Matelem

        ELSE
      CALL PAMPAF_dmeshMatVertData(dm, PAMPA_ENTITY_NODE, Iin, Jin, Imat, ierr)
      MatCSR%Vals(Imat) = MatCSR%Vals(Imat) + Matelem

        END IF

  END SUBROUTINE assemblage_addCSR

  !!-- mise a Id du bloc diag et a zero du reste de la ligne
  SUBROUTINE assemblage_setidentblokline(Iin)
        CHARACTER(LEN = *), PARAMETER  :: mod_name = "assemblage_setidentblokline"
        INTEGER,            INTENT(IN) :: Iin
        INTEGER                        :: i1, IFin,statut

        MatCSR%Diag(Iin) = 1.0

        CALL PAMPAF_dmeshMatLineData(dm, PAMPA_ENTITY_NODE, Iin, I1, IFin, statu
t)
        MatCSR%Vals(i1: IFin) = 0.0

  END SUBROUTINE assemblage_setidentblokline

  SUBROUTINE SolveSystem()
        CHARACTER(LEN = *),           PARAMETER :: mod_name = "SolveSystem"
        INTEGER                                 :: istat, np
        REAL                                    :: temps_fin, temps_deb
        REAL,    DIMENSION(:),     ALLOCATABLE :: cput

        WRITE(6,FMT=iofmt)       "-> ", rang, mod_name

        resdfile = "Residus_" // nprocstr // ".dat"
        resdunit = 21

        OPEN( UNIT = resdunit, FILE = resdfile, IOSTAT = istat, STATUS = "REPLA
CE" )
        IF (istat /= 0) THEN
      PRINT *, " ERROR : Unable to open ", resdfile, " -> ", istat
      STOP
        END IF
        REWIND(resdunit)

        WRITE(6,*)
        WRITE(6,*) "***********************************"
        WRITE(6,*) rang, "Resolution du systeme ... DEB"

        CALL Cpu_time(temps_deb)
        CALL Jacobi()
        !CALL GaussSeidel()
        CALL Cpu_time(temps_fin)
        temps_fin = temps_fin - temps_deb

        WRITE(6,*) rang, "Resolution du systeme ... FIN"
        WRITE(6,*) rang, "Temps solve system:", temps_fin
        WRITE(6,*) "***********************************"
        WRITE(6,*)
        FLUSH(6)

        CLOSE(resdunit)

        ! Print cpu time of each proc in output file
        ALLOCATE(cput(0:nproc-1))
        cput    = 0.
        CALL MPI_GATHER(temps_fin,      1, type_real,   cput,    1, type_real,
  0, MPI_COMM_WORLD, code)
        IF (RANG == 0) THEN
      WRITE(outunit,*)
      WRITE(outunit,*) "Collect Data on Solve System"
      WRITE(outunit,*)
      WRITE(outunit,*) " proc rank  / cpu time"
      DO np = 0, nproc-1
        WRITE(outunit,*) np, cput(np)
      END DO
      WRITE(outunit,*)
      WRITE(outunit,*) "min =", minval(cput)
      WRITE(outunit,*) "max =", maxval(cput)
      WRITE(outunit,*) "mid =", sum(cput)/nproc
      WRITE(outunit,*) "tot =", sum(cput)
      WRITE(outunit,*)
      FLUSH(outunit)
        END IF
        DEALLOCATE(cput)
        WRITE(6,FMT=iofmt)       "<- ", rang, mod_name

  END SUBROUTINE SolveSystem

  SUBROUTINE Jacobi()
    CHARACTER(LEN = *),          PARAMETER :: mod_name = "Jacobi"
    INTEGER,                     PARAMETER :: Nrelax = 25000000
    REAL,                        PARAMETER :: Resdlin = 1.E-8 !1.E-12
    INTEGER                                :: is, irelax
    INTEGER                                :: js, iv, i1, I1Fin
    REAL                                   :: deltaU, deltaUglb, res, res0, resg
lb, res0glb
    REAL, DIMENSION(size(Ua, dim=1))       :: UaPrec
    INTEGER                                :: statut
    TYPE(PAMPAF_Iterator)                  :: it_vrt, it_ngb
    REAL(PAMPAF_DOUBLE),DIMENSION(PAMPAF_DMESHHALOREQSDIM)   :: req

    UaPrec = 0.        ! Suppose A = L + D + U, system to solve : A x = b
    CALL PAMPAF_dmeshItInit(dm,  PAMPA_ENTITY_NODE,  PAMPA_ENTITY_NODE,  it_ngb,
statut)
    DO irelax = 1, Nrelax
      res       = 0.
      CALL PAMPAF_dmeshItInitStart(dm,  PAMPA_ENTITY_NODE,  PAMPAF_VERT_BOUNDARY,
it_vrt,  statut)
      DO WHILE (PAMPAF_itHasMore(it_vrt))
        is = PAMPAF_itCurEnttVertNum(it_vrt)
        CALL PAMPAF_dmeshMatLineData(dm, PAMPA_ENTITY_NODE, is, I1, I1Fin, statu
t)
        CALL PAMPAF_itStart( it_ngb,  is,  statut)
        res0    = RHS(is)                              ! res0  = b

        iv = i1
        DO WHILE (PAMPAF_itHasMore(it_ngb))
          js = PAMPAF_itCurEnttVertNum(it_ngb)
          PAMPAF_itNext(it_ngb)
          res0  = res0 - MatCSR%Vals(iv) * UaPrec(js) !res0  = b - (L + U) x^n
          iv = iv + 1
        END DO
        Ua(is)  = res0 / MatCSR%Diag(is)              !x^n+1 = ( b - (L + U) x^n
 )/D
        res0    = res0 - MatCSR%Diag(is) * UaPrec(is) !res0  = ( b - (L + D + U)
 x^n )
        res0    = res0 * res0                         !res0  = ( b - (L + D + U)
```

```
 x^n )**2
        res     = res  + res0
        PAMPAF_itNext(it_vrt)
      END DO
      CALL PAMPAF_dmeshItExit(it_vrt,  statut)

      CALL PAMPAF_dmeshHaloValueAsync(dm, PAMPA_ENTITY_NODE, PAMPA_TAG_SOL, req,
 statut)

      CALL PAMPAF_dmeshItInitStart(dm,  PAMPA_ENTITY_NODE,  PAMPAF_VERT_INTERNAL,
it_vrt,  statut)
      DO WHILE (PAMPAF_itHasMore(it_vrt))
        is = PAMPAF_itCurEnttVertNum(it_vrt)
        CALL PAMPAF_dmeshMatLineData(dm, PAMPA_ENTITY_NODE, is, I1, I1Fin, statu
t)
        CALL PAMPAF_itStart( it_ngb,  is,  statut)
        res0    = RHS(is)                              ! res0  = b

        iv = i1
        DO WHILE (PAMPAF_itHasMore(it_ngb))
          js = PAMPAF_itCurEnttVertNum(it_ngb)
          PAMPAF_itNext(it_ngb)
          res0  = res0 - MatCSR%Vals(iv) * UaPrec(js) !res0  = b - (L + U) x^n
          iv = iv + 1
        END DO
        Ua(is)  = res0 / MatCSR%Diag(is)              !x^n+1 = ( b - (L + U) x^n
 )/D
        res0    = res0 - MatCSR%Diag(is) * UaPrec(is) !res0  = ( b - (L + D + U)
 x^n )
        res0    = res0 * res0                         !res0  = ( b - (L + D + U)
 x^n )**2
        res     = res  + res0
        PAMPAF_itNext(it_vrt)
      END DO
      CALL PAMPAF_dmeshItExit(it_vrt,  statut)

      CALL PAMPAF_dmeshHaloWait(req, statut)

      ! Compute global residu
      deltaU    = 0.
      deltaU    = DOT_PRODUCT( UaPrec(:MatCSR%Nin) - Ua(:MatCSR%Nin) , UaPrec(:M
atCSR%Nin) - Ua(:MatCSR%Nin) )

      deltaUglb = 0.
      CALL MPI_ALLREDUCE(deltaU, deltaUglb, 1,type_real,MPI_SUM,MPI_COMM_WORLD,
code)
      deltaUglb = sqrt(deltaUglb)

      resglb    = 0.
      CALL MPI_ALLREDUCE(res, resglb, 1, type_real, MPI_SUM, MPI_COMM_WORLD, co
de)
      resglb    = sqrt(resglb)

      res0      = DOT_PRODUCT( RHS   (:MatCSR%Nin)                  , RHS   (:M
atCSR%Nin)                  )
      res0glb   = 0.
      CALL MPI_ALLREDUCE(res0, res0glb, 1, type_real, MPI_SUM, MPI_COMM_WORLD,
code)
      res0glb   = sqrt(res0glb)

      resglb    = resglb / res0glb

      if (isnan(deltauglb)) exit
      if (deltauglb>huge(1.0)) exit

      IF ((rang == 0) .AND. (mod(irelax,1000) == 0)) THEN
        WRITE(resdunit,*) irelax, deltaUglb, log(deltaUglb), resglb, log(resglb)
        FLUSH(resdunit)
      END IF

      IF (deltaUglb < Resdlin) THEN ! .OR. resglb < Resdlin) THEN
        IF (rang == 0) THEN
          WRITE(resdunit,*) irelax, deltaUglb, log(deltaUglb), resglb, log(resgl
b)
          FLUSH(resdunit)
        END IF
        EXIT
      END IF

      UaPrec    = Ua

    END DO ! end loop on irelax
    CALL PAMPAF_dmeshItExit(it_ngb,  statut)

    IF (irelax >= Nrelax) THEN
      WRITE(6,*) rang, mod_name, " max number of iterations reached :", Nrelax, "delta U =", d
eltaU, deltaUglb, " residu =", res, resglb
    END IF

    IF (rang == 0) THEN
      WRITE(6,*) rang, mod_name, " Exit after",irelax, "iterations. Delta U =", deltaUglb,
"Residu =", resglb
    END IF

  END SUBROUTINE Jacobi

  SUBROUTINE GaussSeidel()
        CHARACTER(LEN = *),          PARAMETER :: mod_name = "GaussSeidel"
        INTEGER,                     PARAMETER :: Nrelax = 25000000
        REAL,                        PARAMETER :: Resdlin = 1.E-8 !1.E-12
        INTEGER                                :: is, irelax
        INTEGER                                :: js, iv, i1, I1Fin
        REAL                                   :: deltaU, deltaUglb, res, res0,
res1, res2, resglb, res0glb
        REAL, DIMENSION(size(Ua, dim=1))       :: UaPrecGS
        INTEGER                                :: statut
        TYPE(PAMPAF_Iterator)                  :: it_vrt, it_ngb
    REAL(PAMPAF_DOUBLE),DIMENSION(PAMPAF_DMESHHALOREQSDIM)   :: req

        UaPrecGS = 0.
      CALL PAMPAF_dmeshItInit(dm,  PAMPA_ENTITY_NODE,  PAMPA_ENTITY_NODE,  it_ngb,
  statut)
        DO irelax = 1, Nrelax

          res       = 0.
      CALL PAMPAF_dmeshItInitStart(dm,  PAMPA_ENTITY_NODE,  PAMPAF_VERT_BOUNDARY,
it_vrt,  statut)
      DO WHILE (PAMPAF_itHasMore(it_vrt))
      is = PAMPAF_itCurEnttVertNum(it_vrt)
        CALL PAMPAF_dmeshMatLineData(dm, PAMPA_ENTITY_NODE, is, I1, I1Fin, statu
t)
        CALL PAMPAF_itStart( it_ngb,  is,  statut)
              res0     = RHS(is)                              ! res0  = b
              res1     = MatCSR%Diag(is) * UaPrecGS(is)       ! res1  = D x^
n
              res2     = 0.

              iv = i1
              DO WHILE (PAMPAF_itHasMore(it_ngb))
          js = PAMPAF_itCurEnttVertNum(it_ngb)
          PAMPAF_itNext(it_ngb)
          IF (js < is) THEN
            res2 = res2 + MatCSR%Vals(iv) * Ua(js)        ! res2 += L x^n+1
            res1 = res1 + MatCSR%Vals(iv) * UaPrecGS(js)  ! res1 += L x^n   => r
es1 = (D+L) x^n
          ELSE
            res0 = res0 - MatCSR%Vals(iv) * UaPrecGS(js)  ! res0 += - U x^n => r
es0 = b - U x^n
          END IF
                iv = iv + 1
              END DO
              Ua(is)   = (res0 - res2) / MatCSR%Diag(is)        ! x^n+1 = D^-1
 (b - U x^n - L x^n+1)
              res0     = res0 - res1                            ! res0  = b -
U x^n - (D+L) x^n = b - A x^n
              res0     = res0 * res0
              res      = res  + res0
        PAMPAF_itNext(it_vrt)
        END DO

        CALL PAMPAF_dmeshHaloValueAsync(dm, PAMPA_ENTITY_NODE, PAMPA_TAG_SOL,
req, statut)

      CALL PAMPAF_dmeshItInitStart(dm,  PAMPA_ENTITY_NODE,  PAMPAF_VERT_INTERNAL,
it_vrt,  statut)
      DO WHILE (PAMPAF_itHasMore(it_vrt))
      is = PAMPAF_itCurEnttVertNum(it_vrt)
        CALL PAMPAF_dmeshMatLineData(dm, PAMPA_ENTITY_NODE, is, I1, I1Fin, statu
t)
        CALL PAMPAF_itStart( it_ngb,  is,  statut)
              res0     = RHS(is)                              ! res0  = b
              res1     = MatCSR%Diag(is) * UaPrecGS(is)       ! res1  = D x^
n
              res2     = 0.

              iv = i1
              DO WHILE (PAMPAF_itHasMore(it_ngb))
          js = PAMPAF_itCurEnttVertNum(it_ngb)
          PAMPAF_itNext(it_ngb)
          IF (js < is) THEN
            res2 = res2 + MatCSR%Vals(iv) * Ua(js)        ! res2 += L x^n+1
            res1 = res1 + MatCSR%Vals(iv) * UaPrecGS(js)  ! res1 += L x^n   => r
es1 = (D+L) x^n
          ELSE
            res0 = res0 - MatCSR%Vals(iv) * UaPrecGS(js)  ! res0 += - U x^n => r
es0 = b - U x^n
          END IF
                iv = iv + 1
              END DO
              Ua(is)   = (res0 - res2) / MatCSR%Diag(is)        ! x^n+1 = D^-1
 (b - U x^n - L x^n+1)
              res0     = res0 - res1                            ! res0  = b -
U x^n - (D+L) x^n = b - A x^n
              res0     = res0 * res0
              res      = res  + res0
        PAMPAF_itNext(it_vrt)
        END DO

      CALL PAMPAF_dmeshHaloWait(req, statut)

      ! Compute global residu
          deltaU    = 0.
          deltaU    = DOT_PRODUCT( UaPrecGS( : MatCSR%Nin) - Ua( : MatCSR%Nin) ,
 UaPrecGS( : MatCSR%Nin) - Ua( : MatCSR%Nin) )

          deltaUglb = 0.
          CALL MPI_ALLREDUCE(deltaU, deltaUglb, 1, type_real, MPI_SUM, MPI_COMM
_WORLD, code)
          deltaUglb = sqrt(deltaUglb)

          resglb    = 0.
          CALL MPI_ALLREDUCE(res, resglb, 1, type_real, MPI_SUM, MPI_COMM_WORLD
, code)
          resglb    = sqrt(resglb)

      res0      = DOT_PRODUCT( RHS     ( : MatCSR%Nin)                    , RHS
     ( : MatCSR%Nin)                    )
          res0glb   = 0.
          CALL MPI_ALLREDUCE(res0, res0glb, 1, type_real, MPI_SUM, MPI_COMM_WOR
LD, code)
          res0glb   = sqrt(res0glb)

          resglb    = resglb / res0glb

          IF ((rang == 0) .AND. (mod(irelax,1000) == 0)) THEN
        WRITE(resdunit,*) irelax, deltaUglb, log(deltaUglb), resglb, log(resglb)
          END IF

          !write(6,*),rang,resglb,deltauglb
          if (isnan(deltauglb)) exit
          if (deltauglb>huge(1.0)) exit
          IF (deltaUglb < Resdlin .OR. resglb < Resdlin) THEN
        IF (rang == 0) THEN
          WRITE(resdunit,*) irelax, deltaUglb, log(deltaUglb), resglb, log(resgl
b)
          FLUSH(resdunit)
        END IF
        EXIT
          END IF

      UaPrecGS  = Ua

        END DO ! end loop on irelax

        IF (irelax >= Nrelax) THEN
      WRITE(6,*) rang, mod_name, " max number of iterations reached :", Nrelax, "delta U =", d
eltaU, deltaUglb, " residu =", res, resglb
        END IF
        IF (rang == 0) THEN
      WRITE(6,*) rang, mod_name, " Exit after",irelax, "iterations. Delta U =", deltaUglb,
"Residu =", resglb
        END IF

  END SUBROUTINE GaussSeidel

  SUBROUTINE WriteDistributedMeshAndSolFiles()
        CHARACTER(LEN = *),           PARAMETER  :: mod_name = "WriteDistributedMesh
AndSolFiles"
        CHARACTER(LEN = 13)                      :: Ascfile
        INTEGER                                  :: mshUnit, ifr, is, jt, tag, i
stat, ierr
        INTEGER(KIND=PAMPAF_NUM)                 :: nodenbr, elmtnbr, segfrnbr
        INTEGER(KIND=PAMPAF_NUM)                 :: nodeovnbr, elmtovnbr, segfro
vnbr
        INTEGER(KIND=PAMPAF_NUM)                 :: nodeghstnbr, elmtghstnbr, se
gfrghstnbr
        TYPE(PAMPAF_Iterator)                    :: it_vrt, it_ngb

        CALL PAMPAF_dmeshEnttSize(dm, PAMPA_ENTITY_NODE, dummy,    nodenbr,  node
ovnbr,  nodeghstnbr,  retval = ierr)
        CALL PAMPAF_dmeshEnttSize(dm, PAMPA_ENTITY_ELMT, dummy,    elmtnbr,  elmt
ovnbr,  elmtghstnbr,  retval = ierr)
        CALL PAMPAF_dmeshEnttSize(dm, PAMPA_ENTITY_BNDEDGE, dummy, segfrnbr, segf
rovnbr, segfrghstnbr, retval = ierr)

        mshUnit = 21

        Ascfile = Trim(Adjustl("Mesh_" // rangstr // ".mesh"))

        OPEN( UNIT = mshUnit, FILE = Trim(Adjustl(Ascfile)), IOSTAT = istat, STA
TUS = "REPLACE" )
        IF (istat /= 0) THEN
      PRINT *, " ERROR : Unable to open ", Trim(Ascfile), " -> ", istat
      STOP
```

```
        END IF
        REWIND(mshUnit)

        ! Write SubMesh.msh file following medit format
        WRITE(mshUnit,FMT='(A)') "MeshVersionFormatted 2"
        WRITE(mshUnit,FMT='(A)') "Dimension"
        WRITE(mshUnit,FMT='(A)') "2"

        WRITE(mshUnit,FMT='(A)') "Vertices"
        WRITE(mshUnit,FMT='(I8)') nodeovnbr
    ! Do not use iterator first level here because nodes are not treated in the
same order than their index
    ! PaMPA reorganized node indexes in the following order: internal, local bou
ndary, overlap and halo
        DO is = 1, nodeovnbr
          tag = 0
          WRITE(mshUnit,FMT='(2(F12.8,1X),I2)') Coor(1,is), Coor(2,is), tag
        END DO

        WRITE(mshUnit,FMT='(A)') "Edges"
        WRITE(mshUnit,FMT='(I8)') segfrovnbr
        CALL PAMPAF_dmeshItInitStart(dm,  PAMPA_ENTITY_BNDEDGE,  PAMPAF_VERT_ANY,
  it_vrt,  ierr)
        CALL PAMPAF_dmeshItInit     (dm,  PAMPA_ENTITY_BNDEDGE,  PAMPA_ENTITY_NODE
,  it_ngb,  ierr)
        DO WHILE (PAMPAF_itHasMore(it_vrt))
      ifr   = PAMPAF_itCurEnttVertNum(it_vrt)
      CALL PAMPAF_itStart(it_ngb, ifr, ierr)
      DO WHILE (PAMPAF_itHasMore(it_ngb))
        is= PAMPAF_itCurEnttVertNum(it_ngb)
        WRITE(mshUnit,FMT='(I8,1X)', ADVANCE="NO") is
        PAMPAF_itNext(it_ngb)
      END DO
      WRITE(mshUnit,FMT='(I8)') LogSegFr(ifr)
      PAMPAF_itNext(it_vrt)
        END DO

        WRITE(mshUnit,FMT='(A)') "Triangles"
        WRITE(mshUnit,FMT='(I8)') elmtovnbr
        CALL PAMPAF_dmeshItInitStart(dm,  PAMPA_ENTITY_ELMT,  PAMPAF_VERT_ANY,    i
t_vrt,  ierr)
        CALL PAMPAF_dmeshItInit     (dm,  PAMPA_ENTITY_ELMT,  PAMPA_ENTITY_NODE,
it_ngb,  ierr)
        DO WHILE (PAMPAF_itHasMore(it_vrt))
      jt   = PAMPAF_itCurEnttVertNum(it_vrt)
      CALL PAMPAF_itStart(it_ngb, jt, ierr)
      DO WHILE (PAMPAF_itHasMore(it_ngb))
        is = PAMPAF_itCurEnttVertNum(it_ngb)
        WRITE(mshUnit,FMT='(I8,1X)', ADVANCE="NO") is
        PAMPAF_itNext(it_ngb)
      END DO
      WRITE(mshUnit,FMT='(I8)') 1 ! dummy tag
      PAMPAF_itNext(it_vrt)
        END DO

        WRITE(mshUnit,FMT='(A)'), "End"

        CLOSE(mshUnit)


    !
    ! ... Write solution ... .sol
    !
    Ascfile = Trim(Adjustl("Mesh_" // rangstr // ".sol"))

    OPEN( UNIT = mshUnit, FILE = Trim(Adjustl(Ascfile)), IOSTAT = istat, STATUS
= "REPLACE" )
    IF (istat /= 0) THEN
      PRINT *, " ERROR : Unable to open ", Trim(Ascfile), " -> ", istat
      STOP
    END IF
    REWIND(mshUnit)

    WRITE(mshUnit,FMT='(A)') "MeshVersionFormatted 2"
    WRITE(mshUnit,FMT='(A)') "Dimension"
    WRITE(mshUnit,FMT='(A)') "2"
    WRITE(mshUnit,FMT='(A)') "SolAtVertices "
    WRITE(mshUnit,FMT='(I8)') nodeovnbr
    WRITE(mshUnit,FMT='(A)') "1 1"
    ! Do not use iterator first level here because nodes are not treated in the
same order than their index
    ! PaMPA reorganized node indexes in the following order: internal, local bou
ndary, overlap and halo
    DO is = 1, nodeovnbr
      WRITE(mshUnit,FMT='(F12.8)')  Ua(is)
    END DO
    WRITE (mshUnit, FMT='(A)') "End "
    CLOSE (mshUnit)
  END SUBROUTINE WriteDistributedMeshAndSolFiles


  SUBROUTINE WriteDistributedMeshAndSolFiles3d()
    CHARACTER(LEN = *),           PARAMETER  :: mod_name = "WriteDistributedMeshAndSo
lFiles3d"
    CHARACTER(LEN = 13)                      :: Ascfile
    INTEGER                                  :: mshUnit, ifr, is, jt, tag, istat
, ierr
    INTEGER(KIND=PAMPAF_NUM)                 :: nodenbr, elmtnbr, segfrnbr
    INTEGER(KIND=PAMPAF_NUM)                 :: nodeovnbr, elmtovnbr, segfrovnbr
    INTEGER(KIND=PAMPAF_NUM)                 :: nodeghstnbr, elmtghstnbr, segfrg
hstnbr
    TYPE(PAMPAF_Iterator)                    :: it_vrt, it_ngb

    CALL PAMPAF_dmeshEnttSize(dm, PAMPA_ENTITY_NODE, dummy,    nodenbr,  nodeovnb
r,  nodeghstnbr,  retval = ierr)
    CALL PAMPAF_dmeshEnttSize(dm, PAMPA_ENTITY_ELMT, dummy,    elmtnbr,  elmtovnb
r,  elmtghstnbr,  retval = ierr)
    CALL PAMPAF_dmeshEnttSize(dm, PAMPA_ENTITY_BNDEDGE, dummy, segfrnbr, segfrovn
br, segfrghstnbr, retval = ierr)

    mshUnit = 21

    Ascfile = Trim(Adjustl("Mesh_" // rangstr // ".mesh"))

    OPEN( UNIT = mshUnit, FILE = Trim(Adjustl(Ascfile)), IOSTAT = istat, STATUS
= "REPLACE" )
    IF (istat /= 0) THEN
      PRINT *, " ERROR : Unable to open ", Trim(Ascfile), " -> ", istat
      STOP
    END IF
    REWIND(mshUnit)

    ! Write SubMesh.msh file following medit format
    WRITE(mshUnit,FMT='(A)') "MeshVersionFormatted 2"
    WRITE(mshUnit,FMT='(A)') "Dimension"
    WRITE(mshUnit,FMT='(A)') "3"

    WRITE(mshUnit,FMT='(A)') "Vertices"
    WRITE(mshUnit,FMT='(I8)') nodeovnbr
    ! Do not use iterator first level here because nodes are not treated in the
same order than their index
    ! PaMPA reorganized node indexes in the following order: internal, local bou
ndary, overlap and halo
    DO is = 1, nodeovnbr
      tag = 0
      WRITE(mshUnit,FMT='(3(F12.8,1X),I2)') Coor(1,is), Coor(2,is), Coor(3,is), tag
    END DO

    WRITE(mshUnit,FMT='(A)') "Triangles"
    WRITE(mshUnit,FMT='(I8)') segfrovnbr
    CALL PAMPAF_dmeshItInitStart(dm,  PAMPA_ENTITY_BNDEDGE,  PAMPAF_VERT_ANY,    it
_vrt,  ierr)
    CALL PAMPAF_dmeshItInit     (dm,  PAMPA_ENTITY_BNDEDGE,  PAMPA_ENTITY_NODE,  i
t_ngb,  ierr)
    DO WHILE (PAMPAF_itHasMore(it_vrt))
      ifr   = PAMPAF_itCurEnttVertNum(it_vrt)
      CALL PAMPAF_itStart(it_ngb, ifr, ierr)
      DO WHILE (PAMPAF_itHasMore(it_ngb))
        is= PAMPAF_itCurEnttVertNum(it_ngb)
        WRITE(mshUnit,FMT='(I8,1X)', ADVANCE="NO") is
        PAMPAF_itNext(it_ngb)
      END DO
      WRITE(mshUnit,FMT='(I8)') LogSegFr(ifr)
      PAMPAF_itNext(it_vrt)
    END DO
    CALL PAMPAF_dmeshItExit(it_vrt,  ierr)
    CALL PAMPAF_dmeshItExit(it_ngb,  ierr)

    WRITE(mshUnit,FMT='(A)') "Tetrahedra"
    WRITE(mshUnit,FMT='(I8)') elmtovnbr
    CALL PAMPAF_dmeshItInitStart(dm,  PAMPA_ENTITY_ELMT,  PAMPAF_VERT_ANY,    it_vr
t,  ierr)
    CALL PAMPAF_dmeshItInit     (dm,  PAMPA_ENTITY_ELMT,  PAMPA_ENTITY_NODE,  it_n
gb,  ierr)
    DO WHILE (PAMPAF_itHasMore(it_vrt))
      jt   = PAMPAF_itCurEnttVertNum(it_vrt)
      CALL PAMPAF_itStart(it_ngb, jt, ierr)
      DO WHILE (PAMPAF_itHasMore(it_ngb))
        is = PAMPAF_itCurEnttVertNum(it_ngb)
        WRITE(mshUnit,FMT='(I8,1X)', ADVANCE="NO") is
        PAMPAF_itNext(it_ngb)
      END DO
      WRITE(mshUnit,FMT='(I8)') 1 ! dummy tag
      PAMPAF_itNext(it_vrt)
    END DO
    CALL PAMPAF_dmeshItExit(it_vrt,  ierr)
    CALL PAMPAF_dmeshItExit(it_ngb,  ierr)

    WRITE(mshUnit,FMT='(A)'), "End"

    CLOSE(mshUnit)


    !
    ! ... Write solution ... .sol
    !
    Ascfile = Trim(Adjustl("Mesh_" // rangstr // ".sol"))

    OPEN( UNIT = mshUnit, FILE = Trim(Adjustl(Ascfile)), IOSTAT = istat, STATUS
= "REPLACE" )
    IF (istat /= 0) THEN
      PRINT *, " ERROR : Unable to open ", Trim(Ascfile), " -> ", istat
      STOP
    END IF
    REWIND(mshUnit)

    WRITE(mshUnit,FMT='(A)') "MeshVersionFormatted 2"
    WRITE(mshUnit,FMT='(A)') "Dimension"
    WRITE(mshUnit,FMT='(A)') "3"
    WRITE(mshUnit,FMT='(A)') "SolAtVertices "
    WRITE(mshUnit,FMT='(I8)') nodeovnbr
    WRITE(mshUnit,FMT='(A)') "1 1"
    ! Do not use iterator first level here because nodes are not treated in the
same order than their index
    ! PaMPA reorganized node indexes in the following order: internal, local bou
ndary, overlap and halo
    DO is = 1, nodeovnbr
      WRITE(mshUnit,FMT='(F12.8)')  Ua(is)
    END DO
    WRITE (mshUnit, FMT='(A)') "End "
    CLOSE (mshUnit)
  END SUBROUTINE WriteDistributedMeshAndSolFiles3d


END PROGRAM helloPAMPA
```

Bibliographie

[1] Frédéric Alauzet, Xiangrong Li, E Seegyoung Seol et Mark S Shephard : Parallel anisotropic 3d mesh adaptation by mesh modification. *Engineering with Computers*, 21(3): 247–258, 2006.

[2] Nina Amenta, Sunghee Choi et Günter Rote : Incremental constructions con brio. *In Proceedings of the nineteenth annual symposium on Computational geometry*, pages 211–219. ACM, 2003.

[3] Timothy J Baker : Generation of tetrahedral meshes around complete aircraft. *Numerical grid generation in computational fluid mechanics'88*, pages 675–685, 1988.

[4] Timothy J Baker et John C Vassberg : Tetrahedral mesh generation and optimization. *In Proceedings of the 6th International Conference on Numerical Grid Generation*, pages 337–349, 1998.

[5] S. T. Barnard et H. D. Simon : A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency : Practice and Experience*, 6(2):101–117, 1994.

[6] A. Basermann, J. Clinckemaillie, T. Coupez, J. Fingberg, H. Digonnet, R. Ducloux, J.-M. Gratien, U. Hartmann, G. Lonsdale, B. Maerten, D. Roose et C. Walshaw : Dynamic load balancing of finite element applications with the drama library, 2000.

[7] A Basermann, J Clinckemaillie, T Coupez, J Fingberg, H Digonnet, R Ducloux, J.-M Gratien, U Hartmann, G Lonsdale, B Maerten, D Roose et C Walshaw : Dynamic load-balancing of finite element applications with the drama library. *Applied Mathematical Modelling*, 25(2):83 – 98, 2000. Dynamic load balancing of mesh-based applications on parallel.

[8] R. Battiti et A. A. Bertossi : Greedy, prohibition, and reactive heuristics for graph partitioning. *IEEE Transactions on Computers*, 48(4):361–385, 1999.

[9] Milind Bhandarkar et Laxmikant Kalé : A parallel framework for explicit fem. *In* Mateo Valero, Viktor Prasanna et Sriram Vajapeyam, éditeurs : *High Performance Computing - HiPC 2000*, volume 1970 de *Lecture Notes in Computer Science*, pages 385–394. Springer Berlin / Heidelberg, 2000. 10.1007/3-540-44467-X_35.

[10] E. G. Boman, K. D. Devine, L. A. Fisk, R. Heaphy, B. Hendrickson, V. Leung, C. Vaughan, U. Catalyurek, D. Bozdag et W. Mitchell : Zoltan home page. http://www.cs.sandia.gov/Zoltan, 1999.

[11] Brix, K., Melian, S., Müller, S. et Bachmann, M. : Adaptive multiresolution methods : Practical issues on data structures, implementation and parallelization*. *ESAIM : Proc.*, 34:151–183, 2011.

[12] T. N. Bui et B. R. Moon : Genetic algorithm and graph partitioning. *IEEE Trans. Comput.*, 45(7):841–855, 1996.

[13] Thang Nguyen Bui et Byung Ro Moon : Genetic algorithm and graph partitioning. *Computers, IEEE Transactions on*, 45(7):841–855, 1996.

[14] D.A. Burgess et M.B. Giles : Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines. *Advances in Engineering Software*, 28(3):189–201, 1997.

[15] G. W. Brown C. Farhat, N. Maman : Mesh partitioning for implicit computations via iterative domain decomposition : Impact and optimization of the subdomain aspect ratio. *International Journal for Numerical Methods in Engineering*, 38(6):989–1000, 1995.

[16] A Casagrande, P Leyland, L Formaggia et M Sala : Parallel mesh adaptation. *Series on Advances in Mathematics for Applied Sciences*, 69:201, 2005.

[17] Jose G Castanos et John E Savage : The dynamic adaptation of parallel mesh-based computation. 1996.

[18] P. Cavallo, G. Feldman et N. Sinha : Parallel unstructured mesh adaptation method for moving body applications. *AIAA Journal*, 43:1937–1945, septembre 2005.

[19] Peter A Cavallo, Neeraj Sinha et Gregory M Feldman : Parallel unstructured mesh adaptation method for moving body applications. *AIAA journal*, 43(9):1937–1945, 2005.

[20] P. Charrier et J. Roman : Partitioning and mapping for parallel nested dissection on distributed memory architectures. Rapport de recherche 92-12, LaBRI, Université Bordeaux I, mars 1992.

[21] P. Charrier et J. Roman : Partitioning and mapping for parallel nested dissection on distributed memory architectures. *In LNCS 634 – Proceedings of CONPAR'92*, pages 295–306. Springer-Verlag, septembre 1992.

[22] Rohit Chaube, Ricolindo L. Carino et Ioana Banicescu : Effectiveness of a dynamic load balancing library for scientific applications. *In ISPDC '07 : Proceedings of the Sixth International Symposium on Parallel and Distributed Computing*, page 32, Washington, DC, USA, 2007. IEEE Computer Society.

[23] JL Chenot, L Fourment, T Coupez, R Ducloux et E Wey : Forge3® : a general tool for practical optimisation of forging sequence of complex 3-d parts in industry. *Proc. ICFT*, 98:113–122.

[24] Cédric Chevalier : *Conception et mise en oeuvre d'outils efficaces pour le partitionnement et la distribution parallèles de problème numériques de très grande taille.* Thèse de doctorat, Université Sciences et Technologies-Bordeaux I, 2007.

[25] Cédric Chevalier et François Pellegrini : Pt-scotch : A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6):318–331, 2008.

[26] Cédric Chevalier et Ilya Safro : Comparison of coarsening schemes for multilevel graph partitioning. *In Learning and Intelligent Optimization*, pages 191–205. Springer, 2009.

[27] Nikos Chrisochoides : Parallel mesh generation. *In in Numerical Solution of Partial Differential Equations on Parallel Computers*. Springer, 2005.

[28] Nikos Chrisochoides et Demian Nave : Simultaneous mesh generation and partitioning for delaunay meshes. *Mathematics and Computers in Simulation*, 54(4):321–339, 2000.

[29] Nikos Chrisochoides et Démian Nave : Parallel delaunay mesh generation kernel. *International Journal for Numerical Methods in Engineering*, 58(2):161–176, 2003.

[30] P.G. Ciarlet : Basic error estimates for elliptic problems. *In Finite Element Methods (Part 1)*, volume 2 de *Handbook of Numerical Analysis*, pages 17 – 351. Elsevier, 1991.

[31] Paolo Cignoni, Domenico Laforenza, Raffaele Perego, Roberto Scopigno et Claudio Montani : Evaluation of parallelization strategies for an incremental delaunay triangulator in e3. *Concurrency : Practice and Experience*, 7(1):61–80, 1995.

[32] T. Coupez, H. Digonnet et R. Ducloux : Parallel meshing and remeshing. *Applied Mathematical Modelling*, 25(2):153 – 175, 2000. Dynamic load balancing of mesh-based applications on parallel.

[33] Thierry Coupez : Génération de maillage et adaptation de maillage par optimisation locale. *Revue européenne des éléments finis*, 9(4):403–423, 2000.

[34] Thierry Coupez, Hugues Digonnet et Richard Ducloux : Parallel meshing and remeshing. *Applied Mathematical Modelling*, 25(2):153–175, 2000.

[35] Tierry Coupez : A mesh improvement method for 3d automatic remeshing. *Numerical Grid Generation in Computational Fluid Dynamics and Related Fields*, pages 615–626, 1994.

[36] Architecture des ordinateurs. http://dept-info.labri.fr/ENSEIGNEMENT/archi/cours/archi.pdf.

[37] WN Dawes, SA Harvey, S Fellows, N Eccles, D Jaeggi et WP Kellar : A practical demonstration of scalable, parallel mesh generation. *In 47th AIAA Aerospace Sciences Meeting & Exhibit*, pages 5–8, 2009.

[38] H. L. de Cougny et M. S. Shephard : Parallel refinement and coarsening of tetrahedral meshes. *International Journal for Numerical Methods in Engineering*, 46:1101–1125, novembre 1999.

[39] Herman Deconinck, Kurt Sermeus et Rémi Abgrall : Status of multidimensional upwind residual distribution schemes and applications in aeronautics. 2000.

[40] Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson et Courtenay Vaughan : Zoltan data management services for parallel dynamic applications. *Computing in Science & Engineering*, 4(2):90–96, 2002.

[41] Karen Devine, Bruce Hendrickson, Erik Boman, Matthew St John, Courtenay Vaughan et WF Mitchell : Zoltan : A dynamic load-balancing library for parallel applications ; userś guide. *Sandia National Laboratories Tech. Rep*, 1999.

[42] Karen Devine, Bruce Hendrickson, Erik Boman, Matthew St. John et Courtenay Vaughan : Design of dynamic load-balancing tools for parallel applications. *In ICS '00 : Proceedings of the 14th international conference on Supercomputing*, pages 110–118, New York, NY, USA, 2000. ACM.

[43] Karen D. Devine, Erik G. Boman, Robert T. Heaphy, Bruce A. Hendrickson, James D. Teresco, Jamal Faik, Joseph E. Flaherty et Luis G. Gervasio : New challenges in dynamic load balancing. *Applied Numerical Mathematics*, 52(2-3):133–152, 2005. ADAPT '03 : Conference on Adaptive Methods for Partial Differential Equations and Large-Scale Computation.

[44] Hugues Digonnet, Marc Bernacki, Luisa Silva et Thierry Coupez : Adaptation de maillage en parallèle, application à la simulation de la mise en forme des matériaux. *In Congrès Français de Mécanique Grenoble-CFM 2007*, page 6 pages, Grenoble, France, 2007. http ://hdl.handle.net/2042/16046.

[45] Hugues Digonnet, Marc Bernacki, Luisa Silva, Thierry Coupez *et al.* : Adaptation de maillage en parallèle, application à la simulation de la mise en forme des matériaux. *Congrès Français de Mécanique Grenoble-CFM 2007*, 2007.

[46] Hugues Digonnet, Thierry Coupez *et al.* : Object-oriented programming for'fast and easy'development of parallel applications in forming processes simulation. *Computational Fluid And Solid Mechanics 2003, Vols 1 and 2, Proceedings*, 2003.

[47] Hugues Digonnet, Luisa Silva et Thierry Coupez : Cimlib : A fully parallel application for numerical simulations based on components assembly. *AIP Conference Proceedings*, 908(1):269–274, 2007.

[48] C. Dobrzynski : *Adaptation de maillage anisotrope 3D et application Ã l'aéro-thermique des bâtiments*. Thèse de doctorat, Université Pierre et Marie Curie - Paris VI, novembre 2005.

[49] C Dobrzynski et JF Remacle : Parallel mesh adaptation. *International Journal for Numerical Methods in Engineering., in preparation*.

[50] MMG3D : Anisotropic tetrahedral remesher/moving mesh generation. http://www.math.u-bordeaux1.fr/~cdobrzyn/logiciels/mmg3d.php.

[51] J. J. Dongarra, J. Du Croz, S. Hammarling et R. J. Hanson : An extended set of FORTRAN Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 14(1):1–17, mars 1988.

[52] C. M. Fiduccia et R. M. Mattheyses : A linear-time heuristic for improving network partitions. *In Proceedings of the 19th Design Automation Conference*, pages 175–181. IEEE, 1982.

[53] J. Fingberg, A. Basermann, G. Lonsdale, J. Clinckemaillie, J.-M. Gratien et R. Ducloux : Developments in engineering computational technology. chapitre Dynamic load balancing for parallel structural mechanics simulations with DRAMA, pages 199–205. Civil-Comp press, Edinburgh, UK, UK, 2000.

[54] P.J. Frey et P.-L. George : *Maillages : applications aux éléments finis*. Hermès Science, 1999.

[55] M. R. Garey, D. S. Johnson et L. Stockmeyer : Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[56] Paul-Louis George et Houman Borouchaki : *Delaunay triangulation and meshing : application to finite elements*. Hermes Paris, 1998.

[57] Stanisław Gepner, Jerzy Majewski et Jacek Rokicki : Dynamic load balancing for adaptive parallel flow problems. *In Parallel Processing and Applied Mathematics*, pages 61–69. Springer, 2010.

[58] Jens Gerlach : *Domain engineering and generic programming for parallel scientific computing*. Thèse de doctorat, 2002.

[59] Gilles Grospellier et Benoit Lelandais : The arcane development framework. *In Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, POOSC '09, pages 4 :1–4 :11, New York, NY, USA, 2009. ACM.

[60] B. Hendrickson et R. Leland : A multilevel algorithm for partitioning graphs. *In Proceedings of Supercomputing*, 1995.

[61] A Hosangadi, RA Lee, PA Cavallo, N Sinha et BJ York : Hybrid, viscous, unstructured mesh solver for propulsive applications. *AIAA paper*, pages 98–3153, 1998.

[62] Ashvin Hosangadi, Robert A Lee, Brian J York, Neeraj Sinha et Sanford M Dash : Upwind unstructured scheme for three-dimensional combusting flows. *Journal of Propulsion and Power*, 12(3):494–502, 1996.

[63] IEEE. *The Open Group Technical Standard Base Specifications*, IEEE Std 1003.1 édition, avril 2004.

[64] Martin Isenburg, Yuanxin Liu, Jonathan Shewchuk et Jack Snoeyink : Streaming computation of delaunay triangulations. *In ACM Transactions on Graphics (TOG)*, volume 25, pages 1049–1056. ACM, 2006.

[65] Quotient isopérimétrique. http://fr.wikipedia.org/wiki/Quotient_isop%C3%A9rim%C3%A9trique.

[66] The itaps mesh interface imesh. http://www.itaps.org/software/iMesh_html/index.html.

[67] Laxmikant V Kale et Sanjeev Krishnan : Charm++ : Parallel programming with message-driven objects. *Parallel Programming using C+*, pages 175–213, 1996.

[68] George Karypis, Kirk Schloegel et Vipin Kumar : Parmetis : Parallel graph partitioning and sparse matrix ordering library. *Version 1.0, Dept. of Computer Science, University of Minnesota*, 1997.

[69] MeTiS : Family of multilevel partitioning algorithms. http://glaros.dtc.umn.edu/gkhome/views/metis.

[70] B. W. Kernighan et S. Lin : An efficient heuristic procedure for partitioning graphs. *BELL System Technical Journal*, pages 291–307, février 1970.

[71] Ing Lutz Laemmer : Parallel mesh generation. *In Solving Irregularly Structured Problems in Parallel*, pages 1–12. Springer, 1997.

[72] B. G. Larwood, N.P. Weatherhill, O. Hassan et K. Morgan : Domain decomposition approach for parallel unstructured mesh generation. *In International Journal for Numerical Methods in Engineering*, volume 58, pages 177–188, 2003.

[73] BG Larwood, Nigel P Weatherill, O Hassan et K Morgan : Domain decomposition approach for parallel unstructured mesh generation. *International journal for numerical methods in engineering*, 58(2):177–188, 2003.

[74] Orion Lawlor, Sayantan Chakravorty, Terry Wilmarth, Nilesh Choudhury, Isaac Dooley, Gengbin Zheng et Laxmikant Kalé : Parfum : a parallel framework for unstructured meshes for scalable dynamic physics applications. *Engineering with Computers*, 22:215–235, 2006. 10.1007/s00366-006-0039-5.

[75] Rainald Löhner : Some useful renumbering strategies for unstructured grids. *International Journal for Numerical Methods in Engineering*, 36(19):3259–3270, 1993.

[76] Rainald Löhner : Renumbering strategies for unstructured-grid solvers operating on shared-memory, cache-based parallel machines. *Computer methods in applied mechanics and engineering*, 163(1):95–109, 1998.

[77] Adrien Loseille et Frédéric Alauzet : Shrimp User Guide. A Fast Mesh Renumbering and Domain Partionning Method. Technical Report RT-0362, INRIA, 2009.

[78] M. Luby : A simple parallel algorithm for the maximal independent set problem. *SIAM Journal on Computing*, 15(4):1036–1055, 1986.

[79] Bart Maerten, Dirk Roose, Achim Basermann, Jochen Fingberg et Guy Lonsdale : Drama : A library for parallel dynamic load balancing of finite element applications. *In* Patrick Amestoy, Philippe Berger, Michel Daydé, Daniel Ruiz, Iain Duff, Valérie Frayssé et Luc Giraud, éditeurs : *Euro-Par'99 Parallel Processing*, volume 1685 de *Lecture Notes in Computer Science*, pages 313–316. Springer Berlin / Heidelberg, 1999. 10.1007/3-540-48311-X_40.

[80] Mpi-2.0 standards. http://www.mpi-forum.org/docs/docs.html, 1997.

[81] S.N. Muthukrishnan, P.S. Shiakolas, R.V. Nambiar et K.L. Lawrence : Simple algorithm for adaptive refinement of three-dimensional finite element tetrahedral meshes. *AIAA journal*, 33(5):928–932, 1995.

[82] Kengo Nakajima, Jochen Fingberg et Hiroshi Okuda : Parallel 3d adaptive compressible navier-stokes solver in geofem with dynamic load-balancing by drama library. *In* Bob Hertzberger, Alfons Hoekstra et Roy Williams, éditeurs : *High-Performance Computing and Networking*, volume 2110 de *Lecture Notes in Computer Science*, pages 183–193. Springer Berlin / Heidelberg, 2001. 10.1007/3-540-48228-8_19.

[83] Leonid Oliker et Rupak Biswas : Plum : Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 52(2):150–177, 1998.

[84] Leonid Oliker, Rupak Biswas et Harold N Gabow : Parallel tetrahedral mesh adaptation with dynamic load balancing. *Parallel Computing*, 26(12):1583–1608, 2000.

[85] Leonid Oliker, Rupak Biswas et Harold N Gabow : Parallel tetrahedral mesh adaptation with dynamic load balancing. *Parallel Computing*, 26(12):1583 – 1608, 2000. Graph Partitioning and Parallel Computing.

[86] Steven J Owen : A survey of unstructured mesh generation technology. *In IMR*, pages 239–267, 1998.

[87] C Ozturan, HL deCougny, MS Shephard et JE Flaherty : Parallel adaptive mesh refinement and redistribution on distributed memory computers. *Computer Methods in Applied Mechanics and Engineering*, 119(1):123–137, 1994.

[88] C. H. Papadimitriou : The NP-completeness of the bandwidth minimization problem. *Computing*, 16:263–270, 1976.

[89] Scotch : Static mapping, graph partitioning, and sparse matrix block ordering package. http://www.labri.fr/~pelegrin/scotch/.

[90] Alex Pothen, Horst D Simon et Kang-Pu Liou : Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications*, 11(3):430–452, 1990.

[91] M. Ramadan, L. Fourment et H. Digonnet : A parallel two mesh method for speeding-up processes with localized deformations : application to cogging. *International Journal of Material Forming*, 2:581–584, 2009. 10.1007/s12289-009-0440-x.

[92] Mohamad Ramadan, Lionel Fourment et Hugues Digonnet : A parallel two mesh method for speeding-up processes with localized deformations : application to cogging. *International Journal of Material Forming*, 2(1):581–584, 2009.

[93] Jean-François Remacle, Ottmar Klaas, Joseph E. Flaherty et Mark S. Shephard : Parallel algorithm oriented mesh database. *Engineering with Computers*, 18:274–284, 2002. 10.1007/s003660200024.

[94] M Cecilia Rivara : Algorithms for refining triangular grids suitable for adaptive and multigrid techniques. *International journal for numerical methods in Engineering*, 20(4): 745–756, 1984.

[95] R Said, NP Weatherill, K Morgan et NA Verhoeven : Distributed parallel delaunay mesh generation. *Computer methods in applied mechanics and engineering*, 177(1):109–125, 1999.

[96] Kirk Schloegel, George Karypis et Vipin Kumar : A unified algorithm for load-balancing adaptive scientific simulations. *In Supercomputing, ACM/IEEE 2000 Conference*, pages 59–59. IEEE, 2000.

[97] M. Tchiboukdjian, V. Danjean et B. Raffin : Binary mesh partitioning for cache-efficient visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 16(5): 815 –828, sept.-oct. 2010.

[98] Matthias Tief et Jens Gerlach : High-Quality Geometric Repartitioning with Mosaik. *In* H.R. Arabnia, éditeur : *Proceeding of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 542–548, Las Vegas, Nevada, USA, 2001. http ://hdl.handle.net/2042/16046.

[99] SA Transvalor : Drama project deliverable d1.3a. final report on re-partitioning algorithms. 1998.

[100] U. Tremel, K. A. Sørensen, S. Hitzel, H. Rieger, Oubay Hassan et Nigel P. Weatherill : Parallel remeshing of unstructured volume grids for cfd applications. *In International Journal for Numerical Methods in Fluids*, volume 53, pages 1361–1379. John Wiley & Sons, 2007.

[101] U Tremel, KA Sørensen, S Hitzel, H Rieger, Oubay Hassan et Nigel P Weatherill : Parallel remeshing of unstructured volume grids for cfd applications. *International journal for numerical methods in fluids*, 53(8):1361–1379, 2007.

[102] A. Trifunović : *Parallel Algorithms for Hypergraph Partitioning.* Thèse de doctorat, Imperial College London, février 2006.

[103] A. Trifunović et W. Knottenbelt : Par*k*way2.0 : A Parallel Multilevel Hypergraph Partitioning Tool. *In Proc. 19th International Symposium on Computer and Information Sciences*, volume 3280 de *Lecture Notes in Computer Science*, pages 789–800. Springer, 2004.

[104] M. G. Vallet : *Génération de maillages anisotropes et adaptatifs.* Thèse de doctorat, Université Pierre et Marie Curie - Paris VI, 1992.

[105] R. van Driessche et D. Roose : A graph contraction algorithm for the calculation of eigenvectors of the laplacian matrix of a graph with a multilevel method. Rapport technique TW 209, Katholieke Universiteit Leuven, mai 1994.

[106] R. Verfürth : *A review of a posteriori error estimation and adaptive mesh-refinement techniques. Vol. 1.* Wiley-Teubner New York, Stuttgart, 1996.

[107] N.A. Verhoeven, N.P. Weatherill, K. Morgan *et al.* : Dynamic load balancing in a 2d parallel delaunay mesh generator. *In Parallel Computational Fluid Dynamics*, pages 641–648, 1995.

[108] C. Walshaw et M. Cross : Parallel optimisation algorithms for multilevel mesh partitioning. *Parallel Computing*, 26(12):1635 – 1660, 2000. Graph Partitioning and Parallel Computing.

[109] Chris Walshaw, Mark Cross et MG Everett : Dynamic load-balancing for parallel adaptive unstructured meshes. *In PPSC*, 1997.

[110] Numérotations de matrices de type cuthill-mckee ou gibbs-poole-stockmeyer. http://en.wikipedia.org/wiki/Cuthill%E2%80%93McKee_algorithm.

[111] Mémoire cache. http://fr.wikipedia.org/wiki/M%C3%A9moire_cache.

[112] Charm++ : Parallel languages/paradigms : Charm ++ - parallel objects. http://ppl.cs.illinois.edu/research/charm.

[113] Cmake : cross-platform, open-source build system. http://www.cmake.org/.

[114] Tgcc curie. http://www-hpc.cea.fr/en/complexe/tgcc-curie.htm.

[115] Triangulation de delaunay. http://fr.wikipedia.org/wiki/Triangulation_de_Delaunay.

[116] écart interquartile. http://fr.wikipedia.org/wiki/%C3%89cart_interquartile.

[117] Méthode des éléments finis. http://fr.wikipedia.org/wiki/M%C3%A9thode_des_%C3%A9l%C3%A9ments_finis.

[118] Courbes de hilbert. http://fr.wikipedia.org/wiki/Courbe_de_Hilbert.

[119] Interpolation numérique. http://fr.wikipedia.org/wiki/Interpolation_%28math%C3%A9matiques%29.

[120] Jostle : Graph partitioning software. http://staffweb.cms.gre.ac.uk/~c.walshaw/jostle/.

[121] Estimation par noyau. http://fr.wikipedia.org/wiki/Estimation_par_noyau.

[122] Netgen - automatic mesh generator. http://www.hpfem.jku.at/netgen/.

[123] The openmp api specification for parallel programming. http://www.openmp.org.

[124] Phg : Parallel hierarchical grid. http://lsec.cc.ac.cn/phg/.

[125] Plafrim/dihpes : Plateforme fédérative pour la recherche en informatique et mathématiques. https://plafrim.bordeaux.inria.fr.

[126] Calculateur quantique ou ordinateur quantique. http://fr.wikipedia.org/wiki/Calculateur_quantique.

[127] Shrimp : A fast mesh renumbering and domain partitionning method. http://hal.inria.fr/inria-00362994.

[128] Tgcc : Très grand centre de calcul du cea. http://www-hpc.cea.fr/fr/complexe/tgcc.htm.

[129] Yusuf Yilmaz, Can Özturan, Oğuz Tosun, Ali Haydar Özer et Seren Soner : Parallel mesh generation, migration and partitioning for the elmer application.