Partitionnement de maillages pour l'équilibrage de charge de simulations multi-physiques

Hubert Hirtz

université PARIS-SACLAY

Partitionnement de maillages pour l'équilibrage de charge de simulations multi-physiques

Mesh partitioning for load balancing multiphysics simulations

Thèse de doctorat de l'université Paris-Saclay

École doctorale n°580 Sciences et Technologies de l'Information et de la Communication (STIC)
Spécialité de doctorat : Informatique
Graduate School : Informatique et sciences du numérique. Référent : Faculté des sciences d'Orsay

Thèse préparée dans l'unité de recherche **LIHPC (Université Paris-Saclay, CEA)**, sous la direction de **Franck LEDOUX**, ingénieur-chercheur au CEA.

Thèse soutenue à Paris-Saclay, le 12 décembre 2023, par

Hubert HIRTZ

Composition du jury

Membres du jury avec voix délibérative

Patrick CARRIBAULT Ingénieur-chercheur, CEA	Président
Lélia BLIN Professeure des universités, Université Paris Cité	Rapportrice et examinatrice
Bora UÇAR Directeur de recherche, CNRS, ENS Lyon	Rapporteur et examinateur
Aurélien ESNARD Maître de conférences, Université de Bordeaux	Examinateur

THESE DE DOCTORAT
NNT : 2023UPASG096

université PARIS-SACLAY

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Partitionnement de maillages pour l'équilibrage de charge de simulations multi-physiques

Mots clés : parallélisme, maillage, équilibrage de charge, multi-physiques, graphes

Résumé : Cette étude s'inscrit dans le domaine de l'optimisation de performances de simulations numériques distribuées à grande échelle à base de maillages. Dans ce domaine, nous nous intéressons au bon équilibre de charge entre les unités de calcul sur lesquelles la simulation s'exécute. Pour équilibrer la charge d'une simulation à base de maillage, il faut généralement prendre en compte de la quantité de calcul nécessaire pour chaque maille, ainsi que la quantité de données qui doivent être transférées entre les unités de calcul. Les outils communément utilisés pour résoudre ce problème le solvent d'une manière, qui n'est pas forcément optimale pour une simulation donné, car ils s'appliquent à de nombreux cas autres que l'équilibrage de charge et le partitionnement de maillage. Notre étude consiste à concevoir et implémenter un nouvel outil de partitionnement dédié aux maillages et à l'équilibrage de charge. Après une explication approfondie du contexte de l'étude, des problèmes de partitionnement ainsi que de l'état de l'art des algorithmes de partitionnement, nous montrons l'intérêt de chaîner des algorithmes pour optimiser de différentes façon une partition de maillage. Ensuite, nous étoffons cette méthode de chaînage en deux points : d'abord, en étendant l'algorithme de partitionnement de nombres VNBest pour l'équilibrage de charge où les unités de calcul sont hétérogènes, puis en spécialisant l'algorithme de partitionnement géométrique RCB, pour améliorer ses performances sur les maillages cartésiens. Nous décrivons en détails le processus de conception de notre outil de partitionnement, qui fonctionne exclusivement en mémoire partagée. Nous montrons notre outil peut obtenir des partitions avec un meilleur équilibre de charge que deux outils de partitionnement en mémoire partagée existants, Scotch et Metis. Cependant, nous ne minimisons pas aussi bien les transferts de données entre unités de calcul. Nous présentons les caractéristiques de performance des algorithmes implémentés en *multithread*.

Title: Mesh partitioning for load balancing multiphysics simulations
Keywords: parallelism, mesh, load balancing, multi-physics, graphs

Abstract: This study aims to improve the performance of large-scale, distributed, mesh-based simulations. More specifically, we take an interest in improving the balance between the load on each computation unit on which the simulations execute. Load balancing mesh-based simulations generally involve both the load of each mesh cell and the data transfers between computation units. State-of-the-art tools most commonly solve this problem in one way, which is not always the most optimal for a given simulation, because their scope is broader than that of mesh partitioning and load-balancing. Our study aims to fill this gap and implement a new partitioning tool dedicated to meshes and load-balancing. After an in-depth presentation of the context and issues related to balancing the load of distributed, mesh-based simulations, as well as the definition of common mesh-partitioning-related problems and associated, state-of-the-art algorithms to solve each of these problems, we present a new method to partition meshes, which involves carefully chaining algorithms to optimise given mesh partition metrics. We then develop this method in two ways: first, we expand the number partitioning algorithm called VNBest so that it can balance load against target weights. Second, we adapt the geometric algorithm called RCB for cartesian meshes, to improve its performance on such meshes. We implement the algorithm chaining method and its extensions in a new partitioning tool and describe its design thoroughly. We show our tool can compute partitions with better load-balance than state-of-the-art tools Scotch and Metis. However, the edgecut is not as optimised as in state-of-the-art. We present the scaling of our parallel implementation of partitioning algorithms.

Table des matières

Introduction

Prévoir la météo à partir des résultats de milliers de stations, modéliser des hélices d'éoliennes pour maximiser leur quantité d'énergie produite, étudier des phénomènes particulaires et quantiques, connaître les conséquences d'un choc automobile à haute vitesse sur le conducteur pour différents types de carrosserie. Toutes ces exemples nécessitent de nombreuses expérimentations « réelles » ou « virtuelles » pour apporter des éléments de réponses. Par virtuel, nous entendons l'utilisation d'un modèle qui peut être physique, mathématique et/ou informatique. En pratique, la modélisation de phénomènes physiques tels que ceux précédemment cités est réalisée en cinq temps, illustrés sur la Figure 1 :

1. Tout d'abord des modèles physiques sont définis pour décrire le problème à étudier. Dans de nombreux cas, les problèmes rencontrés considèrent plusieurs questions physiques simultanément et les problèmes dits multi-physiques sont complexes à modéliser précisément. Une modélisation usuelle à ce niveau est de disposer d'un modèle consistant en un ou plusieurs systèmes d'équations aux dérivées partielles.
2. Une fois le problème physique modélisé, il est discrétisé et approché à l'aide de méthodes numériques pour obtenir un modèle discret qui puisse être résolu à l'aide d'un ordinateur. Cette étape peut être vu comme le passage d'un monde continu (le monde réel) à un monde discret permettant des calculs efficaces sur un ordinateur. Cette discrétisation se fait à la fois en temps et en espace généralement. À titre d'exemple, considérons la déformation d'une carrosserie de voiture lors d'un impact. Cette déformation se fait de manière continue au cours du temps, mais nous modélisons mathématiquement l'écoulement du temps en considérant que le temps « avance » par pas de temps discrets. De même, pour modéliser mathématiquement la déformation de la carrosserie, nous découpons celle-ci en un ensemble de petits morceaux qui servent de support aux calculs mathématiques. Ces petits morceaux sont communément nommés des **mailles**, et leur ensemble forme un **maillage**. Il existe de nombreux types de maillage (par exemple structurés, non-structurés, cartésiens) formés de plusieurs types de mailles (par exemple des triangles, quadrilatères en 2D et tétraèdres, hexaèdres, prismes, pyramides en 3D). Le type de maillage utilisé dépend, entre autres, des méthodes numériques utilisées (éléments finis, volumes finis, différences finies).

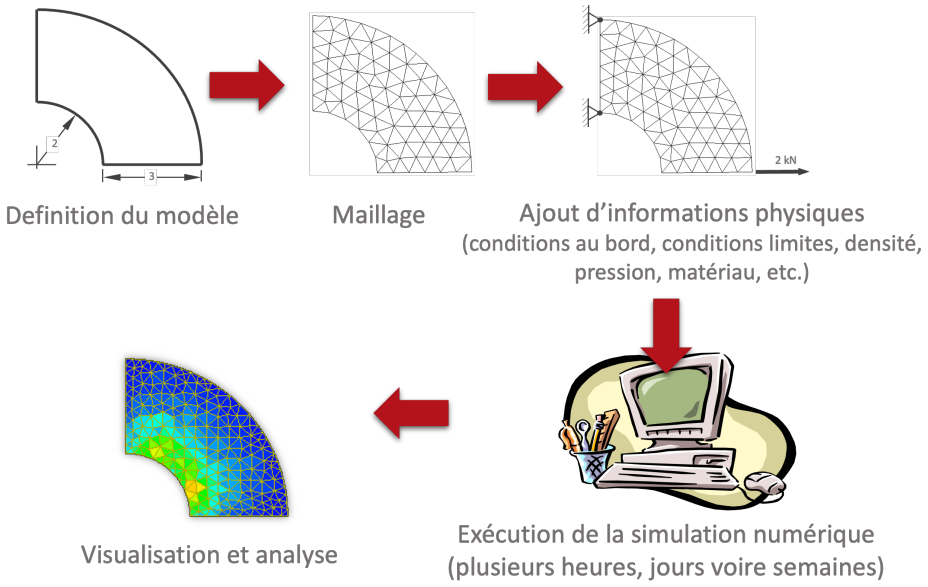


Figure 1 – D'un modèle physique à l'analyse de résultats de simulation. Nous commençons par définir le modèle géométriquement, puis nous le discrétisons à l'aide d'un maillage et nous ajoutons les informations physico-numériques nécessaires au code de simulation. Après l'exécution du code, nous pouvons alors visualiser et analyser les valeurs portées par le maillage.

3. L'information nécessaire pour représenter les conditions initiales du problème est ajoutée sur le maillage. Par exemple, des conditions aux bords du maillage, ou des valeurs initiales d'une grandeur en chaque maille.
4. Une fois le modèle discret défini, nous pouvons envisager des expérimentations « virtuelles », autrement appelées simulations numériques, codes de calcul, ou codes de simulation, qui sont exécutées sur matériel informatique. Selon le problème étudié, la discrétisation en temps et en espace nécessitent des capacités de calcul très importantes pour obtenir un résultat en un temps raisonnable. Si la notion de « raisonnable » est très dépendante du domaine d'application, nous considérons dans notre contexte la simulation de phénomènes physiques complexes qui peuvent nécessiter plusieurs heures, voire plusieurs jours de calculs sur des supercalculateurs.
5. Une fois la simulation terminée, les résultats peuvent être analysés.

Les machines d'aujourd'hui mélangent différents niveaux de parallélisme – distribution des données entre nœuds de calculs ne partageant

pas de mémoire, puis partage de la mémoire entre unités de calcul d'un même nœud et cohabitation avec des unités accélératrices massivement vectorielles (GPU aujourd'hui). Il est nécessaire de tenir compte de la distribution des données, ce qui est l'objet de cette thèse. L'un des objectifs de la distribution des données est de rendre la simulation plus rapide à exécuter, mais aussi tout simplement possible. Ainsi, lorsque la taille des données de simulation dépasse la capacité mémoire d'une machine, ou afin de diminuer le temps d'exécution, nous répartissons les calculs et les données sur plusieurs machines. C'est ce que nous entendons par « équilibrage de charge ».

Nous considérons dans ce travail des simulations utilisant la notion de maillage pour discrétiser l'espace d'étude (notre carrosserie de voiture par exemple). L'équilibrage de charge de telles simulations consiste à **partitionner** le maillage, ainsi que les données associées à chaque maille – un scalaire modélisant une température ou une pression, un vecteur modélisant un déplacement – et distribuer les parties sur les unités de calcul, en prenant en compte la charge de calcul et la quantité de données associées à chaque maille, de façon à minimiser les temps d'attente et les transferts réseau, qui influent sur le temps total d'exécution de la simulation.

De nombreux outils de partitionnement existent aujourd'hui [1]–[7]. La plupart de ces outils fonctionnent sur des graphes ou hypergraphes arbitraires, qu'ils soient issus de maillages, réseaux d'énergie ou relations sociales. Pour des simulations reposant sur des maillages, l'utilisation de tels outils nécessite de représenter le maillage par un graphe approprié au traitement qui nous intéresse. Par exemple, supposons une simulation en dimension 2 utilisant un maillage triangulaire tel que celui de la Figure 2a, et que cette simulation calcule une valeur scalaire V_i en chaque triangle i, et pour ce calcul, elle a besoin d'accéder à la valeur V_j de tout triangle j voisin par une arête au triangle i. Dans ce cas de figure, on modélisera le maillage de la Figure 2a par le graphe (S, A) de la Figure 2b où chaque sommet s de S modélise un triangle du maillage et chaque arête a de A entre deux nœud du graphe modélise le fait que les mailles correspondant aux sommets extrémités de l'arête a sont voisins par une arête dans le maillage initial. Si le calcul de la valeur V_i à la maille i avait nécessité les valeurs des mailles partageant un sommet avec la maille i, nous aurions modélisé le problème par un graphe différent, en l'occurrence celui de la Figure 2c.

Les outils reposant sur des modèles de graphe ou d'hypergraphe minimisent les coûts de communication sous une contrainte d'équilibre. En d'autres termes, ils minimisent le nombre d'arêtes coupées dans le graphe tout en faisant en sorte que le travail réalisé sur les sommets de chaque

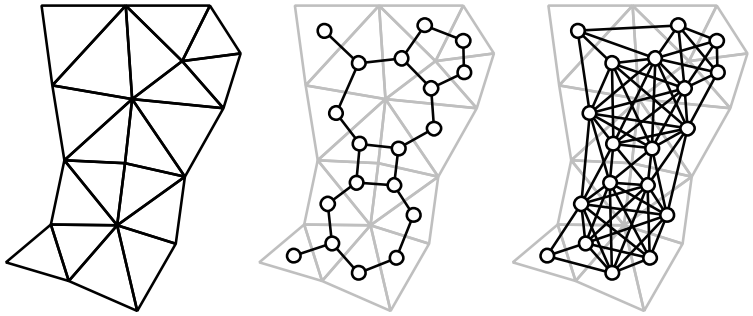


Figure 2 – Modélisation d'un maillage triangulaire (a) à l'aide de deux graphes (b) et (c), où les sommets sont représentés par un cercle. Sur la figure (b), les arêtes relient deux sommets qui représentent des mailles partageant une arête du maillage. Sur la figure (c), les arêtes relient deux sommets qui représentent des mailles partageant un nœud.

partie soit « équitable ». Dans un cadre général, les arêtes et sommets du graphe sont pondérés : le poids sur un sommet modélise la charge de travail associé à ce sommet ; le poids sur une arête modélise le coût de la communication des données entre les sommets extrémités.

Il existe d'autres manières de partitionner un maillage. Par exemple, il est aussi possible d'utiliser sa géométrie [8]–[11]. Dans le cas du partitionnement de graphe et d'hypergraphes, seule la structure, i.e. la topologie, du maillage est utilisée. Dans le cas des partitionneurs géométriques, le parti pris est inverse : la topologie du maillage n'est pas prise en compte, seule la géométrie est considérée. Par exemple le maillage de la Figure 2a peut être partitionné géométriquement en 2 parties en considérant les barycentres des mailles selon l'axe X ou Y (voir la Figure 3).

Le partitionnement géométrique a comme avantage de mieux exploiter le matériel informatique pour donner des partitions en moins de temps que le partitionnement de graphe ou hypergraphe.

Une autre manière de partitionner le maillage est de considérer uniquement des points associés aux mailles. Ce problème correspond alors au partitionnement de nombres, problème qui permet d'obtenir des partitions au déséquilibre de charge minimal, indépendamment de la topologie du maillage. Enfin, de nouvelles méthodes [12] prenant en compte la quantité mémoire induite par chaque partie ont été développées, pour s'assurer que chaque unité de calcul peut travailler sur sa partie avec sa capacité mémoire.

Lors du partitionnement de maillages, bien qu'il soit possible d'appliquer chacune de ces méthodes, les outils de partitionnement existants ne s'intéressent qu'à l'un de ces aspects du partitionnement. L'objectif de

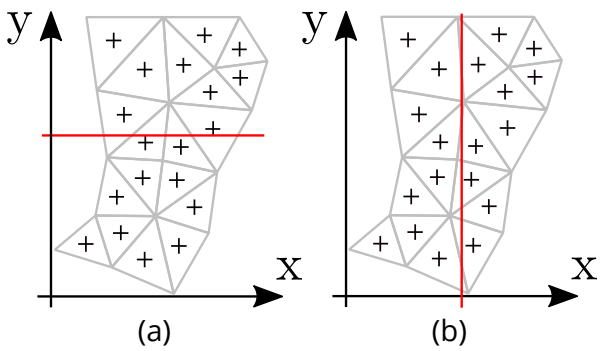


Figure 3 – Exemples de bi-partitionnement en 2D. Le barycentre des mailles (représenté par une croix) est projeté sur l'axe des abscisses en (a), ou des ordonnées en (b), puis une valeur pivot est recherchée. Cette valeur pivot, représenté par une droite rouge, doit séparer l'ensemble des barycentres en deux parties de cardinal le plus proche l'un de l'autre. Ainsi, sur les deux images, les deux parties du maillage ont dix et neuf mailles respectivement.

cette thèse est de répondre à ce manque d'outil de partitionnement dédié aux maillages et à l'équilibrage de charge de simulations numériques grande échelle, prenant en compte à la fois de la géométrie et de la topologie du maillage. Dans cette optique, nous avons réalisé deux choix majeurs qui ont piloté le travail de thèse :

1. Tout d'abord, nous considérons le problème de partitionnement de maillage comme un problème nécessitant de tenir compte à la fois la topologie et la géométrie du maillage. Pour cela, nous proposons de partitionner un maillage de manière itérative et incrémentale en proposant un chaînage d'algorithmes de partitionnement. Le premier algorithme appliqué partitionne le maillage, les suivants optimisent le partitionnement.
2. Ensuite, dans l'optique d'obtenir une implémentation efficace, sûre et propice à une parallélisation en mémoire partagée, nous avons fait le choix du langage de programmation Rust [13].

Structure du manuscrit et contributions principales

Afin de présenter les réalisations de ce travail de thèse, le manuscrit est structuré de la manière suivante. Le Chapitre 1 fournit des éléments de contexte pour la compréhension de notre travail. Il présente en particulier le modèle d'exécution de simulation numérique utilisé pendant la thèse, des notions minimales de maillage, et des métriques d'intérêt lors de l'exécution sur supercalculateurs.

Les Chapitres 2 et 3 présentent l'état de l'art des problèmes et algo-

rithmes de partitionnement. Les différentes approches de partitionnement qui ont été utilisées pendant cette thèse sont décrites, ainsi que les algorithmes qui leur sont associés. L'ensemble des problèmes et algorithmes sont ainsi présentés de manière unifiée.

Les Chapitres 4 à 6 constituent différentes contributions de notre travail. Tout d'abord, nous présentons au Chapitre 4, notre méthode de partitionnement de maillage par chaînage d'algorithmes. Sachant qu'il existe déjà des algorithmes de partitionnement de nombres, géométrique et topologiques qui créent ou optimisent des partitions, nous montrons comment les assembler et les appliquer ensemble. Pour cela, nous étudions l'évolution de différentes métriques des partitions lors de l'application des algorithmes. Cette méthode est la base de notre outil de partitionnement, et les travaux présentés aux Chapitres 5 et 6 participent à l'étoffer.

L'objet du Chapitre 5 a été de permettre d'appliquer la méthode de chaînage d'algorithmes lorsque chaque partie ne doit pas forcément avoir autant de charge que les autres. On considère ici le cas de figure où les unités de calcul n'ont pas les mêmes capacités de calcul. Pour cela, nous étendons l'algorithme VNBest [14], un algorithme de partitionnement de nombres, afin qu'il prenne en compte un poids cible pour chaque partie. Ensuite, nous continuons d'étendre l'algorithme pour qu'il puisse être appliqué sur un nombre arbitraire de parties.

Dans le Chapitre 6, nous considérons le cas particulier du partitionnement de maillage cartésien. Partitionner un maillage cartésien demande dans certains cas que les parties soient elles aussi cartésiennes. Des méthodes issues de l'état de l'art consistent à « ajuster » des partitions issues de partitionneurs de graphe de façon à rendre les parties cartésiennes, ce qui ne permet pas toujours de garder un bon équilibre, ni un taux faible de coûts de communications. Au contraire, des algorithmes géométriques comme RCB – *Recursive Coordinate Bisection* – donnent par définition des parties cartésiennes. Dans ce chapitre, nous cherchons à spécialiser l'implémentation de RCB pour les maillages cartésiens, afin d'améliorer ses performances – notamment, diminuer le temps et la quantité mémoire requis à son exécution. Nous comparons ensuite la version de l'état de l'art et la version cartésienne de RCB sur des maillages cartésiens de différentes tailles.

Enfin les Chapitres 7 et 8 abordent le développement du partitionneur Coupe, logiciel conçu et développé durant cette thèse. Ce logiciel permet de chaîner ces algorithmes programmatiquement à l'aide d'une interface de programmation modulaire, ce qui nous permet de choisir les objectifs d'optimisation du partitionnement. Pour valider nos algorithmes et l'apport du chaînage proposé dans Coupe, nous proposons un cadriciel d'expérimentation permettant de facilement tester des chaînes d'algorithmes

sur des maillages et des distributions de charge de calcul et de taille de données donnés. Cet outil nous a permis de mener, entre autres, les expérimentations dont les résultats sont présentés au Chapitre 8.

Communications scientifiques

Les travaux présentés aux Chapitres 4 et 7 ont fait l'objet de deux publications :

1. La première intitulée « *Coupe : A Modular, Multi-threaded Mesh Partitioning Platform.* » [15] a été acceptée et présenté à un workshop de la conférence Europar 2022.
2. La seconde intitulée « *Coupe : A mesh partitioning platform.* » [16] a été accepté et présentée à la conférence *International Meshing Roundtable*, en mars 2023 à Amsterdam. La sélection se fait sur soumission de l'article, et acceptation par trois rapporteurs. L'article est publié dans les proceedings de la conférence.

Ces travaux ont aussi été présenté à la conférence *SIAM Conference Science and Engineering* en février 2023 à Amsterdam, ainsi qu'à la journée des doctorants du CEA en 2023. Ils ont aussi fait l'objet d'un poster à la journée des doctorants du CEA en 2022.

Finalement, des logiciels développés pendant cette thèse ont été développé à code ouvert, sur Github :

- Coupe [1], l'outil de partitionnement conçu pendant cette thèse ;
- Des *bindings* [2] [3] en Rust pour les outils de partitionnement existants Scotch [3] et Metis [1], qui ont permis l'intégration de ces deux outils de partitionnement dans le cadriciel d'expérimentation développé au Chapitre 7.

1. https://github.com/LIHPC-Computational-Geometry/coupe
2. https://github.com/LIHPC-Computational-Geometry/scotch-rs
3. https://github.com/LIHPC-Computational-Geometry/metis-rs

Notations algorithmiques

La Table 1 recense les notations algorithmiques utilisées dans ce document.

Table 1 – Notations algorithmiques

Notation	Signification
$\emptyset$	Le tableau vide.
$\lvert S \rvert$	La taille du tableau S.
$S[i]$	Le i-ème élément du tableau S. Les tableaux sont indicés à partir de zéro.
$[0, .^{n}., 0]$	Le tableau de taille n ne contenant que des zéros.
$[f(i) \mid 0 \leq i < \lvert S \rvert]$	$[f(0), f(1), \ldots, f(\lvert S \rvert - 1)]$

La Table 2 décrit la liste des fonctions utilisées. Ces fonctions peuvent modifier leurs arguments. Par conséquent, il est préférable de les considérer comme des procédures.

Table 2 – Fonctions, triées par ordre alphabétique

Nom	Description
$append(S, s)$	Ajoute s à la fin du tableau S.
$argmin(S)$	Donne un indice i tel que $S[i]$ est le minimum de S.
$insertSorted(S, s)$	Ajoute s au tableau trié S, de sorte que S reste trié.
$pop(S)$	Pour un tableau S de taille non nulle, retire et donne le dernier élément de S.
$random(S)$	Donne un élément aléatoire du tableau S.
$sort(S)$	Trie le tableau S sur place par ordre croissant. Les tuples sont triés par ordre lexicographique.

Première partie

Problèmes, modèles et état de l’art

1 - Origines du partitionnement de maillage

Préalablement à la présentation de nos contributions, nous allons introduire le contexte et des définitions associés à notre travail. Tout d'abord, la Section 1.1 présente les problèmes soulevés lors de l'exécution de simulations numériques sur machines distribuées. Ensuite, la Section 1.2 (page 22) spécifie ces problèmes pour le cas des simulations reposant sur l'utilisation de maillages. Enfin, la Section 1.3 (page 29) présente les logiciels utilisés pour répondre à ces problèmes.

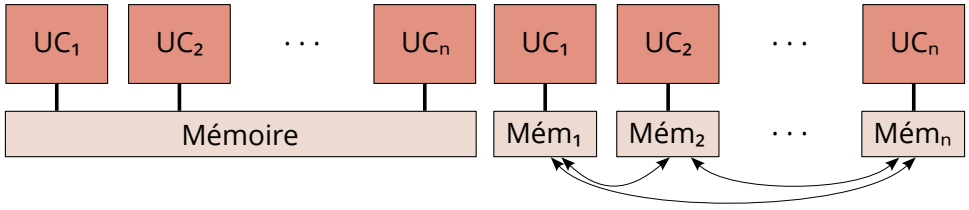
1.1 . Problématiques des codes de calcul sur machines distribuées

Un des intérêts du calcul haute performance est d'obtenir le résultat de calculs en le moins de temps (ou le moins d'énergie) possible. Pour atteindre cet objectif, outre acheter un grand nombre d'ordinateurs ou processeurs (ci-après appelés « unités de calcul »), il est aussi nécessaire de bien les utiliser.

Cette tâche est déjà difficile sur machine séquentielle – avec une seule unité de calcul – où de nombreux aspects de l'exécution doivent être pris en compte lors du développement d'un code pour maximiser la vitesse de calcul : *pipelining*, localité des données, prédiction de branche. Sur machine parallèle – avec plusieurs unités de calcul – deux coûts viennent s'y ajouter :

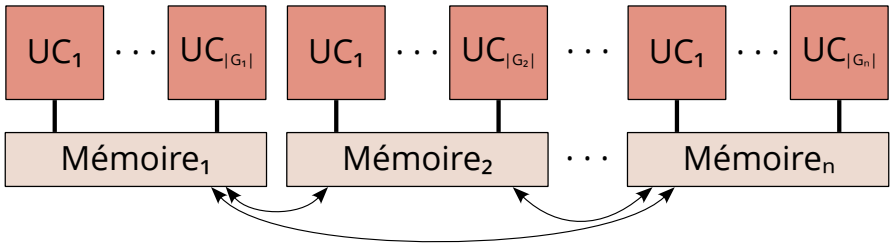
- les coûts de synchronisation, qui apparaissent lorsque une unité attend le travail d'une autre pour continuer, par exemple lors de contention de verrous, ou de collectives MPI bloquantes ;
- les coûts de communication, qui apparaissent lorsque deux unités doivent échanger des données, par exemple lors d'invalidations de cache, ou de transferts réseau.

Nous considérons dans ce travail les machines parallèles comme étant à mémoire distribuée, à mémoire partagée, ou bien les deux à la fois. Dans le cas des machines à mémoire distribuée, chaque unité de calcul dispose d'une mémoire propre, qui n'est pas accessible par les autres unités de calcul. Le développeur de code doit faire explicitement transférer l'information d'une unité de calcul vers une autre si besoin (voir Figure 1.1b). Pour une machine à mémoire partagée, les unités de calcul partagent la mémoire (voir Figure 1.1a). Elles peuvent donc écrire et lire des données en mémoire de façon concurrente. Contrairement au parallélisme en mémoire distribuée, l'information est transmise implicitement entre les uni-



(a) Architecture à mémoire partagée, où les unités de calcul partagent la mémoire.

(b) Architecture à mémoire distribuée, où les unités de calcul doivent échanger explicitement l'information de mémoire en mémoire.



(c) Architecture à mémoire hiérarchique, où les unités de calcul forment des groupes. Une unité de calcul partage la mémoire avec les unités de son groupe, mais doit explicitement transférer l'information vers les autres unités.

Figure 1.1 – Les différents types de machines parallèles qui nous intéressent. Les unités de calcul sont représentées par l'indication « UC ».

tés de calcul. Dans le cas des machines mixtes, à mémoire à la fois distribuée et partagée, nous avons usuellement une décomposition hiérarchique où n unités de calcul sont réparties en une famille $(G_i)_{0 \leq i < p}$ de p familles d'unités de calcul telles que (voir Figure 1.1c) :

1. Le nombre total d'unités de calcul est n, c'est-à-dire $\sum_{0 \leq i < p} |G_i| = n$;
2. Les unités de calcul d'une famille G_i partagent la mémoire entre elles;
3. Les familles d'unités fonctionnent en mémoire distribuée. La manière de communiquer explicitement entre familles peut se faire de plusieurs façons. Par exemple, une unité de calcul peut être dédiée aux communications.

Parmi les codes qu'il est possible d'exécuter sur ces machines parallèles, nous nous intéressons particulièrement aux codes de calcul adoptant le modèle de *Bulk Synchronous Parallel* [17]. Dans ce modèle, les unités de calcul travaillent de manière synchronisée, par itérations. À la fin de chaque itération, une barrière de synchronisation empêche une unité de calcul de commencer l'itération suivante avant que les autres ne soient prêtes. Pendant une itération, les unités de calcul peuvent calculer et com-

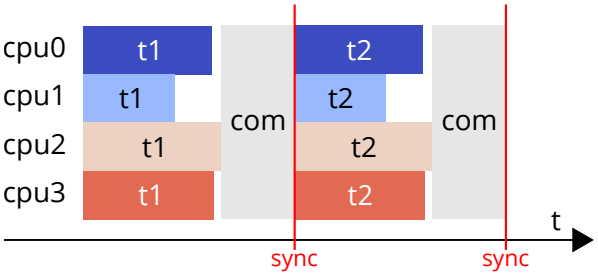


Figure 1.2 – Exemple d'exécution d'un code qui adopte le modèle BSP, sur une machine parallèle à quatre unités de calcul. Le cycle de calcul (`t1`, `t2`), communication (`com`) et synchronisation (`sync`) se répète jusqu'à la fin du code de simulation.

muniquer avec les autres. Par exemple, la Figure 1.2 illustre une exécution en BSP avec deux itérations où les unités effectuent du calcul (`t1` et `t2`), et des communications (`com`).

Dans l'optique de mesurer et contrôler le temps de simulation ressenti par l'utilisateur, une métrique intéressante est l'efficacité. Par efficacité, nous entendons, pour un entier k donné, le rapport entre le meilleur temps séquentiel, et k fois le temps théorique sur k unités de calcul. Idéalement, une simulation s'exécute k fois plus rapidement sur k unités de calcul, et son efficacité vaut 1. Sinon, son efficacité est inférieure à 1. Définissons maintenant formellement cette notion :

Définition 1.1 (Efficacité)**.** *Étant donné une simulation numérique S, considérons $t : \mathbb{N}^* \longrightarrow \mathbb{R}^+$ l'application donnant, pour un nombre d'unités de calcul, le meilleur temps d'exécution de S sur une machine à tant d'unités de calcul. L'efficacité de S est l'application $\eta : \mathbb{N}^* \longrightarrow \mathbb{R}^+$ telle que :*

$$\eta(k) = \frac{t(1)}{k \cdot t(k)}$$

Dans le cas des simulations adoptant le modèle BSP, où chaque itération se termine par une synchronisation, l'efficacité dépend de l'équilibre de la charge de calcul, car elle dépend du temps passé par l'unité de calcul la plus lente. Dans le pire des cas, la charge n'est pas du tout équilibrée et une seule unité fait tout le travail, le temps de calcul est alors égal au temps séquentiel et l'efficacité est inférieure à $1/k$. Dans le meilleur des cas, toutes les unités ont une charge de travail équitable par rapport à leurs capacités de calcul, finissent en même temps, et l'efficacité vaut 1. Nous nous intéressons donc au déséquilibre de charge, que nous définissons dans la Définition 1.2.

Définition 1.2 (Déséquilibre de charge d'une simulation numérique sous modèle BSP)**.** *Soit une simulation numérique S ayant adopté le modèle BSP.*

Cette simulation s'exécute sur k unités de calcul. Soit $(t_i)_{0\leq i<k}$ le temps passé par chaque unité pendant une phase de calcul. Le déséquilibre de charge de S est :

$$\max_{0\leq i<k} \frac{t_i}{T/k} - 1,$$

où $T = \sum_{0\leq i<k} t_i$.

Le déséquilibre est compris entre 0, quand chaque unité prend T/k de temps à calculer, et $k-1$, quand une seule unité fait tout le travail. Pour minimiser le temps de calcul de simulations numériques adoptant le BSP, il faut alors :

- distribuer équitablement le travail à effectuer sur l'ensemble des unités de calcul : une exécution du code sur k unités de calcul doit idéalement prendre k fois moins de temps qu'une exécution sur machine séquentielle ;
- lorsque les données de simulation sont de trop grande taille, les distribuer de façon à minimiser les communications nécessaires entre unités de calcul lors de l'exécution.

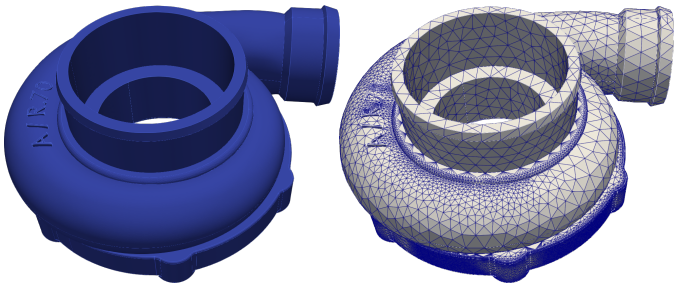
Les coûts de communications sont mieux définis par la section suivante pour les simulations qui nous intéressent : les simulations à base de maillages.

1.2 . Discrétisation de l'espace par un maillage

De nombreuses simulations numériques étudient des phénomènes physiques qui nécessitent de discrétiser un espace géométrique Ω. En mécanique des fluides par exemple, on modélise l'écoulement d'un liquide dans un réseau de tuyaux, ou l'écoulement de l'air le long d'un fuselage d'avion. Dans de tels cas, les équations physiques utilisées sont modélisées mathématiquement par des méthodes nécessitant de découper l'espace géométrique Ω en un ensemble d'éléments simples, appelés *mailles*, qui servent de support aux calculs mathématiques. Nous allons introduire dans cette section la notion de *maillage*, comme un ensemble de mailles qui discrétise Ω.

Un maillage, dans le cadre de cette thèse, consiste donc en un ensemble d'éléments géométriques, appelés mailles, qui recouvrent Ω, comme définit dans la Définition 1.3. Pour le lecteur intéressé, des définitions plus précises des concepts mathématiques sont présentées dans [18].

Définition 1.3 (Maillage et maille)**.** *Soit $\Omega \subset \mathbb{R}^3$ un ensemble borné et fermé. Alors, $\mathcal{M} \subset \mathcal{P}\left(\mathbb{R}^3\right)$ est un maillage de Ω si :*



(a) (b)

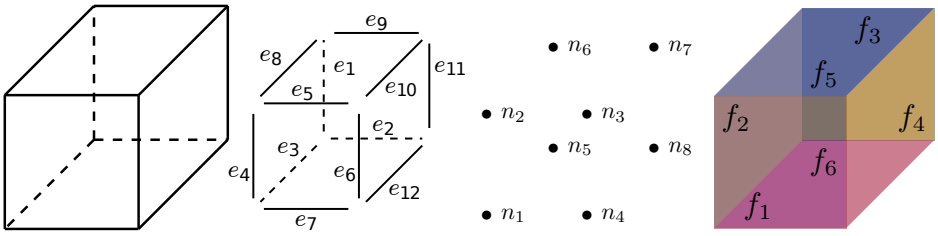
Figure 1.3 – Exemple de discrétisation d'un turbocompresseur. La figure (a) montre le modèle 3D du turbocompresseur. La figure (b) en montre une décomposition en maillage. Les mailles sont des tétraèdres délimitées par des arêtes bleues. Maillage issu du jeu de données *Hex Me If You Can* [19]

$$\begin{cases} \bigcup_{m \in \mathcal{M}} \overline{m} = \Omega; & (1.1) \\ \forall m, m' \in \mathcal{M},\ m \neq m' \implies \mathring{m} \cap \mathring{m}' = \emptyset. & (1.2) \end{cases}$$

Les éléments m de $\mathcal{M}$ sont appelés mailles. $\overline{m}$ dénote l'adhérence de la maille m, et $\mathring{m}$ l'intérieur de la maille m. Pour un ensemble X, $\mathcal{P}(X)$ dénote l'ensemble des parties de X.

La condition 1.1 assure que l'ensemble des mailles m recouvrent géométriquement Ω en entier, tandis que la condition 1.2 indique que deux mailles de $\mathcal{M}$ ne peuvent pas être superposées et ce en aucun point de Ω. À titre d'exemple, la Figure 1.3a montre un turbocompresseur modélisé en 3D, et la Figure 1.3b montre une décomposition de celui-ci en maillage, où toutes les mailles 3D sont des tétraèdres. Il est aussi courant de travailler sur des maillages très simples comme des grilles 2D. En 2D, les mailles sont généralement des triangles ou des quadrilatères. En 3D, ce sont des tétraèdres ou des hexaèdres. Par ailleurs, de manière usuelle, nous utiliserons les termes faces, arêtes et sommets pour caractériser les éléments de dimensions inférieures participant à décrire le bord des mailles, comme illustré sur la Figure 1.4.

Nous considérons des codes de simulations numériques qui effectuent des calculs pour chaque maille. Généralement, calculer une quantité physique ou mathématique en une maille nécessite des données présentes dans ses mailles « voisines ». L'ensemble des mailles dont les données sont nécessaires au calcul associé à m est appelé le voisinage de m. La notion mathématique de voisinage est définie dans la Définition 1.4.



(a) Une maille en trois dimensions.

(b) Les arêtes de la maille (a).

(c) Les nœuds de la maille (a).

(d) Les faces de la maille (a).

Figure 1.4 – Différentes composantes des mailles d'un maillage.

Définition 1.4 (Relation de voisinage)**.** *Soit $\mathcal{M}$ un maillage, alors $\mathcal{N} : \mathcal{M} \longrightarrow \mathcal{P}(\mathcal{M})$ est une relation de voisinage sur $\mathcal{M}$ si*

$$\forall m_1, m_2 \in \mathcal{M}, m_2 \in \mathcal{N}(m_1) \implies m_1 \in \mathcal{N}(m_2).$$

$\mathcal{N}(m)$ est le voisinage de m, et ses éléments sont les voisines de m.

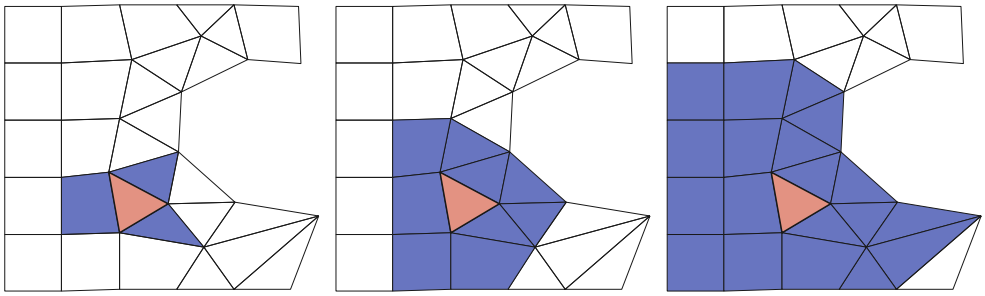
Cette définition très abstraite mérite d'être illustrée. En pratique, le voisinage d'une maille m est composé de mailles « géométriquement proches », voire adjacentes (dont l'adhérence partage arête ou sommet en 2D, voir la Figure 1.4), et dépend des schémas numériques utilisés par le code de simulation. Par exemple, pour calculer le gradient d'une quantité scalaire (comme la température), un code 2D peut nécessiter un voisinage par arête, un voisinage par sommet, ou encore un voisinage par sommet à une distance topologique de deux. La Figure 1.5 illustre ces trois cas de figure pour une maille triangulaire.

Pour exécuter une simulation numérique à base de maillages sur une machine parallèle, il faut distribuer la charge de travail et les données associées à chaque maille sur les différentes unités de calcul. Pour cela, une partition du maillage est créée, c'est l'objet de la définition qui suit :

Définition 1.5 (Partition d'un maillage)**.** *Soient $\mathcal{M}$ un maillage, k un entier naturel supérieur ou égal à deux, et $\Pi = (P_i)_{0 \leq i < k}$ une famille de parties de $\mathcal{M}$. Alors Π est une partition de $\mathcal{M}$ si ses éléments sont deux à deux disjoints, et leur union est le maillage $\mathcal{M}$:*

$$\begin{cases} \forall P, P' \in \Pi,\ P \neq P' \implies P \cap P' = \emptyset; \\ \bigcup_{P \in \Pi} P = \mathcal{M}. \end{cases}$$

Chaque maille est alors associée à une unité de calcul. Un exemple est montré sur la Figure 1.6, où le maillage de turbocompresseur de la



(a) Relation de voisinage par arêtes à distance topologique de un.

(b) Relation de voisinage par sommet à distance topologique de un.

(c) Relation de voisinage par sommet à distance topologique de deux.

Figure 1.5 – Trois cas de relations de voisinage pour un triangle en 2D. Les mailles voisines du triangle rouge, pour différentes relations de voisinage, sont en bleu.

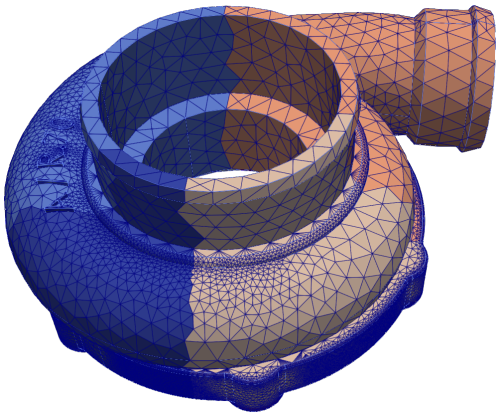


Figure 1.6 – Partitionnement du maillage de turbocompresseur en quatre parties. Chaque couleur représente une partie.

Figure 1.3b (page 23) est découpé en quatre parties, représentées par des couleurs différentes. Les communications s'effectuent alors pour les mailles à la frontière des parties, notion définie dans la Définition 1.6.

Définition 1.6 (Frontière d'une partie, maille fantôme)**.** *Soient $\mathcal{M}$ un maillage, $\mathcal{N}$ une relation de voisinage sur $\mathcal{M}$, Π une partition de $\mathcal{M}$, et une partie $P \in \Pi$. Alors, la frontière interne de P est l'ensemble des mailles de P ayant des voisines hors de P :*

$$InnerBound(P) = \{m \in P \mid \mathcal{N}(m) \backslash P \neq \emptyset\}\,.$$

La frontière externe de P est l'ensemble des mailles hors de P ayant des voisines dans P :

$$OuterBound(P) = \{m \in \mathcal{M} \backslash P \mid \mathcal{N}(m) \cap P \neq \emptyset\}.$$

Si les mailles de $OuterBound(P)$ sont dupliquées sur l'unité de calcul associée à P, ces doublons sont appelés mailles fantômes*.*

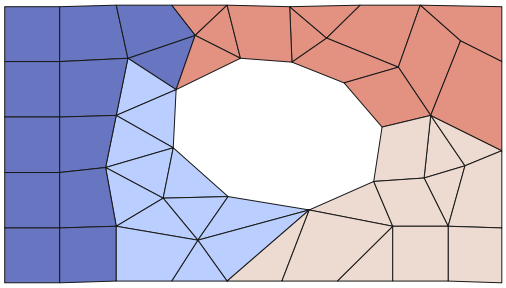
Pour bien comprendre l'intérêt des frontières internes et externes, il faut se rappeler que pour calculer une quantité en une maille m, il est le plus souvent nécessaire de connaitre les valeurs de cette quantité dans des mailles voisines de m, la notion de voisinage variant selon les calculs effectués. Dans le cas où m et ses voisines sont dans une même partie P, il n'y a pas de problème de cohérence de données. Dans le cas contraire, les mailles de $InnerBound(P)$ contiennent des données qui doivent être transmises de P aux autres unités de calcul afin que ces dernières puissent calculer la prochaine itération de la simulation. De manière symétrique, les mailles de $OuterBound(P)$ contiennent des données dont P a besoin pour calculer la prochaine itération.

Pour illustrer le concept de maille fantôme, sur la Figure 1.7a, un maillage 2D est partitionné en quatre parties. Les Figures 1.7b et 1.7c montrent en transparent les mailles fantômes présentes sur chaque partie, pour une relation de voisinage par arête et par sommet respectivement. Lorsqu'une simulation utilise des mailles fantômes, le nombre de mailles associées à chaque unité de calcul augmente. Par conséquent, il faut s'assurer que chaque unité dispose d'assez de mémoire pour y stocker les données associées aux mailles de la partie ainsi que celles des mailles fantômes.

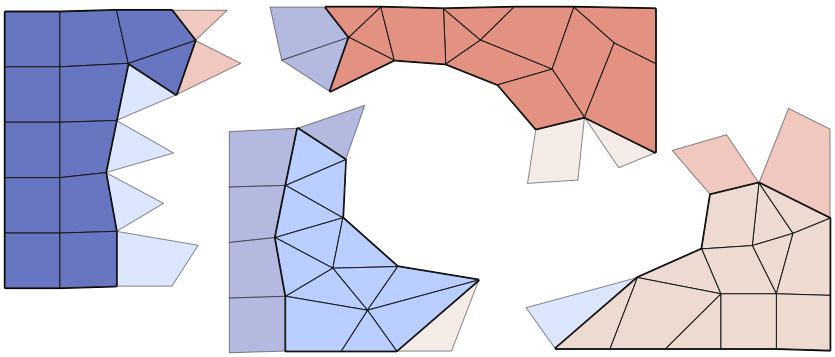
Une fois les mailles réparties sur les différentes unités de calcul, ces dernières effectuent les traitements associés aux mailles qu'elles possèdent. Nous considérons que le traitement de chaque maille requiert une certaine charge de calcul, et que cette charge ne dépend pas de l'ordre de traitement. Alors la charge de calcul requise pour traiter une partie du maillage est la somme des charges de calcul des mailles de cette partie. Nous pouvons ainsi spécifier le déséquilibre de charge (précédemment défini par la Définition 1.2, page 21) pour les simulations à base de maillages :

Définition 1.7 (Charge de calcul d'une partie, déséquilibre de charge d'une partition)**.** *Soient $\mathcal{M}$ un maillage, $\Pi = (P_i)_{0 \leq i < k}$ une partition de $\mathcal{M}$, et une application $w : \mathcal{M} \longrightarrow \mathbb{R}^+$ associant une charge de calcul à chaque maille de $\mathcal{M}$. Les charges de calcul $(c_i)_{0 \leq i < k}$ respectivement associées aux parties $P_i \in \Pi$ sont la somme des charges de calcul des mailles de la partie :*

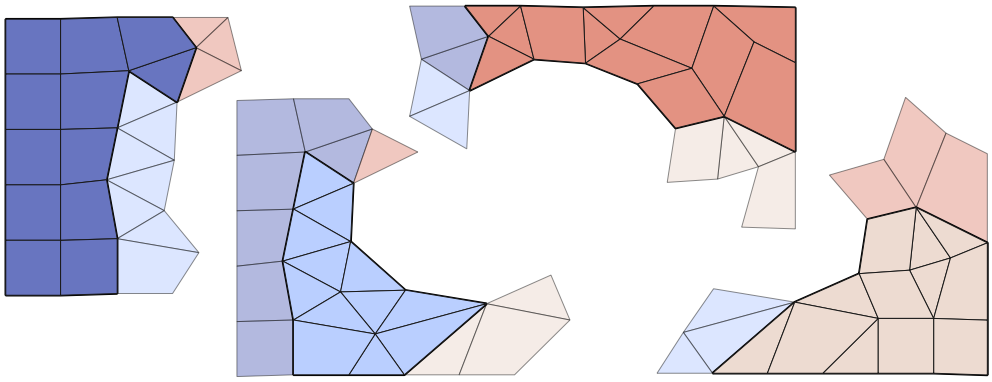
$$\forall 0 \leq i < k, c_i = \sum_{m \in P_i} w(m)$$



(a) Cas sans maille fantôme. Les unités de calcul ne stockent que les données associées aux mailles de leur partie.



(b) Même partitionnement que (a), mais avec des mailles fantômes, pour une relation de voisinage par arête.



(c) Même partitionnement que (a), mais avec des mailles fantômes, pour une relation de voisinage par sommet.

Figure 1.7 – Partitionnement d'un maillage 2D en quatre parties, sans (a), et avec (b,c) des mailles fantômes. Les mailles fantômes sont représentées avec des couleurs effacées. Remarquons que dans chaque figure, le nombre de mailles associées aux unités de calcul est plus grand que dans la figure précédente.

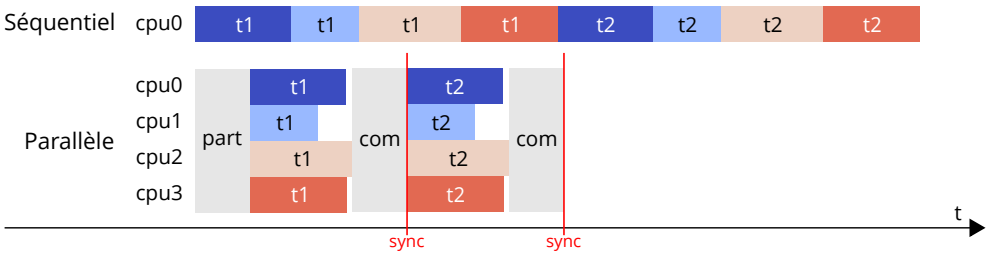


Figure 1.8 – Exécution d'une simulation numérique sur une vs quatre unités de calcul, avec le maillage de turbocompresseur.

La charge de calcul totale C est la somme de la charge de calcul de toutes les mailles :

$$C = \sum_{m \in \mathcal{M}} w(m)$$

Le déséquilibre de charge de la partition Π est :

$$Imb(\Pi) = \max_{P \in \Pi} \frac{c_i}{C/k} - 1$$

Par exemple, considérons l'exécution d'une simulation numérique sur le maillage de turbocompresseur, représentée sur la Figure 1.8. Sur machine séquentielle en haut, la simulation traite les mailles de chaque partie successivement, à chaque itération (`t1`, `t2`, ...). Sur machine parallèle, il faut d'abord partitionner le maillage et distribuer les parties à chaque unité de calcul (`part`). Ensuite, les unités de calcul traitent leur partie en parallèle pour la première itération (`t1`), avant de communiquer le résultat du traitement pour les mailles de leur frontière interne (`com`). Après une barrière de synchronisation, la deuxième itération peut commencer. Sur la figure, c'est la partie `cpu2` (en rose) qui prend le plus de temps de calcul. Les autres unités passent du temps à l'arrêt (en blanc, en attendant d'être synchronisées).

Diminuer le déséquilibre de charge de la partition permet de diminuer les temps d'arrêt, et ainsi diminuer le temps global de la simulation. Simultanément, diminuer la taille de la frontière des parties réduit le temps passé dans les phases de communication. Pour l'utilisateur d'un code parallèle de simulation, le but du partitionnement est de diminuer le temps global de la simulation en minimisant le déséquilibre et les communications. Il est aussi possible que la partition est une influence sur la qualité des résultats obtenus par la simulation, ou alors le nombre d'itérations requises pour obtenir ces résultats. Des définitions plus formelles sont données dans le chapitre suivant.

Sur la Figure 1.8, les charges de calcul restent constantes au fil des itérations. Il est aussi possible que les charges et le déséquilibre évoluent. Alors, les codes de calcul repartitionnent le maillage, par exemple périodiquement, ou lorsque le déséquilibre est jugé trop important.

Maintenant que nous avons présenté les problématiques soulevées lors de l'exécution, sur machines parallèles, de simulations BSP nécessitant des maillages, nous allons nous intéresser à l'outil de partitionnement. Plus particulièrement, la section suivante présente les logiciels utilisés pour développer ces outils.

1.3 . Logiciels pour le calcul haute performance

Le partitionnement de maillage peut être effectué à la volée, au début de l'exécution d'une simulation, voire plusieurs fois pendant l'exécution. Pour l'utilisateur de la simulation, le temps écoulé lors du partitionnement fait partie du temps d'exécution de la simulation et, il est par conséquent favorable de faire en sorte que ce temps écoulé soit le plus court possible. Nous nous intéressons donc ici aux logiciels pour le calcul haute performance, et la façon dont ils peuvent servir pour le partitionnement.

Les outils de partitionnement existants fonctionnent pour la plupart en mémoire distribuée et nous pouvons considérer deux scénarios :

- En début de simulation, le maillage et les données associées sont partitionnées et distribuées sur les différentes unités de calcul. Ce partitionnement initial peut être réalisé de manière séquentielle, concurrente en mémoire partagée, ou concurrente en mémoire distribuée selon la localité initiale des données [1] ;
- En cours de simulation, on peut constater que le déséquilibre de charge s'amplifie et envisager alors une étape de repartitionnement avec l'objectif de diminuer ce déséquilibre.

Les partitionneurs en mémoire distribuée tels que ParMetis [20] et PT-Scotch [21] ont l'avantage de pouvoir être exécutés sur les mêmes unités de calcul que l'application et, par conséquent, peuvent directement travailler sur le maillage dans la mémoire locale des machines. Ceci est intéressant quand le maillage ou les charges de calcul changent au cours de la simulation (par exemple lors du raffinement automatique de maillage). Les partitionneurs en mémoire partagée peuvent être utilisés en regroupant les données nécessaires sur une des plages mémoire.

Le calcul sur machine distribuée s'appuie généralement sur le standard MPI [22]. MPI est une interface de programmation (API), pour les langages de programmation C et Fortran, permettant à plusieurs « pro-

1. Par exemple, une des unités de calcul peut lire un fichier de maillage, ou bien plusieurs unités peuvent générer le maillage en parallèle.

cessus MPI » de se transmettre et de recevoir des messages. Il en existe plusieurs implémentations, dont OpenMPI [23] et MPICH [24], deux implémentations généralistes, ainsi que MPC [25], développée au CEA en collaboration avec Paratools. L'intérêt de MPI par rapport à l'API réseau du système d'exploitation est double :

1. MPI est portable et permet d'automatiquement tirer parti d'autres couches réseaux qu'Ethernet, comme Infiniband ou BXI, et ainsi atteindre des débits plus élevés et des temps de latence plus faibles [26] ;
2. MPI est plus facile d'utilisation, et offre plus de fonctionnalités.

Cependant, MPI seul ne permet pas de pleinement utiliser des machines hiérarchiques (vues sur la Figure 1.1c, page 20), où de nombreuses unités de calcul partagent chaque mémoire. En effet, bien qu'il soit possible d'exécuter un processus MPI par unité de calcul, le transfert de messages engendre une duplication de données en mémoire qui n'est pas négligeable [25]. Ce problème s'aggrave lors de l'utilisation de mailles fantômes. Il faut alors des logiciels explicitement conçus pour la mémoire partagée. Nous citons notamment :

- L'API *pthreads* [27], pour le langage de programmation C, permet de créer et de gérer des processus légers, ou fils d'exécution, qui partagent le même espace mémoire. Elle offre aussi des primitives de synchronisation, comme des verrous et des mécanismes de notification, qui permettent aux fils d'exécution de se coordonner plus rapidement qu'avec des messages MPI. Nous pouvons noter que l'API pthreads est utilisée par PT-Scotch [21].
- OpenMP [28], pour le C, le C++ et le Fortran, abstrait les fils d'exécution et permet de paralléliser automatiquement des traitements sur des tableaux, d'exécuter des tâches de manière asynchrone et de décharger du travail vers des accélérateurs comme les GPU.

Par ailleurs, il existe aussi d'autres logiciels permettant de construire des codes parallèles. Par exemple, Kokkos [29], pour le langage C++, permet d'abstraire le matériel sous-jacent, afin de décrire du parallélisme à la fois en mémoire partagée et en mémoire distribuée. À l'instar d'OpenMP, la bibliothèque Kokkos permet de transférer une partie du calcul sur GPU. Elle est utilisée par Zoltan 2, le partitionneur de la boite à outils Trilinos [2], pour l'implémentation d'algorithmes géométriques hautement efficaces.

Finalement, il existe d'autres paradigmes de programmation haute performance. Le langage Rust [13] est un langage de programmation bas niveau, qui offre un accès au matériel similaire à celui du C et du C++, tout en évitant diverses classes de bogues mémoires, dont les erreurs de segmentation et les accès concurrents. Pour ce faire, le langage impose des règles à la compilation sur la durée de vie des objets en mémoire et

leur nombre de références. Pour le parallélisme, Rust offre des primitives de synchronisation et de partage de données, similaires à celles de l'API pthreads, dans sa bibliothèque standard. Il existe aussi dans l'écosystème des bibliothèques supplémentaires :

- *Crossbeam* [30], qui expose des files MPMC (Multi-Producteur, Multi-Consommateur) et des structures de données pour le vol de travail ;
- *Rayon* [31], qui offre une interface de parallélisme automatique à base d'itérateurs. Elle permet d'écrire du code sous le paradigme du parallélisme de données.

Conclusion

Nous avons vu dans ce chapitre que le partitionnement de maillages permet d'équilibrer la charge de simulations numériques destinées à être exécutées sur machines à mémoire distribuée. Le partitionnement doit prendre en compte à la fois la charge de calcul et la quantité de données associées à chaque maille, de façon à diminuer le temps d'exécution total de la simulation. Le partitionnement doit aussi être le plus rapide possible et, pour cela les outils sont développés à l'aide de logiciels haute performance. Dans le chapitre suivant, nous allons traiter de la façon dont les maillages sont partitionnés.

2 - Quatre problèmes de partitionnement de maillage

Dans le chapitre précédent, nous avons introduit des éléments de contexte relatifs à la simulation numérique et au besoin d'outils de partitionnement de maillage. Le partitionnement de maillage se doit de trouver une partition qui minimise le temps d'exécution de la simulation, en minimisant le déséquilibre et les communications. En pratique, la façon dont ce problème est posé dépend des caractéristiques de la simulation. C'est pourquoi dans ce chapitre, nous présentons le problème de partitionnement de maillages, puis quatre de ses variantes : le partitionnement de nombres, le partitionnement géométrique, le partitionnement topologique et le partitionnement sous contrainte mémoire.

Commençons par définir le problème de partitionnement de maillage. Celui-ci consiste à trouver une partition du maillage qui équilibre la charge de calcul associée aux parties tout en minimisant le coût associé à la partition. Une formulation de ce problème est donnée dans la Définition 2.1 :

Définition 2.1 (k-partitionnement de maillage)**.** *Étant donné un maillage $\mathcal{M}$ et une application $w : \mathcal{M} \longrightarrow \mathbb{R}^+$ associant une charge de calcul à chaque maille de $\mathcal{M}$, le problème du k-partitionnement de maillage consiste à trouver une famille $\Pi = (P_i)_{0 \leq i < k}$ de sous-ensembles de $\mathcal{M}$ qui satisfait :*

- **Contrainte de partition :** *Π est une partition de $\mathcal{M}$, telle que définie dans la Définition 1.5 (page 24);*
- **Objectif d'équilibre :** *le déséquilibre (cf. Définition 1.7, page 26) de Π est minimal;*
- **Objectif de moindre sur-coût :** *un coût (par exemple, celui lié aux temps de communication) induit par la partition Π de $\mathcal{M}$ est minimal.*

Le sur-coût représente une métrique d'intérêt tel que le temps de communication, ou le temps passé à accéder à des ressources de calcul partagées.

Ce problème est très générique, d'abord parce que les maillages sont des entités complexes, riches en information et pouvant être approchés sous différents angles, mais aussi parce que chaque simulation possède des besoins différents, de part son utilisation des ressources matérielles de calcul, son paradigme de programmation et les structures de données utilisées.

D'autre part, outre le fait que le problème est NP-difficile [32], [33], il impose d'optimiser pour deux objectifs, qui peuvent être contradictoires :

par exemple, la partition où toutes les mailles sont dans une partie a un déséquilibre maximal, mais n'engendre aucune communication. De l'autre côté, minimiser le déséquilibre peut augmenter le sur-coût. En pratique, les algorithmes existants utilisés pour le partitionnement résolvent le problème où l'un des objectifs est reformulé sous forme de contrainte [8], [9], [34]–[37]. Parmi ces problèmes, nous en présentons quatre, que nous allons définir dans les sections suivantes.

2.1 . Partitionnement de nombres

Une façon simplifiée d'aborder le problème général de partitionnement de maillage est de ne pas prendre en compte l'objectif de moindre sur-coût. Il s'agit de répartir les charges de telle sorte que l'unité de calcul ayant le plus de travail fasse attendre les autres le moins possible. Ce problème est décrit dans la Définition 2.2.

Définition 2.2 (k-partitionnement de nombres)**.** *Étant donné un maillage $\mathcal{M}$ et une application $w : \mathcal{M} \longrightarrow \mathbb{R}^+$ associant une charge de calcul à chaque maille de $\mathcal{M}$, le problème du k-partitionnement de nombres consiste à trouver une famille $\Pi = (P_i)_{0 \leq i < k}$ de sous-ensembles de $\mathcal{M}$ qui satisfait :*

- **Contrainte de partition :** *Π est une partition de $\mathcal{M}$, telle que définie dans la Définition 1.5 (page 24);*
- **Objectif de moindre temps :** *la charge de calcul la plus élevée de celle des ensembles P_i, définie comme* $\max_{0 \leq i < k} \sum_{m \in P_i} w(m)$*, est minimale.*

La Figure 2.1 montre deux exemples de partitionnement de nombres. Les deux configurations disposent du même ensemble de nombres. Dans la première configuration, la deuxième et la troisième partie sont les plus chargées avec une somme de poids égale à huit. Dans la seconde configuration, le poids de chacune des parties est égal à sept. La seconde configuration a donc un poids de partie maximal plus faible, ce qui en pratique conduit à des temps d'exécution plus faibles. La seconde configuration est donc de meilleure qualité pour ce problème.

Notons que l'objectif de moindre temps ne concerne que la partie la plus chargée, car c'est celle-ci qui ralentit l'exécution dans le modèle de *Bulk Synchronous Parallel* donné en Section 1.1 (page 19).

Lors de la résolution de ce problème, aucune contrainte sur le sur-coût de la partition n'est pris en compte. Ainsi, rien n'empêche une maille et ses voisines d'être dans des parties différentes. En pratique, cela conduit à des coûts de communication arbitrairement élevés et, si le code de simulation duplique les mailles à la frontière des parties (mailles dites *fantômes*, cf. la Définition 1.6, page 25), des coûts mémoire arbitrairement élevés. Dans ce cas, pour atténuer l'impact du coût des communication sur le temps

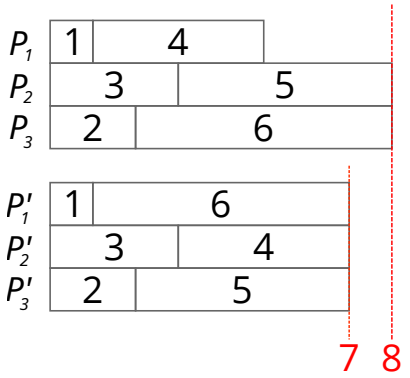


Figure 2.1 – Deux exemples de partitions, (P_1, P_2, P_3) et (P'_1, P'_2, P'_3), d'un maillage où la charge des mailles sont les $\{1, 2, 3, 4, 5, 6\}$. Dans la première partition, P_2 et P_3 ont une charge de calcul de 8, alors les unités de calcul associées feront attendre celle qui traite P_1. La deuxième partition a une charge de calcul maximale de 7, et peut être donc traitée plus rapidement que la première.

d'exécution de la simulation, il est possible de recouvrir le calcul par des communications [38], c'est-à-dire initier les transferts de données au plus tôt, parfois même pendant le temps de calcul, de façon à limiter le temps passé à communiquer après la phase de calcul.

2.2 . Partitionnement géométrique

Contrairement au partitionnement de nombres qui ne prend ni en compte la topologie, ni la géométrie du maillage, l'idée du partitionnement géométrique est de découper le domaine géométrique Ω que le maillage modélise, afin d'obtenir des parties aux frontières moins grandes. De cette façon, le sur-coût d'une partition peut être plus faible qu'une partition obtenue en résolvant le problème de partitionnement de nombres. Cependant, bien qu'il soit possible de parfaitement partitionner un domaine continu rien qu'avec des lignes droites [39], [40], cela n'est pas prouvé pour un domaine discret comme un maillage. C'est pourquoi les méthodes géométriques travaillent typiquement sous une contrainte d'équilibre. Nous donnons maintenant une définition du k-partitionnement de maillages sous contrainte d'équilibre :

Définition 2.3 (k-partitionnement de maillage sous contrainte d'équilibre)**.** *Soient un maillage $\mathcal{M}$, une application $w : \mathcal{M} \longrightarrow \mathbb{R}^+$ associant une charge de calcul à chaque maille de $\mathcal{M}$, une tolérance $\varepsilon \in [0; k-1]$ sur le déséquilibre, le problème du k-partitionnement sous contrainte d'équilibre consiste à trouver une famille $\Pi = (P_i)_{0 \leq i < k}$ de sous-ensembles de $\mathcal{M}$ qui*

satisfait :

- **Contrainte de partition :** Π *est une partition de* $\mathcal{M}$*, telle que définie dans la Définition 1.5 (page 24);*
- **Contrainte d'équilibre :** *pour toute partie* $P_i \in \Pi$*, la charge de calcul de la partie est inférieure à* $1 + \varepsilon$ *fois la moyenne :*

$$\max_{0 \leq i < k} \sum_{m \in P_i} w(m) \leq (1 + \varepsilon) \cdot \sum_{m \in \mathcal{M}} w(m)/k.$$

Notons que la Définition 2.3 ne repose sur aucune notion géométrique. Cependant, bien que cela ne soit pas requis, le problème est souvent résolu en utilisant des méthodes géométriques.

Un exemple de partition obtenue par une approche géométrique est illustrée à la Figure 2.2, où un maillage 2D composé de triangles et quadrilatères se trouve en arrière-plan. Nous appliquons ici un algorithme dit par bissection récursive de coordonnées (RCB) [8]. Pour cela, chaque maille est modélisée par son barycentre, représenté par un plus (+) violet, puis le nuage de points est découpé récursivement selon les axes. RCB découpe de telle sorte à ce que les sommes des charges des mailles de chaque côté d'une découpe soient suffisamment proches l'une de l'autre – ici, $1 + 0.05$ fois la moitié de la charge totale. Une découpe est d'abord faite selon les abscisses en $x = 6{,}2$, puis selon les ordonnées en $y = 4{,}6$ et $y = 6{,}1$. Nous obtenons alors quatre parties identifiées par des couleurs différentes.

Remarquons que le nombre de mailles par partie n'est pas le même entre chaque partie. Dans cet exemple, les mailles en bas à gauche du maillage ont une charge de calcul plus élevée que les autres.

2.3 . Partitionnement topologique

Les problèmes des Définitions 2.2 et 2.3 traitent de l'équilibre de charge du code de calcul et minimisent ou contraignent de déséquilibre de charge de calcul (cf. Définition 1.7, page 26). Ils ne traitent pas des coûts de communication, c'est-à-dire qu'ils ne cherchent pas à minimiser la quantité de données aux frontières $OuterBound$ et $InnerBound$ (cf. Définition 1.6, page 25).

En pratique, tenir compte des cos de communication revient à utiliser une relation de voisinage (cf. Définition 1.4, page 23) significative pour le code de calcul, et à modéliser les coûts de communication par des « poids » associés aux transferts de données entre voisins. Ceci se traduit le plus souvent par l'utilisation d'une structure de graphe [1], [41] ou d'hypergraphe [42], où les sommets du graphe correspondent aux mailles du maillage à partitionner, et les arêtes ou hyperarêtes à la relation de voi-

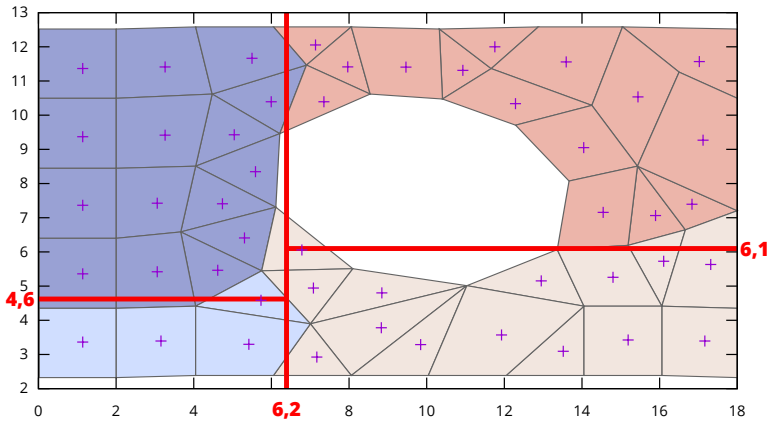


Figure 2.2 – Nuage de points, issu d'un maillage 2D, partitionné par bissections récursives de coordonnées (RCB). Le nuage de points formé du barycentre des mailles, représentés par des plus (+) violets, est découpé d'abord selon l'axe des abscisses, puis selon l'axe des ordonnées. La première découpe est faite selon l'axe des abscisses à $x = 6.2$, afin d'avoir des charges similaires à gauche et à droite. Ensuite, les deux bouts sont découpés de la même façon selon l'axe des ordonnées à $y = 4.6$ et $y = 6.1$.

sinage. Un exemple classique utilisé par les outils de partitionnements existants [41] est le *graphe dual*, défini par la Définition 2.4.

Définition 2.4 (Graphe dual d'un maillage)**.** *Soient $\mathcal{M}$ un maillage et $\mathcal{N}$ une relation de voisinage sur $\mathcal{M}$. Le graphe dual de $\mathcal{M}$ selon la relation de voisinage $\mathcal{N}$ est le couple (V, E) tel que :*

$$\begin{cases} V = \mathcal{M} \\ E = \{\{m, m'\} \mid m, m' \in \mathcal{M}; m \in \mathcal{N}(m')\} \end{cases}$$

Un exemple de graphe dual est donné sur la Figure 2.3. La somme du poids des arêtes coupées (représentées en pointillés sur la figure) est appelé la *coupe*, et est utilisée pour estimer le volume des communications associées à la partition. Plus formellement, nous pouvons considérer la Définition 2.5.

Définition 2.5 (Coupe)**.** *Soient un graphe (V, E) et une partition Π de V, où nous notons $\Pi(v)$ la partie dans laquelle est $v \in V$, définissons $cut(E, \Pi)$, l'ensemble des arêtes de E ayant des sommets dans deux parties différentes de Π :*

$$cut(E, \Pi) = \left\{\{v, v'\} \in E \mid \Pi(v) \neq \Pi(v')\right\}.$$

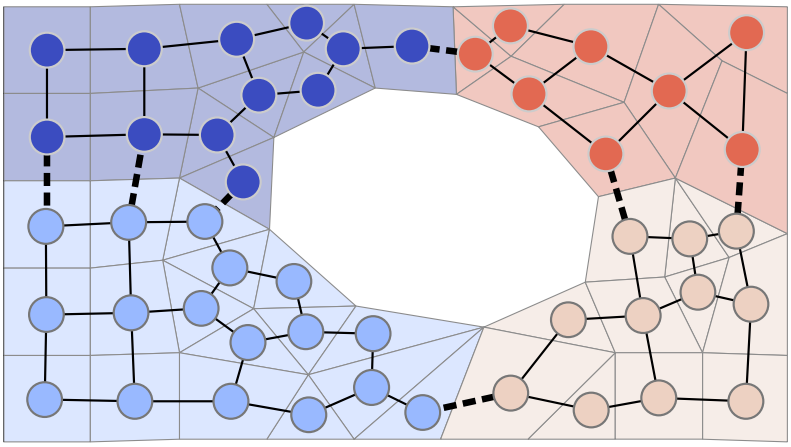


Figure 2.3 – Graphe associé à un maillage 2D pour une relation de voisinage par arête (cf. Définition 1.4), partitionné en quatre parties. Les arêtes coupées sont représentées en pointillés gras et relient deux sommets de parties différentes.

Soit une application $w : E \longrightarrow \mathbb{R}^+$*, donnant un poids à chaque arête de* E*, alors la coupe de* Π *désigne la somme des poids des arêtes de* $cut(E, \Pi)$ *:*

$$coupe(E, \Pi) = \sum_{e \in cut(E,\Pi)} w(e).$$

Certains partitionneurs [2], [7], [43], [44] utilisent un hypergraphe pour représenter la relation de voisinage, comme définit dans la Définition 2.6. L'hypergraphe représente plus fidèlement la relation de voisinage, et permet donc de mieux modéliser les communications [42].

Définition 2.6 (Hypergraphe dual d'un maillage)**.** *Soient* $\mathcal{M}$ *un maillage et* $\mathcal{N}$ *une relation de voisinage sur* $\mathcal{M}$*. L'hypergraphe dual de* $\mathcal{M}$ *selon la relation de voisinage* $\mathcal{N}$ *est le couple* (V, H) *tel que :*

$$\begin{cases} V = \mathcal{M}, \\ H = \{\mathcal{N}(m) \cup \{m\} \mid m \in \mathcal{M}\}. \end{cases}$$

Une méthode classiquement utilisée pour minimiser le volume des communications lors de l'utilisation du modèle d'hypergraphe est appelé la $\lambda - 1$ *cut* [42] (« *lambda cut* »), définie dans la Définition 2.7.

Définition 2.7 ($\lambda - 1$ *cut*)**.** *Soient un hypergraphe* (V, H)*, une application* $w : H \longrightarrow \mathbb{R}^+$*, donnant la quantité de données de chaque hyperarête de* H*, une partition* Π *de* V*, où nous notons* $\Pi(v)$ *la partie dans laquelle est* $v \in V$*, nous définissons* $\lambda : H \longrightarrow \mathbb{N}^*$ *comme étant le nombre de parties de* Π *qui intersectent chaque hyperarête :*

$$\lambda(h) = |\{\Pi(v) \mid v \in h\}|.$$

La $\lambda - 1$ cut *(dite «* lambda cut *») est définie comme telle :*

$$lambdacut(H, \Pi) = \sum_{h \in H} (\lambda(h) - 1) \cdot w(h).$$

L'expression $\lambda(h) - 1$ représente le nombre de fois que les données associées à l'hyperarête h doivent être échangées entre unités de calcul ou dupliquées en mémoire. La $\lambda - 1$ *cut* représente la quantité totale de données échangées (ou dupliquées).

Une fois la représentation choisie (graphe ou hypergraphe), il est possible de définir un problème de partitionnement topologique à l'aide de la Définition 2.8.

Définition 2.8 (k-partitionnement topologique)**.** *Soient un maillage* $\mathcal{M}$*, une application* $w : \mathcal{M} \longrightarrow \mathbb{R}^+$ *associant une charge de calcul à chaque maille de* $\mathcal{M}$*, une tolérance* $\varepsilon \in [0; k-1]$ *sur le déséquilibre, le problème du* k*-partitionnement topologique consiste à trouver une famille* $\Pi = (P_i)_{0 \leq i < k}$ *de sous-ensembles de* $\mathcal{M}$ *qui satisfait :*

- **Contrainte de partition :** Π *est une partition de* $\mathcal{M}$*, telle que définie dans la Définition 1.5 (page 24);*
- **Contrainte d'équilibre :** *pour toute partie* $P_i \in \Pi$*, la charge de calcul de la partie est inférieure à* $1 + \varepsilon$ *fois la moyenne :*

$$\max_{0 \leq i < k} \sum_{m \in P_i} w(m) \leq (1 + \varepsilon) \cdot \sum_{m \in \mathcal{M}} \frac{w(m)}{k};$$

- **Objectif de moindre sur-coût :** *la représentation des communications (coupe ou* $\lambda - 1$ cut*) est minimale.*

Ce problème n'a de limite que le choix de la représentation sous-jacente (graphe, hypergraphe) et du modèle de communication (coupe, $\lambda - 1$ *cut*). Les graphes et même les hypergraphes sont des visions simplifiées des échanges entre les unités de calcul. Les applications se reposent généralement sur des sémantiques plus complexes mélangeant les communications point-à-point, globales, pseudo-globales, unidirectionnelles, parfois recouvertes par du calcul [45]. Il est plus difficile de modéliser toutes ces subtilités, et à notre connaissance les outils actuels de partitionnement ne les prennent pas en compte.

2.4 . Partitionnement sous contrainte mémoire

Même si de nombreux outil ne résolvent que le problème présenté en Définition 2.8 [3]–[7], certaines problématiques restent inexplorées. C'est le cas par exemple du contrôle direct de la consommation mémoire induite par la partition. Cette métrique est importante, car dépasser la capacité mémoire d'une unité empêche souvent le code de calcul de s'exécuter. Ce problème s'intensifie quand le code duplique les mailles à la frontière (cf. la Définition 1.6) pour éviter certains échanges de données, ce qui augmente la capacité mémoire requise pour leur exécution. Ce coût peut être modélisé par la définition suivante :

Définition 2.9 (k-partitionnement sous contrainte mémoire)**.** *Soient un maillage $\mathcal{M}$, une application $w : \mathcal{M} \longrightarrow \mathbb{R}^+$ associant une charge de calcul à chaque maille de $\mathcal{M}$, une application $\mu : \mathcal{M} \longrightarrow \mathbb{R}^+$ donnant la taille mémoire requise par chaque maille, une famille $(M_i)_{0\leq i<k}$ de capacités mémoire, le problème du k-partitionnement sous contrainte mémoire consiste à trouver une famille $\Pi = (P_i)_{0\leq i<k}$ de sous-ensembles de $\mathcal{M}$ qui satisfait :*

- **Contrainte de partition :** *Π est une partition de $\mathcal{M}$, telle que définie dans la Définition 1.5 (page 24);*
- **Objectif d'équilibre :** *la charge de calcul la plus élevée de celle des ensembles P_i, définie comme $\max_{0\leq i<k} \sum_{m\in P_i} w(m)$, est minimale;*
- **Contrainte mémoire :** *l'occupation mémoire des mailles et des mailles fantômes associées à chaque ensemble P_i est plus faible que M_i :*

$$\max_{0\leq i<k} \sum_{m\in P_i \cup OuterBound(P_i)} \mu(m) < M_i.$$

Ce problème a été étudié dans [46] où le maillage $\mathcal{M}$ et la relation de voisinage $\mathcal{N}$ sont modélisés par un graphe biparti pondéré $\mathcal{B} = (C, D, E)$, où C est l'ensemble des traitements à effectuer, $D = \mathcal{M}$ est l'ensemble des mailles, et E est l'ensemble des dépendances de données pour chaque calcul. L'ensemble E est $\{\{c, m\} \mid c \in C; m \in \mathcal{M}\}$ tel que le traitement c dépend des données de m. La Figure 2.4 montre un exemple de maillage et le graphe biparti associé, dans le cas d'un traitement sur les mailles ($C = \mathcal{M}$) et où les mailles partageant une arête sont voisines.

2.5 . Autres intérêts au partitionnement

Les sections précédentes présentent quatre variantes du problème de partitionnement de maillage. En pratique, elles ne suffisent pas à couvrir toutes les métriques d'intérêt pour une simulation physique.

Par exemple, les quatre problèmes précédents supposent que la partition ne modifie que le temps d'exécution de l'application. Cependant, elle peut aussi influencer la valeur des résultats lors des méthodes de décomposition de domaine [47], ou le nombre d'itérations avant convergence

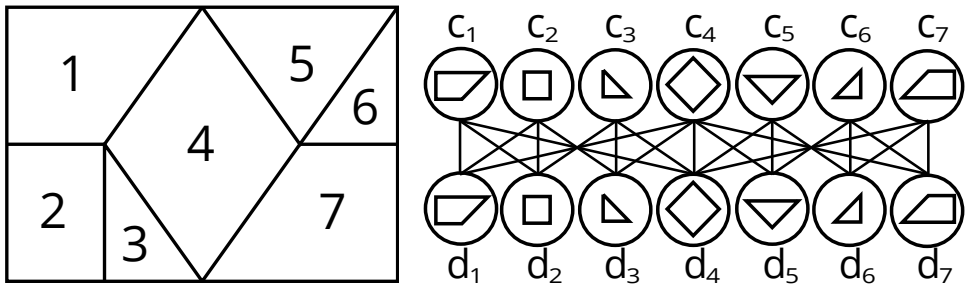


Figure 2.4 – Exemple de représentation d'un maillage 2D en graphe biparti, pour un traitement sur les mailles et des dépendances de données sur les arêtes.

des solveurs linéaires [48]. Il faut alors optimiser ou contraindre d'autres propriétés, telles que le rapport d'aspect des parties [49], [50].

Par ailleurs, les problèmes énoncés précédemment traitent du partitionnement initial, au début de l'exécution de l'application. Il est aussi possible de les étendre au re-partitionnement pour l'équilibrage dynamique de charge [51], [52]. Lorsque la charge des mailles change au cours de l'exécution d'une simulation, il est possible, de modifier la partition pour re-équilibrer la charge. Il faut alors en plus minimiser la quantité de données à transférer entre les machines pour distribuer la nouvelle partition.

Enfin, ces problèmes considèrent un équilibrage de charge ou les parties doivent être à charge égale. Lorsque les unités de calcul sont de différent types, par exemple, des CPU et des GPU, est aussi possible de les définir avec des poids cibles pour chaque partie.

3 - Algorithmes pour le partitionnement de maillage

Le chapitre précédent présentait plusieurs définitions du problème de partitionnement de maillages. Nous nous intéressons maintenant à des algorithmes classiquement utilisés pour résoudre chacun de ces problèmes.

3.1 . Algorithmes pour le partitionnement de nombres

Le problème de partitionnement de nombres tel que défini à la Définition 2.2 est un problème NP-difficile [32], [33]. En pratique, seules des heuristiques sont utilisées pour le résoudre. Nous présentons dans cette section deux heuristiques : l'heuristique gloutonne [34] et l'heuristique de Karmarkar et Karp [35].

Commençons par l'heuristique gloutonne, décrite dans l'algorithme 1 (voir si besoin les notations à la page 15). Cette heuristique place les poids un à un, du plus lourd au moins lourd, dans la partie la moins chargée. Elle a été développée par Horowitz et Sahni [34].

Algorithme 1 Algorithme glouton, partitionnant l'ensemble de nombre S en k parties

function Glouton(S, k)
    $\Pi \leftarrow [0, \overset{|S|}{\ldots}, 0]$
    $\Sigma \leftarrow [0, \overset{k}{\ldots}, 0]$
    $S' \leftarrow [(S[i], i) \mid 0 \leq i < |S|]$
    $sort(S')$
    while $S' \neq \emptyset$ **do**
        $(s, i) \leftarrow pop(S')$
        $p \leftarrow argmin(\Sigma)$
        $\Pi[i] \leftarrow p$
        $\Sigma[p] \leftarrow \Sigma[p] + s$
    end while
    return Π
end function

L'algorithme 1 effectue une visite de S dans l'ordre décroissant selon les poids. Cette visite peut se faire à l'aide d'un tableau trié (ce qui est le cas ci-dessus), ou d'un tas. Dans les deux cas, nous sauvegardons l'indice

de chaque poids sous forme de couple $(poids, indice)$ dans S' de façon à pouvoir créer la partition Π, où $\Pi[i]$ est le numéro de la partie du poids i. L'algorithme suit le poids de chaque partie à l'aide du tableau Σ, où $\Sigma[i]$ est le poids de la $i$ème partie au cours de l'exécution. Enfin, la boucle *while* itère sur S' et assigne chaque poids à la partie la moins chargée, puis met à jour le tableau Σ des poids de parties.

Pour un nombre de poids n largement supérieur à k, la complexité de l'algorithme glouton est bornée par celle du tri, $O(n \cdot \log n)$.

Les résultats de l'algorithme glouton ne sont pas forcément des solutions : prenons par exemple $S = [4, 3, 4, 6, 3]$ et $k = 2$. L'algorithme effectue les opérations notées dans la Table 3.1 pour conduire à la partition $[[6, 3], [4, 4, 3]]$ (déséquilibre $11/10 - 1 = 0{,}1$), alors que la partition $[[6, 4], [4, 3, 3]]$ a un déséquilibre nul.

Table 3.1 – Exemple d'exécution de l'algorithme glouton. De haut en bas, les poids de S sont successivement déplacés dans la partie la plus légère (P_0 ou P_1). Le poids des parties est noté dans les deux dernières colonnes.

S, trié	P_0	P_1	$\Sigma[0]$	$\Sigma[1]$
$[6, 4, 4, 3, 3]$	$[\,]$	$[\,]$	0	0
$[4, 4, 3, 3]$	$[6]$	$[\,]$	6	0
$[4, 3, 3]$	$[6]$	$[4]$	6	4
$[3, 3]$	$[6]$	$[4, 4]$	6	8
$[3]$	$[6, 3]$	$[4, 4]$	9	8
$[\,]$	$[6, 3]$	$[4, 4, 3]$	9	11

Karmarkar et Karp [35] proposent une autre approche pour le partitionnement de nombres. Au lieu de placer le poids le plus lourd dans la partie la plus légère, il s'agit de placer les deux poids les plus lourds dans deux parties différentes. Une version sans retour arrière, ou *backtracking*, pour un partitionnement en deux parties est décrite dans l'algorithme 2 (voir si besoin les notations à la page 15).

L'idée de l'algorithme de Karmarkar et Karp est que l'action de placer les deux plus grands poids de S dans deux parties différentes est équivalente à celle de placer leur différence dans une des deux parties. Il est donc possible de transformer le tableau S vers un nouveau tableau qui a un élément de moins, mais qui donne le même déséquilibre. À la sortie de la boucle *while*, S n'a plus qu'un seul élément : le déséquilibre de charge de la partition trouvée par l'algorithme.

La version de l'algorithme sans backtracking ne donne pas la partition. Pour l'obtenir, il faut garder l'historique des couples de poids placés dans

deux parties différentes, et effectuer une phase de backtracking. L'algorithme 3 décrit la version de l'algorithme avec backtracking. Dans cette nouvelle version, l'indice de chaque poids est sauvegardé dans S' de façon à identifier le poids après le tri à la ligne 4. La première boucle *while* (lignes 5 à 10), similaire à celle de l'algorithme sans backtracking, sauvegarde l'indice des deux plus grands poids dans une nouvelle pile X, et réinsère leur différence avec l'indice d'un des deux poids (ici, i_1). Pour initialiser le backtracking, le dernier élément de S' est mis dans une des deux parties (ici, la partie n°0) à la ligne 13. Ensuite, la deuxième boucle *while* (lignes 14 à 17) dépile les couples de X, et place les poids dans deux parties différentes. À la ligne 16, l'algorithme assigne à $\Pi[i_2]$ la valeur $1 - \Pi[i_1]$ et non le contraire ($\Pi[i_1] \leftarrow 1 - \Pi[i_2]$), car nous avons choisi i_1 pour représenter la différence des deux plus grands poids à la ligne 8, alors $\Pi[i_1]$ est initialisé et $\Pi[i_2]$ ne l'est pas.

Algorithme 2 Heuristique de Karmarkar et Karp pour le cas du partitionnement en 2 parties, version sans backtracking

```
1 : function KarmarkarKarp(S)
2 :     sort(S)
3 :     while |S| ≥ 2 do
4 :         s_1 ← pop(S)
5 :         s_2 ← pop(S)
6 :         insertSorted(S, s_1 − s_2)
7 :     end while
8 :     imbalance ← pop(S)
9 :     return imbalance
10 : end function
```

La complexité de cet algorithme, avec ou sans backtracking, est aussi bornée par celle du tri et par le traitement de la première boucle *while*, et est de $O(n \cdot \log n)$.

Les résultats donnés par cet algorithme sont sensiblement meilleurs que ceux donnés par l'heuristique gloutonne [35]. Par exemple, le cas précédent $S = [4, 3, 4, 6, 3]$ et $k = 2$ donne bien la partition à déséquilibre nul. Les opérations effectuées dans ce cas sont notées dans la Table 3.2. Toutefois, cet algorithme est aussi une heuristique. Par exemple, pour $S = [10, 10, 6, 6, 6]$ et $k = 2$, l'algorithme donne la partition $[[10, 6, 6], [10, 6]]$ au déséquilibre de $22/19 - 1 = 0{,}16$, alors que la partition $[[10, 10], [6, 6, 6]]$ a un déséquilibre de $20/19 - 1 = 0{,}05$.

Algorithme 3 Heuristique de Karmarkar et Karp pour le cas du partitionnement en 2 parties, version avec backtracking

```
1 : function KarmarkarKarp(S)
2 :     X ← ∅
3 :     S' ← [(S[i], i) | 0 ≤ i < |S|]
4 :     sort(S')
5 :     while |S'| ≥ 2 do
6 :         (s1, i1) ← pop(S')
7 :         (s2, i2) ← pop(S')
8 :         insertSorted(S', (s1 − s2, i1))
9 :         append(X, (i1, i2))
10 :    end while
11 :    Π ← ∅
12 :    (imbalance, i) ← pop(S')
13 :    Π[i] ← 0
14 :    while X ≠ ∅ do                          ▷ Backtracking
15 :        (i1, i2) ← pop(X)
16 :        Π[i2] ← 1 − Π[i1]
17 :    end while
18 :    return Π
19 : end function
```

Table 3.2 – Exemple d'exécution de l'algorithme de Karmarkar et Karp avec backtracking. Les poids de S sont notés avec leur indice d'apparition dans le tableau original : $[4, 3, 4, 6, 3]$ donne $[(4, 0), (3, 1), (4, 2), (6, 3), (3, 4)]$, qui est ensuite trié.

S', trié	X	Π	P_0	P_1
$[(6,3),(4,0),(4,2),(3,1),(3,4)]$	[ ]	[_, _, _, _, _]	[ ]	[ ]
$[(4,2),(3,1),(3,4),(2,3)]$	$[(3,0)]$	[_, _, _, _, _]	[ ]	[ ]
$[(3,4),(2,3),(1,2)]$	$[(3,0),(2,1)]$	[_, _, _, _, _]	[ ]	[ ]
$[(1,4),(1,2)]$	$[(3,0),(2,1),(4,3)]$	[_, _, _, _, _]	[ ]	[ ]
$[(0,4)]$	$[(3,0),(2,1),(4,3),(4,2)]$	[_, _, _, _, 0]	[3]	[ ]
$[(0,4)]$	$[(3,0),(2,1),(4,3)]$	[_, _, 1, _, 0]	[3]	[4]
$[(0,4)]$	$[(3,0),(2,1)]$	[_, _, 1, 1, 0]	[3]	[4, 6]
$[(0,4)]$	$[(3,0)]$	[_, 0, 1, 1, 0]	[3, 3]	[4, 6]
$[(0,4)]$	[ ]	[0, 0, 1, 1, 0]	[3, 3, 4]	[4, 6]

Korf [33], [53] approfondit l'heuristique gloutonne, ainsi que l'heuristique de Karmarkar et Karp en proposant des versions « complètes », qui retournent une solution optimale. L'avantage des algorithmes de Korf est qu'ils peuvent arrêter de s'exécuter à tout moment et retourner la

meilleure solution jusque-là trouvée (par exemple, après un temps fixé, ou lors de la validation d'une contrainte d'équilibre de charge).

3.2 . Algorithmes pour le partitionnement géométrique

Les algorithmes de partitionnement géométriques travaillent sur des ensembles de points, possiblement pondérés. L'idée est de produire une partition non seulement à l'aide de la charge de calcul associée aux mailles – via la pondération – mais aussi à l'aide des coordonnées des mailles, afin de limiter indirectement la taille de la frontière des parties.

Notons d'abord la classe d'algorithmes à base de bissection. Ces algorithmes projettent les points sur un espace 1D – par exemple un axe, ou une courbe – puis résolvent le problème de médiane pondérée. Ce problème consiste à trouver un point pivot x tel que la somme des poids des points plus petits que x soit la plus proche de celle des points plus grands que x. C'est une généralisation de la médiane, où les poids sont tous unitaires. Un exemple d'algorithme de recherche de médiane pondérée est donné par l'algorithme 4. La fonction « MédianePondérée » donne, pour une famille X de points, une famille W de poids associés et une tolérance ε, un point x_p qui sépare X en deux parties tel que le déséquilibre des deux parties soit inférieur à ε.

Algorithme 4 Recherche dichotomique de la médiane pondérée

```
1 : function MédianePondérée(X, W, ε)
2 :     w_total ← Σ_{w∈W} w
3 :     w_ε_min ← w_total/2 · (1 − ε)
4 :     w_ε_max ← w_total/2 · (1 + ε)
5 :     x_min ← min(X)
6 :     x_max ← max(X)
7 :     while X ∩ [x_min; x_max] ≠ ∅ do
8 :         x ← (x_min + x_max)/2
9 :         w_x_bas ← Σ_{i|X_i<x} W_i
10 :        if w_x_bas < w_ε_min then
11 :            x_min ← x
12 :        else if w_x_bas > w_ε_max then
13 :            x_max ← x
14 :        else
15 :            return x
16 :        end if
17 :    end while
18 :    fail « pas de solution »
19 : end function
```

L'algorithme 4 fonctionne comme une recherche dichotomique sur X. Le domaine de recherche de la médiane pondérée est initialisé à $[x_{min}; x_{max}]$ et, tant qu'il existe des points de X dans ce domaine, l'algorithme teste si le milieu sépare les points en deux ensembles dont la différence de poids est tolérable. Sinon, la taille du domaine de recherche est divisée par deux. Les éléments de X et de W n'étant pas triés, il faut visiter l'ensemble de ces deux tableaux à chaque itération. La complexité de l'algorithme est alors $O(n \cdot \log n)$. Il est aussi possible de trouver la médiane pondérée en triant X et W, puis en parcourant la somme cumulée des poids jusqu'à atteindre un point pivot satisfaisant.

Plusieurs algorithmes sont dérivés de l'algorithme 4. Nous pouvons citer la bissection récursive de coordonnées [8] (communément appelé RCB pour *Recursive Coordinate Bisection*), qui applique récursivement la recherche de médiane selon les différents axes, de façon à obtenir des parties dont le ratio d'aspect est proche de 1. Cet algorithme a été l'objet de plusieurs variantes, dont RIB (*Recursive Inertia Bisection* [54]), qui effectue un changement de base préliminaire de façon à aligner les axes x, y, z avec les axes d'inertie du maillage. Une autre variante plus récente est *MultiJagged* [9], qui se permet de découper le maillage en trois parties ou plus à chaque itération, pour mieux équilibrer le poids des parties.

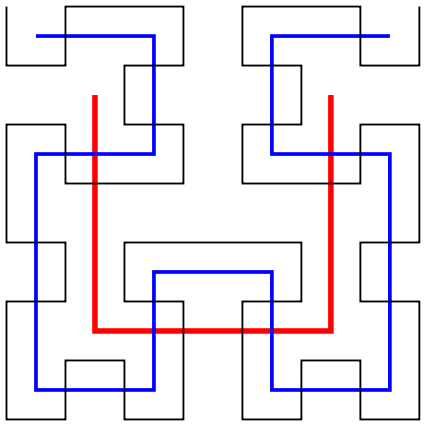
Une autre approche dérivée de la médiane pondérée consiste à réaliser des bissections selon des courbes couvrantes [55], [56]. Ces courbes associent des points sur un segment $[0; p^n] \subset \mathbb{N}$ vers $[0; p]^n$. L'exemple le plus simple de courbe couvrante est la courbe en Z [57] montrée en Figure 3.1a. Les indices de la courbe sont construits en alternant les chiffres binaires des coordonnées x et y. Certaines courbes couvrantes, comme celles de Hilbert [58], montrée en Figure 3.1b, garantissent que deux points voisins sur la courbe sont voisins dans l'espace nD, ce qui permet d'obtenir des parties connexes [59]. Les algorithmes de partitionnement dérivés de courbes couvrantes appliquent l'algorithme de médiane pondérée sur la courbe [60], [61]. Il est aussi possible d'appliquer une variante de cet algorithme qui partitionne la courbe d'un coup, sans besoin de récursion [9].

Les algorithmes à base de médiane pondérée sont hautement parallèles. Ils permettent ainsi de donner des résultats rapidement. Cependant, la qualité des résultats dépend fortement du fait que les points pivots possibles pour la médiane pondérée ne sont pas les points aux poids les plus élevés. Dans ces cas pathologiques, la condition à la ligne 7 de l'algorithme 4 peut être fausse, et l'algorithme ne trouve alors pas de solution.

D'autres algorithmes de partitionnement géométrique existent, notamment une adaptation des k-moyennes [62]. Cet algorithme consiste à faire grandir des boules à partir du centre des parties, de façon à obtenir

+	0	1	2	3	4	5	6	7
0	0	1	4	5	16	17	20	21
1	2	3	6	7	18	19	22	23
2	8	9	12	13	24	25	28	29
3	10	11	14	15	26	27	30	31
4	32	33	36	37	48	49	52	53
5	34	35	38	39	50	51	54	55
6	40	41	44	45	56	57	60	61
7	42	43	46	47	58	59	62	63

(a) La courbe en Z d'ordre 3



(b) Les courbes de Hilbert d'ordre 1 (en rouge), 2 (en bleu) et 3 (en noir)

Figure 3.1 – Courbes couvrantes, utilisées pour partitionner des points selon un seul axe, sans besoin de récursion.

des parties connexes et au ratio d'aspect proche de 1. Contrairement aux précédents, c'est un algorithme de raffinement : il lui faut une partition en entrée afin de calculer le centre des parties.

3.3 . Algorithmes pour le partitionnement topologique

Pour le partitionnement topologique, il existe des algorithmes de recherche locale. Ces algorithmes visitent l'espace des solutions (les partitions respectant une tolérance de déséquilibre ε, voir le problème de partitionnement topologique en Définition 2.8) de proche en proche, en faisant diminuer les coûts de communication à chaque déplacement. Ce sont des algorithmes de raffinement, et non de partitionnement initial. Ils nécessitent donc une partition, possiblement aléatoire, en entrée.

Deux exemples de tels algorithmes sont celui de Kernighan et Lin (KL) [36] et celui de Fiduccia et Mattheyses (FM) [37]. Pour une solution donnée (une partition qui satisfait la contrainte d'équilibre), les deux algorithmes effectuent des passes, dans lesquelles ils changent les sommets de partie. Une passe s'arrête quand tous les sommets ont été déplacés, alors la partition ayant la plus faible coupe (Définition 2.5) lors de la passe est retenue pour la passe suivante. Les algorithmes s'arrêtent quand une passe ne diminue pas la coupe.

KL et FM diffèrent par la façon dont ils visitent l'espace de solutions. KL échange les sommets deux à deux, de telle sorte à ne pas changer le

nombre de sommets des parties. Cela permet, lorsque les sommets ont tous le même poids, de garder le déséquilibre initial lors de l'optimisation de la coupe. Pour cela, KL parcours l'ensemble des pairs de sommets pouvant être échangés et la complexité d'une passe est de $O(n^2 \cdot \log n)$, où n est le nombre de sommets du graphe.

L'un des buts de FM est d'être moins complexe que KL. Pour cela, l'algorithme change les sommets de partie un à un et maintient deux index : le premier répertorie les sommets qui donnent le meilleur gain sur la coupe, et le deuxième stocke le gain de chaque sommet. Cela permet de trouver le meilleur mouvement en $O(g \cdot d)$, où g est le poids de l'arête la plus lourde et d le degré maximal du graphe. La mise à jour des index se fait en $O(d)$. Pour des arêtes à poids égaux, une passe de FM a une complexité de $O(n \cdot d)$.

Un exemple des deux algorithmes est illustré dans la Figure 3.2. Un graphe est partitionné en deux parties bleue et rouge. La coupe du graphe, représentée par des arêtes en pointillés, contient cinq arêtes. Chacun des deux algorithmes effectue un mouvement. FM déplace un sommet bleu vers la partie rouge : la coupe perd deux arêtes et en gagne une. KL échange les parties de deux sommets : la coupe perd trois arêtes et en gagne deux.

Les deux algorithmes paraissent pouvoir donner les mêmes solutions. Cependant, Barat [14] montre qu'ils visitent des parties distinctes de l'espace des solutions.

3.4 . Algorithme pour le partitionnement sous contrainte mémoire

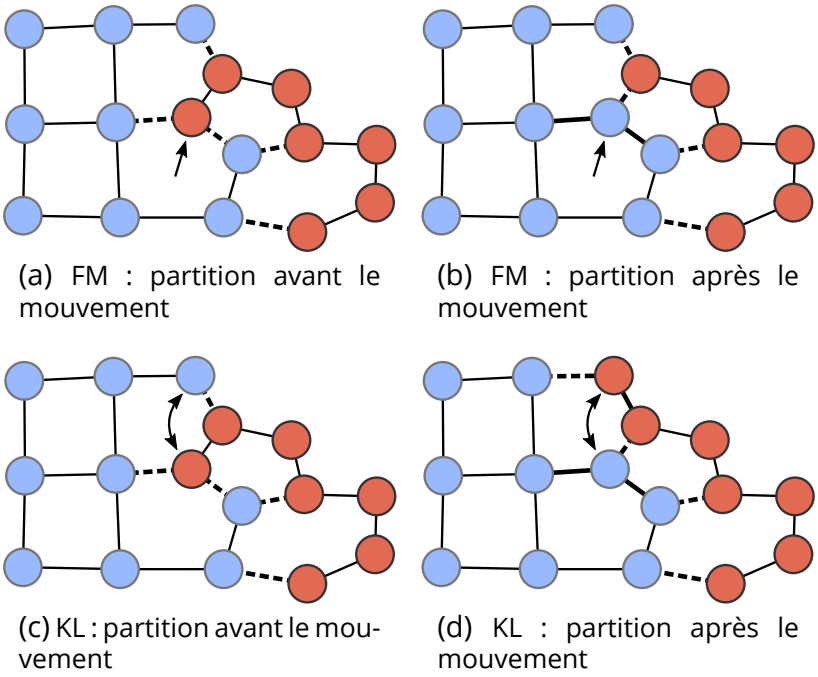
Morais et al. [12], [46], [63], [64] ont développé un algorithme glouton pour le partitionnement sous contrainte mémoire en utilisant le modèle à base de graphe biparti. Pour cela, ils définissent deux applications :

- $\omega : \mathcal{M} \longrightarrow \mathbb{R}^+$, qui représente la quantité de données associée à chaque maille ;
- $\mathcal{D} : \{0, \ldots, k-1\} \longrightarrow \mathcal{P}(\mathcal{M})$, qui donne l'ensemble des mailles dont les données sont requises par chaque unité de calcul :

$$\mathcal{D}(i) = \{d \in \mathcal{M} \mid \{c, d\} \in E; c \in P_i\}.$$

L'idée de l'algorithme est d'associer chaque traitement $c \in C$ à l'une des unités de calcul. L'ordre de parcours de C est défini par un tri en deux étapes :

1. trier les traitements par ordre décroissant des données requises ;



(a) FM : partition avant le mouvement

(b) FM : partition après le mouvement

(c) KL : partition avant le mouvement

(d) KL : partition après le mouvement

Figure 3.2 – Optimisation d'une bi-partition par Kernighan-Lin et Fiduccia-Mattheyses.

2. trier chaque ensemble de traitements ayant la même quantité de données requises par ordre décroissant selon la quantité de données partagées avec des traitements déjà triés.

L'unité de calcul associée à c est alors sélectionnée de la façon suivante :

1. l'algorithme sélectionne les unités de calcul pouvant garder en mémoire les données nécessaires au traitement de c :

$$M_i \geq \sum_{d \in N(c) \cup \mathcal{D}(i)} \omega(d);$$

2. parmi ces unités de calcul, l'algorithme choisit celle dont la taille totale des données requises supplémentaires pour le traitement de c est minimale :

$$\text{minimiser} \sum_{d \in N(c) \setminus \mathcal{D}(i)} \omega(d);$$

3. enfin, parmi les unités de calcul restantes, l'algorithme choisit aléatoirement une des unités de calcul qui ont le plus de mémoire libre :

$$\text{maximiser } M_i - \sum_{d \in \mathcal{D}(i)} \omega(d);$$

L'algorithme complet est décrit dans l'Algorithme 5.

Algorithme 5 Algorithme glouton pour le partitionnement de maillage sous contrainte mémoire

function GloutonMem($\mathcal{B}$, M)
  $\Pi \leftarrow [0, \overset{|S|}{\ldots}, 0]$
  $SortedC \leftarrow SortCVertices(\mathcal{B})$
  for all $c \in SortedC$ **do**
    $M_0 \leftarrow \left\{ 0 \leq i < |M| \mid M_i \geq \sum_{d \in N(c) \cup \mathcal{D}(i)} \omega(d) \right\}$
    $M_1 \leftarrow \left\{ \arg\min_{i \in M_0} \sum_{d \in N(c) \setminus \mathcal{D}(i)} \omega(d) \right\}$
    $M_2 \leftarrow \left\{ \arg\max_{i \in M_1} M_i - \sum_{d \in \mathcal{D}(i)} \omega(d) \right\}$
    $\Pi[c] \leftarrow random(M_2)$
  end for
  return Π
end function

Conclusion

Nous avons vu dans le Chapitre 1 que les simulations physiques à grande échelle sont exécutées sur des machines à mémoire distribuée et disposant de nombreuses unités de calcul. Cette architecture matérielle impose aux simulations de répartir les données et les calculs sur les unités de calcul. Cette répartition influence le temps d'exécution de la simulation de deux manières : le déséquilibre de charge entre les unités de calcul, et le taux de communication requis.

Dans le Chapitre 2, nous avons vu que répartir les données et calculs de manière optimale est un problème qui, pour les simulations à base de maillage, se traduit par le problème de k-partitionnement de maillage (Définition 2.1). De ce problème découlent plusieurs approches, chacune ayant ses avantages et inconvénients.

Dans le chapitre précédent, nous avons décrit des algorithmes existants pour plusieurs de ces approches au partitionnement de maillage. Certains algorithmes créent une partition alors que d'autres optimisent une partition donnée.

Dans le reste de ce manuscrit, nous allons utiliser les algorithmes décrits pendant cette section pour permettre aux utilisateurs d'outils de partitionnement de choisir précisément comment optimiser leurs partitions.

Deuxième partie

Chaînes d’algorithmes de partitionnement de maillage

4 - Mise en place d'une méthode de partitionnement de maillage multi-objectif par chaînes d'algorithmes

Nous cherchons à proposer un nouvel outil de partitionnement de maillages, problème défini par la Définition 2.1 (page 33), de façon à minimiser le temps d'exécution d'une simulation. Comme dit précédemment, ce problème de partitionnement est souvent impossible à résoudre, et l'utilisateur doit donc reformuler le problème pour le rendre plus simple.

De nombreux codes de simulation utilisent des outils de partitionnement de graphe. De tels outils calculent le graphe dual du maillage pour une relation de voisinage donnée, et minimisent la coupe du graphe sous une contrainte d'équilibre donnée par l'utilisateur. En pratique, bien que les partitions produites par ces outils ont une coupe très faible, elles ne respectent pas toujours la contrainte d'équilibre [14]. Dans certains cas, le non respect de cette contrainte peut mener à un temps d'attente, dû aux synchronisations, qui est supérieur aux gains en temps associés à la réduction de la coupe.

Les outils de partitionnement existants ne prennent pas en compte certaines attentes d'un utilisateur de code de calcul. Premièrement, ils ne considèrent pas directement la notion de maille fantôme. Bien que la minimisation de la coupe agit indirectement sur les charges mémoires associées aux mailles fantômes, la coupe même est une métrique globale et peut conduire à des charges mémoires très hétérogènes sur les unités de calcul. Ainsi, il est possible qu'une simulation ne puisse pas s'exécuter correctement, pour cause de dépassement mémoire. Deuxièmement, ces outils de partitionnement ne prennent pas en compte la capacité mémoire des unités de calcul. Ainsi, la consommation mémoire d'une unité de calcul peut excéder sa capacité mémoire, que le code de calcul utilise des mailles fantômes ou non.

Ensuite, le partitionnement de graphe n'est pas la seule façon de partitionner un maillage : certains algorithmes de partitionnement géométrique sont connus pour mieux passer à l'échelle [8], [9], [62] et ainsi donner des partitions plus rapidement que les outils de partitionnement de graphe lors de l'exécution sur un grand nombre d'unité de calcul. Ce gain en temps de restitution peut avoir un impact non négligeable sur le temps d'exécution du code de simulation.

Enfin, un utilisateur peut aussi vouloir des parties ayant une certaine forme. Dans ce cas, un algorithme de partitionnement géométrique peut

être utilisé.

Un utilisateur d'outil de partitionnement peut donc attendre de celui-ci qu'il permette d'obtenir une partition :

- où les parties sont à charge égale,
- le plus rapidement possible,
- où les parties ont une certaine forme,
- ayant un coût de communications minimal,
- qui permet de distribuer les données de la simulation sans dépasser la capacité mémoire des unités de calcul.

Respecter ces cinq propriétés est difficile. Néanmoins, comme nous l'avons vu dans le Chapitre 3, il existe des algorithmes permettant de respecter chacune de ces cinq propriétés. En fonction des attentes de l'utilisateur, il devrait être possible de trouver un compromis sur plus d'une de ces propriétés.

Pour trouver des compromis au plus grand nombre de cas d'utilisation du partitionnement de maillages, nous nous intéressons au chaînage d'algorithmes de partitionnement.

Par exemple, prenons le maillage 3D présenté en Figure 4.1, où les poids varient entre 0 et 1 000. Le poids associé à une maille est déterminé par l'abscisse de son barycentre. Maintenant, partitionnons ce maillage sur huit unités de calcul. Pour cela, il est possible d'utiliser un algorithme géométrique tel qu'une bissection récursive en Figure 4.2. La partition obtenue a un déséquilibre de 0.030, une coupe de 8.70×10^6 et une $\lambda - 1$ *cut* de 6.47×10^6.

Nous pouvons optimiser cette partition afin de diminuer le déséquilibre de charge et la coupe. Par exemple, nous pouvons chercher à diminuer le déséquilibre en appliquant un algorithme d'optimisation locale tel que VNBest [14]. VNBest est un algorithme de partitionnement de nombres qui optimise une partition existante. Il déplace les mailles de partie en partie afin de diminuer le poids le plus possible à chaque déplacement. La figure Figure 4.3 illustre la partition obtenue après optimisation de la partition présentée en Figure 4.2. La nouvelle partition a un déséquilibre de 1.9×10^{-6} (précédemment 0.030), une coupe de 10.8×10^6 (précédemment 8.70×10^6) et une $\lambda - 1$ *cut* de 8.07×10^6 (précédemment 8.70×10^6). Remarquons que les parties ne sont plus connexes. En effet, VNBest change les mailles de partie en partie sans prendre en compte de relation de voisinage, tant que le changement de partie diminue le déséquilibre de charge. Remarquons aussi que les poids déplacés se concentrent autour de la zone où les poids sont les plus élevés. Nous supposons que cela est dû au fait que VNBest est un algorithme glouton, qui change les mailles de parties de telle sorte à diminuer le plus possible le déséquilibre. Le futur Chapitre 5 détaillera

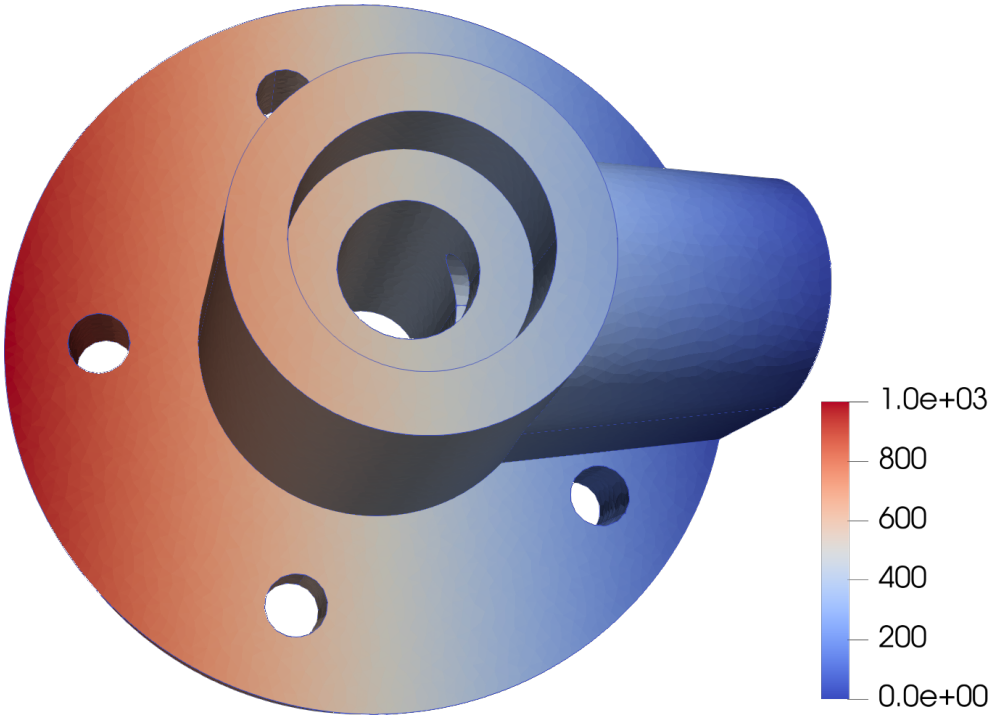


Figure 4.1 – Un maillage 3D composé de 232 698 mailles. La couleur des mailles représente leur poids – la charge de calcul qui leur est associée. Le poids de chaque maille dépend exclusivement de l'abscisse de son barycentre, et évolue linéairement de 0 à 1 000, de droite à gauche.

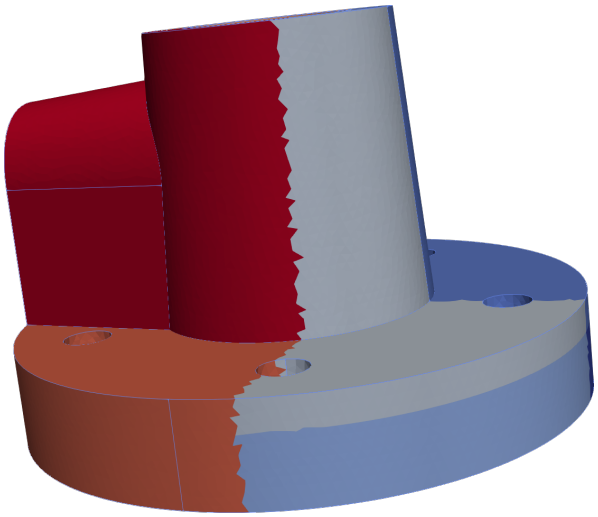


Figure 4.2 – Un maillage 3D partitionné en 8 parties suivant une bissection récursive. Chaque couleur représente une partie. Sur la figure, seules cinq des parties sont visibles.

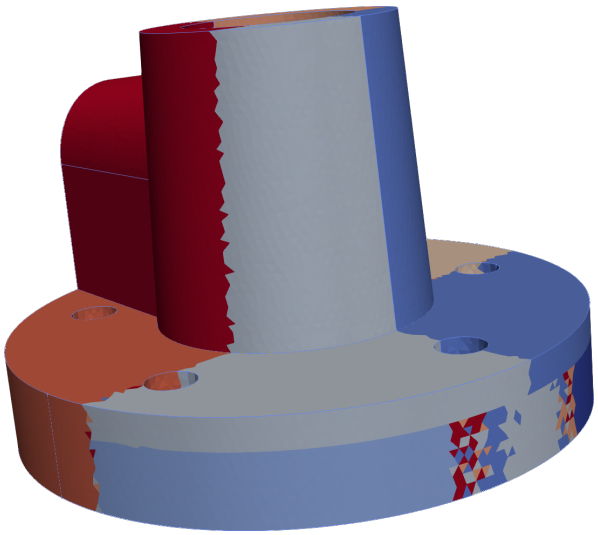


Figure 4.3 – Un maillage 3D partitionné en 8 parties suivant d'abord une bissection récursive, puis une passe d'optimisation de l'équilibre par VNBest. Durant la seconde passe, les mailles sont changées de parties pour diminuer le déséquilibre le plus possible à chaque mouvement, sans prendre en compte la topologie du maillage. Chaque couleur représente une partie.

les raisons précises du choix des mailles par VNBest.

Enfin, nous pouvons optimiser la partition afin de diminuer la coupe, sous une contrainte d'équilibre. Ici, nous appliquons Arcswap avec une tolérance sur le déséquilibre de valeur 0.01. Nous obtenons ainsi le résultat visible sur la Figure 4.4. Arcswap est un algorithme d'optimisation locale pour le partitionnement topologique développé pendant cette thèse. Similaire à celui de Fiduccia et Mattheyses (FM) [37], il déplace les mailles de partie en partie pour faire diminuer la coupe, tant que ces déplacements n'augmentent pas le déséquilibre de la partition au-dessus de la tolérance donnée. Contrairement à FM, Arcswap ne cherche pas le déplacement qui fait diminuer le plus possible la coupe. Une telle partition a un déséquilibre de 0.008, une coupe de 7.7×10^6 et une $\lambda - 1$ *cut* de 7.1×10^6. Cette partition a non seulement un meilleur déséquilibre que celle obtenue par RCB seul, elle a aussi une meilleure coupe. Elle a cependant un moins bon déséquilibre que la partition obtenue par RCB suivi de VNBest.

Chaîner des algorithmes de partitionnement permet ainsi à l'utilisateur de l'outil de partitionnement de donner priorité aux métriques qui l'intéresse. L'idée de chaîner des algorithmes était déjà présente dans Scotch [3], qui permet de spécifier en grand détail la façon dont se déroule le partitionnement ainsi que les algorithmes utilisés. Cependant, le problème de partitionnement résolu est le même à chaque maillon de

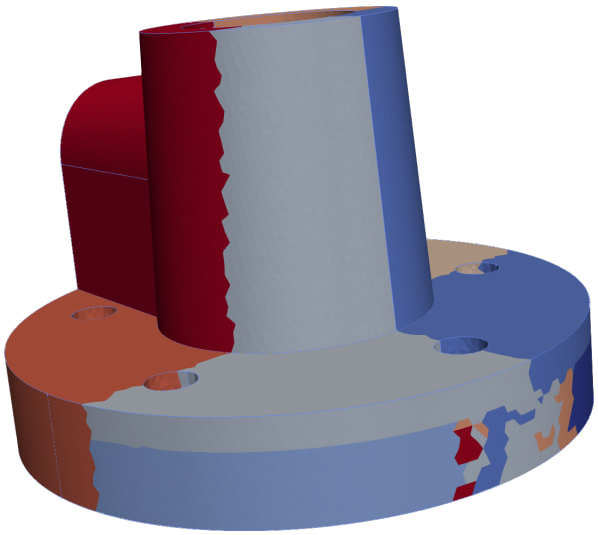


Figure 4.4 – Un maillage 3D partitionné en 8 parties suivant d'abord une bissection récursive, puis une passe d'optimisation de l'équilibre par VN-Best, puis une passe d'optimisation des communications par Arcswap. Arcswap change les mailles de partie de façon à faire diminuer la coupe. Chaque couleur représente une partie.

la chaîne : Scotch résout le problème de partitionnement topologique.

Pour le partitionnement de maillage, il est intéressant de changer le modèle utilisé à chaque maillon de la chaîne. Par exemple, commencer par générer une partition rapidement avec un algorithme géométrique, puis appliquer une optimisation sur les nombres pour améliorer l'équilibre de charge, avant de finir par une optimisation topologique pour minimiser la coupe, comme vu précédemment. Un tel procédé devrait permettre d'obtenir une partition dont la forme est bonne grâce à la décomposition géométrique, mais aussi bien équilibrée et avec un volume de communications bas grâce aux étapes de raffinement.

Pour illustrer ce phénomène avec plus d'algorithmes, prenons le maillage 2D présenté sur la Figure 4.5, et composé de 19 312 triangles, où le poids des mailles est généré de la même façon que l'exemple précédent – croissant selon l'axe des abscisses. Maintenant, appliquons différents algorithmes de partitionnement de nombres, géométrique et topologique, de façon à obtenir des bi-partitions, c'est-à-dire des partitions à deux parties. Les algorithmes sont chacun appliqués plusieurs fois (seize), car certains ne sont pas déterministes.

Les algorithmes utilisés sont l'heuristique de partitionnement de nombres de Karmarkar et Karp [35], vu à la page 43, les algorithmes géométriques RCB [8] et les courbes couvrantes de Hilbert [58] (d'ordre 12), vus à la page 47, ainsi que l'algorithme de partitionnement de nombre

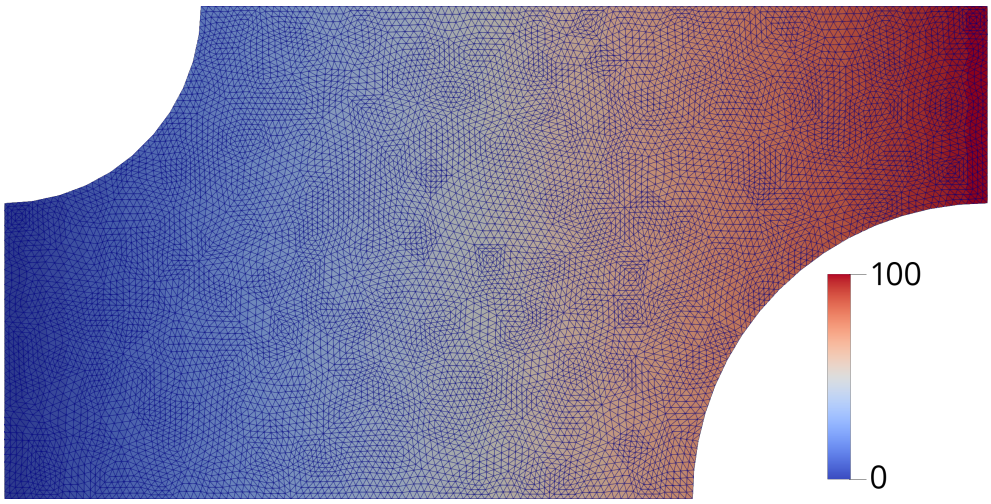


Figure 4.5 – Maillage 2D triangulaire composé de 19 312 éléments. La couleur représente le poids – la charge de calcul – associée à chaque maille.

VNBest [14], et l'algorithme de partitionnement topologique Arcswap (avec une tolérance sur le déséquilibre de 0.01).

Plaçons ensuite la coupe et le déséquilibre de chaque partition sur un graphe en Figure 4.6. Sur ce graphe, chaque point représente une des seize partitions obtenues pour chaque chaîne d'algorithmes. Les échelles utilisées sont logarithmiques. Nous remarquons d'abord des grands écarts de coupe et de déséquilibre entre les partitions données par les différents algorithmes, et qu'il n'y a pas de solution meilleure que les autres. Typiquement, Karmarkar-Karp (représenté sous forme de triangle bleu vers le bas) donne des partitions avec le meilleur déséquilibre mais au détriment de la coupe. Au contraire, le chaînage Arcswap sur RCB (représenté sous forme de carré vide) – c'est-à-dire l'application de RCB puis d'Arcswap – permet d'obtenir les partitions ayant la meilleure coupe, mais au prix d'un déséquilibre élevé. Un bon compromis pour ce maillage serait le chaînage VNBest sur RCB (carré plein), mais donne une coupe plus élevée que les autres chaînes avec RCB. Notons que certains algorithmes, comme Arcswap, ne donnent pas toujours les mêmes résultats. Cela est principalement dû au non-déterminisme induit par le parallélisme de l'algorithme.

Il faut remarquer ensuite que Arcswap et VNBest, deux algorithmes optimisant les partitions en minimisant respectivement la coupe et le déséquilibre, déplacent légèrement les points sur le graphe. Par exemple, les partitions, une fois optimisées par Arcswap, apparaissent plus à gauche sur le graphe. De façon similaire, les partitions obtenues par VNBest apparaissent plus en bas du graphe. Ces deux algorithmes permettent ainsi d'ajuster la coupe et le déséquilibre d'une partition préalablement

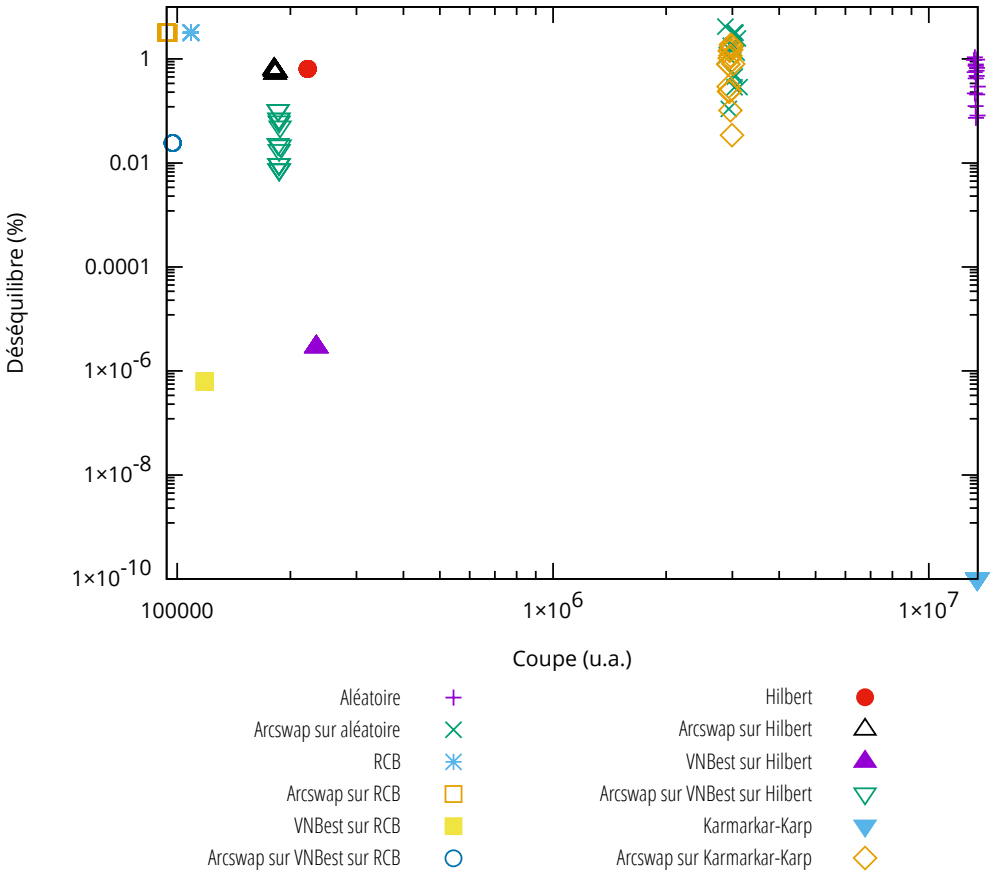


Figure 4.6 – Coupe et déséquilibre de partitions obtenues par différents algorithmes. Le déséquilibre est donné en pourcentages. La coupe est donnée en unités arbitraires (même unité que celle des poids des mailles). Le déséquilibre de la partition obtenue par Karmarkar-Karp vaut 0, et a été élevé à 10^{-10}% pour cette visualisation sur échelle logarithmique.

obtenue. C'est l'idée de notre nouvelle méthode de partitionnement de maillage : créer une partition de zéro, ou partir d'une solution existante, et utiliser différents problèmes et modèles de partitionnement pour ajuster la partition en fonction des intérêts de l'utilisateur.

Conclusion

Dans ce chapitre, nous avons mis en place une nouvelle méthode de partitionnement de maillage, impliquant de chaîner plusieurs algorithmes choisis de façon à optimiser le déséquilibre ou la coupe d'une partition de maillage. Or, comme évoqué dans la Section 2.5 (page 40), il existe d'autres intérêts au partitionnement de maillage, que nous allons couvrir dans les chapitres suivants :

- Certaines simulations sont exécutées sur des machines où les unités de calcul ne sont pas homogènes : certaines unités peuvent calculer plus rapidement que d'autres. Lors de l'équilibrage de charge, il faut alors allouer plus de charge sur certaines parties. Dans le Chapitre 5, nous allons étendre l'algorithme d'optimisation pour le partitionnement de nombres, de façon à équilibrer la charge en fonction de poids cibles attribués à chaque partie ;
- Certaines simulations fonctionnent exclusivement à l'aide de maillages dits « cartésiens » – c.à.d. des grilles, 2D ou 3D – à des fins de performance. Ces simulations supposent généralement que l'outil de partitionnement donne des parties qui sont aussi des grilles. Dans le Chapitre 6, nous allons spécialiser l'algorithme de bissection récursive RCB pour les maillages cartésiens, afin de traiter de plus grands maillages plus rapidement ;

5 - Extension de VNBest pour le partitionnement à cibles de charge

Lors du partitionnement d'un maillage, il peut s'avérer intéressant d'allouer plus de charge de calcul à certaines parties qu'à d'autres. Par exemple, lorsque les unités de calcul ne calculent pas à la même vitesse. Ce chapitre se focalise sur le développement d'une solution à ce problème, dans le cadre du chaînage d'algorithmes vu dans le chapitre précédent. Pour cela, nous étendons l'algorithme d'optimisation pour le partitionnement de nombres, VNBest [14], pour prendre en compte un poids cible pour chaque partie.

D'abord, la Section 5.1 définit formellement le problème de partitionnement de nombre à cibles de charge et introduit VNBest. Ensuite, dans la Section 5.2 (page 65), nous étendons l'algorithme pour prendre en compte un poids cible par partie, dans le cas du partitionnement en deux parties. Enfin, dans la Section 5.3 (page 70), nous étendons l'algorithme pour un nombre arbitraire de parties.

5.1 . VNBest et le problème de partitionnement de nombres à cibles de charge

Une définition du partitionnement de nombres a déjà été donnée dans le Chapitre 2. Cependant, et comme annoncé dans la Section 2.5, il existe une définition plus générale du problème, où les parties ont chacune un poids cible. Par exemple, on peut vouloir chercher à équilibrer la charge de mailles dans deux parties de telle façon qu'un tiers de la charge totale soit allouée à la première partie et deux tiers à l'autre. Nous appelons ce nouveau problème le « problème de partitionnement de nombres à cibles ». Avant de le définir formellement, définissons d'abord, dans la Définition 5.1, le déséquilibre d'une partition selon des cibles.

Définition 5.1 (Déséquilibre de charge d'une partie et d'une partition selon des cibles)**.** *Soient un maillage $\mathcal{M}$, une partition $\Pi = (P_i)_{0 \leq i < k}$ de $\mathcal{M}$, une application $w : \mathcal{M} \longrightarrow \mathbb{R}^+$ associant une charge de calcul à chaque maille de $\mathcal{M}$, et $T = (t_i)_{0 \leq i < k}$ une famille de réels positifs dont la somme égale la somme des charges de toutes les mailles, c'est-à-dire :*

$$\sum_{i=0}^{k-1} t_i = \sum_{m \in \mathcal{M}} w(m),$$

Alors le déséquilibre d'une partie $P_i \in \Pi$ est défini comme :

$$Im(P_i, t_i) = \sum_{m \in P_i} w(m) - t_i.$$

Le déséquilibre de la partition Π *est défini comme :*

$$Imb(\Pi, T) = \max_{i=0}^{k-1} Imb(P_i, t_i).$$

Nous pouvons ainsi définir le problème de partitionnement de nombres à cibles dans la Définition 5.2.

Définition 5.2 (Problème de partitionnement de nombres à cibles)**.** *Soient un maillage* $\mathcal{M}$*, une application* $w : \mathcal{M} \longrightarrow \mathbb{R}^+$ *associant une charge de calcul à chaque maille de* $\mathcal{M}$*, et* $T = (t_i)_{0 \leq i < k}$ *une famille de réels positifs dont la somme égale la somme des charges de toutes les mailles. Le problème de partitionnement de nombres à cibles consiste à trouver une partition* Π *de* $\mathcal{M}$ *qui minimise le déséquilibre* $Imb(\Pi, T)$*.*

Ce problème est une généralisation du partitionnement de nombres pour des cibles t_i qui ne sont pas forcément égales. En effet, minimiser le déséquilibre $Imb(\Pi, T)$ revient, dans le cas où les cibles sont égales, à minimiser la charge de la partie la plus chargée.

Pour résoudre ce problème dans le cadre du chaînage d'algorithmes introduit dans le chapitre précédent, deux algorithmes nous intéressent : VNFirst [14] et VNBest [14]. Ces deux algorithmes s'inspirent de la méthode de « hill-climbing » présentée par Korf [33], et consistent à déplacer les mailles, une à une, d'une partie à l'autre, afin de faire diminuer le déséquilibre. D'un côté, VNBest choisit les mailles à déplacer de façon à diminuer le déséquilibre le plus possible à chaque déplacement. De l'autre, VNFirst déplace n'importe quelle maille tant que cela fait diminuer le déséquilibre. Les deux algorithmes s'arrêtent quand plus aucun déplacement ne diminue le déséquilibre. En guise d'illustration, le principe de VNBest est détaillé dans l'Algorithme 6. L'hypothétique fonction $Move$ donne, pour une partition Π, une maille m et l'indice p d'une partie, la partition qui résulte du déplacement de m dans la $p$ième partie de Π.

Algorithme 6 Squelette de VNBest

1 : **procedure** VNBest($\mathcal{M}$, Π, k)
2 : $M \leftarrow \{(m,p) \in \mathcal{M} \times [0;k-1] \mid Imb(Move(\Pi,m,p),T) < Imb(\Pi,T)\}$
3 : **while** $M \neq \emptyset$ **do**
4 : $(m_{\text{best}},p) \leftarrow \arg\min_{(m,p)\in M} Imb(Move(\Pi,m,p),T)$
5 : $\Pi \leftarrow Move(\Pi, m_{\text{best}}, p)$
6 : $M \leftarrow \{(m,p) \in \mathcal{M} \times [0;k-1] \mid Imb(Move(\Pi,m,p),T) < Imb(\Pi,T)\}$
7 : **end while**
8 : **end procedure**

Dans le cas de VNFirst, seule la ligne 4 de l'algorithme 6 change pour ne pas prendre le minimum, mais un élément arbitraire de M. De cette façon, M peut être construit en parallèle et plusieurs déplacements peuvent être trouvés à la fois.

VNBest et VNFirst ont différents avantages dans le cadre du chaînage d'algorithmes. VNBest, qui est une heuristique gloutonne, laisse supposer que le nombre de déplacements effectués est presque minimal, ce qui est intéressant quand chacun de ces déplacements ne prend en compte ni la topologie ni la géométrie du maillage, et donc peut augmenter les coûts de communication. De l'autre côté, VNFirst n'est pas glouton, et est plus facilement parallélisable, ce qui permet de traiter de plus grands maillages plus rapidement.

Dans ce chapitre, nous nous intéressons exclusivement à VNBest. Pour VNBest, le recherche de m_{best} est la difficulté principale de la conception de l'algorithme. Une version naïve de cette recherche calculerait le déséquilibre de toutes les partitions qu'il est possible d'obtenir avec le déplacement d'une des mailles, puis prendrait le minimum. Chaque itération aurait alors une complexité de $O(|\mathcal{M}| \cdot k)$. Barat [14] montre qu'il est possible, dans le cas du bi-partitionnement, d'effectuer la recherche du meilleur mouvement en $O(\log |\mathcal{M}|)$. Nous allons montrer qu'il est possible d'arriver au même résultat lorsque les parties ont des poids cibles, puis nous allons étendre la méthode de recherche au k-partitionnement.

5.2 . Recherche des meilleurs déplacements dans le cas du bi-partitionnement

VNBest, comme dit précédemment, déplace les mailles une à une de façon à baisser le déséquilibre le plus possible à chaque fois et cela tant que c'est possible. Dans la version originale, la recherche de la maille à déplacer repose sur la définition d'un gain en équilibre de charge pour

chaque maille. Nous allons faire de même, mais en utilisant la définition de déséquilibre selon des poids cibles par partie. Définissons d'abord le déplacement d'une maille dans la Définition 5.3.

Définition 5.3 (Déplacement d'une maille)**.** *Pour un maillage $\mathcal{M}$, nous notons $\mathcal{P}_2(\mathcal{M})$ l'ensemble des bi-partitions de $\mathcal{M}$. Définissons alors le déplacement des mailles d'un maillage par l'application $Move$:*

$$\begin{array}{rccl} Move: & \mathcal{P}_2(\mathcal{M}) \times \mathcal{M} & \longrightarrow & \mathcal{P}_2(M) \\ & ((P_1, P_2), m) & \longmapsto & \begin{cases} (P_1 \setminus \{m\}, P_2 \cup \{m\}), \textit{ si } m \in P_1; \\ (P_1 \cup \{m\}, P_2 \setminus \{m\}), \textit{ sinon.} \end{cases} \end{array}$$

L'application $Move$ donne, pour une bi-partition d'un maillage et une maille m de ce même maillage, une nouvelle partition où la maille m est dans l'autre partie. Nous appelons cela un « déplacement ». De cette définition du déplacement, nous pouvons définir le gain en équilibre d'un déplacement dans la Définition 5.4, comme la différence de déséquilibre entre la partition initiale et la partition finale.

Définition 5.4 (Gain d'un déplacement)**.** *Soient un maillage $\mathcal{M}$, une bi-partition $\Pi = (P_0, P_1)$ de $\mathcal{M}$, et $T = (t_0, t_1)$ un couple de réels positifs dont la somme est égale à la somme des charges de toutes les mailles. Nous définissons alors le gain des déplacements comme l'application $Gain$:*

$$\begin{array}{rccl} Gain: & \mathcal{P}_2(\mathcal{M}) \times \mathcal{M} & \longrightarrow & \mathbb{R} \\ & (\Pi, m) & \longmapsto & Imb(\Pi, T) - Imb(Move(\Pi, m), T). \end{array}$$

Le gain du déplacement d'une maille est positif quand déplacer cette maille dans l'autre partie diminue le déséquilibre. Il est négatif sinon. Nous cherchons à obtenir le maximum de la fonction $Gain$. Pour cela, au lieu de calculer naïvement le déséquilibre des deux parties, il est possible d'exploiter la formule du déséquilibre pour obtenir une formule plus explicite du gain, en fonction de la charge d'une maille m et du déséquilibre de la partition Π.

Avant de pouvoir obtenir une formule explicite du gain, nous montrons une propriété sur le déséquilibre de charge selon des cibles :

Proposition 1 (Symétrie du déséquilibre)**.** *Soient un maillage $\mathcal{M}$, une bi-partition $\Pi = (P_0, P_1)$ de $\mathcal{M}$, et $T = (t_0, t_1)$ un couple de réels positifs dont la somme égale à la somme des charges de toutes les mailles. Alors, le déséquilibre des parties de Π sont des opposés :*

$$Imb(P_0, t_0) + Imb(P_1, t_1) = 0.$$

Nous pouvons ainsi écrire :

$$Imb(\Pi, T) = |Imb(P_0, t_0)| = |Imb(P_1, t_1)|.$$

Démonstration.

$$\begin{aligned} Imb(P_0, t_0) + Imb(P_1, t_1) &= \sum_{m \in P_0} w(m) - t_0 + \sum_{m \in P_1} w(m) - t_1, \text{ par la déf. 5.1} \\ &= \sum_{m \in \mathcal{M}} w(m) - (t_0 + t_1) \\ &= 0. \end{aligned}$$

Par la Définition 5.1,

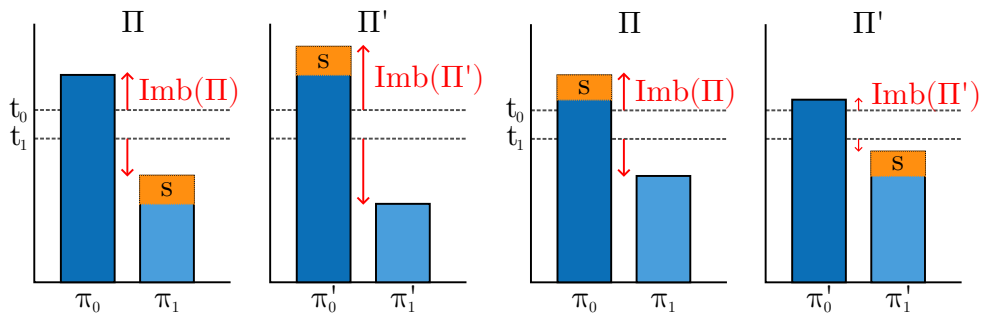
$$Imb(\Pi, T) = \max(Imb(P_0, t_0), Imb(P_1, t_1)) = |Imb(P_0, t_0)| = |Imb(P_1, t_1)|.$$

□

Une façon de voir la proposition 1 est la suivante : la charge qu'il manque à l'une des parties est la charge qu'il y a en trop dans l'autre partie. Cette propriété est vraie pour n'importe quelles cibles de charge, tant que ces cibles ont une somme égale à toutes les charges.

Nous explicitons maintenant dans la proposition 2 la valeur du déséquilibre après un déplacement, en utilisant la propriété précédente. Afin de faciliter la compréhension de cette proposition et de la preuve associée, nous introduisons la Figure 5.1 qui détaille visuellement les différents cas à séparer lors du calcul du gain. La Figure 5.1a correspond au premier cas de la formule du gain, c'est-à-dire qu'un déplacement de la partie souschargée vers la partie surchargée fait toujours augmenter le déséquilibre. Le deuxième cas est représenté sur la Figure 5.1b : le gain apporté par le déplacement d'une charge plus petite que le déséquilibre est égal à cette charge. On gagne aussi à déplacer des charges plus grandes que le déséquilibre (Figure 5.1c), mais le gain alors diminue avec la valeur de la charge, jusqu'à atteindre des valeurs négatives pour les charges supérieures au double du déséquilibre (Figure 5.1d). Les différents cas sont aussi récapitulés dans le tableau de variation 5.1.

Proposition 2 (Formule explicite du gain)**.** *Soient un maillage $\mathcal{M}$, une bipartition $\Pi = (P_0, P_1)$ de $\mathcal{M}$, une application $w : \mathcal{M} \longrightarrow \mathbb{R}^+$ associant une charge de calcul à chaque maille de $\mathcal{M}$, $T = (t_0, t_1)$ un couple de réels positifs dont la somme égale à la somme des charges de toutes les mailles, et $m \in P_0$. Alors,*



(a) La charge s est dans la partie souschargée. Le déséquilibre augmente.

(b) La charge s est plus petite que le déséquilibre. Le déséquilibre diminue.

(c) La charge s est plus grande que le déséquilibre. Le déséquilibre diminue.

(d) La charge s est plus grande que le double du déséquilibre. Le déséquilibre augmente.

Figure 5.1 – Différents cas de déplacement de poids et leur impact sur le déséquilibre.

$$Gain(\Pi, m) = \begin{cases} -w(m) & \textit{si } Imb(P_0, t_0) \leq 0, \\ w(m) & \textit{si } Imb(P_0, t_0) \geq 0 \textit{ et } w(m) \leq Imb(\Pi, T), \\ 2 \cdot Imb(\Pi, T) - w(m) & \textit{si } Imb(P_0, t_0) \geq 0 \textit{ et } w(m) \geq Imb(\Pi, T). \end{cases}$$

$w(m)$	0	$Imb(\Pi,T)$	$2\cdot Imb(\Pi,T)$	$+\infty$
$Gain(\Pi,m)$	$w(m)$		$2\cdot Imb(\Pi,T)-w(m)$	
$Gain(\Pi,m)$	0 ↗	$Imb(\Pi,T)$	↘ 0 ↘	

Table 5.1 – Tableau de variation du gain en fonction de la charge de la maille déplacée, pour une partition donnée et pour un déplacement depuis la partie la plus surchargée.

Démonstration. Notons que l'expression est bien définie malgré les chevauchements dûs aux inégalités larges :

- Si $Imb(P_0,t_0)=0$ alors, par la proposition 1, $Imb(\Pi,T)=0$. Ainsi, la charge $w(m)$ de m étant positive, le premier et le troisième cas de la formule sont identiques. D'autre part, le deuxième cas n'est possible que si $w(m)$ est nul.
- Si $Imb(P_0,t_0)\geq 0$ et $w(m)=Imb(\Pi,T)$, le deuxième et le troisième cas donnent la même valeur $Imb(\Pi,T)$.

Quand à la preuve de la formule même du gain, nous utilisons d'abord la Définition 5.4 du gain :

$$\begin{aligned} Gain(\Pi,m) &= Imb(\Pi,T)-Imb(Move(\Pi,m),T)\\ &= |Imb(P_0,t_0)|-|Imb(P_0,t_0)-w(m)|, \text{ par la proposition 1.}\end{aligned}$$

Si $Imb(P_0,t_0)\leq 0$, alors comme $w(m)$ est positif, $Imb(P_0,t_0)-w(m)\leq 0$, et $Gain(\Pi,m)=-Imb(P_0,t_0)+Imb(P_0,t_0)-w(m)=-w(m)$.

Sinon, par la proposition 1, nous avons $Imb(P_0,t_0)=Imb(\Pi,T)$. Nous distinguons alors deux autres cas :

- Soit $w(m)\leq Imb(\Pi,T)$, et dans ce cas

$$Gain(\Pi,m)=Imb(P_0,t_0)-(Imb(P_0,t_0)-w(m))=w(m).$$

- Soit $w(m)\geq Imb(\Pi,T)$, et alors

$$Gain(\Pi,m)=Imb(P_0,t_0)+(Imb(P_0,t_0)-w(m))=2\cdot Imb(P_0,t_0)-w(m).$$

□

La formule explicite du gain nous permet de remarquer deux choses. Premièrement, le gain est alors seulement positif pour les mailles ayant une charge plus petite que le double du déséquilibre de la partition ;

deuxièmement, le meilleur déplacement est celui de la charge la plus proche du déséquilibre, dans la partie surchargée. Maintenant que nous savons quelle maille déplacer, décrivons la version de VNBest pour résoudre le problème du partitionnement de nombres à cibles de charge, dans le cas du bi-partitionnement.

Algorithme 7 Version détaillée de VNBest, dans le cas du partitionnement en deux parties, et ou les parties ont des cibles de charge

1 : **procedure** VNBest($\mathcal{M}$, $\Pi = (P_0, P_1)$, $T = (t_0, t_1)$, w)
2 : **loop**
3 : $P_{\text{souschargée}}, P_{\text{surchargée}} \leftarrow \Pi$
4 : $m \leftarrow SearchClosest(P_{\text{surchargée}}, Imb(\Pi, T), w)$
5 : **if** $2 \cdot Imb(\Pi, T) \leq w(m)$ **then**
6 : **break** ▷ Aucun déplacement ne baisse le déséquilibre
7 : **end if**
8 : $\Pi \leftarrow Move(\Pi, m)$
9 : **end loop**
10 : **end procedure**

L'Algorithme 7 déplace les mailles de la partie la plus chargée vers la partie la moins chargée, tant que des mailles de la partie surchargée ont une charge plus petite que le double du déséquilibre. La fonction $SearchClosest$ donne, pour un ensemble non vide de mailles X, un nombre c et une application w associant des charges aux mailles, l'élément de X ayant la charge la plus proche de c. À l'aide d'une recherche dichotomique, $SearchClosest$ peut être exécutée en $O(\log |X|)$ opérations. L'algorithme s'arrête lorsque le double du déséquilibre est plus petit que la charge minimum de la partie surchargée.

5.3 . Généralisation de la recherche au k-partitionnement

VNBest, ainsi que son extension réalisée dans la section précédente, se limite au partitionnement en deux parties (bi-partitionnement). En pratique, le nombre de parties peut être arbitrairement grand. Dans cette section, nous proposons une généralisation de la méthode de recherche vue dans la section précédente au k-partitionnement.

L'idée pour la généralisation de VNBest au k-partitionnement est de ne déplacer les mailles que depuis la partie la plus surchargée – celle dont le déséquilibre est le plus élevé – vers la partie la plus souschargée – celle dont le déséquilibre est le plus faible. De cette façon, de nombreux déplacements n'ont pas besoin d'être considérés, et cela nous permet aussi de

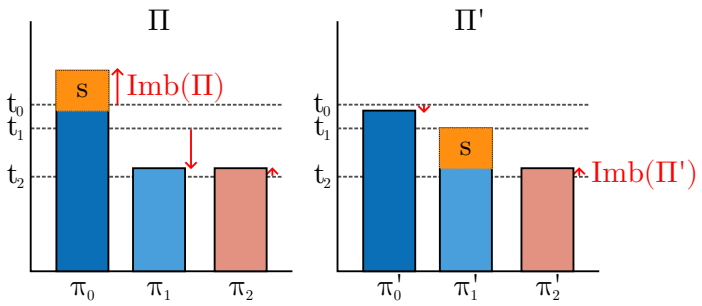


Figure 5.2 – Déplacement d'un poids de la partie la plus surchargée (π_0) vers la partie la plus souschargée (π_1). Après le déplacement, π'_0 est souschargée, π'_1 est parfaitement chargée, et π'_2 est surchargée. Le déséquilibre de Π' vaut donc le déséquilibre de la partie π'_2.

déplacer des mailles aux charges plus grandes avant de rendre la partie souschargée la plus surchargée.

La méthode de recherche du meilleur déplacement que nous avons mis en place dans la section précédente repose sur le calcul du gain de chaque maille. Le calcul de ce gain repose sur la proposition 1 (page 66), qui avance que, lorsqu'il n'y a que deux parties, le déséquilibre de ces deux parties sont des nombres opposés. Lorsqu'il y a au moins trois parties, la somme des déséquilibres est toujours nulle mais, comme montré sur la Figure 5.2, il est possible qu'une troisième partie puisse devenir la partie la plus surchargée après un déplacement. La formule du gain donnée en proposition 2 n'est doit alors prendre en compte cette troisième partie.

Nous pouvons adapter la définition du déplacement d'une maille, ainsi que la définition du gain de ce déplacement, préalablement dans la Définition 5.3 et la Définition 5.4, à k parties. Dans notre cas, les déplacements se font exclusivement de la partie la plus surchargée vers la partie la plus souschargée. Proposons maintenant une formule explicite du gain pour le k-partitionnement.

Proposition 3 (Formule explicite du gain, cas du k-partitionnement)**.** *Soient un maillage $\mathcal{M}$, une partition $\Pi = (P_i)_{0 \leq i < k}$ de $\mathcal{M}$, une application $w : \mathcal{M} \longrightarrow \mathbb{R}^+$ associant une charge de calcul à chaque maille de $\mathcal{M}$, et $T = (t_i)_{0 \leq i < k}$ une famille de réels positifs dont la somme égale à la somme des charges de toutes les mailles. Notons P_a la partie la plus surchargée, P_x la deuxième partie la plus surchargée, et P_b la partie la plus souschargée. Alors, pour tout $m \in P_a$, le gain en équilibre du déplacement de m de la partie P_a vers la partie P_b est :*

$$Gain(\Pi, m) = \begin{cases} w(m) & \textit{si } w(m) \leq \sigma-, \\ Imb(P_a, t_a) - Imb(P_x, t_x) & \textit{si } \sigma- < w(m) < \sigma+, \\ Imb(P_a, t_a) - Imb(P_b, t_b) - w(m) & \textit{si } \sigma+ \leq w(m). \end{cases}$$

où

$$\begin{cases} \sigma_0 & = \frac{Imb(P_a,t_a)-Imb(P_b,t_b)}{2}, \\ \sigma_- & = \mathsf{min}(\sigma_0, Imb(P_a, t_a) - Imb(P_x, t_x)), \\ \sigma_+ & = \mathsf{max}(\sigma_0, Imb(P_x, t_x) - Imb(P_b, t_b)). \end{cases}$$

Démonstration. Nous appliquons d'abord la définition du gain (cf. Définition 5.4), puis la définition du déséquilibre (cf. Définition 5.1).

$$\begin{aligned} Gain(\Pi, m) &= Imb(\Pi, T) - Imb(Move(\Pi, m), T) \\ &= Imb(P_a, t_a) - \\ &\quad \mathsf{max}\left(Imb(P_a, t_a) - w(m), Imb(P_b, t_b) + w(m), Imb(P_x, t_x)\right) \end{aligned}$$

Nous obtenons une formule avec un maximum. Ce maximum concerne l'ensemble des parties de $Move(\Pi, m)$. Cependant, il est possible de ne considérer que les parties sujettes au déplacement, ainsi que la partie P_x, car toutes les autres ont en tout cas un déséquilibre plus faible que celui de P_x. Pour le calcul du gain, nous distinguons alors trois cas :

1. Ou bien $Imb(P_a, t_a) - w(m)$ est le maximum, alors nous avons :

   $$Gain(\Pi, m) = w(m).$$

   Ce cas est vrai si et seulement si :

   $$\begin{cases} Imb(P_b, t_b) + w(m) \leq Imb(P_a, t_a) - w(m), \\ Imb(P_x, t_x) \leq Imb(P_a, t_a) - w(m). \end{cases}$$

   Ce qui est équivalent à dire que :

   $$\begin{cases} w(m) \leq \frac{Imb(P_a,t_a)-Imb(P_b,t_b)}{2}, \\ w(m) \leq Imb(P_a, t_a) - Imb(P_x, t_x). \end{cases}$$

2. Ou bien $Imb(P_b, t_b) + w(m)$ est le maximum, alors nous avons :

   $$Gain(\Pi, m) = Imb(P_a, t_a) - Imb(P_b, t_b) - w(m).$$

Ce cas est vrai si et seulement si :

$$\begin{cases} Imb(P_a, t_a) - w(m) \leq Imb(P_b, t_b) + w(m), \\ Imb(P_x, t_x) \leq Imb(P_b, t_b) + w(m). \end{cases}$$

Ce qui est équivalent à dire que :

$$\begin{cases} w(m) \geq \frac{Imb(P_a, t_a) - Imb(P_b, t_b)}{2}, \\ w(m) \geq Imb(P_x, t_x) - Imb(P_b, t_b). \end{cases}$$

3. Ou bien $Imb(P_x, t_x)$ est le maximum, alors nous avons :

$$Gain(\Pi, m) = Imb(P_a, t_a) - Imb(P_x, t_x).$$

Ce cas est vrai si et seulement si :

$$\begin{cases} Imb(P_a, t_a) - w(m) \leq Imb(P_x, t_x), \\ Imb(P_b, t_b) + w(m) \leq Imb(P_x, t_x). \end{cases}$$

Ce qui est équivalent à dire que :

$$Imb(P_a, t_a) - Imb(P_x, t_x) \leq w(m) \leq Imb(P_x, t_x) - Imb(P_b, t_b).$$

Reprenons les notations σ_0, σ_- et σ_+. D'après les trois points que nous venons d'énumérer, nous avons la formule de gain qui est la suivante :

$$Gain(\Pi, m) = \begin{cases} w(m) & \text{si } w(m) \leq \sigma-, \\ Imb(P_a, t_a) - Imb(P_x, t_x) & \text{si } (*), \\ Imb(P_a, t_a) - Imb(P_b, t_b) - w(m) & \text{si } \sigma+ \leq w(m). \end{cases}$$

$$(*) \quad Imb(P_a, t_a) - Imb(P_x, t_x) \leq w(m) \leq Imb(P_x, t_x) - Imb(P_b, t_b)$$

Nous montrons maintenant que la condition $(*)$ est équivalente à $\sigma_- \leq w(m) \leq \sigma_+$. Pour cela nous montrons :

$$\begin{aligned} & Imb(P_a, t_a) - Imb(P_x, t_x) \leq Imb(P_x, t_x) - Imb(P_b, t_b) \\ \iff & \sigma_- = Imb(P_a, t_a) - Imb(P_x, t_x) \\ \iff & \sigma_+ = Imb(P_x, t_x) - Imb(P_b, t_b). \end{aligned}$$

□

Le gain du déplacement d'une maille en fonction de la charge de cette maille peut avoir l'une des deux allures montrées dans les tableaux de variations 5.2 et 5.3. Le tableau de variation 5.2 décrit le cas où $\sigma_- = Imb(P_a, t_a) - Imb(P_x, t_x) < Imb(P_x, t_x) - Imb(P_b, t_b) = \sigma_+$, c'est-à-dire qu'une partie autre que celles du déplacement devient la plus surchargée, alors le gain reste constant entre σ_- et σ_+. Le tableau de variation 5.3 décrit le cas contraire, où aucune autre partie peut devenir la plus surchargée.

$w(m)$	0	σ_-	σ_+	$2 \cdot \sigma_0$	$+\infty$
$Gain(\Pi, m)$	$w(m)$	h	$f(w(m))$		
$Gain(\Pi, m)$	0 ↗	h →	h ↘	0 ↘	

Table 5.2 – Tableau de variation du gain en fonction de la charge de la maille déplacée, pour une partition donnée et pour un déplacement depuis la partie la plus surchargée vers la partie la plus souschargée, cas du k-partitionnement, cas où une autre partie peut devenir la plus surchargée. Notons $h = Imb(P_a, t_a) - Imb(P_x, t_x)$ et $f(x) = Imb(P_a, t_a) - Imb(P_b, t_b) - x$.

$w(m)$	0	σ_0	$2 \cdot \sigma_0$	$+\infty$
$Gain(\Pi, m)$	$w(m)$	$f(w(m))$		
$Gain(\Pi, m)$	0 ↗	σ_0 ↘	0 ↘	

Table 5.3 – Tableau de variation du gain en fonction de la charge de la maille déplacée, pour une partition donnée et pour un déplacement depuis la partie la plus surchargée vers la partie la plus souschargée, cas du k-partitionnement, cas où il n'y a pas d'autre partie pouvant devenir la plus surchargée. Notons $f(x) = Imb(P_a, t_a) - Imb(P_b, t_b) - x$.

Comme nous avons toujours $\frac{\sigma_- + \sigma_+}{2} = \sigma_0$, le meilleur déplacement est celui de la charge la plus proche de σ_0. Nous détaillons donc notre extension de VNBest pour le k-partitionnement dans l'Algorithme 8.

Algorithme 8 Version détaillée de VNBest, dans le cas du k-partitionnement, et ou les parties ont des cibles de charge

1 : **procedure** VNBest($\mathcal{M}$, $\Pi = (P_i)_{0 \leq i < k}$, $T = (t_i)_{0 \leq i < k}$, w)
2 : **loop**
3 : $P_{\text{souschargée}}, P_{\text{surchargée}} \leftarrow \Pi$
4 : $\sigma_0 \leftarrow \frac{Imb(P_{\text{souschargée}}, t_{\text{souschargée}}) - Imb(P_{\text{surchargée}}, t_{\text{surchargée}})}{2}$
5 : $m \leftarrow SearchClosest(P_{\text{surchargée}}, \sigma_0, w)$
6 : **if** $2 \cdot \sigma_0 \leq w(m)$ **then**
7 : **break** ▷ Aucun déplacement ne baisse le déséquilibre
8 : **end if**
9 : $\Pi \leftarrow Move(\Pi, m)$
10 : **end loop**
11 : **end procedure**

L'algorithme 8 est similaire à celui pour le bi-partitionnement (page 70). La seule différence est que la maille choisie est celle dont la charge est la plus proche de σ_0. Cet algorithme est une généralisation du précédent, car dans le cas du bi-partitionnement σ_0 vaut le déséquilibre de la partition.

Conclusion

Dans ce chapitre, nous avons défini le problème de partitionnement de nombres à cibles de charges, qui intervient quand l'ensemble des unités de calcul sur lequel il faut répartir la charge de calcul n'est pas homogène. Pour résoudre ce problème dans le cadre du chaînage d'algorithmes, nous avons étendu l'algorithme de bi-partitionnement de nombres VNBest de façon à prendre en compte les cibles de charge. Notamment, il a fallu modifier la méthode de recherche des meilleurs déplacements. Ensuite, nous avons généralisé cette méthode pour que VNBest puisse être appliqué sur des partitions ayant plus de deux parties.

Afin d'étoffer la méthode de chaînage d'algorithmes que nous avons défini dans le chapitre précédent, nous allons, dans le chapitre suivant, nous intéresser au partitionnement de maillages cartésiens.

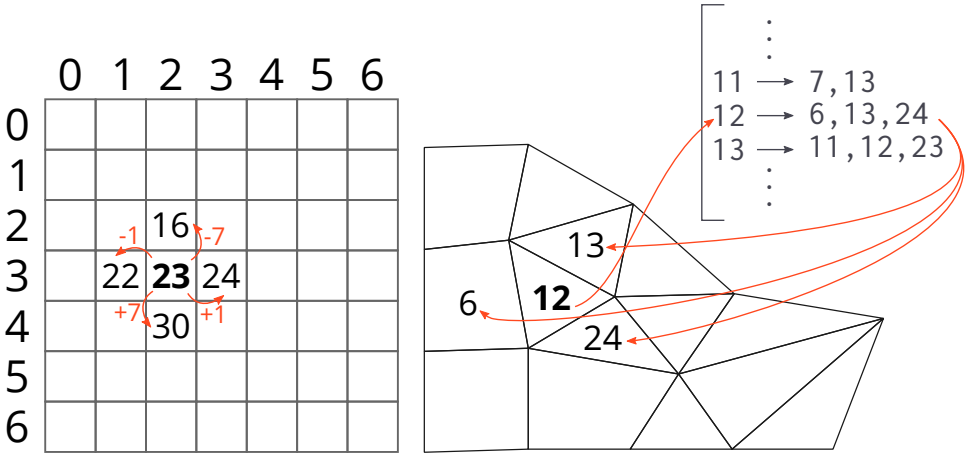
6 - Développement d'une méthode de partitionnement de maillages cartésiens

Dans une perspective de performances, les simulations physiques peuvent se tourner vers des maillages plus simples, dits « structurés », qui offrent des propriétés supplémentaires par rapport aux maillages généraux, dits « non-structurés ». Parmi les maillages structurés, nous pouvons citer les maillages cartésiens, qui sont des grilles régulières (voir Figure 6.1a) dont la taille des mailles peut toutefois varier. Les propriétés de tels maillages peuvent permettre par exemple de diminuer la quantité de données à stocker et traiter, et de diminuer les accès mémoire. Cette section considère les maillages cartésiens, leurs propriétés, et la façon dont le partitionnement peut en tirer parti, d'abord dans le cadre d'un algorithme de partitionnement géométrique bien connu : la bissection récursive de coordonnées, puis avec un nouvel algorithme de réduction de sur-coûts.

Comme indiqué précédemment, les maillages cartésiens sont dans notre cas des simples grilles, 2D ou 3D, comme montré sur la Figure 6.1. Travailler sur ces structures offre deux avantages notables par rapport aux maillages non structurés :

1. D'abord, le voisinage de chaque maille est connu à l'avance, sans besoin de le stocker en mémoire. Typiquement, pour une grille 2D, si les mailles de la grille sont numérotées par des indices $(i, j),\ 0 \leq i < n,\ 0 \leq j < p$, alors les mailles voisines sont les mailles $\{(i + d_i, j + d_j) \mid d_i \in \{-1, 1\},\ d_j \in \{-1, 1\}\}$. Lorsque les mailles sont numérotées lignes par lignes, par exemple, l'indice des mailles voisines sont ceux indiqués par la Figure 6.1a. Par conséquent, les accès mémoire peuvent être contigus et prévisibles. Par contre, pour un maillage non structuré, accéder aux voisines d'une maille demande une recherche dans un index (par exemple une matrice d'adjacence), comme montré sur la Figure 6.1b. Les accès mémoire sont alors moins prévisibles.
2. Deuxièmement, il est possible de libérer de l'espace mémoire en ne stockant pas les coordonnées des sommets de mailles, et d'utiliser à la place leur indice. En effet, la taille des mailles d'une grille et leurs coordonnées n'ont que peu d'importance pour le partitionnement, car leur indice (i, j) donne suffisamment d'information sur la relation de voisinage du maillage. Le poids associé à chaque maille peut dépendre de sa taille, mais ces poids sont stockés de toute manière.

Certaines simulations physiques travaillant sur maillage cartésien



(a) Accès aux voisines de la maille n°23 d'une grille (maillage structuré) de taille 7×7, numérotée ligne par ligne. Le nombre de voisines et leurs indices sont connus à l'avance.

(b) Accès aux voisines de la maille n°12 d'un maillage non structuré. Le nombre de voisines et leurs indices sont récupérés depuis un index à part.

Figure 6.1 – Exemples d'accès aux voisines d'une maille pour le cas structuré et non-structuré.

et s'exécutant sur machine distribuée tirent partie des avantages précédemment cités. Pour cela, chaque unité de calcul travaille sur un sous-ensemble cartésien du maillage. Cependant, cela pose une contrainte supplémentaire au partitionnement et, certains problèmes de partitionnement sont alors plus adaptés que d'autres. Le partitionnement topologique (Définition 2.8), par exemple, ne prend pas en compte une telle contrainte et, le code de simulation doit alors « ajuster » la partition à volée de façon à obtenir des parties cartésiennes. Par contre, les bissections géométriques sont plus adaptées. Par exemple, la bissection récursive de coordonnées (RCB) [8] découpe le maillage avec des lignes droites parallèles aux axes et obtient donc des parties toujours cartésiennes.

Les implémentations typiques de RCB s'appliquent à tout maillage, structuré ou non [9]. Dans ce chapitre, nous allons spécialiser RCB pour les maillages cartésiens, afin de profiter des améliorations de performances précédemment citées.

6.1. Définition du problème de partitionnement de maillages cartésien

Avant de présenter deux algorithmes dédiés au partitionnement de maillage cartésien, cette section définit formellement le maillage cartésien et le problème de partitionnement de tels maillages. Les définitions sont pour des maillages 3D, mais il est possible de les restreindre aux maillages 2D.

Définition 6.1 (Maillage cartésien, grille)**.** *Soit $S = (s_0, s_1, s_2) \in \mathbb{N}^{*3}$. Alors la grille 3D $\mathcal{G}$ de taille S est l'ensemble des points de $\mathbb{N}^3$ dont les composantes sont une à une plus petites que celles de S :*

$$\mathcal{G} = \left\{(c_0, c_1, c_2) \in \mathbb{N}^3 \mid \forall i, 0 \leq c_i < s_i\right\}.$$

Nous appelons les éléments de $\mathcal{G}$ mailles*.*

Définition 6.2 (Sous-grille)**.** *Soient $\mathcal{G}$ une grille 3D de taille S, ainsi que deux points $A = (a_0, a_1, a_2), B = (b_0, b_1, b_2) \in \mathbb{N}^3$. Alors la sous-grille γ de $\mathcal{G}$ comprise entre A et B est l'ensemble des éléments de $\mathcal{G}$ dont les composantes sont une à une comprises entre celles de A et celles de B :*

$$\gamma = \{(c_0, c_1, c_2) \in \mathcal{G} \mid \forall i, a_i \leq c_i < b_i\}.$$

Alors le problème de partitionnement de maillage, précédemment défini en Définition 2.1 (page 33), est spécialisé dans la Définition 6.3, où une contrainte sur les parties est ajoutée.

Définition 6.3 (k-partitionnement de maillage cartésien)**.** *Étant donné une grille $\mathcal{G}$, une application $w : \mathcal{M} \longrightarrow \mathbb{R}^+$ donnant la charge de calcul de chaque maille, le problème du k-partitionnement de maillage cartésien consiste à trouver une famille $\Pi = (P_i)_{0 \leq i < k}$ de sous-ensembles de $\mathcal{G}$ qui satisfait :*

- Partition *: Π est une partition de $\mathcal{G}$;*
- Objectif d'équilibre *: la charge de calcul de chaque ensemble P_i, définie comme $\sum_{m \in P_i} w(m)$, est la plus proche possible de celle des autres ensembles ;*
- Objectif de moindre surcoût *: le coût (par exemple communication) induit par la décomposition Π de $\mathcal{G}$ est minimal,*
- Contrainte cartésienne *: chaque partie P_i de Π est une sous-grille de $\mathcal{G}$.*

6.2 . Spécialisation de RCB pour les maillages cartésiens

Le partitionnement de maillages cartésien doit satisfaire une contrainte supplémentaire par rapport au problème de partitionnement défini dans le Chapitre 2 : toutes les parties doivent aussi être cartésiennes.

Il se trouve que la bissection récursive de coordonnées (RCB) répond déjà à ce problème : elle peut travailler sur l'ensemble $\mathcal{G}$, où découper récursivement selon les axes revient à créer deux nouvelles sous-grilles à chaque itération. Nous obtenons alors toujours des parties cartésiennes.

Toutefois, la version traditionnelle de RCB, définie en Algorithme 9, fait beaucoup d'opérations redondantes quand le maillage à partitionner est une grille. Le but de cette section est de mettre en valeur ces redondances, et de montrer qu'il est possible de spécialiser l'algorithme pour tirer parti de la structure de grille, afin de diminuer le nombre d'opérations et la quantité mémoire requis par son exécution.

En guise de comparaison, nous décrivons dans l'Algorithme 9 une itération de RCB pour les maillages non structurés. Étant donné un tableau de points X sur un axe, leur poids W, la somme de ces poids w_{total}, et la tolérance sur le déséquilibre ε, la fonction « RcbIter » fait deux choses :

- Trouver une valeur x tel que le poids des éléments de X plus petits que x est la moitié de w_{total}, plus ou moins $\varepsilon \cdot w_{total}$;
- Réordonner X et W de telle sorte que les éléments plus petits que x soient stockés avant les autres en mémoire, de façon à pouvoir exécuter les itérations suivantes de RCB.

La première étape consiste à obtenir, grâce à ε et w_{total}, les bornes min et max du poids des deux parties. Nous réalisons ensuite une recherche dichotomique, tant qu'il y a des éléments de X dans l'intervalle de recherche $[x_{min}; x_{max}]$. Pour chaque valeur testée x, nous regardons si le poids des éléments plus petits que x (le poids de la partie de gauche) est compris entre les bornes de poids de parties w_{ε_min} et w_{ε_max}. Si oui, nous utilisons la fonction « TriPartiel » pour réordonner les poids dans X et nous retournons les nouveaux tableaux. Si l'intervalle de recherche n'a plus d'élément de X, alors il n'y a pas de découpage qui satisfait la tolérance ε.

La valeur de retour de « RcbIter » permet de démarrer directement l'itération suivante de RCB. En effet, une fois les tableaux X et W réordonnés, leurs éléments peuvent être utilisés comme données d'entrée pour les deux prochains appels à « RcbIter ». La valeur de w_{moitie} est utilisée pour calculer le prochain w_{total}.

Dans le cas du partitionnement de maillage cartésien, il est possible de réduire le nombre d'opérations sur trois points :

Algorithme 9 Itération de RCB, version pour les maillages non structurés

```
1 : function RcbIter(X, W, w_total, ε)
2 :     w_ε_min ← w_total/2 · (1 − ε)
3 :     w_ε_max ← w_total/2 · (1 + ε)
4 :     x_min ← min(X)
5 :     x_max ← max(X)
6 :     while X ∩ [x_min; x_max] ≠ ∅ do
7 :         x ← (x_min + x_max)/2
8 :         w_moitie ← Σ_{i|X[i]<x} W[i]
9 :         if w_moitie < w_ε_min then
10 :            x_min ← x
11 :        else if w_moitie > w_ε_max then
12 :            x_max ← x
13 :        else
14 :            X', W' ← TriPartiel(X, W, x)
15 :            return X', W', x, w_moitie
16 :        end if
17 :    end while
18 :    X', W' ← TriPartiel(X, W, x_min)
19 :    return X', W', x_min, w_moitie
20 : end function
```

1. La taille de X et W ;
2. La somme w_{moitie} des poids des éléments plus petits que x ;
3. Le tri partiel des tableaux X et W.

Pour le premier point, la recherche dichotomique se fait sur un seul axe. Pour un maillage cartésien, cela signifie que la recherche se conduit colonne par colonne, ou ligne par ligne, selon l'axe de recherche. Il est alors possible d'effectuer cette recherche non pas sur l'ensemble des poids, mais sur un tableau de poids de plus petite taille qui ne contient que la somme des poids par colonne, ligne, ou aile respectivement.

Pour le deuxième point, nous supposons que la grille est numérotée ligne par ligne. Alors, le tableau W contenant les sommes des charges de calcul des mailles étant à la même coordonnée sur l'axe, est trié selon l'axe. Ainsi, les éléments à sommer sont contigus dans W, et il est possible d'effectuer la somme sans visiter tout W.

Pour le troisième point, une itération de RCB sur grille n'a pas besoin de réordonner le tableau de poids avant de le retourner, car nous supposons que les données d'entrée sont déjà triées. Ce changement ne diminue pas la complexité, mais permet d'augmenter les performances en

parallèle, car le tri partiel n'est pas trivialement parallélisable, car les accès mémoire ne sont ni contigus ni localisés (car les éléments à réordonner ne le sont pas non plus).

D'autre part, la condition d'arrêt de l'algorithme $X \cap [x_{min}; x_{max}] \neq \emptyset$ peut être changée en la proposition équivalente $x_{min} < x_{max}$. Ainsi, il n'y a plus besoin de suivre l'évolution de l'ensemble $X \cap [x_{min}; x_{max}]$.

Algorithme 10 Itération de RCB, version cartésienne

```
1 : function RcbIter(W, w_total, ε)
2 :     w_ε_min ← w_total/2 · (1 − ε)
3 :     w_ε_max ← w_total/2 · (1 + ε)
4 :     x_min ← 0                                  ▷ Bornes de X en O(1).
5 :     x_max ← |W|
6 :     w_min ← 0
7 :     while x_min < x_max do                     ▷ Condition d'arrêt plus simple.
8 :         x ← ⌊(x_min + x_max)/2⌋
9 :         w_moitie ← w_min + Σ_{i=x_min}^{x} W[i]   ▷ O(|X|/2^iter) éléments sommés.
10 :        if w_moitie < w_ε_min then
11 :            x_min ← x
12 :            w_min ← w_moitie
13 :        else if w_moitie > w_ε_max then
14 :            x_max ← x
15 :        else
16 :            return x, w_moitie                 ▷ Pas de réordonnement.
17 :        end if
18 :    end while
19 :    return x_min, w_min                        ▷ Pas de réordonnement.
20 : end function
```

6.3 . Résultats et analyse

Nous avons aussi effectué des mesures de performance comparant les deux versions de RCB. Les résultats sont présentés sur la Table 6.1. L'idée est de mesurer la taille du plus gros maillage qu'il est possible de partitionner à l'aide de chacune des versions, puis de mesurer le temps et la quantité mémoire requise pour le partitionnement d'un tel maillage.

Nous prenons trois grilles 2D de 600, $40\,000$ et $120\,000$ mailles de côté respectivement. La première grille a donc $360\,000$ mailles, la deuxième en a 1.6×10^9, et la troisième en a 14×10^9. La charge de chaque maille m est fixée à $x + y$, où (x, y) est l'indice de la maille m. Choisir une distribu-

tion de charges qui n'est pas uniforme est important pour l'expérience, car notre implémentation de RCB effectue une recherche dichotomique, et le nombre d'itérations de cette recherche dépend de la charge des mailles.

Sur ces deux grilles, nous exécutons chaque version de RCB trois fois : la première fois pour partitionner en deux parties (un seul niveau de récursion de RCB), la deuxième en quatre parties (deux niveaux de récursion), et la troisième fois en 4096 parties (douze niveaux de récursion).

L'exécution de RCB se fait sur une machine « AMD Milan » à deux processeurs, 128 threads au total, et une capacité mémoire de 250 Go. Certains tests ont fait déborder la capacité mémoire, et les résultats sont notés avec un tiret. Les mesures de temps d'exécution sont faites avec la bibliothèque de benchmarking Criterion [65]. Les mesures de mémoire sont la métrique « *Maximum resident set size* » de l'outil « GNU Time ».

Table 6.1 – Résultats d'expérience pour comparer le temps d'exécution (avant-dernière colonne) et la quantité mémoire requise (dernière colonne) des deux versions de RCB sous plusieurs cas. Nous faisons varier le nombre de parties entre 2 et 4096 (deuxième colonne) et la taille du maillage cartésien (troisième colonne).

Version de RCB	Parties	Taille de la grille	Temps	Mémoire
Traditionnel	2	600×600	19 ms	27 Mo
Cartésien	2	600×600	3.6 ms	15 Mo
Traditionnel	4	600×600	24 ms	27 Mo
Cartésien	4	600×600	4.2 ms	16 Mo
Traditionnel	4096	600×600	38 ms	28 Mo
Cartésien	4096	600×600	72 ms	17 Mo
Traditionnel	2	$40\,000 \times 40\,000$	4.4 s	75 Go
Cartésien	2	$40\,000 \times 40\,000$	0.42 s	25 Go
Traditionnel	4	$40\,000 \times 40\,000$	7.2 s	75 Go
Cartésien	4	$40\,000 \times 40\,000$	1.3 s	25 Go
Traditionnel	4096	$40\,000 \times 40\,000$	30 s	75 Go
Cartésien	4096	$40\,000 \times 40\,000$	6.9 s	25 Go
Traditionnel	2	$120\,000 \times 120\,000$	—	—
Cartésien	2	$120\,000 \times 120\,000$	3.2 s	0.22 To
Traditionnel	4	$120\,000 \times 120\,000$	—	—
Cartésien	4	$120\,000 \times 120\,000$	13 s	0.22 To
Traditionnel	4096	$120\,000 \times 120\,000$	—	—
Cartésien	4096	$120\,000 \times 120\,000$	59 s	0.22 To

À partir des résultats de la Table 6.1, nous remarquons d'abord que le nombre de parties – le niveau de récursion de RCB – a très peu d'influence sur l'occupation mémoire des deux versions. En effet, les deux versions

sont implémentées de façon à éviter les allocations mémoire, c'est-à-dire que les niveaux de récursion réutilisent les zones mémoires qui stockent les tableaux de charges et de points des niveaux précédents. Pour la version cartésienne effectue par ailleurs l'allocation du tableau des sommes de charges par axe de découpe, mais ce tableau est de plus petite taille au fil des niveaux, ce qui annule l'influence du nombre de niveaux de récursion sur l'occupation mémoire de l'algorithme.

La vitesse d'exécution de la version cartésienne est généralement environ cinq fois plus rapide que celle de la version traditionnelle. Les cas particuliers sont le cas du partitionnement en 4096 parties sur la grille de 600×600, où la version traditionnelle est deux fois plus rapide, et le cas du bi-partitionnement sur la grille de $40\,000 \times 40\,000$, où la version cartésienne est dix fois plus rapide.

Il est sûrement possible d'améliorer le temps d'exécution du RCB cartésien. En particulier, aux premiers niveaux de récursion, quand la partie du maillage à découper en deux est encore grande, la plupart du temps est passé dans les accès mémoire. Actuellement, la version cartésienne commence la première découpe selon l'axe des ordonnées pour les grilles numérotées ligne par ligne, et l'axe des abscisses pour les grilles numérotées colonne par colonne. De cette manière, le calcul des sommes des poids suivant l'axe effectue des accès mémoires prévisibles par le matériel informatique, et donc est plus rapide d'exécution. Cependant, nos grilles de test sont carrées, et la deuxième itération doit alors effectuer des sommes de même taille, mais suivant l'axe opposé. Dans ce cas, le matériel informatique ne prévoit pas correctement les accès mémoire de notre implémentation, et ce niveau de récursion prend plus de temps que nécessaire. En utilisant une meilleure disposition des charges de calcul en mémoire, par exemple sous forme de tuiles, il devrait être possible d'améliorer le temps total de RCB cartésien.

Conclusion

Dans cette section nous avons spécialisé une implémentation de la *Recursive Coordinate Bisection* (RCB) pour les maillages cartésiens, afin de la rendre plus rapide à exécuter, et de diminuer la capacité mémoire requise pour l'exécuter. Lors de nos expériences, la version cartésienne de RCB est en moyenne cinq fois plus rapide que la version traditionnelle, et utilise deux à trois fois moins d'espace mémoire. Nous pensons que la version cartésienne peut être encore plus rapide en changeant l'agencement des charges de calcul en mémoire.

Dans les trois derniers chapitres, nous avons proposé une nouvelle méthode pour partitionner des maillages à base de chaînage d'algorithmes, afin de pouvoir optimiser différentes métriques, comme la

coupe ou le déséquilibre de charge. Nous avons étoffé cette méthode à l'aide d'une extension de VNBest qui prend en charge des poids cibles par partie, afin d'équilibrer la charge pour des machines où les unités de calcul sont hétérogènes. Nous avons ensuite spécialisé RCB pour les maillages cartésiens pour accélérer l'étape de partitionnement dans le cas où une simulation utilise des maillages cartésiens.

Dans le chapitre suivant, nous allons proposer un nouvel outil de partitionnement, qui implémente les idées présentées dans les trois derniers chapitres.

Troisième partie

Développement de Coupe

7 - Conception d'un nouvel outil de partitionnement modulaire, robuste et dédié aux maillages

Dans la partie précédente, nous avons proposé une nouvelle méthode de partitionnement de maillages, à base de chaînage d'algorithmes. Nous avons aussi développé des extensions à deux algorithmes existants : VN-Best et RCB. Pour que ces développements soient utilisables par un grand nombre de codes de calculs existants, il faut les implémenter dans un outil de partitionnement.

Bien qu'il serait possible d'implémenter cette composition d'algorithmes et métriques dans un outil existant – rappelons que Scotch permet déjà de chaîner des algorithmes arbitrairement choisis par l'utilisateur – il peut être difficile d'adapter leur fonctionnement à de nouvelles métriques, comme la prise de l'occupation mémoire des parties et des mailles fantômes [12]. C'est pourquoi nous nous concentrons sur la création d'un nouvel outil de partitionnement de maillages, que nous appelons « Coupe », permettant de chaîner des algorithmes résolvant différents problèmes de partitionnement.

Dans ce chapitre, nous allons spécifier les choix technologiques qui ont été fait pour développer Coupe. D'abord, dans la Section 7.1, nous allons sélectionner une pile logicielle pour développer Coupe. Ensuite, dans la Section 7.2 (page 93), nous allons permettre l'intégration de Coupe dans des codes de calcul existants. Enfin, dans la Section 7.3 (page 99), nous allons créer une boite à outils pour facilement mesurer la qualité des résultats et le temps d'exécution de différentes chaînes d'algorithmes, sous différentes conditions.

7.1 . Sélection d'une pile logicielle haute performance et robuste pour le développement de Coupe

Quand un code de calcul partitionne le maillage au début de son exécution, ou repartitionne le maillage pendant son exécution, il est préférable que l'outil de partitionnement partitionne le plus rapidement possible, et ait une efficacité proche de un (cf. Définition 1.1, page 21). Dans un modèle de programmation BSP, encore très utilisé en pratique, l'étape de partitionnement peut être bloquante.

Un autre critère pour le choix de la pile logicielle est qu'elle doit nous permettre de construire un outil versatile. En effet, chaque simulation nu-

mérique travaille avec des structures de données qui lui sont adaptées. L'outil de partitionnement doit pouvoir travailler à partir de ces structures de données, ou du moins faciliter la conversion, sans demander de format précis.

Enfin, notre outil de partitionnement doit être robuste, afin d'éviter que l'outil se comporte de manière surprenante lors de cas pathologiques ou entrées invalides. Dans notre cas, où les algorithmes implémentés résolvent des problèmes de partitionnement différents (cf. Chapitre 2, page 33), et où les algorithmes sont chaînés, la robustesse est une problématique encore plus importante. Différents types de structures de données sont utilisées. Il ne faut pas que cela engendre des bogues.

Nous choisissons donc de développer notre outil en Rust, qui nous permet d'assurer de ne pas corrompre la mémoire, ou déclencher des erreurs de segmentation, ce qui pose problème notamment lorsque l'outil de partitionnement s'exécute sur les mêmes unités de calcul que la simulation.

Rust nous permet d'être versatiles grâce à la programmation générique. Une fonction en Rust peut être définie pour une famille de types (T_i). Par exemple, le Listing 7.1 décrit une fonction `part_loads` qui calcule le déséquilibre d'une famille de charges de calcul. Cette fonction est définie pour un ensemble de couples (W, I) de types. Ainsi, il est possible d'implémenter une fonction une fois, et pouvoir l'appliquer pour une multitude de types, ici par exemple `u32` pour W et `HashMap<u32, u32>` pour I.

Pour effectuer des opérations sur un ensemble de types, il est souvent nécessaire de restreindre cet ensemble à celui des types qui héritent d'une interface, de façon à garantir qu'il est possible d'en passer des valeurs à certaines fonctions. En Rust, ces interfaces sont appelées *traits*. La bibliothèque standard en définit pour les opérations numériques `Add`, `Mul`, etc., ainsi qu'une interface `Iterator`, permettant de traverser n'importe quel type de collection de données. La fonction `part_loads` n'est donc valide que pour les types `I` qui héritent de `Iterator<Item = (W, PartId)>`. Nous pouvons ainsi implémenter les algorithmes de partitionnement pour un grand nombre de types de données.

Dans le Listing 7.1, le type `I`, doit implémenter l'interface (le *trait*) `Iterator`, c'est-à-dire qu'il doit exister une implémentation de chaque fonction définie dans l'interface `Iterator`. Les interfaces en Rust peuvent aussi spécifier des paramètres : ici, le paramètre `Item` indique que `weights` permet d'itérer sur des couples `(W,PartId)`, avec `W` étant un autre type générique possédant l'opérateur d'addition-affection (`+=`, défini par le trait `AddAssign`) ainsi qu'une valeur par défaut (définie par le trait `Default`). Ainsi, nous pouvons appeler `fold` sur notre itérateur `weights`. `fold` applique la fonction passée en deuxième argument successivement sur

l'accumulateur initialisé par `HashMap::new()` et les élément de `weights`, et retourne la valeur finale de l'accumulateur.

Listing 7.1 Programmation générique en Rust.

```
type PartId = u32;

fn part_loads<W, I>(weights: I) -> HashMap<PartId, W>
where
  // `I` est une collection de tuples.
  I: Iterator<Item = (W, PartId)>,
  // `W` est additionnable sur place et a un élément nul.
  W: AddAssign + Default,
{
  // `Iterator` donne accès à `I::fold()`.
  weights.fold(HashMap::new(), |hmap, (w, id)| {
    // `Default` donne accès à `W::default()`.
    let pload = *hmap.entry(id).or_insert(W::default());
    // `AddAssign` donne accès à l'opérateur `+=`.
    *pload += w;
    hmap
  })
}
```

La programmation générique en Rust a plusieurs avantages par rapport aux templates C++. D'abord, spécifier les restrictions sur les types (`Add`, `Iterator`, ...) est obligatoire. L'utilisateur obtient ainsi de meilleurs messages d'erreur lors de l'utilisation de bibliothèques externes. L'équivalent C++ (*concepts* et *constraints*) n'existe que depuis la version C++20 du langage, et leur utilisation n'est pas obligatoire.

Aussi, elles permettent de contraindre un type à être *thread-safe*. Il existe deux définitions de *thread-safe* en Rust :

- `Send` : les objets de types qui implémentent ce trait peuvent être lus et écrits par plusieurs threads, mais seulement un en même temps.
- `Sync` : un type `T` est `Sync` si et seulement si les références partagées `&T` vers ce type sont `Send`, c'est-à-dire, plusieurs threads peuvent lire les objets de ce type en même temps.

Listing 7.2 Algorithme de partitionnement implémenté de manière générique

```
// Trait qui définie une relation de voisinage.
trait Topology {
  // neighbors retourne l'indices des voisines
  // de la `i`ème maille.
  fn neighbors(&self, i: u32) -> impl Iterator<Item=u32>;
}

// Structure de grille.
struct Grid { size: (u32,u32,u32) }

impl Topology for Grid {
  fn neighbors(&self, i: u32) -> impl Iterator<Item=u32> {
    // Ici se trouve l'implémentation de
    // `neighbors` pour la structure de grille.
  }
}

fn fiduccia_mattheyses<T>(partition: &mut [PartId],
                          adjacency: T)
where
  T: Topology,
{
  // `Topology` donne accès à `T::neighbors()`.
  for m in adjacency.neighbors(0) {
    // Itération sur les voisines de la première maille.
  }
  // ...
}

// Application sur une grille de taille 3 par 3.
let mut partition = [0; 9];
let grid = Grid { size: (2,2,2) };
fiduccia_mattheyses(partition, grid);
```

La plupart des types sont `Send` et `Sync`, car leurs objets ne peuvent être modifiés que par une référence exclusive, mais certains types ne le sont pas, et ces traits permettent de s'assurer qu'ils ne sont pas utilisés comme arguments de fonctions parallèles. Ces traits sont par exemple beaucoup

utilisés dans la bibliothèque de parallélisation automatique Rayon [31], que Coupe utilise pour implémenter des algorithmes parallèles.

Il est aussi possible de définir ses propres traits. Comme illustré par le Listing 7.2, où le trait `Topology` représente une relation de voisinage. Ce trait est implémenté sur la structure de données de grille `Grid`, ce qui permet d'utiliser `Grid` dans la fonction suivante, `fiduccia_mattheyses`.

Par ailleurs, même si Rust permet d'éviter de nombreux bogues mémoire, il ne garanti rien sur la validité de l'implémentation des algorithmes. Pour cela, nous utilisons Proptest [66], une bibliothèque de tests probabilistes. Cette bibliothèque génère des cas de tests automatiquement pour de nombreux types de données, et ainsi permet de tester les implémentations des algorithmes sans à écrire chaque cas test à la main. À la rencontre d'un mauvais résultat, ou d'un crash, Proptest réduit les entrées pathologiques vers un cas minimal pour faciliter la correction du bogue.

Nous pouvons donc, implémenter avec confiance les algorithmes pour les différents problèmes de partitionnement de maillage. Le langage permet à l'outil de facilement accepter des entrées sous plusieurs formes, grâce aux traits, et ainsi permet d'être versatile. Cependant, du code écrit en Rust n'est pas trivialement utilisable depuis les autres langages, et donc depuis les codes de calcul existants. Dans la prochaine section, nous allons développer une interface entre Coupe et les codes de calcul.

7.2 . Intégration de Coupe dans les codes de calculs

Les codes de calcul existants se reposent généralement sur des logiciels de simulation écrits dans des langages de programmation comme le C, le C++ ou le Fortran. Ces codes ne peuvent donc pas directement utiliser Coupe, qui écrit en Rust, car de nombreux aspects de Rust ne sont ni stables entre les versions du compilateur, ni spécifiés : l'encodage des noms de fonctions dans les binaires exécutables, la convention d'appel, l'agencement des membres des structures, etc.

Afin de permettre l'utilisation de Coupe par des logiciels écrits dans d'autres langages de programmation, nous exposons une interface C, dont les noms de fonction ne sont pas transformés, dont la convention d'appel est la même que celle du compilateur C de l'utilisateur, dont l'agencement des membres des structures suit celui du standard C, etc. Cette interface permet ainsi à Coupe d'être accessible depuis le C, mais aussi depuis de nombreux autres langages qui supportent ce standard, comme le C++ (avec `extern "C"`), Python (avec `ctypes`), Go (avec Cgo) et Java (avec *Java Native Interface*).

Il existe plusieurs façons de concevoir cette interface C. La première,

et la plus simple pour les utilisateurs, est de dupliquer l'interface d'un outil de partitionnement existant. C'est ce que nous avons fait avec Metis, qui expose peu de fonctions. Par exemple, Metis expose la fonction `METIS_PartGraphKway`, qui, étant donné un certain nombre de paramètres, partitionne un graphe en plusieurs parties. Nous pouvons exposer la même fonction, avec les mêmes arguments, mais en utilisant les algorithmes implémentés dans Coupe. De cette façon, les codes de calcul peuvent fonctionner avec Coupe sans aucune modification de leur part. Cependant, cette interface ne leur permet pas de chaîner des algorithmes.

La deuxième façon de procéder est de concevoir une interface modulaire, qui permet aux codes de calcul de chaîner les algorithmes qu'ils souhaitent. Pour cela, l'interface C de Coupe expose chaque algorithme sous forme de fonction qui met à jour une partition. Le Listing 7.3 montre l'exemple de la bissection récursive de coordonnées (RCB) [8] :

Listing 7.3 Structure générale des signatures de fonctions pour appeler les algorithmes.

```
1 enum coupe_err { COUPE_OK, COUPE_OUT_OF_MEMORY, /* .. */ };
2 typedef size_t coupe_id; // entier positif
3 enum coupe_err
4   coupe_rcb(coupe_id *partition,
5             /* arguments pour RCB... */);
```

Ce morceau de signature de fonction est commun aux outils de partitionnement : `partition` pointe vers un tableau d'indices sur lequel est écrit le résultat du partitionnement. Ce tableau est de taille égale au nombre de mailles, et le k-ième élément est l'indice de la partie à laquelle la k-ième maille appartient. Par exemple, pour un maillage à 5 mailles numérotées de 0 à 4 (inclus), la partition $\{\{0,2\},\{1,4\},\{3\}\}$ est encodée comme `[0,1,0,2,1]`. Cela permet d'éviter de gérer des tableaux multidimensionnels, qui peuvent prendre plus de place mémoire.

Les paramètres qui suivent sont ceux nécessaires à l'exécution de RCB. Ces paramètres peuvent représenter un maillage, mais RCB travaille avant tout sur un ensemble de points pondérés. Bien qu'il serait possible d'accepter directement des maillages pour chaque algorithme, cela pose plusieurs problèmes :

- La conversion du maillage vers les données spécifiquement utilisées par l'algorithme, bien qu'en $O(n)$ dans le cas géométrique et suffisamment parallélisable, elle induit des coûts de calcul et de mémoire

non négligeables. Il convient de ne pas les répéter, que ce soit lors d'un repartitionnement ou simplement par l'utilisation de plusieurs algorithmes du même type dans une chaîne d'algorithmes ;
- Il existe de nombreuses représentations de maillages, certains étant non-structurés, d'autres étant de simples grilles. Accepter tout type de maillages en entrée pour chaque type d'algorithme augmente la surface de l'interface et les coûts de maintenance ;
- Les logiciels de simulation de maillage offrent déjà des structures de données de maillages ainsi que des fonctionnalités associées (entrée/sortie, raffinement, ...). Répéter ces fonctionnalités ferait doublon non seulement dans le code de simulation même, mais aussi de part des connaissances à avoir sur les différentes API pour maintenir l'intégration de Coupe dans les logiciels de simulation.

C'est pourquoi les fonctions qui exécutent chaque algorithme acceptent en entrée les types de données directement utilisés par ces algorithmes : des nuages de points pondérés pour les algorithmes géométriques, des graphes et hypergraphes pour les algorithmes topologiques, des ensembles de poids pour les algorithmes de partitionnement de nombres.

On obtient donc une signature de cette forme :

Listing 7.4 Ajout des arguments référent les structures de données d'entrée pour RCB.

```
1 enum coupe_err { COUPE_ERR_OK, COUPE_ERR_ALLOC, /* .. */ };
2 typedef size_t coupe_id; // entier positif
3 enum coupe_err
4   coupe_rcb(coupe_id *partition,
5             void *points, void *weights);
```

Il faut maintenant préciser le type des paramètres `points` et `weights`. Même si la fonction n'accepte pas des maillages, plusieurs types de données sont possibles :

- La façon dont ces points et poids sont stockés. Ils peuvent être placés dans un tableau dense, éparse, ou dans des structures de données plus complexes comme des *octrees*, souvent utilisées lors du raffinement de maillage [67], [68] ;
- Les éléments mêmes peuvent être ou bien des entiers, ou bien des flottants.

Notons que Scotch [3] et Metis [1] n'acceptent que des tableaux denses, et les types de données sont restreints à des entiers de 32 ou 64

bits, selon la façon dont ils ont été compilés. Les charges de calcul doivent être encodées sur des entiers de 31 ou 63 bits. Les données doivent être stockées dans des tableaux. Un utilisateur ayant des données incompatibles doit alors les convertir. Typiquement, cela se fait par une normalisation des valeurs d'entrée en agrandissant ou en réduisant l'échelle de $[\min_W w; \max_W w]$ à $[1; U]$, avec un U grand, mais pas trop pour éviter les « *integer overflows* » lors du partitionnement. Ceci demande non seulement de l'attention supplémentaire de la part du développeur du code de calcul, mais aussi une étape en plus qui prend du temps de calcul et de la place en mémoire. Zoltan [2], dans sa première version, fonctionne exclusivement à l'aide de *callbacks* pour supporter différentes structures de données. Cette approche rend les cas simples, comme l'utilisation de tableaux, plus verbeux.

Ce problème de manque de généricité en C est aussi présent dans d'autres API comme celle de MPI [22]. Un grand nombre de fonctions MPI permet d'envoyer et de recevoir des tableaux de différents types (tableaux d'entiers, tableaux de booléens, ...). MPI permet de spécifier en grand détail les éléments du tableau à considérer (*stride*, *displacement*). Les fonctions d'envoi et de réception acceptent donc un argument supplémentaire `MPI_Datatype datatype`, qui contient le type des données, ainsi que des constantes comme `MPI_INT` et `MPI_DOUBLE` pour prédéfinir les cas les plus simples.

Pour Coupe, nous choisissons d'exposer un type opaque `coupe_data`, qui représente les données de différentes façons :

Listing 7.5 Définition des constructeurs pour les structures de données d'entrée de Coupe.

```
typedef struct {} coupe_data;
typedef
  enum { COUPE_INT, COUPE_DOUBLE, /* ... */ } coupe_type;

coupe_data *coupe_data_array(
        size_t n, coupe_type type, const void *data);
coupe_data *coupe_data_callback(
        const void *data_structure,
        size_t n,
        coupe_type type,
        const void *(*get_element)(const void *, size_t));
coupe_data *coupe_data_constant(
        size_t n, coupe_type type, const void *data);
```

Nous pouvons ainsi créer un `coupe_data` à partir :

- d'un tableau de taille `size_t n` pointé par `void *data` ;
- d'un *callback* `get_element` qui va chercher l'élément pour un indice donné dans la structure de données `void *data_structure` ;
- d'une constante, par exemple dans le cas où les points ne sont pas pondérés.

Chacune de ces fonctions accepte un type de données (`COUPE_INT`, `COUPE_DOUBLE`, ...) représentant le type des éléments, permettant à Coupe d'interpréter les valeurs pointées par les `const void *`.

La signature finale des fonctions d'algorithmes, après y avoir ajouté les options spécifiques à RCB, notamment le nombre d'itérations `niter` et la tolérance de déséquilibre `tolerance`, est de la forme suivante :

Listing 7.6 Définition complète de la signature de fonction de RCB.

```
1 enum coupe_err { COUPE_ERR_OK, COUPE_ERR_ALLOC, /* .. */ };
2 typedef size_t coupe_id; // entier positif
3 typedef struct {} coupe_data; // défini plus haut
4 enum coupe_err
5   coupe_rcb(coupe_id *partition,
6             coupe_data *points, coupe_data *weights,
7             size_t niter, double tolerance);
```

De la même manière, les structures de graphes sont encodées par un type opaque `coupe_adjncy` qu'il est possible de construire de différentes manières.

Un exemple complet d'utilisation de l'interface C de Coupe est donné dans le Listing 7.7. Dans cet exemple, l'utilisateur définit un ensemble de 4 points (lignes 8 à 11) ainsi qu'un graphe associé (lignes 19 à 23), puis appelle l'algorithme de partitionnement géométrique RCB (lignes 27 à 31) suivit de l'algorithme de partitionnement topologique Fiduccia-Mattheyses (lignes 34 à 36). Fiduccia-Mattheyses accepte plus d'arguments pour paramétrer plusieurs aspects de l'algorithme, qui ont été omis dans ce listing.

Listing 7.7 Exemple d'utilisation de l'interface C de coupe

```c
#include <coupe.h>

int main(void) {
  // L'association maille -> partie
  coupe_id partition[4];

  // Le barycentre des mailles
  double center_array[4][2] = {{0.0,0.0}, {1.0,0.0},
                               {1.0,1.0}, {0.0,1.0}};
  coupe_data *centers =
    coupe_data_array(4, COUPE_DOUBLE, center_array);

  // Le poids des mailles est constant et égal à 1
  int one = 1;
  coupe_data *weights =
    coupe_data_constant(4, COUPE_INT, &one);

  // La matrice d'adjacence au format CSR
  size_t xadj[5] = {0, 2, 4, 6, 8};
  size_t adjncy[8] = {1, 3, 0, 2, 1, 3, 0, 2};
  int64_t edge_weights[8] = {1, 2, 1, 2, 2, 3, 2, 3};
  coupe_adjncy *adjacency = coupe_adjncy_csr(
    4, xadj, adjncy, COUPE_INT64, edge_weights);

  // Partitionnement géométrique à l'aide de RCB
  // On partitionne en 2 parties
  size_t iter = 1;
  double tolerance = 0.05;
  enum coupe_err err = coupe_rcb(
    partition, 2, centers, weights, iter, tolerance);
  assert(err == COUPE_ERR_OK);

  // Raffinement à l'aide de FM
  err = coupe_fiduccia_mattheyses(
    partition, adjacency, weights, /* ... */);
  assert(err == COUPE_ERR_OK);

  coupe_data_free(points);
  coupe_data_free(weights);
  return 0;
}
```

Le graphe est encodé sous forme de matrice d'adjacence au format CSR, ou *Compressed Sparse Row* [69]. CSR est un format pour représenter les matrices creuses, qui contiennent beaucoup de zéros. Seules les valeurs non nulles y sont représentées, dans `edge_weights`. La valeur `edge_weight[j]` est à la colonne `adjncy[j]`, et à la ligne `i` tel que `j` est compris entre `xadj[i]` et `xadj[i+1]`. Autrement dit, `xadj[i]` contient l'indice de départ dans `adjncy` de chaque ligne, et `adjncy[xadj[i]..xadj[i+1]]` donne les colonnes qui ont des valeurs non nulles, stockées dans `edge_weights[xadj[i]..xadj[i+1]]`.

L'encodage CSR dans le Listing 7.7 (lignes 19 à 21) représente donc la matrice d'adjacence suivante :

$$\begin{pmatrix} 0 & 1 & 0 & 2 \\ 1 & 0 & 2 & 0 \\ 0 & 2 & 0 & 3 \\ 2 & 0 & 3 & 0 \end{pmatrix}$$

En conclusion, Coupe expose deux interfaces C, afin d'être utilisé par les codes de calculs, qui sont écrits dans divers langages de programmation. L'une de ces interfaces est une copie de celle de Metis. Cette interface a l'avantage de ne requérir aucune modification du code de calcul, mais ne permet pas de chaîner différents algorithmes. L'autre interface est exclusive à Coupe, et permet le chaînage d'algorithmes.

Pour le développement même des algorithmes de partitionnement, ces interfaces sont trop bas niveau pour être pratiques. C'est pourquoi, dans la section suivante, nous développons en plus un ensemble d'outils en ligne de commande, afin de facilement expérimenter sur les chaînes d'algorithmes.

7.3 . Développement d'une boite à outils d'expérimentation sur les chaînes d'algorithmes

En parallèle du développement de Coupe, nous avons développé une suite d'outils en ligne de commande. Ces outils se branchent à l'interface Rust de Coupe et exposent à leur tour une interface en ligne de commande, plus simple, pour expérimenter avec les différents algorithmes développés dans Coupe.

Ces outils couvrent plusieurs cas d'utilisation, mais nous servent notamment pour le développement de nouveaux algorithmes de partitionnement de maillage. Ils permettent notamment de calculer différentes métriques sur les partitions obtenues par le chaînage de plusieurs algorithmes. Ils permettent aussi de mesurer la vitesse d'exécution d'une chaîne. Par ailleurs, les données d'entrée peuvent être changées, notam-

ment les charges de calcul associées aux mailles et les différents paramètres d'entrée de chaque algorithme.

Présentons d'abord la commande « mesh-part », qui fait tourner une chaîne d'algorithme sur un maillage et donne la partition résultat en sortie standard. Son utilisation est la suivante :

Listing 7.8 Exemple d'utilisation de mesh-part

```
mesh-part --mesh MESH.mesh --weight WEIGHT_FILE \
          --algorithm ALGO1 --algorithm ALGO2 ... \
          >PART_FILE
```

Les paramètres `ALGO1` et `ALGO2`, etc. sont à remplacer par des noms d'algorithmes tel `rcb,3` pour « Recursive Coordinate Bisection, 3 itérations ». Le fichier de poids `WEIGHT_FILE` contient la charge associée à chaque maille. Il est important de tester les chaînes d'algorithmes avec différentes distributions de charge de calcul. La distribution de charge influence, par exemple, le nombre d'opérations effectuées par RCB lors de la recherche du point pivot. La commande à utiliser pour générer des fichiers de charges de calcul est « weight-gen », qui est invoqué de cette manière :

Listing 7.9 Exemple d'utilisation de weight-gen

```
weight-gen --distribution DISTRIBUTION \
           <MESH.vtk \
           >WEIGHT_FILE
```

Cette commande prend en entrée un maillage, ainsi qu'une distribution de charges de calcul, et donne en sortie un fichier qui stocke les charges de calcul associées à chaque maille. Un exemple de distribution de charge, que nous avons utilisé dans le Chapitre 4 à la page 57, est une distribution dite linéaire, où les charges varient linéairement entre deux valeurs données selon un axe donné. Il est aussi possible de générer des distributions où les charges sont constantes, et d'autres où les charges forment un pic d'une certaine taille en un certain point.

Avec weight-gen et mesh-part, nous pouvons ainsi appliquer des chaînes d'algorithmes sur de nombreux maillages, et ce en changeant de nombreux paramètres. Pour en analyser les résultats, nous développons la commande « part-info » qui, pour une partition et un fichier de charges

donnés, donne le déséquilibre de charge, la coupe et la $\lambda - 1$ *cut* de la partition.

Par ailleurs, pour mesurer la vitesse d'exécution des algorithmes de partitionnement, nous développons la commande « part-bench », qui fonctionne de manière similaire à mesh-part, mais qui mesure le temps d'exécution au lieu de retourner la partition. Cette commande s'appuie sur « Criterion.rs » [65], une bibliothèque de *benchmarking*. En parallèle, deux commandes « mesh-refine » et « mesh-dup » permettent de créer des maillages arbitrairement grands, pour reproduire les conditions trouvées dans les codes de calcul à grande échelle. La première subdivise les mailles, alors que la seconde duplique le maillage.

D'autres commandes ont aussi été développées pour visualiser les partitions et les distributions de poids. Les illustrations de maillage dans ce document ont été créées en partie à l'aide de ces commandes.

Conclusion

Dans ce chapitre, nous avons élaboré l'architecture et les choix techniques de notre nouvel outil de partitionnement dédié au partitionnement de maillages et à l'équilibrage de charge de simulations numériques. Cet outil est écrit en Rust, de façon à garantir l'absence de bogues de corruption mémoire. Comme le Rust n'est pas facilement utilisable depuis les autres langages, nous avons exposé deux interfaces en C : l'une duplique les fonctionnalités de Metis pour être utilisé sans modification de la part du code de calcul, l'autre est plus modulaire et permet de chaîner les différents algorithmes. Afin d'expérimenter, en tant que développeur d'outil de partitionnement, sur les différents algorithmes, nous avons développé une suite d'outil en ligne de commande.

Le chapitre suivant présente, à l'aide de la suite d'outils d'expérimentation, une analyse du chaînage d'algorithmes tel qu'implémenté par Coupe, et compare les résultats avec Scotch [3] et Metis [1].

8 - Analyse critique et expérimentations

Dans ce chapitre, nous cherchons à mesurer la qualité des résultats donnés par les algorithmes implémentés dans coupe, ainsi que leur vitesse d'exécution.

Pour la comparaison à l'état de l'art, nous choisissons de se comparer à Metis et Scotch. Ces deux partitionneurs sont les versions en mémoire partagée de ParMetis et PT-Scotch respectivement. Nous les avons choisis pour les raisons suivantes :

- Ils fonctionnent exclusivement en mémoire partagée, comme coupe ;
- ParMetis et PT-Scotch ont été éprouvés par l'industrie ;
- Ils sont écrits en C, donc sont facilement utilisables depuis Rust.

Nous intégrons donc ces deux partitionneurs à `mesh-part` afin qu'ils soient chainables avec les autres algorithmes.

8.1 . Mesures de qualité du partitionnement

Dans cette section, nous nous concentrons sur les tests de qualité des partitions obtenues par les différents algorithmes, ainsi que Metis et Scotch. Comme vu dans le Chapitre 2, il y a plusieurs objectifs optimisables, selon le contexte du partitionnement. De ce fait, il y a différentes façons de mesurer la qualité d'une partition. Nous nous intéressons au déséquilibre et à la coupe.

Les algorithmes et outils de partitionnement qui font partie des tests sont les suivants :

- Scotch avec les paramètres par défaut et une tolérance sur le déséquilibre de charge de 1%,
- Metis, dont les deux procédures "kway" (`METIS_PartGraphKway`) et "bissection recursive" (`METIS_PartGraphRecursive`), avec la même tolérance de déséquilibre de charge de 1%.
- La courbe de Hilbert d'ordre 12,
- RCB (*Recursive Coordinate Bisection*),
- Karmarkar-Karp,
- Le partitionnement de nombres glouton.

Les mesures sont faites sur les algorithmes ci-dessus, seuls, ainsi que suivis de VNBest puis Arcswap.

Les mesures sont réalisées sur plusieurs maillages :

- `engine-2.5-270k.meshb` : maillage 3D de la surface d'un moteur à combustion, composé de 269 640 triangles,

- `heart-2.5-287k.meshb` : maillage 3D de la surface d'un cœur humain, composé de 287 080 triangles,
- `hole-2-197k.meshb` : maillage 2D avec un trou, composé de 196 608 triangles et quadrilatères,
- `i08u_m8-3-2500k.meshb` : maillage 3D d'une hélice à trois ailes, composé de 2 499 668 tétraèdres,

Les expériences sont conduites avec les distributions de poids suivantes :

- `constant,1` : tous les poids sont égaux à 1,
- `linear,x,0,1` : les poids sont fonction affine de l'abscisse du centre de la maille, et varient de 0 à 1.
- `spike,1,0,0` : les poids forment un pic de taille 1 à l'origine.

8.1.1 . Pré-traitement des poids pour Metis et Scotch

Metis et Scotch n'acceptent que des poids entiers (`int` ou `long`) strictement positifs. Or dans notre cas, les poids sont des nombres flottants (`double`) qui varient entre 0 et 1. Il est alors nécessaire de les ajuster, de façon à les rendre acceptables par ces partitionneurs.

Pour cela, nous transformons les poids x de leur plage initiale $[min; max]$ vers la plage $[1; M]$, où M est choisi de telle sorte que la somme des poids entiers soit représentable par un entier. Plus précisément, nous appliquons la fonction affine f suivante :

$$f(w) = \frac{(w - min) \cdot M + max - w}{max - min}.$$

Nous remarquons bien que $f(min) = 1$ et $f(max) = M$.

La valeur de M est choisie de telle sorte que $\sum_w f(w) = \texttt{INTMAX}$, où `INTMAX` est l'élément le plus grand représentable par le type d'entier accepté par Metis ou Scotch. Nous obtenons donc :

$$M = \frac{(max - min) \cdot \texttt{INTMAX} - \sum_w (max - w)}{\sum_w (w - min)}.$$

Dans le cas où les poids sont tous égaux ($min = max$, le cas de la distribution `constant,1`), nous ne donnons pas de poids à Scotch et Metis, qui assument donc que les sommets du graphe ont des poids égaux.

Pour éviter les *integer overflows*, toutes les opérations de prétraitement se font avec des nombres flottants. Les données sont converties en entier à la fin seulement. Il faut alors calculer la somme des poids w à l'aide d'une somme compensée [70] pour ne pas accumuler d'erreurs lors des opérations avec max, min et `INTMAX`.

8.1.2 . Commandes et résultats

Les expériences sont effectuées à l'aide des outils en ligne de commande conçus dans la Section 7.3. Prenons par exemple le maillage `heart-2.5-287k.meshb` et la distribution de poids linéaire. Nous générons d'abord cette distribution de poids pour le maillage à l'aide de `weight-gen`. Pour cela, la commande à entrer est la suivante :

Listing 8.1 Commande pour générer les poids de chaque maille pour un des maillages et l'une des distributions de poids de l'expérience

```
weight-gen --distribution linear,x,0,1 \
           heart-2.5-287k.meshb \
           weights.dat
```

La commande du Listing 8.1 crée un fichier `weights.dat`, qui contient la charge de chaque maille du maillage `heart-2.5-287k.meshb`, selon la distribution `linear,x,0,1`.

Ensuite, nous exécutons une des chaînes d'algorithmes sur ce maillage et cette distribution de poids. La commande suivante partitionne le maillage `heart-2.5-287k.meshb` à l'aide d'une courbe de Hilbert d'ordre 16 en 256 parties, puis optimise son équilibre de charge à l'aide de VNBest, et enfin optimise la coupe à l'aide d'Arcswap, sous une tolérance de déséquilibre de 1%. La partition résultant de cette chaîne d'algorithmes est sauvegardée dans un fichier `heart-hva.part` :

Listing 8.2 Commande pour partitionner un maillage selon les poids générés par la commande précédente

```
mesh-part --mesh heart-2.5-287k.meshb \
          --weights weights.dat \
          --algorithm hilbert,256,16 \
          --algorithm vn-best \
          --algorithm arcswap,0.01 \
          heart-hva.part
```

Nous avons limité le temps d'exécution de `mesh-part` à trente secondes, afin de pouvoir tester un grand nombre de paramètres.

Pour mesurer l'équilibre de charge de la partition stockée dans le fichier `heart-hva.part`, nous utilisons la commande `part-info`. Cette commande mesure le déséquilibre de charge et la coupe d'une partition don-

née, pour des poids donnés. La commande que nous entrons est la suivante :

Listing 8.3 Commande pour analyser la partition produite par la commande précédente

```
part-info --mesh heart-2.5-287k.meshb \
          --weights weights.dat \
          --part heart-hva.part
```

Les résultats des expériences réalisées sont disponibles sur les Figures 8.1 à 8.3, sous forme de graphes où l'abscisse représente la coupe (en unités arbitraires) et les ordonnées le déséquilibre de charge (en pourcentages). Chaque point représente un résultat obtenu par une chaîne d'algorithme.

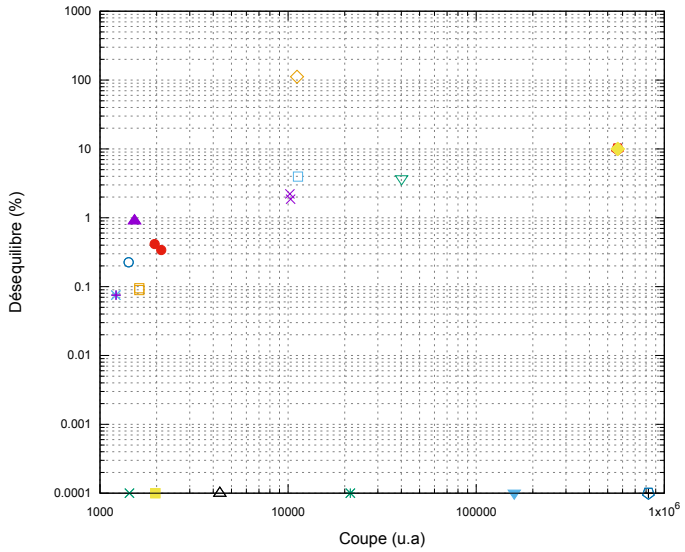
Nous remarquons d'abord, comme vu dans le Chapitre 4 (page 55) que l'application VNBest et Arcswap déplace les points vers le bas et vers la gauche respectivement.

Ensuite, nous remarquons que l'heuristique gloutonne pour le partitionnement de nombre, ainsi que l'heuristique de Karmarkar et Karp trouvent souvent une partition avec un déséquilibre nul (affiché comme valant 0.0001 pour l'échelle logarithmique). Nous pensons que le grand nombre de mailles est ce qui permet à l'heuristique gloutonne de donner des solutions au déséquilibre minimal.

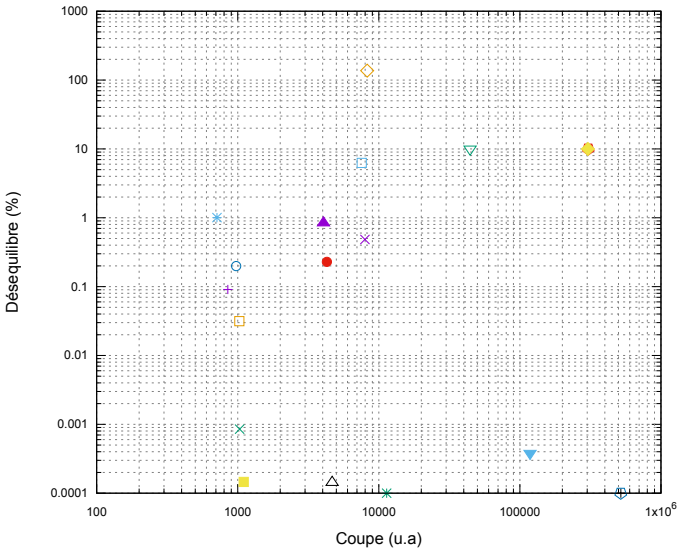
Dans la plupart des cas, les meilleures coupes sont obtenues à l'aide de MetisKway et Scotch. L'application de VNBest après ces deux outils permet de trouver le meilleur compromis entre déséquilibre et coupe. L'application d'Arcswap ne permet pas de trouver de meilleure partition par rapport à Scotch et Metis seuls, sûrement parce qu'Arcswap ne visite pas suffisamment l'espace des solutions. Ce problème est visible aussi sur les solutions données par Glouton et KarmarkarKarp suivis de Arcswap : Arcswap tombe trop facilement dans un minimum local où plus aucun déplacement ne peut faire diminuer la coupe. Metis et Scotch utilisent aussi des algorithmes d'optimisation locale, comme FM (cf. Section 3.3), mais leur contexte d'application est différent. Ces deux outils de partitionnement utilisent principalement la méthode dite de « multi-niveaux » [1], [71], [72], qui guide les algorithmes d'optimisation locale. Dans le multi-niveau, les algorithmes d'optimisation locale sont appliqués sur des versions grossières du graphe d'entrée. Alors, un mouvement fait par l'algorithme peut équivaloir de nombreux mouvements sur le graphe d'origine. Nous pensons que cela donne à l'algorithme une vision plus globale de l'espace des

solutions, et donc qu'ils tombent plus difficilement dans des minimum locaux.

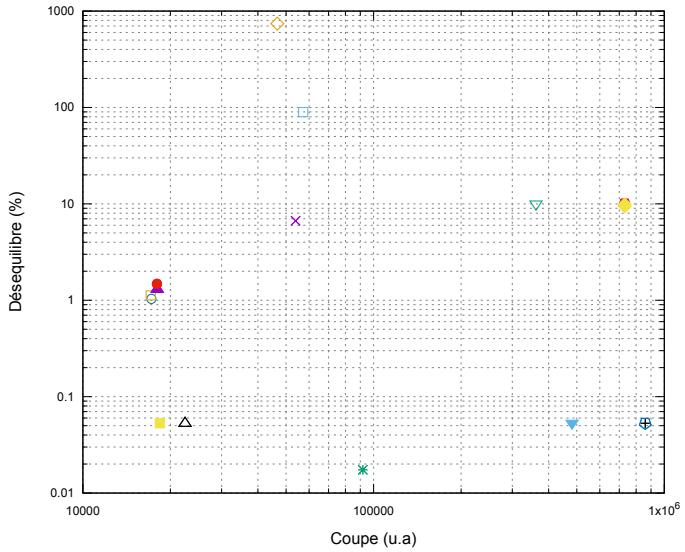
Parmi les algorithmes géométriques, RCB est celui qui donne les meilleures coupes, et dans un des cas (celui de la Figure 8.3b) une coupe meilleure que Scotch. Notons que dans ce cas, et ce pour l'ensemble des tests sur `i08u_m8-3-2500k`, Metis n'a pas su retourner de solution dans les trente secondes imparties.



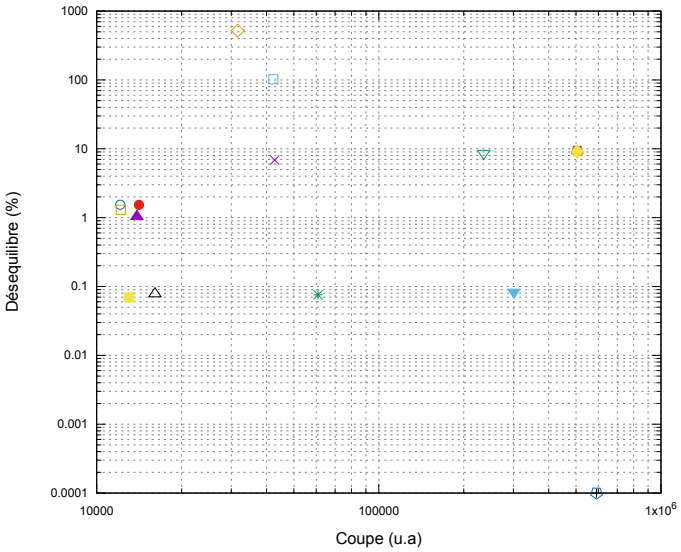
(a) 8 parties, distribution constante



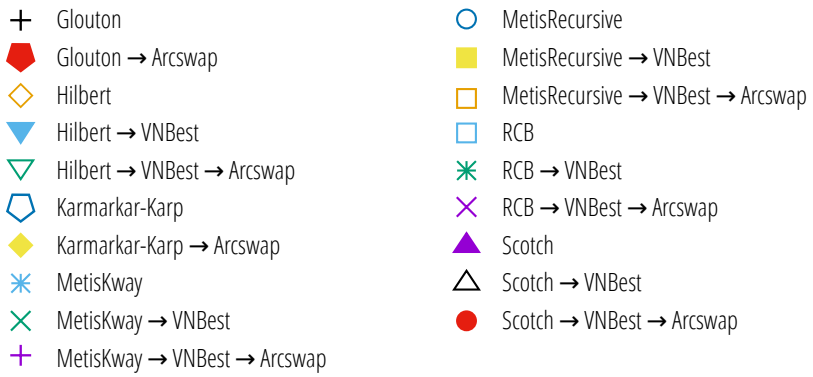
(b) 8 parties, distribution linéaire



(c) 256 parties, distribution constante

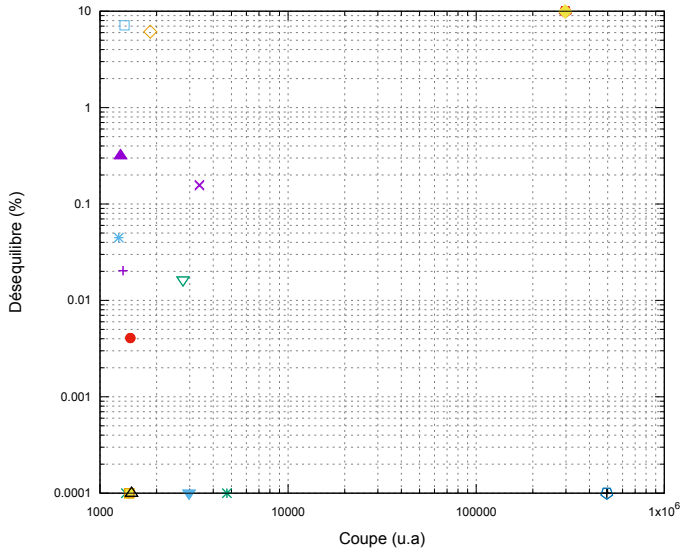


(d) 256 parties, distribution linéaire

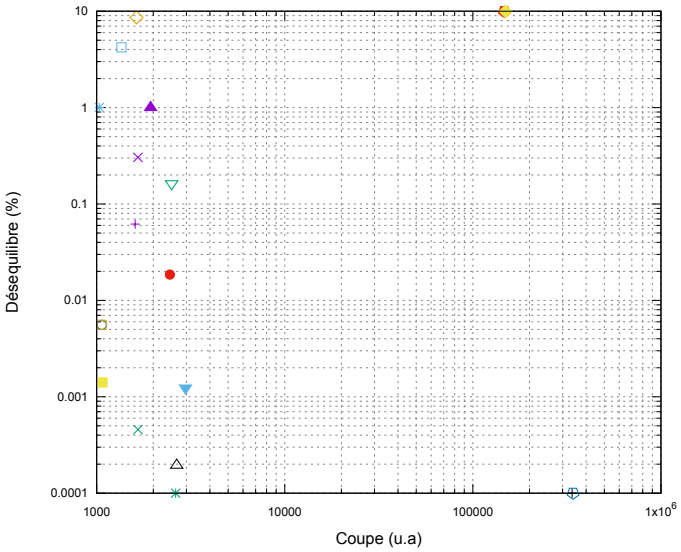


(e) Légende

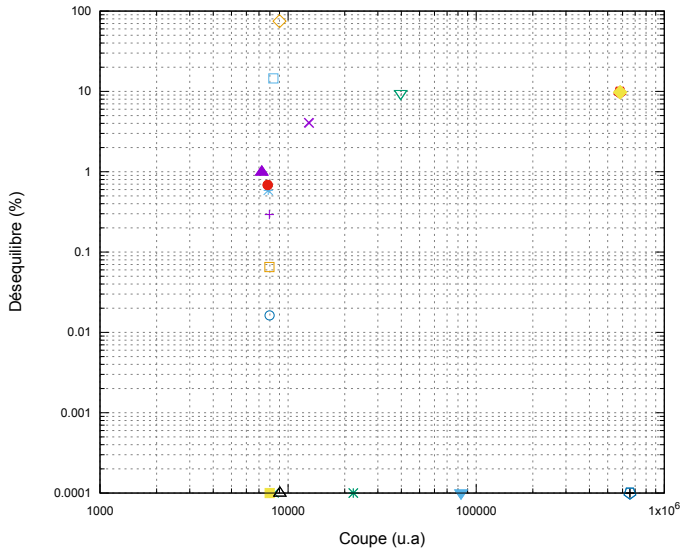
Figure 8.1 – Résultats d'expérience de qualité de partitionnement avec le maillage `heart-2.5-287k`. Les échelles utilisées sont logarithmiques. Les déséquilibres nuls ont été rehaussés à 0.0001% pour l'affichage avec échelle logarithmique.



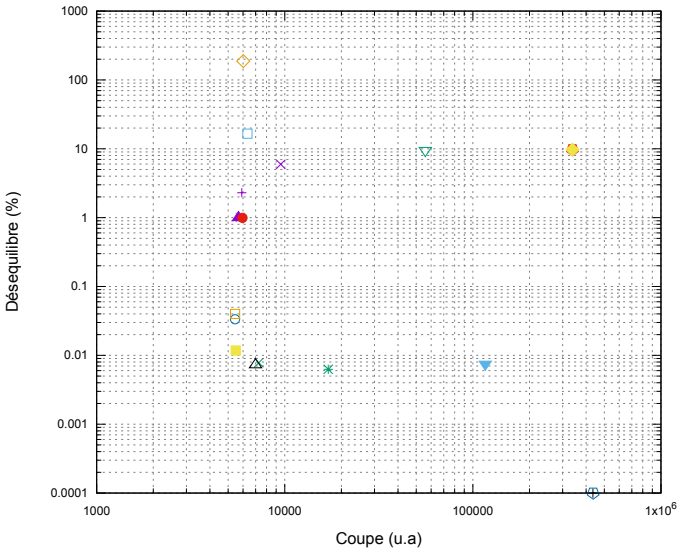
(a) 4 parties, distribution constante



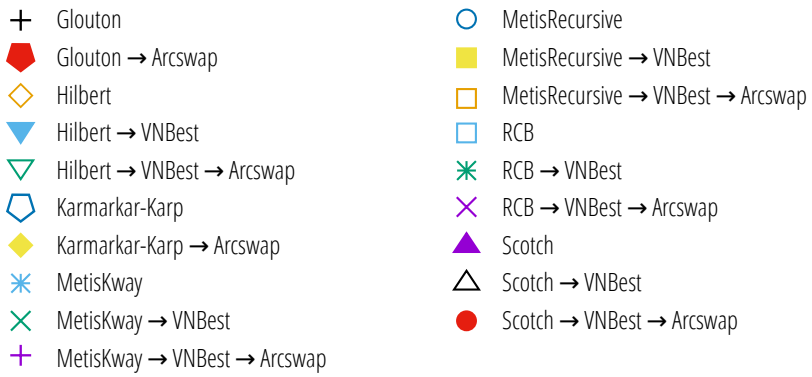
(b) 4 parties, distribution linéaire



(c) 32 parties, distribution constante

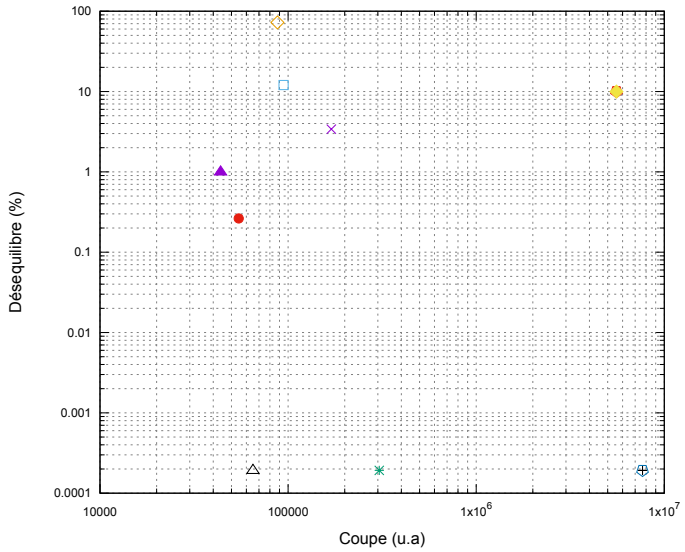


(d) 32 parties, distribution linéaire

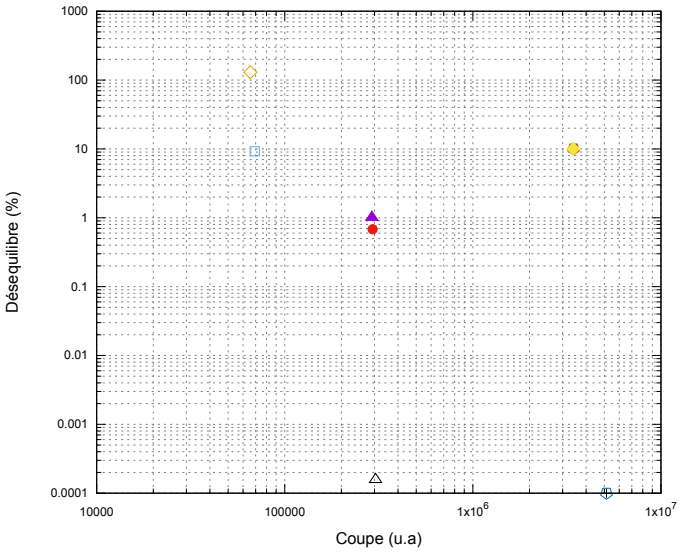


(e) Légende

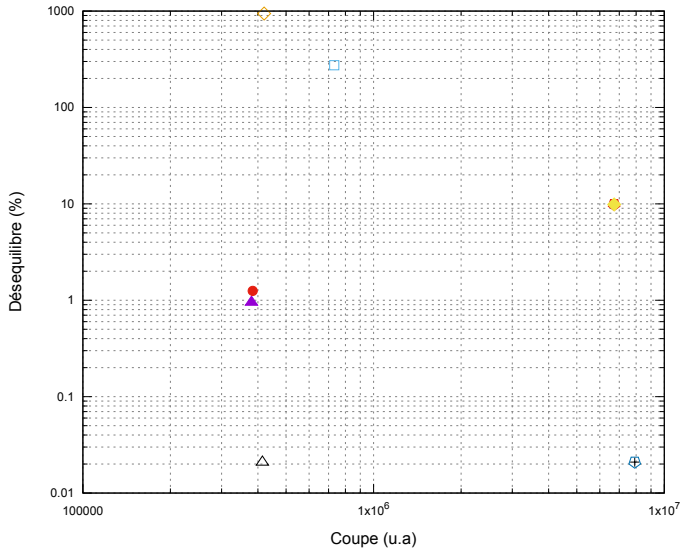
Figure 8.2 – Résultats d'expérience de qualité de partitionnement avec le maillage `hole-2-197k`. Les échelles utilisées sont logarithmiques. Les déséquilibres nuls ont été rehaussés à 0.0001% pour l'affichage avec échelle logarithmique.



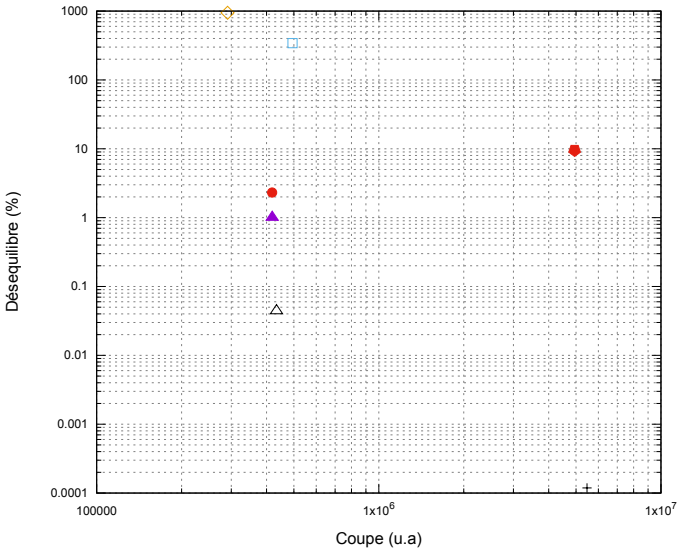
(a) 16 parties, distribution constante



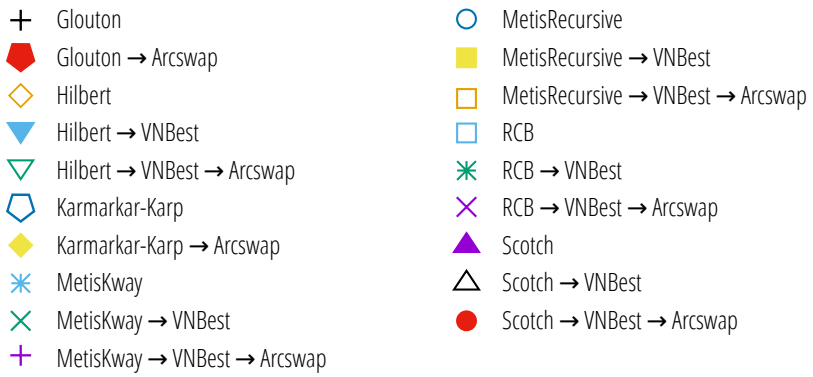
(b) 16 parties, distribution linéaire



(c) 1024 parties, distribution constante

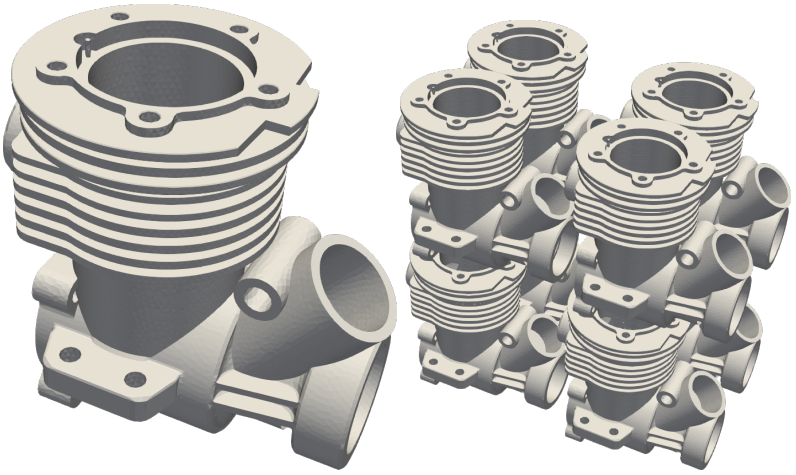


(d) 1024 parties, distribution linéaire



(e) Légende

Figure 8.3 – Résultats d'expérience de qualité de partitionnement avec le maillage `i08u_m8-3-2500k`. Les échelles utilisées sont logarithmiques. Les déséquilibres nuls ont été rehaussés à 0.0001% pour l'affichage avec échelle logarithmique.



(a) Maillage avant la duplication.

(b) Maillage dupliqué huit fois.

Figure 8.4 – Exemple de duplication du maillage. Le maillage `engine-2.5-270k.meshb` est dupliqué huit fois.

8.2 . Mesures de performance du partitionnement

Dans cette section, nous nous intéressons aux caractéristiques de performance des algorithmes implémentés dans Coupe. D'autres résultats de performance, spécifiques aux maillages cartésiens, ont été donnés dans le Chapitre 6. Ici, nous mesurons le temps d'exécution de certaines chaînes d'algorithmes, en fonction du nombre de threads et de la taille du maillage d'entrée. En effet, l'implémentation de certains algorithmes dans Coupe est parallèle, notamment RCB [8], les courbes de Hilbert [58], ainsi qu'Arcswap, une variante parallèle et non gloutonne de l'heuristique de Fiduccia et Mattheyses [37].

Pour un nombre de threads T donné, le maillage d'entrée est dupliqué T fois puis partitionné par la chaîne d'algorithmes. En utilisant les mêmes maillages, et distributions de poids que ceux de la Section 8.1, nous mesurons ici le temps d'exécution de la chaîne d'algorithme.

La façon dont le maillage est dupliqué est illustrée par la Figure 8.4. La duplication se fait de sorte à conserver les propriétés topologiques locales du maillage, comme le nombre de voisins de chaque maille, ou le nombre d'arêtes incidentes aux nœuds. Aucune topologie ne relie les doublons du maillage.

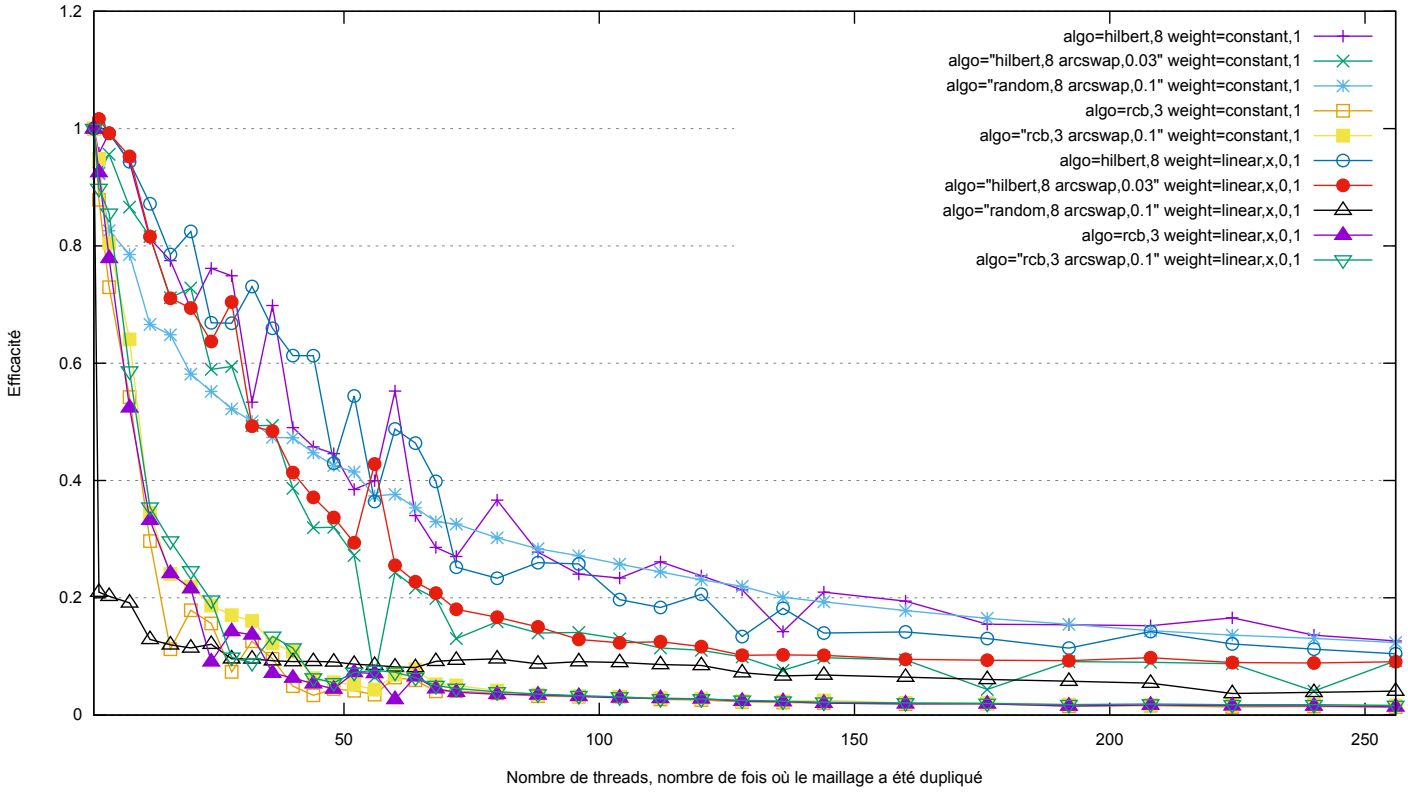
Pour effectuer les expériences, nous utilisons les commandes `mesh-dup`, qui duplique un maillage donné, et `part-bench`, qui mesure le temps d'exécution d'une chaîne d'algorithmes. Un exemple est montré sur le Listing 8.4 :

Listing 8.4 Lignes de commande pour mesurer le temps d'exécution d'une chaîne d'algorithmes dans le cadre de l'expérience

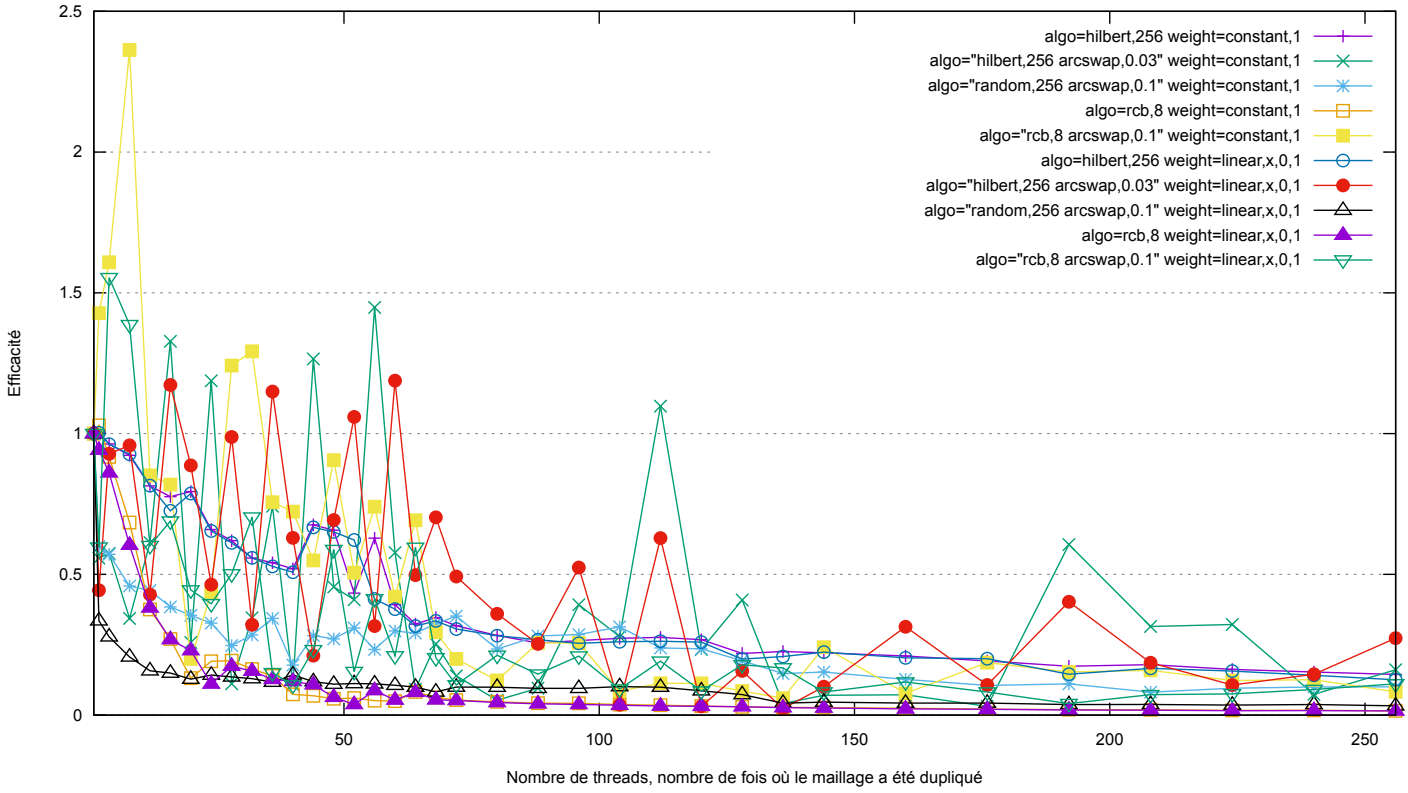
```
mesh-dup --times 100
         <heart-2.5-287k.meshb
         >heart-2.5-29m.meshb

part-bench --mesh heart-2.5-29m.meshb \
           --weights weights.dat \
           --algorithm hilbert,256,16 \
           --algorithm arcswap
```

Le Listing 8.4 montre les commandes pour dupliquer cent fois le maillage `heart-2.5-287k.meshb`, puis exécuter une courbe de Hilbert puis Arcswap sur le maillage dupliqué.

Les résultats sont présents sur les Figures 8.5 et 8.6, sous forme de graphe où l'abscisse représente le nombre de threads utilisés pour exécuter les chaînes d'algorithmes, et les ordonnées représente l'efficacité de la chaîne d'algorithmes.

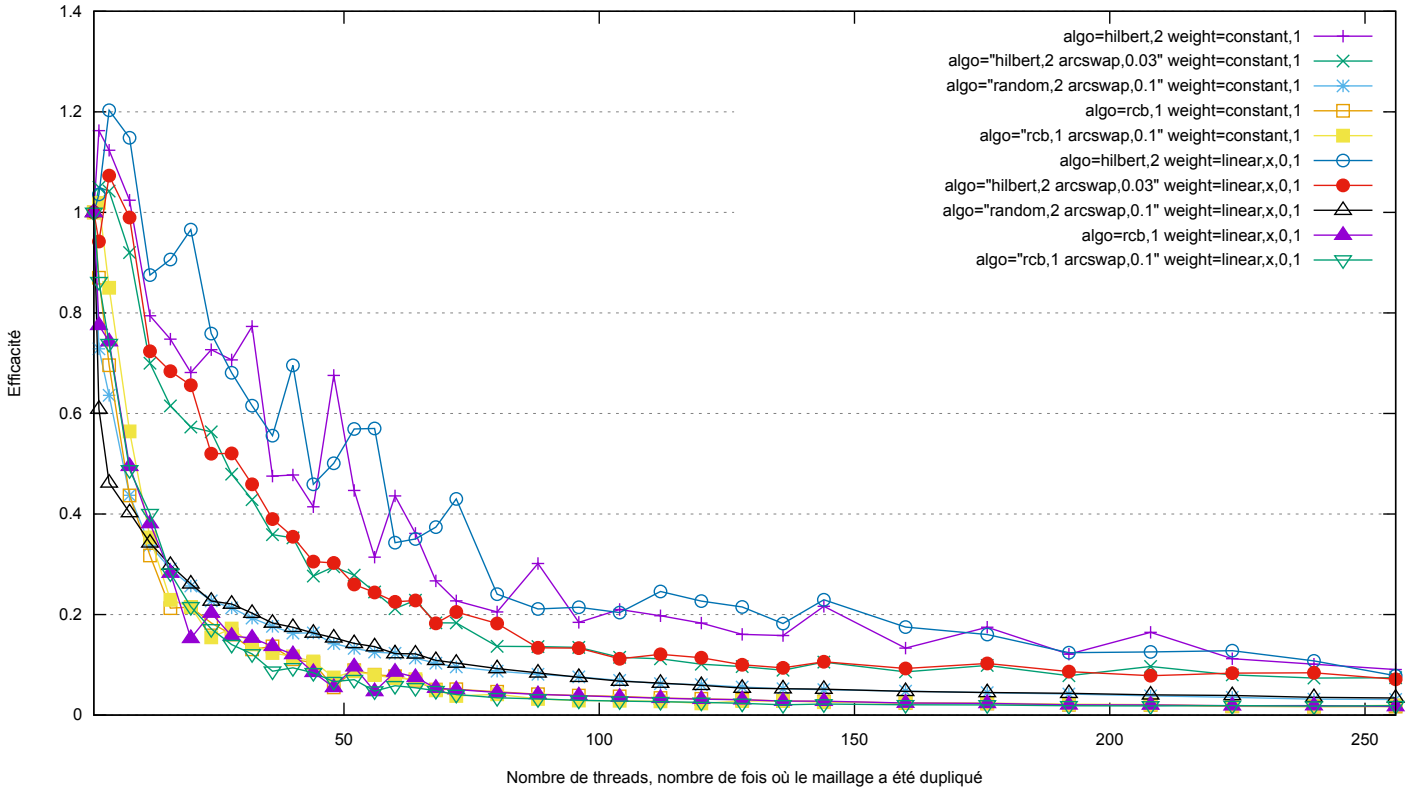


(a) 8 parties

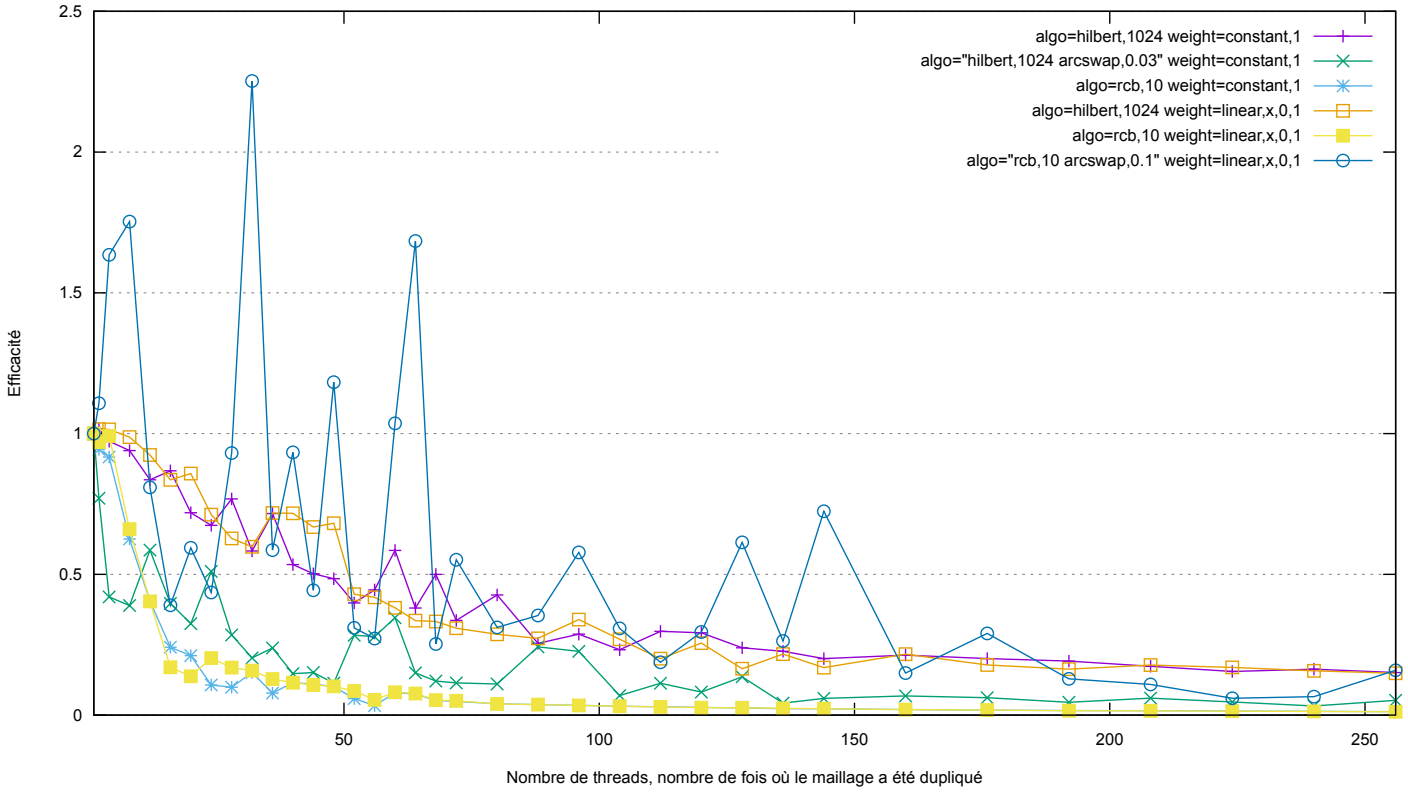


(b) 256 parties

Figure 8.5 – Résultats d'expérience de performance avec le maillage `engine-2.5-270k`.



(a) 2 parties



(b) 1024 parties

Figure 8.6 – Résultats d'expérience de performance avec le maillage `heart-2.5-287k`.

Nous remarquons qu'Arcswap introduit beaucoup de bruit dans les mesures. Parfois même, tellement de bruit que l'efficacité peut être supérieure à 1. Nous pensons que cela est dû au fait que, en fonction du nombre de fois où le maillage est dupliqué, RCB ou la courbe de Hilbert peut donner des partitions au déséquilibre trop élevé, et donc qu'Arcswap ne fait aucun mouvement. Alors, son temps d'exécution est plus court que pendant l'exécution sur un seul thread, où Arcswap a dû faire des mouvements. Aussi, nous pensons que la façon dont le maillage est dupliqué influe sur le nombre de mouvements même d'Arcswap. En effet, comme les doublons ne sont pas connectés entre eux, ils peuvent être placés dans des parties séparées sans aucun coût sur les communications. Cela influe sur les mouvements d'Arcswap car les mailles au bord des doublons ont donc peu d'intérêt à être déplacées dans une partie différentes de l'intérieur du doublon.

Enfin, l'efficacité de l'ensemble des chaînes tend rapidement vers zéro. Nous pensons que cela est dû au placement des données en mémoire. La bibliothèque que Coupe utilise pour implémenter les algorithmes parallèles, Rayon [31], n'offre pas de fonctionnalités pour placer précisément les données en mémoire, et n'ordonnance pas automatiquement les calculs sur les threads les plus proches des données. Sur une machine où la mémoire dispose d'un certain nombre de caches, il est préférable que chaque thread calcule en fonction de données qui lui sont physiquement proches, car les temps de latence mémoire sont beaucoup plus élevés que l'horloge du processeur. Pour palier à ce problème, une approche classique est une implémentation hiérarchique du parallélisme [73], où les threads sont regroupés avec ceux qui partagent un même niveau de cache. Cependant, cela augmenterait grandement la complexité du code de Coupe.

9 - Conclusion et perspectives

Cette thèse porte sur l'optimisation du temps d'exécution de codes de calcul utilisant des maillages et fonctionnant sur des machines à mémoire distribuée. Dans le Chapitre 1, nous avons décrit les codes de calcul qui nous intéressent. Ces codes fonctionnent par itérations, où à chaque itération il faut effectuer des calculs pour chaque maille. Lorsque ces calculs sont exécutés sur machines distribués, où de nombreuses unités de calcul ne partagent pas forcément la même mémoire, il faut répartir équitablement les calculs à effectuer tout en faisant en sorte de minimiser le nombre de transferts entre unités de calcul. Quand la simulation travaille sur un maillage, nous pouvons modéliser ce problème comme le partitionnement de maillage. Dans le Chapitre 2, nous avons décrit mathématiquement ce problème, ainsi que quatre sous-problèmes qui sont résolus en pratique lors du partitionnement de maillage. Dans le Chapitre 3, nous avons présenté des algorithmes communément utilisés pour résoudre chacun de ces quatre problèmes. Certains créent une partition de zéro, d'autres optimisent une partition existante.

En utilisant ces algorithmes nous avons, dans le Chapitre 4, montré l'intérêt de chaîner ces algorithmes, pour permettre à un code de calcul d'optimiser selon les métriques de son choix. Afin de compléter les problèmes de partitionnement solvables par le chaînage d'algorithmes nous avons, dans le Chapitre 5, étendu l'algorithme de partitionnement de nombre « VNBest ». Nous l'avons étendu de deux façon : la première est par la prise en charge de poids cible pour chaque partie, afin d'équilibrer la charge pour des codes de calcul où les unités sont hétérogènes. La deuxième est par l'extension au partitionnement en un nombre de parties arbitraire. Dans le Chapitre 6, nous avons spécifié l'implémentation de l'algorithme pour le partitionnement géométrique « RCB » à un type de maillages particulier, les maillages cartésiens. Les propriétés de ces maillages permettent à RCB de trouver des partitions en moins d'opérations, et en utilisant moins de mémoire.

Dans le Chapitre 7, nous avons conçu un nouvel outil de partitionnement de maillages, appelé « Coupe », qui s'appuie sur le chaînage d'algorithmes. Nous avons décrit toutes les problématiques liées à la conception de cet outil : notamment, l'attention à la robustesse et les formats de données d'entrée, et la conception d'interfaces pour l'utilisation par les codes de calcul. Nous avons agrémenté l'outil d'une suite de commandes permettant de facilement expérimenter sur les chaînes d'algorithmes.

Dans le Chapitre 8, nous avons utilisé cette suite de commandes pour mener des expérimentations sur les algorithmes implémentés dans

Coupe. Nous avons mesuré la qualité des partitions obtenues, par le déséquilibre de charge et la coupe, et nous avons mesuré les performances de certains algorithmes. Pour les expériences de qualité, nous nous sommes comparés aux versions séquentielles de Scotch et Metis. Les résultats ont montré que les algorithmes implémentés dans Coupe et les différentes chaînes testées ne permettent pas d'obtenir une meilleure coupe que celles obtenues par Scotch et Metis. Cependant, nous pouvons obtenir de meilleurs déséquilibres de charges à l'aide de VNBest – la plupart du temps, nuls. Pour les résultats de performance, nous n'avons pas de comparaison avec les outils existants, qui fonctionnent pour la plupart en mémoire distribué.

Plusieurs pistes d'exploration sont possibles pour étendre ce travail.

Bissections géométriques sur GPU Afin d'améliorer les performances de RCB, des courbes de Hilbert et autres bissections géométriques, l'utilisation de processeurs graphiques (GPU), par rapport à celle des processeurs généralistes (CPU) parait avantageuse. Ces bissections, comme vu dans la Section 3.2 et le Chapitre 6, effectuent principalement des sommes, possiblement cumulées, d'une grande quantité de nombres. Ce type de calcul est typiquement plus avantageux à faire sur GPU [74] pour deux raisons : premièrement, ces algorithmes sont exprimables de façon très parallèles – avec peu d'opérations séquentielles – et les GPU ont un nombre de threads très supérieur aux CPU. Deuxièmement, ce type de calcul opère sur un grand nombre de données mémoire, et les GPU ont généralement une meilleure bande passante mémoire que les CPU. Dans le cas où les données nécessaire à la bissection géométrique sont déjà présentes sur GPU, il est possible de gagner beaucoup de temps.

Meilleure optimisation de la coupe Comme les expériences l'ont montré dans le Chapitre 8, l'application directe d'un algorithme d'optimisation locale typé Fiduccia-Mattheyses et Kernighan-Lin ne permet pas de suffisamment optimiser la coupe, car l'algorithme tombe trop rapidement dans un minimum local. Les outils existants résolvent ce problème en utilisant la méthode de « multi-niveaux » [1], [71], [72]. Il devrait être possible d'implémenter cette méthode, ou du moins une partie, lors du chaînage d'algorithme, de façon à mieux optimiser la coupe, et peut être d'autres métriques d'intérêt. Une autre solution pour mieux optimiser la coupe serait de concevoir des algorithmes ayant plus de recul sur l'espace des solutions que les algorithmes d'optimisation locale, afin de passer outre les minima locaux.

Implémentation et accès mémoire Notre implémentation des algorithmes de partitionnement, Coupe, est écrite dans le langage de programmation Rust, et le parallélisme est implémenté en grande partie à l'aide de la bibliothèque de parallélisation automatique « Rayon ». Contrairement aux logiciels haute performance classiques comme Kokkos [29], Rayon propose peu de fonctionnalités pour minimiser les transferts de données en mémoire. Nous en avons vu les conséquences dans les expériences de performance au Chapitre 8. Comme les algorithmes de partitionnement de maillage ont des performances limitées par la bande passante mémoire, il est important que l'implémentation fasse attention à limiter les accès superflus. Pour résoudre ce problème, il faut une approche au parallélisme plus fine que ce qui est implémenté actuellement. Pour l'instant, le parallélisme dans Coupe est plat : les opérations parallèles sont exécutées sur l'ensemble des threads matériel. Cela pose problème car les temps d'accès à la mémoire dépendent de la localisation des données, sur le matériel, et du thread qui souhaite y accéder. Pour faire en sorte que les temps d'accès soient plus rapides, il faut placer explicitement les données en mémoire, et spécifier les threads qui exécutent chaque opération. Une approche classique pour ce faire est d'organiser le parallélisme sous forme de hiérarchie [73], où les opérations parallèles qui accèdent aux mêmes données sont exécutées par des threads qui sont « proches » en termes d'accès mémoire (par exemple, partagent un niveau de cache). Par ailleurs, cette approche est grandement utilisée pour exécuter du code sur GPU, qui fonctionne sous forme hiérarchique. Cependant, cela est difficile à implémenter, et augmente considérablement la complexité de l'implémentation.

Partitionnement avec de l'intelligence artificielle Plusieurs raisons poussent à utiliser les méthodes d'intelligence artificielle (IA) pour le partitionnement. La première est que, pour certaines simulations numériques, le partitionnement est une opération qui n'influence pas la valeur des résultats du calcul. Il est alors possible d'utiliser des méthodes plus approximatives, mais qui donnent des résultats plus rapidement. Ces méthodes pourraient reposer sur une base de données de simulations, qui répertorie les caractéristiques d'exécution en fonction de la partition et des charges du maillage. La seconde raison d'utiliser des méthodes est le développement récent des puces électroniques spécialisées pour l'apprentissage et l'exécution de réseaux de neurones. Ces puces sont plus efficaces en énergie, notamment grâce à l'utilisation de nombres flottants à faible précision. Enfin, si le partitionnement est un problème dont les algorithmes sont classiquement limités par la bande passante ou la latence mémoire, comme nous l'avons vu dans les expériences des Chapitres 6

et 8, utiliser l'IA peut être une façon de le transformer en problème limité par la puissance de calcul. Ceci est avantageux, car la puissance de calcul des puces électroniques a augmenté plus rapidement que la latence et la bande passante mémoire ces dernières années [75].

Références

[1] G. Karypis et V. Kumar, « A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs », University of Minnesota, Technical report 95-035, juin 1995.

[2] T. Trilinos Project Team, *The Trilinos Project Website*. 2020. Via : https://trilinos.github.io

[3] F. Pellegrini, « Scotch and PT-Scotch Graph Partitioning Software : An Overview », in *Combinatorial Scientific Computing*, O. S. Uwe Naumann, Éd., Chapman and Hall/CRC, 2012, p. 373--406. doi : 10.1201/b11644-15. Via : https://inria.hal.science/hal-00770422

[4] G. M. Slota, K. Madduri, et S. Rajamanickam, « PuLP : Scalable Multi-Objective Multi-Constraint Partitioning for Small-World Networks. », oct. 2014, doi : 10.1109/BigData.2014.7004265. Via : https://www.osti.gov/biblio/1242087

[5] H. Meyerhenke, P. Sanders, et C. Schulz, « Parallel Graph Partitioning for Complex Networks », *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, n° 9, p. 2625--2638, 2017, doi : 10.1109/TPDS.2017.2671868. Via : https://doi.org/10.1109/TPDS.2017.2671868

[6] U. V. Catalyurek et C. Aykanat, « PaToH : Partitioning Tool for Hypergraphs ». mars 2011. Via : https://hal.archives-ouvertes.fr/hal-00410327

[7] B. Vastenhouw et R. H. Bisseling, « A Two-Dimensional Data Distribution Method for Parallel Sparse Matrix-Vector Multiplication », *SIAM Review*, vol. 47, n° 1, p. 67--95, 2005, doi : 10.1137/S0036144502409019. Via : https://doi.org/10.1137/S0036144502409019

[8] M. J. Berger et S. H. Bokhari, « A Partitioning Strategy for Nonuniform Problems on Multiprocessors », *IEEE Transactions on Computers*, vol. C–36, n° 5, p. 570--580, mai 1987, doi : 10.1109/TC.1987.1676942

[9] M. Deveci, S. Rajamanickam, K. D. Devine, et Ü. V. Çatalyürek, « Multi-Jagged : A Scalable Parallel Spatial Partitioning Algorithm », *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, n° 3, p. 803--817, mars 2016, doi : 10.1109/TPDS.2015.2412545

[10] A. Pınar et C. Aykanat, « Fast optimal load balancing algorithms for 1D partitioning », *Journal of Parallel and Distributed Computing*, vol. 64, n° 8, p. 974--996, août 2004, doi : 10.1016/j.jpdc.2004.05.003. Via : https://linkinghub.elsevier.com/retrieve/pii/S0743731504000851

[11] Y. Han, B. Narahari, et H.-A. Choi, « Mapping a chain task to chained processors », *Information Processing Letters*, vol. 44, n° 3, p. 141--148, nov. 1992, doi : 10.1016/0020-0190(92)90054-Y. Via : https://linkinghub.elsevier.com/retrieve/pii/002001909290054Y

[12] C. Chevalier, F. Ledoux, et S. Morais, « A Multilevel Mesh Partitioning Algorithm Driven by Memory Constraints », in *2020 Proceedings of the SIAM Workshop on Combinatorial Scientific Computing (CSC)*, in Proceedings. Society for Industrial and Applied Mathematics, 2020, p. 85--95. doi : 10.1137/1.9781611976229.9

[13] N. D. Matsakis et F. S. Klock, « The rust language », *Ada Lett.*, vol. 34, n° 3, p. 103--104, oct. 2014, doi : 10.1145/2692956.2663188. Via : https://doi.org/10.1145/2692956.2663188

[14] R. Barat, « Load Balancing of Multi-physics Simulation by Multi-criteria Graph Partitioning », thèse de doctorat, Université de Bordeaux, 2017. Via : https://tel.archives-ouvertes.fr/tel-01713977

[15] H. Hirtz, « Coupe : A Modular, Multi-threaded Mesh Partitioning Platform », in *Euro-Par 2022 : Parallel Processing Workshops*, J. Singer, Y. Elkhatib, D. Blanco Heras, P. Diehl, N. Brown, et A. Ilic, Éd., in Lecture Notes in Computer Science. Cham : Springer Nature Switzerland, 2023, p. 274--279. doi : 10.1007/978-3-031-31209-0_21

[16] C. Chevalier, H. Hirtz, F. Ledoux, et S. Morais, « COUPE : A MESH PARTITIONING PLATFORM », *International Meshing Roundtable*, 2023.

[17] L. G. Valiant, « A Bridging Model for Parallel Computation », *Commun. ACM*, vol. 33, n° 8, p. 103--111, août 1990, doi : 10.1145/79173.79181. Via : https://doi.org/10.1145/79173.79181

[18] P. J. Frey et P. L. George, *Maillages : Applications Aux Éléments Finis*. Hermès Science Publications, 1999. Via : https://books.google.fr/books?id=RLJ9AAAACAAJ

[19] P.-A. Beaufort, M. Reberol, D. Kalmykov, H. Liu, F. Ledoux, et D. Bommes, « Hex Me If You Can », *Computer Graphics Forum*, 2022, doi : 10.1111/cgf.14608

[20] G. Karypis et V. Kumar, ParMetis *Parallel Graph Partitioning and Sparse Matrix Ordering Library*. Minneapolis, MN 55455, U.S.A. : University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, 2003.

[21] C. Chevalier et F. Pellegrini, « PT-Scotch : A Tool for Efficient Parallel Graph Ordering », *Parallel Computing*, vol. 34, n° 6–8, p. 318--331, 2008, doi : 10.1016/j.parco.2007.12.001

[22] Message Passing Interface Forum, *MPI : A Message-Passing Interface Standard Version 4.0*. 2021. Via : https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[23] E. Gabriel *et al.*, « Open MPI : Goals, Concept, and Design of a Next Generation MPI Implementation », in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, 2004, p. 97--104.

[24] W. Gropp, E. Lusk, N. Doss, et A. Skjellum, « A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard », *Parallel Computing*, vol. 22, n° 6, p. 789--828, sept. 1996.

[25] M. Perache, H. Jourdren, et R. Namyst, « MPC : A Unified Parallel Runtime for Clusters of NUMA Machines », in *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, in Euro-Par '08. Berlin, Heidelberg : Springer-Verlag, 2008, p. 78--88. doi : 10.1007/978-3-540-85451-7_9. Via : http://dx.doi.org/10.1007/978-3-540-85451-7_9

[26] J. Liu *et al.*, « MPI over InfiniBand : Early Experiences », 2003.

[27] *The Open Group Technical Standard Base Specifications*, IEEE Std 1003.1., vol. 6. IEEE, 2004.

[28] L. Dagum et R. Menon, « OpenMP : an industry standard API for shared-memory programming », *IEEE Computational Science and Engineering*, vol. 5, n° 1, p. 46--55, janv. 1998, doi : 10.1109/99.660313

[29] C. R. Trott *et al.*, « Kokkos 3 : Programming Model Extensions for the Exascale Era », *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, n° 4, p. 805--817, 2022, doi : 10.1109/TPDS.2021.3097283

[30] « crossbeam ». Via : https://github.com/crossbeam-rs/crossbeam

[31] J. Stone et N. D. Matsakis, « Rayon ». rayon-rs, 2022. Via : https://github.com/rayon-rs/rayon

[32] M. R. Garey et D. S. Johnson, *Computers and Intractability : A Guide to the Theory of NP-Completeness*. New York, NY, USA : W. H. Freeman & Co., 1979.

[33] R. E. Korf, « From Approximate to Optimal Solutions : A Case Study of Number Partitioning », in *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, in IJCAI'95. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1995, p. 266--272.

[34] E. Horowitz et S. Sahni, « Computing Partitions with Applications to the Knapsack Problem », *Journal of the ACM (JACM)*, vol. 21, n° 2, p. 277--292, 1974, doi : 10.1145/321812.321823

[35] N. Karmarkar et R. M. Karp, « The Differencing Method of Set Partitioning », University of California at Berkeley, Berkeley, CA, USA, 1983.

[36] B. W. Kernighan et S. Lin, « An Efficient Heuristic Procedure for Partitioning Graphs », *The Bell System Technical Journal*, vol. 49, n° 2, p. 291--307, févr. 1970, doi : 10.1002/j.1538-7305.1970.tb01770.x

[37] C. M. Fiduccia et R. M. Mattheyses, « A Linear-Time Heuristic for Improving Network Partitions », in *19th Design Automation Conference*, juin 1982, p. 175--181. doi : 10.1109/DAC.1982.1585498

[38] V. M. Nguyen, « Compile-time Validation and Optimization of MPI Nonblocking Communications », These de doctorat, Bordeaux, 2022. Via : https://www.theses.fr/2022BORD0415

[39] L. E. Dubins et E. H. Spanier, « How to Cut A Cake Fairly », *The American Mathematical Monthly*, vol. 68, n° 1, p. 1--17, 1961, doi : 10.2307/2311357. Via : https://www.jstor.org/stable/2311357

[40] J. V. Peters, « The Ham Sandwich Theorem and Some Related Results », *The Rocky Mountain Journal of Mathematics*, vol. 11, n° 3, p. 473--482, 1981, Via : https://www.jstor.org/stable/44236614

[41] G. Karypis, « Multi-Constraint Mesh Partitioning for Contact/Impact Computations », in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, in SC '03. New York, NY, USA : ACM, 2003, p. 56--. doi : 10.1145/1048935.1050206. Via : http://doi.acm.org/10.1145/1048935.1050206

[42] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, et U. V. Catalyurek, « Parallel Hypergraph Partitioning for Scientific Computing », in *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, avr. 2006, p. 10 pp.--. doi : 10.1109/IPDPS.2006.1639359

[43] Ü. V. Çatalyürek et C. Aykanat, « PaToH : A Multilevel Hypergraph Partitioning Tool, Version 3.0. », *Bilkent University, Departement of Computer Engineering, Ankara, 06533 Turkey*, 1999.

[44] G. Karypis, R. Aggarwal, V. Kumar, et S. Shekhar, « Multilevel Hypergraph Partitioning : Applications in VLSI Domain », *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, n° 1, p. 69--79, 1999, doi : 10.1109/92.748202

[45] V. Sarkar, W. Harrod, et A. E. Snavely, « Software Challenges in Extreme Scale Systems », *Journal of Physics : Conference Series*, vol. 180, p. 012045, juill. 2009, doi : 10.1088/1742-6596/180/1/012045

[46] S. Morais, « Study and Obtention of Exact, and Approximation, Algorithms and Heuristics for a Mesh Partitioning Problem under Memory Constraints », Theses, Université Paris Saclay, 2016. Via : https://hal.archives-ouvertes.fr/tel-01447665

[47] G. W. B. 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*, vol. 38, n° 6, p. 989--1000, 1995.

[48] I. S. Duff et G. A. Meurant, « The effect of ordering on preconditioned conjugate gradients », *BIT*, vol. 29, n° 4, p. 635--657, déc. 1989, doi : 10.1007/BF01932738. Via : https://doi.org/10.1007/BF01932738

[49] C. Walshaw, M. Cross, R. Diekmann, et F. Schlimbach, « Multilevel Mesh Partitioning for Optimising Aspect Ratio », in *VECPAR*, 1998, p. 285--300.

[50] R. Diekmann, R. Preis, F. Schlimbach, et C. Walshaw, « Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM », *Parallel Computing*, vol. 26, n° 12, p. 1555--1581, nov. 2000, doi : 10.1016/S0167-8191(00)00043-0. Via : https://www.sciencedirect.com/science/article/pii/S0167819100000430

[51] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdag, R. Heaphy, et L. A. Riesen, « Hypergraph-Based Dynamic Load Balancing for Adaptive Scientific Computations », in *2007 IEEE International Parallel and Distributed Processing Symposium*, mars 2007, p. 1--11. doi : 10.1109/IPDPS.2007.370258

[52] C. Chevalier, G. Grospellier, F. Ledoux, et J. C. Weill, « Load Balancing for Mesh Based Multi-Physics Simulations in the Arcane Framework », présenté au The Eighth International Conference on Engineering Computational Technology, Dubrovnik, Croatia, p. 4. doi : 10.4203/ccp.100.4. Via : http://www.ctresources.info/ccp/paper.html?id=7144

[53] R. E. Korf, « A Complete Anytime Algorithm for Number Partitioning », *Artificial Intelligence*, vol. 106, n° 2, p. 181--203, déc. 1998, doi : 10.1016/S0004-3702(98)00086-1

[54] R. D. Williams, « Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations », *Concurrency : Practice and Experience*, vol. 3, n° 5, p. 457--481, 1991, doi : 10.1002/cpe.4330030502. Via : http://dx.doi.org/10.1002/cpe.4330030502

[55] J. R. Pilkington et S. B. Baden, « Partitioning with Spacefilling Curves », Dept. of Computer Science and Engineering, University of California, San Diego, 1994.

[56] M. Bader, *Space-Filling Curves : An Introduction with Applications in Scientific Computing*. Springer Publishing Company, Incorporated, 2012.

[57] G. M. Morton, *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. International Business Machines Company, 1966.

[58] D. Hilbert, « Ueber die stetige Abbildung einer Line auf ein Flächenstück », *Math. Ann.*, vol. 38, n° 3, p. 459--460, sept. 1891, doi : 10.1007/BF01199431. Via : https://doi.org/10.1007/BF01199431

[59] E. G. Boman, U. V. Catalyurek, C. Chevalier, K. D. Devine, I. Safro, et M. M. Wolf, « Advances in Parallel Partitioning, Load Balancing and Matrix Ordering for Scientific Computing », présenté au Journal of Physics : Conference Series, 2009. doi : 10.1088/1742-6596/180/1/012008

[60] J. R. Pilkington, J. R. Pilkington, S. B. Baden, et S. B. Baden, « Partitioning with Spacefilling Curves », Dept. of Computer Science and Engineering, University of California, San Diego, 1994.

[61] M. Bader, *Space-Filling Curves*, vol. 9. in Texts in Computational Science et Engineering, vol. 9. Berlin, Heidelberg : Springer, 2013. doi : 10.1007/978-3-642-31046-1. Via : http://link.springer.com/10.1007/978-3-642-31046-1

[62] M. von Looz, C. Tzovas, et H. Meyerhenke, « Balanced k-means for Parallel Geometric Partitioning », arXiv, arXiv :1805.01208, mai 2018. doi : 10.48550/arXiv.1805.01208. Via : http://arxiv.org/abs/1805.01208

[63] S. Morais, E. Angel, C. Chevalier, F. Ledoux, K. T. Nguyen, et D. Regnault, « Algorithme Approché Pour Un Problème de Partitionnement de Maillage Sous Contrainte Mémoire », in *ROADEF - 15ème Congrès Annuel de La Société Française de Recherche Opérationnelle et d'aide à La Décision*, Bordeaux, France : Société française de recherche opérationnelle et d'aide à la décision, févr. 2014. Via : https://hal.archives-ouvertes.fr/hal-00946454

[64] S. Morais, « Etude et Obtention d'heuristiques et d'algorithmes Exacts et Approchés Pour Un Problème de Partitionnement de Maillage Sous Contraintes Mémoire », Theses, Université Paris Saclay, 2016. Via : https://hal.archives-ouvertes.fr/tel-01447665

[65] B. Heisler, « Criterion.rs ». 17 mai 2022. Via : https://github.com/bheisler/criterion.rs

[66] J. Lingle, « Proptest ». 12 mai 2022. Via : https://github.com/AltSysrq/proptest

[67] C. Burstedde, L. C. Wilcox, et O. Ghattas, « `p4est` : Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees », *SIAM Journal on Scientific Computing*, vol. 33, n° 3, p. 1103--1133, 2011, doi : 10.1137/100791634

[68] J. Teunissen et U. Ebert, « Afivo : A framework for quadtree/octree AMR with shared-memory parallelization and geometric multigrid methods », *Computer Physics Communications*, vol. 233, p. 156--166, 2018, doi : https://doi.org/10.1016/j.cpc.2018.06.018. Via : https://www.sciencedirect.com/science/article/pii/S0010465518302261

[69] W. F. Tinney et J. W. Walker, « Direct Solutions of Sparse Network Equations by Optimally Ordered Triangular Factorization », *J. Proc. IEEE*, vol. 55, p. 1801--1809, 1967.

[70] W. Kahan, « Pracniques : further remarks on reducing truncation errors », *Commun. ACM*, vol. 8, n° 1, p. 40, janv. 1965, doi : 10.1145/363707.363723. Via : https://dl.acm.org/doi/10.1145/363707.363723

[71] T. N. Bui et C. Jones, « A Heuristic for Reducing Fill-in in Sparse Matrix Factorization », United States : Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA (United States), 1993.

[72] B. Hendrickson et R. Leland, « A Multilevel Algorithm for Partitioning Graphs », in *Proceedings of Supercomputing*, 1995.

[73] H. Carter Edwards, C. R. Trott, et D. Sunderland, « Kokkos : Enabling manycore performance portability through polymorphic memory access patterns », *Journal of Parallel and Distributed Computing*, vol. 74, n° 12, p. 3202--3216, déc. 2014, doi : 10.1016/j.jpdc.2014.07.003. Via : https://www.sciencedirect.com/science/article/pii/S0743731514001257. [Consulté le : 23 octobre 2023]

[74] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, et J. Manferdelli, « Fast scan algorithms on graphics processors », in *Proceedings of the 22nd annual international conference on Supercomputing*, in ICS '08. New York, NY, USA : Association for Computing Machinery, juin 2008, p. 205--213. doi : 10.1145/1375527.1375559. Via : https://dl.acm.org/doi/10.1145/1375527.1375559

[75] D. Efnusheva, A. Cholakoska, et A. Tentov, « A Survey of Different Approaches for Overcoming the Processor - Memory Bottleneck », *International Journal of Information Technology and Computer Science*, vol. 9, p. 151, avr. 2017, doi : 10.5121/ijcsit.2017.9214