Load Balancing of Multi-physics Simulation by Multi-criteria Graph Partitioning

Remi Barat

université de BORDEAUX

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Rémi Barat**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Équilibrage de charge pour des simulations multi-physiques par partitionnement multi-critères de graphes

---

Date de soutenance : 18 décembre 2017

Devant la commission d'examen composée de :

Cevdet AYKANAT, *Professor*, Bilkent University – Rapporteur
Rob BISSELING, *Professor*, Universiteit Utrecht – Rapporteur et Président du jury
Lélia BLIN, *Maître de conférence*, Université Évry-Val-d'Essonne – Examinatrice
Cédric CHEVALIER, *Ingénieur-chercheur*, CEA – Co-encadrant
Aurélien ESNARD, *Maître de conférence*, Université de Bordeaux – Examinateur
François PELLEGRINI, *Professeur*, Université de Bordeaux – Directeur de thèse

– 2017 –

Mots-clés Partitionnement, Graphe, Multi-critères, Multi-niveaux, Optimisation combinatoire

Résumé Les simulations dites multi-physiques couplent plusieurs phases de calcul. Lorsqu'elles sont exécutées en parallèle sur des architectures à mémoire distribuée, la minimisation du temps de restitution nécessite dans la plupart des cas d'équilibrer la charge entre les unités de traitement, pour chaque phase de calcul. En outre, la distribution des données doit minimiser les communications qu'elle induit.

Ce problème peut être modélisé comme un problème de partitionnement de graphe multi-critères. On associe à chaque sommet du graphe un vecteur de poids, dont les composantes, appelées « critères », modélisent la charge de calcul porté par le sommet pour chaque phase de calcul. Les arêtes entre les sommets, indiquent des dépendances de données, et peuvent être munies d'un poids reflétant le volume de communication transitant entre les deux sommets. L'objectif est de trouver une partition des sommets équilibrant le poids de chaque partie pour chaque critère, tout en minimisant la somme des poids des arêtes coupées, appelée « coupe ». Le déséquilibre maximum toléré entre les parties est prescrit par l'utilisateur. On cherche alors une partition minimisant la coupe, parmi toutes celles dont le déséquilibre pour chaque critère est inférieur à cette tolérance.

Ce problème étant NP-Dur dans le cas général, l'objet de cette thèse est de concevoir et d'implanter des heuristiques permettant de calculer efficacement de tels partitionnements. En effet, les outils actuels renvoient souvent des partitions dont le déséquilibre dépasse la tolérance prescrite.

Notre étude de l'espace des solutions, c'est-à-dire l'ensemble des partitions respectant les contraintes d'équilibre, révèle qu'en pratique, cet espace est immense. En outre, nous prouvons dans le cas mono-critère qu'une borne sur les poids normalisés des sommets garantit que l'espace des solutions est non-vide et connexe. Nous fondant sur ces résultats théoriques, nous proposons des améliorations de la méthode multi-niveaux. Les outils existants mettent en œuvre de nombreuses variations de cette méthode. Par l'étude de leurs codes sources, nous mettons en évidence ces variations et leurs conséquences à la lumière de notre analyse sur l'espace des solutions.

Par ailleurs, nous définissons et implantons deux algorithmes de partitionnement initial, se focalisant sur l'obtention d'une solution à partir d'une partition potentiellement déséquilibrée, au moyen de déplacements successifs de sommets. Le premier algorithme effectue un mouvement dès que celui-ci améliore l'équilibre, alors que le second effectue le mouvement réduisant le plus le déséquilibre. Nous présentons une structure de données originale, permettant d'optimiser le choix des sommets à déplacer, et conduisant à des partitions de déséquilibre inférieur en moyenne aux méthodes existantes.

Nous décrivons la plate-forme d'expérimentation, appelée Crack, que nous

avons conçue afin de comparer les différents algorithmes étudiés. Ces comparaisons sont effectuées en partitionnant un ensembles d'instances comprenant un cas industriel et plusieurs cas fictifs. Nous proposons une méthode de génération de cas réalistes de simulations de type « transport de particules ».

Nos résultats démontrent la nécessité de restreindre les poids des sommets lors de la phase de contraction de la méthode multi-niveaux. En outre, nous mettons en évidence l'influence de la stratégie d'ordonnancement des sommets, dépendante de la topologie du graphe, sur l'efficacité de l'algorithme d'appariement « Heavy-Edge Matching » dans cette même phase.

Les différents algorithmes que nous étudions sont implantés dans un outil de partitionnement libre appelé Scotch. Au cours de nos expériences, Scotch et Crack renvoient une partition équilibrée à chaque exécution, là où MeTiS, l'outil le plus utilisé actuellement, échoue une grande partie du temps. Qui plus est, la coupe des solutions renvoyées par Scotch et Crack est équivalente ou meilleure que celle renvoyée par MeTiS.

Laboratoire d'accueil CEA, DAM, DIF, Bruyères-le-Châtel, F-91297 Arpajon

Title Load Balancing of Multi-physics Simulation by Multi-criteria Graph Partitioning

Keywords Partitioning, Graph, Multi-criteria, Multilevel, Combinatorial optimization

Abstract Multiphysics simulation couple several computation phases. When they are run in parallel on memory-distributed architectures, minimizing the simulation time requires in most cases to balance the workload across computation units, for each computation phase. Moreover, the data distribution must minimize the induced communication.

This problem can be modeled as a multi-criteria graph partitioning problem. We associate with each vertex of the graph a vector of weights, whose components, called "criteria", model the workload of the vertex for each computation phase. The edges between vertices indicate data dependencies, and can be given a weight representing the communication volume transferred between the two vertices. The goal is to find a partition of the vertices that both balances the weights of each part for each criterion, and minimizes the "edgecut", that is, the sum of the weights of the edges cut by the partition. The maximum allowed imbalance is provided by the user, and we search for a partition that minimizes the edgecut, among all the partitions whose imbalance for each criterion is smaller than this threshold.

This problem being NP-Hard in the general case, this thesis aims at devising and implementing heuristics that allow us to compute efficiently such partitions. Indeed, existing tools often return partitions whose imbalance is higher than the prescribed tolerance.

Our study of the solution space, that is, the set of all the partitions respecting the balance constraints, reveals that, in practice, this space is extremely large. Moreover, we prove in the mono-criterion case that a bound on the normalized vertex weights guarantees the existence of a solution, and the connectivity of the solution space. Based on these theoretical results, we propose improvements of the multilevel algorithm. Existing tools implement many variations of this algorithm. By studying their source code, we emphasize these variations and their consequences, in light of our analysis of the solution space.

Furthermore, we define and implement two initial partitioning algorithms, focusing on returning a solution. From a potentially imbalanced partition, they successively move vertices from one part to another. The first algorithm performs any move that reduces the imbalance, while the second performs at each step the move reducing the most the imbalance. We present an original data structure that allows us to optimize the choice of the vertex to move, and leads to partitions of imbalance smaller on average than existing methods.

We describe the experimentation framework, named Crack, that we implemented in order to compare the various algorithms at stake. This comparison

is performed by partitioning a set of instances including an industrial test case, and several fictitious cases. We define a method for generating realistic weight distributions corresponding to "Particles-in-Cells"-like simulations.

Our results demonstrate the necessity to coerce the vertex weights during the coarsening phase of the multilevel algorithm. Moreover, we evidence the impact of the vertex ordering, which should depend on the graph topology, on the efficiency of the "Heavy-Edge" matching scheme.

The various algorithms that we consider are implemented in an open- source graph partitioning software called Scotch. In our experiments, Scotch and Crack returned a balanced partition for each execution, whereas MeTiS, the current most used partitioning tool, fails regularly. Additionally, the edgecut of the solutions returned by Scotch and Crack is equivalent or better than the edgecut of the solutions returned by MeTiS.

Résumé substantiel

Contexte. Les simulations numériques permettent d'effectuer à l'aide d'un ordinateur des expériences qui peuvent être coûteuses ou difficiles à mettre en œuvre. Néanmoins, effectuer une simulation numérique nécessite des ressources informatiques et du temps. Par conséquent, certaines simulations sont lancées sur plusieurs unités de calcul, pour diminuer leur temps d'exécution, ou bien parce qu'une unité ne suffit pas pour stocker toutes les données de la simulation. On dit que de telles simulations sont effecuées « en mémoire distribuée ».

Par ailleurs, afin de modéliser les phénomènes physiques qu'elles étudient, beaucoup de simulations utilisent un maillage. Un maillage est une discrétisation de l'espace, c'est-à-dire que chacun de ses éléments, appelé une maille, est une portion de l'espace. Une maille modélise la valeur d'un phénomène continu sur la portion de l'espace qu'elle représente. Par conséquent, les mailles servent de support aux données de la simulation. Lorsqu'une simulation reposant sur un maillage est effectuée en mémoire distribuée, le maillage est partitionné, c'est-à-dire que chaque maille est attribuée à une unité de calcul. Afin de minimiser le temps de la simulation, la partition doit équilibrer autant que possible la charge de travail entre les unités de calcul.

La charge de travail d'une unité de calcul dépend des mailles qui lui sont attribuées. En effet, une unité de calcul doit, au fur et à mesure de la simulation, mettre à jour les données de ces mailles. La quantité de calcul nécessaire pour mettre à jour une maille varie selon les mailles et, dans le cas de simulations dites « multi-physiques », elle dépend également de la physique considérée. Dans ces simulations, les phases de calcul pour chaque physique sont effectuées l'une après l'autre car elles sont séparées par des phases de synchronisation. Par conséquent, l'équilibrage de la charge de travail doit être effectué pour chaque phase de calcul.

Les phases de synchronisation permettent aux unités de calcul de s'échanger des données, par envoi de messages. En effet, mettre à jour les données d'une maille nécessite en général de disposer des données de ses voisines, donc les données des mailles à la frontière de la partition doivent être communiquées. Les phases de synchronisation entraînant un surcoût temporel, minimiser le temps de la simulation nécessite aussi de minimiser le temps dû à ces communications.

Ainsi, minimiser le temps d'exécution des simulations multi-physiques,

reposant sur un maillage et effectuées en mémoire distribuée, nécessite entre autres d'équilibrer la charge de travail entre les unités de calcul pour chaque phase de calcul, et de minimiser le temps des phases de communication.

Modèles. Nous présentons plusieurs modèles classiques visant à minimiser le temps d'exécution des simulations multi-physiques en mémoire distribuée. Dans chacun d'entre eux, la charge de travail par maille et par physique est modélisée par un poids affecté à cette maille. Dans le cas des simulations multi-physiques, c'est donc un vecteur de poids qui est affecté à chaque maille, dont chaque composante, appelée « critère », représente la charge pour une phase de calcul. Équilibrer la charge de travail revient alors à déterminer une partition du maillage qui équilibre les poids de chaque partie pour chaque critère.

Pour minimiser le coût des communications, différents modèles existent. Nous présentons deux modèles, l'un partitionnant directement le maillage, l'autre le représentant par un hypergraphe, calculant le volume de communication induit par la partition. Un troisième modèle représente le maillage par un graphe, et estime de manière approchée ce volume de communication. Nous détaillons les limites des ces modèles au vu de la complexité des schémas de communication possibles, pour les machines à mémoire distribuée.

Enfin, afin de pouvoir choisir entre une partition plus équilibrée et une autre de moindre coût de communication, dans chacun des trois modèles, l'objectif d'équilibrer la charge de travail est transformé en contrainte. L'utilisateur doit alors fournir une tolérance de déséquilibre, et toute partition de déséquilibre supérieur à cette tolérance n'est pas considérée comme solution. Parmi toutes les partitions solutions, nous cherchons une solution qui minimise le coût de communication, dite « optimale ». Ce problème, suivant le modèle choisi, est appelé « partitionnement multi-critères de maillage/de graphe/d'hypergraphe », et constitue le problème que nous souhaitons résoudre dans le cadre de cette thèse. Nous nous intéressons plus particulièrement au modèle de graphe, plus simple et très utilisé.

État de l'art. Nous détaillons d'abord des heuristiques visant à partitionner de façon équilibrée un ensemble de nombres. Ce problème, appelé « partitionnement de nombres », cherche une solution (pas forcément optimale) à un problème de partitionnement de graphe mono-critère. De nombreuses heuristiques traitent ce problème. Cependant, très peu se généralisent à notre cas, qui est celui du partitionnement de vecteurs de nombres, aussi appelé « partitionnement multi-dimensionnel de nombres ». Dans cette thèse, deux algorithmes de partitionnement de vecteurs de nombres seront étudiés.

Nous présentons ensuite les algorithmes classiques visant à résoudre le problème du partitionnement de maillage, de graphe ou d'hypergraphe, à chaque fois en expliquant les différences entre les version mono- et multi-critères. La méthode la plus utilisée est la méthode multi-niveaux, composée

de trois phases. Dans la première phase, appelée phase de contraction, une succession de graphes grossiers est construite en groupant des sommets du graphe, de façon à réduire leur taille. Dans la deuxième phase, le graphe le plus grossier est partitionné : c'est la phase de partitionnement initial. La partition obtenue est successivement prolongée sur un graphe moins grossier (au niveau supérieur), et raffinée à l'aide d'un algorithme d'optimisation locale. L'algorithme le plus utilisé pour le raffinement est l'algorithme de Fiduccia-Mattheyses (`FM`), qui change successivement de partie des sommets, de façon à améliorer le plus possible le coût de communication à chaque mouvement. `FM` n'autorise de déplacer un sommets que si la partition obtenue après déplacement est solution, ce qui suppose en général que la partition fournie en entrée soit déjà solution.

Des outils traitant le problème du partitionnement multi-critères de graphe existent déjà, mais ils renvoient souvent des solutions ne respectant pas la tolérance de déséquilibre.

Étude de l'espace des solutions. Étant donné un graphe G, un nombre de parties k et une tolérance t, l'espace des solutions est l'ensemble des partitions à k parties de G de déséquilibre inférieur ou égal à t. Nous montrons par une méthode probabiliste de type Monte-Carlo qu'en pratique, même lorsque la tolérance est très faible, l'espace des solutions est gigantesque.

Par ailleurs, dans le cas mono-critère, nous prouvons que si les poids normalisés des sommets sont bornés par t, alors il existe une solution. Une borne similaire, $t/2$, garantit la connexité de l'espace des solutions pour les algorithmes d'optimisation locale de type `FM` (déplaçant un seul sommet à la fois). Cela signifie que, quelle que soit la solution fournie en entrée, `FM` peut renvoyer une solution optimale. Ces deux résultats nous mènent à conjecturer que, dans le cas multi-critères, plus les poids (normalisés) des sommets sont petits, plus l'espace des solution a de chance d'être grand et connecté, c'est-à-dire qu'il est plus facile pour un algorithme de trouver une solution, et qu'il est aussi plus facile pour un algorithme d'optimisation locale de type `FM` de trouver une solution optimale.

Mise en place d'un algorithme multi-niveaux tenant compte des poids des sommets. Nous basant sur ces conjectures, nous analysons plusieurs variations de la méthode multi-niveaux. Les différents logiciels actuels de partitionnement multi-critères qui reposent sur cette méthode l'implantent d'ailleurs avec des variations. Nous confrontons nos propres choix algorithmiques avec ces implantations, en nous basant, lorsque cela est possible, directement sur le code source de ces outils.

Contraction. Durant cette phase de la méthode multi-niveaux, les sommets du graphe sont regroupés de façon à réduire la taille du graphe. Classiquement,

on essaie d'apparier ensemble les sommets aux extrémités des arêtes de fort poids, selon l'algorithme *Heavy-Edge Matching* (`HEM`). Nous analysons comment des variations de cet algorithme peuvent augmenter la probabilité de renvoyer une solution optimale, en regard de nos conjectures. En effet, lorsque deux sommets sont appariés, leurs poids s'ajoutent (vectoriellement), ce qui modifie la distribution des poids. Nous considérons des restrictions sur les poids des sommets dans le graphe grossier, ou différents ordonnancement des sommets (avant de calculer l'appariement) suivant des critères de poids ou de topologie.

Nous montrons expérimentalement que plus on restreint les poids des sommets, plus le déséquilibre des partitions retournées par des algorithmes de partitionnement initial diminiue. Cela valide notre hypothèse que plus les poids normalisés sont faibles, plus il est facile pour un algorithme de retourner une solution.

Concernant la capacité à trouver une partition optimale, s'il n'est pas toujours efficace de restreindre le plus possible les poids, des restrictions aident dans la plupart des cas l'algorithme multi-niveaux à renvoyer des partitions de plus faible coût de communication.

Les algorithmes choisissant l'ordre des sommets pour calculer l'appariement influencent le coût de communication des solutions renvoyées. Ainsi, ordonner les sommets toujours dans le même ordre mène à des solutions de fort coût de communication. Il est préférable d'utiliser un ordre aléatoire, ou privilégiant les sommets de faible poids, ou bien ordonnant les sommets par degré croissant. Ce dernier algorithme semble être le plus robuste.

Partitionnement initial. Étant donné que beaucoup de logiciels de partitionnement renvoient des solutions ne respectant pas les contraintes d'équilibrage, nous proposons deux algorithmes se focalisant sur l'équilibre de la partition renvoyée, traitant un problème de partitionnement multi-critères de graphe comme un problème de partitionnement de vecteurs de nombres.

Ces algorithmes déplacent successivement un sommet de façon à améliorer l'équilibre. Le premier algorithme, appelé `VNFirst`, itère sur les sommets, changeant un sommet de partie dès lors que cela conduit à une réducation du déséquilibre. Le second algorithme, appelé `VNBest`, change à chaque fois de partie un sommet de manière à diminuer le plus possible le déséquilibre, à la manière de l'algorithme du gradient. Dans le cas du bipartitionnement, nous détaillons une structure de données qui diminue la complexité de cet algorithme, lui permettant d'atteindre un temps d'exécution inférieur à d'autres algorithmes de partitionnement initial, tel que l'algorithme *Greedy Graph Growing* (`GGG`).

Cette structure se base sur l'étude de la fonction du gain en équilibre d'un sommet. Le gain en équilibre d'un sommet est la réduction du déséquilibre induite si on le change de partie. Étant donnée une bipartition Π, de déséquilibre $imb(\Pi)$, le gain en équilibre d'un sommet dépend seulement de son vecteur de poids. Plaçons nous d'abord dans le cas du bipartitionnement mono-critère, et

considérons les variations de cette fonction de gain en équilibre. Cette fonction est croissante avec le poids du sommet, atteint un maximum si le sommet est de poids $imb(\Pi)/2$, puis décroît. Si les sommets sont triés par poids croissant, on peut donc trouver le maximum sans calculer le gain de tous les sommets. En outre, les sommets de poids supérieur à $imb(\Pi)$ sont de gain négatif. Comme, au cours de notre algorithme, le déséquilibre décroit strictement, ces sommets ne changeront plus de partie, et n'ont plus besoin d'être considérés. Enfin, le gain de certains sommets ne change pas après un mouvement, ce qui évite d'avoir à recalculer les gains de tous les sommet après chaque mouvement.

Nous généralisons notre structure de données au cas du bipartitionnement multi-critères. Trouver le sommet de meilleur gain est plus complexe, mais nous détaillons un mécanisme permettant de le trouver sans avoir à considérer tous les sommets. Pour ce faire, notons c_{max} le critère le plus déséquilibré. En itérant sur les sommets par gain décroissant pour c_{max}, dès lors que c_{max} ne change pas lorsqu'on considère bouger le sommet v, alors il n'est pas nécessaire de considérer les sommets de gain inférieur pour c_{max} à celui de v.

Nous comparons `VNFirst` et `VNBest` à `GGG` et à un algorithme de partitionnement aléatoire, `Randomize`. Nous montrons que `VNFirst` et surtout `VNBest` parviennent à obtenir en moyenne des partitions des graphes grossiers de déséquilibre inférieur à celles retournées par `GGG` et `Randomize`. En revanche, lorsque l'algorithme multi-niveaux utilise `VNBest` comme algorithme de partitionnement initial, lorsque la tolérance est plus stricte, les solutions retournées en utilisant `GGG` sont de plus faible coût de communication.

Raffinement. Nous formalisons l'algorithme de raffinement `FM` de manière à mettre en avant les ambigüités possibles sur son implantation, et en étudiant comment ces ambigüités ont été levées par différents logiciels de partitionnement. Par exemple, si `FM` est défini de sorte qu'on choisisse à chaque fois de déplacer un sommet de façon à diminuer le plus le coût de communication, en cas d'égalité, les politiques diffèrent en fonction des logiciels. Surtout, certains logiciels relâchent la tolérance durant cette phase, afin d'augmenter la connexité de l'espace des solutions. Au vu de nos conjectures, nous soutenons qu'un tel relâchement n'est pas nécessaire, et qu'au contraire, il mène à renvoyer des partitions qui ne sont pas des solutions.

Environnement expérimental. Nous avons mis en place une plate-forme d'expérimentation, appelée Crack. Crack est implanté en Python et peut-être considéré comme un logiciel flexible de partitionnement de maillage, de graphe ou d'hypergraphe. Nous décrivons ainsi une représentation originale de l'algorithme multi-niveaux, sous forme d'automate, qui permet de spécifier clairement et simplement l'enchaînement des algorithmes utilisés et leurs paramètres.

Nous définissons les instances nous servant à comparer les différents algorithmes considérés. Ces instances comprennent un cas industriel et cinq

maillages fictifs. Pour ces derniers, nous définissons une méthode de génération des poids qui retourne une distribution de poids similaire à celles qu'on retrouve lors de simulations de type Monte-Carlo, pour le transport de particules.

Chaque algorithme est lancé 100 fois sur chaque instance, afin d'étudier, d'abord, la capacité de chaque algorithme à renvoyer une solution et, ensuite, la distribution du coût de communication des solutions renvoyées. Ces distributions sont comparées à l'aide des profils de performance, qui tracent l'effectif cumulé des solutions en fonction de leur coût de communication. Les courbes obtenues permettent d'analyser et de représenter les résultats de chaque algorithme plus finement que les indicateurs classiques (moyenne, médiane, écart-type).

Comparaison avec d'autres logiciels de partitionnement multi-critères. Pour finir, nous comparons les résultats de Crack avec ceux des logiciels MeTiS et PaToH. En outre, nous présentons les résultats d'une implantation d'une partie de nos algorithmes dans Scotch, un outil industriel de partitionnement de graphe écrit en C. Nos algorithmes renvoient à chaque fois une solution, contrairement à MeTiS et PaToH. Par ailleurs, les solutions renvoyées ont en général un coût de communication équivalent ou meilleur à celui des autres outils considérés.

Acknowledgments

I thank immensely Cédric Chevalier, my supervisor, for his advice and enlightening analysis of the results I got all along the thesis, be they theoretical or experimental. Cédric, it was a pleasure to work with you. Thanks to your knowledge on, it seems, every possible subjects, I have explored many counter-intuitive ideas and learned valuable concepts.

I am also grateful to François Pellegrini, my thesis director, for his reasoning filled with imagery, and for correcting my manuscript so that it would be more digest. I was astonished by how fast you read my thesis and how fast you were able to code in Scotch.

Then, my thanks go to my family. In particular, to Ana, who shares my life and takes care of me. Ana, thank you for supporting me throughout the thesis, even when work entangled with our holidays. Thanks for bringing me around the world together with Trèzétonné. Trèzétonné, thank you for your eternal smile, your lucidity and all your daily pertinent remarks. I thank my parents, and Ilac and JJ, for cheering and helping us in everyday life (and for saving us during our wedding). Thanks to my sister for her trust and energy, that are contagious. Finally, I thank all my family (grand-parents, aunts, uncles, cousins...) because it is so much fun when we are together.

More than a job, this thesis has been a delightful occupation, thanks to my remarkable colleagues. Above all, we had memorable adventures as a climbing team, with Adrien, Arthur and BHugo. Special thanks to Adrien: climbing multi-pitch routes in Mallorca and Meteora, and hiking with you in the cold Alpes were thrilling experiences. Arthur, playing chess with you was a real challenge; I hope you will continue aiming for the top. BHugo, you will definitely succeed in your projects, so cheer up. I also thank my thesis brothers, Gautier, Hoby, Guillaume, and Alexis; we had a fantastic time together, and it is also for that reason that I continued at CEA after the internship. To my partitioning big brother, Sébastien, I will miss your strange sense of communication. Éloise, continue your unexpected drawings, I wish you all the best to you and Adrien, the kindest couple I have met. THugo, thanks for the karting and the tennis, and whenever you lose, remember to keep calm and smile. Xavier, thanks for the music, it was a great concert. Denis, I hope that you stayed the same after you left. Know that we really missed you (and your jokes). Thanks to all the internship students I have met (in particular, Clément, Simon, Ewan,

Alexandre, Maxime×2, Estelle, Quentin, Florent, and Kiki). Last but not least, I thank Hadrien, Anna, and Guillaume, for the exceptional time we spent in Bordeaux together. Hadrien, Our Father, I have carried on thy task of converting people to rock climbing.

I thank all the members of the lab, especially Frank, Nicolas, JC, Benoît, Marie-Pierre, Pierre, Éric, and Bruno, for the funny coffee times. I thank Patrick, Marc, and Julien from the MPC team. Marc, I appreciated your efforts on the post-doc opportunity. Patrick, thank you for showing me the power of Vim during my internship. I am grateful to Brigitte×2 and Isabelle, who have been taking care of us with marvelous kindness at all time.

Finally, my thanks also go to my reviewers, Cevdev Aykanat and Rob Bisseling, who managed to read this long document really fast. Your remarks provided new interesting insights, and I really enjoyed our discussions.

Contents

Introduction

Context. Numerical simulations model complex phenomena, avoiding experiments which would otherwise be expensive or hard, if not impossible, to conduct. However, numerical simulations also require computation resources, memory resources, and time. Therefore, some simulations must be executed in parallel, on distributed memory architectures, because their memory usage does not fit in the memory of a single chip and/or because one needs to reduce their run time.

In order to represent the complex phenomena, and the space in which these phenomena occur, many simulations use a discrete entity called mesh. The mesh elements model physical data that are computed along the simulation. Therefore, the workload is divided up across the available computation units by partitioning the mesh. This means that every mesh element is attributed to a single computation unit.

When the simulation couples several physical models, it is said to be "multiphysics". In multiphysics simulations, the mesh elements involving heavier computations usually vary from one physical model to another. Moreover, data for different physical models are computed during distinct computation phases, that are executed one at a time.

Challenges. In order to minimize the run time of mesh-based simulations executed on distributed memory architectures, the mesh must be partitioned so that every computation unit gets the same workload. Therefore, one needs to find a balanced partition of the mesh that exhibits several properties:

(1) in the case of multiphysics simulation, the partition must minimize, for every computation phase, the maximum workload attributed to the computation units. This implies balancing the workloads for every phase of the computation;

(2) besides, during the simulation, one computation unit may need data owned by another unit, which requires some form of communication. Communications are usually performed after every computation phase, in a synchronization step, thus increasing the run time of the simulation. Therefore, the partition of the mesh must also minimize the induced communication time.

Properties (1) and (2) can be modeled with the multi-criteria mesh partitioning problem, or with the multi-criteria graph or hypergraph partitioning problems when considering only the topology of the mesh (which means, without relying on the mesh geometry).

The multi-criteria mesh/graph/hypergraph partitioning problems are NP-Hard, but heuristics exist, and some multi-criteria partitioning tools have been implemented. However, they often fail to return balanced partitions, and there is limited research investigating the cause of this flaw.

Purpose of this work. This study aims at reducing the run time of simulations that couple several computation phases, by devising and developing a multi-criteria mesh partitioning method which balances the workload between computation units for all computation phases, and additionally minimizes the induced communication cost.

I Context, Model and State of the Art. In Chapter 1, we first present the use of numerical simulations, and explain why some simulations are run on distributed memory machines. As many simulations use meshes to discretize the continuous domains involved, we give a simplified definition of what a mesh is.

In order to model the run time of these mesh-based simulations running on distributed memory architectures, we then examine typical executions of such simulations. Basically, these simulations perform computation phases for a sequence of time steps. The essential characteristics are that the computation phases depend on each other, and that they are separated by communication phases. As a result, minimizing the run time requires to balance the workloads between computation units for every computation phase, and to minimize the duration of the communication phases.

In Chapter 2, we define the multi-criteria mesh, graph and hypergraph partitioning problems, whose solutions should minimize the run time of multiphysics simulations. In all three models, the computational workload of a mesh element for one phase is represented by a weight associated with this element. For multiphysics simulations, which couple several computation phases, a vector of weights is therefore associated with every element. The components of a vector of weights are called criteria, hence the name "multi-criteria" mesh/graph/hypergraph partitioning.

The three models define communication costs to represent the communication time, and they all change the objective of minimizing the imbalance into a constraint. This constraint is set by the user, who defines the tolerance, the maximum imbalance acceptable for a partition to consider it as a solution. A solution minimizing the communication cost is called optimal solution.

The problem of finding a solution amounts to solving a vector-of-numbers partitioning problem, which we also define. Besides, in order to study heuristics

addressing the mesh, graph or hypergraph partitioning problem, we introduce a more general formulation that uses fitness landscapes, which help study properties of the solution space, the set of all solutions. Characterizing the solution space of an instance informs on whether an algorithm is suited to solve this instance.

Chapter 3 defines algorithms addressing the vector-of-numbers partitioning problem. Then, Chapter 4 defines heuristics addressing the multi-criteria mesh, graph, or hypergraph partitioning problems. Among them, the multilevel algorithm is very efficient and is used by most of the existing graph partitioning tools. However, these tools often return imbalanced solutions, so we will investigate how to avoid such a behavior.

II Approach. In Chapter 5, we experimentally study the number of solutions for a set of multi-criteria instances. We also formulate two theorems, in the mono-criterion case, which state, depending on the vertex weights, the non-emptiness and the connection of the solution space. The connection of the solution space depends on the algorithm considered, and examines whether from any starting solution, the algorithm can improve it in order to reach an optimal solution. Our theorem on the connection of the solution space is valid for "Fiduccia-Mattheyses-like" ("FM-like") local optimization algorithms that, given a solution, pass to another solution by switching the part of a single vertex.

These theoretical results lead to two conjectures on how algorithms can benefit from smaller normalized weights. Based on these assumptions, we define and analyze algorithmic choices for the multilevel algorithm. Indeed, throughout the thesis, we will show that the partitioning tools Scotch, MeTiS and PaToH, which all implement the multilevel algorithm, integrate many variations. However, these variations are not always documented; we understood many of them by analyzing their source code. In Chapters from 6 to 8, we rigorously define general versions of the algorithms used in the multilevel framework, analyze variations of them, and compare these variations with existing implementations.

Chapter 6 focuses on the first phase of the multilevel algorithm, called the coarsening phase. This phase coarsens the mesh, modifying the weight distribution of the original mesh by matching cells (the elements of the mesh). In accordance with our conjectures, we propose several matching schemes that aim at forming smaller weights for the coarsened cells. To do so, we use two different mechanisms. The first one directly restricts the weights when coarsening, and the second one orders the cells in different ways before computing the matching.

Then, in Chapter 7, we propose two algorithms for the second phase of the multilevel algorithm, the initial partitioning phase. Our algorithms focus on returning a solution by addressing a vector-of-numbers partitioning problem.

Both of our algorithms are local optimization algorithms: at each step, they switch the part of one vertex. The first algorithm switches the part of any vertex if it decreases the imbalance of the partition, while the second algorithm selects the vertex so that the imbalance decreases the most. For the latter algorithm, we also detail a data structure enabling to find such a vertex in a reduced amount of computation.

Finally, Chapter 8 details the third phase of the multilevel algorithm, the uncoarsening or expansion phase. This phase uses a refinement algorithm to reduce the communication cost of the partitions of the coarse mesh, until a partition of the original mesh is found. Most partitioning tools rely on a local optimization algorithm for refinement, but many tools choose to relax the imbalance tolerance during this phase, and we discuss their choice. Indeed, relaxing the tolerance increases the probability to return imbalanced partitions. In order to fix an excess of imbalance, MeTiS, Scotch and PaToH use various mechanisms, that we also detail in order to be able to draw conclusions from the results obtained with these tools.

III Experiments. In Chapter 9, we first show that existing tools return partitions of various communication cost, so a comparison of algorithms needs to execute them many times. We also introduce cumulative plots (also known as performance profiles), the statistical indicator that we adopted to compare heuristics. Furthermore, we describe the instances that we used for the tests. There is one industrial case, and five meshes for which we generated three fictitious weight distributions per mesh. Given any mesh, we define a method to generate a weight distribution reflecting a particle-in-cell simulation relying on this mesh. Finally, we describe some features of Crack, the flexible graph partitioning tool that we implemented in Python.

Using Crack, in Chapter 10, we compare the various heuristics defined. The aim is twofold: firstly, attempt to validate our conjectures that algorithms benefit from smaller normalized weights, and secondly, find the best heuristic among the studied ones. The heuristics vary with the restriction policy and the cell ordering for the coarsening phase, and with the initial partitioning algorithm. Finally, we also compare the multi-criteria version of Scotch that we implemented and the best Crack heuristics, with MeTiS and PaToH, for an imbalance tolerance $t \in \{5\%, 1\%, 0.2\%\}$ and a number of parts $k \in \{2, 32, 128\}$.

Mathematical Definitions, Pseudo-code Considerations and Notations

Definitions

Definition 1 (Part)
Let S be a set. We call part of S a subset of elements of S.
The set of all parts of S is denoted by $\mathcal{P}(S)$.

Example
If $S = \{a, b, c\}$, then the set of all parts of S is:

$$\mathcal{P}(S) = \{\varnothing, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$$

Definition 2 (Partition)
Let S be a set. We call partition of S a set Π of parts of S such that:

- $\bigcup_{\Pi_p \in \Pi} \Pi_p = S$*: the union of all parts in Π is equal to S;*
- $\forall (\Pi_p, \Pi_q) \in \Pi^2, \Pi_p \neq \Pi_q \implies \Pi_p \cap \Pi_q = \varnothing$*: the intersection of two distinct parts in Π is empty.*

Note that a part *is a set of elements of S, while a* partition *is a (particular) set of parts of S.*

A partition with 2 elements is called a bipartition. A partition with $k > 2$ elements is called a k-partition.

In this document, we will denote by $\mathfrak{P}(S)$ the set of all partitions of the set S, and given $k \in \mathbb{N}^*$, $\mathfrak{P}_k(S)$ the set of all k-partitions of the set S.

Definition 3 (Multiset)
A multiset (or bag) is an extension of the concept of set. We call multiset a pair (S, m) in which S is a set, and $m : S \to \mathbb{N}$ a function called multiplicity. An element $e \in S$ is said to appear $m(e)$ times in the multiset (S, m).

With any set S, we can associate the multiset $(S, \mathbb{1}_S)$, in which $\mathbb{1}_S : e \mapsto 1$ is the unit function.

In this document, we will denote the multiset

$$(\{e_1, ..., e_n\}, m) \text{ by } \{\underbrace{e_1, ..., e_1}_{m(e_1)}, ..., \underbrace{e_n, ..., e_n}_{m(e_n)}\}$$
$$\text{or } \{e_1, \overset{m(e_1)}{\dots}, e_1, ..., e_n, \overset{m(e_n)}{\dots}, e_n\}$$

Remark

As for sets, the order of the elements in a multiset does not matter.

Example

$\{4, 4, 7, 8, 9\}$ and $\{9, 8, 7, 4, 4\}$ designate the same multiset.

$\{4, 7, 8, 9\}$ is both a multiset and a set, because each element appears only once.

Remark

The usual definition of a partition does not allow the elements of a partition to be the empty set. However, considering (multi)sets such as $\{\varnothing, \overset{k-1}{\dots}, \varnothing, S\}$ as k-partitions will be more suitable for our problem. Therefore, in this document, a partition may have elements that are the empty set.

Example

The possible bipartitions of S are:

$$\Big\{\big\{\varnothing, \{a, b, c\}\big\}, \big\{\{a\}, \{b, c\}\big\}, \big\{\{b\}, \{a, c\}\big\}, \big\{\{c\}, \{a, b\}\big\}\Big\}$$

Definition 4 (Floor and Ceiling Functions)

Let $x \in \mathbb{R}$.

$\lfloor x \rfloor \in \mathbb{Z}$ is called floor and $\lceil x \rceil \in \mathbb{Z}$ is called ceiling if:

$$x - 1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x + 1 \tag{1}$$

Definition 5 (Computation Complexity Theory)

A decision problem is a problem that can be posed as a yes/no question of the input values.

P is the set of all problems that can be solved by a deterministic Turing machine using a polynomial amount of computation time.

NP is the set of all problems for which, given a candidate solution `s`, verifying whether `s` is a solution can be performed in polynomial time.

A problem $\mathtt{h}$ *is said to be NP-Hard when for every problem* $\mathtt{p}$ *in NP, there is a polynomial-time reduction from* $\mathtt{p}$ *to* $\mathtt{h}$*. This means that, if an algorithm* `Solver` *can solve* $\mathtt{h}$*, then we can solve* $\mathtt{p}$ *in polynomial-time excluding the time of* $\texttt{Solver}(\mathtt{h})$*.*

A problem is said to be NP-Complete if it is both NP and NP-Hard.

Remarks

- $\mathrm{P} \subset \mathrm{NP}$.
- $\text{NP-Complete} = (\mathrm{NP} \cap \text{NP-Hard})$.
- Answering the question $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$ remains a major challenge in computer science.
- Another formulation of a problem being NP-Hard is that it is at least as hard as the hardest problems in NP.
- By definition of NP-Hard and NP-Complete, the hardest problems in NP are the NP-Complete problems.
- $\text{NP-Complete} \subsetneq \text{NP-Hard}$: some problems, such as the halting problem, which aims at determining, given a computer program and an input, whether the program will terminate, is NP-Hard but not NP-Complete.

Pseudo-code Notations

Many algorithms are described in this manuscript, using pseudo-code. In order to improve readability, we specify in Table i the operators and notations that we will employ to define algorithms.

Notations Used in this Manuscript

Table ii introduces notations that we will use throughout the manuscript. The last column gives the section in which it is actually defined or the definition number, along with the corresponding page number.

Table i – Pseudo-code notations

Pseudo-code	**Signification**
$L \leftarrow [\,]$	L is an empty *ordered* set/multiset – or, in information science, a list
$L \leftarrow [e_1 \overset{n}{\ldots} e_n]$	L is a list of n elements defined implicitly, e_1 being the first element and e_n the last
$L[i]$	The ith element of L ($1 \leq i \leq \texttt{length}(L)$)
$L[-i]$	The ith element of L starting from the end
$L[i_a{:}i_b]$	The list of elements of L of indices i such that $i_a \leq i < i_b$
$\texttt{max}(L)$	Maximum value in L
$\texttt{min}(L)$	Minimum value in L
$\texttt{argmax}(L)$	Index of the first occurrence of $\texttt{max}(L)$ in L: $L[\texttt{argmax}(L)] = \texttt{max}(L)$ and $\Big(L[i] = \texttt{max}(L)\Big) \implies \Big(i \geq \texttt{argmax}(L)\Big)$
$\texttt{argmin}(L)$	Index of the first occurrence of $\texttt{min}(L)$ in L
$\texttt{length}(L)$	Number of elements in L
$\texttt{sum}(L)$	Sum of the elements in L
$L.\texttt{append}(e)$	Append e at the end of L
$L.\texttt{remove}(e)$	Remove the first occurrence of e from L
$L.\texttt{popFirst}()$	Remove and return the first element in L
$L.\texttt{popmFirst}()$	Remove and return the first m elements in L
$L.\texttt{popLast}()$	Remove and return the last element in L
$L.\texttt{renumber}(order)$	Permute L: $L[i]$ becomes $L[order[i]]$
$L.\texttt{reverse}()$	Same as $L.\texttt{renumber}([\texttt{length}(L), ..., 1])$
$L.\texttt{sortAscend}()$	Sort L in ascending order
$L.\texttt{sortDescend}()$	Sort L in descending order
$\texttt{argsortAscend}(L)$	Return a list whose ith element is the ith smallest element in L
$\texttt{argsortDescend}(L)$	Return a list whose ith element is the ith biggest element in L
$\texttt{random}(a, b)$	Return a random number x such that $a \leq x \leq b$

Table ii – Common notations

Notation		Problem parameters	ref.	p.
Ω	$\subset \mathbb{R}^3$	Geometric continuous bounded domain of space	sec. 1.2	15
M		Mesh	def. 6	16
γ	$\in \mathbb{N}^*$	Number of physical models/criteria	sec. 2.1.2	25
W	$: M \to (\mathbb{R}_+)^\gamma$	Weights associated with each cell of M	sec. 2.1.2	25
k	$\in \mathbb{N}^*$	Number of computation units and number of parts into which M will be partitioned		
t	$\in [0, 1]$	Tolerance – Maximum imbalance allowed ($t = 0$ means that we search for a perfectly balanced partition)	sec. 2.1.4	30
		Models		
G	$= (V_G, E_G)$	Graph associated with M; V_G is the set of its vertices and E_G the set of its edges	def. 16	38
H	$= (V_H, E_H)$	Hypergraph associated with M; V_H is the set of its vertices and E_H the set of its hyperedges	def. 12	35
V	$= M$ or V_G or V_H	General notation for the set of cells or vertices that we will partition. W is prolonged from M to V		
f	$: \mathfrak{P}(V) \to \mathbb{R}_+$	Cost function that should be minimized	sec. 2.1.3	27
Π	$= \{\Pi_1, ..., \Pi_k\}$	Partition of V	def. 2	5
		Constants & Variables		
n	$= \lvert V \rvert$	Number of cells or vertices		
i	$\in [\![1, n]\!]$	Index of a cell or a vertex		
p	$\in [\![1, k]\!]$	Index of a part in Π		
c	$\in [\![1, \gamma]\!]$	Index of a criterion		
v_i	$\in V$	ith element of V		
$w_{i,c}$	$= W(v_i)[c]$	Weight of v_i for criterion c		
Σ_c	$= \sum_{v_i \in V} w_{i,c}$	Total weight on V for criterion c		
$\Sigma_{c,p}$	$= \sum_{v_i \in \Pi_p} w_{i,c}$	Total weight on part Π_p for criterion c		

Part I

Problem, Model and State of the Art

Chapter 1

Numerical Simulations in High Performance Computing

How a Data Distribution Problem Emerges from Simulations in High Performance Computing

Contents

This chapter investigates how the run time of mesh-based multiphysics simulations can be minimized, when they are executed on a distributed memory machine. Section 1.1 first provides some reasons underlying the use of numerical simulations, and explains the challenge raised when such simulations are run on distributed memory architectures. We restrict our study to simulations based on a mesh, a complex entity for which a rough definition is given in Section 1.2. Then, we study typical examples of mesh-based simulations running on a distributed memory machine, in order to characterize how to minimize their duration. Section 1.3 handles the case of mono-physics simulations, and Section 1.4 extends the study to multiphysics simulations.

1.1 The Challenge Raised by Numerical Simulations on Distributed Memory Architectures

A number of scientific domains use numerical simulation to model complex phenomena. Indeed, making experiments to validate or invalidate any model (be it physical, biological, political, ...) can require rare and expensive resources. Simulation seeks to reproduce the conditions observed in the real world using a computer. This calls for new endeavors: firstly, being able to design a model that reproduces a situation as accurately as needed, and secondly, an algorithmic and implementation effort to create efficient and reliable software. By reliable, we mean that the software does what it was designed for. By efficient, we mean that it benefits as much as possible from the available resources, which implies that the software performs its task as fast as possible.

Examples

Meteorology and climatology use past data to simulate (at different time scales) probable future weather. Economics can test the effects of various policy actions to determine their consequences. Biology also tries to model cells and organs in order to understand better these biological systems.

The French Alternative Energies and Atomic Energy Commission (CEA) uses simulation to study, among others, renewable and nuclear energies. Simulation thus helps to design the installations that will serve in practice. Nevertheless, the CEA also combines simulation with experiments, in order to validate the models used. This is why the CEA has built in Bordeaux the Laser Mega Joule, whose objectives are detailed in CEA [2015].

Making a simulation running fast calls for a fast machine, meaning that it is capable of performing many operations per second. Over the years, machine computing speed has been mainly improved by increasing processor frequency. However, the energy consumption E of a central processing unit (CPU) is roughly proportional to the cube of its frequency. So, instead of one machine of frequency f of energy consumption $E_1 \propto f^3$, it can be much more profitable to use two of frequency $f/2$, that theoretically achieve the same computational performance for one fourth of the energy ($E_2 \propto (f/2)^3 \times 2 = f^3/4$). Therefore, nowadays, all high performance computing machines use several CPUs. Machines with several CPUs are called parallel machines (as opposed to sequential machines) and can have millions of computation units.

There is also a memory aspect behind the use of parallel machines. Indeed, the main memory capacity of a sequential machine is limited by physical factors, such as the chip size.

Efficient use of a sequential machine is already a difficult task: pipelining, data locality improvement, loop unrolling, and other techniques that speedup code execution are not only machine- and software-dependent, they also require expertise. This task is even more complex with parallel machines, leading to new constraints:

- designing a parallel algorithm that distributes evenly the work between computation units. Unfortunately, a perfectly balanced distribution rarely exists, because some sequential tasks cannot be parallelized. Nevertheless, in most simulations, equitably distributing tasks from at least a portion of the code can dramatically reduce the total run time;
- on parallel machines, computation units can have separated memories: we say that they are distributed memory machines, as opposed to shared-memory machines, in which all computation units have access to the same memory. On a distributed memory machine, when a computation unit needs a piece of data which is not in its memory, a communication must be performed, which may increase the run time of the application. Therefore, data distribution must take care of the induced communications.

Thus, parallelizing an application for a distributed memory machine is a challenging task. It implies designing a parallel algorithm that:

- distributes the work equitably between computation units;
- distributes the data so that each computation unit owns most of the data it needs, in order to limit communications.

Before going into further details on the efficiency of a simulation, we will clarify several points, such as what kinds of work and data need to be distributed. Indeed, as numerical simulations are used in numerous fields, they can handle various kinds of data.

Numerical simulations usually study physical phenomena occurring in space, which is a continuous domain. Meshes are entities created to model a continuous domain using a set of discrete elements. Indeed, in order to be represented by a computer, a continuous domain must be split into a finite number of elements, like the pixels of an image. The next section will give a rough definition of the complex entity that a mesh is.

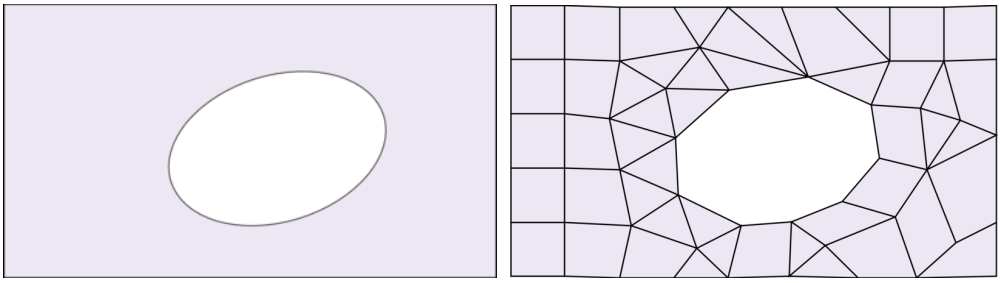
1.2 Space Discretization using a Mesh

In this section, we describe a model used by many simulations to approximate a continuous domain in space named Ω. This model constructs a set of geometrical *cells* that cover Ω. A *mesh* is a sophisticated entity, which will be simplified in this thesis as the set of all the cells, as stated in Definition 6. More information on meshes can be found in the book of Frey and George [1999].

Example

In $2D$, cells are usually triangles or quadrilaterals, as illustrated with the mesh on the right side of Figure 1.2.1, which discretizes the continuous domain Ω on the left side.

Figure 1.2.1 – Discretization of a bounded domain in $2D$



(a) A $2D$ bounded domain Ω in $2D$-space

(b) A mesh discretizing Ω using triangles and quadrilaterals

In $3D$, cells are usually tetrahedra, pyramids, triangular prisms or hexahedra.

Definition 6 (Mesh)

Let $\Omega \subset \mathbb{R}^3$ be a bounded domain (note that Ω is also a set). In the scope of this thesis, we call mesh a set:

$$M = \{m_q \text{ open bounded set}\} \text{ such that } \begin{cases} \bigcup \overline{m_q} = \Omega \\ \forall m_p \in M, m_p \neq m_q \implies m_p \cap m_q = \varnothing \end{cases}.$$

The m_q are called the cells of the mesh. $\overline{m_q}$ denotes the closed set of the open set m_q. Note that $m_q \subset \mathbb{R}^3$, so they have a dimension.

This thesis considers mesh-based numerical simulations running on distributed memory machines. As explained in the previous section, data are therefore distributed between computation units. Data are represented by the cells of the mesh, and the data distribution is usually determined by partitioning the mesh. A partition of a mesh, according to Definition 2 on page 5 and Definition 7, will attribute each cell to a single computation unit (every part of the partition will be attributed to a different computation unit).

Definition 7 (Partition of a Mesh)

A mesh is a set of cells. Therefore, we call partition of a mesh a partition of its cells.

A simulation computes physical values stored in a cell. Such data are called the state of a cell, and computing it usually involves the states of neighboring

cells. Neighboring cells are characterized in Definition 8, and Definition 9 introduces the boundary of a part, which serves to characterize the set of cells that needs to be communicated.

Definition 8 (Neighboring Cells)

Let D be the dimension of Ω ($D \leq 3$), and d an integer such that $0 \leq d < D$.

Two distinct cells are neighbors of dimension d if the intersection of their closed sets is a non-empty set of dimension d. More formally, cells m_p and m_q are neighbors if $m_p \neq m_q$, $\overline{m_p} \cap \overline{m_q} \neq \varnothing$, and $\dim(\overline{m_p} \cap \overline{m_q}) = d$.

The neighborhood of $m_p \in M$ (the set of all the cells that are neighbors of m_p) is denoted by:

$$\mathcal{N}(m_p) = \{m_q \in M : m_p \text{ and } m_q \text{ are neighbors}\}$$

Example

- In $2D$, we can consider neighbors by edge ($d = 1$) or by vertex ($d = 0$).
- In $3D$, we can also consider neighbors by face ($d = 2$).

Definition 9 (Boundary)

Let Π be a partition of M and Π_p a part of Π. Then we call inner boundary of Π_p the set:

$$\mathit{InnerBound}(\Pi_p) := \{m_p \in \Pi_p : \exists m_q \in \mathcal{N}(m_p), m_q \notin \Pi_p\} \ .$$

We also call outer boundary of Π_p the set:

$$\mathit{OuterBound}(\Pi_p) := \{m_q \notin \Pi_p : \mathcal{N}(m_q) \cap \Pi_p \neq \varnothing\} \ .$$

Note that $\mathit{InnerBound}(\Pi_p) \subset \Pi_p$, while $\mathit{OuterBound}(\Pi_p) \subset (M \setminus \Pi_p)$.

Finally, we call boundary of Π the set:

$$\mathit{Boundary}(\Pi) := \bigcup_{\Pi_p \in \Pi} \mathit{InnerBound}(\Pi_p) = \bigcup_{\Pi_p \in \Pi} \mathit{OuterBound}(\Pi_p) \ .$$

At the beginning of the simulation, each cell has an initial state. The next section describes how the next states are computed along a mono-physics simulation.

1.3 Minimizing the Run Time of a Parallel Mono-physics Simulation

We consider a mono-physics simulation based on a mesh M. Algorithm 1 is a simplified model of a parallel mono-physics simulation using k computation units in distributed memory. Our formulation does not aim to be applicable to all mono-physics simulations, but rather to illustrate a typical trend in numerical simulations.

Remark

The computation units usually communicate data in "messages". In order to specify communication schemes, such as when to send or receive messages, a commonly used message-passing standard is MPI. MPI stands for "Message Passing Interface", and its current version is fully described in MPI [2015].

The objective of the simulation is to compute the states for every cell of M from time $\tau = 0$ to time $\tau = \tau_{end}$. The simulation ends when the states for every cell of M up to time τ_{end} have been computed. We call simulation time the overall time taken to compute all the states until the last time step, which corresponds to the duration of the function `Simulate_mono`.

The state of $m \in M$ at time τ is denoted by $X_\tau(m)$. The initial states X_0 and the initial time step $d\tau$ are given as input. Computing the state of m at time $\tau + d\tau$ is performed by the function `f`. `f` takes as argument the current time τ, the current time step $d\tau$, m and its neighborhood $\mathcal{N}(m)$, and the previous states of the cells X_τ.

Algorithm 1 A simulation computes the state X for all the cells in the mesh.

procedure `Simulate_mono`(M, k, X_0, $d\tau$, τ_{end})
Ensure: Fills the structure X with the state of every cell for every time step.
  $\Pi \leftarrow$ `Partition`(M, k) # Can sometimes be done in a preprocessing step
  `Distribute`(M, Π) # Communicate one part to every computation unit
  $\tau \leftarrow 0$
  while $\tau < \tau_{end}$ **do**
    for in parallel $p \in [\![1, k]\!]$ **do**
      for $m \in \Pi[p]$ **do**
        $X_{\tau+d\tau}(m) \leftarrow$ `f`$(\tau, d\tau, m, \mathcal{N}(m), X_\tau)$
      end for
    end parallel for
    # Wait for all units to complete, then communicate the boundary cell states
    `Synchronize`$(\{X_{\tau+d\tau}(m), m \in Boundary(\Pi)\})$
    $d\tau \leftarrow$ `UpdateTimeStep`$(d\tau, M)$ # $d\tau$ may depend on the current cell states
    $\tau \leftarrow \tau + d\tau$
  end while
end procedure

First, the `Partition` routine partitions the mesh into k parts. Then, the `Distribute` routine sends a part to every computation unit. In addition, we assume that each unit gets the states of the neighbors of its boundary cells.

At each time step, unit p computes the new states for cells in part $\Pi[p]$ using function f. In order to compute the new state of a cell m, f notably needs the states of the neighbors of m. Then, the `Synchronize` routine waits

for all computation units to complete their work. When the new states for all cells have been computed, the `Synchronize` routine updates the boundary cell states on all computation units. Finally, $d\tau$ is updated.

Remark

In some cases, the partition can be computed before running the simulation. Hence, the same partition will be used for different settings of the same simulation, and the time of the `Partition` function will not be counted in the simulation time.

Example

We consider a numerical simulation of 3 time steps, and compare on Figure 1.3.1 its sequential execution with a parallel execution with 2 computation units.

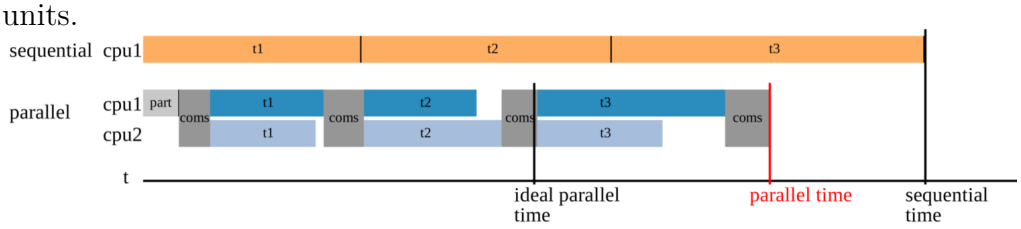


Figure 1.3.1 – Comparison between a sequential and a parallel execution.

The top line (in orange) is the sequential execution. t_1, t_2 and t_3 correspond to the duration of each time step. In sequential, the duration of a time step is the time needed to compute the states of all the cells in M, because the single computation unit gets all the work.

The two lines below (in blue) represent a parallel execution. Each line corresponds to a computation unit: cpu_1 and cpu_2. The ideal parallel time is the sequential time divided by the number of computation units, so half the sequential time t_{seq} in this example. Nevertheless, the actual parallel time t_{par} ends up to be, in this example, $t_{par} = 0.80 \times t_{seq}$ because of the overheads induced by the parallel execution.

First, cpu_1 computes a partition of the mesh (in the "part" sequence in gray that corresponds to the call to the `Partition` routine in Algorithm 1, which could also be performed in parallel). Then, a "coms" sequence (also in gray) distributes the parts across the computation units (this is the `Distribute` call in Algorithm 1).

Then, computation of the cell states begins. Computations are represented by the "t_1", "t_2" and "t_3" sequences in blue. Considering a time step, with 2 computation units, the ideal parallel time is half the sequential time. However, this optimum can be reached only if the work is evenly balanced between cpu_1 and cpu_2 and if no communication is required. Indeed, the first unit to finish has to wait for the other before synchronizing, hence some idle time. Then, the `Synchronize` call communicates the boundary cells states in the "coms" sequence, delaying the end of the simulation as well.

In the example, the parallel execution enables one to reduce the run time of the simulation, compared with the sequential execution. However, it is still quite far for the ideal parallel time. The gap between ideal and real parallel time is measured by the parallel efficiency.

Definition 10 (Speedup and Parallel Efficiency)
We consider a simulation that runs in a time t_{seq} on a sequential machine and in a time $t_{par}(k)$ on a parallel machine with k computation units. Then, we call speedup $Sp(k)$ and parallel efficiency $\eta(k)$:

$$Sp(k) = \frac{t_{seq}}{t_{par}(k)} \qquad \eta(k) = \frac{Sp(k)}{k}$$

The parallel efficiency is equal to 1 when the elapsed parallel time is equal to the ideal parallel time.

Example
In the previous example, for which the speedup is $Sp = 1.3$, the parallel efficiency is $\eta = 0.65$.

The aim of this thesis is to draw the parallel efficiency as close as possible to 1. To do so, the distribution of the cells in the mesh must minimize:

- the workload imbalance between computation units, that induces idle time before the call to the `Synchronize` function;
- the `Synchronize` time, which is the time required to communicate the boundary cells;
- the `Partition` time.

These considerations stem for a mono-physics simulation. The next section considers the case of multiphysics simulations.

1.4 Minimizing the Run Time of a Parallel Multiphysics Simulation

The particularity of this thesis is that it considers multiphysics simulations. Algorithm 2 is a simple model of a simulation that couples two physical models. Some simulations can be far more complex, yet the principle remains the same: the different physical models are coupled. Also, the example may be transposed easily to more than two physical models.

Remark
Multiphysics simulations commonly couple fluid/structure, thermal/mechanical or electric/thermal interactions. For example, the deformation of an aircraft wing during flight is a fluid/structure interaction, while the asphalt

deformation on hot days is a thermal/mechanical interaction.

Algorithm 2 A model of a simulation computing variables X and Y corresponding to two different physical models.

procedure `Simulate_multi`(M, k, X_0, Y_0, $d\tau$, τ_{end})
  $\Pi \leftarrow$ `Partition`(M, k) # Can sometimes be done in a preprocessing step
  `Distribute`(M, Π) # Communicate one part to each computation unit
  $\tau \leftarrow 0$
  while $\tau < \tau_{end}$ **do**
    for in parallel $p \in [\![1, k]\!]$ **do**
      for $m \in \Pi[p]$ **do**
        $X_{\tau+d\tau}(m) \leftarrow \mathtt{f}(\tau, d\tau, m, \mathcal{N}(m), X_\tau \quad , Y_\tau)$
      end for
    end parallel for
    # Wait for all units to complete, then communicate the boundary cell states
    `Synchronize`$(\{X_{\tau+d\tau}(m), m \in \mathit{Boundary}(\Pi)\})$
    for in parallel $p \in [\![1, k]\!]$ **do**
      for $m \in \Pi[p]$ **do**
        $Y_{\tau+d\tau}(m) \leftarrow \mathtt{g}(\tau, d\tau, m, \mathcal{N}(m), X_{\tau+d\tau}, Y_\tau)$
      end for
    end parallel for
    # Wait for all units to complete, then communicate the boundary cell states
    `Synchronize`$(\{Y_{\tau+d\tau}(m), m \in \mathit{Boundary}(\Pi)\})$
    $d\tau \leftarrow$ `UpdateTimeStep`$(d\tau, M)$ # $d\tau$ may depend on the current cell states
    $\tau \leftarrow \tau + d\tau$
  end while
end procedure

Phase X Phase Y

In Algorithm 2, the two variables corresponding to the two physical phenomena are X and Y. The initial states for all cells are X_0 and Y_0 and the initial time step is $d\tau$. We search for the states of all cells for each time step up to $X_{\tau_{end}}$ and $Y_{\tau_{end}}$. The algorithm uses the same notations as the mono-physics case of Algorithm 1 on page 18. The differences are firstly, that the function `f` needs the states of m and its neighborhood $\mathcal{N}(m)$ for both the variables X and Y, and secondly, that there is another function `g` that computes the new states at time $\tau + d\tau$ for the variable Y.

The variables X and Y are coupled: to compute $X_{\tau+d\tau}$, state Y_τ is needed, and to compute state $Y_{\tau+d\tau}$, state $X_{\tau+d\tau}$ is needed. This is why the computation phase of $Y_{\tau+d\tau}$ does not begin before that of $X_{\tau+d\tau}$ finishes. Thus, in parallel executions, *load balancing must be achieved for the computations of both physical models*.

Remark

There are other formulations of Algorithms 1 and 2. For example, the `Partition` routine can be called more than once. In the mono-physics case, it would be called at some time step when the load balance is not satisfied. In the multiphysics case, another possibility is to call it after the computation phase of each physical model, balancing the workloads of the computation phases one at a time.

However, in addition to the time needed to compute the new partition, the new partition also needs to be distributed to the computation units. As the `Distribute` routine can lead to a large communication overhead, our goal is to be able to balance the workloads for all phases at the same time.

Conclusion

To sum up, this thesis is aimed at minimizing the run time of multiphysics simulations that use meshes and run in parallel on distributed memory architectures. The following Problem 1 synthesizes the implications of bringing the simulation time closer to the ideal parallel time.

Problem 1 (Multi-objective Mesh Partitioning)

Design an algorithm that, given a mesh M, returns a partition of M such that:

- *the partition balances the workloads between computation units for the computation phases of* every *physical moedel;*
- *the partition minimizes the communication time;*
- *the algorithm computes the partition in a minimum amount of time.*

This problem appears to be a multi-objective partitioning problem. However, some points need to be clarified: how are the so-called workloads attributed to each computation unit quantified? How can we estimate the communication time induced by a partition of the mesh? The next chapter will discuss several models that are commonly used to address Problem 1.

Chapter 2

Models of the Minimum Run Time of a Multiphysics Simulation

Contents

This chapter formulates several problems modeling how to minimize the run time of a multiphysics simulation running on a distributed memory architecture. The previous chapter has already formulated conditions to minimize the simulation time, namely balancing the workload between computation units and minimizing the communication and partitioning times. In Section 2.1, we introduce models of the workloads and the communication time, and we explain why we do not consider the partitioning time in our models.

Handling several objectives can lead to ambiguities over the choice of the solution, so the multi-objective formulation is transformed into a constrained mono-objective problem. The most straightforward formulation leads to the multi-criteria mesh partitioning problem, which is stated in Section 2.2. Nevertheless, other models exist: Section 2.3 relies on a hypergraph to represent the topology of the mesh, while Section 2.4 relies on a graph. Graphs model differently than hypergraphs and meshes the communication cost of a partition, in a simpler way.

Section 2.5 formulates the vector-of-numbers partitioning problem, which is a subproblem of the multi-criteria mesh partitioning problem. Indeed, solving a vector-of-numbers partitioning problem amounts to finding a solution (not necessarily optimal) to a multi-criteria mesh partitioning problem. This thesis will define and experiment on two partitioning algorithms for vector-of-numbers. The mono-criterion version of the vector-of-numbers partitioning problem, the number partitioning problem, has been well studied (as we will show in Section 3.2).

Finally, Section 2.6 gives another formulation of the multi-criteria mesh partitioning problem, using fitness landscapes. This formulation provides an abstracted view of the problem, especially useful when considering local optimization techniques. We introduce the terminology of fitness landscape analysis, and these techniques will be further used in Chapters 5 and 8.

2.1 Modeling the Simulation Time

The present section gives several approximations that will simplify and reformulate the global objectives given in the previous chapter. We remind that these objectives were, in order to minimize the run time of a multiphysics simulation, to 1) balance the workloads between computation units; 2) minimize the communication time; and 3) minimize the partitioning time.

If we remind the example given while illustrating a multiphysics execution in Algorithm 2 on page 21, the simulation time is (see the notations in Table 2.1.1):

$$t_{sim} = t_{part} + \sum_{\tau=0}^{\tau_{end}} \sum_{\phi} \left(\max_{cpu} t_{cmp}(cpu, \phi, \tau) + t_{com}(\phi, \tau) \right) \tag{2.1}$$

The following sections will discuss each term appearing in Equation 2.1.

2.1.1 Partitioning Time

First, when the number of time steps increases, the partitioning time is less and less preponderant. Actually, the choice of the partitioning algorithm is a trade-off between the quality of the partition found and the time to find a

Table 2.1.1 – Time notations

t_{sim}	Simulation time
t_{part}	Partitioning time
$t_{cmp}(cpu, \phi, \tau)$	Computation time of computation unit cpu for the physical model ϕ at time step τ
$t_{com}(\phi, \tau)$	Communication time after computations for the physical model ϕ for time step τ

partition. This choice must be made by the user, who should know the duration of the simulation and thus which algorithm to select.

We chose not to consider the partitioning time in our models. However, in the experiments, in Chapter 10, we will compare the run times of every algorithm used. We will also compare the run time of our implementation with the run times of existing partitioning tools.

2.1.2 Computation Time

This section focuses on the $\max_{cpu} t_{cmp}(cpu, \phi, \tau)$ term in Equation 2.1, which is the duration of the computation phase for the physical model ϕ at time step τ. Estimating this term requires knowing, when assigning a cell of the mesh to a computation unit, the workload of this cell for the physical model ϕ and time step τ. This workload, which depends on the given cell, physical model and time step, is provided by the user.

The workload of a cell for one phase and time step will be represented as a weight given to this cell for this phase and time step. In this thesis, we consider that the input weights are accurate estimations of the workload.

Besides, a usual assumption is that the provided weights do not depend on the time step, so $t_{cmp}(cpu, \phi, \tau) = t_{cmp}(cpu, \phi)$. In practice, the imbalance is measured after some time steps, and a new partition is computed when the imbalance becomes greater than an imposed threshold. Again, such a policy is defined by the user.

To sum up, in order to be able to balance the workload across computation units for every physical model, the user must give to each cell a vector of weights, each component representing the amount of computations of one physical model. Such a component is called a *criterion*. The weights do not depend on the time step. So, if we denote by $\gamma \in \mathbb{N}^*$ the number of criteria, we model the weights with a function $W : M \to (\mathbb{R}_+)^\gamma$, such that if $m \in M$ and $c \in [\![1, \gamma]\!]$, then $W(m)[c]$ is the workload of m for the cth physical model.

Remark

Walshaw *et al.* [2000] describe another type of multiphase mesh simulation, in which the cells usually belong only to one phase (or, equivalently, their weights are non-null only for one criterion). When a cell belongs to several phases, one phase is considered as preponderant, and the weights for the other phases are set to 0.

Our model needs to be more general, since in our simulations it is not always possible to choose a preponderant criterion for a cell.

If we denote by $\Sigma_{c,p}$ the weight of part Π_p for criterion c, the goal to balance the workloads becomes to find a partition Π^{best} of M such that, for any criterion c, the maximum workload among all parts is minimized:

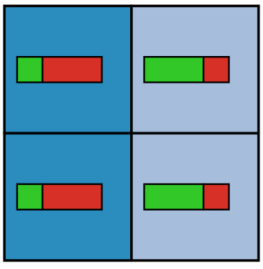
$$\Pi^{best} \text{ is such that } \forall c \in [\![1,\gamma]\!], \max_{\Pi_p \in \Pi^{best}} \Sigma_{c,p} = \min_{\Pi \in \mathfrak{P}_k(M)} \left(\max_{\Pi_p \in \Pi} \Sigma_{c,p} \right) . \tag{2.2}$$

Remark

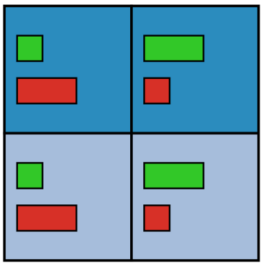
As explained in Section 1.4, vertex weights should not be summed, else partitions that are imbalanced could be considered balanced, as illustrated by the following example.

Linearizing the vertex weights means that the computation weight of $m \in M$ becomes $W(m) = \sum_{c \in [\![1,\gamma]\!]} W(m)[c]$. Let us consider a mesh with 4 cells, with two criteria. The two figures below are two bipartitions of this mesh. Cell weights are represented by the red and green bars within each cell. The length of a bar is proportional to the weight of the cell for this criterion. The color of a cell represents the part it belongs to.

A partition balanced for the linearized weights



A partition balanced for each criterion



The partition on the left balances the linearized weights, but not the weights for each criterion, while the partition on the right is balanced for each criterion.

2.1.3 Communication Time

In the current section, we explain how we model the communication time of multiphysics simulation. Regarding communication time modeling, Hendrickson [1998] early warned the community that it is a very complex task, involving significant approximations. Then, Hendrickson and Kolda [2000] gave more precisely the limitations of several models and showed how to fix some of them. Cai and Bouhmala [2007], Deveci *et al.* [2015b] and Selvakkumaran and Karypis [2006] have proposed several ways to model the cost of communications induced by a data distribution. All of them considered this cost as a multi-objective function. Hereafter, we do not describe explicitly these models, but rather explain the various assumptions that we make and that can be encountered in these models.

Firstly, in Equation 2.1, the simulation time appears as the sum of the times used for computation and for communication. This can already be a simplification, because one can overlap communications with computations. Overlapping raises algorithmic and implementation challenges, and is not always feasible. So, in this document, we consider that the communication time has to be added to the computation time.

Secondly, for the same reason as for the computation time, we consider that the communication time is the same at each time step.

Thirdly, estimating the communication time implies to know the communications induced when a cell is assigned to a computation unit. As explained in Chapter 1, computing the new state for a cell m requires the states of the neighbors of m. This is why, before each computation phase, a communication step is performed, as illustrated in the `Simulate` functions of Algorithms 1 on page 18 and 2 on page 21. In this thesis, we consider that a communication step consists in sending the data on the boundary cells.

Therefore, similarly to the weights the user must provide to model the workload of each cell, he also must input a communication cost for each cell. The cost for one cell is expressed as a (scalar) weight on this cell, and represents the time to send data on this cell to another computation unit. We model the weights with a function $W_{com} : M \to \mathbb{R}_+$ that associates with every cell its communication cost.

Fourthly, using a simplification of the model by Forouzan [2007], the time to send a message depends on the latency λ (that Forouzan names "propagation time"), the bandwidth of the network β, the size of the message w_{msg}, and the maximum size of a message w_{max}. The time to send a message is:

$$t_{msg} = \left(\lambda + \frac{w_{max}}{\beta}\right) \times \left\lfloor \frac{w_{msg}}{w_{max}} \right\rfloor + \left(\lambda + \frac{w_{msg} \bmod w_{max}}{\beta}\right)$$

In order to simplify the computation of the total communication cost of a partition, common models consider that latency is negligible, which entails

that $t_{msg} = \frac{w_{msg}}{\beta}$: the time to send a message becomes proportional to its size. The communication cost models used in practice and that we define in the following paragraphs rely on this approximation.

In this document, we will denote by f the communication cost function that we want to minimize. Depending on the communication scheme used and of its implementation, f can be expressed in various ways. We will now describe possible formulations for f (that are based on the previous approximations).

Given two computation units, we explained in the previous chapter that they need to exchange data on their cells in their common border. We will denote by $w_{\Pi_p \to \Pi_q}$ the amount of data that the unit in charge of part Π_p will need to send to the unit in charge of part Π_q:

$$w_{\Pi_p \to \Pi_q} = \sum_{m \in \Pi_p \cap OuterBound(\Pi_q)} W_{com}(m) \ .$$

A first way to model f is to consider that the communications are serialized (executed one after another). In this case, the communication time would be:

$$t_{com} = \frac{f_{tot}(\Pi)}{\beta} = \frac{1}{\beta} \sum_{\Pi_p \in \Pi} \sum_{\Pi_q \in \Pi, \Pi_q \neq \Pi_p} w_{\Pi_p \to \Pi_q} \ .$$

A second way considers that all units may communicate their border simultaneously. In this case, the communication time would be

$$t_{com} = \frac{f_{\max}(\Pi)}{\beta} = \frac{1}{\beta} \max_{\Pi_p \in \Pi} \sum_{\Pi_q \in \Pi, \Pi_q \neq \Pi_p} w_{\Pi_p \to \Pi_q} \ .$$

A third way is to consider that communications between couples of units can occur at the same time, as in the following example. Note that estimating the communication time is more complex, because it involves computing many maxima.

Example

We consider $k = 4$ units, and we will represent with $(p \leftrightarrow q \quad || \quad r \leftrightarrow s)$ the fact that units cpu_p and cpu_q exchange data at the same time as cpu_r and cpu_s. Then, the scheme:

1. $(1 \leftrightarrow 2 \quad || \quad 3 \leftrightarrow 4)$
2. $(1 \leftrightarrow 3 \quad || \quad 2 \leftrightarrow 4)$
3. $(1 \leftrightarrow 4 \quad || \quad 2 \leftrightarrow 3)$

performs simultaneous pairwise communications. For this scheme, the communication time is:

$$\begin{aligned} t_{com} = 1/\beta \times f_{simultaneous(\Pi)} &= \max(w_{\Pi_1 \leftrightarrow \Pi_2}, w_{\Pi_3 \leftrightarrow \Pi_4}) \\ &+ \max(w_{\Pi_1 \leftrightarrow \Pi_3}, w_{\Pi_2 \leftrightarrow \Pi_4}) \\ &+ \max(w_{\Pi_1 \leftrightarrow \Pi_4}, w_{\Pi_2 \leftrightarrow \Pi_3}) \ . \end{aligned}$$

Remark

A more accurate model should take into account that the time to communicate between units cpu_p and cpu_q and between cpu_p and cpu_r may vary, depending on their "distance" (which is the number of switches that need to be traversed), the buffer mechanism, and other mechanisms.

Some softwares, such as Scotch (Pellegrini and Roman [1996b]), provide mechanisms to map a partition to a given topology. More recently, Deveci *et al.* [2015a] have proposed new methods to improve such mapping algorithms.

Example

This example illustrates the difference between each formulation of the communication cost f, with $W_{com}(m) = 1$ for each $m \in M$.

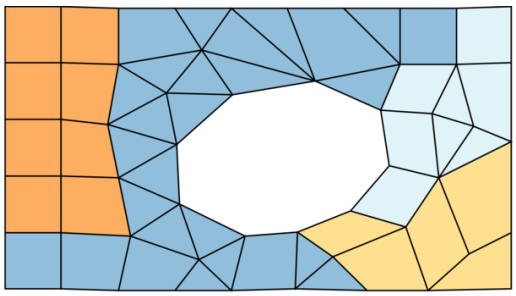


Figure 2.1.3 – A 4-partition of a mesh (one color corresponds to one part)

The following Table 2.1.2 counts the bordering cells between parts. It displays in line p and column q the number of cells in part Π_p that have at least one neighbor in part Π_q, which corresponds to the number of cells that cpu_p needs to send to cpu_q. The last column shows the total number of cells that each part has to send. The sum of the last column is f_{tot}, and its maximum is f_{max}.

Table 2.1.2 – Counts in the cell on line p and column q the number of cells that unit cpu_p has to send to unit cpu_q

	orange	blue	light blue	yellow	*send*
orange		5	0	0	5
blue	6		2	2	10
light blue	0	2		2	4
yellow	0	2	3		5
					$f_{tot} = 24, f_{max} = 10$

If we want to estimate the communication cost if the communications would be performed simultaneously, using the communication scheme:

1. (orange ↔ blue || light blue ↔ yellow)
2. (orange ↔ light blue || blue ↔ yellow)
3. (orange ↔ yellow || blue ↔ light blue) ,

then the communication volume would be $f_{simultaneous}(\Pi) = \max(5+6, 2+3) + \max(0+0, 2+2) + \max(0+0, 2+2) = 19$.

Remarks

f_{tot} is commonly called the communication volume. As stated by Hendrickson and Kolda [2000], f_{tot} works well in practice for the mesh partitioning problem, because the communication volume is usually equitably distributed across units.

Nevertheless, the maximum communication volume f_{max} was used in a subchallenge of the 10th DIMACS Challenge on Graph Partitioning and Graph Clustering, as reported by Bader *et al.* [2013].

Conclusion. In this section, we have described several approximations enabling to estimate the communication cost of a partition. This communication cost is represented by a function f, and the goal to minimize the communication time consists in finding a partition Π^{best} of M such that:

$$f(\Pi^{best}) = \min_{\Pi \in \mathfrak{P}_k(M)} f(\Pi) \ . \tag{2.3}$$

It is up to the user to choose a formulation adapted to his simulation and architecture. Thereafter, we will use $f = f_{tot}$, which has been, as reported by Buluç *et al.* [2015], adopted as a kind of standard. However, many of the algorithms that we will present further in this document can be adapted to various formulations of f.

2.1.4 From a Multi-objective to a Constrained Mono-objective Formulation

Equations 2.2 and 2.3 devised in the previous sections give us a multi-objective problem, which amounts to finding a k-partition Π^{best} of M such that:

$$\begin{cases} \forall c \in [\![1, \gamma]\!], \max_{\Pi_p \in \Pi^{best}} \Sigma_{c,p} = \min_{\Pi \in \mathfrak{P}_k(M)} \left(\max_{\Pi_p \in \Pi} \Sigma_{c,p} \right) \ , \\ f(\Pi^{best}) = \min_{\Pi \in \mathfrak{P}_k(M)} f(\Pi) \ . \end{cases} \tag{2.4}$$

This problem is unlikely to have a solution, because usually, finding a balanced partition leads to a high communication volume, and conversely.

Example

The partition $\{\varnothing, \overset{k-1}{\ldots}, \varnothing, M\}$ always minimizes the communication cost, but it also maximizes the imbalance.

A solution minimizing both objectives is unlikely, but there are partitions that minimize either the communication cost, or the imbalance. Between two partitions, the first of which has larger imbalance but smaller communication cost than the second, which one is the best suited? The answer is at least architecture-dependent, so it is not possible to decide in the general case. Therefore, a ranking must be defined to compare partitions. To rank partitions, there are two possibilities: linearize the objectives or give priority to some of them.

The first possibility, to linearize the objectives, can be used for mono-criterion partitioning. For example, Buluç *et al.* [2015] define the conductance of a partition:

$$f_\sigma(\Pi) = \frac{f_{tot}(\Pi)}{\min\limits_{\Pi_p \in \Pi} \Sigma_{c_1,p}}$$

f_σ prevents parts from being underweighted. There are several drawbacks with this formulation. Firstly, it prevents a part from being empty, although a partition with empty parts may minimize the imbalance (for example, for the weights $W = \{2, 1, 1\}$, the 3-partition $\{\{2\}, \{1, 1\}, \varnothing\}$ is of minimal imbalance). This drawback can be corrected by changing f_σ to $f'_\sigma = f_{tot}(\Pi) \times \max_{\Pi_p \in \Pi} \Sigma_{c_1,p}$. Secondly, this objective function may favor partitions that are very imbalanced but of very small communication cost. For our multiphysics simulations, in which obtaining balanced partitions is essential, linearization is not feasible.

The second possibility to rank the partitions is to define an order on the objectives. For example, we can choose to prioritize the communication cost, in which case the solution to our problem is the partition of minimal communication cost. If two partitions are of same communication cost, the "best" one is that of minimal imbalance.

This formulation would always lead to consider the partition $\{\varnothing, M\}$ as optimal (because of null communication cost), yet it is of maximal imbalance. So, it is preferable to prioritize the imbalance objectives. However, in order to be able to return partitions of smaller communication cost, in practice, the relation order is transformed into constraints.

Therefore, the classic model transforms a multi-objective problem into a mono-objective problem, by adding some balance constraints. The user is asked to provide an imbalance tolerance t, and among all partitions whose imbalance is smaller than t, we search for the one that minimizes the communication cost. As this maximum imbalance must be respected for all criteria, we call this the *multi-criteria mesh partitioning problem*.

This formulation discriminates two partitions that would have, for one, a smaller communication cost, and, for the other, a smaller imbalance. In this case, the one with a smaller communication cost is preferred if and only if its imbalance is smaller than t. Thus, the user can adjust the tolerance to his need: a tight tolerance if computation time is preponderant, or a loose tolerance if communication cost must be minimized at all cost.

Remark

Of course, when two valid partitions have the same communication cost, other criteria might be added to discriminate between them, but both of them would be acceptable (which was not the case with the multi-objective formulation).

Conversely, using this formulation, it is also possible that, for a given tolerance, there is no solution. In this case, the user must increase the tolerance.

The previous chapter explained the challenges raised by multiphysics simulation in parallel computing. Until now, the current chapter justified how the multi-criteria mesh partitioning model emerged in order to minimize the run time of such simulations. The following section will define formally the multi-criteria mesh partitioning problem.

2.2 Modeling a Distributed Execution with the Multi-criteria Mesh Partitioning Problem

This section defines the multi-criteria mesh partitioning problem that this thesis addresses. Notations for the following definitions are specified in Table 2.2.1.

Definition 11 (Imbalance)

Using the notations of Table 2.2.1, the imbalance of part Π_p *for criterion* c *is:*

$$imb_c(\Pi_p) = \frac{\Sigma_{c,p} - \frac{\Sigma_c}{k}}{\frac{\Sigma_c}{k}}$$

And the imbalance of partition Π *is:*

$$imb(\Pi) = \max_{c \in [\![1,\gamma]\!]} \max_{\Pi_p \in \Pi} imb_c(\Pi_p)$$

Table 2.2.1 – Mesh notations

Notation		Problem parameters	ref.	p.
M		Mesh	def. 6	16
k	$\in \mathbb{N}^*$	Number of computation units or parts in which we partition M		
γ	$\in \mathbb{N}^*$	Number of physical models/criteria	sec. 2.1.2	25
W	$: M \to (\mathbb{R}_+)^\gamma$	Computation weights associated with every cell in M	sec. 2.1.2	25
t	$\in \mathbb{R}_+$	[Tolerance] Maximum imbalance allowed ($t = 0$ means that we search for a perfectly balanced partition)	sec. 2.1.4	30
f	$: \mathfrak{P}(V) \to \mathbb{R}_+$	Cost function that we want to minimize	sec. 2.1.3	27
W_{com}	$: M \to \mathbb{R}_+^*$	Communication weights associated with every cell in M	sec. 2.1.3	27
Π	$= (\Pi_1, ..., \Pi_k)$	Partition of M	def. 2	5
		Constants & Variables		
n	$= \lvert M \rvert$	Number of cells		
p	$\in [\![1, k]\!]$	Index of a part in Π		
c	$\in [\![1, \gamma]\!]$	Index of a criterion	sec. 2.1.2	25
Σ_c	$= \sum_{m \in M} W(m)[c]$	Total weight on M for criterion c	sec. 2.1.2	25
$\Sigma_{c,p}$	$= \sum_{m \in \Pi_p} W(m)[c]$	Total weight on part Π_p for criterion c	sec. 2.1.2	25

Remarks

- $-(k-1) \leq imb_c(\Pi_p) \leq k-1$.
- imb is usually given as a percentage.
- $imb(\Pi) = 0\%$ means that all parts are perfectly balanced for all criteria.
- $imb_c(\Pi_p) = k-1$ means that part Π_p is attributed all the work for criterion c.
- $\forall c, \sum_p imb_c(\Pi_p) = 0$.

Problem 2 (Multi-criteria Mesh Partitioning)
Given a mesh M, weights $W : M \to (\mathbb{R}_+)^\gamma$, a number of parts $k \in \mathbb{N}^*$, a tolerance $t \in \mathbb{R}_+$ and a communication cost function f, the multi-criteria mesh partitioning problem amounts to finding a partition Π^{best} such that:

- $\forall c \in [\![1, \gamma]\!], \max_{p \in [\![1,k]\!]} imb_{c,p}(\Pi^{best}) \leq t \quad ;$ *(constraints)*
- $f(\Pi^{best}) = \min_{\Pi \in \mathfrak{P}_k(M),\, imb(\Pi) \leq t} f(\Pi) \quad .$ *(objective)*

Problem 2 formulates the multi-criteria mesh partitioning problem. Note that the "s" in "constraints" is underlined: it highlights a key point of this thesis, which addresses a *multi*-constraints problem. As we will see in Chapter 4, a number of algorithms were designed to address the mono-criterion partitioning problem, but few studies consider the multi-criteria version. Finally, as stated in Section 2.1.3, we consider that $f = f_{tot}$, which sums up the total communication weight.

In the mesh partitioning problem as formulated in Problem 2, the geometric coordinates of the mesh do not appear anywhere. Indeed, when formulating the communication cost function, in our case, a cell requires the data of its neighbors, so the relation between two cells is only topological. This is why the mesh partitioning problem is often formulated using topological entities such as hypergraphs and graphs. They also define the communication cost in a simpler way. The next two sections will successively explain how a mesh can be modeled with a hypergraph and with a graph.

2.3 Emphasis on the Mesh Topology in the Multi-criteria Hypergraph Partitioning Problem

Hypergraphs were first introduced by Catalyurek and Aykanat [1996] and Catalyurek and Aykanat [1999], in order to correctly encode the minimization of the communication volume in the context of sparse matrix partitioning for the parallelization of the sparse matrix-vector (SpMV) operation. As the mesh partitioning problem relies only on the mesh topology, a mesh can also be modeled with a hypergraph.

In this section, we first define what a hypergraph is, and then show how a hypergraph can model a mesh. Finally, we formulate the multi-criteria hypergraph partitioning problem, which is, for our communication cost function, equivalent to the mesh partitioning problem.

Remark

The partitioning software Mondriaan, whose implementation is detailed by Bisseling and Meesen [2005], specializes in sparse matrix partitioning. For example, Pelt and Bisseling [2014] presents a matrix partitioning algorithm relying on the hypergraph model.

Definition 12 (Hypergraph)

A hypergraph H is a pair $H = (V, E)$ where V is a set of elements called vertices *and E is a multiset of non-empty subsets of V. The elements of E are called* hyperedges. *(For what a multiset is, see Definition 3 on page 6.)*

With each mesh, we can associate a hypergraph. V corresponds directly to the set of cells: to each cell corresponds a unique vertex. The hyperedges represent the neighboring relations between the cells. With each vertex/cell is associated a hyperedge that links the cell to all of its neighbors. Definition 13 sums up how to construct the hypergraph associated with a mesh.

Definition 13 (Dual Hypergraph of a Mesh)

Given a mesh M, we define $H = (V, E)$ such that:

- $V = M$
- *E is the multiset* $\left\{e_m = \{m\} \cup \mathcal{N}(m), \text{ where } m \in M\right\}$.

Thus, to each cell $m \in M$ corresponds a unique vertex, that we denote by v_m, and a unique hyperedge, that we denote by e_m. Reciprocally, to each vertex $v \in V$ (respectively hyperedge $e \in E$) corresponds a unique cell that we denote by m_v (respectively m_e). Given $e \in E$, the vertex v_{m_e} is called the center *of e.*

Let $W : M \to (\mathbb{R}_+)^\gamma$ and $W_{com} : M \to \mathbb{R}_+$ be the computation and communication weights of M, then we define the weight functions on H:

- $\forall v \in V, W(v) = W(m_v)$
- $\forall e \in E, W_{com}(e) = W_{com}(m_e)$.

Example

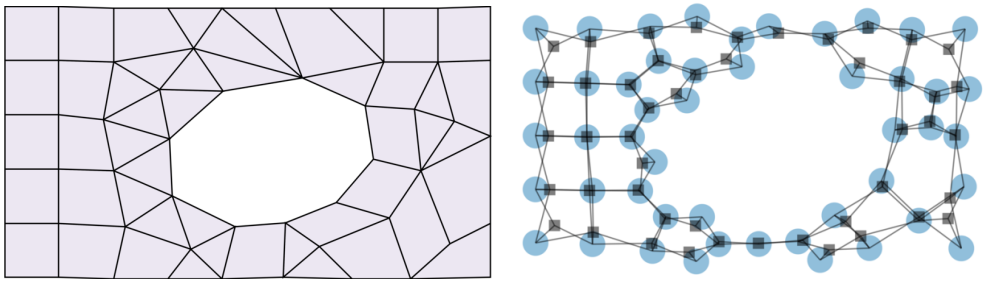


Figure 2.3.1 – A mesh and its corresponding hypergraph

The mesh is represented on the left, and the corresponding dual hypergraph on the right. To each cell of the mesh corresponds a vertex (symbolized by a disk) and a hyperedge (symbolized by a square from which edges point

to the ends of the hyperedge) that links the vertex to all of its neighbors.

Definition 14 (Partition of a Hypergraph)
We call partition of a hypergraph $H = (V, E)$ a partition of its vertices V.
To each partition of a mesh corresponds a unique partition of its dual hypergraph, and reciprocally.
Given a vertex $v \in V$ and a partition Π of H, we denote by $\Pi(v)$ the (unique) part to which v belongs.

If $H = (V, E)$ is a hypergraph, then a partition Π of H induces that some hyperedges in E are "cut", meaning that at least two of its ends belong to different parts. The following definition of the $\lambda - 1$ cut, as used by Devine *et al.* [2002], models the communication volume induced by a partition.

Definition 15 ($\lambda - 1$ Cut)
Let $H = (V, E)$ be a hypergraph, and Π a partition of H. Let $e \in E$ be a hyperedge of H. We define

$$\lambda(e) = \left|\left\{\Pi(v), v \in e\right\}\right| .$$

Note that $\lambda(e) \in \mathbb{N}^*$, because each hyperedge is a non-empty set and each vertex belongs to one part.
Given a communication weight function W_{com}, the $\lambda - 1$ cut of Π is:

$$cut_{\lambda-1}(\Pi) = \sum_{e \in E} W_{com}(e) \times (\lambda(e) - 1)$$

Remark
$\lambda(e)$ counts the number of different parts the ends of e belong to.
$\lambda(e) - 1$ counts the number of parts to which data on the center of e will need to be communicated.

Example
Figure 2.3.2 on the facing page displays an example of a 4-partition of a mesh and the partition of its corresponding hypergraph. Each color stems for one part.
Let e_{red} be the hyperedge colored in red, in the bottom right corner of the hole. $\lambda(e_{red}) = 3$, because its center belongs to the yellow part and it has one neighbor in the light blue part, and one in the blue part. Thus, the $\lambda(e_{red}) - 1 = 2$, which means that it will have to be communicated to 2 computation units.
Then, let us denote by e_{green} the hyperedge colored in green, on the right of e_{red}. $\lambda(e_{green}) = 2$, because its center lies in the light blue part, and it has neighbors in the yellow part. Thus, $\lambda(e_{green}) - 1 = 1$, which means that it

will have to be communicated to 1 computation units.

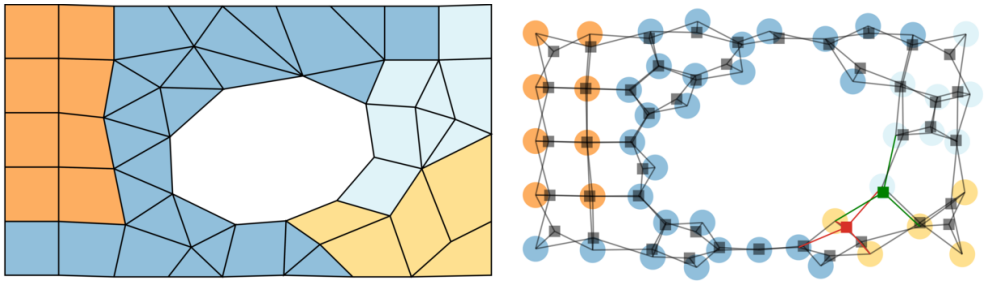


Figure 2.3.2 – Example of a 4-partition of a mesh and of its corresponding hypergraph

Thanks to the $cut_{\lambda-1}$ metric, the hypergraph model counts the communication volume corresponding exactly to the function f_{tot} defined in Section 2.1.3. Other metrics exist, and Fortmeier *et al.* [2013] recently proposed a new one, but the $cut_{\lambda-1}$ is a very common metric. For example, Catalyurek and Aykanat [1999] and Devine *et al.* [2006] use the $cut_{\lambda-1}$ to model the communication cost. Problem 3 gives the formulation of the multi-criteria hypergraph partitioning problem corresponding to the multi-criteria mesh partitioning Problem 2 on page 34.

Problem 3 (Multi-criteria Hypergraph Partitioning)
Given a hypergraph $H = (V, E)$, weights $W : V \to (\mathbb{R}_+)^\gamma$, a number of parts $k \in \mathbb{N}^*$, a tolerance $t \in \mathbb{R}_+$ and a communication cost function f, the multi-criteria hypergraph partitioning problem searches for a partition Π^{best} such that:

- $\forall c \in [\![1, \gamma]\!], \max_{\Pi_p \in \Pi^{best}} imb_c(\Pi_p) \leq t$ *(constraints)*
- $f(\Pi^{best}) = \min_{\Pi \in \mathfrak{P}_k(H),\, imb(\Pi) \leq t} f(\Pi)$ *(objective)*

Some partitioning tools, such as PaToH (see Catalyurek and Aykanat [2011]), use the hypergraph model. However, computing values with the $cut_{\lambda-1}$ function is complex and can result in an increase of partitioning time and complexity. This is why many tools use the graph model, which leads to easier and usually faster computations. The following section will present the graph model.

2.4 A Simpler Formulation with the Multi-criteria Graph Partitioning Problem

This section describes the graph model, which enables computing easily an approximation of the communication volume induced by a partition. We first define what a graph is, and then show how a graph can model a mesh, before formulating the multi-criteria graph partitioning problem.

Definition 16 (Graph)
A graph G is a pair $G = (V, E)$ where V is a set of elements called vertices *and E is a set of pairs of vertices. The elements of E are called* edges*. The elements of an edge are called its ends.*

Note that, in our case, the elements of E are unordered pairs of V, so the graph is said undirected. If the elements of E were ordered pairs, then the graph would have been said directed.

Remark
A graph is a particular type of hypergraph. The distinction lies in the elements of E: edges have exactly 2 ends, while hyperedges have a non-negative number of ends.

With each mesh, we can associate a graph. V corresponds directly to the set of cells: to each cell corresponds a unique vertex. The edges represent the neighboring relations between the cells. There is an edge between the vertices corresponding to cells m_1 and m_2 if m_1 and m_2 are neighbors. Definition 17 formally describes how to build the graph associated with a mesh, called its dual graph.

Definition 17 (Dual Graph of a Mesh)
Given a mesh M, we define $G = (V, E)$ such that:

- $V = M$
- $E = \left\{\{m_i, m_j\} \in M^2 \text{ if and only if } m_i \in \mathcal{N}(m_j)\right\}$

Thus, to each $m \in M$ corresponds a unique vertex, denoted by v_m. Reciprocally, to each vertex $v \in V$ corresponds a unique cell denoted by m_v.

Let $W : M \to (\mathbb{R}_+)^\gamma$ and $W_{com} : M \to \mathbb{R}_+$ be the computation and communication weights of M. We define the weight functions on G:

- $\forall v \in V, W(v) = W(m_v)$
- $\forall e = (u, v) \in E, W_{com}(e) = W_{com}(m_u) + W_{com}(m_v)$

Note that because an edge of the graph represents a couple of cells in the mesh, the communication weights are defined on E by attributing to an edge the sum of the weights of its ends.

Example

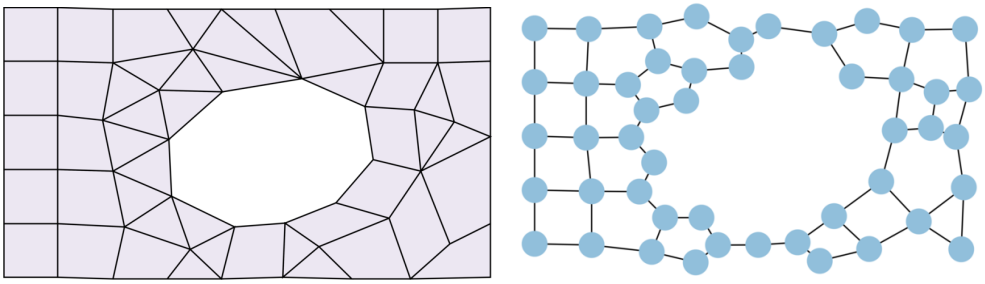


Figure 2.4.1 – Example of a mesh and its corresponding dual graph

The mesh is represented on the left and the corresponding dual graph on the right. To each cell of the mesh corresponds a vertex (symbolized by a disk). The neighboring relations are symbolized by the strokes between the vertices, that are the edges of the graph.

Definition 18 (Partition of a Graph)

We call partition of a graph $G = (V, E)$ a partition of its vertices V.

Thus, to each partition of a mesh corresponds a unique partition of its dual graph, and reciprocally.

Given a vertex $v \in V$ and a partition Π of G, we denote by $\Pi(v)$ the (unique) part to which v belongs.

If $G = (V, E)$ is a graph, then a partition Π of G induces that some edges in E are said to be "cut", meaning that their ends belong to different parts. The following definition of the edgecut, which is another formulation of the $cut_{\lambda-1}$ specific to the graph model, approximates the communication volume induced by a partition.

Definition 19 (Edgecut)

Let $G = (V, E)$ be a graph, and Π a partition of G. Let $e = \{u, v\} \in E$. We define:

$$cut(e) = \begin{cases} 1 \text{ if } \Pi(u) \neq \Pi(v) \\ 0 \text{ else} \end{cases}$$

Given a communication weight function W_{com}, the edgecut of Π is:

$$edgecut(\Pi) = \sum_{e \in E} W_{com}(e) \times cut(e)$$

The edgecut counts the number of edges cut by a partition. This is an approximation of the total communication volume f_{tot} defined in Section 2.1.3, as illustrated by the following example.

Example

Figure 2.4.2 shows an example of a 4-partition of a mesh and the partition of its corresponding graph. Each color stems for one part. The edges that are cut by the partition are dashed.

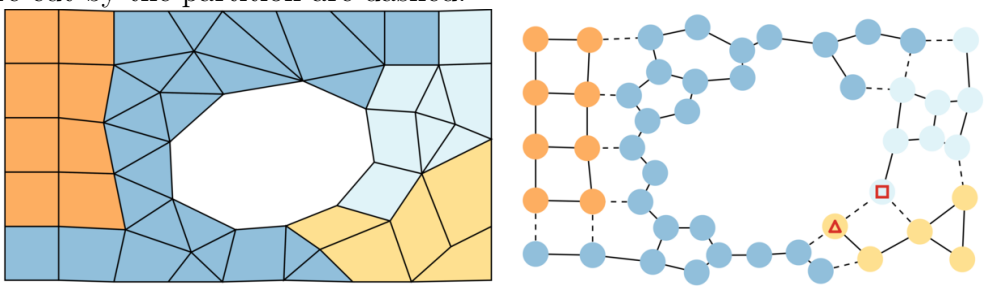


Figure 2.4.2 – 4-partition of a mesh and of its corresponding graph

Let v_{square} be the vertex marked with a square. Two of its edges are cut. This means that, when computing the edge cut, the communication weight of $m_{v_{square}}$ will be counted twice, whereas it must be sent only once to the yellow part.

Nevertheless, in the case of $v_{triangle}$, the vertex marked with a triangle, it is right to count the communication weight of $m_{v_{triangle}}$ twice, since it must be communicated once to the light blue part and once to the blue part.

Now assume that all communication weights of the cells are set to 1. The total amount of communication induced by this partition was computed using function f_{tot} defined in Section 2.1.3, $f_{tot}(\Pi) = 24$.

Using the graph model, we count 14 edges cut. Each edge $e = (u, v)$ has a weight of $W_{com}(m_u) + W_{com}(m_v) = 2$, which yields a communication volume of 28.

This example illustrates that the graph model, unlike the hypergraph model, will only approximate the total communication volume corresponding to the function f_{tot}.

The graph model is very common, maybe because it is simpler to understand than the hypergraph model. Although Hendrickson and Kolda [2000] state many limitations to this model, they also explain that, for mesh-based simulations, the estimated communication cost is usually rather correct, as explained by Catalyurek and Aykanat [1999]. However, note that this assumption may fail for $3D$ meshes, when two cells in the mesh are considered to be neighbors if they share a vertex or an edge (which corresponds to taking $d = 0$ or $d = 1$ in Definition 8, instead of $d = 2$). The following Problem 4 gives the formulation of the multi-criteria graph partitioning problem corresponding to the multi-criteria mesh partitioning Problem 2.

Problem 4 (Multi-criteria Graph Partitioning)
Given a graph $G = (V, E)$, weights $W : V \to (\mathbb{R}_+)^\gamma$, a number of parts $k \in \mathbb{N}^*$, a tolerance $t \in \mathbb{R}_+$ and a communication cost function f, the multi-criteria graph partitioning problem aims at finding a partition Π^{best} such that:

- $\forall c \in [\![1, \gamma]\!], \max_{\Pi_p \in \Pi^{best}} imb_c(\Pi_p) \leq t$ *(constraints)*
- $f(\Pi^{best}) = \min_{\Pi \in \mathfrak{P}_k(G),\, imb(\Pi) \leq t} f(\Pi)$ *(objective)*

As for the multi-criteria mesh partitioning problem, our formulation of the objective concerns the common case. It is up to the reader to adapt the f objective function to its particular environment. Nevertheless, usually $f = edgecut$, and in this case the multi-criteria graph partitioning problem is NP-Hard.

NP-Hardness of the Classic Multi-criteria Graph Partitioning Problem

Definition 5 on page 7 recalled the complexity theory concepts such as NP-Completeness and NP-Hardness. Roughly, a problem is said to be NP-Hard if it is at least as hard as the hardest problems in NP, which are called NP-Complete problems.

The proof of the NP-Hardness of the multi-criteria graph partitioning problem involves the decision version of this problem, which is defined in Problem 5. In the remainder of this section, we will denote by GP_γ the multi-criteria graph partitioning problem, GP_1 the mono-criterion graph partitioning problem, and GPD_γ and GPD_1 the decision versions of respectively the multi-criteria and mono-criterion graph partitioning problems.

Problem 5 (Multi-Criteria Graph Partitioning – Decision Problem)
Given a graph $G = (V, E)$, weights $W : V \to (\mathbb{R}_+)^\gamma$, a number of parts $k \in \mathbb{N}^*$, a tolerance $t \in \mathbb{R}_+$, a communication cost function $f : \mathfrak{P}(G) \to \mathbb{R}_+$ and a communication cost $f_0 \in \mathbb{R}_+$, the decision version of the multi-criteria graph partitioning problem searches whether there exists a partition Π such that:

- $\forall c \in [\![1, \gamma]\!], \max_{\Pi_p \in \Pi} imb_c(\Pi_p) \leq t$
- $f(\Pi) \leq f_0$

If we can solve GP_γ, then we can easily solve GPD_γ. Indeed, if Π^{best} is the optimal solution of an instance of GP_γ, then for $f_0 < f(\Pi^{best})$, the answer to GPD_γ is "no", and "yes" for any $f_0 \geq f(\Pi^{best})$.

Therefore, GP_γ is at least as hard as GPD_γ. Moreover, it is straightforward that GPD_γ is at least as hard as GPD_1. Hyafil and Rivest [1973] have proved that GPD_1 is NP-Complete when the communication cost is the *edgecut* function. Therefore, GP_γ is at least as hard as an NP-Complete problem, which means that GP_γ is NP-Hard when the communication cost is the *edgecut* function.

Conclusion

Lots of partitioning tools, including Scotch (see Pellegrini [2008]) and MeTiS (see Karypis and Kumar [1998a]), use the graph model. The graph model is to our knowledge the simplest topological model to represent a mesh. It formulates the multi-criteria mesh partitioning problem when the objective function counts the communication volume, using the edgecut of the graph, as an approximation.

However, before minimizing the objective function, the first task in our formulation remains to find a partition respecting the balance constraints. In the next section, we will ignore the topology and define a subproblem of the multi-criteria mesh partitioning problem: the vector-of-numbers partitioning problem.

2.5 Parallel Between Load Balancing Constraints and Vector-of-numbers Partitioning

This section considers a simplified version of the multi-criteria mesh partitioning Problem 2. This problem is expressed in the form of several constraints, namely to find a partition balanced for each criterion, and an objective, namely minimizing the communication cost. In this section, we will ignore the minimization objective of the multi-criteria mesh partitioning problem, to focus on finding a partition that satisfies the balance constraints.

If the number of criteria is 1, and assuming that the computation weights are integers, this leads to a well-known problem called the number partitioning problem.

To match our needs, we extended the number partitioning problem to the vector-of-numbers partitioning problem, formulated in Problem 6. This formulation, unlike the classic number partitioning problem, considers that the weights are real numbers. Whenever an algorithm requires the weights to be integers, it will be specified.

Until now, we have considered that the computation weights on the mesh were given through a weight function $W : M \to (\mathbb{R}_+)^\gamma$. However, to simplify readability, in the scope of number partitioning, we will use multisets. Multisets,

as stated in Definition 3 on page 6, are an extension of the concept of set in which the same value may appear several times.

Definition 20 (Multiset Associated with the Computation Weight Function of a Mesh)

Considering a mesh M and a computation cost function $W : M \to (\mathbb{R}_+)^\gamma$, we define the multiset:

$$S = \{W(m), m \in M\}$$

Note that S is a multiset of vectors, and that each vector in S has γ components.

Table 2.5.1 defines the notations that will be used in the scope of the vector-of-numbers partitioning problem.

Table 2.5.1 – Vector-of-numbers partitioning notations

Notation	Problem parameters	ref.	p.
$\gamma \in \mathbb{N}^*$	Number of physical models/criteria	sec. 2.1.2	25
$S \subset \mathbb{R}_+^\gamma$	Multiset of the computation weights, of $\texttt{length}(M)$ elements	def. 3	6
$k \in \mathbb{N}^*$	Number of parts in which we will partition S		
$t \in \mathbb{R}_+$	[Tolerance] Maximum imbalance allowed ($t = 0$ means that we search for a perfectly balanced partition)	sec. 2.1.4	30
$\Pi = (\Pi_1, ..., \Pi_k)$	Partition of S	def. 2	5

Problem 6 (Vector-of-Numbers Partitioning)

Given a finite multiset $S \subset (\mathbb{R}_+)^\gamma$, a number of parts $k \in \mathbb{N}^*$, a tolerance $t \in \mathbb{R}_+$, the vector-of-numbers partitioning problem amounts to finding a partition Π such that:

$$imb(\Pi) = \max_{c \in [\![1,\gamma]\!]} \max_{\Pi_p \in \Pi} imb_c(\Pi_p) \leq t \ .$$

However, the formulation in Problem 6 is rather unusual, with respect to the number partitioning problem. The two formulations that follow are the most classic definitions of this problem. They will be used in Chapter 3.

Problem 7 (Number Partitioning – Decision Problem)
Given a finite multiset $S \subset \mathbb{N}$ and $d \in \mathbb{N}$, determine if:

$$\exists \Pi \in \mathfrak{P}(S), imb(\Pi) \leq d$$

Problem 8 (Number Partitioning – Optimization Problem)
Given a finite multiset $S \subset \mathbb{N}$, find Π^{best}, partition of S, such that:

$$imb(\Pi^{best}) = \min_{\Pi \in \mathfrak{P}(S)} imb(\Pi)$$

This section has described a subproblem of the multi-criteria mesh partitioning problem, the vector-of-numbers partitioning problem. Solving this problem amounts to finding one valid partition of a multi-criteria mesh problem.

Before describing the algorithms addressing all the problems defined until now, the next section will introduce the last model that will be used in this thesis, which is the notion of fitness landscapes. Fitness landscapes help to get the big picture on the multi-criteria mesh partitioning problem, as it is aimed at analyzing a search space. In our case, the search space is the set of all k-partitions of a mesh. Fitness landscapes study if an algorithm moving across the search space may reach an optimal solution.

2.6 Exploration of the Search Space of the Multi-criteria Mesh Partitioning Problem with Fitness Landscapes

This section describes the fitness landscape model. Briefly, given an instance of the multi-criteria mesh partitioning problem, the fitness landscape model aims at answering whether an algorithm can reach an optimal solution for this instance. Malan and Engelbrecht [2013] and Richter and Engelbrecht [2013] define and sum up recent work that has been carried out on fitness landscapes.

Fitness landscapes can be applied to other problems than the mesh partitioning problem. Therefore, in this section, we will consider a problem $\mathbb{P}$ and an algorithm $\mathbb{A}$ that aims at solving $\mathbb{P}$.

Definition 21 (Instance)
Given a problem $\mathbb{P}$, an instance is a concrete representation of it or, in other words, the actual case that we are trying to solve.

Example

In this thesis, we consider $\mathbb{P}$ = the multi-criteria mesh partitioning Problem 2. An instance of this problem is given by the inputs (M, k, W, t, f), where:

- M is a mesh;
- $k \in \mathbb{N}^*$ is the number of parts;
- $W : M \to (\mathbb{R}_+)^\gamma$ is the computation weight function;
- $t \in \mathbb{R}_+$ is the imbalance tolerance;
- f is the communication cost function, which usually defines a communication weigh function $W_{com} : M \to \mathbb{R}_+$.

Given any instance $\mathcal{I}$ of the problem $\mathbb{P}$, $\mathbb{P}$ defines a function $f : \mathbb{X}_\mathcal{I} \to \mathbb{R}$. Solving $\mathbb{P}$ for the instance $\mathcal{I}$ amounts to finding an element of $\mathbb{X}_\mathcal{I} \to \mathbb{R}$ that minimizes f.

Definition 22 (Search or Configuration Space)

The set $\mathbb{X}_\mathcal{I}$ is called search space or configuration space, and depends on the instance $\mathcal{I}$.

The elements of $\mathbb{X}_\mathcal{I}$ are called candidate solutions.

Example

Given an instance $\mathcal{I} = (M, k, W, t, f)$ of the multi-criteria mesh partitioning problem, the search space is, $\mathfrak{P}_k(M)$, the set of all k-partitions of M.

Nevertheless, the problem $\mathbb{P}$ may impose some constraints, as for the multi-criteria mesh partitioning problem. In this case, all candidate solutions are not solutions for an instance of the problem.

Definition 23 (Solution Space)

Given some constraints defined by the problem $\mathbb{P}$ and the considered instance $\mathcal{I}$ of $\mathbb{P}$, an element of $\mathbb{X}_\mathcal{I}$ respecting the constraints is called a solution. The set of all solutions is called the solution space and will be denoted by $\mathbb{S}_\mathcal{I}$.

In other words, we are looking for $\Pi^{best} \in \mathbb{S}_\mathcal{I}$ such that

$$f(\Pi^{best}) = \min_{\Pi \in \mathbb{S}_\mathcal{I}} f(\Pi) \ .$$

Example

Given an instance $\mathcal{I} = (M, k, W, t, f)$ of the multi-criteria mesh partitioning problem, the solution space is the set of all k-partitions of M whose imbalance is smaller than t.

Thus, given an instance $\mathcal{I}$ of $\mathbb{P}$, algorithm $\mathbb{A}$ must return an element of $\mathbb{S}_\mathcal{I}$ minimizing f.

Solving $\mathbb{P}$ amounts to finding an algorithm that, given any instance $\mathcal{I}$ and the corresponding function $f : \mathbb{X}_\mathcal{I} \to \mathbb{R}$ to minimize, returns an element of $\mathbb{S}_\mathcal{I}$ that minimizes f if $\mathbb{S}_\mathcal{I}$ is not empty.

Remark

Given an instance $\mathcal{I} = (M, k, W, t, f)$ of the multi-criteria mesh partitioning problem, a partition Π of M can be encoded as a vector of length n containing in slot number i the part number of the ith cell in M (where n is the number of cells in the mesh).

For example, the partition vector for the partition of Figure 2.6.1, whose cells are numbered, is `[1, 1, 2, 2]`. Indeed, cells 1 and 2 belong to the first part and cells 3 and 4 to the second part.

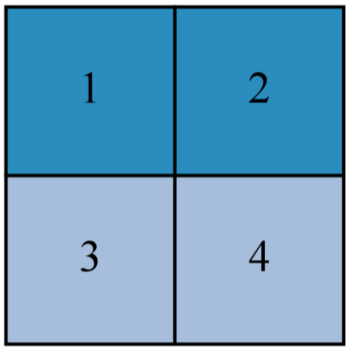


Figure 2.6.1 – A bipartition of a 4-cell mesh, which can be encoded with the vector `[1,1,2,2]`

Fitness landscapes consider that algorithm $\mathbb{A}$ will proceed in the search space, looking for a solution that minimizes f. From a given candidate solution, $\mathbb{A}$ explores $\mathbb{X}_\mathcal{I}$, switching from one candidate solution to another, until it finally stops and returns one, which in our case is a partition.

Definition 24 (Neighborhood Structure)

Given an instance $\mathcal{I}$ of a problem $\mathbb{P}$, the way algorithm $\mathbb{A}$ can switch from one candidate solution to another defines a neighborhood relation between candidate solutions. The neighborhood structure defined by $\mathbb{A}$ will be represented by $n_\mathbb{A} : \mathbb{X}_\mathcal{I} \to \mathcal{P}(\mathbb{X}_\mathcal{I})$, which means that given a candidate solution, $n_\mathbb{A}$ returns a subset of candidate solutions that are neighbors.

Example

Let $\mathbb{A}$ be a local optimization algorithm for the multi-criteria mesh partitioning problem. It means that $\mathbb{A}$ switches from a partition to another by changing the part assignments of a few cells. Using the vector notation of

a partition, that means changing some components in the vector.

In particular, this thesis will study local optimization algorithms that can switch at most one or two cells of part at a time. So, for these algorithms, given a partition Π represented with the vector v, the partitions that are neighbors of Π will be those which are represented with a vector equal to v except for at most one or two components.

With all these definitions, Definition 25 formulates what a fitness landscape is.

Definition 25 (Fitness Landscape)
Let $\mathcal{I}$ be an instance of problem $\mathbb{P}$, and $f : \mathbb{X}_{\mathcal{I}} \to \mathbb{R}$ be the objective function to minimize using algorithm $\mathbb{A}$. We call fitness landscape the entity:

$$\Lambda_{\mathcal{I}} = (\mathbb{X}_{\mathcal{I}}, n_{\mathbb{A}}, f)$$

where $n_{\mathbb{A}} : \mathbb{X}_{\mathcal{I}} \to \mathcal{P}(\mathbb{X}_{\mathcal{I}})$ is the neighborhood structure defined by $\mathbb{A}$.

By studying the properties of fitness landscapes, we can determine some characteristics on some classes of algorithms. For example, Chapter 5 will study whether the solution space is connected when using algorithms that move one vertex at a time.

Remark
A fitness landscape can be represented with a weighted oriented graph $(G_\Lambda = (V_\Lambda, E_\Lambda), W_\Lambda)$, with:

- $V_\Lambda = \mathbb{X}_{\mathcal{I}}$: a vertex in the graph is a candidate solution for the given instance, so, in the case of mesh partitioning, a partition of the mesh;
- E_Λ is defined by $n_{\mathbb{A}}$: given $(u, v) \in V_\Lambda^2$, $(u, v) \in E_\Lambda$ if $v \in n_{\mathbb{A}}(u)$;
- $W_\Lambda = f$: with each vertex is associated a value (in our case, the communication cost of the partition), and we search for a vertex of minimal value in G_Λ.

A local optimization algorithm starts from a vertex in V_Λ and iterates by passing from a neighboring vertex to another in order to find a vertex of minimal weight. Chapter 5 will study the capacity of a local optimization algorithm to find such vertex.

Conclusion

In this chapter, we have described and discussed the mesh, hypergraph, and graph models that express in a more concrete way the minimization of the run time of a multiphysics simulation. A common characteristic of these models is that the user needs to attribute computation and communication weights to each cell of the mesh.

The various models provide different visions of the same problem. The mesh model (Section 2.2) may lead to the most accurate formulation of it. The hypergraph model (Section 2.3) gets rid of the mesh geometry but remains topologically equivalent to the mesh model. The graph model (Section 2.4) differs from both the mesh and the hypergraph model, since it makes one more approximation on the communication cost of a partition. However, it is currently the most popular model, maybe because it is simpler to formulate and to understand.

Finally, this chapter has also introduced quite original problems, when considering the multi-criteria mesh partitioning problem. Firstly, the vector-of-numbers partitioning problem (Section 2.5) is a subproblem of the multi-criteria mesh partitioning problem, and is a generalization of a well-known problem, the number partitioning problem. Solving the vector-of-numbers partitioning problem amounts to finding a solution to a multi-criteria mesh partitioning problem. Secondly, the fitness landscape model (Section 2.6) is a generalization of the multi-criteria mesh partitioning model. It aims at studying the ability for an algorithm to, from a partition, find its way to an optimal solution.

The remaining chapters of this part will define existing algorithms that were designed to address the various problems that we defined in this chapter.

Chapter 3

Survey on Algorithms for Vector-of-Numbers Partitioning

Contents

Chapter 2 has described different problems modeling the minimization of the run time of multiphysics simulations. They rely on balance constraints, which means to find a partition of imbalance smaller than the input tolerance. This amounts to solving a vector-of-numbers partitioning problem.

The present chapter introduces existing studies addressing the vector-of-numbers partitioning Problem 6 on page 43. Given a set of vectors of numbers, we call *criteria* the components of a vector. Roughly, the vector-of-numbers

partitioning problem consists in finding a partition of the set such that for each all criteria, the weight of each part is nearly as equal as possible.

The mono-criterion case is known as the number partitioning problem and has been quite overlooked. Some of its properties have forged its nickname of the "Easiest-Hard problem", as will be explained in Section 3.1. Despite being called the easiest, it remains an NP-Complete problem, as proved by Garey and Johnson [1979]. The heuristics addressing it will be described in Section 3.2, and their characteristics summed up in Section 3.3. Finally, the extension of some of these heuristics to the multi-criteria case, will be described in Section 3.4.

Definitions and Notations

In our case, the entity that we need to partition is a multiset. Multisets were introduced in Definition 3 on page 6. In multisets, two equal elements can be distinct elements, which cannot be the case in sets. For ease of reading, in this chapter, the word "set" will designate a multiset (and "subset" will mean submultiset).

Besides, the following table recalls and extends the notations defined at the beginning of the manuscript in Table ii:

Reminder of problem parameters	
M	The mesh to partition
$k \in \mathbb{N}^*$	Number of computation units/parts into which we will partition M
$\gamma = 1$	Number of physical models/criteria
$W : M \to \mathbb{R}_+^\gamma$	Weights associated with each cell of the mesh
$t \in \mathbb{R}_+$	[Tolerance] Maximum imbalance allowed ($t = 0$ means that we search for a perfectly balanced partition)
Notations for number partitioning ($\gamma = 1$)	
$S = \{W(m), m \in M\}$	Set (in fact: multiset) of numbers to partition (here $W : M \to \mathbb{R}_+$ since $\gamma = 1$)
$S_e = \{4, 4, 7, 8, 9\}$	Example that we will use to illustrate the algorithms defined further in this section
$n = \lvert S \rvert$	Number of numbers in S
$u = \max(S)$	Highest number in S
$\Sigma = \sum_{w \in S} w$	Sum of all the numbers in S
$\Pi = \{\Pi_1, ..., \Pi_k\}$	Partition of S
$\Sigma_p = \sum_{w \in \Pi_p} w$	Sum of all the numbers in Π_p

Note that the usual number partitioning problem considers only integers,

while we allow for any positive real number. This does not change the way the following algorithms behave.

Besides, the common number partitioning problem is commonly formulated either as:

1. the decision Problem 7 on page 44: in this case, the objective is to find if a perfectly balanced partition exists ($t = 0$), but not to actually compute such a partition; or as
2. the optimization Problem 8 on page 44: in this case, the objective is to find a partition whose imbalance is minimal.

Our formulation, as given in Problem 6 on page 43 with $\gamma = 1$, is to find a partition of imbalance smaller than or equal to the input tolerance t. Firstly, we need to return a partition, and secondly, this partition does not have to be of minimum imbalance, so our formulation slightly differs from both common formulations.

Nevertheless, the properties on the number partitioning problem that will be explained in Section 3.1 apply to our formulation, and in Section 3.2, we will define classic approaches and adapt them to our particular formulation.

3.1 Properties of the Number Partitioning Problem

Number partitioning (in its decision version) is one of the six basic NP-complete problems defined by Garey and Johnson [1979]. However, it is not NP-complete in the strong sense, since dynamic programming can solve the number partitioning problem in pseudo-polynomial time. The dynamic programming algorithm, described in section 3.2.3, runs in time and space bounded by a low polynomial in $n \cdot u$. If the numbers provided can be arbitrarily large, then the number partitioning problem belongs to NP.

Among the instances of NP-complete problems, some are considered "easy", either because among the possible partitions, the proportion of solutions is high (these are said to be underconstrained), or because they are easily found insoluble because the number of candidate solutions to compute can be greatly reduced (these are said to be overconstrained).

In between lie instances that are "critically constrained". They form what is called a phase transition. Gent and Walsh [1998] provide a way to measure the constrainedness of an instance of the number partitioning (decision) problem. Given numbers drawn uniformly and at random from $[\![1, u]\!]$, they show that $\kappa = \frac{\log_2(u)}{n}$ characterizes the number of perfectly balanced partitions. Figure 3.1.1 on the next page shows the probability that a perfect partition exists when κ varies. Instances for which $\kappa \ll 1$ are underconstrained, meaning that a great proportion of candidate solutions are solutions, whereas $\kappa \gg 1$ indicates that the instance is overconstrained and very unlikely to have a solution. The phase

transition occurs when $\kappa \approx 1$.

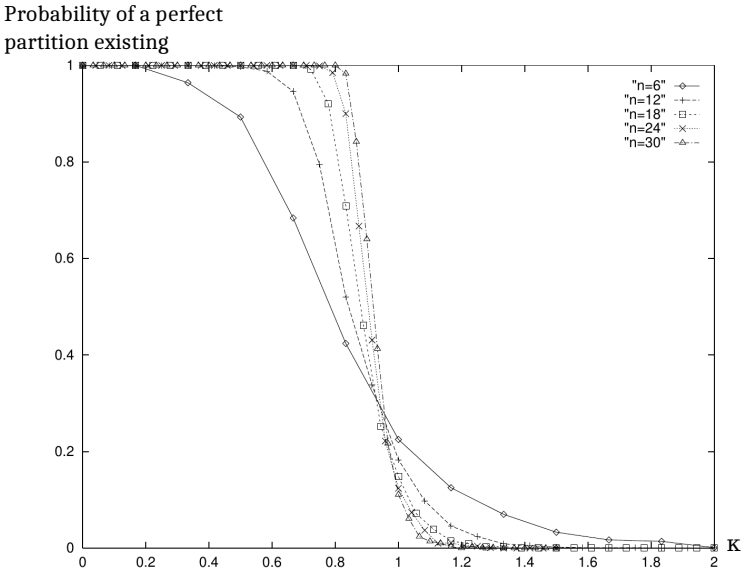


Figure 3.1.1 – Image from Gent and Walsh [1998] showing that the probability that a perfect partition exists drops for $\kappa \approx 1$. This phenomenon is called a phase transition, and characterizes the hard instances of the number partitioning problem (the ones for which $\kappa \approx 1$).

Their experiments are based on very small instances ($n \leq 30$) with potentially large numbers ($u \leq 2^{2n}$ in order to test various values of κ), because they need to compute all solutions. However, they subsequently apply finite-size scaling methods to show that their result scales with problem size.

Gent and Walsh also provide κ when the problem is to find an "imperfect partition", which means finding a partition of imbalance smaller than some tolerance t. In this case, they define $\kappa = \frac{\log_2(u/(t \cdot \Sigma))}{n}$, and the phase transition occurs again for $\kappa \approx 1$.

Being able to characterize the phase transition of the (uniformly random) number partitioning problem forged its "Easiest Hard Problem" nickname. However, Mertens [2003], while reviewing this phase transition in detail, highlights the poor quality of the heuristics addressing it. These heuristics will be detailed in the next section, and the following section will compare them.

3.2 Number Partitioning Algorithms

In this section, we consider the number partitioning Problem 6: we search for a partition whose imbalance is smaller than some input threshold t.

3.2.1 Greedy Algorithm (GA)

The greedy number partitioning algorithm GA returns a partition that may not be a solution. This is why GA is called a heuristic. It was introduced by Horowitz and Sahni [1974], and is also known as the best-fit decreasing heuristic. Algorithm 3 details how GA partitions a set S into k parts.

GA starts with all parts being empty and, at each step, places the largest remaining number into the part with the lightest weight. Therefore, the complexity of GA's complexity is the one of sorting the elements in decreasing order, which is $O(n \log(n))$.

Algorithm 3 Greedy Algorithm

```
1:  function GA(S, k, t)
2:      Π ← [[ ] ..k.. [ ]]
3:      Σ ← [0 ..k.. 0]
4:      S.sortDescend()
5:      while S ≠ [ ] do
6:          w ← S.popFirst()                  # Largest remaining number
7:          p_min ← argmin(Σ)                 # Index of lightest part
8:          Π[p_min].append(w)
9:          Σ[p_min] ← Σ[p_min] + w
10:     end while
11:     return Π
12: end function
```

Example

Table 3.2.1 shows the execution of GA on the example multiset S_e, searching for a bipartition. The last two columns display the weights of Π_0 and Π_1. At each step, the largest remaining number is put in the lightest part. The imbalance of the obtained partition is 6.25%.

Table 3.2.1 – The greedy algorithm always puts the largest remaining number in the lightest part

S_e (sorted)	Π_0	Π_1	Σ_0	Σ_1
[9, 8, 7, 4, 4]	[ ]	[ ]	0	0
[8, 7, 4, 4]	[9]	[ ]	9	0
[7, 4, 4]	[9]	[8]	9	8
[4, 4]	[9]	[8, 7]	9	15
[4]	[9, 4]	[8, 7]	13	15
[ ]	[9, 4, 4]	[8, 7]	17	15

3.2.2 Karmarkar-Karp Heuristic (KK)

The greedy algorithm took the largest remaining number and put it in the lightest part. Karmarkar and Karp [1983] proposed an algorithm that, instead of directly putting a number in a part, replaces the largest two remaining numbers by their difference, meaning that they will be in different parts. Meanwhile, the actual choice of which number goes to which part is saved for later use.

Like GA, KK is a heuristic, so the returned partition may not be a solution. The bipartitioning version of KK is simpler than the k-partitioning version.

Bipartitioning Case

Algorithm 4 defines the bipartitioning version, which is based on Proposition 1 formulated by Karmarkar and Karp.

Proposition 1 (Partitioning $S_{\setminus\{a,b\}} \cup \{a-b\}$)
Given $a \geq b$ elements of S, from a partition of $S' = S_{\setminus\{a,b\}} \cup \{a-b\}$, we can build a partition of S of same imbalance as S'.

Proof
Let $\Pi' = \{\Pi'_1, \Pi'_2\}$ be a partition of S', and assume that $a - b \in \Pi'_1$.
Consider $\Pi := \{\Pi_1, \Pi_2\}$ such that

$$\begin{aligned}\Pi_1 &:= \Pi'_{1\setminus\{a-b\}} \cup \{a\}\\ \Pi_2 &:= \Pi'_2 \cup \{b\}\end{aligned}$$

First, we recall a simple equality when bipartitioning:

$$\begin{aligned}imb(\Pi_1) &= \frac{\Sigma_1 - \frac{\Sigma}{2}}{\frac{\Sigma}{2}} = \frac{\Sigma_1 - \frac{\Sigma_1+\Sigma_2}{2}}{\frac{\Sigma}{2}} = \frac{\frac{\Sigma_1}{2} - \frac{\Sigma_2}{2}}{\frac{\Sigma}{2}}\\ &= \frac{\Sigma_1 - \Sigma_2}{\Sigma} = -imb(\Pi_2)\end{aligned}$$

Thus,

$$\begin{aligned}imb(\Pi) &= \max(imb(\Pi_1), imb(\Pi_2)) = |imb(\Pi_1)|\\ &= \left|\frac{\Sigma_1 - \Sigma_2}{\Sigma}\right| = \left|\frac{\Sigma'_1 - (a-b) + a - (\Sigma'_2 + b)}{\Sigma}\right| = \left|\frac{\Sigma'_1 - \Sigma'_2}{\Sigma}\right|\\ &= imb(\Pi')\end{aligned}$$

So, from a partition of S', we built a partition of S with the same imbalance, which proves the proposition.

□

Remark

Given a bipartition $\Pi = (\Pi_1, \Pi_2)$, the quantity $\Delta \coloneqq |\Sigma_1 - \Sigma_2|$ is called the absolute difference of Π. We have $imb(\Pi) = \frac{\Delta}{\Sigma}$.

Algorithm 4 Karmarkar-Karp Bipartitioning Algorithm

```
1:  function KK_Bipart(S, t)
2:      S.sortDescend()
3:      AmB ← [ ]                                   # Stems for "a − b"
        # 1st phase: build AmB
4:      while length(S) ≥ 2 do
5:          a, b ← pop2First(S)
6:          AmB.append([a, b])
7:          S.InsertInOrderedList(a − b)
8:      end while
        # 2nd phase: build the partition
9:      Π1, Π2 ← (S, [ ])                           # The last element in S is assigned to Π1
10:     while AmB ≠ [ ] do
11:         [a, b] ← AmB.popLast()
12:         if a − b ∈ Π1 then
13:             Π1.remove(a − b)
14:             Π1.append(a)
15:             Π2.append(b)
16:         else
17:             Π2.remove(a − b)
18:             Π2.append(a)
19:             Π1.append(b)
20:         end if
21:     end while
22:     return [Π1, Π2]
23: end function
```

The algorithm works as follows. First of all, S is sorted in descending order.

Then, in a first phase, the list AmB is constructed. At each step, the two largest numbers in S are removed from S, and stored in AmB. Their difference is inserted in order in S, which means that S remains sorted in descending order after the insertion. The first phase completes when there remains only one number in S, whose value is the absolute difference of the partition built in the second phase.

In the second phase, the remaining element is added to Π_1, while Π_2 is set to the empty set. Then, for each element $(a-b) \in AmB$, Proposition 1 is used. Hence, $(a-b)$ is replaced by a in the part it belongs to, while b is inserted in the other part, keeping the same imbalance for $\{\Pi_1, \Pi_2\}$. The algorithm stops

when all the elements in AmB have been processed and returns the resulting partition, regardless of whether the imbalance is below the tolerance or not.

Example

Table 3.2.2 – How the Karmarkar-Karp Algorithm Bipartitions S_e

First phase					
S_e (sorted)	a	b	a - b	Π_0	Π_1
$[9, 8, 7, 4, 4]$	9	8	1	$[7, 9]$	$[4, 4, 8]$
$[7, 4, 4, 1]$	7	4	3	$[1, 7]$	$[4, 4]$
$[4, 3, 1]$	4	3	1	$[1, 3]$	$[4]$
$[1, 1]$	1	1	0	$[1]$	$[1]$
$[0]$	-	-	-	$[0]$	$[\ ]$
				Second phase	

Table 3.2.2 shows how `KK` partitions S_e. The first phase is displayed in the left and middle columns, while the second phase (read from bottom to top) uses the middle and right columns.

The first phase fills-in columns a, b and $a - b$ from top to bottom, successively replacing in S_e its 2 largest numbers by their difference.

Then, the partition is built (columns Π_1 and Π_2), from bottom to top: at each step, the "$a - b$" from the previous line is replaced by a and b, a being put in the part that contained $a - b$. The obtained partition of S is on the top line, and its imbalance is 0, hence it is a perfect partition.

Remark

Algorithm 4 can be optimized by using the priority queue concept. Indeed, we do not need to sort the set of numbers, but rather to get at each step the maximum number from the set.

On the example set, `KK` returns a perfectly balanced partition. However, `KK` always puts the two largest numbers in different parts, so it will fail on a set whose partition would be perfect only if its two largest numbers are in the same part.

k-partitioning Case

The k-partitioning version of `KK` is detailed in Algorithm 5. It is more complex but has the same structure as the bipartitioning version. It still comprises two phases. In the first one, the differencing set AmB is built, while in the second one, the partition is built.

It begins with a sorting of S in descending order. Then, each number in S is transformed into a tuple of size k, the first entry being the number itself,

and the others being zeros. This creates the matrix M. Then, the algorithm behaves as the example in Table 3.2.3, which partitions S_e into 3 parts.

Example

Table 3.2.3 – How the Karmarkar-Karp Algorithm Partitions S_e into 3 Parts

First phase				
M	$(a+x, b+y, c+z)$		Partition	
$(a,b,c)(x,y,z)$	$\xrightarrow[\text{and sort}]{\text{normalize}}$	Π_0	Π_1	Π_2
$(9,0,0)(0,0,8)(7,0,0)(4,0,0)(4,0,0)$	$(9,0,8) \to (9,8,0)$	$[4,8]$	$[4,7]$	$[9]$
$(9,8,0)(0,0,7)(4,0,0)(4,0,0)$	$(9,8,7) \to (2,1,0)$	$[4,8]$	$[4,7]$	$[9]$
$(4,0,0)(0,0,4)(2,1,0)$	$(4,0,4) \to (4,4,0)$	$[4,1]$	$[4]$	$[2]$
$(4,4,0)(0,1,2)$	$(4,5,2) \to (3,2,0)$	$[4,1]$	$[4]$	$[2]$
$(3,2,0)$		$[3]$	$[2]$	$[0]$
		Second phase		

The first phase corresponds to the left and middle columns of the table. In this phase, the greatest numbers are successively separated. In the table, (a,b,c) and (x,y,z) denote the tuples with the largest numbers, with $a \geq b \geq c$ and $x \leq y \leq z \leq a$.

At each step, (a,b,c) and (x,y,z) are removed from M, and a new tuple is formed in place: $E := (a+x, b+y, c+z)$, meaning that a and x belong to the same part, b and y to another, and c and z to the last one. The tuple is normalized by its minimum ($\forall p, E[p]$ becomes $E[p] - \min(E)$) and sorted in descending order as shown in the middle column of the table, before inserting it back in M in order. The insertion function ensures that $M_1[1] > M_2[1] > \ldots$.

The second phase builds the partition as shown on the right column of the table, from bottom to top, which means that the last elements are considered first. The colors in the last line of Table 3.2.3 help to understand the first step, namely how to switch from the last line to the before-last line. On the last line, the $(3) \in \Pi_0$ corresponds to the (5) before normalization and sorting. Number (5) came from $b + y = 4 + 1$, so the (3) is replaced by $(4,1)$ in Π_0. The same is done for $2 \leftrightarrow 4 = 4 + 0$ in Π_1 and $0 \leftrightarrow 2 = 0 + 2$ in Π_2, resulting in the partition $\{\{4,1\},\{4\},\{2\}\}$ on the before-last line.

The algorithm continues until all elements in AmB have been processed, and returns the obtained partition, regardless of its imbalance.

This section has introduced the `KK` algorithm, one of the most famous heuristic to partition numbers. Its complexity is that of its sorting algorithm: $O(n \log(n))$, so it is quite fast. Whereas there is no guarantee on the imbalance of the returned partition, it has been shown by Yakir [1996] that when the numbers are uniformly distributed, for bipartitioning, `KK_Bipart` is expected

Algorithm 5 Karmarkar-Karp k-partitioning Algorithm

function KK(S, k, t)
  # For readability, L_i is the ith element of the list L
  $n \leftarrow \texttt{length}(S)$
  $S.\texttt{sortDescend}()$
  # Initialization: transform numbers into k-tuples
  $M \leftarrow \left[\left[S_1, 0\ \overset{k-1}{\ldots}\ 0\right]\ \overset{n}{\ldots}\ \left[S_n, 0\ \overset{k-1}{\ldots}\ 0\right]\right]$
  $AmB, norms \leftarrow ([\,], [\,])$
  # *1st phase:* build AmB
  while $\texttt{length}(M) \geq 2$ **do**
    $A, B \leftarrow \texttt{pop2First}(M)$
    $AmB.\texttt{append}\Big(\left[[A_1, B_k]\ \overset{k}{\ldots}\ [A_k, B_1]\right]\Big)$ # Save the old elements
    $E \leftarrow \left[A_1 + B_k\ \overset{k}{\ldots}\ A_k + B_1\right]$ # Compute the new element
    $e_{min} \leftarrow \texttt{min}(E)$ # Normalization factor
    $norms.\texttt{append}(m)$ # Save the normalization factor
    $E \leftarrow [E_1 - e_{min}\ \overset{k}{\ldots}\ E_k - e_{min}]$ # Normalize
    $E.\texttt{sortDescend}()$
    $M.\texttt{InsertInOrderedMatrix}(E)$ # So that $M_1[1] \geq M_2[1] \geq$ ...
  end while
  $\Pi \leftarrow M_1$ # First and last list in M
  # *2nd phase:* build the partition
  while $AmB \neq [\,]$ **do**
    $E, m \leftarrow (AmB.\texttt{popLast}(), norms.\texttt{popLast}())$
    $P \leftarrow [1\ \overset{k}{\ldots}\ k]$
    for $[a, b] \in E$ **do**
      $p \leftarrow \Pi(a + b - m)$ # Index of the part $(a + b - m)$ belongs to.
      $P.\texttt{remove}(p)$
      $\Pi_p.\texttt{remove}(a + b - m)$
      $\Pi_p.\texttt{append}(a)$
      $\Pi_p.\texttt{append}(b)$
    end for
  end while
  return Π
end function

to work better than GA. Then, Michiels *et al.* [2003] confirmed this result both experimentally and by computing the worst case performance of KK in the general case of k-partitioning.

3.2.3 Pseudo-Polynomial-Time Algorithm Based on Dynamic Programming

Garey and Johnson [1979] define Algorithm 6, a dynamic programming algorithm that solves the number bipartitioning problem in polynomial time in Σ. It builds the array is_sum, such that $is\_sum[i]$ is set to `True` if there is a subset of S of sum i (for a better readability, in this section, we assume that the tables indexes begin at 0).

Therefore, this algorithm works only if the elements of S are integers. Moreover, it is only designed for bipartitioning and has no known extension to k-partitioning, for $k > 2$ (though Korf and Schreiber [2013] compare the performance of exact number algorithms for both bi- and k-partitioning).

Algorithm 6 Dynamic Programming Algorithm

1: **function** DynProg(S, k, t)
Require: The elements of S are integers and $k = 2$.
2: $\Sigma \leftarrow \texttt{sum}(S)$
3: $n_{\mathcal{P}} \leftarrow \lceil \Sigma/2 \rceil + 1$ # Number of tracked subset sums
4: $\mathcal{P} \leftarrow [[\,]^{n_{\mathcal{P}}}]$ # Subsets of S
5: $is\_sum \leftarrow [\texttt{True}, \texttt{False}^{n_{\mathcal{P}}-1}\ \texttt{False}]$
6: **for** $number \in S$ **do**
7: **for** $i \leftarrow 0, n_{\mathcal{P}} - number$ **do**
8: **if** $is\_sum[i]$ and not $is\_sum[i + number]$ **then**
9: $is\_sum[i + number] \leftarrow \texttt{True}$
10: $\mathcal{P}[i + number] \leftarrow \mathcal{P}[i] \cup [number]$
11: **if** $\frac{\frac{\Sigma}{2} - (i + number)}{\frac{\Sigma}{2}} \leq t$ **then** # Found a solution
12: **return** $\left[\mathcal{P}[i + number], S_{\setminus \mathcal{P}[i+number]}\right]$
13: **end if**
14: **end if**
15: **end for**
16: **end for**
17: **return** None # Did not find a solution
18: **end function**

The algorithm works as follows. Initially, each cell in is_sum is set to `False`, except $is\_sum[0]$ which is set to `True`. Indeed, the empty set is a subset of S of sum 0. We also initialize each cell in the array $\mathcal{P}$ to an empty list. At each step, we consider a $number$ in S. For each cell $is\_sum[i]$ such

that $is\_sum[i]$ is `True`, if $is\_sum[i + number]$ is `False`, then we found for the first time a subset of S whose sum is $i + number$. Hence, $is\_sum[i + number]$ is set to `True`, and $\mathcal{P}[number + i]$ is updated to the subset found, which is the subset in $\mathcal{P}[i]$ to which $number$ is appended.

Setting the cell of index Σ_1 to `True` means that a partition of sums $(\Sigma_1, \Sigma_2 = \Sigma - \Sigma_1)$ has been found. Note that $\Sigma_1 \leq \frac{\Sigma}{2} \leq \Sigma_2$. Therefore, this partition is a solution if its imbalance is less than t, which corresponds to $\frac{\Sigma_2 - \frac{\Sigma}{2}}{\frac{\Sigma}{2}} = \frac{\frac{\Sigma}{2} - \Sigma_1}{\frac{\Sigma}{2}} \leq t$. The corresponding partition is $\{\mathcal{P}[\Sigma_1], S_{\setminus \mathcal{P}[\Sigma_1]}\}$. If no such partition is found, `None` is returned.

Figure 3.2.1 illustrates the construction of the is_sum array on the example multiset S_e, with $t = 0$. In this case, the partition is found on the line before the last.

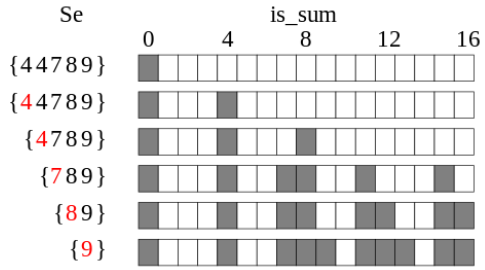


Figure 3.2.1 – The dynamic programming algorithm enumerates all possible subset sums of S_e. Cell i is colored in grey if a subset of sum i has been found.

This algorithm requires a space of size $O(n \cdot u)$, so it is hardly applicable to our problem, for which there can be many numbers ($n \sim 10^6$) that can be very large.

3.2.4 Optimized Dynamic Programming with the Horowitz and Sahni Algorithm (HS) and the Schroeppel and Shamir Algorithm

Horowitz and Sahni [1974]'s algorithm `HS` uses more memory to improve upon the time complexity of the previous algorithm. It has no k-partitioning version for $k > 2$. As it uses the `DynProg` Algorithm 6 defined in the previous section, it requires the numbers to partition to be integers. `DynProg` is assumed to have been modified to return the is_sum and $\mathcal{P}$ variables. We will now explain how `HS` operates; it is formulated in Algorithm 7, and an example is given in Table 3.2.4 on page 62.

Step 0. Compute some bounds Σ_{inf} and Σ_{sup} such that

$$\Pi = \{\Pi_1, \Pi_2\} \; solution \implies \begin{cases} \Sigma_1, \Sigma_2 \leq \Sigma_{sup} \\ \Sigma_1, \Sigma_2 \geq \Sigma_{inf} \end{cases}.$$

Algorithm 7 Horowitz and Sahni Partitioning Algorithm

1: **function** HS(S, k, t)
Require: The elements of S are integers and $k = 2$.
2: $n \leftarrow \texttt{length}(S)$
3: $n_A \leftarrow \lfloor \frac{n}{2} \rfloor$
4: $n_B \leftarrow n - n_A$
5: $\Sigma \leftarrow \texttt{sum}(S)$
6: $\Sigma_{sup}, \Sigma_{inf} \leftarrow \frac{\Sigma}{2}(1+t), \frac{\Sigma}{2}(1-t)$ # Bounds on the sum of a part *(Step 0)*
7: $S.\texttt{sortDescend}()$ # *(Step 1)*
8: $S_A \leftarrow S[1{:}n_A]$ # Largest n_A elements in S
9: $S_B \leftarrow S[n_A+1{:}n]$ # Smallest n_B elements in S
Algorithm 6 modified to return the subsets $\mathcal{P}$ of S *(Step 2)*
10: $\mathcal{P}_A \leftarrow \texttt{DynProg}(S_A, t=-1)$
11: $\mathcal{P}_B \leftarrow \texttt{DynProg}(S_B, t=-1).\texttt{reverse}()$
12: $\Sigma_A \leftarrow [\texttt{sum}(\mathcal{P}_A[1]) \dots \texttt{sum}(\mathcal{P}_A[n_A]]$ # ascending order
13: $\Sigma_B \leftarrow [\texttt{sum}(\mathcal{P}_B[1]) \dots \texttt{sum}(\mathcal{P}_B[n_B]]$ # descending order
14: $i_A, i_B \leftarrow 1, 1$
15: **while** $i_A \leq n_A$ and $i_B \leq n_B$ **do** # *(Step 3)*
16: $\Sigma_1 \leftarrow \Sigma_A[i_A] + \Sigma_B[i_B]$ # Sum of the numbers in Π_1
17: **if** $\Sigma_1 < \Sigma_{inf}$ **then** # Need to increase Σ_1 *(Case 1)*
18: $i_A \leftarrow i_A + 1$
19: **else if** $\Sigma_1 > \Sigma_{sup}$ **then** # Need to decrease Σ_1 *(Case 2)*
20: $i_B \leftarrow i_B + 1$
21: **else** # Found a solution *(Case 3)*
22: **return** $\left[\mathcal{P}_A[i_A] \cup \mathcal{P}_B[i_B], L_{\setminus(\mathcal{P}_A[i_A] \cup \mathcal{P}_B[i_B])}\right]$
23: **end if**
24: **end while**
25: **return** None
26: **end function**

These bounds will eliminate some partitions, when their sums do not range between Σ_{inf} and Σ_{sup}.

Step 1. Divide S into two sets of equal size, S_A and S_B. S_A contains the largest numbers in S and S_B the smallest.

Step 2. Compute the set Σ_A (respectively Σ_B) of all the possible sums of subsets of S_A (respectively S_B) using Algorithm 6 called with an imbalance impossible to reach, so that all subsets sums are computed. It is assumed that Algorithm 6 has been modified to return its $\mathcal{P}$ variable, which contains in its ith cell a subset of sum i.

Thus, Σ_A (respectively Σ_B), which contains in its ith cell the sum of the list $\mathcal{P}[i]$ (respectively $\mathcal{P}[i]$), is sorted in ascending (respectively descending) order.

Step 3. Iterate on i_A and i_B such that $\Sigma_A[i_A]$ is the smallest non-processed element in Σ_A and $\Sigma_B[i_B]$ the largest non-processed element in Σ_B. Their sum, denoted by Σ_1, corresponds to the sum of part Π_1.

If Σ_1 is between Σ_{inf} and Σ_{sup} (Case 3), a solution is found. Otherwise, if we need to increase the sum (Case 1), then we take the next element (larger) in A. Otherwise, (Case 2), we take the next element (smaller) in B.

The algorithm stops when the elements of Σ_A and Σ_B are exhausted or when a solution is found. To return the solution, the parts whose sum is Σ_1 are $\mathcal{P}_A[i_A]$ and $\mathcal{P}_B[i_B]$.

Example

Table 3.2.4 – The Horowitz and Sahni algorithm first divides the input set S_e in sets S_A and S_B. In step 2, all possible subset sums of A and B are computed. In step 3, we iterate on the elements of Σ_A and Σ_B, computing the possible sums of subsets of S_e, until a solution is found.

$S_e = \{4, 4, 7, 8, 9\}$, $d = 0$				
Step 0	$\Sigma_{inf} = \lfloor 31/2 \rfloor = 16$		$\Sigma_{sup} = \lfloor 33/2 \rfloor = 16$	
Step 1	$S_A = \{9, 8\}$		$S_B = \{7, 4, 4\}$	
Step 2	$\Sigma_A = \{0, 8, 9, 17\}$		$\Sigma_B = \{15, 11, 8, 7, 4, 0\}$	
Step 3	a	b	$a+b$	Case (in alg. 7)
	0	15	15	1
	8	15	23	2
	8	11	19	2
	8	7	15	1
	9	7	16	3

Horowitz and Sahni's algorithm returns a balanced partition when one exists. It was improved by Schroeppel and Shamir [1981] to use less memory. We will not describe their algorithm, but the idea is, instead of computing all the possible sums of subsets Σ_A and Σ_B, to generate them on demand. Schreiber [2014] gives its time complexity $O(n2^{\frac{n}{2}})$ and space complexity $O(2^{\frac{n}{4}})$.

3.2.5 Complete Greedy Algorithm (CGA)

Korf [1995] extended some heuristics to make "complete" algorithms, which return the optimal solution. Unlike the dynamic programming approaches, the complete algorithms do not require much memory. To do so, Korf enumerates all the possible partitions using a k-ary tree. Differences between complete algorithms lie in the way the tree is explored.

Bipartitioning Case. Algorithm 8 details the complete greedy bipartitioning algorithm defined by Korf. This algorithm creates the tree following the greedy algorithm idea. Figure 3.2.2 on the following page illustrates the construction of the binary tree for S_e.

Algorithm 8 Complete Greedy Bipartitioning Algorithm

```
1: function CGA_Bipart (S, t)
2:     S.sortDescend()
3:     return CGA_rec(S, t, [[ ], [ ]])
4: end function

5: function CGA_rec (S, t, Π)
6:     if S = [ ] then                                   # No number remaining
7:         if imb(Π) ≤ t then return Π else return None end if
8:     else                          # Add the largest remaining number to the lightest part
9:         i ← popFirst(S)
10:        p_heavy, p_light ← sortDescend([1, 2], key : p ↦ sum(Π_p))
11:        Π ←     CGA_rec(S, t, [Π_{p_light} ∪ [i], Π_{p_heavy}])
12:        if Π ≠ None then return Π end if
13:        return CGA_rec(S, t, [Π_{p_heavy} ∪ [i], Π_{p_light}])
14:    end if
15: end function
```

`CGA_Bipart` uses a recursive procedure, `CGA_rec`, to enumerate all possible solutions, returning a solution when one exits, or `None` otherwise. `CGA_rec` starts with empty parts and the list of all numbers in the S variable, which will record the numbers not already put in a part.

When all numbers have been assigned to a part, then the imbalance of the partition is computed. If it is smaller than t, meaning that we found a solution,

then the partition is returned. Otherwise, `None` is returned.

If some numbers have not been put to a part yet, we attribute them following the idea of `GA`. So, the largest remaining number, i, is removed from the list of the remaining numbers, and added to the lightest part before recursively calling `CGA_rec`. If the obtained partition is a solution (it is not `None`), then the solution is returned. Else, it tries to put i in the heaviest part before recursively calling `CGA_rec`.

Figure 3.2.2 shows the tree built when applying `CGA` to the example multiset S_e. Note that only half of the tree is shown, the other half being identical (but with number 9 in Π_2). The nodes are numbered in the order in which they are computed: the tree is explored in a depth-first way, the left branch of each node first. From a node, the largest remaining number is attributed to the lightest part on the left branch, and to the heaviest part on the right branch.

The first leaf computed corresponds to the application of `GA`. Here, the optimal solution is found on the 5th computed leaf (node 11).

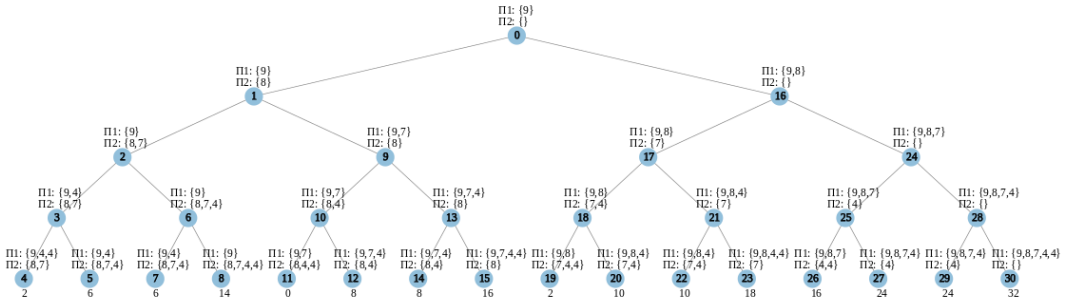


Figure 3.2.2 – The complete greedy bipartitioning algorithm tree applied to the example set $S_e = \{9, 8, 7, 4, 4\}$. The leaves are all the partitions of S_e. Their absolute differences are indicated below. The node numbers correspond to the order in which they are visited. In this example, the optimal (and perfect) partition is obtained at node 11. Note that only half of the tree is shown, the other half being symmetric (but with number 9 in Π_2).

Some techniques help avoid exploring all nodes. Firstly, if the absolute difference of a partition is greater than the sum of the remaining numbers, then all remaining numbers are assigned to the lightest part. For example, the absolute difference at node 9 is $\Delta = 16 - 8 = 8 \geq 4 + 4$, so we can immediately put the remaining numbers (4 and 4) in the lightest part, which corresponds to consider only node 11 in the subtree of root node 9. This would also work at node 16. Secondly, if the weights of each part are equal, then there is no need to compute 2 different branches because the absolute differences of their leaves would be the same. This is notably the case at the root of the tree (not shown here, we only see the left branch of the root), where the weight of each

part is null, so the choice of the part into which the first number is placed does not matter. These optimizations are not shown in Algorithm 8 to improve readability.

k-partitioning case. The tree becomes a k-ary tree, in which each branch assigns the largest remaining number to one part, beginning by assigning the number to the lightest part. We will not detail the algorithm, because it is very similar to the bipartitioning case.

3.2.6 Complete Karmarkar-Karp Heuristic (`CKK`)

Korf [1998] extended the `KK` algorithm to the complete Karmarkar-Karp algorithm. As `CGA`, `CKK` uses a k-ary tree to return an optimal solution.

Bipartitioning example. Algorithm 9 defines the bipartitioning version of `CKK`, and Figure 3.2.3 shows the tree that is created from the example set S_e. In the tree, the nodes are numbered according to their creation order. Below the leaves are the partitions obtained in each path from the root to the leaf.

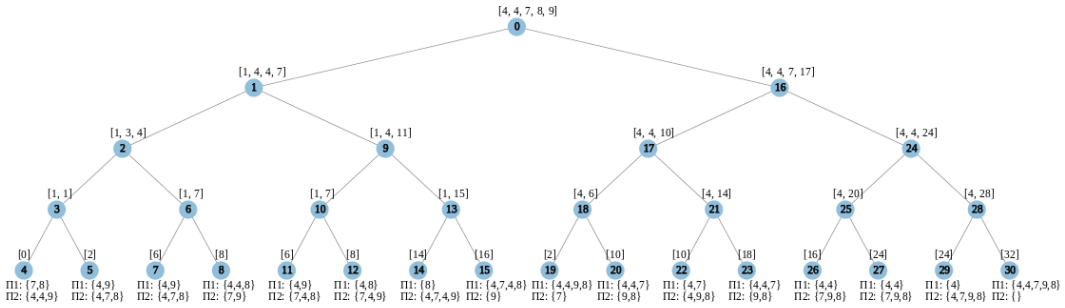


Figure 3.2.3 – The complete Karmarkar-Karp bipartitioning algorithm tree applied to S_e: the two largest numbers of a node are replaced in the left branch by their difference, and in the right branch by their sum. The nodes are numbered according to their visited order. The leaves are the partitions, with their absolute differences indicated above. They are computed a posteriori, using the method explained in Section 3.2.2. In this example, the optimal (and perfect) partition is the first computed one (obtained at node 4).

Algorithm 9 is implemented in a recursive way. Besides the set of numbers to partition, it takes two more arguments:

- AB: saves at each step the two largest numbers, their relation (`True` if they are to be put in different parts, `False` otherwise) and the number that was created (named ab; we will see that $ab = (a - b)$ or $(a + b)$);

Algorithm 9 Complete Karmarkar-Karp Bipartitioning Algorithm

```
1: function CKK_Bipart(S, t)
2:     S.sortDescend()
3:     return CKK_rec(S, t · sum(S), [ ])
4: end function

5: function CKK_rec(S, d, AB)
6:     if length(S) = 1 then
7:         if S[0] > d then                                  # Dead end
8:             return None
9:         else                                              # Found a solution
10:            return BuildKKPartition(S, AB)
11:        end if
12:    else
13:        a, b ← S.pop2First()
14:        S.InsertInOrderedList(a − b)      # (1st try) a and b in different parts
15:        AB.append([True, a, b, a − b])                    # Record try
16:        Π ← CKK_rec(S, d, AB)
17:        if Π ≠ None then return Π end if
18:        S.remove(a − b)
19:        S.InsertInOrderedList(a + b)      # (2nd try) a and b in the same part
20:        AB[−1] ← [False, a, b, a + b]                     # Replace 1st try record
21:        return CKK_rec(S, d, AB)
22:    end if
23: end function

24: function BuildKKPartition(S, AB)
25:    Π ← [S, [ ]]
26:    AB.reverse()
27:    for i ← 1, length(AB) do
28:        separate, a, b, ab ← AB[i]
29:        p ← Π(ab)                          # Index of the part ab belongs to
30:        Π[p].remove(ab)
31:        Π[p].append(a)
32:        if separate then                   # Add a and b to different parts
33:            Π[3 − p].append(b)                        # 3 − p: other part index
34:        else                               # Add a and b to the same part
35:            Π[p].append(b)
36:        end if
37:    end for
38:    return Π
39: end function
```

- $d = t \cdot \Sigma$: the difference between Σ_1 and Σ_2 (respective sums of each part) to consider a partition as a solution. Indeed, as shown in the proof of Proposition 1 on page 54, $imb(\Pi) = \frac{|\Sigma_1 - \Sigma_2|}{\Sigma} = \frac{d}{\Sigma}$.

We now detail the `CKK_rec` procedure. If S has at least two elements, the two largest (namely $a > b$) are removed from S and arranged as follows:

- first try: $(a - b)$ is added to S, which corresponds to applying the classic `KK` algorithm, or taking the left branch of the tree. Then `CKK_rec` is called recursively;
- second try: if the partition found in the first try was not a solution, then we try doing the opposite: the two largest elements are added to the same part. This amounts to appending $(a + b)$ to S, or taking the right branch of the tree.

The recursion stops when S has only one element left. As in `KK`, the last element value is Δ, the absolute difference of the partition. So, if the imbalance of the partition ($\frac{\Delta}{\Sigma}$) is smaller than the tolerance, we return the partition built using the `BuildKKPartition` function. Otherwise, `None` is returned, which means that another branch of the tree has to be explored.

The `BuildKKPartition` function works just as the second phase of the `KK_Bipart` Algorithm 4 on page 55, except that it has to take care whether a and b must be put in different parts or in the same part.

Like `CGA`, `CKK` uses techniques to avoid exploring all nodes. Firstly, Gent and Walsh [1998] proved that `KK` always gives the optimal solution if $n \leq 4$, so whenever the set of a node has 4 or fewer elements, the right branch is discarded. Secondly, if the largest number remaining is greater than or equal to the sum of all the other remaining numbers, the best solution of this subtree is obtained by placing this number in one part and all the other ones in the other.

k-partitioning case. We will not detail the k-partitioning algorithm. Its principle follows the k-partitioning `KK` algorithm that uses tuples of length k. Besides, the tree is no longer binary: each node possesses as many branches as there are combinations to sum-up the two largest tuples. For example, for $k = 3$, if (a, b, c) and (x, y, z) are the two largest tuples, then there are 6 possibilities to combine them: $(a + x, b + y, c + z)$, $(a + x, b + z, c + y)$, $(a + y, b + x, c + z)$, $(a + y, b + z, c + x)$, $(a + z, b + x, c + y)$ and $(a + z, b + y, c + x)$. Hence, in the general case, each node has $k!$ branches.

The last two sections have defined two exact algorithms, `CGA` and `CKK`, which explored the solution space as a binary tree. Unlike the other exact algorithms `DynProg` and `HS` defined in Sections 3.2.3 and 3.2.4, they do not need a tremendous amount of memory. They are based respectively on the greedy and Karmarkar-Karp algorithms `GA` and `KK`. The latter has been reported

to return more balanced solutions, and Korf [1995] noticed that `CKK` usually return solutions quicker than `CGA`. However, since these algorithms compute all possible partitions, they cannot apply to a large set of numbers.

The next section introduces another way of exploring the solution space, using stochastic search algorithms. These algorithms do not compute all solutions, but are not exact algorithms.

3.2.7 Stochastic Search Algorithms

Stochastic search consists in exploring the search space to find a solution. In our case, the search space has been defined in Definition 22: it is the set of all partitions of S. A stochastic search combines the following functions:

- `Evaluate`: compute the value associated with some partition Π. In our case, it is the imbalance of Π;
- `Randomize`: draw a partition from the search space;
- `Perturb`: generate a new partition from a given one.

Remark

Stochastic search algorithms usually try to maximize or minimize the `Evaluate` function, whereas we just need to drive it under some threshold, t. Nevertheless, stochastic algorithms can also be applied to the mesh partitioning Problem 2 with the communication cost as the `Evaluate` function. Therefore, we will formulate the algorithms in the usual way, assuming that their goal is to minimize the `Evaluate` function.

Because these algorithms are not guaranteed to return the optimal solution, they are "heuristics". Furthermore, because these algorithms can apply to various problems with different specifications of the `Evaluate`, `Randomize` and `Perturb` functions, they are called "meta-heuristics".

Definition of the Algorithms

Ruml *et al.* [1996] define four stochastic search algorithms for the number partitioning problem. They all use a variable $n_{iter} \in \mathbb{N}^*$ that specifies the maximum number of search steps.

Random Search. Algorithm 10 defines the random search, that picks partitions at random, only recording the best found.

Stochastic Descent Algorithm. Algorithm 11 defines this classic stochastic algorithm. From a random partition, the perturbations are accepted only when they yield some improvement.

Algorithm 10 Random Search

1: $\Pi_{best} \leftarrow$ `Randomize`()
2: **for** $i \leftarrow 1, n_{iter}$ **do**
3: $\quad \Pi \leftarrow$ `Randomize`()
4: $\quad$ **if** `Evaluate`(Π) $<$ `Evaluate`(Π_{best}) **then** $\Pi_{best} \leftarrow \Pi$ **end if**
5: **end for**
6: **return** Π_{best}

Remark

Ruml *et al.* named this algorithm `HillClimbing`, but as we try to minimize the imbalance function, `Descent` is more appropriate. Moreover, we sometimes call `HillClimbing` the algorithms that select partitions that have a worse `Evaluate` than the current one: they accept to climb up in order to descend deeper later.

Algorithm 11 Stochastic Descent Algorithm

1: $\Pi_{best} \leftarrow$ `Randomize`()
2: **for** $i \leftarrow 1, n_{iter}$ **do**
3: $\quad \Pi \leftarrow$ `Perturb`(Π_{best})
4: $\quad$ **if** `Evaluate`(Π) $<$ `Evaluate`(Π_{best}) **then** $\Pi_{best} \leftarrow \Pi$ **end if**
5: **end for**
6: **return** Π_{best}

Parallel Descent Algorithm. Ruml *et al.* also define a parallel descent Algorithm 12. From a population of partitions generated randomly, a solution is selected at random using the `SelectIndv` function. The selection process is such that better solutions are more likely to be chosen. The selected solution is duplicated before being changed and added to the population. Then, the `DeleteWorst` function removes the worst (according to the `Evaluate` function) partition from the population. Finally, the best partition according to the `Evaluate` function is returned.

Simulated Annealing. Finally, Algorithm 13 defines a stochastic version of a well-known algorithm. Its principle originated from annealing in metallurgy, and was introduced by Kirkpatrick *et al.* [1983]. From a random partition, at each step, the current partition is perturbed. If the new partition is better than the current one, the new one replaces the current one. Else, the difference Δf between the `Evaluate` values obtained for each partition is computed. The new partition replaces the current one with probability $e^{-\frac{\Delta f}{T(i)}}$, where T is a monotonically decreasing function called the temperature function. The

Algorithm 12 Parallel Descent Algorithm

```
1: Population ← [ ]
2: for j ← 1, n_indv do Population[j] ← Randomize() end for
3: for i ← 1, n_iter do
4:     Π ← selectIndv(1, n_indv)                    # Prioritize the best solutions
5:     Population[n_indv] ← Perturb(Π)
6:     DeleteWorst(Population)                      # "worst" according to Evaluate
7: end for
8: return min([Population, key : Π ↦ Evaluate(Π)])
```

temperature function controls the acceptance of partitions that are worse than the current one. This acceptance decreases with the iteration number i.

Algorithm 13 Simulated Annealing

```
1: Π_ini ← Randomize()
2: Π_best ← Π_ini
3: for i ← 1, n_iter do
4:     Π ← Perturb(Π_ini)
5:     Δf ← Evaluate(Π) − Evaluate(Π_best)
6:     if Δf ≤ 0 then
7:         Π_best ← Π
8:     else if rand(0, 1) < 10^(−Δf/T(i)) then
9:         Π_best ← Π
10:    end if
11: end for
12: return Π_best
```

Ruml *et al.* show that the performance of the four search algorithms defined here depends highly on the definitions of the `Evaluate`, `Randomize` and `Perturb` functions, that they call the encoding, and which corresponds to the problem representation.

Problem Representation

The problem representation defines the `Evaluate`, `Randomize` and `Perturb` functions of the stochastic search algorithms. For the number partitioning problem, the `Evaluate` function is always the computation of the imbalance of the partition, but the `Randomize` and `Perturb` functions vary. Ruml *et al.* define four problem representations, which are described in Table 3.2.5.

The direct representation is the reference. The `Randomize` function selects a partition among all the possible ones with the same probability. The `Perturb`

Table 3.2.5 – Problem representations defined by Ruml *et al.* [1996]

	Direct representation
`Randomize`	Generate one of the k^n possible partitions of S.
`Perturb`	Toggle the part of a random number, and randomly assign the part of another number.
	Greedy representation
`Randomize`	Generate one of the $n!$ permutations of $[\![1, n]\!]$ to order S. Compute the partition according to the greedy algorithm defined in Section 3.2.1.
`Perturb`	Toggle the part of a random number, and randomly assign the part of another number.
	Prepartitioning representation
`Randomize`	Generate a sequence O of random integers between 1 and n. If $O[i] = O[j]$, then the ith and jth elements of S are constrained to lie in the same part. The partition is then computed using the Karmarkar-Karp (`KK`) Algorithm 4 on page 55.
`Perturb`	Assign $O[i] = p$ where i and p are chosen randomly between 1 and n, then compute the partition using `KK`.
	Index-rules representation
`Randomize`	Generate a sequence R of random integers, with $0 \leq R[i] \leq n-2-i$. The partition is then computed using an adaptation of `KK`. Recall that `KK` always puts in different parts the two greatest remaining numbers. Here, at step i, this algorithm puts the greatest number and the $2+R[i]$th greatest number in different parts. For example, if $R[i] = 1$, then the 3rd largest number is considered.
`Perturb`	Draw at random i between 1 and $n-3$, and j between 1 and $n-2$. If $i \geq j$, then replace i with $n-3-i$. Else, replace j with $n-2-j$. At this point, every pair (i, j) such that $0 \leq i \leq n-3$ and $0 \leq j \leq n-3-i$ is equally probable. If $j = R[i]$, then replace j with $n-2-i$. Thus, we have $1 \leq j \leq n-2-i$ with $j \neq R[i]$. Compute the partition using the same adaptation of `KK` as in the randomize step.

function may perform one move (the part of one number is switched), two moves (the parts of two numbers are switched) or an exchange.

The greedy representation uses the same `Perturb` function as the direct representation, but the `Randomize` function generates a partition with the greedy algorithm `GA` defined in Section 3.2.1.

Ruml *et al.* use the Karmarkar-Karp algorithm `KK`, defined in Section 3.2.2 for the prepartitioning representation. This representation, in the `Randomize` function, forces some numbers to be in the same part, before applying `KK`. The `Perturb` function changes the numbers that have to be in the same part and applies `KK` once again.

The index-rules representation uses a variation of `KK`. While the original version always considers putting the two largest numbers in different parts, this algorithm puts the largest and another number in different parts. The `Randomize` function defines which number other than the largest should be considered at each step of `KK`. The `Perturb` function changes, for one step, the random number to consider.

3.3 Comparison of Number Partitioning Algorithms

Time and space complexities. Table 3.3.1 sums up the time and space complexities of the algorithms defined in the previous section, except for the stochastic algorithms, for which the complexities depend on the functions `Evaluate`, `Randomize` and `Perturb` defined by the problem representation.

While the `Evaluate` function is the same for each search algorithm, the `Randomize` and `Perturb` functions differ greatly. Their time complexities are displayed in Table 3.3.2, before the time complexities of the stochastic search algorithms.

The space complexity of the stochastic search algorithms is $O(n)$, except for the parallel hill-climbing algorithm whose space complexity is $O(n \cdot n_{indv})$.

Scope of the reported results. The following reports a number of studies comparing the capacity of a heuristic to find a solution and the actual time needed to compute it.

However, for all the reported studies, the results are based on the fact that the numbers derive from a uniformly random distribution. This assumption cannot be made in our case, for which we do not know the statistic distribution of the numbers, which are issued from the weights of the cells. Moreover, in some heuristics such as the multilevel one, the weights change. And finally, there is no guarantee that these results apply to the vector-of-numbers partitioning problem.

Table 3.3.1 – Time and space complexities for the number partitioning algorithms defined in Section 3.2, with $n = |S|$ and $u = \max(S)$.

Name		Algorithm type	Time complexity	Space complexity
Greedy Algorithm	`GA`	Heuristic	$n \log(n)$	n
Karmarkar-Karp	`KK`	Heuristic	$n \log(n)$	n
Dynamic programming	`DynProg`	Exact	$n \cdot u$	$n \cdot u$
Horowitz-Sahni	`HS`	Exact	$n \cdot 2^{\frac{n}{2}}$	$2^{\frac{n}{2}}$
Schroeppel-Shamir		Exact	$n \cdot 2^{\frac{n}{2}}$	$2^{\frac{n}{4}}$
Complete Greedy Algorithm	`CGA`	Exact	2^n	n
Complete Karmarkar-Karp	`CKK`	Exact	2^n	n

Table 3.3.2 – Time and space complexities for the stochastic search algorithms when used with different representations.

	Direct	Greedy	Prepartitioning | Index-rules
`Evaluate`	n	n	n
`Randomize`	n	$n \cdot \log(n)$	$n \cdot \log(n)$
`Perturb`	1	1	$n \cdot \log(n)$
Random	$n \cdot n_{iter}$	$n \log(n) \cdot n_{iter}$	$n \log(n) \cdot n_{iter}$
Stochastic Descent	$n \cdot n_{iter}$	$n(\log(n) + n_{iter})$	$n \log(n) \cdot n_{iter}$
Simulated Annealing	$n \cdot n_{iter}$	$n(\log(n) + n_{iter})$	$n \log(n) \cdot n_{iter}$
Parallel Descent	$n \cdot n_{iter} \cdot n_{indv}$	$n \cdot n_{iter} \cdot n_{indv}$	$n \cdot n_{iter}(\log(n) + n_{indv})$

Efficiency of Exact Algorithms. Among the exact algorithms, the Schroeppel-Shamir algorithm has the best time complexity for finding the optimal bipartition. Nevertheless, its application is only possible for small sets of integers. On the other hand, `CGA` and `CKK` have the worst-case complexity, but they do not need as much space.

`CGA` and `CKK` have the same complexity, but the latter will in fact find a solution much faster than the former, according to Korf [1998].

Efficiency of Heuristics. The two heuristics defined in this manuscript are `GA` (Algorithm 3) and `KK` (Algorithm 4). However, while `KK`'s worst-case complexity is the same as that of `GA`, `KK` actually outperforms `GA` in returning a solution when one exists. Korf [2009] argues that the imbalance of a partition

returned by GA is on the order of the smallest number, whereas for KK the imbalance is on the order of the last remaining number. Yet, for KK, the last remaining number comes from repeated differences, so it should be much smaller than the smallest number of the set.

Ruml *et al.* [1996] compared the stochastic search algorithms defined in the previous section. Their major conclusion is that the search algorithm has little influence on the solution found, while the problem representation is of great importance. Ruml *et al.* launched 100 times each of the 16 combinations of *problem representation* × *search algorithm* on 1 random instance of 100 numbers. After each run, the absolute difference u after 30 000 iterations is measured and compared with $u_{\texttt{KK}}$, the absolute difference of the partition obtained with KK.

Figure 3.3.1 shows $\log(\hat{u}) = \log(\frac{mean(u)}{u_{\texttt{KK}}})$. $\log(\hat{u}) > 0$ means that, on average, the heuristic did not find a better solution after 30 000 iterations than the application of KK. All representations except the prepartitioning representation lead to solutions that are on average worse than that returned by KK. The only algorithms that found on average a better solution than KK were the ones based on the prepartitioning representation. Among the heuristics based on the prepartitioning representation, there is little difference. Note that the Random search (RGT on the figure) performs notably well, even better on average than the Hill Climbing and the Simulated Annealing heuristics.

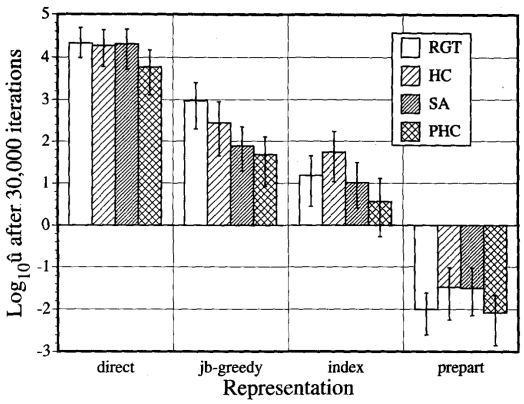


Figure 3.3.1 – Image from Ruml *et al.* [1996] showing that the search algorithm has little influence on finding the optimal solution, unlike the problem representation. The latter are listed on the x-axis, with all the search algorithms (RGT: Random; HC: Descent Algorithm; SA: Simulated Annealing; PHC: Parallel Descent Algorithm) using each representation. The y-axis is the average of $\log(\hat{u})$ over 100 runs, where $\hat{u}$ is the quotient between the absolute difference of the best partition found after 30 000 iterations, and the absolute difference of the partition returned by KK.

Up until now, we have focused on the number partitioning algorithm, for

which quite a few heuristics, as well as some exact algorithms, have been designed. However, most of them do not generalize to the vector of number partitioning problem, as we will see in the next section.

3.4 Vector-of-Numbers Partitioning Approaches

Kojić [2010] is one of the first to consider the vector-of-numbers bipartitioning Problem 6 on page 43, which she calls the "Multidimensional Two-way Number Partitioning Problem". Note that her formulation differs slightly from our, because it searches for a partition minimizing the imbalance, while we only search for a solution whose imbalance is smaller than a threshold. Also, as she explains, both of our formulations differ from the clustering problem, and clustering algorithms would not be able to handle the multidimensional two-way number partitioning problem.

In Section 3.4.1, we describe similarities between the vector-of-numbers partitioning problem and another classic problem, the multiple multi-dimensional knapsack problem. However, we will see that, as for clustering algorithms, we cannot adapt algorithms addressing the knapsack problem to the vector-of-numbers partitioning problem. Then, Section 3.4.2 will discuss if extensions of the number partitioning algorithms to the vector-of-numbers partitioning problem are possible. Finally, Section 3.4.3 will describe existing approaches addressing the multidimensional two-way number partitioning problem.

3.4.1 Difference with the Multiple, Multi-dimensional Knapsack Problem

The knapsack problem, studied for example by Chekuri and Khanna [1999], selects a number of items to pack into one bag of given weight or volume (the constraint), so that the utility (the objective function) of the selected items is maximized. The "multiple" version of the knapsack problem has several bags, and the multi-dimensional version has several constraints.

A possible analogy with the vector-of-numbers partitioning algorithm is to consider that the items are the vectors of numbers, and the bags are the parts of the partition. Packing each item in one bag gives a partition of the set of vector of numbers. The constraints (maximum weight and volume of one bag) are the imbalance tolerances for each criterion on the partition.

However, there is no utility function in the vector of number partitioning problem, and it would have no meaning since all numbers have to belong to one part. This means that all items have to be packed. There lies the difference between the two problems: whereas the knapsack problem needs to select some

items to pack, the number partitioning problem needs to pack all items such that they fit in the bags.

Therefore, a solution to a vector-of-numbers partitioning problem can be adapted to the corresponding knapsack problem (and since all items have been packed, the utility function is maximized). However, it does not really make sense to rely on a heuristic addressing the knapsack problem to solve a vector-of-number partitioning problem, because it may not pack all items, thus not returning a partition of the set of vectors of numbers.

This section has explained why the algorithms addressing the multiple multi-dimensional knapsack problem are not suited for the vector-of-numbers partitioning problem. The next section will define possible extensions of the number partitioning algorithms to the vector-of-numbers partitioning problem.

3.4.2 Extension of Number Partitioning Algorithms to the Vector-of-Numbers Partitioning Problem

Table 3.4.1 sums up the possible extensions of the number partitioning algorithms defined in the previous sections. A "✓" means that we propose a straightforward extension, while a "✗" means that an extension would need additional definitions. Note that Kojić [2010] carried out a quite similar analysis.

As explained in the table, the only algorithms whose extension can be naturally defined are the dynamic programming algorithm and the stochastic search algorithms, because they do not need any ordering notion.

Nevertheless, as explained in Sections 3.2.3 and 3.3, the dynamic programming algorithm can only *bi*-partition a set of *integers*, whereas in the general case, we need to k-partition a set of real vectors of numbers ($k \geq 2$). Besides, `DynProg` can demand a tremendous amount of space, because it requires a table of $n \cdot u$ cells (where $u = \max(S)$). In the multi-criteria case, we need as many tables as there are criteria, so the space complexity becomes $n \cdot u \cdot c$, which cannot fit for our applications.

Kratica *et al.* [2014] and Rodríguez *et al.* [2017] consider applying stochastic search algorithm to the multidimensional two-way number partitioning problem. They will be considered in the next section, which reports existing algorithms tackling the multidimensional two-way number partitioning problem.

3.4.3 State of the Art for the multidimensional two-way number partitioning problem

Kojić [2010] first defined an integer linear programming algorithm, and implemented it using CPLEX. She tested her approach on small instances, from 50 to 500 integers, with a run time ranging from 1min to half an hour. Such run times are not suitable in our case.

Table 3.4.1 – Discussion on the extension of the number partitioning algorithms defined in Section 3.2 to the vector-of-numbers partitioning problem

Algorithm		section	extension
Greedy algorithm	`GA`	3.2.1	✗
Puts the largest remaining element in the lightest part. However, the "largest" element and the "lightest" part have several possible definitions when dealing with vectors of numbers.			
Karmarkar-Karp algorithm	`KK`	3.2.2	✗
Puts the two largest remaining numbers in different parts and replaces them with their difference. It is then possible to build a partition of absolute difference equal to the last remaining element divided by Σ (sum of all the original elements). As for `GA`, defining the largest remaining numbers is not straightforward. Moreover, subtracting two vectors can produce non positive numbers, which is not defined in `KK`.			
Dynamic programming algorithm	`DynProg`	3.2.3	✓
Fills a table that records in cell i whether there is a subset of sum i. The adaptation would need as many tables as there are criteria.			
Horowitz and Sahni algorithm	`HS`	3.2.4	✗
Needs to sort the set of vector of numbers, as for `GA`.			
Complete Greedy algorithm	`CGA`	3.2.5	✗
Complete Karmarkar-Karp algorithm	`CKK`	3.2.6	✗
The complete algorithms respectively use `GA` and `KK`, which are not directly adaptable to the vector-of-numbers partitioning problem.			
Stochastic algorithms		3.2.7	✓
`Evaluate` is the imbalance function as in Definition 11.			

Pop and Matei [2013] then proposed to use a genetic approach. Genetic algorithms use crossover-like operators, which combine two partitions, and mutation-like operators, which perturb an existing partition. The crossover operator selects, between two partitions called the parents, the one of smallest imbalance. In the "child" partition obtained, one or two vector-of-numbers are attributed the same part as in the second parent. The mutation operator then randomly switches the part for approximately 10% of the vector-of-numbers. Their approach achieves to return solutions of smaller imbalance than the integer linear programming algorithm, and in less time. To partition 400 integers, the run time of their algorithm ranges from 5 to 10min, which is still too long for our applications.

Very recently, Kratica *et al.* [2014] and Rodríguez *et al.* [2017] used stochastic methods. Kratica *et al.* introduced variable neighborhood search (`VNS`) and an electromagnetism-like metaheuristic (`EM`), and Rodríguez *et al.* a GRASP algorithm with path relinking (`GRASP + PR`). All rely on several random starts (like the parallel descent algorithm) with various perturbation mechanisms. `VNS` explores several neighborhoods of solutions using a local optimization algorithm. `EM` defines an attraction-repulsion relation between partitions, modeling a "charge" that corresponds to the difference in imbalance with the "optimal" imbalance (which is the minimum imbalance found at runtime). `GRASP + PR` builds randomized initial partitions that are then improved by exploring paths between partitions of small imbalance (called "high-quality partitions"). Kratica *et al.* report that on average, `VNS` and `EM` return partitions of smaller imbalance than the genetic algorithm of Pop and Matei and the linear programming algorithm. Although `VNS` and `EM` results are very close, `EM` is slightly better. However, once again, the run time in general exceeds 5min for partitioning instances of 100 integers. As for `GRASP + PR`, Rodríguez *et al.* set a maximum computation time of $n/10$, for instances of n ranging from $n = 50$ to $n = 500$ vector-of-numbers, and for this time limit, `GRASP + PR` returns on average partitions of smaller imbalance than `VNS`, which is said to be its "main competitor".

To conclude, a common feature of the existing approaches is that they search for a partition of minimal imbalance, while we only search for a partition of imbalance smaller than a threshold. Searching for an optimal partition, the integer linear programming algorithm and the genetic algorithm run times are not suited for our instances. The stochastic algorithms seem to be able to return partitions of minimal imbalance when the computation time is smaller. Finally, all existing approaches only consider bipartitioning.

Conclusion

In this chapter, we have described a number of vector-of-numbers partitioning algorithms. The only number partitioning algorithms that can apply to our problem in practice, be it because of their worst-case complexity or their space complexity, are heuristics. Among the defined heuristics, stochastic search algorithms are the most promising algorithms to address the vector-of-numbers partitioning problem. In Chapters 5 and 7, we will study the descent algorithm. However, we will consider non-stochastic versions of it, because we may not need several runs to obtain a solution.

A vector-of-numbers partitioning algorithm aims at returning a solution to the corresponding mesh partitioning problem. Nevertheless, the communication cost of this solution can be high. Existing algorithms that also take into account the communication cost of a partition will be defined in the next chapter.

Chapter 4

Mesh Partitioning Algorithms

Contents

This chapter describes the existing partitioning algorithms addressing the multi-criteria mesh partitioning Problem 2 on page 34 or its variations, the multi-criteria hypergraph and graph partitioning Problems 3 on page 37 and 4 on page 41. Basically, the problem is to partition into k parts a mesh/hypergraph/graph whose cells/vertices are given a vector of γ weights (γ is the number of criteria), such that for each criterion, the partition balances the weights of the parts. Therefore, a partition is a solution of the problem only if its imbalance for each criterion is smaller than some input tolerance t.

Among all partitions respecting such balance constraints, or, in other words, among all candidate solutions, we search for an optimal solution, which is one that minimizes the communication cost induced by the partition. As explained previously in Chapter 2, mesh, hypergraph and graph model differently the communication cost.

The multi-criteria graph partitioning problem is NP-Hard when the communication cost is the classic *edgecut* function, as explained in Section 2.4.

Similarly, the corresponding multi-criteria mesh and hypergraph partitioning problems are also hard problems, and in our case, we will have to rely on heuristics. Heuristics are algorithms that do not guarantee to return an optimal solution. The goal of this chapter is to provide an overview of the existing heuristics that can be used to partition a multi-criteria mesh, hypergraph or graph.

First, Section 4.1 will describe an algorithm which is not a partitioning algorithm in itself. This algorithm, called recursive bisection (`RB`), extends a bipartitioning algorithm into a k-partitioning algorithm. Many algorithms rely on `RB` for k-partitioning when $k > 2$.

For example, the geometric algorithms described in Section 4.2 often rely on `RB`. Geometric algorithms partition a mesh using the geometric coordinates of its cells.

On the opposite, some algorithms rely only on the mesh topology, modeling the mesh with a hypergraph or a graph as explained previously in Sections 2.3 and 2.4. In this document, we have divided the topological algorithms into two categories: (i) the algorithms usually used to build a partition from scratch, called direct algorithms, will be described in Section 4.3, and (ii) the algorithms that refine an input partition, called refinement algorithms, will be described in Section 4.4.

Finally, Section 4.5 will introduce the multilevel algorithm. This algorithm is used by most of the partitioning tools, and considered by Buluç *et al.* [2015] as the most successful heuristic for partitioning large graphs. Nevertheless, as stated by Pellegrini [2008], this algorithm may also be seen as a partitioning strategy, because it usually relies on other partitioning algorithms.

Remarks

Evolutionary algorithms and simulated annealing have also been considered to address the mesh partitioning problem. They will not be described in this document, but Bichot and Siarry [2010] describe them very well. Concerning mesh partitioning, they have been supplanted by the multilevel algorithm.

Topological algorithms do not apply only on mesh partitioning. Their application is wide: they appear among others in the physical design of digital circuits for very large scale integration (VLSI), in the classification of individuals for complex networks such as power grids and biological or social networks, in image processing... Buluç *et al.* [2015] have provided a good survey on graph partitioning algorithms and their applications.

Notations

All the algorithms defined in this section take as arguments:

- the mesh M or the corresponding graph/hypergraph G to partition;
- the number of part $k \in \mathbb{N}^*$;
- the computation weights $W : M \to \mathbb{R}_+^\gamma$ associated with each cell/vertex;
- the imbalance tolerance t;
- the communication cost function $f : \mathfrak{P}(M) \to \mathbb{R}_+$ that should be minimized.

Table ii on page 9 details and references the meaning of these parameters.

4.1 Recursive Bisection (RB)

This section describes the recursive bisection algorithm `RB`, which forms a k-partitioning algorithm from a bipartitioning algorithm `Bisect`. `RB` was introduced by Kernighan and Lin [1970]. Since then, many bipartitioning algorithms such as geometric algorithms (Section 4.2) or spectral bisection (Section 4.3.1) have relied on `RB` for k-partitioning.

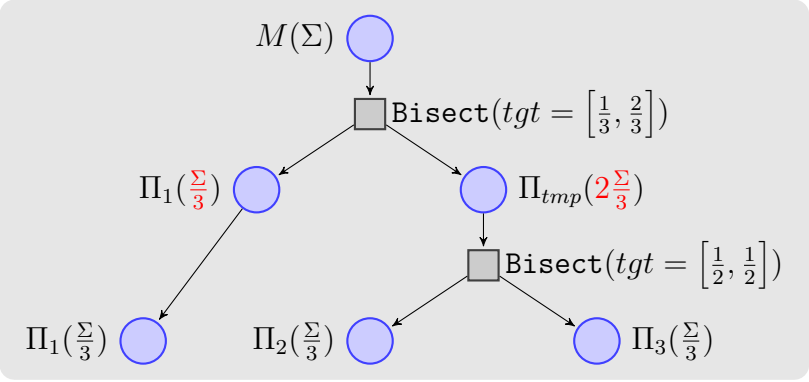
Overview and Example

The idea is to first bipartition the mesh using `Bisect`. Then, each part is bipartitioned with `Bisect`, which forms a 4-partition of the mesh. By repeating this scheme, we obtain a 2^m-partition of the mesh for any $m \in \mathbb{N}$.

If k is not a power of 2, then at least one partition call has to require a partition whose weight for one part is not one half. The following example illustrates how to handle $k = 3$.

Example

The following diagram shows the different steps to partition a mono-criterion mesh M of total weight Σ into $k = 3$ parts of weight $\frac{\Sigma}{3}$. The circles show the parts, with their name and weight indicated nearby. The squares symbolize the application of the bipartitioning function `Bisect`.



First, Bisect is applied on M, targeting a weight of $\frac{\Sigma}{3}$ for part Π_1 and $\frac{2\Sigma}{3}$ for part Π_{tmp}. Then, Bisect is called on Π_{tmp} with "regular" target part weights, which means half the weight of Π_{tmp} for both parts. This forms Π_2 and Π_3, both of weight $\frac{2\frac{\Sigma}{3}}{2} = \frac{\Sigma}{3}$.

Remark

In order to make k-partitions when k is not a power of 2, the Bisect algorithm must take another argument, which is the list of the target part weights. In the following, this argument will be called $tgt \in [0,1]^{\gamma \times k}$.

Algorithm

Algorithm 14 describes the recursive bisection function RB. If $k = 1$ then the partition is immediate. Otherwise, first, the tolerance is adapted to k, as will be explained further. Then, the target weights for the next bipartitioning are computed. If k is even, the target weights are $\left[\frac{1}{2}, \frac{1}{2}\right]$, but if k is odd, the target weights are $\left[\frac{k-1}{2k}, \frac{k+1}{2k}\right]$.

The bipartitioning algorithm Bisect computes a bipartition $\Pi = \{\Pi_1, \Pi_2\}$ with the adapted target weights. Finally, RB is recursively applied to Π_1 and Π_2 with the right number of parts: if k is even, respectively $\frac{k}{2}$ and $\frac{k}{2}$, and if k is odd, respectively $\frac{k-1}{2}$ and $\frac{k+1}{2}$.

Algorithm 14 Recursive Bisection

function RB(M, k, W, t, f, Bisect)
  if $k = 1$ **then return** $[M]$ **end if**
  $t \leftarrow$ AdaptTolerance(t, k)
  $k_1, k_2 \leftarrow \lfloor \frac{k}{2} \rfloor, \lfloor \frac{k+1}{2} \rfloor$
  $\Pi \leftarrow$ Bisect$(M, W, t, f, tgt = \left[\frac{k_1}{k}, \frac{k_2}{k}\right])$
  # If $imb(\Pi) < t$, the next bisection steps have a greater tolerance
  $\Pi' \leftarrow$ RB$(\Pi_1, k_1, W, t_1, f,$ Bisect$)$
  $\Pi'' \leftarrow$ RB$(\Pi_2, k_2, W, t_2, f,$ Bisect$)$
  return $\Pi' \cup \Pi''$
end function

Adapting the Tolerance

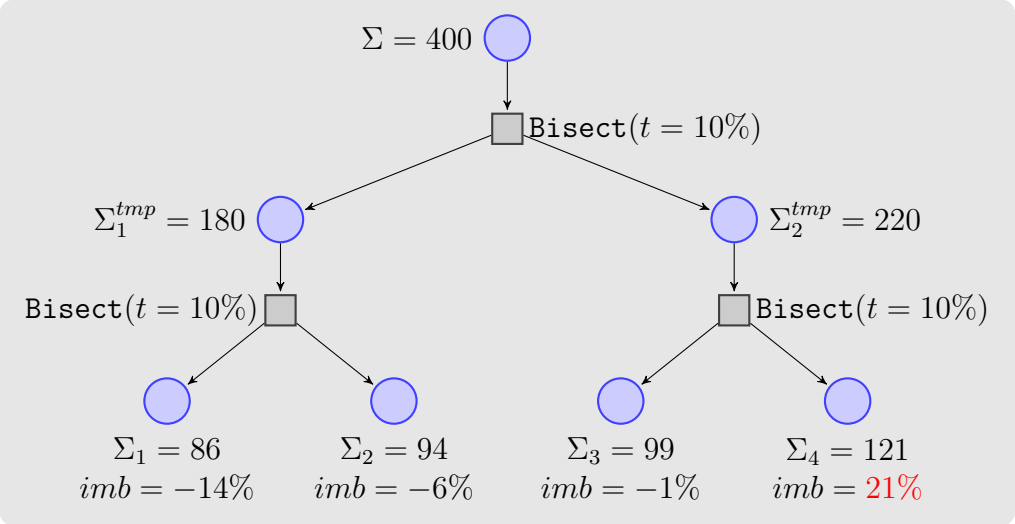
When using RB, Karypis and Kumar [1998b] give an example illustrating why the imbalance tolerance set at each step must be adapted. This is the purpose of the AdaptTolerance function used in Algorithm 14. We will first provide such an example, and then give in Proposition 2 the maximum imbalance that

a partition computed using `RB` can reach if the same tolerance is given at each call to the `Bisect` algorithm.

Example

For example, consider the execution of `RB` below using some bipartitioning algorithm `Bisect`. The input tolerance is $t = 10\%$. We consider a mono-criterion mesh of total weight $\Sigma = 400$, and we want to partition it into $k = 4$ parts.

The following graph shows possible weights of each part when `Bisect` is called with $t = 10\%$ every time.



The heaviest part, Π_4, reaches a weight of $\Sigma_4 = 121$, which accounts for an imbalance of $imb = \frac{\Sigma_4 - \frac{\Sigma}{k}}{\frac{\Sigma}{k}} = \frac{121-100}{100} = 21\%$. Thus, the final imbalance of the partition far exceeds the input tolerance of 10%.

Proposition 2 (Imbalance when using recursive bisection)

After m calls to a bipartitioning algorithm that returns partitions of imbalance at most t, the final imbalance can reach $(1+t)^m - 1$.

Proof

Let Σ be the sum of the weights of the mesh for one criterion. We will show by induction that after m steps, the weight of the first part for this criterion can reach $\frac{\Sigma}{2^m} \times (1+t)^m$.

Basis: this is true for $m = 0$ by hypothesis.

Inductive step: assume that the statement holds for m, then assume that the weight of the first part is $\Sigma_1^m = \frac{\Sigma}{2^m} \times (1+t)^m$. When partitioning with an algorithm that returns partitions of imbalance at most t, the first

part, in the worst case, will have a weight of:

$$\Sigma_1^{m+1} = \frac{\Sigma_1^m}{2} + t \times \frac{\Sigma_1^m}{2}$$

(which corresponds to a partition of imbalance t).

Thus,

$$\begin{aligned}\Sigma_1^{m+1} &= (1+t) \times \frac{\Sigma_1^m}{2} \\ &= (1+t) \times \frac{\frac{\Sigma}{2^m} \times (1+t)^m}{2} \qquad \text{using the inductive assumption} \\ &= \frac{\Sigma}{2^{m+1}} \times (1+t)^{m+1} \ .\end{aligned}$$

Conclusion: we have proved by induction that after m calls to an algorithm returning partitions of imbalance at most t, the weight of the first part is in the worst case $\frac{\Sigma}{2^m} \times (1+t)^m$.

That means that the imbalance after m calls is in the worst case $(1+t)^m - 1$. □

As stated in Proposition 2, if the recursive bisection algorithm is used without care, the imbalance can exceed greatly the given tolerance. Karypis and Kumar [1998b] study several schemes in order to guarantee that the final imbalance, when using recursive bisection, does not exceed the input tolerance.

Karypis and Kumar first scheme considers allowing a constant tolerance at each step. Thus, the tolerance at each step is computed once at the beginning of the algorithm; when computing a k partition, if $m = log_2(k)$, it is $\sqrt[m]{1+t} - 1$. However, this can lead to very tight tolerances; for example, with $k = 32$ and $t = 5\%$, then the tolerance at each step is $\sqrt[5]{1+0.05} - 1 = 0.98\%$.

Therefore, Karypis and Kumar propose other schemes that vary the tolerance at each step: the tolerance is either relaxed in the first steps, then tightened in the last steps, or on the contrary tight in the first steps and relaxed in the lasts.

In Scotch-5.1.2, the tolerance is dynamically adapted after each call to `Bipart`, to the maximum possible tolerance ensuring that the final partition will respect the balance constraints.

Benefits of RB

Recursive bisection is a simple scheme that enables one to use a bipartitioning algorithm for k-partitioning. It was first proposed by Kernighan and Lin [1970]. It is usually simpler to implement than extensions of bipartitioning algorithms to k-partitioning when $k > 2$. For example, the Fiduccia-Mattheyses algorithm that will be introduced in Section 4.4.2 becomes very complex for

direct k-partitioning when $k > 2$.

Limitations of RB

Simon and Teng [1997] claim that direct k-partitioning algorithms lead to partitions of smaller communication cost than RB, and show it with experiments on mono-criterion graph partitioning. Aykanat *et al.* [2008] extend this claim to multi-criteria hypergraph partitioning. In particular, they advance possible reasons for the performance degradation when using RB, such as the need to tighten the tolerance at each bisection level (in the `AdaptTolerance` function), which, as we will see in the next chapter, restricts the solution space. Nevertheless, note that Aykanat *et al.* still rely on RB for the initial partitioning phase of the multilevel algorithm, which will be defined in Section 4.5.

Besides, even if Karypis and Kumar [1998b] study several schemes for setting the tolerance using the `AdaptTolerance`, they do not provide a clear conclusion. Therefore, how to set the tolerance at each bisection level remains an open question.

Conclusion

We have defined and analyzed the recursive bisection strategy RB. RB is not a partitioning algorithm by itself, but it transforms a bipartitioning algorithm `Bisect` into a k-partitioning algorithm. RB is a simple scheme that calls `Bisect` several times. However, the imbalance tolerance used at each step must be adapted from the global input tolerance, and the best way to set this tolerance is still an open problem. Besides, several works argue that RB cannot, in many cases, return partitions of communication costs as small as the ones that direct k-partitioning algorithms can reach.

Some of the following algorithms, such as geometric and spectral algorithms, or any bipartitioning algorithm, may rely on RB in order to k-partition a mesh when $k > 2$. We will also use RB in our experiments, in Chapter 10.

4.2 Mesh Partitioning using Geometric Algorithms

This section defines and analyzes partitioning algorithms that rely on the mesh geometry. Such approaches can be justified by the "ham sandwich" theorem, whose origin is not well-known but is related by Beyer and Zardecki [2004]. Basically, this theorem states that γ objects in a γ-dimensional Euclidean space, such as two chunks of bread and ham in $3D$, can all be simultaneously bisected with a single $(\gamma - 1)$-dimensional hyperplane (*i.e.* in $3D$, a plane).

Note that this theorem stems only for the continuous domain Ω that the mesh M models, but not for M, which is a discrete object. For example, the

discrete weights assigned to each mesh cell do not have direct equivalents in the ham sandwich theorem.

4.2.1 Recursive Coordinate Bisection (RCB)

Overview

The recursive coordinate bisection algorithm (RCB) was first proposed by Berger and Bokhari [1987]. As shown in Algorithm 15, it uses the recursive bisection Algorithm 14 introduced in the previous section.

The bipartitioning algorithm used, `CoordBipart`, takes an axis as argument (usually, the most elongated axis is given), and cuts the mesh along the axis or plane orthogonal to this axis. As the place of the cut depends on the weights of the cells, the common versions of `CoordBipart` differ between mono and multi-criteria partitioning. Nevertheless, in both cases, it does not directly take into account the objective function f, but is based on the idea that a "straight" cut will not induce a high communication cost.

Algorithm 15 Recursive Coordinate Bisection

```
1: function RCB(M, k, W, t, f)
2:     return RB(M, k, W, t, f, CoordBipart)        # See Algorithms 14 and 16
3: end function
```

Remark

As explained in the previous section, the bipartitioning algorithms used by RB must take one more argument: the target weights. For example, $tgt = \left(\frac{1}{5}, \frac{4}{5}\right)$ means that the weight of the first part should be close to one fifth of the total weight.

Mono-criterion Bipartitioning Algorithm

Algorithm 16 defines the `CoordBipart` function when dealing with a mono-criterion instance. First, the cells are ordered according to their coordinate along the input axis. The input axis is usually the one along which the mesh is the most elongated. In the algorithm, for a cell $m \in M$, $m.\texttt{coord}(d)$ denotes the coordinate of m along the d axis. Then, the `CutHalfWeight_mono` function will take the elements in order and put them in the first part, stopping when the first part outweighs $tgt[1] \cdot \Sigma$.

Remark

The `CutHalfWeight_multi` function could be used for mono-criterion partitioning, but `CutHalfWeight_mono` is the common algorithm in the mono-criterion case.

Algorithm 16 Coordinate Bipartition

```
1: function CoordBipart(M, W, t, f, tgt, d: axis)
2:     order ← sortAscend([M[1].coord(d) .. M[n].coord(d)])
3:     return CutHalfWeight(M, W, order, tgt)
4: end function

5: function CutHalfWeight_mono(M, W, order, tgt) # Mono-criterion version
6:     i, Σ_1, Σ ← 1, 0, sum(W)
7:     Π_1 ← [ ]
8:     while Σ_1 < tgt · Σ do
9:         Π_1.append(M[order[i]])
10:        Σ_1 ← Σ_1 + W[order[i]]
11:        i ← i + 1
12:    end while
13:    return [Π_1, M \ Π_1]
14: end function

15: function CutHalfWeight_multi(M, W, order, tgt) # Multi-criteria version
16:    Π_1, Π^best ← [ ], [M, [ ]]
17:    for i ← 1, length(M) do
18:        Π_1.append(M[order[i]])
19:        Π^new ← [Π_1, M \ Π_1]
20:        if imb(Π^new, tgt) < imb(Π^best, tgt) then
21:            Π^best ← Π^new
22:        end if
23:    end for
24:    return Π^best
25: end function
```

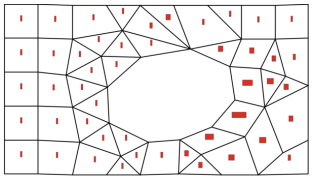
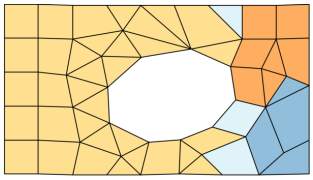
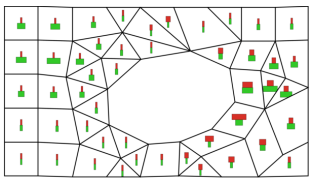
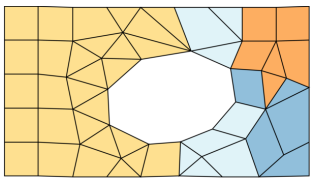
Multi-criteria Bipartitioning Algorithm

Boman *et al.* [2007] use RCB for multi-criteria mesh partitioning. Compared to the mono-criterion case, the only difference is the `CutHalfWeight_multi` function. They propose two alternatives: minimize the maximum imbalance between criteria, or the sum of the imbalance per criterion. In our case, as explained in Chapter 2, the objective is to select the bipartition leading to the minimal imbalance.

Example

Table 4.2.1 displays two 4-partitions: the first row of a mono-criterion mesh, and the second row of a two-criteria mesh.

Table 4.2.1 – Mono-criterion 4-partitions using RCB

Input weights	RCB partition	Statistics
		$imb_c(\Pi_p)(\%)$: P_1 P_2 P_3 P_4 c_1 −33.7 −0.6 +12.8 +21.5 Number of cells to send: P_1 P_2 P_3 P_4 total P_1 1 1 0 2 P_2 2 2 3 7 P_3 2 2 1 5 P_4 0 3 1 4 Imbalance 21.5% Communication 18
		$imb_c(\Pi_p)(\%)$: P_1 P_2 P_3 P_4 c_1 +15.2 +84.5 −36.1 −63.6 c_2 −15.5 −70.9 −6.6 +93.0 Number of cells to send: P_1 P_2 P_3 P_4 total P_1 2 3 0 5 P_2 2 2 2 6 P_3 3 2 0 5 P_4 0 2 0 2 Imbalance 93.0% Communications 18

The weight distributions are displayed on the left column, the bar in each cell of the mesh being proportional to its weight. In the second row, weights for the first criterion are in red, while weights for the second criterion are in green. The middle column shows the partitions computed by RCB, each color corresponding to one part. Finally, the last column gives the imbalance and communication volume produced by the two partitions.
The imbalances are given relatively to each part and criterion, and the communication table counts in the cell in the pth row and qth column the number of cells that unit p must send to unit q.
This example illustrates the kind of partitions that RCB may produce. In

particular, we can see that the imbalance of the partitions may be quite high, especially in this case where RCB puts the heaviest cells at the border of the partition.

Benefits of RCB

RCB is a simple and fast algorithm that works well for mono-criterion meshes with a regular geometry, when the place of the cut does not involve heavy vertices.

Limitations of RCB

Firstly, the communication cost may be high when irregular shapes like circles or holes appear, or when computation weights are not uniform (which is especially the case for multi-criteria partitioning).

Secondly, it does not take into account the objective function, and in particular the communication weights, assuming that a straight cut will lead to fewer communications.

Thirdly, there is no guarantee that the obtained partition will respect the balance constraints. In fact, as seen in the previous example, RCB can lead to very imbalanced partitions, especially in the multi-criteria case.

Fourthly, it is very dependent on the mesh orientation: a rotation of the axes leads to a different partition. The next section will describe the recursive inertial bisection algorithm, which computes its own axes.

4.2.2 Recursive Inertial Bisection (RIB)

Williams [1991] is one of the first to define the recursive inertial bisection algorithm (RIB), a method that is very similar to the RCB Algorithm 15 analyzed in the previous section. Although it would be possible to use a similar extension to multi-criteria than for RCB, in our knowledge, there has not been any usage of RIB in multi-criteria.

Overview

The RIB Algorithm 17, unlike RCB, adapts the axes to the mesh it partitions.

Algorithm 17 Recursive Inertial Bisection

```
1: function RIB(M, k, W, t, f)
2:     return RB(M, k, W, t, f, InertialBipart)      # See Algorithms 14 and 18
3: end function
```

Mono-criterion Algorithm

Algorithm 18 implements the core of RIB. It first computes a direction in which the mesh will be cut: unlike RCB, this direction is not given as input, but is computed by the OrthogonalInertialAxis function, whose implementation is not displayed here. Basically, it considers the weights as a mass, and the principal axis is set as the axis that minimizes the angular momentum when the mesh rotates around it. In our knowledge, there is no extension of this function to multi-criteria.

Then, as for RIB, the cut axis is set as orthogonal to the inertial axis, and a CutHalfWeight defined in Algorithm 16 is used.

Algorithm 18 Inertial Bipartition

```
1: function InertialBipart(M, W, t, f, tgt)
2:     d ← OrthogonalInertialAxis(M, W)      # No extension to multi-criteria
3:     order ← sortAscend([M[1].coord(d) .. M[n].coord(d)])
4:     return CutHalfWeight(M, W, order, tgt)          # See Algorithm 16
5: end function
```

Benefits

Unlike RCB, RIB is not dependent on the original orientation of the mesh, so it can consider meshes with more irregular shapes (though not every shape). Despite needing more computation than RCB, it remains quite fast compared to the topological algorithms that will be described in the next section.

Limitations

Basically, RIB relies on the same principle as RCB, that straight cuts will lead to small communication cost. Therefore, as for RCB, RIB does not directly take into account the communication cost function, so it may be deceived by non-uniform communication weights W_{com} (corresponding to edge or hyperedge weights).

Moreover, as for RCB, RIB may return partitions of high communication cost if it attempts to cut next to heavy cells. In this case, it may also return a very imbalanced partition. Finally, in our knowledge, there is no extension of the OrthogonalInertialAxis function in multi-criteria.

4.2.3 Spacefilling Curves (SFC)

Another algorithm quite similar to RIB and RCB has been introduced by Pilkington *et al.* [1994]. They claim that their algorithm, which we named SFC and which uses spacefilling curves to order the vertices, leads to more balanced

partitions than RCB. One main difference between SFC and RIB and RCB is that SFC does not need to rely on RB for k-partitioning.

A spacefilling curve is a curve that covers an entire continuous domain of space. In our case, as explained in Section 1.2, the domain is discretized to form a mesh, and the spacefilling curve defines an order of the cells such that cells that are close will have close numbers. Examples of spacefilling curves are the Peano curve or the Hilbert curve.

Algorithm 19 describes the SFC algorithm for the mono-criterion version. First, an order using a spacefilling curve is determined using the OrderSFC function, whose implementation is not defined here. Then, we use a similar algorithm than the CutHalfWeight_mono function defined in Algorithm 15, but so that each part gets roughly $1/k$th of the total weight instead of one half.

Algorithm 19 Partitioning with Spacefilling Curves

```
1:  function SFC(M, k, W, t, f)
2:      n, Σ ← length(M), sum(W)
3:      order ← OrderSFC(M)                          # Order the cells
4:      i ← 1
5:      Π ← [[ ] .k. [ ]]
6:      for p ← 1, k − 1 do
7:          Σ_p ← 0
8:          while i ≤ n and Σ_p < Σ/k do             # Mono-criterion version
9:              Π_p.append(M[order[i]])
10:             Σ_p ← Σ_p + W[order[i]]
11:             i ← i + 1
12:         end while
13:     end for
14:     Π_k ← [M[order[i]] ..n..i M[order[n]]]
15:     return Π
16: end function
```

It would be possible to adapt the CutHalfWeight_multi function in order to extend the SFC algorithm to multi-criteria partitioning, but we do not know of any work relating such adaptation, which does not seem to be straightforward when RB is not used.

Conclusion on the Analyzed Geometric Algorithms

To sum up, RCB, RIB and SFC are heuristics that partition a mesh using the geometric coordinates of the cells. They are quite simple and fast, but suffer from several limitations:

- the imbalance of the partition returned may be high, especially when the partition puts heavy cells in the border;

- they do not directly take into account the communication cost;
- extensions to multi-criteria for `RIB` and `SFC` are not straightforward.

The topological algorithms that will be analyzed in the next section can overcome some of these limitations, usually using more computations.

4.3 Topological Direct Partitioning Algorithms

This section defines algorithms that rely on the topology of the mesh. They always operate on the hypergraph or graph model of the mesh, which were defined in Sections 2.3 and 2.4. In this section, we will focus on algorithms that are usually used to build a partition from scratch, while the next section will deal with algorithms that refine an existing partition.

4.3.1 Spectral Graph Partitioning (`SpectralBipart`)

Donath and Hoffman [1973] and Fiedler [1975] introduced the spectral graph partitioning method, one of the first methods designed to partition a graph. More recently, Slininger [2013] provided a good overview of spectral graph partitioning, which does not work for hypergraphs as it uses the Laplacian matrix of a graph.

Definition 26 (Laplacian Matrix of a Graph)
Let $G = (V, E)$ be a graph. We call Laplacian matrix of G the matrix:

$$(L_G)_{i,j} = \begin{cases} worddeg(v_i) & \text{if } i = j, \\ -1 & \text{if } i \neq j \text{ and } (v_i, v_j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

If the edges are valued, i.e. if we consider the function $W_{com} : E \to \mathbb{R}_+$, then the Laplacian matrix becomes

$$(L_G)_{i,j} = \begin{cases} \sum_{ngbr \in \mathcal{N}(v_i)} W_{com}((v_i, ngbr)) & \text{if } i = j, \\ -W_{com}((v_i, v_j)) & \text{if } i \neq j \text{ and } (v_i, v_j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Remark

The Laplacian matrix is symmetric and its rank is 1 since the sum of a row or of a column is null.

Thus, 0 is an eigenvalue of L_G, of eigenvector 1_n (where $n = |V|$). Indeed, the multiplicity of the eigenvalue 0 is the number of connected components in G.

Overview

The spectral bisection algorithm is defined in Algorithm 20. It uses the recursive bisection Algorithm 14 on page 84 and uses the smallest non-zero eigenvalue to determine where to cut, whose eigenvector is called the Fiedler's vector.

Algorithm 20 Spectral Bisection

1: **function** `SpectralPart`(G, k, W, t, f)
2: **return** `RB`$(G, k, W, t, f,$ `SpectralBipart`$)$ # See Algorithms 14 and 5
3: **end function**

Algorithm

The `SpectralBipart` function is very similar to the `CoordBipart` and `InertialBipart` functions defined in Section 4.2. `SpectralBipart` first orders the vertices of the graph according to the Fiedler's vector $vec_{fiedler}$ corresponding to the smallest non-zero eigenvalue. Vertex of index i becomes the jth element if $vec_{fiedler}[i]$ is the jth smallest component of $vec_{fiedler}$.

Then, the `CutHalfWeight` function is called. It has been defined with the `RCB` algorithm: roughly, it fills the first part with the first elements of $order$ so that the imbalance of the obtained partition is minimized.

Algorithm 21 Spectral Bipartitioning Algorithm

1: **function** `SpectralBipart`$(G = (V, E), k, W, t, f, tgt)$
Require: $W_{com} : E \to \mathbb{R}_+$, the edge weights used to compute f
2: $vec_{fiedler} \leftarrow$ `ComputeVectorOfFiedler`(L_G, W_{com})
3: $order \leftarrow$ `argsortAscend`$(vec_{fiedler})$
4: **return** `CutHalfWeight`$(G, W, order, tgt)$ # See Algorithm 15
5: **end function**

Benefits

The spectral bisection algorithm, unlike the geometric approaches, takes into account the communication cost of the graph, through the edge weights that are inserted in the Laplacian matrix.

Limitations

As for the geometric approaches, the spectral bisection algorithm may return very imbalanced partitions when the partition is cut between heavy vertices.

Furthermore, the spectral bisection needs to compute eigenvalues and eigenvectors, which are complex problems that take time to solve, especially when the matrix is large. For example, one algorithm commonly used (according to von Luxburg [2007]) is the Lanczos algorithm, whose complexity is $O(e \times n^2)$, where n is the number of vertices of the graph and e its number of edges.

4.3.2 Greedy Graph Growing (GGG)

Overview

Graph growing comes from the idea that each part should be connected in order to minimize the communication costs. The objective of greedy graph growing is to obtain quickly a balanced partition whose communication cost is quite good, by growing the parts from so-called centers in a breadth-first-search manner.

The graph growing algorithm has been first proposed by George and Liu [1981], and Goehring and Saad [1995] and Jain *et al.* [1998] then defined some variations of this algorithm. In this section, we will focus on a popular variation introduced by Karypis and Kumar [1998a] for mono-criterion partitioning and Karypis and Kumar [1998b] for multi-criteria partitioning, called greedy graph growing (GGG).

The principal difference between graph growing and greedy graph growing lies in the way the partition is grown: graph growing uses a breadth-first search, while GGG grows the parts so that the communication cost of the partition is minimized at each step.

Mono-criterion Algorithm

Algorithm 22 details the implementation of GGG for the mono-criterion case. Using the GenerateSeeds function, the algorithm first generates $k-1$ random numbers, all different, that are called the "seeds". Then, in the GrowPartsFromSeeds function, each seed will be put in one part, and each part will grow until its weight reaches the target weight. The implementation of this function is not detailed here, because it is not defined clearly by Karypis and Kumar [1998a]. Roughly, it repeatedly moves the vertices to the lightest part, by selecting at each step, an unmoved vertex that decreases the most (or increases the least) the communication cost. The GrowPartsFromSeeds function stops when no more vertices can be moved.

Finally, because the partition returned by the GrowPartsFromSeeds depends a lot on the seeds, it is repeated n_{iter} times with different seeds at each step, selected by the GenerateNewSeeds function. While it is common to use the centers of the parts as new seeds for the next step, as defined by Goehring and Saad, Karypis and Kumar simply use random seeds at each step, in a stochastic manner.

Remark

Graph growing and greedy graph growing are a kind of stochastic algorithm (which were introduced in Section 3.2.7), because they iterate a number of times before returning the best partition found.

Algorithm 22 Mono-criterion Greedy Graph Growing

1: **function** GGG($G = (V, E)$, k, W, t, f)
Require: $n_{iter} \in \mathbb{N}^*$ number of times the parts are grown
2: $n \leftarrow$ length(V)
3: $\Pi^{best} \leftarrow [V, [\,] \overset{k-1}{\ldots} [\,]]$
4: $seeds \leftarrow$ GenerateSeeds($k-1$) # Generate $k-1$ different random numbers
5: **for** $iter \leftarrow 1, n_{iter}$ **do**
6: $\Pi \leftarrow$ GrowPartsFromSeeds($G, k, W, t, f, seeds$)
7: **if** $\Big(imb(\Pi) \leq t$ and $(imb(\Pi^{best}) > t$ or $f(\Pi) < f(\Pi^{best}))\Big)$ **then**
8: $\Pi^{best} \leftarrow \Pi$
9: **end if**
10: $seeds \leftarrow$ GenerateNewSeeds(Π, k)
11: **end for**
12: **return** Π^{best}
13: **end function**

Multi-criteria Partitioning Algorithm

Karypis and Kumar [1998b] extend GGG to multi-criteria bipartitioning, as defined in function GGG_Bipart_multi of Algorithm 23. This function starts by separating the vertices into different lists, depending on their heaviest normalized weight. Given a vertex v_i, the criterion for which its normalized weight is the heaviest is denoted by c^i_{max}: we have $W(v_i)[c^i_{max}] = \max_{c \in [1,\gamma]} W(v_i)[c]$.

Basically, GGG_Bipart_multi starts with all vertices in the first part Π_1, except for one vertex called the seed, which is put in the second part Π_2. Then, as long as the normalized weight of Π_2 for at least one criterion does not exceed 0.5, it selects the criterion c_{max} for which Π_1 is the heaviest. The "heaviest criterion" is determined by the function SelectCmax, which also defines how to deal with the case when the $list\_per\_c_{max}$ is empty for the criterion for which Π_1 is the heaviest (in this case, we consider the second criterion for which Π_1 is the heaviest, and so on).

Then, the function SelectBestGain determines the vertex v_i in Π_1 whose c^i_{max} is equal to c_{max}, and which leads to the smallest communication cost. This vertex is moved to Π_2 using the Move function, which also removes v_i from $list\_per\_c_{max}[c^i_{max}]$. Thus, once a vertex has been moved to Π_2, it cannot be moved back into Π_1 until the end of the algorithm.

Algorithm 23 Multi-criteria Adaptation of Greedy Graph Growing for Bipartitioning

function `SelectCmax`(Σ, $list\_per\_c_{max}$)
  $\Sigma s \leftarrow \Sigma$
  while $True$ **do**
    $c_{max} \leftarrow$ `argmin`(Σs)
    if $list\_per\_c_{max}[c_{max}] \neq [\,]$ **then return** c_{max} **end if**
    Σs.`remove`($\Sigma s[c_{max}]$)
  end while
end function

procedure `Move`(Π, i, W, $list\_per\_c_{max}$)
  $\Pi[i] \leftarrow 2$
  $c^i_{max} \leftarrow$ `argmax`($W[i]$)
  $list\_per\_c_{max}[c^i_{max}]$.`remove`($i$) # Updates
  $\Sigma \leftarrow [\Sigma[0] + W[i][0] \mathbin{.\!.} \Sigma[\gamma] + W[i][\gamma]]$
end procedure

function `GGG_Bipart_multi`($G = (V, E)$, k, W, t, f)
  Require: $\forall c, sum(W_c) = 1$ (normalized weights)
  $n \leftarrow |V|$
  $\Pi \leftarrow [V, [\,]]$
  $\Sigma \leftarrow [0 \mathbin{.\!.} 0]$
  $list\_per\_c_{max} \leftarrow [[\,] \mathbin{.\!.} [\,]]$
  for $i \leftarrow 1, n$ **do** # Sort the vertices into distinct lists depending on their c^i_{max}
    $c^i_{max} \leftarrow$ `argmax`($W[i]$)
    $list\_per\_c_{max}[c^i_{max}]$.`append`($i$)
  end for
  `Move`(Π, `random`$(1, n)$, W, $list\_per\_c_{max}$) # Move a random seed
  while $\forall c, \Sigma[c] < 0.5$ **do**
    $c_{max} \leftarrow$ `SelectCmax`(Σ, $list\_per\_c_{max}$)
    $i \leftarrow$ `SelectBestGain`($list\_per\_c_{max}[c_{max}]$)
    `Move`($\Pi, i, W, list\_per\_c_{max}$)
  end while
  return Π
end function

Therefore, this algorithm tries to achieve balance using a simple greedy scheme, which moves at each step a vertex in order to decrease the heaviest normalized weight of Π_1 the most. Nevertheless, it also tries to achieve a partition with a small communication cost, because among the possible movements, it selects the one that leads to the smallest communication cost.

Finally, Karypis and Kumar [1998b] use the recursive bisection algorithm in case of $k - partitioning$ with $k > 2$. In this case, the algorithm stops when the normalized weight of Π_2 for at least one criterion exceeds the target weight tgt.

Benefits

`GGG` involves fewer computations than spectral partitioning; it is a simple and fast heuristic. Moreover, unlike the geometric algorithms introduced in Section 4.2, it directly takes into account the communication cost of a partition.

Limitations

`GGG` provides no guarantee to find a balanced partition, especially for multi-criteria graph partitioning. Moreover, `GGG` favors partitions with connected parts, especially in the mono-criterion case. Nevertheless, partitions with non-connected parts should be considered in the case of non-unitary computation or communication weights.

Karypis and Kumar [1998b] report that in some cases, `GGG_Bipart_multi` will fail to return partitions respecting the balance constraints, in which case they use a rebalancing step. This step is also used in MeTiS' refinement phase, which will be detailed in Chapter 8.

Finally, Buluç *et al.* [2015] report that `GGG` needs to be run several times, because the communication cost of the partition highly depends on the seed positions.

Conclusion on `SpectralPart` and `GGG`

In this section, we have described the spectral graph partitioning and the greedy graph growing methods. These are direct algorithms: they build a partition from scratch. Both focus on finding a partition that is likely to possess a small communicaton cost, while the imbalance of the partition is not their main concern. Whereas the spectral graph partitioning computation cost is high, the greedy graph growing method is fast, and in the mono-criterion case, it can usually easily find balanced partitions, with a small communication cost. The capacity of `GGG` to find balanced partitions for multi-criteria meshes will be examined in Section 10.1.

The communication cost of the obtained partition is usually improved in a second step, using a refinement algorithm such as the ones that will be

introduced in the next section.

4.4 Topological Refinement Partitioning Algorithms

Refinement algorithms modify an existing partition, usually in order to improve its communication cost. The two methods that we will describe are local optimization algorithms, which were mentioned in Section 2.6. As their name tells, local optimization algorithms turn a partition onto another by small changes, of only one or two vertices at a time. Nevertheless, the partition obtained after all these little changes can differ greatly from the initial one.

Section 4.4.1 will describe the Kernighan-Lin algorithm (`KL`), which was historically the first local optimization algorithm for graph partitioning. `KL` inspired the Fiduccia-Mattheyses algorithm (`FM`), which will be described in Section 4.4.2.

4.4.1 Kernighan-Lin Algorithm (`KL`)

Overview

Kernighan and Lin [1970] define a vertex swapping algorithm called the Kernighan-Lin algorithm (`KL`). This algorithm is a local optimization algorithm that makes successive exchanges of vertices, each time so that the communication cost decreases the most.

Algorithm

`KL` is described in Algorithm 24. It performs a succession of passes, which are sequences of swaps. A swap is an exchange of two vertices that are not in the same part. At each step, the swap that decreases the most the communication cost is performed. Note that this decrease can be negative (becoming an increase) when there is no other choice. And, after that two vertices exchanged their parts, they are locked and cannot change of part until the end of the pass.

When there is no more possible swap, the pass ends. The partition of smallest communication cost reached in the pass is recovered. If the communication cost decreased compared to the previous pass, then a new pass starts over, and all vertices are unlocked. Otherwise, the algorithm ends.

A few more conditions usually characterize the `KL` algorithm. First, the selected vertex pair should always lead to a partition respecting the balance constraints. This is always the case when vertex weights are all equal to 1 and the initial partition is balanced but induces more work when computation weights are not unitary and especially for multi-criteria mesh partitioning. Thus, the `SelectBestSwap` function could be called `SelectBestSwapPreservingBalance`.

Algorithm 24 Kernighan-Lin

```
1: function KL(M, k, W, t, f, Π^ini)
   Require: imb(Π^ini) ≤ t
2:     Π^best ← Π^ini
3:     repeat                                          # Perform a pass
4:         f_old_best ← f(Π^best)
5:         Π ← Π^best
6:         while RemainSwap(Π) do                      # Condition for stopping the pass
7:             (i, j) ← SelectBestSwap(M, W, k, t, f, Π)
8:             Π[i], Π[j] ← Π[j], Π[i]                 # Perform a swap
9:             Lock(i, j)
10:            if f(Π) ≤ f(Π^best) then Π^best ← Π end if
11:        end while
12:    until f(Π^best) = f_old_best
13:    return Π^best
14: end function
```

Benefits

KL refine an existing balanced partition. When the computation costs are unitary, it is straightforward: if the input partition Π^{ini} is balanced, then any swap can be performed. Moreover, Kernighan and Lin define their algorithm as a hill-climbing algorithm, because they authorize performing exchanges that increase the communication cost, if this increase is later in the pass cancelled by a decrease of the communication cost.

Besides, as long as the imbalance of the input partition is smaller than the tolerance, the partition returned by KL will always respect the balance constraints.

Limitations

First, KL requires a balanced partition as input. Therefore, KL alone does not suffice as a partitioning algorithm; it must be combined with a direct partitioning algorithm. Then, the greatest drawback of KL is its computation cost. Indeed, its complexity is in $O(n^2 \log n)$ because of the SelectBestSwap function. Indeed, in order to find the swap leading to the lowest communication cost, it must try out $\frac{m(m-1)}{2}$ pairs, where m is the number of vertices unlocked.

The next section introduces another local optimization algorithm, which instead of performing exchanges, moves one vertex at a time.

4.4.2 Fiduccia-Mattheyses Algorithm (FM)

Overview

Fiduccia and Mattheyses [1982]'s algorithm, FM, is close to the KL algorithm introduced in the previous section. Instead of considering exchanges, Fiduccia and Mattheyses allow any single vertex to switch of part and call this action a *move*. Moreover, a *gain table* records the reduction in communication cost of each move, enabling constant time access to the best move. After a move, the gain table is updated for the vertex moved and its neighbors.

Algorithm

The FM Algorithm 25 is very similar to KL. It performs a sequence of moves (a move is the change of part of *one* vertex) that is called a pass. Each vertex moved is then locked until the end of a pass. During a pass, we keep track of the best partition found. If its communication cost strictly decreased during the pass, a new pass begins: the best partition found is retrieved and all vertices are unlocked. Otherwise, the algorithm stops.

The selection of the move is performed using a gain table. In the bipartitioning case, the gain table stores in cell g the vertices which, when changed of part, decrease the communication cost by g units (we say that these vertices are of *gain* g). Using Fiduccia and Mattheyses's gain table structure, the move of greatest gain is retrieved in constant time. After moving a vertex v, the gain table must be updated for the vertices whose gain changes. When considering the *edgecut*, the gain changes for v and its neighbors. When considering the $cut_{\lambda-1}$, the gain changes for v, its neighbors and the neighbors of its neighbors.

Note that as for KL, the selected move must lead to a partition that still respects the balance constraints. If the initial partition Π^{ini} respects them as required, this guarantees that the returned partition will always respect the constraints. Thus, among the possible moves, we select the one that decreases the most the communication cost, or, in other words, the move of best gain. Note that the gain can be negative, which helps FM to escape from a local minimum.

Karypis and Kumar [1998d] proposed two modifications of FM. Firstly, they authorize only the boundary vertices to move, and secondly, a pass may be stopped after x moves of negative gains made in a row, where x is an algorithmic parameter. These two improvements accelerate even more the FM algorithm, and will be discussed in Chapter 8.

Pellegrini [2007] proposed to restrain the possible moves to a band that contains only the vertices at a distance at most 3 from the border of Π^{ini}. This speeds up the algorithm, since fewer gains have to be computed (reducing the complexities of the `GainTableInit` and `GainTableUpdate` functions) and fewer moves are considered (possibly reducing the `SelectMove` complexity

Algorithm 25 Fiduccia-Mattheyses

```
1: function FM(M, k, W, t, f, Π^ini)
   Require: imb(Π^ini) ≤ t
2:     Π^best ← Π^ini
3:     repeat                                          # Perform a pass
4:         Π ← Π^best
5:         f_old ← f(Π^best)
6:         gains, locked ← GainTableInit(M, f, Π), [ ]
7:         while RemainMove(Π, t) do                   # Condition for stopping the pass
8:             i, p ← SelectMove(gains, locked)
9:             Π[i] ← p
10:            Lock(i)
11:            GainTableUpdate(gains, i, p)
12:            if f(Π) ≤ f(Π^best) then Π^best ← Π end if
13:        end while
14:    until f(Π^best) = f_old
15:    return Π^best
16: end function
```

and stopping the passes earlier). Moreover, Pellegrini reported that, when coupled with the multilevel algorithm (which will be defined in Section 4.5), such restrictions on the possible moves also decreased the communication cost of the returned partition, possibly because FM is less likely to get trapped in a local minimum far from the global minimum sketched at the coarsest level and from which Π^{ini} is close.

Benefits

FM is a fast algorithm which, given an initial partition respecting the balance constraints, aims at reducing its communication cost. When using the graph model $G = (V, E)$, at each step, the move of best gain is obtained in constant time. Then, the gain for the vertex moved and its neighbors are updated, and the vertex is locked so that it will not move anymore during this pass.

As the KL algorithm defined in the previous section, FM is a hill-climbing algorithm able to bypass local minima. Indeed, it performs moves of negative gain if they decrease the communication cost obtained at the end of the pass. Moreover, despite being a local optimization algorithm, at the beginning of a pass, any partition is reachable in less than n moves, provided that these moves respect the balance constraints.

Limitations

First, FM needs an initial partition that respect the balance constraints. Otherwise, the returned partition is not guaranteed to be a solution.

Then, considering that a move can be performed only if it leads to a partition respecting the balance constraints, some partitions may be inaccessible. This can be illustrated using the fitness landscape induced by FM. A fitness landscape (as introduced in Definition 25) is a triplet $\Lambda_{\mathcal{I}} = (\mathbb{X}_{\mathcal{I}}, n_{\texttt{FM}}, f)$ where $\mathcal{I}$ is the instance (mesh, weights, tolerance) and $n_{\texttt{FM}}$ a neighborhood structure. $n_{\texttt{FM}}$ defines how one can move from one solution to another; here, we move from one partition to another by doing a move so that the obtained partition respects the constraints. So, being able to reach any partition from any initial partition would mean that $\Lambda_{\mathcal{I}}$ is connected.

A main drawback of FM is its increased complexity when dealing with partitions of more than two parts ($k > 2$). Sanchis [1989] first described an implementation of FM for k-partitioning, using $k(k-1)$ gain tables. Indeed, one vertex has to maintain a gain for each part it does not belong to, so the search and data structure become more complex.

Conclusion on KL and FM

This section has described two refinement techniques. Both need an input partition respecting the balance constraints. The first algorithm, KL, uses vertex swaps to reduce the communication cost, but forbid swaps leading to a partition that does not respect the constraints. The second algorithm, FM, does the same thing, but changes vertices of part one after another (one change is called a move) instead of performing swaps. Both algorithms need a balanced partition as input, and provided that the input partition respect the balance constraints, the returned partition is guaranteed to also respect these constraints.

The high time complexity of KL has led to the supremacy of FM, whose complexity is far lower. Nevertheless, note that for both of them, the extension to direct k-partitioning (when $k > 2$) leads to increased complexity.

Finally, whereas any partition is reachable through a reduced number of moves, as noted by Pellegrini [2007], FM and KL may be stuck in a local minimum far worse than the global minimum. The following multilevel algorithm remedies to this drawback of using a refinement algorithm, but so that the partition given as input will be close to the optimal solution.

4.5 The Multilevel Algorithm

Overview

The multilevel algorithm was first introduced by Bui and Jones [1993], Barnard and Simon [1994], van Driessche and Roose [1994] and Hendrickson

and Leland [1995]. It is described in Algorithm 26 and illustrated in Figure 4.5.1.

As explained by Barnard and Simon [1994], it enables one to compute a partition fast. Moreover, the multilevel algorithm reduces the search space by getting rid of the partitions that are unlikely to be solutions of small communication cost. Therefore, if indeed the solutions of small communication cost are preserved, the partition returned can be of high quality.

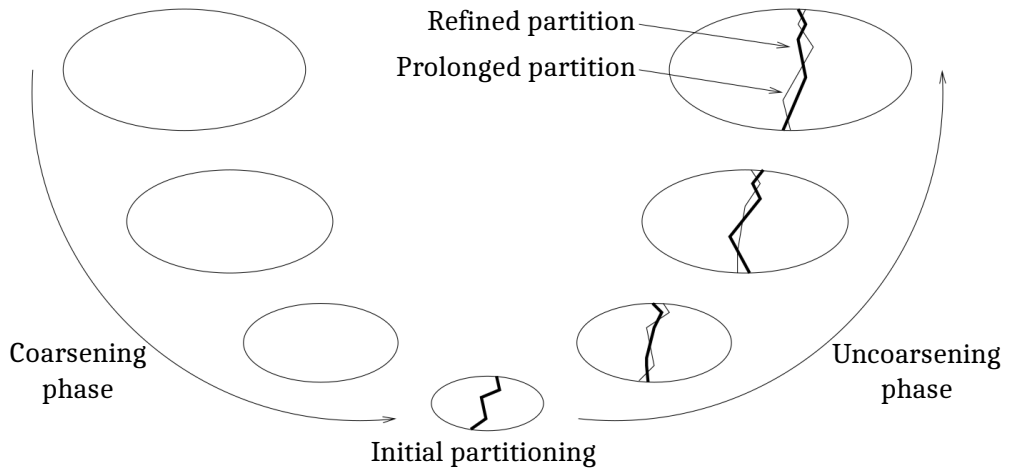


Figure 4.5.1 – The Three Phases of the Multilevel Algorithm (image from the Scotch-6.0.4manual of Pellegrini [2008])

In order to reduce the search space, the multilevel algorithm reduces the size of the graph by clustering some of its vertices. This is the *coarsening phase*, which is divided into several "levels" of coarsening. In Algorithm 26, the `Coarsen` function computes successive aggregations, each one corresponding to one level.

When the produced graph is small enough, a direct partitioning algorithm is used through the `Partition` function. This is the *initial partitioning phase*, in which the search space is reduced. Finally, the partition found at a lower level is prolonged to the upper level (function `Prolong`), and refined using a local refinement algorithm (function `Refine`). This is the *expansion* or *uncoarsening phase*, which ends when a partition of the original graph has been obtained and refined.

The next sections will detail each of the three phases of the multilevel algorithm.

Coarsening Phase

The aim of the coarsening phase is to reduce the search space. However, reducing the search space may also reduce the solution space, so this reduction

Algorithm 26 Multilevel Algorithm

function `Multilevel`(M, k, W, t, f)
Require: `Coarsen`, `Partition` and `Refine` functions, and $n_{coarse} \in \mathbb{N}^*$
  $level \leftarrow 0$
  $M_0^{coarse}, W_0^{coarse} \leftarrow M, W$
  while `length`(M^{coarse}) $> n_{coarse}$ **do** # Coarsening phase
    $M_{level+1}^{coarse}, W_{level+1}^{coarse} \leftarrow$ `Coarsen`($M_{level}^{coarse}, W_{level}^{coarse}$)
    $level \leftarrow level + 1$
  end while
  $\Pi \leftarrow$ `Partition`($M_{level}^{coarse}, k, W_{level}^{coarse}, t, f$) # Initial partitioning phase
  while $level > 0$ **do** # Expansion phase
    $\Pi, M_{level-1}^{coarse}, W_{level-1}^{coarse} \leftarrow$ `Prolong`($\Pi, M_{level}^{coarse}, W_{level}^{coarse}$)
    $\Pi \leftarrow$ `Refine`($M_{level-1}^{coarse}, k, W_{level-1}^{coarse}, t, f, \Pi$)
  end while
end function

must try to:

- remove from the solution space the partitions that have a high communication cost. However, the optimal solution for the coarsened graph may not be the optimal solution for the original graph;
- preserve the optimal solution, which means that the optimal solution in the original search space is also the optimal solution in the reduced search space.

The reduction of the search space is done by clustering some vertices. The basic idea, in order to fulfill the two objectives of the coarsening phase stated above, is to aggregate together vertices that belong to the same part in the optimal partition.

Usually, the clustering is performed by matching vertices into pairs. At each level, one vertex is matched with at most one other vertex. Usually, only vertices that are neighbors are matched together. Several strategies can be used to choose which pairs of neighbors will be matched together:

- random matching: an unmatched vertex is matched with one of its unmatched neighbors. This matching scheme was the first ever used.
- heavy-edge matching (`HEM`): an unmatched vertex is matched with its unmatched neighbor along the edge of heaviest weight. This scheme was proposed by Karypis and Kumar [1998a] and has been used by most partitioning tools since then. The idea is twofold: firstly, removing the heaviest edges should eliminate partitions of high communication cost, secondly, the heaviest edges are less likely to be cut in the optimal partition.

There are other algorithmic choices introduced in partitioning tools such as MeTiS or Scotch, which modify the coarsening phase. These will be described

and studied in Chapter 6. The following will define how to build a coarsened graph or hypergraph from a matching.

Definition 27 (Matching Function)

Given a mesh, a graph or a hypergraph, we call V the set of cells or vertices. A matching is a function $\mathcal{M} : V \to V^c \subset V$ such that $\forall v^c \in V^c$, there is at least one and at most two vertices in V whose image by $\mathcal{M}$ is v^c.

Two vertices that have the same image are said to be matched together.

Remarks

- A matching is a surjective function.
- In practice, $\mathcal{M}(v) = v$ or $\mathcal{M}(v) \in \mathcal{N}(v)$.

Definition 28 (Coarsened Graph)

Given a graph $G = (V, E)$, computation weights $W : V \to \mathbb{R}_+^\gamma$, communication weights $W_{com} : E \to \mathbb{R}_+^*$ and a matching $\mathcal{M} : V \to V_c$, the coarsened graph is $G^c = (V^c, E^c)$, such that:

- $V^c = \{\mathcal{M}(v), v \in V\}$,
- $E^c = \left\{ (\mathcal{M}(u), \mathcal{M}(v)) \text{ such that } \begin{cases} (u,v) \in E \\ \mathcal{M}(u) \neq \mathcal{M}(v) \end{cases} \right\}$.

The weight functions are projected in the following way:

$$W(v^c) = \sum_{v \in V,\ \mathcal{M}(v)=v^c} W(v) \ ,$$

$$W_{com}((u^c, v^c)) = \sum_{(u,v)\in E,\ \mathcal{M}(u)=u^c,\ \mathcal{M}(v)=v^c} W_{com}((u,v)) \ .$$

Remark

Since $W(v)$ is a vector, $W(v) + W(u)$ is a classic vector addition.

In the coarsened graph, when two vertices are matched together, their common edge is dropped. If two edges end on the same vertex, they are fused in the coarsened graph: the weight of the resulting edge is the sum of their weights. The other edges are kept as they are. Also, the weight of a coarsened vertex is the sum of the weights of the vertices that formed it.

Definition 29 (Coarsened Hypergraph)

Given a hypergraph $H = (V, E)$, computation weights $W : V \to \mathbb{R}_+^\gamma$, communication weights $W_{com} : E \to \mathbb{R}_+^*$ and a matching $\mathcal{M} : V \to V^c$, the coarsened hypergraph is $H^c = (V^c, E^c)$. We use the notations introduced in Section 2.3 where $e_{v^c} \in E^c$ is the hyperedge of center $v^c \in V^c$, leading to the definition:

- $V^c = \{\mathcal{M}(v),\ v \in V\}$,

- $E^c = \left\{ \left\{ \mathcal{M}(u),\ u \in \bigcup_{v \in V,\ \mathcal{M}(v)=v^c} e_v \right\},\ v^c \in V^c \right\}$.

Note that, in the above definition, V^c is considered as a set, and E^c as a multiset of sets (a hyperedge is a set).

The weight functions are projected in the following way:

$$W(v^c) = \sum_{v \in V,\ \mathcal{M}(v)=v^c} W(v) \ ,$$

$$W_{com}(e_{v^c}) = \sum_{e_v \in E,\ \mathcal{M}(v)=v^c} W_{com}(e_v) \ .$$

The coarsened hypergraph vertices are built similarly as those of a coarsened graph. The difference is how to build the coarsened hyperedges. Given a coarsened vertex v^c, which is the fusion of vertices u and v, the coarsened hyperedge of center v^c is the union of the hyperedges of centers u and v. Besides, the weight of e_{v^c} is the sum of the weights of e_u and e_v.

Remarks

- Even if the computation or communication weights given to the original graph are unitary, after a matching, the possibility that they stay unitary is very small, since the weight of a coarsened vertex is the sum of the weights of the vertices matched to form it. The same holds for edges or hyperedges.
- Karypis and Kumar [1995] have studied the evolution of the degrees of the vertices. We will investigate this property in Section 6.2.
- Chekuri and Khanna [1999] and Chevalier and Safro [2009] compare coarsening schemes allowing vertices to belong to different sets with some probabilities. This scheme is complicated to use, especially during the uncoarsening phase, in which it is thus difficult to know to which part an uncoarsened vertex belongs.

The coarsening phase reduced the search space by reducing the size of the graph. In the next section, the coarsened graph will be partitioned: this is the initial partitioning phase.

Initial Partitioning Phase

The coarsening phase has reduced the search space, trying to get rid of partitions with high communication cost. The role of the initial partitioning phase can be seen in various ways:

- to return the optimal solution of the reduced search space, which, if the coarsening phase fulfilled its task, is also the optimal solution of the original search space;
- to return a solution, that is, a partition that respects the balance constraints. This solution is not optimal but is expected to become so thanks to the refinement in the expansion phase;

- to return a partition which is not necessarily a solution, but whose communication cost is small. The imbalance will be fixed in the expansion phase.

Of course, the first option is in practice impossible to fulfill, because there is no guarantee that the optimal solution of the reduced search space is the optimal solution of the original. This is why we use a refinement algorithm in the expansion phase.

Then, between the two other possibilities, the choice varies between partitioning tools. For example, MeTiS has chosen the third one: for the initial partitioning phase, the tolerance is relaxed in order to enlarge the solution space and accept partitions of smaller communication cost. The tolerance is then tightened during the expansion phase.

On the opposite, Barat *et al.* [2016] have chosen the second option: focusing on returning a solution, be it not optimal for the communication cost. For multi-criteria instances, they show that using this policy, they obtain partitions respecting the balance constraints, unlike MeTiS, which often returns partitions that are not solutions. They also study a modified version of MeTiS called "MeTiS_Eq", which does not relax the tolerance, and show that the communication cost of the solutions returned by "MeTiS_Eq" does not degrade on average over the solutions of MeTiS.

In this document, we will study in detail the effect of the initial partitioning algorithm. More precisely, Chapter 7 will study new initial partitioning algorithms, and Section 10.1 will compare the imbalance of the partitions of the coarsest graph that they return. When they are used in a multilevel scheme, we will also study the communication cost of the partitions of the original graph that they induce.

Expansion Phase

The expansion phase successively prolongs and refines the partition obtained at a coarser level.

Definition 30 (Prolongation or Uncoarsening)
Let $\mathcal{M} : V \to V^c$ be a matching between the set of vertices V and its coarsened set of vertices V^c. Let Π^c be a partition of V^c.

We prolong Π^c from V^c to V by defining the partition Π of V, such that $\mathcal{M}(v) \in \Pi_p \iff v \in \Pi_p$.

In order to perform the refinement, many partitioning tools use the `FM` algorithm defined in Section 4.4.2, with some variations that will be discussed further, in Chapter 8.

Nevertheless, one common assumption is that the partition found at a lower level is close to the optimal at the current level. The objective of the refinement technique is to fix a possible gap between the raw output of the prolongation

and the optimal solution at this level. Even more, according to Pellegrini [2007], it is better to stay close to the partition of the lower level, which is more likely to be close to the optimal solution. Therefore, he proposes to restrict the possible moves of `FM` to a "band", composed of the vertices at a distance at most 3 from the border of the partition of the lower level.

Benefits of the Multilevel Algorithm

The multilevel algorithm has two main benefits. Firstly, it computes a partition faster with direct heuristics, because the only direct partitioning algorithm is run on a graph of reduced size. Then, the refinement algorithm is likely to operate only on the boundary vertices, which are usually far less than the total size of the graph, hence a smaller complexity. The actual complexity of the multilevel algorithm is hard to evaluate, because coarsening time and complexity depends on the graph topology of the considered graph. Usually, it is considered linear in the number of vertices.

Second, the multilevel algorithm returns solutions of "good" quality, that is, with a small communication cost. Indeed, the coarsening phase gets rid of a lot of "bad" candidate solutions. In case the optimal solution of the reduced search space is not the optimal solution of the original search space, it should not be far, and the refinement algorithm should succeed in finding the optimal solution of the original search space. Indeed, note that refinement algorithms have more degrees of freedom at finer levels, because even if a coarsen vertex cannot be moved at a coarsen level, its constituent vertices might move at finer levels.

These two major advantages of the multilevel algorithm have led many graph partitioning tools to rely on it.

Limitations of the Multilevel Algorithm

Yet, the multilevel algorithm has its drawbacks. Firstly, coarsening the graph makes the weights of the vertices grow, which reduces the solution space, as will be seen in Chapter 5. Consequently, Chapter 6 will deal with coarsening schemes that take into account the computation weights.

Besides, the multilevel algorithm can be improved. Most of its parameters were set experimentally or based on intuition, but some variations can lead to great improvement in the communication cost of the returned solution. As illustrated by the variety of implementations of this method among the partitioning tools, there is a lack of studies for setting some parameters. Chapters 6 and 8 will describe and study variations of the coarsening and refinement phases.

In this chapter, we have described the multilevel algorithm. It is currently the most efficient framework to partition meshes, graphs or hypergraphs, both

thanks to its run time and to the quality of the returned partitions, whose communication cost is small on average.

Conclusion on Mesh Partitioning Algorithms

At this stage, we have explained in Chapter 1 the motivations underlying the multi-criteria mesh, hypergraph and graph partitioning problems. Chapter 2 has formulated these problems and discussed their benefits and limitations, as well as the definition of one subproblem, the vector-of-numbers partitioning problem.

Chapter 3 has detailed the existing approaches to address the vector-of-numbers partitioning problem. These will be compared to those dealing with the vector-of-numbers partitioning problem in Chapter 7. Indeed, this chapter will introduce new initial partitioning algorithms that focus on obtaining a partition respecting the balance constraints, by addressing the vector-of-numbers partitioning problem.

The present chapter has introduced the existing heuristics addressing the multi-criteria mesh, hypergraph or graph partitioning problems. Among these heuristics, the multilevel algorithm is currently the one that returns the solutions with the smallest communication cost. However, the current approaches were designed for mono-criterion mesh, hypergraph or graph partitioning, and can return partitions that do not respect the balance constraints. Moreover, as we will see in Chapters 6 and 8, some parameters of the multilevel algorithm can be tuned in order to improve firstly the probability to find a solution, and secondly the probability to find an optimal solution.

Part II

Approach

Chapter 5

Analysis of the Fitness Landscapes of the Multi-criteria Mesh Partitioning Problem

Contents

In the previous part, we have first formulated in Chapter 2 the problems that this thesis focuses on, starting with the mesh partitioning Problem 2. All problems try to partition an instance $\mathcal{I} = (M, k, W, t, f)$, where M is a mesh, a graph or a hypergraph, k is the required number of parts, $W : M \to \mathbb{R}_+^\gamma$ are the computation weights of each cell or vertex for each criterion, $t \in \mathbb{R}_+$ is the imbalance tolerance, and $f : \mathfrak{P}(M) \to \mathbb{R}$ is the communication cost function that we want to minimize.

Given an instance, the problem is to find a partition whose imbalance is smaller than the tolerance t (the existing algorithms addressing this vector-of-numbers partitioning problem were examined in Chapter 3). Such a partition is called a solution, which may not be an optimal solution, though. The set of all solutions is called the solution space and has been defined in Section 2.6.

Among all solutions, the objective is to find an optimal solution, which is one that minimizes a given function f (called the objective function).

In this chapter, we do not describe algorithms aiming at finding the optimal solution, but rather we study the existence of a solution, and the reachability of the optimal solution from other solutions when using local optimization algorithms. Indeed, the algorithms presented in Section 4.4, namely `FM` and `KL`, refine an input solution by performing "local" changes, meaning that a few vertices switch to another part at a time. These refinement algorithms topologically define a neighborhood relationship between solutions, which thus defines the connection of the solution space. If the solution space associated with an algorithm is connected, it means that, given any initial solution, this refinement algorithm can return any solution, and in particular it can return an optimal one.

Section 5.1 will first analyze the difference between mono- and multi-criteria partitioning. Then, Section 5.2 will study the size of the solution space, which means that this section does not depend on any refinement algorithm. On the opposite, Section 5.3 will focus on the connection of the solution space, which is based on the way the considered refinement algorithm passes from a partition to another.

5.1 How a Multi-criteria Instance Differs from a Mono-criterion Instance

In this section, we will examine if a mono-criterion mesh partitioning algorithm can be used on a multi-criteria mesh. First, we will discuss relationships between the solution space of a multi-criteria instance and the solution spaces of its mono-criterion subinstances. Then, we will study if some characteristics on a multi-criteria weight distribution can lead a mono-criterion algorithm to perform well.

Relation with the Subinstances Solution Spaces

Given an instance $\mathcal{I} = (M, k, W : M \to \mathbb{R}_+^\gamma, t, f)$, we define the (mono-criterion) subinstances of $\mathcal{I}$ as, for $c \in [1, \gamma]$, $\mathcal{I}_c = (M, k, W_c : m \mapsto W(m)[c], t, f)$. We denote by $\mathbb{S}_\mathcal{I}$ the solution space of the instance $\mathcal{I}$ and $\mathbb{S}_{\mathcal{I}_c}$ the solution space of its subinstance $\mathcal{I}_c$.

Then, recall that a solution is a partition whose imbalance is smaller than t *for all* criteria, so:

$$\mathbb{S}_\mathcal{I} = \bigcap_{c=1}^{\gamma} \mathbb{S}_{\mathcal{I}_c} \ .$$

Therefore, the size of the solution space of a multi-criteria instance is bounded by the sizes of the solution spaces of its corresponding mono-criterion

instances: $|\mathbb{S}_{\mathcal{I}}| \leq min_c(|\mathbb{S}_{\mathcal{I}_c}|)$.

Besides, this equation illustrates a major difference between solving a multi-criteria problem and a mono-criterion problem. Indeed, it is not enough to be able to find a solution to a mono-criterion problem; to solve a multi-criteria problem, we need to find a partition which is a solution for all the corresponding mono-criterion problems. However, depending on some characteristics on the weight distributions, a mono-criterion algorithm can perform well on a multi-criteria instance.

Qualitative Analysis Using the Weight Distribution

Firstly, consider a multi-criteria weight distribution for which each vertex has the same weight for each criterion. Partitioning this instance is similar to partitioning a mono-criterion instance. This case may be extreme, but given some similarities between the weight distributions of each criterion (*i.e.* , heavy and light vertices are the same for all criteria), the imbalance for all criteria would be always roughly the same, which means that a mono-criterion algorithm would be able to handle this multi-criteria instance as "easily" as a mono-criterion instance.

We consider now the opposite: if the vertices of heavy weight differ for each criterion, then it is easy to rebalance some criterion without unbalancing the others. Therefore, applying a mono-criterion refinement algorithm a few times can allow one to find a partition balanced for all criteria.

These characteristics can be taken into account in Chapter 9, in which we introduce a model to generate multi-criteria weight distributions.

Conclusion

In this section, we have analyzed whether a mono-criterion algorithm can perform well on a multi-criteria instance. In particular, we have explained that the solution space of a multi-criteria instance is the intersection of the solution spaces of the corresponding mono-criterion instances.

Besides, even though we did not investigate further on this property, it might be possible to determine if a mono-criterion algorithm can succeed in returning a balanced partition for a multi-criteria instance by analyzing its weight distribution.

5.2 Study of the Size of the Solution Space

This section focuses on the size of the solution space, the space of all partitions of imbalance smaller than or equal to the input tolerance t. To begin with, in Section 5.2.1, we will define a sufficient condition for the existence of a solution in the mono-criterion case. Then in Section 5.2.2, we will perform a

study to estimate the size of the solution spaces of the instances that we will use later to compare partitioning algorithms. We will also provide an upper bound on the size of the solution space.

5.2.1 Bounding the Maximal Vertex Weight Ensures the Existence of a Solution

Theorem 1 (Existence of a Solution to a Mono-criterion Bipartitioning Problem)

We consider the mono-criterion case ($\gamma = 1$) and aim at bipartitioning ($k = 2$) a set of non negative numbers W with an imbalance tolerance $t \in [0, 1]$.

$$\max_{w \in W} \frac{w}{\Sigma} \leq t \implies \textit{the solution space is not empty.}$$

Proof

Assuming $\max_{w \in W} \frac{w}{\Sigma} \leq \frac{t}{2}$, we will build a bipartition of W that is a solution.

Let $\Pi = \{\varnothing, W\}$ be a partition of W. Using the fact that as long as $\Sigma_1 < \Sigma$, then Π_2 is not empty, we move numbers from Π_2 to Π_1 until Σ_1 becomes greater than or equal to $\frac{\Sigma}{2} \times (1 - t)$. By construction, at this step, there is a number $w \in \Pi_1$ such that $\Sigma_1 - w < \frac{\Sigma}{2} \times (1 - t) \leq \Sigma_1$.

We will now prove by contradiction that Π is a balanced partition.

Let us assume that Π is not balanced. First, we have $\Sigma_1 > \Sigma_2$, otherwise Π would be solution because we would have $\frac{\Sigma}{2} \times (1 - t) \leq \Sigma_1 \leq \frac{\Sigma}{2}$. So, this means that $imb(\Pi) = \max(imb(\Pi_1), imb(\Pi_2)) = imb(\Pi_1)$.

Then, Π is not a solution means that:

$$\begin{aligned}
imb(\Pi) > t &\iff imb(\Pi_1) > t \\
&\iff \frac{\Sigma_1 - \frac{\Sigma}{2}}{\frac{\Sigma}{2}} > t \\
&\iff \frac{w + \Sigma_1 - w - \frac{\Sigma}{2}}{\frac{\Sigma}{2}} > t \\
&\iff w > t\frac{\Sigma}{2} + \frac{\Sigma}{2} - (\Sigma_1 - w) \\
&\implies w > t\Sigma \qquad \text{since } \Sigma_1 - w < \frac{\Sigma}{2} \times (1 - t) \ ,
\end{aligned}$$

which contradicts the assertion that $\frac{w}{\Sigma} \leq t$. Thus, $imb(\Pi) \leq t$, which means that Π is a balanced partition and therefore that the solution space is not empty.

□

Table 5.2.1 – Numerical values of $\frac{u}{\Sigma}$ for the instances that will be used to compare partitioning algorithms in Chapter 10. u is the maximum weight of a vertex: $u = \max(W)$.

Mesh	n	w	$\frac{u}{\Sigma}$
Mushroom	22800	1	0.00045018
		2	0.00055498
		3	0.00053642
Onera	85567	1	0.00097113
		2	0.00121591
		3	0.00163818
Wave	156316	1	0.00082836
		2	0.00092409
		3	0.00086265
Linkrodsok1	174218	1	0.00000601
		2	0.00000601
		3	0.00000601
Shock	1196252	1	0.00002105
		2	0.00000492
		3	0.00002716

Remark

The opposite is not true. For example, the set $W = \{1, 1\}$ has a solution for $t = 0$ (which is $\Pi = \{\{1\}, \{1\}\}$, whereas $\max_{w \in W} \frac{w}{\Sigma} = 0.5 > \frac{t}{2} = 0$).

Unfortunately, we were not able to find a similar result in the multi-criteria case. Nevertheless, since the solution space of a multi-criteria instance is the intersection of the solution spaces of its corresponding mono-criterion instances, having normalized weights below $\frac{t}{2}$ for each criterion means that the multi-criteria instance is more likely to have a solution.

Numerical Application. Table 5.2.1 displays $\frac{u}{\Sigma}$ for each of the 45 instances that will be used in Chapter 10 to compare partitioning algorithms. These instances will be described in Section 9.1. They are composed of 5 meshes of n cells, for which 3 weight distributions of three criteria each have been generated. Since the instances are multi-criteria, $\frac{u}{\Sigma} = \max_c(\frac{u_c}{\Sigma_c})$.

Each instance will be partitioned with 3 imbalance tolerances: $t \in \{0.05, 0.01, 0.002\}$. For all instances but $(\textit{Onera}, w \in \{2, 3\}, t = 0.2\%)$, $\frac{u}{\Sigma} < \frac{t}{2}$. Note that in the multi-criteria case, this does not guarantee the existence of a solution.

5.2.2 Estimation of the Size of the Solution Space

To increase readability, we will restrict our study to the bipartitioning case, but similar estimations can be computed for k-partitioning, with $k > 2$.

Eliminating Some Invalid Partitions

First, recall that the solution space $\mathbb{S}$ is included in the search space $\mathbb{X}$. In the mesh bipartitioning (or vector-of-numbers bipartitioning) case, the search space is the set of all bipartitions of the mesh (or of the vector-of-numbers). Given a mesh M of n cells, the set of all bipartitions has 2^{n-1} elements.

Indeed, as explained in Section 2.6, a partition can be represented as a vector of size n of ones and twos, whose ith value indicates the part to which the ith cell belongs. This yields 2^n different vectors. However, given a partition Π and a partition Π' in which $\forall m \in M$, $\Pi(m) \neq \Pi'(m)$ ($\Pi(m)$ and $\Pi'(m)$ are the parts to which m belongs in partitions Π and Π'), Π and Π' designate the same partition but will be encoded by different vectors. Therefore, there are $2^n/2 = 2^{n-1}$ possible partitions, which is the size of the search space. As the solution space is included in the search space, its size is at most 2^{n-1}.

We can improve this bound. Firstly, we consider the mono-criterion case, and number the vertices by decreasing weight, so that $W(v_1) \geq ... \geq W(v_n)$. Let us denote by v_s the vertex such that:

$$\sum_{i=1}^{s-1} W(v_i) < \frac{\Sigma}{2}(1-t) \leq \sum_{i=1}^{s} W(v_i) \ .$$

s is the minimum number of vertices that a part must process in order to be a solution. Indeed, $(v_1, ..., v_{s-1})$ are the $s-1$ vertices of greatest weight, so any other set of $s-1$ vertices would have a weight smaller than $\sum_{i=1}^{s-1} W(v_i)$ and thus smaller than $\frac{\Sigma}{2}(1-t)$.

Therefore, any partition for which one part has less than s vertices is not a solution, which eliminates $S_{inv} = \sum_{i=1}^{s-1} \binom{n}{i}$ partitions, hence the following bound on the number of solutions: $|\mathbb{S}| \leq 2^{n-1} - S_{inv}$.

For a multi-criteria instance, if we denote by s_c the s value corresponding to the weight distribution of criterion c, then $s = \max_c(s_c)$, because s_c is the minimal number of cells that one part must hold in order for a partition to be solution.

Numerical Application. Table 5.2.2 displays the s and $|S_X|$ corresponding to each of the 45 instances mentioned previously and that will be described in Section 9.1. We will only report the values for the tightest tolerance, $t = 0.2\%$, which leads to the largest $|S_X|$.

Table 5.2.2 – Values of $\mathbb{X}$, s and $|S_X|$. $|S_X|$ provides a lower bound on the number of invalid solutions. An upper bound on $|\mathbb{S}|$ is given by $|\mathbb{X}| - |S_X|$, which in this case is very close to $|\mathbb{X}|$.

mesh	n	$\lvert\mathbb{X}\rvert$	$t = 0.2\%$	
			s	S_X
Mushroom	22800	10^{9902}	11378	10^{6863}
Onera	85567	10^{37161}	42698	10^{25758}
Wave	156316	10^{67887}	78003	10^{47055}
Linkrodsok1	174218	10^{75661}	86935	10^{52444}
Shock	1196252	10^{519530}	596980	10^{360110}

However, the third criterion in each weight distribution assigns a unitary weight to each vertex, by definition. A unitary weight distribution is a weight distribution for which s is the greatest possible, because s is the minimal number of vertices in a part for a solution. Therefore, for each mesh, s is the same for the 3 weight distributions.

The number of partitions that we can eliminate with this method is huge, but remains negligible compared with the total number of partitions. Therefore, we will use another method in order to estimate more precisely the number of solutions.

Estimation of $|\mathbb{S}|$ Using a Monte-Carlo Method

The Monte-Carlo method, which is explained in detail by Kalos and Whitlock [2009], consists in generating random partitions and counting the number of partitions that are indeed solutions. This is the same as measuring the surface of a lake by sending random shots in a surface S embracing the lake and counting the proportion p of shots that fall in the lake; the surface of the lake is then $p \times S$.

In order to generate a random partition, we assign a random part to each vertex. Using this method, a partition will have a probability of $(\frac{1}{2})^{n-1}$ to be found ($\frac{1}{2}$ for each vertex, divided by 2 because if the parts of all vertices are reversed, the partition is the same). It is a uniform distribution.

Numerical Application. Table 5.2.3 estimates the proportion of partitions that are solutions, $\frac{|\mathbb{S}|}{|\mathbb{X}|}$, by running the random partitioning algorithm 10000 times on each instance.

The proportion of solutions found using a random algorithm is quite astonishing. Indeed, for all the instances, when the imbalance tolerance is 5%, most of the time, all partitions are solutions. Therefore, with this tolerance, more runs of the random partitioning algorithm may be needed.

Table 5.2.3 – Estimation of the size of the solution space using a Monte-Carlo method: it is given by $|\mathbb{X}| \times \frac{|\mathbb{S}|}{|\mathbb{X}|}$.

Mesh	n	$\lvert\mathbb{X}\rvert$	**w**	$t = 5\%$ $\frac{\lvert\mathbb{S}\rvert}{\lvert\mathbb{X}\rvert}$	$t = 1\%$ $\frac{\lvert\mathbb{S}\rvert}{\lvert\mathbb{X}\rvert}$	$t = 0.2\%$ $\frac{\lvert\mathbb{S}\rvert}{\lvert\mathbb{X}\rvert}$
Mushroom	22800	10^{9902}	1	0.99	0.24	0.0032
			2	0.99	0.26	0.0040
			3	0.99	0.31	0.0051
Onera	85567	10^{37161}	1	1.00	0.96	0.12
			2	1.00	0.98	0.23
			3	0.99	0.48	0.025
Wave	156316	10^{67887}	1	1.00	0.84	0.13
			2	1.00	0.90	0.11
			3	1.00	0.71	0.098
Linkrodsok1	174218	10^{75661}	1	1.00	0.99	0.57
			2	1.00	1.00	0.61
			3	1.00	1.00	0.58
Shock	1196252	10^{519530}	1	1.00	0.99	0.40
			2	1.00	1.00	0.62
			3	1.00	0.99	0.30

When the imbalance tolerance is 1%, the proportion of solution is still very high. And when the imbalance tolerance is 0.2%, the proportion remains non-negligible, especially for the big instances. This shows that the solution space, even for a multi-criteria instance, remains huge.

Nevertheless, different algorithms will have different ways to explore this space. For example, the multilevel algorithm, when coarsening, will reduce the search space and, probably, the solution space. Refinement algorithms usually do not step out of the solution space, but depending on the local refinement that they perform (*e.g.* swaps, move of a single vertex, *etc.*), they may not be able to reach the same solutions. Some relax the imbalance tolerance in order for the refinement algorithms to consider partitions that are nearly solutions.

5.3 Study of the Connection of the Solution Space

The purpose of this section is to examine if an algorithm may reach an optimal solution. In particular, we will focus on local optimizations algorithms, that perform at each step a few moves (a move corresponds to switching the part of a vertex). Usually, a local optimization algorithm cannot pass from a solution to a partition which is not a solution, and the allowed moves define a

neighborhood relationship between solutions.

This section focuses on the connection of the solution space, which is, given two solutions, if there exists a sequence of neighboring solutions that links the two solutions. In particular, we will show in Section 5.3.1 that the local optimization algorithms `FM` and `KL`, that were defined in Section 4.4, can have different sets of reachable solutions. Therefore, whereas `FM` is sometimes presented as a variation of `KL`, these algorithms are in fact intrinsically different.

Then, we will state a theorem that guarantees, in the mono-criterion case, the connection of the solution space for local optimization algorithms that move one vertex at a time, such as `FM`. The condition for connection is a bound on the vertex weights, which should not exceed $\frac{t}{2}$, where t is the imbalance tolerance. We were not able to generalize this result to the multi-criteria case, but it gives a hint on the influence of the vertex weights on the connection of the solution space, which will be taken into account in Chapter 6, when considering coarsening algorithms.

Remark

Section 2.6 introduced the notion of fitness landscape, which is in our case the combination of the search space (set of all k-partitions of the input mesh) and of the neighborhood relationship defined by an algorithm. The fitness landscape can be seen as an oriented graph, whose vertices are the partitions, and whose edges link partitions that are neighbors according to the considered algorithm. For example, with `FM`, two partitions Π and Π' are neighbors if, between Π and Π', there exists a unique cell that changed of part. If the solution space is connected, it means that, given any initial solution, the algorithm can find an optimal solution. If it is not connected, however, it means that we may have to relax the tolerance, or to try with different initial partitions.

The graph of the solution space can be seen as a landscape with peaks and valleys defined by the communication cost of a partition. The algorithm aims at exploring this landscape, trying to find a valley among those of lowest altitude, which will contain a partition of minimal communication cost. A final notion that we tried to investigate was the "ruggedness" of the solution space, which refers to the density of the valleys and the slope of the mountains. The ruggedness may give some information on the best suited algorithm to solve an instance. For example, in a smooth landscape, an algorithm always following the steepest descending slope might discover an optimal solution quickly. However, a steepest-descent algorithm is likely to be stuck in a local minimum when the landscape is rugged, for which an algorithm descending little by little or allowing hill-climbing can be more suited.

One of the main difficulties was how to define the ruggedness for a discrete space. A second challenge was how to sample the solution space in order to measure it.

5.3.1 Comparison of the Solution Space of KL and FM

This section considers the solution space of two famous local optimization algorithms, FM and KL, which were introduced in Section 4.4. Basically, FM passes from one partition to another by changing one vertex of part at a time, while KL exchanges the parts of two vertices. Therefore, these two algorithms define the following neighborhood structure between partitions.

Definition 31 (Neighborhood Structure for FM and KL)
We denote by $\Pi(m)$ *the part to which* $m \in M$ *belongs in partition* Π.
For FM*, we say that two partitions* Π *and* Π' *are neighbors if:*

$$|\{m \in M, \Pi'(m) \neq \Pi(m)\}| = 1 \ .$$

This means that there is one unique cell in M *that changed of part between* Π *and* Π'.

For KL*, we say that* Π *and* Π' *are neighbors if*

$$\begin{cases} \{m \in M, \Pi'(m) \neq \Pi(m)\} = \{m_{\rightarrow}, m_{\leftarrow}\} \ ; \\ \Pi'(m_{\leftarrow}) = \Pi(m_{\rightarrow}) \ ; \\ \Pi'(m_{\rightarrow}) = \Pi(m_{\leftarrow}) \ . \end{cases}$$

This means that there are two cells in M *that changed of part between* Π *and* Π'*, and that these two cells have switched their respective parts.*

KL exchanges at each step two cells of part, which means that the number of cells in some part is always the same. Therefore, from a starting partition $\Pi^{ini} = \{\Pi_1^{ini}, ..., \Pi_k^{ini}\}$, KL can only reach another partition $\Pi' = \{\Pi'_1, ..., \Pi'_k\}$ for which $|\Pi_1^{ini}| = |\Pi'_1|, ..., |\Pi_k^{ini}| = |\Pi'_k|$. This is not the case for FM, which moves a single vertex at a time. Therefore, if the cardinal of the parts for two solutions differ, the solution space for KL is not connected, while that of FM might be.

However, in some cases, the converse can be true. Consider for example two cells m_1 and m_2 of weight greater than $t \cdot \Sigma$. It means that, from a solution, moving m_1 or m_2 would lead to a partition which is not a solution. Therefore, if there is a solution $S_=$ in which m_1 and m_2 are in the same part and another $S_{\neq}$ in which m_1 and m_2 are in different parts, and if $\Pi^{ini} = S_=$, then there is no mean for FM to get to $S_{\neq}$ without unbalancing the partition (this statement will be proved in Section 7.3.2). However, it might be possible for KL, which may swap m_1 and m_2. Therefore, in this case, the solution space for FM might not be connected, while that of KL might be.

Thus, moving one cell at a time and performing exchanges induces an intrinsic difference in the connection of the solution space. This analysis shows that there is actually a major difference between KL and FM. In the next section, we will focus on FM-like optimization algorithms, and will formulate a bound on the vertex weights that guarantees the connection of the solution space in the mono-criterion bipartitioning case.

5.3.2 Bound on the Vertex Weights Ensuring the Connection of the Solution Space for FM-like Local Optimization Algorithms

This section considers the mono-criterion bipartitioning case (using the notations of Table ii on page 9, that is, $\gamma = 1$ and $k = 2$). Therefore, given a mesh M, we define $W = \{W(m)[1], m \in M\}$ (W is a multiset, as in Definition 3 on page 6). Bipartitioning W is equivalent to bipartitioning M.

Theorem 2 (Connection of the Solution Space of the Mono-criterion Bipartitioning Problem)

We consider the mono-criterion case and bipartitioning a multiset of non-negative numbers W with an imbalance tolerance $t \in [0, 1]$.

$$\max_{w \in W} \frac{w}{\Sigma} \leq \frac{t}{2} \implies \textit{the solution space is connected.}$$

Proof

Assume that $\max_{w \in W} \frac{w}{\Sigma} \leq \frac{t}{2}$ (1). By Theorem 1, the solution space is not empty.

Let Π and Π' be two solutions, which means that $imb(\Pi) \leq t$ and $imb(\Pi') \leq t$. We will build a sequence of neighboring solutions $(\Pi^1, ..., \Pi^f)$ such that $\Pi^1 = \Pi$ and $\Pi^f = \Pi'$. Π^i and Π^{i+1} are neighbors means that only one number changes of part between Π^i and Π^{i+1} (we say that only one number *moves*).

For that, we define $\mathcal{M}$ as the set of all numbers that have to move to transform Π into Π'. Noting $\Pi(w)$ the part to which w belongs in Π and $\Pi'(w)$ the part to which w belongs in Π', we have:

$$\mathcal{M} = \{w \in W, \Pi(w) \neq \Pi'(w)\} \ .$$

Our algorithm operates in two main steps:

1. as long as there is a number in $\mathcal{M}$ which also belongs to the heaviest part, move it to the lightest part and remove it from $\mathcal{M}$ (if both parts have the same weight, any part can be considered as the heaviest);
2. then, when there are no more numbers in $\mathcal{M}$ which also belong to the heaviest part, and as long as $\mathcal{M}$ is not empty, successively move the remaining numbers in $\mathcal{M}$ to the heaviest part (as we move a number from the lightest part to the heaviest part, the lightest part is still the lightest one after the move).

The algorithm stops when $\mathcal{M}$ is empty.

1. Moving from the Heaviest Part

Assume that partition Π^i is a solution (so $imb(\Pi^i) \leq t$ (2)), and that there exists $w \in \mathcal{M}$ that belongs to the heaviest part. We will first prove that Π^{i+1}, obtained by moving w to the lightest part, is also a solution.

Without loss of generality, we assume that $\Pi^i(w) = \Pi_1^i$ (which means that $\Pi^{i+1} = (\Pi_1^i \setminus \{w\}, \Pi_2^i \cup \{w\})$). Hence $imb(\Pi_1^i) \geq imb(\Pi_2^i)$ and $imb(\Pi_1^i) + imb(\Pi_2^i) = 0$, so $imb(\Pi_2^i) \leq 0$ (3).

The imbalances of the parts of Π^{i+1} are:

$$
\begin{aligned}
imb(\Pi_1^{i+1}) &= \frac{\Sigma_1^{i+1} - \frac{\Sigma}{2}}{\frac{\Sigma}{2}} = \frac{\Sigma_1^i - w - \frac{\Sigma}{2}}{\frac{\Sigma}{2}} = imb(\Pi_1^i) - \frac{w}{\frac{\Sigma}{2}} \\
&\leq t - \frac{2w}{\Sigma} && \text{by (2)} \\
&\leq t && \text{because } w \geq 0 \ ; \\
imb(\Pi_2^{i+1}) &= \frac{\Sigma_2^i + w - \frac{\Sigma}{2}}{\frac{\Sigma}{2}} = imb(\Pi_2^i) + \frac{2w}{\Sigma} \leq \frac{2w}{\Sigma} && \text{by (3)} \\
&\leq t && \text{by (1)} \ .
\end{aligned}
$$

Thus, $imb(\Pi^{i+1}) = \max(imb(\Pi_1^{i+1}), imb(\Pi_2^{i+1})) \leq t$, which means that Π^{i+1} is a solution.

We have proved that given a solution, moving any number from the heaviest part leads to another solution. Since the initial partition $\Pi^1 = \Pi$ is solution by hypothesis, this shows by induction that all the partitions built in the first step are solutions.

2. Moving from the Lightest Part

Without loss of generality, we assume as before that Π_1^i is the heaviest part. For the second step of the algorithm, we have by definition $\mathcal{M} \cap \Pi_1^i = \varnothing$. Besides, we will consider that $\mathcal{M} = \{w_1, ..., w_b\}$, and we define $\Pi^{i+j} = \left(\Pi_1^i \cup \{w_1, ..., w_j\}, \Pi_2^i \setminus \{w_1, ..., w_j\}\right)$.

We will prove by induction that $\forall j \in [0, b]$, Π^{i+j} is a solution and Π_1^{i+j} is still the heaviest part.

The condition is valid for $j = 1$ by definition of Π^i.

Then, given $j \in [1, b-1]$, we assume that the condition is valid for j and we will show that it still holds for $j+1$.

First, we have $\Sigma_1^{i+j+1} = \Sigma_1^{i+j} + w_{j+1} \geq \Sigma_1^{i+j}$, so the heaviest part is still the heaviest one after moving w_{j+1}.

Then,

$$\begin{aligned} imb(\Pi_1^{i+j+1}) &= \frac{\Sigma_1^{i+j+1} - \frac{\Sigma}{2}}{\frac{\Sigma}{2}} = \frac{\Sigma_1^i + \sum_{a=1}^{j+1} w_a - \frac{\Sigma}{2}}{\frac{\Sigma}{2}} \\ &\leq \frac{\Sigma_1^i + \sum_{a=1}^{b} w_a - \frac{\Sigma}{2}}{\frac{\Sigma}{2}} = imb(\Pi') \\ &\leq t \qquad \text{by hypothesis,} \end{aligned}$$

which proves that Π^{i+j+1} is a solution.

By induction, we have proved that the partitions $\Pi^{i+1}, ..., \Pi^{i+b}$ are solutions.

Conclusion

By construction, when $\mathcal{M}$ is empty, we have $\Pi^{i+b} = \Pi^f$. By definition of the Π^is, Π^i and Π^{i+1} are neighbors, and we have proved that they are all solutions. This shows that from any solution Π, there is a sequence of neighboring partitions leading to any solution Π', such that these partitions are all solutions.

Therefore, if the normalized weight of all vertices is smaller than half the imbalance tolerance, the solution space for the mono-criterion mesh partitioning problem is connected.

□

Unfortunately, we were not able to generilize this theorem to the multi-criteria case. The issue is that the notion of heaviest part does not extend to the multi-criteria case, and that moving a vertex can decrease the weight of a part for one criterion but increase it for another.

Nevertheless, this result seems to indicate that when the normalized weights are large, the solution space is less likely to be connected. This statement is likely to hold for multi-criteria instances, even if other factors intervene, such as the entanglement of the weight distributions (as discussed in Section 5.1).

Therefore, this result will influence the design of the coarsening phase, which is the topic of the next chapter. Besides, it will also justify the design of the initial partitioning algorithms that will be introduced in Chapter 7.

Conclusion

In this chapter, we have first discussed what makes a multi-criteria instance more difficult to partition than a mono-criterion instance. We have explained that the entanglement of the weight distributions (that is, whether the heavy

vertices are the same for all criteria) is a key issue.

Then, we have studied the size of the solution space using various approaches. First, we have stated, for the mono-criterion case, a bound on the vertex weights that guarantees the existence of a solution. This result was not generalized to multi-criteria, but it strengthens the assumption that, if the vertex weights are small, there will be more solutions. The same bound appeared to guarantee the connection of the solution space in the bipartitioning mono-criterion case, for `FM`-like algorithms. Therefore, we believe that having light vertices in general means that finding balanced partition is easier.

In the next chapters, we will keep this assumption in mind while analyzing partitioning algorithms. Then, in Chapter 10, we will compare the algorithms that make use of this assumption with the ones that do not consider it.

Chapter 6

Analysis of New Coarsening Schemes

Contents

Chapter 4 has exposed existing algorithms to partition a mesh. Among them, the multilevel framework is well spread among partitioning tools. Indeed, as reported in a survey by Buluç *et al.* [2015], it has been applied successively on various meshes. Nevertheless, many variations exist among its implementations. This chapter focuses on variations of the coarsening phase of the multilevel algorithm.

The coarsening phase has conventional objectives which will be detailed in Section 6.1. However, partitioning tools use different ways to fulfill these objectives. In this chapter, we will review existing coarsening schemes and link them with the implementations of the partitioning tools Scotch, MeTiS and PaToH when possible. In particular, Section 6.2 will focus on ordering the vertices before computing a matching, and Section 6.3 will concentrate on restrictions on the coarsened vertex weights.

Remarks

We chose to focus on MeTiS and PaToH because they are multi-criteria partitioning tools. Zoltan is another partitioning tool that also handles multi-criteria partitioning, yet relies only on geometrical algorithms, that are usually not suited for our meshes. Finally, Scotch is for now a mono-criterion partitioning tool, for which we wish to implement multi-criteria support.

Moreover, we have noticed that MeTiS-5.1.0 and Scotch-6.0.4 partitioning tools do not always behave as described by Karypis and Kumar [1998a] and Karypis and Kumar [1998b] for MeTiS, and by Pellegrini [2008] for Scotch.

The algorithms that we will define were deduced from a careful study of the source codes of MeTiS and Scotch, and we will provide references directly from the source code of these tools.

Unfortunately, we were not able to perform the same analysis for PaToH, whose source code is not available. Therefore, information about PaToH algorithms is based on the user manual of Catalyurek and Aykanat [2011].

Therefore, this chapter will not introduce any new coarsening scheme. Its aim is to define and analyze the objectives and consequences of coarsening schemes on the coarsened graph. Whenever possible, we will clarify the coarsening algorithms implemented in partitioning tools. The various schemes will be compared in Chapter 10, because their performance also depends on the initial partitioning algorithm and on the refinement algorithm used. These two issues will be addressed respectively in Chapters 7 and 8, while Chapter 9 will define the set of instances used for the comparison.

6.1 Conventional Goals of the Coarsening Scheme

This section aims at formulating the objectives of the coarsening phase and point out its possible variations. The coarsening phase is one of the three phases in the multilevel Algorithm 26 on page 106. In this phase, a function named `Coarsen` is called as long as the number of vertices in the graph does not become smaller than a threshold. This threshold, which we called n_{coarse}, is a parameter of the coarsening phase. To our knowledge, no study has ever been carried out on the setting of this value, but for Scotch, MeTiS and PaToH, the values are close.

Implementation – MeTiS-5.1.0

MeTiS defines $n_{coarse} = 100$ in the multi-criteria case (and $n_{coarse} = 20$ in the mono-criterion case).

`Source:` variable `ctrl->CoarsenTo` set in function `SetupCtrl` of file `options.c`.

Implementation – Scotch-6.0.4

Scotch defines $n_{coarse} = 120$.

`Source:` When displaying the default strategy (option `-vs`), it is the parameter `vert` as described in the user manual of Pellegrini [2008].

Implementation – PaToH-3.2

PaToH defines n_{coarse}'s value to lie between 10 and max_{int}, but does not indicate how it is actually computed. For the instances that will be presented in Chapter 9, we observed that n_{coarse} ranges from 48 to 89, which is close

to the values of MeTiS and Scotch.

In this document, we will only consider performing the coarsening through the computation of matchings, which were described in Definition 27: a matching is a function $\mathcal{M} : V \to V^c \subset V$ such that $\forall v^c \in V^c$, there is at least one and at most two vertices in V whose image by $\mathcal{M}$ is v^c. In other words, a matching pairs some vertices together. Matchings are usually opposed to aggregations, which can merge an arbitrary number of vertices at one level. Last but not least, we will only consider matching a vertex with one of its neighbors.

Implementation – MeTiS-5.1.0, Scotch-6.0.4, PaToH-3.2

Scotch and MeTiS both perform successive matchings to coarsen the graph, while PaToH relies on aggregations (though it also implements a number of matching-based coarsening schemes). In fact, at each level of PaToH's coarsening scheme, each vertex is fused with at least one of its neighbors.

Conventional objectives. As asserted by Buluç *et al.* [2015], the coarsening phase aims at fulfilling several conventional objectives.

- The first one is to try to minimize the weights of the edges of the coarsened graphs, so that the possible partitions of the coarsened graphs exhibit small communication cost. In other words, this mechanism aims at removing partitions of high communication cost from the search space.
- Another conventional requirement is that the weights of the vertices in the coarsened graphs should be uniform, so that it becomes easier to find a solution. This assertion is supported by Theorem 1, formulated in the previous chapter, which guarantees, in the mono-criterion case, the non-emptiness and the connection of the solution space when the vertex weights are bounded.
- An early objective was to improve the behavior of the `FM` algorithm used in the refinement phase. Indeed, when partitioning VLSI circuits, `FM` was found to suffer from large number of ties on the move gains, because of the small degrees of the vertices. A solution was to create coarsened vertices with larger degrees.
- Finally, one commonly tries to avoid letting vertices unmatched for several levels. Indeed, especially for meshes whose topology is very regular, regularly pairing vertices may help the coarsened graph maintain a topology similar to that of the original graph.

Actually, the coarsening phase may be seen as a maximum weight matching problem: the objective is to maximize the sum of the weights of the edges between matched vertices. As indicated by Buluç *et al.* [2015], this problem can be solved in polynomial time, but optimal algorithms are too slow in practice. Therefore, many works proposed and compared matching schemes. For

example, Holtgrewe *et al.* [2009] compares three matching schemes (including one approximating that of MeTiS) in the context of parallel graph partitioning.

Algorithm 27 defines a quite generic version of the coarsening algorithm (though it is at least restricted to matchings computed using neighborhoods). The `Coarsen` function fills the *matching* list so that the ith vertex in mesh M will be matched with the $matching[i]$th vertex. $matching[i] = 0$ means that the ith vertex is unmatched. A vertex can only be matched with at most one of its neighbors.

Despite being restricted, this version of the coarsening phase allows some degrees of freedom. In particular, the three aforementioned functions may differ between partitioning tools. We will now describe the coarsening algorithm and point out the possible variations of it.

Algorithm 27 Coarsening

```
1:  function Coarsen(M, W, W_com)
2:      n ← length(M)
3:      matching ← [0 .. 0]
4:      ord_vtxs ← Order(M)
5:      for i ← 1, n do
6:          v ← M[ord_vtxs[i]].index()
7:          if matching[v] = 0 then        # Try to match v with one of its neighbors
8:              ord_ngbrs ← OrderNgbrs(v, N(v), W, W_com)   # N(v): neighbors of v
9:              j ← 1
10:             while matching[v] = 0 and j ≤ length(N(v)) do
11:                 u ← N(v)[ord_ngbrs[j]].index()
12:                 if matching[u] = 0 and not Restriction(v, u, W) then
13:                     matching[v] = u                  # Match v with u
14:                     matching[u] = v
15:                 end if
16:                 j ← j + 1
17:             end while
18:             if matching[v] = 0 then   # All neighbors of v were already matched
19:                 matching[v] = v       # v will remain unmatched at this level
20:             end if
21:         end if
22:     end for
23:     return matching
24: end function
```

Prior to choosing which vertices are matched together, an ordering on the vertices is first computed using the `Order` function. This function differs across

partitioning tools, and will be the object of Section 6.2.

Considering the ith vertex v according to the defined order, if v is unmatched, we will try to match it with one of its yet unmatched neighbors. To do so, the neighbors are first ordered using the `OrderNgbrs` function, which slightly differs between partitioning tools. Nevertheless, the basic idea, which is to use the Heavy-Edge-Matching scheme (`HEM`) sketched in Section 4.5, remains the same. `HEM` tries to match in priority a vertex with some neighbor along the edge or hyperedge of greatest communication weight. The rational is to get rid, in the coarsened graph, of the heaviest edges.

Algorithm 28 Heavy-Edge-Matching, a Very Common Implementation of the `OrderNgbrs` Function

1: **function** `OrderNgbrsHEM`(v, $\mathcal{N}_v$, W, W_{com})
2: # If graph model
3: **return** `sortDescend`($\mathcal{N}(v)$, `key` : $u \mapsto W_{com}[(v,u)]$)
If mesh or hypergraph model
4: **return** `sortDescend`($\mathcal{N}(v)$, `key` : $u \mapsto W_{com}[u]$)
5: **end function**

Nevertheless, the `OrderNgbrsHEM` function is ambiguous. Indeed, if one vertex has several edges with identical communication weight, several orders are possible. In order to compare matching schemes, in Chapter 10, we will use the default order on the vertices as a tie-breaker. This is also the policy of Scotch, but MeTiS uses a different one.

Implementation – MeTiS-5.1.0

MeTiS' `OrderNgbrsHEM` function implements a tie-breaking policy, which considers the computation weights. In multi-criteria, the preference goes to the neighbors which, after matching, would lead to the most "uniform" computation weights.

This uniformity is defined in MeTiS as $\sum_{c=1}^{\gamma} \left| \frac{W(v)[c]}{\Sigma_c} - \overline{W(v)} \right|$, where $\overline{W(v)} = \frac{1}{\gamma} \sum_{c=1}^{\gamma} \frac{W(v)[c]}{\Sigma_c}$ is the mean of the normalized weights of a vertex. For tie-breaking, it selects the vertex that minimizes this quantity. Note that this is still ambiguous, for example, when the neighbors of one vertex have the same communication and computation weights.

`Source:` function `BetterVBalance` called by the function `Match_SHEM`.

The global `Coarsen` function tries to match a vertex v with one of its neighbors. After the determination of the neighbors ordering, we can match v with the first of its unmatched neighbor, provided it is not forbidden matching. Indeed, as we will see in Section 6.3, partitioning tools define various `Restriction` functions to prevent the creation of vertices with unwanted properties, and in particular, aiming at producing uniform vertex weights. Thus, if

the considered neighbor u is unmatched and that the matching is not forbidden, v and u are matched together.

Conclusion. This section has formulated a quite generic version of the coarsening algorithm and has pointed out possible variations. Some of these variations (including those of existing partitioning tools) will be compared in Chapter 10, because their effects remain unclear.

In the next sections, we will describe and analyze several matching schemes. More precisely, the next section will deal with the `Order` function, while the one after will focus on the `Restriction` function.

6.2 Analysis of Ordering Schemes when Computing a Matching

The `HEM` policy aims at matching a vertex with its neighbor along the heaviest edge, which is not always possible, for example when two different vertices have their heaviest edge ending on the same vertex. Thus, different orderings of the vertices can lead to different matchings. In this section, we describe several ordering strategies, and analyze their respective objectives.

Remark

A basic observation is that vertices that are considered first have a greater chance to be matched than vertices considered last, whose neighbors are more likely to be already matched. Moreover, vertices that are considered first have a greater chance to be matched along their heaviest edge, whereas at the end, the remaining unmatched vertices can only be matched with their unmatched neighbors, not really taking into account the edge weights.

Basic Ordering Strategies

We propose in Algorithm 29 two ordering strategies. The first one, named `OrderFirst`, will act in contradiction with the conventional objectives, because it tends to always match the same vertices. Indeed, the vertices of small index will have a greater probability to be matched, and in this scheme, the vertices of small index remain of small index at each level. If its performance is comparable to that of other ordering strategies, it will mean that the conventional objectives are unfounded.

The second ordering strategy in Algorithm 29 is `OrderRandom`, which returns a random order of the cells. It aims at preventing vertices from remaining unmatched for several levels, because the vertices that have a bigger probability to be matched change at each level. However, there is no guarantee to produce uniform vertex weights.

Implementation – PaToH-3.2

PaToH uses the `OrderRandom` strategy to order the vertices when matching.

Algorithm 29 Basic Ordering Strategies

```
1: function OrderFirst(M)
2:     n ← length(M)
3:     return [1 .. n]
4: end function

5: function OrderRandom(M)
6:     n ← length(M)
7:     return shuffle([1 .. n])
8: end function
```

Ordering Strategies Based on Degrees and Edge Weights

Karypis and Kumar [1995] studied the average degrees of the vertices of the coarsened graphs. They reported that the average degree increases in the first levels, and decreases in the last levels. Indeed, for meshes, at the beginning of the multilevel algorithm, the vertices have roughly the same number of neighbors. However, when two vertices u and v are matched together to form vertex $u+v$, $deg(u+v)$ is given by:

$$deg(u+v) = deg(u) + deg(v) - |\mathcal{N}(u) \cap \mathcal{N}(v)| - 2 \ .$$

Therefore, the coarsened vertices that result from the matching of two finer vertices will in general have a higher degree than coarsened vertices that remained unmatched in the previous levels. A means to prevent vertices from remaining unmatched for several levels is to order first the vertices of lower degree.

However, using the `HEM` policy, the vertices that have a heavy edge have a higher probability to be matched than the others. Therefore, if two vertices have several heavy edges, matching them together will mean that most of these heavy edges will remain in the coarsened graph (maybe as even heavier edges), whereas matching such vertices with vertices with less heavy edges may remove more heavy edges.

This may be avoided using another definition of the degree of a vertex, which is the sum of the weights of its edges. We will call this the "weighted degree" and denote it by $wdeg(v) = \sum_{u \in \mathcal{N}_v} W_{com}((u,v))$. The weighted degree is used in particular in the Laplacian matrix of the graph (used for spectral

partitioning, which was defined in Section 4.3.1). When vertices u and v are matched together to form vertex $u+v$, the weighted degree of $u+v$ is:

$$wdeg(u+v) = wdeg(u) + wdeg(v) - 2W_{com}((u,v)) \ .$$

Algorithm 30 defines an ordering strategy, named `OrderDegrees`, based on vertex degrees or weighted degrees. It orders the vertices by increasing degrees or weighted degrees.

Algorithm 30 Ordering Strategies Based on Degrees and Edge Weights

```
1: function deg(v, W_com)
2:     # Using the regular degree deg
3:         return length(N(v))
       # Using the weighted degree wdeg (graph model)
4:         return sum_{u∈N(v)} W_com((u,v))
       # Using the weighted degree wdeg (mesh or hypergraph model)
5:         return W_com(v)
6: end function

7: function OrderDegrees(M)
8:     n ← length(M)
9:     return argsortAscend([deg(M[1]) ... deg(M[n])])
10: end function
```

We have tested the `OrderDegrees` using the degrees and the weighted degrees, and the results were similar for many instances, and inconclusive for others. In Chapter 10, we will report the results obtained using the regular degrees. However, a study to understand the difference between using the degrees and the weighted degrees may be worthwhile.

Implementation – MeTiS-5.1.0

MeTiS first calls the `OrderRandom` and then a pseudo "`OrderDegree`" method (using the degree of the vertices). Indeed, it sorts only the vertices whose degree is smaller than the average degree.

`Source:` functions `irandArrayPermute` and `BucketSortKeysInc(_, _, avgdegree, degrees, _, _)` called by function `Match_SHEM`.

Remark

In order to avoid sorting all the vertices by degree, we have also implemented another version of the `OrderDegree` function. Instead of sorting all the vertices, it divides them into b buckets and sorts each bucket independently. Then, the order is the first elements of each bucket, followed by the second elements of each bucket and so on. This function can be parallelized more easily than the `OrderDegree` function, because each process can sort

one bucket independently from the others. This aims at simulating the concurrent execution of b independent matching tasks, each of them handling a subset of the vertex set.

We tested this function (using the *wdeg*) for $b \in \{2, 8, 16, 32\}$ buckets, and observed that the results did not really change with the number of buckets, though this should be investigated further. However, in this document, we will report the results for the basic `OrderDegree` algorithm (using the *wdeg* criterion).

Ordering Strategies Based on Vertex Weights

Finally, Algorithm 31 defines a strategy based on vertex weights. Indeed, during the coarsening, the weights of the vertices are increasing, because the weight of a coarsened vertex is the sum of the weights of the original vertices. So, the left-behind vertices should have smaller weights. As we have seen in Chapter 5, creating heavy vertices is likely to reduce the size and the connection of the solution space. Therefore, given a threshold $t^{priority} \in \mathbb{R}_+$, the `OrderVwgts` function orders first the vertices of weight smaller than $t^{priority} \times \frac{\Sigma_c}{n}$ for any criterion c ($\frac{\Sigma_c}{n}$ is the average weight of a vertex for criterion c).

Algorithm 31 Ordering Strategies Based on Vertex Weights

```
1:  function OrderVwgts(M, W)
    Require: t^priority ∈ ℝ+ weight threshold
2:      n ← length(M)
3:      l_inf, l_sup ← [ ], [ ]
4:      for i ← 1, n do
5:          if ∀c ∈ ⟦1, γ⟧, W(M[i])[c] < t^priority · Σ_c/n then
6:              l_inf.append(i)
7:          else
8:              l_sup.append(i)
9:          end if
10:     end for
11:     return l_inf ∪ l_sup
12: end function
```

Implementation – Scotch-6.0.4

Scotch uses the `OrderVwgts` function with $t^{priority} = 0.25$, combined with a so-called "cache-friendly" permutation (which may be approximated to a localized `OrderRandom`).

`Source:` function `GRAPHMATCHSCANNAME` in file `graph_match_scan.c`.

Algorithm 31 takes into account the vertex weights, but does not explicitly forbid to create heavy vertices, which can reduce the solution space, as shown in Chapter 5. The next section proposes some restrictions that forbid to match vertices when the formed vertex would be too heavy.

6.3 Taking Balance into Account with Weight Restrictions

In Chapter 5, we have shown that when the normalized weights are small, the size and connection of the solution space are likely to increase. Therefore, many partitioning tools introduce a `Restriction` function in order to prevent the creation of such vertices. A version of this function is defined in Algorithm 32.

The goal of the restriction function is to prevent the matching of vertices if the merged vertex would be too heavy. In order to define what "too heavy" means, the `Restriction` function requires a parameter called $W^{restrict} \in (\mathbb{R}_+)^\gamma$ in Algorithm 32: The `Restriction` function forbids to merge two vertices if the resulting vertex weight for criterion c were above $W^{restrict}[c]$.

Algorithm 32 Restriction Algorithm

```
1: function Restriction(v, u, W)
   Require: W^restrict ∈ (ℝ+)^γ
2:     for c ← 1, γ do                          # γ is the number of criteria
3:         if W(v)[c] + W(u)[c] > W^restrict[c] then
4:             return False
5:         end if
6:     end for
7:     return True
8: end function
```

Implementation – MeTiS-5.1.0

MeTiS policy for multi-criteria graph partitioning considers the total weight of a criterion for the current level, and defines $W^{restrict}[c] = 0.015 \cdot \Sigma_c$ (or $W^{restrict} = 0.075 \cdot \Sigma$ in the mono-criterion case), so it will not match vertices if the resulting vertex weight for one criterion is heavier than 1.5% of the total weight for this criterion.

`Source:` See the function `CoarsenGraph` in file `coarsen.c`.

Implementation – Scotch-6.0.4

Scotch considers the average vertex weight of the coarsened graph, which is defined as $\frac{\Sigma_c}{n_{level-1}}$, where $n_{level-1}$ is the maximum number of vertices at the coarser level, defined as $n_{level-1} = n_{level} \cdot rat$, where n_{level} is the number of vertices at the current level, and rat the coarsening ratio, set to 0.8 by

default. Then, Scotch forbids to match vertices if the resulting vertex weight is heavier 4 times the average weight, so $W^{restrict}[c] = 4 \cdot \frac{\Sigma_c}{n_{level} \cdot 0.8}$.

`Source:` Variable `coarvelomax` in function `GRAPHMATCHSCANNAME` of file `graph_match_scan.c`.

Conclusion

In this chapter, we have devised a quite general version of the matching algorithm, and have linked it with the famous Heavy-Edge-Matching coarsening scheme. Nevertheless, the behavior of `HEM` is not fully explicitly defined. Therefore, various implementations are possible and may lead to very different coarsened graphs.

We have pointed out the parts that were ambiguous and have linked them to the conventional objectives of the coarsening phase. Whenever possible, we have clearly defined the coarsening strategies of the partitioning tools Scotch, MeTiS and PaToH. This illustrated the variations of implementations for this phase.

The consequences of these variations on the returned solutions will be examined in Chapter 10. Indeed, the performance of a coarsening strategy also depends on the initial partitioning algorithm and on the refinement phase, which are the topics of the next two chapters.

Chapter 7

Definition of Two Vector-of-Numbers Partitioning Algorithms for the Initial Partitioning Phase

Contents

Section 4.5 has introduced the famous multilevel algorithm, used to solve the multi-criteria mesh partitioning Problem 2 on page 34. The first phase of this algorithm, the coarsening phase, has been studied in Chapter 6. The current chapter focuses on the initial partitioning phase, whose aim is to find a balanced partition of the coarsest graph. "Balanced" means that among all criteria, the maximum imbalance is smaller than the input tolerance, as stated in Definition 11. Such a balanced partition is called a solution.

Most of the existing partitioning algorithms focus on returning a partition of small communication cost, and we have observed that partitioning tools return a large proportion of partitions that are not solutions. Therefore, we have decided, for the initial partitioning phase, to focus on returning a solution. So, for this phase, we will reduce the mesh partitioning problem to a vector-of-numbers partitioning problem (as defined in Problem 6 on page 43).

This amounts to dropping the objective function, which will be considered later, in the refinement phase. Providing a solution to a refinement algorithm allows it to focus exclusively on reducing the communication cost, without having to reduce also the imbalance. Besides, if the solution space is connected, which was studied in Chapter 5, once a solution is found, the refinement function can reach the partition that minimizes the objective function, passing only through solutions. This allows one to guarantee that the returned partition will be a solution to our problem. The only condition is then to find a balanced initial partition.

Dropping the objective function may simplify the problem, but it remains a difficult one, as proved in Chapter 3, which presented various algorithms addressing the mono-criterion version of the vector-of-numbers partitioning problem, the famous number partitioning problem. Besides, despite having been well-studied, as shown in Section 3.4.3, only a few algorithms designed for number partitioning can be used for vector-of-numbers partitioning. Actually, the only algorithms that may be directly extended to vector-of-numbers partitioning are the dynamic programming algorithm of Section 3.2.3 and the stochastic algorithms introduced in Section 3.2.7.

The dynamic programming algorithm is only applicable to integer weights, and also may require huge amounts of memory, depending on the total sums of the weights Σ_c, so we chose to focus on the stochastic algorithms. Section 7.1 will define a hill-climbing vector-of-numbers partitioning algorithm, which moves a vertex from a part to another as long as it reduces the partition imbalance. Then, Section 7.2 will consider moving the vertex that reduces the most the partition imbalance. Finding such a vertex may require many computations. This is why we will implement in Section 7.3 (in the case of mono-criterion bipartitioning) and in Section 7.4 (in the case of multi-criterion bipartitioning) a method to speedup the search. Finally, Section 7.5 will discuss the extension of the method to k-partitioning, when $k > 2$.

Definitions and Useful Functions

Definition 32 introduces the gain in imbalance of a move, that we denote by *igain*, and that will be used throughout this chapter. Note that the *igain* is defined in the same way for mono- and multi-criteria partitioning.

Definition 32 (Gain in Imbalance of a Move)
Given a mesh M and a partition $\Pi = \{\Pi_1, \overset{k}{\ldots}, \Pi_k\}$ of M, let v be a cell or its corresponding vertex in M.

We assume that $v \in \Pi_p$, and consider moving it to $\Pi_q \neq \Pi_p$, which would lead to the partition $\Pi' = \{\Pi'_1, \overset{k}{\ldots}, \Pi'_k\}$ such that $\Pi'_p = \Pi_p \backslash \{v\}$ and $\Pi'_q = \Pi_q \cup \{v\}$ and $\forall r \notin \{p, q\}, \Pi'_r = \Pi_r$.

Then, the gain in imbalance for this move is defined as:

$$igain(v \to \Pi_q) = imb(\Pi) - imb(\Pi') \ .$$

We also define the gain in imbalance of a move for one criterion c as:

$$igain_c(v \to \Pi_q) = imb_c(\Pi) - imb_c(\Pi') \ .$$

Π_q is called the target part of v.

In the bipartitioning case, there is only one possible target part for a vertex, so we will simplify the notations: $igain(v)$ instead of $igain(v \to \Pi_q)$ and $igain_c(v)$ instead of $igain_c(v \to \Pi_q)$.

Then, we define the most imbalanced criterion when dealing with multi-criteria partitioning.

Definition 33 (Most Imbalanced Criterion, c_{max})
Given a partition Π, we call "most imbalanced criterion" and we denote by c_{max} any integer in $[\![1, \gamma]\!]$ such that $imb_{c_{max}}(\Pi) = imb(\Pi)$.

Note that there can be several possible values for c_{max}, when Π exhibits the same imbalance for several criteria. In this case, c_{max} is one of these values.

Algorithm 33 describes the `RangePart` function that will be used by many other algorithms in this chapter. This function returns a list containing, in its ith cell, the index of the ith heaviest part. In case of bipartitioning ($k = 2$), it returns in the first cell the index of the overweighted part, and in the second cell the index of the underweighted part.

Algorithm 33

```
1: function RangeParts(Π, k)
2:     return sortDescend([1 .. k], key : p ↦ imb(Π_p))
3: end function
```

Finally, in this chapter, the word "weight" will mean "normalized weight". For a cell $m \in M$, its normalized weight for criterion c is $w = \frac{W(m)[c]}{\Sigma_c}$.

7.1 Descent Vector-of-Numbers Partitioning Algorithm

This section proposes a new vector-of-numbers partitioning algorithm called `VNFirst`, defined in Algorithm 34. It is very similar to the descent Algorithm 11 on page 69. A notable difference is that, to achieve a more generalized algorithm, we chose to allow the user to give as an argument an initial partition. `VNFirst` will try to move vertices in order to reduce the imbalance, starting from this initial partition. As an example, we tested `VNFirst` by feeding it with a random partition, a partition that had all vertices in one part, or a partition balanced for one criterion.

Algorithm 34 Descent Vector-of-Numbers Partitioning

Require: Π: an initial partition of M (may be random)
1: **function** `VNFirst`(M, k, W, t, f, Π)
2: $n \leftarrow$ `length`(M)
3: $i, i_{last} \leftarrow 1, 1$ # i_{last}: index of last moved vertex
4: **repeat** # Iterate on the vertices in order
5: **for** $q \leftarrow 1, k,\ q \neq \Pi[i]$ **do**
6: **if** $igain(M[i] \rightarrow \Pi_q) > 0$ **then** # If this move decreases the imbalance
7: $\Pi[i] \leftarrow q$ # Perform the move
8: $i_{last} \leftarrow i$
9: **end if**
10: **end for**
11: $i \leftarrow (i \mod n) + 1$ # i starts back at 1 after reaching n
12: **until** $i = i_{last}$ # Until no more move decreases the imbalance
13: **return** Π
14: **end function**

To reduce the imbalance of the input partition, the `VNFirst` function considers the vertices in order and moves a vertex as long as it reduces the imbalance. It keeps track in the variable i_{last} of the index of the last vertex moved and will stop if no more move can reduce the imbalance.

The partition returned is not guaranteed to have an imbalance smaller than the input tolerance, and it is not possible to reduce its imbalance with only one move.

The next section considers performing at each step the move that reduces the most the imbalance.

7.2 Steepest Descent Vector-of-Numbers Partitioning Algorithm (Greedy Implementation)

This section proposes a new vector-of-numbers partitioning algorithm, defined in Algorithm 35, called `VNBest_greedy`. As `VNFirst`, it is also quite similar to the hill climbing Algorithm 11. `VNBest_greedy` differs from `VNFirst` because, instead of moving the first vertex found that reduces the imbalance, `VNBest_greedy` moves the vertex that reduces the imbalance the most. Note that moves that increase the imbalance are forbidden. This means that in the returned partition, no more move can reduce the imbalance.

Algorithm 35 Greedy Version of the Steepest Descent-like Vector-of-Numbers Partitioning Algorithm

Require: Π: an initial partition of M (may be random)
1: **function** `FindBestMove_greedy`(M, k, Π)
2: $n \leftarrow$ `length`(M)
3: $i_{best}, q_{best}, g_{best} \leftarrow$ `None`, `None`, `None`
4: **for** $i \leftarrow 1, n$ **do**
5: **for** $q \leftarrow 1, k$; $p \neq \Pi[i]$ **do**
6: $g_{i \to q} \leftarrow igain(M[i] \to \Pi_q)$
7: **if** $g_{best} =$ `None` or $g_{i \to q} > g_{best}$ **then**
8: $i_{best}, q_{best}, g_{best} \leftarrow i, q, g_{i \to q}$
9: **end if**
10: **end for**
11: **end for**
12: **return** i_{best}, q_{best}
13: **end function**

14: **function** `VNBest_greedy`(M, W, W_{com}, k, t, Π)
15: $i, q \leftarrow$ `FindBestMove_greedy`(M, k, Π)
16: **while** $igain(M[i] \to \Pi_q) > 0$ **do**
17: $\Pi[i] \leftarrow \Pi_q$
18: $i, q \leftarrow$ `FindBestMove_greedy`(M, k, Π)
19: **end while**
20: **return** Π
21: **end function**

Finding the move leading to the smallest imbalance implies, as implemented in Algorithm 35, a lot of computations, because at each step, it computes the gain for all possible moves. If n is the number of cells in M, then there are $n \cdot (k-1)$ possible moves, which implies computing $n \cdot (k-1) \cdot \gamma$ imbalances

at each step.

Is there a way to avoid computing the gain for all vertices at each step? Is there a way to find quickly the best move? The next section will investigate these questions, restricting to the mono-criterion case for now.

7.3 An Implementation of VNBest Avoiding Many Computations

In this section, we will introduce a novel implementation of the algorithm VNBest, which moves at each step the vertex whose *igain* is maximum as long as this *igain* is non-negative. This implementation is restricted to mono-criterion bipartition, but it will be adapted to multi-criteria bipartitioning in the next section.

First, Section 7.3.1 will formulate an expression of the *igain* for each vertex. Section 7.3.2 will briefly prove a corollary on the connection of the solution space for algorithms that perform moves, which was discussed in Section 5.3.1. Then, using the expression of the *igain*, Section 7.3.3 will explain that at some point, some vertices will not move until the end of the algorithm. Using these two results, Section 7.3.4 proposes a gain table structure that will allow to record the gains for each vertex, and in Sections 7.3.5 and 7.3.6 we will respectively describe how we can find quickly the move of greatest gain using this structure and how to update the gain table quickly.

7.3.1 Expression of the Gain of a Move

As stated in the following Proposition 3, the gain of a vertex may be computed quite easily.

Proposition 3 (Gain in Imbalance for Mono-criterion Bipartitioning)

Given a mesh M and a bipartition $\Pi = \{\Pi_1, \Pi_2\}$ of M, let v be a cell or its corresponding vertex in M.

We assume that $v \in \Pi_p$, and consider moving it to $\Pi_q \neq \Pi_p$, which would lead to the partition $\Pi' = \{\Pi'_p, \Pi'_q\}$, such that $\Pi'_p = \Pi_p \setminus \{v\}$ and $\Pi'_q = \Pi_q \cup \{v\}$

As specified at the beginning of this chapter, w is the normalized weight of v, which is $w = \frac{W(v)}{\Sigma}$.

Then, the gain of v is:

$$igain(v) = \begin{cases} -2w & \text{if } imb(\Pi_q) \geq 0 \ , \\ 2 \cdot imb(\Pi) - 2w & \text{if } imb(\Pi_q) \leq 0 \text{ and } w \geq \frac{imb(\Pi)}{2} \ ; \\ 2w & \text{if } imb(\Pi_q) \leq 0 \text{ and } w \leq \frac{imb(\Pi)}{2} \ . \end{cases}$$

Remark

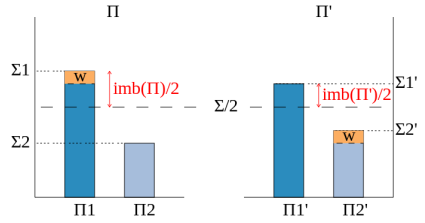
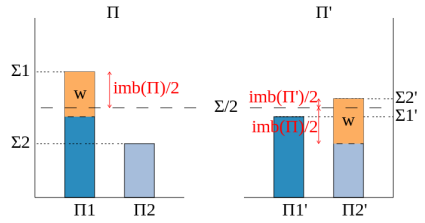
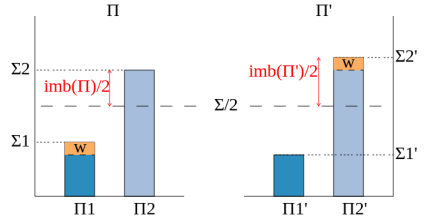
The *igain* function is continuous. Indeed, it is linear if $imb(\Pi_q) \geq 0$, and piecewise linear with no discontinuity at its endpoint $w = \frac{imb(\Pi)}{2}$ otherwise:

$$2 \cdot imb(\Pi) - 2w = 2 \cdot 2w - 2w = 2w \ .$$

Proof

Without loss of generality, we assume that the weights are normalized, so $\Sigma = 1$.

Informal proof. The table below illustrates the formal proof that follows. It is a visual demonstration of the *igain* value. Each line corresponds to a possible expression of *igain*, which is stated in the last column.

Drawing	Observation	$igain(v) = imb(\Pi) - imb(\Pi')$
Moving to the underweighted part when $w \leq \frac{imb(\Pi)}{2}$		
	$\frac{imb(\Pi')}{2} + w = \frac{imb(\Pi)}{2}$	$igain(v) = 2w$
Moving to the underweighted part when $w \geq \frac{imb(\Pi)}{2}$		
	$\frac{imb(\Pi)}{2} + \frac{imb(\Pi')}{2} = w$	$igain(v) = 2 \cdot imb(\Pi) - 2w$
Moving to the overweighted part		
	$\frac{imb(\Pi')}{2} = \frac{imb(\Pi)}{2} + w$	$igain(v) = -2w$

On every figure, Π is displayed on the left and Π' on the right. For each partition, the sums of their parts (Σ_p, Σ'_p, Σ_q and Σ'_q) are displayed as a

vertical bar, and the vertex moved (of weight w) is in orange.

The imbalances for each partition are displayed with red arrows. Indeed, using the fact that when bipartitioning, $imb(\Pi_p) = -imb(\Pi_q)$, we have:

$$imb(\Pi) = \max\left(imb(\Pi_p), imb(\Pi_q)\right) = |imb(\Pi_p)| = \left|\frac{\Sigma_p - \frac{\Sigma}{2}}{\frac{\Sigma}{2}}\right| = 2 \cdot \underbrace{\left|\Sigma_p - \frac{1}{2}\right|}_{\text{red arrow}}$$

The middle column of each line combines the red arrows to form an expression, from which one can deduce the expression of the *igain* in the last column.

Formal proof. Using the notations defined in Proposition 3,

$$\begin{aligned} igain(v) &= imb(\Pi) - imb(\Pi') \\ &= \max\left(imb(\Pi_p), imb(\Pi_q)\right) - \max\left(imb(\Pi'_p), imb(\Pi'_q)\right) \ . \end{aligned}$$

In the case of bipartitioning, $imb(\Pi_p) = -imb(\Pi_q)$, so:

$$\begin{aligned} igain(v) &= |imb(\Pi_q)| - |imb(\Pi'_q)| \\ &= |imb(\Pi_q)| - \left|\frac{\Sigma_q + W(v) - \frac{\Sigma}{2}}{\frac{\Sigma}{2}}\right| = |imb(\Pi_q)| - \left|\frac{\Sigma_q - \frac{\Sigma}{2}}{\frac{\Sigma}{2}} + \frac{W(v)}{\frac{\Sigma}{2}}\right| \\ &= |imb(\Pi_q)| - |imb(\Pi_q) + 2w| \ . \end{aligned}$$

If $imb(\Pi_q) \geq 0$ (the part in which we place v is already overweighted):

$$\begin{aligned} igain(v) &= imb(\Pi_q) - (imb(\Pi_q) + 2w) \\ &= -2w \ . \end{aligned}$$

Else, $imb(\Pi_q) \leq 0$, so in the case of bipartitioning, $imb(\Pi) = -imb(\Pi_q)$ and

$$\begin{aligned} igain(v) &= -imb(\Pi_q) - |imb(\Pi_q) + 2w| \\ &= \begin{cases} -imb(\Pi_q) + (imb(\Pi_q) + 2w) & \text{if } imb(\Pi_q) + 2w \leq 0 \ ; \\ -imb(\Pi_q) - (imb(\Pi_q) + 2w) & \text{otherwise} \end{cases} \\ &= \begin{cases} 2w & \text{if } w \leq \frac{-imb(\Pi_q)}{2} = \frac{imb(\Pi)}{2} \ ; \\ 2 \cdot imb(\Pi) - 2w & \text{otherwise.} \end{cases} \end{aligned}$$

Which concludes the proof.

□

Before going into further details on the implementation, the next section will formulate a corollary on the connection of the solution space when using a local optimization algorithm similar to VNBest.

7.3.2 A Corollary: a Necessary Condition for the Connection of the Solution Space

Corollary 1 (Connection of the Solution Space for Local Optimization Algorithms Moving One Vertex At A Time)

Given a local optimization algorithm that moves one vertex at a time, and an instance for which at least two vertices u and v have a weight heavier than $t \cdot \Sigma$.

If there is a solution in which u and v are in the same part and another solution in which u and v are in different parts, then the solution space for this algorithm is not connected.

Proof

Let Π be a partition in which u and v are in the same part. We assume that Π is a solution, so $imb(\Pi) \leq t$. We will prove that moving u or v leads to a partition which is not a solution, or in other words that its imbalance is higher than t.

Without loss of generality, we consider that u and v belong to Π_1 and will consider moving u. We define $w = \frac{W(u)}{\Sigma}$; by assumption $w > t$. Besides, $\Pi' = \{\Pi_1 \setminus \{u\}, \Pi_2 \cup \{u\}\}$.

Firstly, we have $imb(\Pi) \leq t < w$, so using Proposition 3, the gain of u is either $-2w$ or $2 \cdot imb(\Pi) - 2w$. Therefore, the imbalance of Π' is either:

$$\begin{aligned} imb(\Pi') &= imb(\Pi) + 2w > imb(\Pi) + 2t \\ &> 2t && \text{Because } imb(\Pi) \geq 0 \ , \end{aligned}$$

or:

$$\begin{aligned} imb(\Pi') &= imb(\Pi) - (2 \cdot imb(\Pi) - 2w) \\ &= 2w - imb(\Pi) > 2t - imb(\Pi) \\ &> t && \text{Because } imb(\Pi) \leq t \ . \end{aligned}$$

We have shown that the imbalance of Π' is higher than t, so Π' is not a solution.

A sequence of neighboring solutions that would lead from a solution in which u and v belong to the same part to a solution in which u and v are not in the same part would need to, from one solution in which u and v, move either u or v, which would thus lead to a partition which not a solution. Therefore, such a sequence does not exist, which proves that the solution space in this case is not connected.

□

This theorem can be used to determine if the solution space of an instance is connected, when there are at least two vertices of weights heavier than $t\Sigma$.

Indeed, as we have seen in Section 5.2.2, it is easy to find solutions using a random partitioning algorithm. Nevertheless, as we can see in Table 5.2.1, which reports the value of $u/\Sigma = \max_{c\in[1,\gamma]} \max_{m\in M}(W(m)[c]/\Sigma_c)$, for the instances that we will use for our test, fulfilling the condition $u/\Sigma < t$ requires a value of t below 0.17%, while the tightest prescribed tolerance in our tests is $t = 0.2\%$.

7.3.3 Settled Vertices

Proposition 3 of the previous section leads to the following Proposition 4, which states that the vertices of weights heavier than $imb(\Pi)$ will have a negative gain until the end of the algorithm. As VNBest only performs moves of non-negative gain, we say that such vertices are *settled*.

Proposition 4 (Settled Vertices for the Mono-criterion Bipartitioning Case)
Given a mesh M and a bipartition Π of M, let v be a cell or its corresponding vertex in M, and let w be the (normalized) weight of v. We have:

$$w \geq imb(\Pi) \implies v \text{ will have a negative igain until the end of algorithm}$$

Proof

Let Π_q be the target part of v.

We assume that $w \geq imb(\Pi)$. Then, using Proposition 3, we know that the gain for v is:

$$igain(v) = \begin{cases} -2w & \text{if } imb(\Pi_q) \geq 0 \ ; \\ 2 \cdot imb(\Pi) - 2w & \text{if } imb(\Pi_q) \leq 0 \text{ and } w \geq \frac{imb(\Pi)}{2} \ ; \\ 2w & \text{if } imb(\Pi_q) \leq 0 \text{ and } w \leq \frac{imb(\Pi)}{2} \ . \end{cases}$$

Since $imb(\Pi) \geq 0$, we have $w \geq \frac{imb(\Pi)}{2}$, so

$$igain(v) = \begin{cases} -2w & \text{if } imb(\Pi_q) \geq 0 \ ; \\ 2 \cdot imb(\Pi) - 2w & \text{if } imb(\Pi_q) \leq 0 \ . \end{cases}$$

Therefore, in either case, the *igain* for v is negative, because $w \geq imb(\Pi)$ so $2 \cdot imb(\Pi) - 2w$ is negative, and $-2w$ is negative because $W : M \to \mathbb{R}_+$ so $w \geq 0$.

Moreover, since the imbalance is strictly decreasing by definition of VNBest, for the rest of the algorithm, w will stay heavier than $imb(\Pi)$, which means that v will have a negative *igain* until the end of the algorithm.

□

Proposition 4 shows that, depending on the current imbalance, some vertices will not move until the end of the algorithm. Therefore, there is no need to compute their *igain* until the end of the algorithm.

In the following section, we will use the results from Propositions 3 and 4 to propose a gain table structure that will allow us to find quickly the maximal *igain* at each step, and to avoid recomputing all gains after each move.

7.3.4 Gain Table Structure (Mono-criterion Case)

The *igain* of a move is the reduction in imbalance induced by this move, as in Definition 32. Our gain table records the *igain*s of the vertices in order to speedup the search and avoid unnecessary computations.

Ordering the Vertices to Order the *igain*s

Proposition 3 allows us to build Table 7.3.1, which displays data on the *igain* as a function of the vertex weights, for the vertices in the overweighted part:

- on the first row, the expression of the *igain* of a vertex depending on its weight and on the current imbalance;
- on the second row, the variations of the *igain* function. The *igain* is here considered as a function of the weight, for a given imbalance;
- on the last row, the sign of the *igain* function.

Table 7.3.1 – Variation array of the gain as a function of the weight, for a given imbalance and for the vertices in the overweighted part

$W(v)$	0	$\frac{imb(\Pi)}{2}$	$imb(\Pi)$	$+\infty$
$igain(v)$	$2w$		$2 \cdot imb(\Pi) - 2w$	
$igain(v)$	0 ↗	$imb(\Pi)$	↘ 0 ↘	$-\infty$
$igain(v)$	0	$+$	0	$-$

Gain Table Organization

We will make use of the simple variations of *igain* when the weight is increasing to build a gain table whose variations follow those of the *igain* function. Thus, our gain table will have n cells (where n is the number of cells in the mesh), and the ith cell will store the *igain* for the cell with the ith smallest weight in the mesh.

Table 7.3.2 – The gain table structure for a fictitious mesh of n cells, and for a fictitious partition

				$\frac{imb(\Pi)}{2}$ ↓			$imb(\Pi)$ ↓			
Π_p **is overweighted**										
weight	w_1	w_2 ...	w_{inflt}	$w_{inflt+1}$	...	w_{last}	w_{last+1}	...	w_n
part	Π_q	Π_p ...	Π_q	Π_p	...	Π_q	Π_q	...	Π_p
$igain(v \to \Pi_q)$		$2w$				$2 \cdot imb(\Pi) - 2w$			
	–	g_2 ...	–	$g_{inflt+1}$	...	–	x	...	x
							settled vertices		

Table 7.3.2 gives an example of the gain table structure. The vertex weights are indicated on the first row. The cells are ordered such that $w_1 \leq ... \leq w_n$. The part of each vertex in this example is indicated on the second row (with different colors for each part), the expression of *igain* is recalled on the third row (as computed in Proposition 3), and its value for each vertex in the overweighted part in the fourth row. A "–" in the fourth row indicates that the $igain(v \to \Pi_q)$ is not defined for vertex v because v already belongs to Π_q, and an "x" indicates that the gain does not need to be computed because the vertex is settled, as shown in Proposition 4.

Gain Table Initialization

Algorithm 36 details how the gain table is initialized.

First, the vertices are sorted by increasing weights, so that $W(v_1) \leq ... \leq W(v_n)$. Then, the ith cell in the table *gains* is initialized with the gain of v_i, but only if the weight of v_i is lighter than $imb(\Pi)$. Indeed, as seen in Proposition 4, vertices of weight heavier than $imb(\Pi)$ are settled, so we do not need to compute their gain.

Remark

Note that if v_i belongs to the underweighted part, the ith cell does not contain the gain of v_i. If $w_i \leq \frac{imb(\Pi)}{2}$, it contains the opposite of $igain(vi)$, and an incorrect value otherwise.

This is actually a little trick that will reduce the amount of computation when updating the gain after a move (which will be discussed further in Section 7.3.6). The idea is that, since the vertices in the underweighted part will not move, we do not need the value of their gain at this step. However, when the overweighted part changes, the gains of the vertices in the new overweighted part and of weight smaller than $\frac{imb(\Pi)}{2}$ will already be set to the proper value.

In addition to the *gains* table, the `GainTableInit_mono` function also returns the position of the inflection point, *inflt*, which is the index of the

Algorithm 36 Initialization of the Gain Table in the Case of Mono-criterion Bipartitioning

Require: $k = 2$ and $\gamma = 1$ and $\texttt{sum}(W) = 1$ (normalized weights)

```
1:  function GainTableInit_mono(M, W, k, Π)
2:      W.sortAscend()
3:      gains, inflt, i ← [ ], 0, 1                    # inflt: inflection point position
4:      while i ≤ n and W[i] < imb(Π) do
5:          if W[i] ≤ imb(Π)/2 then
6:              gains.append(2 · W[i])
7:              inflt ← i
8:          else
9:              gains.append(2 · imb(Π) − 2 · W[i])
10:         end if
11:         i ← i + 1
12:     end while
13:     last ← i − 1                                   # i corresponds to the first settled vertex
14:     return gains, inflt, last
15: end function
```

last vertex whose weight is just smaller or equal to $\frac{imb_c(\Pi)}{2}$. Also, the function returns in variable *last* the number of vertices whose weight is smaller than $imb(\Pi)$, which is the number of non-settled vertices.

In this section, we have detailed the heart of our implementation of the `VNBest` algorithm: the gain table structure. Note that we also keep track of the position of the inflection point (position around which the gains will begin to decrease) and the number of unsettled vertices. In the next section, we will describe how this structure allows us to find the best move quickly.

7.3.5 Finding the Best Move

According to the variation Table 7.3.1 described in Section 7.3.1, the best move is the vertex in the overweighted part whose weight is the closest to $\frac{imb(\Pi)}{2}$.

Example

In Table 7.3.3, that would be either $v_{inflt-1}$ or $v_{inflt+1}$, but not v_{inflt} which is in the underweighted part (so its gain is negative).

Algorithm

Algorithm 37 details how to find the move of best gain using few computations. It starts from the inflection point *inflt* (which was returned by the

Table 7.3.3 – The candidate moves of best gain in the gain table structure of a fictitious mesh are to be searched among the vertices of weight close to $\frac{imb(\Pi)}{2}$

Π_p **is overweighted**				$\frac{imb(\Pi)}{2}$ ↓			$imb(\Pi)$ ↓			
weight	w_1	w_2 ...	$w_{inflt-1}$	w_{inflt}	$w_{inflt+1}$	...	w_{last}	w_{last+1}	...	w_n
part	Π_q	Π_p ...	Π_p	Π_q	Π_p	...	Π_q	Π_q	...	Π_p
$igain(v \to \Pi_q)$		$2w$					$2 \cdot imb(\Pi) - 2w$			
	–	g_2 ...	$g_{inflt-1}$	–	$g_{inflt+1}$	...	–	x	...	x
				←	→					
				best moves				settled vertices		

`GainTableInit_mono` function, and is the index of the last vertex of weight smaller or equal to $\frac{imb(\Pi)}{2}$). Then, it stores in the variable i_{left} the index of the first vertex which is in the overweighted part and of index smaller than or equal to *inflt*. If no such vertex exists, then i_{left} is set to 0.

Algorithm 37 Finding the Best Move Without Computing All the Imbalance Gains (Mono-criterion Bipartitioning Case)

Require: $k = 2$ and $\gamma = 1$
  function `FindBestMove_mono`(M, k, Π, *gains*, *inflt*, *last*)
    $i_{left}, i_{right} \leftarrow inflt, inflt + 1$
    $p_{over}, p_{undr} \leftarrow$ `RangeParts`$(\Pi, 2)$ # Indexes of overweighted and underweighted parts
    while $i_{left} > 0$ and $M[i_{left}] \notin \Pi_{p_{over}}$ **do** $i_{left} \leftarrow i_{left} - 1$ **end while**
    while $i_{right} \leq last$ and $M[i_{right}] \notin \Pi_{p_{over}}$ **do** $i_{right} \leftarrow i_{right} + 1$ **end while**
    if $i_{right} \leq last$ and $\left(i_{left} = 0 \text{ or } gains[i_{right}] > gains[i_{left}]\right)$ **then**
      return i_{right}, p_{undr}
    else
      return i_{left}, p_{undr} # i_{left} may be 0
    end if
  end function

It acts identically with the variable i_{right} that will contain the index of the first vertex in the overweighted part and of index strictly heavier than *inflt*, but smaller than *last*, the number of non-settled vertices. If no such vertex exists, then i_{right} contains *last*.

In the example Table 7.3.3, $i_{left} = inflt - 1$ and $i_{right} = inflt + 1$. The best move is obtained by comparing $gains[i_{left}]$ to $gains[i_{right}]$, when i_{left} and i_{right} are well-defined (neither 0 or n). If both i_{left} and i_{right} are undefined, then 0 is returned, which means that no more move of positive gain exists.

Once we have found the best move, the imbalance will decrease, which means that the gain of some vertices will change. In the next section, we will

explain how to update the gain table. Besides, we need to keep track of the inflection point position and the number of settled vertices.

7.3.6 Gain Table Update After a Move

Overview

After a move, if we call Π' the new partition:

- if the overweighted part remains the same, only the vertices in the overweighted part and of weight ranging between $\frac{imb(\Pi')}{2}$ and $imb(\Pi')$ may change of gain;
- otherwise, the vertices in the new overweighted part need a gain update. However, the gain for the vertices of weight below $\frac{imb(\Pi')}{2}$ does not depend on the imbalance, so it is still $2w$, the value that it was initialized to (remember when we talked about the little trick in the remark in Section 7.3.4). Therefore, the vertices of weight below $\frac{imb(\Pi')}{2}$ do not need a gain update;
- moreover, the imbalance will decrease by definition. For the vertices of weight heavier than $imb(\Pi')$, they become settled vertices, so we will stop keeping track of their gain.

Thus, thanks to a little trick (an erroneous value of the gain for the vertices in the underweighted part, that turns out to be the correct value when the overweighted part changes and if their weight is still smaller than $\frac{imb(\Pi')}{2}$), if the overweighted part changes, it is actually the same as when the overweighted part does not change: in both cases, *we need to update the gains of the vertices in the (possibly new) overweighted part that have a weight ranging between* $\frac{imb(\Pi')}{2}$ *and* $imb(\Pi')$.

Example

Table 7.3.4 displays in its first row an example of the gain table before any move, and in the second row the updates after one move. The second row therefore distinguishes two cases, depending on whether the overweighted part changes or not after the move. Indeed, in both cases, the vertices needing a gain update are the vertices in the (possibly new) overweighted part and of weight ranging between $\frac{imb(\Pi')}{2}$ and $imb(\Pi')$.

The *igain* for the vertices in the overweighted part is indicated, and "–" is inserted for the vertices for which it is not defined. An "x" in place of the *igain* of a vertex means that this vertex is settled, so there is no need to compute its gain.

Algorithm

Algorithm 38 implements the search for the move of maximum *igain*.

Table 7.3.4 – The candidate moves of best gain in the gain table structure of a fictitious mesh are to be searched among the vertices of weight close to $\frac{imb(\Pi)}{2}$

Before the move														
							$\frac{imb(\Pi)}{2}$ ↓				$imb(\Pi)$ ↓			
Π_p **is overweighted**														
weight	w_1	w_2	...	$w_{inflt-1}$	w_{inflt}	$w_{inflt+1}$	...	w_{last}	‖	...				
part	Π_q	Π_p	...	Π_p	Π_q	Π_p	...	Π_q		...				
$igain(v \to \Pi_q)$				$2w$				$2 \cdot imb(\Pi) - 2w$						
	–	g_2	...	$g_{inflt-1}$	–	$g_{inflt+1}$	...	–	‖	x				
					← best moves	→								
										settled vertices				
After the move					$\frac{imb(\Pi')}{2}$ ↓			$imb(\Pi')$ ↓						
1. Π'_p **is overweighted**														
weight	w_1	w_2	...	$w_{inflt'}$	$w_{inflt'+1}$	...	$w_{last'}$	‖	...					
part	Π_q	Π_p	...	Π_p	Π_q	...	Π_p		...					
$igain(v \to \Pi_q)$		$2w$					$2 \cdot imb(\Pi') - 2w$							
	–	g_2	...	$g_{inflt'}$	–	...	$g'_{last'}$	‖	x					
					need gain update				settled vertices					
2. Π'_q **is overweighted**														
$igain(v \to \Pi_p)$		$2w$					$2 \cdot imb(\Pi') - 2w$							
	g_1	–	...	–	$g'_{inflt'+1}$	...	–	‖	x					

Algorithm 38 Updating the Gain Table in Case of Mono-criterion Bipartitioning

Require: $k = 2$ and $\gamma = 1$ and $\texttt{sum}(W) = 1$ (normalized weights)
  function `GainTableUpdate_mono`(M, W, Π, *gains*, *last*)
    $p_{over}, \_ \leftarrow \texttt{RangeParts}(\Pi, 2)$ # Index of overweighted part
    $i \leftarrow 1$
    while $i \leq last$ and $W[i] \leq \frac{imb(\Pi)}{2}$ **do** $i \leftarrow i + 1$ **end while**
    $inflt \leftarrow i - 1$
    while $i \leq last$ and $W[i] < imb(\Pi)$ **do**
      if $\Pi[i] = p_{over}$ **then**
        $gains[i] \leftarrow 2 \cdot imb(\Pi) - 2W[i]$ # Gain update
      end if
      $i \leftarrow i + 1$
    end while
    $last \leftarrow i$ # i is the number of unsettled vertices
    return *gains*, *inflt*, *last*
  end function

The `GainTableUpdate` function begins by scanning the *gains* table until it finds the inflection point. Before reaching the inflection point, no update is required. This is only after the inflection point that the gains must be recomputed. Note that only the gains of the vertices in the overweighted part need to be computed. Indeed, the vertices in the underweighted part have a negative gain, and they will not be considered by the `FindBestMove` function. When the weights of the vertices become heavier or equal to the imbalance, the update stops because these vertices are settled.

Remark

After a move, by definition of the algorithm, the imbalance decreases. Along the algorithm, the number of settled vertices is thus increasing, and a decreasing number of vertices need to be considered.

7.3.7 Global Algorithm and Conclusion

Throughout this section, we have detailed a gain table structure that allows, in the mono-criterion, bipartitioning case, to reduce the amount of computation. The exact amount of computation required is difficult to estimate, but the thing is that:

- it is at least $n \log(n)$ (n is the number of cells in the mesh), because our implementation needs to sort the cells by increasing weight;
- when a move of great gain is performed, then, the number of settled vertices should increase, because the vertices of weight heavier than the imbalance are settled, so the amount of computation remaining decreases;
- when a move of small gain is performed, then only a few gains should be updated, because only the vertices of weight ranging between half the imbalance and the imbalance change of gain, so it amounts for a reduced amount of computation.

In both cases, the amount of computation is either small or reduced for the rest of the algorithm, so we believe that our implementation of a gain table structure allows a great reduction in the amount of computation required over the greedy implementation of the `VNBest` algorithm. This will be asserted experimentally in Section 10.1.1.

Algorithm 39 implements the `VNBest_mono` function that relies on the gain table structure. To sum up, the key features of the gain table are that:

- the vertices in the gain table are sorted by increasing weight;
- the vertices of weight heavier than or equal to $imb(\Pi)$ do not need to be considered, because they will have a negative gain until the end of the algorithm (we say that they are "settled");
- the vertex of maximum gain can be quickly found, because it is the vertex in the overweighted part and whose weight is closest to $\frac{imb(\Pi)}{2}$;

Algorithm 39 Mono-criterion Version of the Steepest Descent-like Vector-of-Numbers Partitioning Algorithm Using a Gain Table To Reduce the Amount of Computation and Speedup the Search

Require: Π: an initial partition of M (may be random)
1: **function** `VNBest_mono`(M, W, W_{com}, t, Π)
2: $gains, infl t, last \leftarrow$ `GainTableInit_mono`(M, W, Π)
3: $i, q \leftarrow$ `FindBestMove_mono`$(M, \Pi, gains, infl t, last)$
4: **while** $igain(M[i] \rightarrow \Pi_q) > 0$ **do**
5: $\Pi[i] \leftarrow \Pi_q$
6: $gains, infl t, last \leftarrow$ `GainTableUpdate_mono`$(M, W, \Pi, gains, last)$
7: $i, q \leftarrow$ `FindBestMove_mono`$(M, k, \Pi, gains, infl t, last)$
8: **end while**
9: **return** Π
10: **end function**

- after a move, if imb' is the new imbalance, the only update in the gain table concerns the vertices in the (possibly new) overweighted part of weight between half imb' and imb;
- indeed, the vertices in the overweighted part of weight w under $\frac{imb(\Pi)}{2}$ have a constant gain of $2w$. Using a little trick, their gain do not need to be updated after a move if their weight is still below half the new imbalance.

The next section will address a more complex situation, as we will consider the multi-criteria bipartitioning case.

7.4 Multi-criteria, Bipartitioning Case

In this section, we consider bipartitioning a multi-criteria mesh. The difference with the mono-criterion case is that a move that may reduce the imbalance for one criterion may unfortunately increase it for another, so computing the *igain* of one move becomes more difficult, and finding the best move is more tricky.

This section is organized similarly as the previous one.

7.4.1 Expression of the Gain of a Move

Proposition 5 formulates similar equations for the gain per criterion as in the mono-criterion case, but concerning the gain per criterion $igain_c$ (introduced at the beginning of this chapter in Definition 32).

Proposition 5 (Gain in Imbalance per Criterion for Multi-criteria Bipartitioning)

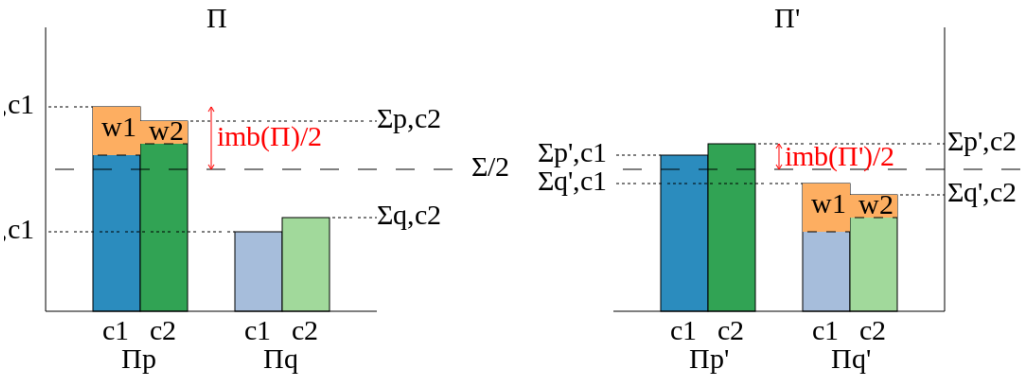


Figure 7.4.1 – Example of how the imbalance changes when a vertex (in orange) moves (left: before move, right: after move)

Given a mesh M and a bipartition $\Pi = \{\Pi_1, \Pi_2\}$ of M, let v be a cell or vertex in M.

We assume that $v \in \Pi_p$, and consider moving it to $\Pi_q \neq \Pi_p$, which would lead to the partition $\Pi' = \{\Pi'_p, \Pi'_q\}$, such that $\Pi'_p = \Pi_p \setminus \{v\}$ and $\Pi'_q = \Pi_q \cup \{v\}$.

We denote by w_c the (normalized) weight of v for criterion c ($w_c = \frac{W(v)[c]}{\Sigma_c}$).

Then, the gain of v for criterion c is

$$igain_c(v) = \begin{cases} -2w_c & \text{if } imb_c(\Pi_q) \geq 0 \ ; \\ 2 \cdot imb_c(\Pi) - 2w_c & \text{if } imb_c(\Pi_q) \leq 0 \text{ and } w \geq \frac{imb_c(\Pi)}{2} \ ; \\ 2w_c & \text{if } imb_c(\Pi_q) \leq 0 \text{ and } w \leq \frac{imb_c(\Pi)}{2} \ . \end{cases}$$

Proof

Let us consider any weight function $W_c : v \mapsto W(v)[c]$; in this case, the instance (M, W_c) is a mono-criterion mesh, of gain function identical to $igain_c$. So, the mono-criterion Proposition 3 holds for $igain_c$, which concludes the proof.

□

If Proposition 5 is similar to the mono-criterion case, it is not as powerful. Indeed, there are no means to compute the *igain* of a vertex from its $igain_c$ for each criterion. This is due to the fact that usually, $igain(v) \neq \max_c igain_c(v)$, as shown in Figure 7.4.1, which shows an example of the move of a vertex (in orange) when there are two criteria, c_1 and c_2.

On this figure, the vertex of weight (w_1, w_2) in orange is moved from Π_p to Π_q. However, the most imbalanced criterion changes, so there are no means to

express the gain other than using the basic equations:

$$\begin{aligned} gain(v) &= imb(\Pi) - imb(\Pi') = \max_c(imb_c(\Pi)) - \max_c(imb_c(\Pi')) \\ &= \max_c(imb_c(\Pi)) - \max_c(imb_c(\Pi) - gain_c(v)) \end{aligned}$$

Since we cannot compute directly the *igain* of a vertex in the multi-criteria case, we will need to use the $igain_c$ functions. These functions lead to the same property as in the mono-criterion case on settled vertices, as we will see in the next section. Then, we will describe in Section 7.4.3 the gain table structure that we will use in order to find quickly the move of best gain and to reduce the amount of computation.

7.4.2 Settled Vertices

Proposition 5 leads to a property similar to that of the mono-criterion case, about some vertices that are settled.

Proposition 6 (Settled Vertices for the Multi-criteria Bipartitioning Case)
Given a mesh M and a bipartition Π of M, let v be a cell or its corresponding vertex in M. We assume that the weight function W is normalized ($\forall c, \sum_{v \in M} W(v)[c] = 1$). Then we have:

$$\exists c, W(v)[c] \geq imb(\Pi) \implies \begin{array}{l} v \text{ will have a negative igain until the} \\ \text{end of the algorithm.} \end{array}$$

Proof

Given a mesh M and a bipartition Π of M, let v be a cell or its corresponding vertex in M, and let w_c be the (normalized) weight of v for criterion c. Let Π_q be the target part of v.

We assume that $w_c \geq imb(\Pi)$. Then, using Proposition 3, we know that the gain for v is:

$$\begin{aligned} igain(v) &= imb(\Pi) - imb(\Pi') \\ &\leq imb(\Pi) - imb_c(\Pi') && \text{because } imb(\Pi') \geq imb_c(\Pi') \\ &\leq imb(\Pi) - (imb_c(\Pi) - igain_c(v)) = imb(\Pi) - imb_c(\Pi) + igain_c(v) \ . \end{aligned}$$

Using Proposition 5 and the fact that $w_c \geq imb(\Pi) \geq \frac{imb_c(\Pi)}{2}$,

$$igain(v) \leq \begin{cases} imb(\Pi) - imb_c(\Pi) - 2w & \text{if } imb_c(\Pi_q) \geq 0 \ ; \\ imb(\Pi) - imb_c(\Pi) + (2 \cdot imb_c(\Pi) - 2w) & \text{otherwise} \ . \end{cases}$$

$$igain(v) \leq \begin{cases} imb(\Pi) - imb_c(\Pi) - 2w & \text{if } imb_c(\Pi_q) \geq 0 \ ; \\ imb(\Pi) + imb_c(\Pi) - 2w & \text{otherwise} \ . \end{cases}$$

If $imb_c(\Pi_q) \geq 0$, then $imb(\Pi) \leq w$ by our assumption and $w \geq 0$ by definition, so $imb(\Pi) - imb_c(\Pi) - 2w \leq 0$.

Otherwise, $imb_c(\Pi) \leq imb(\Pi) \leq w$, so $imb(\Pi) + imb_c(\Pi) - 2w \leq 2imb(\Pi) - 2w \leq 0$.

We have shown that the gain in imbalance for v is negative. Using the same reasoning as in Proposition 4 (on settled vertices in the case of mono-criterion bipartitioning), we can conclude that, by definition of `VNBest`, v will not switch to another part anymore during the algorithm.

□

Proposition 6 allows us to avoid computing the gain of the settled vertices. In the multi-criteria case, a vertex is settled if its weight for at least one criterion is heavier than the (global) imbalance. Using this property and the expression of the $igain_c$ functions computed in the previous section, the next section will define the gain table that we will use in the multi-criteria bipartitioning case.

7.4.3 Gain Table Structure (Multi-criteria Case)

Gain Table Organization

The multi-criteria gain table is a set of γ gain subtables: one per criterion. We will navigate between each subtable in order to find the move reducing the most the imbalance. In the cth subtable, the vertices are ordered by increasing weight for criterion c, which allows us to keep the same characteristics for $igain_c$ as for a mono-criterion table: the $igain_c$ in the subtable is increasing up to a weight close to $\frac{imb_c(\Pi)}{2}$, then it is decreasing, and become negative when the weights become heavier than $imb_c(\Pi)$.

Example

Table 7.4.1 shows an example of the gain table for a mesh of $n = 12$ cells and a weight distribution of two criteria. It is therefore composed of two gain subtables, one per row.

In the first row, the vertices are ordered by increasing weight according to the first criterion. In this example, we have $W(v_3)[1] \leq W(v_5)[1] \leq ... \leq W(v_8)[1]$. The same holds for the second row, but considering the second criterion, so that $W(v_1)[2] \leq W(v_{12})[2] \leq ... \leq W(v_6)[2]$.

The most imbalanced criterion, c_{max}, is assumed to be c_2. Therefore, as stated in the previous section, all vertices whose weight is heavier than $imb_{c_2}(\Pi)$ for at least one criterion are settled. These vertices are colored in gray and have an "x" as *igain* value. In the example, the settled vertices are v_6, v_8 and v_{12}. Note that some settled vertices, as v_6 or v_{12}, may appear in place of "unsettled" vertices in some subtables, but we just will ignore them.

Table 7.4.1 – The gain table structure for a fictitious multi-criteria mesh

Gain subtable for c_1				$\frac{imbc_1(\Pi)}{2}$ ↓				$imb_{c_1}(\Pi)$ ↓		$imb_{c_2}(\Pi)$ ↓		
vertex	v_3	v_5	v_6	v_1	v_7	v_{11}	v_{10}	v_4	v_2	v_9	v_{12}	v_8
part	Π_p	Π_q	Π_q	Π_p	Π_p	Π_p	Π_q	Π_q	Π_p	Π_q	Π_p	Π_q
$gain_{c_1}$	$g_{3,1}$	$g_{5,1}$	x	$g_{1,1}$	$g_{7,1}$	$g_{11,1}$	$g_{10,1}$	$g_{4,1}$	$g_{2,1}$	$g_{9,1}$	x	x
											settled vertices	

Gain subtable for $c_2 = c_{max}$						$\frac{imbc_2(\Pi)}{2}$ ↓				$imb_{c_{max}}(\Pi)$ ↓		
vertex	v_1	v_{12}	v_3	v_9	v_2	v_{10}	v_5	v_7	v_{11}	v_4	v_8	v_6
part	Π_p	Π_p	Π_p	Π_q	Π_p	Π_q	Π_q	Π_p	Π_p	Π_q	Π_q	Π_q
$gain_{c_2}$	$g_{1,2}$	x	$g_{3,2}$	$g_{9,2}$	$g_{2,2}$	$g_{10,2}$	$g_{5,2}$	$g_{7,2}$	$g_{11,2}$	$g_{4,2}$	x	x
											settled vertices	

Algorithm

Algorithm 40 describes the `GainTableInit` function, that initializes a multi-criteria gain table. It basically calls the `GainSubtableInit` function for each criterion. Note that, instead of one inflection point position as in the mono-criterion case, the multi-criteria version returns a list of inflection point positions, one per criterion. However, the number of unsettled vertices (variable *last*) in each subtable is independent from the criterion.

The `GainSubtableInit` function is nearly the same as the `GainTableInit_mono` function, but using W_c and imb_c instead of W and imb. The only part that changes is when to stop computing the gains, which uses the global imbalance imb instead of imb_c. This is the particularity of the multi-criteria case: a vertex of negative $igain_c$ may be of positive $igain$.

We have described the gain table structure that we will use in the multi-criteria case. In the next section, we will explain how to navigate between the subtables in order to find the move of greatest $igain$ in a few steps.

Finding the Best Move

In order to reduce the imbalance, the imbalance for the most imbalanced criterion c_{max} must decrease. Therefore, the vertex moved must belong to the overweighted part for c_{max}. Unlike for the mono-criterion case, nevertheless, finding the vertex of best $igain_{c_{max}}$ does not guarantee to have found the vertex leading to the smallest imbalance. In order to find the vertex of best gain, we will use the following Proposition 7, which allows us to stop the search when a move does not improve the most imbalanced criterion.

Algorithm 40 Initialization of the Gain Table in Case of Multi-criteria Bipartitioning

Require: $k = 2$ and $\forall c, \texttt{sum}(W_c) = 1$ (normalized weights)

1: **function** `GainSubtableInit`(M, W, Π, c)
2: W_c.`sortAscend`()
3: $gains_c, inflt_c, i \leftarrow [\,], 0, 1$
4: **while** $i \leq n$ and $W_c[i] < imb(\Pi)$ **do** # Independent from c
5: **if** $W_c[i] \leq \frac{imb_c(\Pi)}{2}$ **then** # Depends on c
6: $gains_c$.`append`$(2 \cdot W_c[i])$
7: $inflt_c \leftarrow i$
8: **else**
9: $gains_c$.`append`$(2 \cdot imb_c(\Pi) - 2 \cdot W_c[i])$
10: **end if**
11: $i \leftarrow i + 1$
12: **end while**
13: $last \leftarrow i$
14: **return** $gains_c$, $inflt_c$, $last$
15: **end function**

16: **function** `GainTableInit`(M, W, k, Π)
17: $gains, inflt \leftarrow [\,], [\,]$
18: **for** $c \leftarrow 1, \gamma$ **do**
19: $gains_c, inflt_c, last \leftarrow$ `GainSubtableInit`(M, W, Π, c)
20: $gains$.`append`$(gains_c)$
21: $inflt$.`append`$(inflt_c)$
22: **end for**
23: **return** $gains$, $inflt$, $last$
24: **end function**

Proposition 7 (*igain* for the Vertices of Smaller $igain_{c_{max}}$)

Given a mesh M and a bipartition $\Pi = \{\Pi_1, \Pi_2\}$ of M, let v be a cell or its corresponding vertex in M.

We assume that, when moving v to the part it does not belong to, the most imbalanced criterion remains the same. Hence:

$$\forall \tilde{v} \in M, igain_{c_{max}}(\tilde{v}) \leq igain_{c_{max}}(v) \implies igain(\tilde{v}) \leq igain(v) \ .$$

Proof

Given a mesh M and a bipartition $\Pi = \{\Pi_1, \Pi_2\}$ of M, let v be a cell or its corresponding vertex in M.

We assume that $v \in \Pi_p$, and consider moving v to $\Pi_q \neq \Pi_p$, which would lead to the partition $\Pi' = \{\Pi'_p, \Pi'_q\}$, such that $\Pi'_p = \Pi_p \setminus \{v\}$ and $\Pi'_q = \Pi_q \cup \{v\}$.

Let $\tilde{v}$ be another cell or vertex in M, in part $\Pi_{\tilde{p}}$. We consider moving $\tilde{v}$ to $\Pi_{\tilde{q}} \neq \Pi_{\tilde{p}}$, which would lead to the partition $\tilde{\Pi}' = \{\tilde{\Pi}'_{\tilde{p}}, \tilde{\Pi}'_{\tilde{q}}\}$, such that $\tilde{\Pi}'_{\tilde{p}} = \Pi_{\tilde{p}} \setminus \{\tilde{v}\}$ and $\tilde{\Pi}'_{\tilde{q}} = \Pi_{\tilde{q}} \cup \{\tilde{v}\}$.

Let c_{max} be the most imbalanced criterion of Π. We assume that c_{max} is also the most imbalanced criterion of Π' (which means that the most imbalanced criterion remains the same when moving v). We will denoted by $\tilde{c}_{max}$ the most imbalanced criterion of $\tilde{\Pi}$.

Finally, we assume that $igain_{c_{max}}(\tilde{v}) \leq igain_{c_{max}}(v)$, and we will show that $igain(\tilde{v}) \leq igain(v)$.

We have:

$$\begin{aligned} imb(\tilde{\Pi}') &= imb_{\tilde{c}_{max}}(\tilde{\Pi}') = imb_{\tilde{c}_{max}}(\Pi) - igain_{\tilde{c}_{max}}(\tilde{v}) \\ &\leq imb(\Pi) - igain_{\tilde{c}_{max}}(\tilde{v}) \ . \end{aligned} \tag{7.1}$$

Besides, we also have:

$$\begin{aligned} imb(\tilde{\Pi}') \geq imb_{c_{max}}(\tilde{\Pi}') &= imb_{c_{max}}(\Pi) - igain_{c_{max}}(\tilde{v}) \\ &= imb(\Pi) - igain_{c_{max}}(\tilde{v}) \ . \end{aligned} \tag{7.2}$$

Combining 7.2 and 7.1, we get:

$$imb(\Pi) - igain_{c_{max}}(\tilde{v}) \leq imb(\tilde{\Pi}') \leq imb(\Pi) - igain_{\tilde{c}_{max}}(\tilde{v}) \ ,$$

so

$$igain_{\tilde{c}_{max}}(\tilde{v}) \leq igain_{c_{max}}(\tilde{v})$$

and as we assumed that $igain_{c_{max}}(\tilde{v}) \leq igain_{c_{max}}(v)$,

$$igain_{\tilde{c}_{max}}(\tilde{v}) \leq igain_{c_{max}}(v) \ . \tag{7.3}$$

Firstly, since c_{max} is the most imbalanced criterion for both Π and Π', we have:

$$\begin{aligned} igain_{c_{max}}(v) &= imb_{c_{max}}(\Pi) - imb_{c_{max}}(\Pi') = imb(\Pi) - imb(\Pi') \\ &= igain(v) \ . \end{aligned} \tag{7.4}$$

Secondly, we also have

$$\begin{aligned} igain(\tilde{v}) &= imb(\Pi) - imb(\tilde{\Pi}') \\ &= imb(\Pi) - \max_c(imb_c(\Pi) - gain_c(\tilde{v})) \ , \end{aligned}$$

but

$$\begin{aligned} \max_c(imb_c(\Pi) - gain_c(\tilde{v})) &\geq imb_{\tilde{c}_{max}}(\Pi) - igain_{\tilde{c}_{max}}(\tilde{v}) \\ &\geq imb(\Pi) - igain_{\tilde{c}_{max}}(\tilde{v}) \ , \end{aligned}$$

so

$$\begin{aligned} imb(\Pi) - \max_c(imb_c(\Pi) - gain_c(\tilde{v})) &\leq imb(\Pi) - (imb(\Pi) - igain_{\tilde{c}_{max}}(\tilde{v})) \\ &\leq igain_{\tilde{c}_{max}}(\tilde{v}) \ , \end{aligned}$$

and

$$igain(\tilde{v}) \leq igain_{\tilde{c}_{max}}(\tilde{v}) \ . \tag{7.5}$$

Combining 7.3, 7.4 and 7.5, we finally obtain:

$$igain(\tilde{v}) \leq igain(v) \ ,$$

which concludes the proof.

□

The idea to find the best move is first to consider the moves of best gain for c_{max}. Then, we will have to examine each vertex (by decreasing $igain_{c_{max}}$) in the gain subtable for c_{max} as long as we do not find a vertex which, when moved, does not change c_{max}. Once we have found one, as stated in Proposition 7, all the remaining vertices will have a smaller *igain*.

Algorithm

Algorithm 41 implements the `FindBestMove` function. This function returns the vertex leading to the best gain without browsing all the gain table.

First, the function initializes some data. In particular, the variables *lft* and *rgt* will state if the search should be continued respectively to the left and to the right from the current positions, which are respectively i_{lft} and i_{rgt}. As we consider the vertices of best gain for c_{max}, the initial positions are the inflection

Algorithm 41 Finding the Best Move Without Computing All the Imbalance Gains (Bipartitioning Case)

Require: $k = 2$ and $\forall c, \texttt{sum}(W_c) = 1$ (normalized weights)

function `FindBestMove`(M, k, Π, *gains*, *inflt*, *last*)
  $c_{max} \leftarrow \texttt{argmax}([imb_1(\Pi) \mathbin{.?.} imb_\gamma(\Pi)])$ # Most imbalanced criterion
  $p_{over}, \_ \leftarrow \texttt{RangeParts}(\Pi, 2)$ # Index of overweighted part
  candidates $\leftarrow [\,]$ # Stores the vertices that can have the greatest *igain*
  lft, *rgt* $\leftarrow$ `True`, `True`
  $i_{lft}, i_{rgt} \leftarrow inflt[c_{max}], inflt[c_{max}] + 1$
  while *lft* or *rgt* **do**
    if *lft* **then** # Explore to the left from *inflt* to find a vertex in $\Pi_{p_{over}}$
      while $i_{lft} > 0$ and $M[i_{lft}] \notin \Pi_p$ **do** $i_{lft} \leftarrow i_{lft} - 1$ **end while**
    end if
    if $i_{lft} > 0$ **then** $g_{lft} \leftarrow gains_{c_{max}}[i_{lft}]$ **else** $g_{lft} \leftarrow -1$ **end if**
    if *rgt* **then** # Explore to the right from *inflt* to find a vertex in $\Pi_{p_{over}}$
      while $i_{rgt} \leq last$ and $M[i_{rgt}] \notin \Pi_p$ **do** $i_{rgt} \leftarrow i_{rgt} + 1$ **end while**
    end if
    if $i_{rgt} > 0$ **then** $g_{rgt} \leftarrow gains_{c_{max}}[i_{rgt}]$ **else** $g_{rgt} \leftarrow -1$ **end if**
    lft, *rgt* $\leftarrow$ `False`, `False`
    if $g_{lft} > -1$ and $g_{lft} \geq g_{rgt}$ **then** # If i_{lft} has the best gain
      candidates.`append`(i_{lft})
      # Continue searching to the left if c_{max} does not change
      $lft \leftarrow (\texttt{c}_{\texttt{max}}\texttt{AfterMove}(gains, \Pi, i_{lft}) = c_{max})$
      if lft **then** $i_{lft} \leftarrow i_{lft} - 1$ **end if**
    else if $g_{rgt} > -1$ **then** # Else if i_{rgt} has the best gain
      candidates.`append`(i_{rgt})
      # Continue searching to the right if c_{max} does not change
      $rgt \leftarrow (\texttt{c}_{\texttt{max}}\texttt{AfterMove}(gains, \Pi, i_{rgt}) = c_{max})$
      if *rgt* **then** $i_{rgt} \leftarrow i_{rgt} + 1$ **end if**
    end if # Else, the search will stop (end of table)
  end while
  return `BestAmongCandidates`(Π, *candidates*, *gains*), $p_{overwgt}$
end function

function `BestAmongCandidates`(Π, *candidates*, *gains*)
  $i_{best}, imb_{min} \leftarrow 0, 1$
  for $i \in$ *candidates* **do**
    $imb_i \leftarrow \texttt{max}([imb_1(\Pi) - gains_1[i] \mathbin{.?.} imb_\gamma(\Pi) - gains_\gamma[i])$
    if $imb_i < imb_{min}$ **then** $i_{best}, imb_{min} \leftarrow i, imb_i$ **end if**
  end for
  return i_{best}
end function

point and the inflection point + 1.

Then, while we search to the left and/or to the right, we will repeat the following: explore to the left and/or to the right to find a vertex in the overweighted part. Then, between the vertex on the left and the vertex on the right, select that of greatest $igain_{c_{max}}$, add it to the *candidates* list, and see if the most imbalance criterion improves when it is moved. If it improves, we will need to go on filling the *candidates* list. If it does not improve, then we can stop the search.

At this step, the *candidates* list contains all the vertices that may have the greatest *igain*. Finally, the vertex that actually has the greatest *igain* is selected from the *candidates* list using the `BestAmongCandidates` function.

Finally, after having performed a move, we will need to update all the gain subtables accordingly. The principle is the same as in the mono-criterion case, although it must be performed for each of the gain subtables, and that it is a bit different to define the settled vertices.

Updating the Gains After a Move

Updating the gain table means updating the $igain_c$ values for each criterion, which means to update each of the gain subtables. This is nearly the same as updating a mono-criterion gain table. Algorithm 42 details the `GainSubtableUpdate` function, which is nearly the same as the `GainTableUpdate_mono` function used in the mono-criterion case.

The first difference is that whether the vertices in the cth gain table need a gain update, does not depend only on c, but also on c_{max}. Indeed, vertices in the underweighted part for c_{max} do not need a gain update, because they will have a negative *igain* as long as the overweighted part for c_{max} does not change, so we will update their $igain_c$ values at this time.

The second difference is that the place of the inflection point depends on the criterion considered. Moreover, the place where the settled vertices sub-array begins is computed using the "global" imbalance.

7.4.4 Global Algorithm and Conclusion

In this section, we have detailed the gain table structure for the multi-criteria, bipartitioning case. This structure should allow us to reduce the amount of computation, whereas it is very difficult to estimate it in the multi-criteria case. However, we will see in Section 10.1.1 that, using our implementation, the `VNBest` algorithm is in practice fast enough. Algorithm 43 details our implementation (which is very similar to that of the `VNBest_mono` algorithm) using the functions defined throughout this section.

The key differences with the mono-criterion case are:

Algorithm 42 Updating the Gain Subtable in Case of Bipartitioning

Require: $k = 2$ and $\forall c, \texttt{sum}(W_c) = 1$ (normalized weights)
  function `GainSubtableUpdate`(M, W, Π, *gains*, *last*, c)
    $p_{over}, \_ \leftarrow$ `RangeParts`$(\Pi, 2)$ # Index of overweighted part (for c_{max})
    $i \leftarrow 1$ # No gain update
    while $i \leq last$ and $W_c[i] \leq \frac{imb_c(\Pi)}{2}$ **do** # Depends on imb_c
      $i \leftarrow i + 1$
    end while
    $infIt \leftarrow i - 1$
    while $i \leq last$ and $W_c[i] < imb(\Pi)$ **do** # Depends on imb
      if $\Pi[i] = p_{over}$ **then** # Depends on c_{max}
        $gains_c[i] \leftarrow 2 \cdot imb(\Pi) - 2W_c[i]$ # Gain update
      end if
      $i \leftarrow i + 1$
    end while
    $last \leftarrow i - 1$ # i corresponds to the first settled vertex
    return $gains_c$, *infIt*, *last*
  end function

Algorithm 43 Multi-criteria Version of the Steepest Descent-like Vector-of-Numbers Partitioning Using a Gain Table to reduce the amount of computation and speedup the search

Require: Π: an initial partition of M (may be random)
1: **function** `VNBest`(M, W, W_{com}, t, Π)
2: $gains, infIt, last \leftarrow$ `GainTableInit`(M, W, Π)
3: $i, q \leftarrow$ `FindBestMove`$(M, \Pi, gains, infIt, last)$
4: **while** $igain(M[i] \rightarrow \Pi_q) > 0$ **do**
5: $\Pi[i] \leftarrow \Pi_q$
6: **for** $c \leftarrow 1, \gamma$ **do**
7: $gains, infIt, last \leftarrow$ `GainSubtableUpdate`$(M, W, \Pi, gains, last, c)$
8: **end for**
9: $i, q \leftarrow$ `FindBestMove`$(M, k, \Pi, gains, infIt, last)$
10: **end while**
11: **return** Π
12: **end function**

- there is one gain table for each criterion;
- the vertices in the cth gain table are sorted by increasing weight for criterion c;
- we only need to consider the vertices belonging to the overweighted part for c_{max};
- the vertices whose weight for at least one criterion is heavier or equal to $imb(\Pi)$ do not need to be considered, because they are settled in their current part. This is a notable difference with the mono-criterion case (a straightforward application would be to consider $imb_c(\Pi)$, but in fact it is not correct);
- finding the vertex of maximum *igain* requires more computations than in the mono-criterion case. This is the biggest difference with the mono-criterion case, because the vertex closest to the inflection point may not be that of greatest *igain*;
- however, thanks to the useful Proposition 7, we need to consider only the vertices of weight closest to $\frac{imb_{c_{max}}(\Pi)}{2}$, until moving one of them does not change the most imbalanced criterion c_{max};

The next section will consider the k-partitioning case.

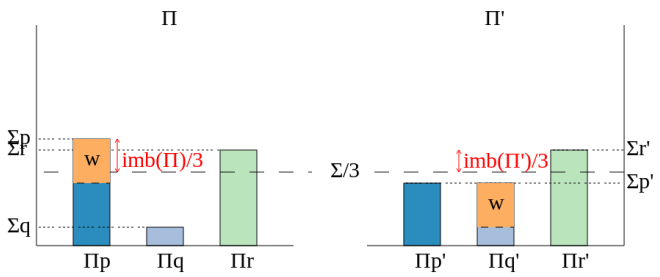
7.5 Discussion on the Extension to k-partitioning

This section will discuss the extension of the `VNBest` algorithm to the k-partitioning case, with $k > 2$. There is a major difference when dealing with $k > 2$ parts. Indeed, `VNBest` searches for a move that minimizes the imbalance, which is a minimization of a maximum. With two parts, the imbalances are opposite, which allow useful properties, for example Proposition 5 that allowed to compute easily $igain_c$. This proposition does not hold anymore when dealing with more than two parts, as shown in the following example, which considers a mono-criterion mesh.

Example

The next figure displays an example of two 3-partition for a mono-criterion instance. The imbalance of a partition is shown by a red arrow. Indeed,

$$\begin{aligned} imb(\Pi) &= \max(imb(\Pi_p), imb(\Pi_q), imb(\Pi_r)) = imb(\Pi_p) \quad \text{(in this example)} \\ &= \frac{\Sigma_p - \frac{\Sigma}{k}}{\frac{\Sigma}{k}} = 3 \times \underbrace{\left(\Sigma_p - \frac{1}{3}\right)}_{\text{red arrow}} \text{ because the weights are normalized.} \end{aligned}$$



We can see that the gain of moving v of weight w is here $imb(\Pi_p) - imb(\Pi_r)$, which cannot be formulated as an expression involving only $imb(\Pi)$ and w, as in to the bipartitioning case.

There is no means to compute directly the gain of a move like in the bipartitioning case, and, more importantly, finding which move will reduce the most the imbalance is complex, especially in the multi-criteria case. Therefore, for k-partitioning, we prefer to rely on the recursive bisection algorithm (`RB` defined in Algorithm 14 on page 84). Indeed, whereas some argue that a direct k-partitioning algorithm may optimize better than a recursive bisection algorithm, we do not need to find the partition with minimal imbalance, but "only" a partition of imbalance smaller than the given tolerance.

Nevertheless, we will state some results that are still valid when dealing with $k > 2$ parts.

- In order to decrease the imbalance, one needs to move a vertex from the heaviest part for the most imbalanced criterion. Therefore, the best move should be searched among the vertices in part $\Pi_{p_{max}}$ such that $imb_{c_{max}}(\Pi_{p_{max}}) = \max_{\Pi_p \in \Pi}(imb_{c_{max}}(\Pi_p))$.
- In the mono-criterion case, the vertices whose weight is heavier than $imb(\Pi)$ are settled, as stated in the following Proposition 8. We assume this should also be true in the multi-criteria case.

Proposition 8 (Settled Vertices for the Mono-criterion Case)

Given a mesh M and a partition Π of M, let v be a cell or its corresponding vertex in M. Let its normalized weight be $w = \frac{W(v)}{\Sigma}$. Then:

$$w \geq imb(\Pi) \implies v \text{ will have a negative igain until the end of algorithm.}$$

Proof

To simplify the notations, we will assume without loss of generality that $v \in \Pi_1$, and we will consider moving v to the target part Π_2.

We denote by $\Pi' = \{\Pi'_1, ..., \Pi'_k\}$ the partition of M obtained after moving v, which means that $\Pi'_1 = \Pi_1 \setminus \{v\}$, $\Pi'_2 = \Pi_2 \cup \{v\}$ and $\forall p \notin \{1, 2\}, \Pi'_p = \Pi_p$.

We assume that $w \geq imb(\Pi)$, and we will show that $imb(\Pi') \geq imb(\Pi)$. We have:

$$
\begin{aligned}
imb(\Pi') &= \max(imb(\Pi'_1), imb(\Pi'_2), imb(\Pi'_3), ..., imb(\Pi'_k)) \\
&= \max\left(\frac{\Sigma_1 - w - \frac{\Sigma}{k}}{\frac{\Sigma}{k}}, \frac{\Sigma_2 + w - \frac{\Sigma}{k}}{\frac{\Sigma}{k}}, imb(\Pi_3), ..., imb(\Pi_k)\right) \\
&= \max(imb(\Pi_1) - kw, imb(\Pi_2) + kw, imb(\Pi_3), ..., imb(\Pi_k)) \\
&\geq \max(imb(\Pi_1), imb(\Pi_2) + kw, imb(\Pi_3), ..., imb(\Pi_k)) \\
&\geq \max(imb(\Pi), imb(\Pi_2) + kw, imb(\Pi), ..., imb(\Pi)) \\
&\geq \max(imb(\Pi), imb(\Pi_2) + kw) \ . \qquad (7.6)
\end{aligned}
$$

Besides, we have $\sum_{p=1}^{k} imb(\Pi_p) = 0$, so:

$$
\begin{aligned}
imb(\Pi_2) + kw &= -\sum_{p \neq 2} (imb(\Pi_p)) + kw \\
&\geq -\sum_{p \neq 2} (imb(\Pi)) + kw \\
&\geq -(k-1) \cdot imb(\Pi) + kw \ ,
\end{aligned}
$$

and using $w \geq imb(\Pi)$:

$$
imb(\Pi_2) + kw \geq w \geq imb(\Pi) \ . \qquad (7.7)
$$

Therefore, combining the inequations 7.6 and 7.7, we obtain that $imb(\Pi') \geq imb(\Pi)$, which concludes the proof.

□

Conclusion

The steepest descent algorithm is much more complex when dealing with $k > 2$ parts. Some results may still be valid, but proving them is more difficult. Therefore, for k-partitioning, we propose to rely on the recursive bisection algorithm, which is very simple to use, and if `VNBest_bipart` returns partitions of small imbalance, the recursive bisection should be able to return a balanced solution.

7.6 Conclusion on the Steepest Descent Algorithm

In this section, we have introduced a steepest descent algorithm to solve the vector-of-numbers partitioning problem. We have first stated some results for the mono-criterion, bipartitioning case, before explaining to which extent they could be adapted to the multi-criteria partitioning case. The key issues are:

- the number of possible moves decreases along the algorithm, because a vertex whose weight for at least one criterion is heavier than the current imbalance will not move anymore during the algorithm (Proposition 6);
- the best move may be found without testing all possible moves. Firstly, we only need to consider vertices in the overweighted part for the most imbalanced criterion c_{max}. Then, starting from the vertices whose gain for c_{max} is the highest, we test if, when moving them, the overweighted part for the most imbalanced criterion changes. When it does not change, it means that the remaining vertices will not decrease the imbalance as much as the current one (Proposition 7). Therefore, the best move is among the already computed ones;
- for bipartitioning, $igain_c$ has a predefined formula (Proposition 5). From this formula, we have designed a gain table that stores $\gamma \times n$ gains, that is, the gain of each vertex for each criterion. This gain table allows us to access quickly the vertices of best gain and to avoid recomputing the gains when they do not change.

The `VNFirst` and the `VNBest` algorithms aim to be applied on the coarsened graph. In Section 10.1, we will therefore compare their performance on different levels of coarsened graphs.

The goal of these initial partitioning algorithms is to return quickly a balanced partition of the coarsest graph. This partition will be successively prolonged and refined at finer levels, until a partition of the original graph is computed. The refinement phase, which is the subject of the next chapter, works on reducing the communication cost of the partition at each finer level, usually while preserving balance.

Chapter 8

Definition of a Local Optimization Refinement Algorithm Reducing the Communication Cost While Preserving Balance

Contents

In this chapter, we describe and justify our implementation of the expansion phase of the multilevel algorithm. The expansion or uncoarsening phase was defined in Algorithm 26 on page 106. In this formulation, the expansion phase is a succession of calls to the `Prolong` and `Refine` functions, that are invoked for each level created in the coarsening phase.

- The `Prolong` function assigns a part to each cell/vertex of the coarsened mesh/graph. We will use the common scheme formulated previously in Definition 30 on page 109, which specifies that a vertex is assigned the same part as its corresponding vertex in the coarsened graph. Note that with this scheme, the imbalance and the communication cost of the finer graph after prolongation are the same as that of the coarsened graph.
- The `Refine` function aims at reducing the communication cost of the prolonged partition. In our case, the `Refine` function is the classical

Fiduccia-Mattheyses algorithm FM, which, when given a solution, will also return a solution.

Section 8.1 will first justify the use of the FM algorithm. Then, Section 8.2 will formulate a general version of FM, that highlights parts that we will develop in the next sections. Indeed, these parts vary from one partitioning tool to another, or need more implementation details, so we will explain our choices in greater detail. Thus, Section 8.3 will focus on allowed moves and tolerance relaxation, Section 8.4 on the selection between moves of same gain, and Section 8.5, on the stop conditions of FM.

Along this chapter, we will also compare our implementation to those of the multi-criteria partitioning tools MeTiS and PaToH, and with the mono-criterion partitioning tool Scotch in which our algorithm will eventually be implemented.

8.1 Refinement Algorithm Overview

The usual algorithms to refine the communication cost during the expansion phase are FM and KL (the Kernighan-Lin algorithm, defined in Section 4.4.1). The main difference between those two algorithms is that FM moves one vertex at a time, while KL swaps two vertices at each step. Therefore, as analyzed in Section 5.3.1, given an initial solution, the reachable solutions of each algorithm can be very different.

In particular, as KL performs exchanges, it cannot reach partitions in which the number of vertices per part is different from that of the initial partition, which can remove quite a lot of solutions when the vertices do not have a unit weight, or when the imbalance tolerance is large.

Implementation – PaToH-3.2

PaToH relies on a "KLFM" refinement algorithm, which seems to be a call to a special version of KL followed by a call to a version of FM quite similar to that of MeTiS.

Another benefit of FM over KL is that it is faster. This is by the way how Fiduccia and Mattheyses [1982] described their algorithm: an improvement in terms of run time of KL (whereas they actually define a new algorithm).

Finally, we will assume that the solution space when using FM is connected. This assumption is inspired by the analysis performed in Section 5.3, which stated a bound on the vertex weights that guaranteed the connection of the solution space for FM-like algorithms in the mono-criterion case. We claim that, in practice, it is possible to reach the optimal solution from any initial solution. Moreover, such optimal solution should be reachable in a few moves. More precisely, in the order of n moves, where n is the number of vertices in the mesh.

Besides, we studied some variations of FM which did not allow moves of negative gain or moves of non-positive gain. These seemed to perform significantly worse, which would mean that hill-climbing, as well as the ability to cross "plateaux" (in a fitness landscape view, a plateau is when we perform a move of null gain) are necessary to get to the optimal solution (getting there in a straight descent is unlikely). Therefore, FM's major characteristic is that it is able to escape from local minima by allowing moves of negative gains.

We also briefly studied other variants of FM, concerning the move selection. A first variant, instead of selecting a move of best gain, selected at each step any move of non-negative gain. A second variant selected at each step a move of worst non-negative gain.

The results seemed to indicate that the variant performing any move of non-negative gain reached varied solutions (and usually, the best solutions reached were as good as the ones of the classic FM algorithm, if not better in some cases). The variant selecting the move of worst gain globally got worse results. Nevertheless, these experiments were performed out of the multilevel framework, which is likely to change the results found.

To conclude, the FM algorithm is a classic and fast refinement algorithm that enables to avoid some local minima using a hill-climbing process. The next sections will describe more precisely our implementation of FM, in order to avoid any ambiguity.

8.2 General Version of FM

We will first give a global description of FM, and show that some of its aspects are ambiguous. Therefore, as in Chapter 6 that described the coarsening phase, we will justify our algorithmic choices for these parts, and try to compare them with those of the partitioning tools MeTiS, PaToH and Scotch. Algorithm 44 gives a more general version of FM than the classic one given in Section 4.4.2. This version comprises additional functions, that are highlighted.

The first added function is the Allowed function, which, given a candidate move, returns if it is allowed. For example, the classic FM algorithm forbids to perform moves that unbalance the partition more than the prescribed tolerance. This will be discussed in Section 8.3.

Then, the BreakTies function defines, if there are several candidate moves of same gain, which move should be performed. This aspect is not precised by Fiduccia and Mattheyses [1982], and will be analyzed in Section 8.4.

Finally, the StopInner and StopOuter functions define, respectively, when a pass must be stopped, and when the algorithm must stop. Some improvements have been proposed by Karypis and Kumar [1998a] in order to speedup the algorithm, and the various partitioning tools all implement slightly different versions for these conditions. Section 8.5 will focus on the conditions that we

Algorithm 44 Generalized Version of the Fiduccia-Mattheyses Algorithm

```
1:  function FM(M, k, W, t, f, Π^ini)
    Require: imb(Π^ini) ≤ t
             gains: structure allowing constant time access to the moves of
             best gain
2:      Π^best, i_pass ← Π^ini, 0
3:      repeat                                                  # Perform a pass
4:          Π ← Π^best          # Recover partitions of smallest communication cost found
5:          f_old, i_pass ← f(Π), i_pass + 1
6:          i_neg ← 0                                  # Counter for StopInner function
7:          gains ← GainTableInit(M, f, Π)
8:          repeat                                                  # Perform a move
9:              g ← gains.getMaxGain()                  # Start from the greatest gain
10:             moves ← [ ]
11:             while moves = [ ] and g ≥ gains.getMinGain() do
12:                 moves ← gains.getMoves(g)                       # Moves of gain g
13:                 for move ∈ moves do
14:                     if Locked(move) or not Allowed(move, W, t) then
15:                         moves.remove(move)
16:                     end if
17:                 end for
18:                 g ← g − 1                          # Try with moves of smaller gain
19:             end while
20:             if moves = [ ] then
21:                 break                              # Remaining moves are forbidden
22:             end if
23:             i, p ← BreakTies(moves)          # Select one among the allowed ones
24:             Π[i] ← p
25:             Lock(i)
26:             if f(Π) ≤ f(Π^best) then
27:                 Π^best ← Π
28:             end if
29:             GainTableUpdate(gains, i, p)
30:             if gains[(i, p)] ≤ 0 then
31:                 i_neg ← i_neg + 1
32:             else
33:                 i_neg ← 0
34:             end if
35:         until StopInner(i_neg)                          # Pass stop condition
36:     until StopOuter(f_old, f(Π^best), i_pass)           # Algorithm stop condition
37:     return Π^best
38: end function
```

chose.

8.3 Restrictions to Preserve Balance

In this section, we focus on the `Allowed` function, which is one of the major source of differences between partitioning tools. Indeed, for MeTiS and Scotch, the imbalance tolerance is relaxed at coarser levels. This means that partitions that are not solutions are accepted at coarser levels, which will improve the connection of the solution space.

We chose not to relax the imbalance tolerance, for several reasons:

- the main reason is that, given a solution returned by the initial partitioning algorithm, the final partition returned may not be a solution, and we observed that some partitioning tools often do not return a solution, as we will see in Section 9.2;
- during the uncoarsening phase, the refinement algorithm will have two objectives: to reduce the imbalance under a threshold, and to reduce the communication cost as much as possible. Having to deal with several objectives at the same time can lead to hinder one when trying to improve the other;
- we assume that there is no need for this relaxation. Indeed, thanks to our study on the connection of the solution space in Section 5.3, we assume that, in practice, the solution space for `FM`-like algorithms is connected.

Algorithm 45 implements existing policies for move restrictions. We will rely on the `AllowedBalance` function, which corresponds to the classic `FM` algorithm.

The different functions are:

- `AllowedBalanced`: the basic restriction. It prevents one from making moves that imbalance the partition more than the prescribed tolerance.
- `AllowedRebalance1`: begins by checking if the current partition respects the balance constraints. If not, it selects only the moves that improve the imbalance; otherwise, it relies on the basic `AllowedBalanced` policy.
- `AllowedRebalance2`: quite similar to the `AllowedRebalance1` function, because it also checks if the current partition respects the balance constraints. However, when the constraints are not respected, this function selects only the vertices that belong to the heaviest part for the most imbalanced criterion. As stated in the previous chapter in Definition 33, the most imbalanced criterion c_{max} is a criterion such that $imb_{c_{max}}(\Pi) = imb(\Pi)$.
- `AllowedBoundary`: allows only to move vertices that are in the boundary of the partition. The boundary of a partition, as formulated previously in Definition 9 on page 17, is the set of the vertices whose part differs from that of at least one of their neighbors. This function is based on

Algorithm 45 Restricting the Possible Moves to Regulate the Imbalance

```
1: function AllowedBalanced(i, p, t, Π)
2:     Π' ← Π
3:     Π'[i] ← p
4:     return imb(Π') ≤ t
5: end function

6: function AllowedRebalance1(i, p, t, Π)
7:     if imb(Π) ≤ t then
8:         return AllowedBalanced(i, p, t, Π)
9:     else
10:        return AllowedBalanced(i, p, imb(Π), Π)
11:    end if
12: end function

13: function AllowedRebalance2(i, p, t, Π)
14:    if imb(Π) ≤ t then
15:        return AllowedBalanced(i, p, t, Π)
16:    else
17:        p ← Π[i]
18:        return imb(Π_p) = imb(Π)      # v_i belongs to the most imbalanced part
19:    end if
20: end function

21: function AllowedBoundary(i, p, t, Π)
    Require: M the mesh
             The Boundary() of a partition as in Definition 9 of Section 1.2
22:    return M[i] ∈ Boundary(Π)
23: end function

24: function AllowedAdaptLevel(i, p, t, Π, AllowedFct)
    Require: level: the current level number
             AdaptTolerance: function returning a new tolerance depending
             on the current level
25:    t_level ← AdaptTolerance(level)
26:    return AllowedFct(i, p, t_level, Π)
27: end function
```

the assumption that a partition whose parts are connected will have a smaller communication cost.

- `AllowedAdaptLevel`: relies on the assumption that the tolerance should be adapted to each level. Usually, this means that the tolerance is more relaxed for coarser graphs. It is usually used in conjunction with one of the two `AllowedRebalance` functions. Indeed, relaxing the tolerance means that the prolonged partition may not respect the balance constraints, and that some improvement on the balance is needed.

Implementation – Scotch-6.0.4

The Scotch implementation is formulated in Algorithm 46: the tolerance is relaxed depending on the current level, and when the partition imbalance is greater than the relaxed tolerance, moves that increase the imbalance are forbidden.

Algorithm 46 – Scotch Restrict Policy

1: **function** `AdaptToleranceScotch`(t, $level$)
2: **return** $t \times (1.05)^{level}$
3: **end function**

4: **function** `AllowedScotch`(i, p, t, Π)
5: **return** `AllowedAdaptLevel`$(i, p, t, \Pi,$ `AllowedRebalance1`$)$
6: **end function**

`Source:` function `BgraphBipartMlCoarsen` in file `bgraph_bipart_ml`.

Implementation – MeTiS-5.1.0

The MeTiS implementation is formulated in Algorithm 47: moving vertices that do not belong to the boundary is forbidden, and the tolerance is relaxed depending on the current level: when the partition imbalance is greater than the relaxed tolerance, only vertices from the heaviest part for the most imbalanced criterion can be moved.

Algorithm 47 – MeTiS Restriction Policy

1: **function** `AdaptToleranceMetis`(t, $level$)
Require: n_{level} the number of vertices in the coarsened graph
2: **return** $t + \frac{0.5}{\max(20,\ n_{level})}$
3: **end function**

4: **function** `AllowedMetis`(i, p, t, Π)
5: **return** $\Big($`AllowedBoundary`(i, p, t, Π)
and `AllowedAdaptLevel`$(i, p, t, \Pi,$ `AllowedRebalance2`$)\Big)$
6: **end function**

`Source:` variable `ffactor` (for "fudge factor") in file `fm.c`.

Moreover, MeTiS uses a rebalancing algorithm in combination with `FM`. In short, this rebalancing algorithm tries to move vertices from the heaviest part for the most imbalanced criterion, until the partition imbalance becomes smaller than the tolerance, or until the number of moves exceeds the number of vertices. The vertices of best gain (for the communication cost) are considered first, and a move is performed:

- if it reduces the imbalance;
- or if it leads to the same imbalance but reduces the cut;
- or if it leads to the same imbalance and the same cut, but the squares of imbalances for each criterion decrease.

`Source:` function `McGeneral2WayBalance` in file `balance.c`.

In this section, we have detailed the `Allowed` function, which differs greatly between partitioning tools. Moreover, note that Scotch and MeTiS relax the tolerance when the graph is coarsened, in order to enlarge the search space. However, they need some rebalancing mechanisms in order to ensure that the returned partition will respect the balance constraints.

8.4 Tie-breaking Between Moves of Same Gain

`FM` specifies that the move of highest gain (commonly called the move of *best* gain) must be performed, even if this gain is negative. However, there can be several moves of best gain, or some moves can be forbidden, for example when some vertices are locked. In this case, we will select the first vertex according to the mesh order, which corresponds to the `BreakTiesFirst` function implemented in Algorithm 48. Indeed, we did not investigate other policies, so we chose to rely on a simple strategy. However, investigations on which policy allows for the best results should be performed. In this section, we will describe the policies of other tools.

Implementation – MeTiS-5.1.0

MeTiS uses the `BreakTiesFirst` function.

Algorithm 48 also defines the `BreakTiesImbalance` function, which returns one candidate that leads to the minimal imbalance.

Implementation – Scotch-6.0.4

Scotch uses the `BreakTiesImbalance` function, which allows it to rebalance the partition if it is not a solution (remember that Scotch relaxes the tolerance at coarser levels), or to try to limit any increase in the imbalance

Algorithm 48 Breaking Ties Between Several Moves of Same Gain

Require: `length`$(candidates) \geq 1$

```
1: function BreakTiesFirst(candidates, Π)
2:     return candidates[1]
3: end function

4: function BreakTiesImbalance(candidates, Π)
5:     imb^best ← 1
6:     for (i, p) ∈ candidates do
7:         Π' ← Π
8:         Π'[i] = p
9:         if imb(Π') ≤ imb^best then
10:            imb^best ← imb(Π')
11:            best_move ← (i, p)
12:        end if
13:    end for
14:    return best_move
15: end function
```

when the current partition is a solution. Therefore, Scotch handles the multi-objective problem raised by the relaxation of the tolerance by prioritizing the minimization of the communication cost, but by always trying to reduce the imbalance as a second objective.

8.5 Stopping the Search

This section defines the `StopInner` and `StopOuter` functions. The `StopInner` function specifies when a pass must be stopped, while the `StopOuter` function specifies if a new pass shall be performed after the current one completes.

In this document, we will not investigate on these functions. Fiduccia and Mattheyses defines the `StopInner` function as to stop when all moves of unlocked vertices are forbidden. Nevertheless, in order to speedup the multilevel algorithm, most partitioning tools stop after a certain number n_{neg} of moves of negative gains have been performed in a row.

Therefore, Algorithm 49 formulates our condition to stop performing moves within the same pass. It relies on the external variable i_{neg}, which is updated during the pass in the `FM` Algorithm 44. It counts the number of moves of negative gain performed in a row, and is reset to 0 at the beginning of a new pass, or when a move of non-negative gain is performed. A pass is stopped when

the i_{neg} variable exceeds the n_{neg} parameter. In our case, we set $n_{neg} = 120$, which is the same value as Scotch. Nevertheless, the values for n_{neg} for different partitioning tools are close, as displayed in Table 8.5.1.

Algorithm 49 Condition to Perform a New Move

1: **function** `StopInner`(i_{neg})
   Require: n_{neg}: maximum number of moves of negative gains that can be performed in a row
2: **return** $i_{moves} = n_{moves}$ or $i_{neg} = n_{neg}$
3: **end function**

Algorithm 50 formulates the condition to stop performing passes. It relies on the external variables f_{old}, f_{new} and i_{pass}, that are updated after each pass in the `FM` Algorithm 44.

i_{pass} counts the number of passes performed, while f_{old} and f_{new} are the communication cost of the partition of, respectively, the last pass and the current pass. The algorithm stops if the number of passes exceeds the n_{pass} parameter, or if the current pass did not improve the communication cost of the partition over the last pass, that is, when $f_{new} = f_{old}$. Note that $f_{new} > f_{old}$ is not possible, because $f_{new} = f(\Pi^{best})$ so f_{new} is initialized to f_{old}, and can only decrease.

Algorithm 50 Condition to Perform a New Pass After that One Finishes

1: **function** `StopOuter`(f_{old}, f_{new}, i_{pass})
   Require: n_{pass} the maximum number of pass (can be infinite)
2: **return** $f_{new} = f_{old}$ or $i_{pass} = n_{pass}$
3: **end function**

In our case, we chose the same value as Fiduccia and Mattheyses and Scotch, which is the default value: $n_{pass} = \infty$. Nevertheless, as displayed in Table 8.5.1, MeTiS sets $n_{pass} = 10$, which means that the improvements of any other pass after having already performed 10 passes are negligible, and only increase the run time of the algorithm (though we are not aware of any work reporting this yet). Therefore, investigations may be performed on this assumption as well.

Implementation – Scotch-6.0.4, MeTiS-5.1.0, PaToH-3.2

	Scotch	MeTiS	PaToH
n_{neg}	120	$\max\left(25, \min(150, \frac{n_{level}}{100})\right)$	$\max(50, \frac{n_{level}}{1000})$
n_{pass}	∞	10	∞

Table 8.5.1 – Scotch-6.0.4, MeTiS-5.1.0 and PaToH-3.2 default parameters for the `StopInner` and `StopOuter` functions of the refinement phase.

Conclusion on the Refinement Phase

This chapter has described our implementation of the refinement phase. Even though we did not propose any new algorithm, we did justify our choices and compare them with the implementations of existing tools, in particular MeTiS and Scotch. The most noticeable difference is that, unlike them, we do not relax the imbalance tolerance at coarser levels. This choice is motivated by our study of the connection of the solution space: we assume that, in practice, the solution space is connected, even for the multi-criteria mesh partitioning problem.

Not relaxing the imbalance tolerance allows us to define a simple version of the `FM` algorithm. The key features follow classic implementations: we do not allow moves that imbalance the partition more than the tolerance, we stop a pass after 120 moves of negative gains, and we stop the algorithm when the last pass did not improve the communication cost. Note that other partitioning tools use more complicated refinement algorithms, because they have to focus on several objectives at the same time.

Conclusion on our Approach

In this part, we have studied the three phases of the multilevel algorithm. However, before going into the details of each phase, Chapter 5 has studied the size and the connection of the solution space for `FM`-like algorithms. In the mono-criterion case, we have found some bounds on the weights of the vertices that guarantee the existence of a solution and the connection of the solution space. We claim that, in practice, the solution space of a mesh partitioning problem is huge and connected, for algorithms that move one vertex at a time.

Therefore, unlike most partitioning tools, which often return partitions that are not solutions because they prioritize the communication cost over the partition imbalance, we designed our multilevel algorithm to first aim at returning a balanced partition, and then minimize the communication cost. To this extent, we first proposed in Chapter 6 several coarsening schemes and analyzed them, and we will compare their performance in the next part. Then, in Chapter 7, we defined two vector-of-numbers initial partitioning algorithms that focus on minimizing the imbalance of the initial partition. Finally, in Chapter 8, we described our refinement algorithm, which is simple because it focuses plainly on minimizing the communication cost. In the next part, we will examine the consequences of using different coarsening schemes and different initial partitioning algorithms.

Part III

Comparison of the Algorithms

Chapter 9

Experimental Environment

Definition of the Instance Space, Comparison Method and Software Implementation

Contents

The previous part introduced in detail algorithms based on the multilevel framework. These algorithms differ in the matching algorithms used for coarsening (even if each method is based on the Heavy-Edge Matching scheme), rely on different initial partitioning algorithms, but use a common refinement scheme. This part aims at analyzing experimentally the effects of each policy. However, to do so, we need to define the instance space.

The instance space is the set of instances that we will use to compare algorithms. As described by Blackburn *et al.* [2012], the analysis that we will perform will be valid at least for the instances that belong to the instance space. Section 9.1 defines one industrial test case, and present other meshes for which

we generate fictitious weight distributions. The method used to generate such multi-criteria weight distributions is based on the way numerical simulations occur at CEA (in particular, particle-in-cell simulations).

Besides, in practice, a heuristic returns various partitions of a given instance, as we will see in Section 9.2. Indeed, heuristics often rely on the mesh numbering, or on random mechanisms. For example, Scotch and MeTiS use random seeds for their greedy graph growing initial partitioning algorithm. Therefore, with a different numbering of the mesh, or a change in the random seed generator, the returned partition differ greatly between runs.

Thus, comparing heuristics is not obvious. This is why Section 9.3 will introduce a method helping to compare and analyze the characteristics of each heuristic. Finally, Section 9.4 will describe the flexible software, named Crack, that we implemented in order to compare the various heuristics.

9.1 Definition of the Instance Space

This section defines the instances that we will use to compare the heuristics. We call instance space the set of our instances. Note that, even if we hope that our results may generalize to all possible instances, in fact, as noticed by Blackburn *et al.* [2012], they may be valid only on this instance space. In particular, variations on the weight distributions can lead to other conclusions on the properties of each heuristic.

9.1.1 *LMJ*, an Industrial Test Case

We will use a $3D$ mesh named *LMJ*, displayed in Figure 9.1.1 It models a capsule that will be heated by a laser, and has 484 356 vertices. This particular mesh is used for multiphysics simulations for experiments on plasma physics, carried out in the Laser Mégajoule, a research device situated in Bordeaux, France (for more information, see CEA [2015]).

The capsule is composed of different structures, among which gold and foam. Particles will move into the capsule, changing the structures. The first weight of a cell depends on the number of particles in the cell. The second weight depends on the structures contained in a cell, and cells containing gold are particularly heavier. The third weight is a unitary weight, so that a partition must attribute roughly the same number of cells to each part. Finally, edge weights correspond to the volume of data that must be exchanged if two neighboring cells are not in the same part.

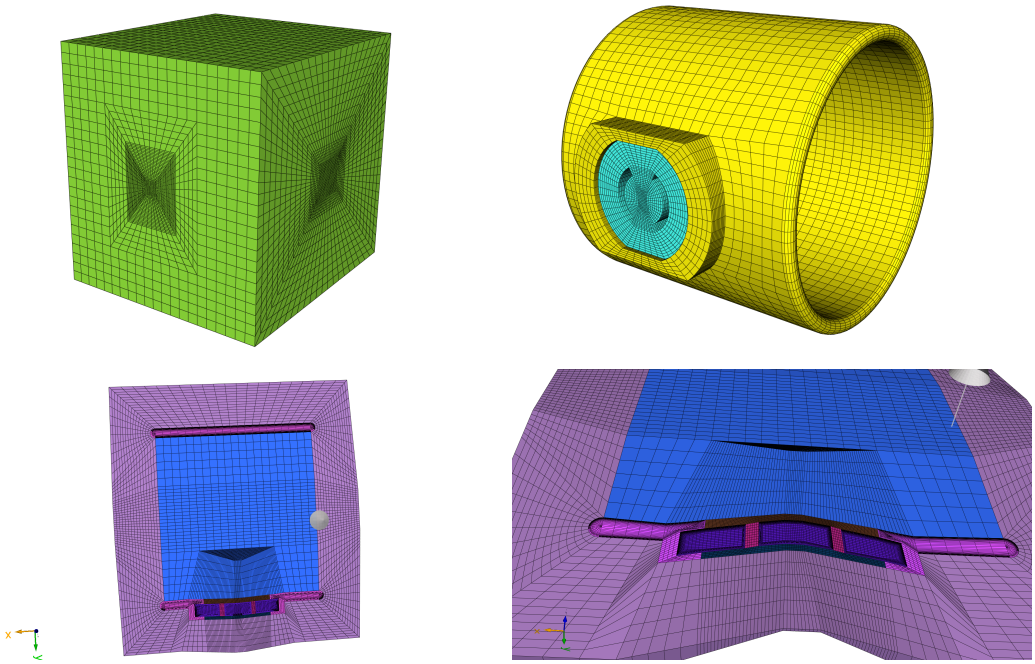


Figure 9.1.1 – The mesh *LMJ*, which is our industrial test case. Top left: the whole domain, which is in a cube. Top right: the capsule that is inside the cube. Bottom left: upper view of the interior of the capsule. The interior is colored in blue, cells containing gold are in pink, cells containing foam are in purple. Bottom right: side view of the capsule. Red cells are the slits in the foam.

Table 9.1.1 – The mesh used to perform the tests

Mesh				t	k
Name	Geometry	$\|V\|$	$\|E\|$		
Mushroom	$2D$	22 800	45 253	$\in \{0.2\%, 1\%, 5\%\}$	$k = 2$
Onera	$3D$	85 567	166 817		
Wave	$3D$	156 317	1 059 331		
Linkrodsok1	$3D$	174 218	322 771		
Shock	$2D$	1 196 352	1 793 115		

9.1.2 Generation of Weight Distributions Corresponding to Particle-In-Cell Simulations

Table 9.1.1 sums up some characteristics of the other instances that we will use. There are 5 meshes of various sizes and geometries. The experiments will be performed for 3 different imbalance tolerances: a very tight tolerance of 0.2%, a tight one of 1%, and a classic laxer one of 5%. Figures 9.1.2, 9.1.3 and 9.1.4 respectively display the meshes *Mushroom*, *Linkrodsok1* and *Shock*.

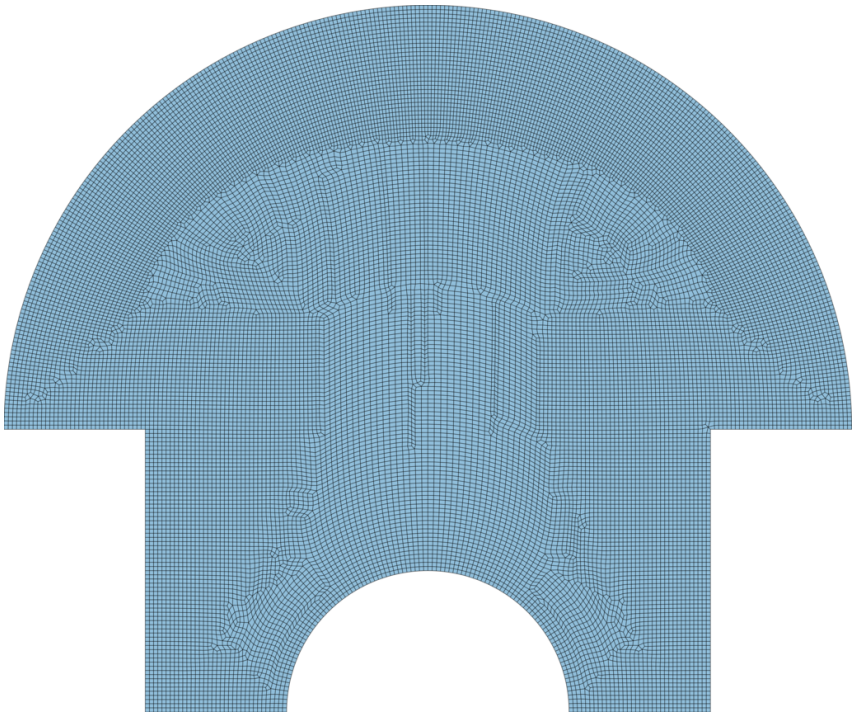


Figure 9.1.2 – The smallest mesh, *Mushroom*, of 22 800 cells

Unfortunately, there is no public repository of weight distributions of meshes used in multiphysics simulations, and in the numerical simulations that are

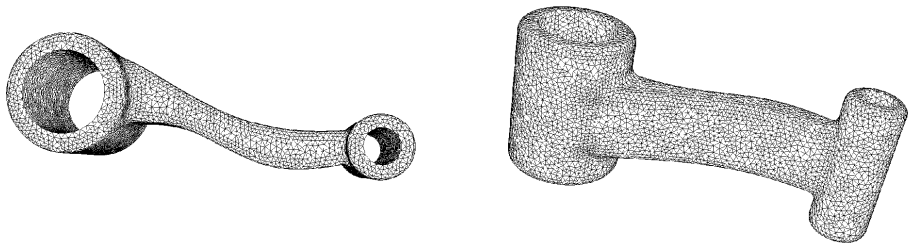


Figure 9.1.3 – Two different views of the $3D$ mesh *Linkrodsok1*, of 174 218 cells

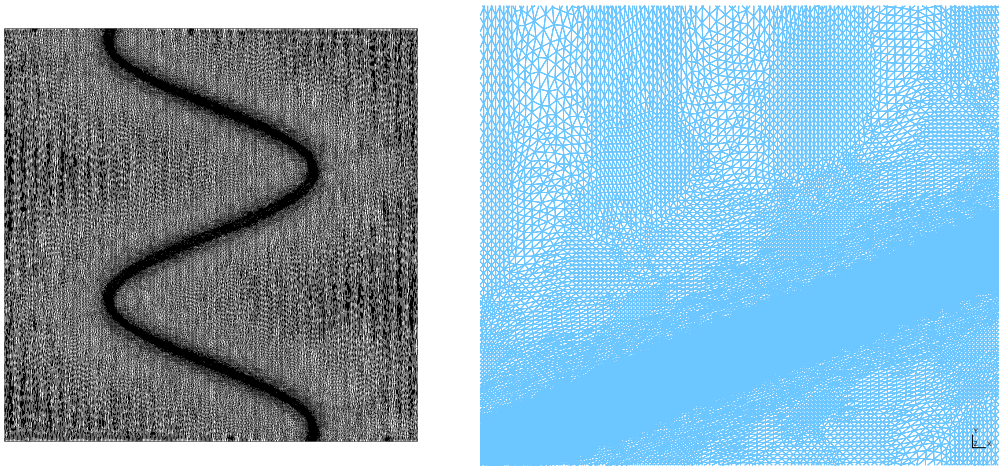


Figure 9.1.4 – The biggest mesh, *Shock*, of 1 196 352 cells. On the left: the whole mesh. On the right: a zoom on the center of the picture.

run at CEA, the computation weights are tightly integrated to the code, which makes them difficult to extract. Therefore, we introduce a model that enables one to simulate multi-criteria weight distributions of some multiphysics simulations.

Whereas these instances are not industrial test cases, we will define a method to generate weight distributions similar to that of particle-in-cell simulations. In such simulations, computations are linked to the number of particles located in a given cell. Usually, particles are scattered from one or several sources (for example, light sources generate photons), and the number of particles decreases linearly with the distance to the source.

Besides, computation linked to one cell usually increases along with the square of the number of particles in this cell, while the memory required to store a cell is linear with the number of particles in this cell.

Algorithm 51 formulates the two functions that will be used to generate a weight distribution for one criterion. The first one, named `VWgtsUnit`, associates a unit weight with each cell. The second one, named `VWgtsMountains`, assigns heavier weights to the cells that are close to some vertices called the "peaks" (which model a source of particles).

The `VWgtsMountains` function needs some additional parameters, that are: *peaks*, the list of the cells that will serve as peaks; *radius*, the list of the radius for every peak; $g : \mathbb{R} \to \mathbb{R}_+$, a function that, given the distance of a cell to a peak, returns a weight which corresponds to the amount of computation that the cell gets from being at this distance from the peak; and finally, some weights that act as bounds.

Using these parameters, for each peak, the cells at a distance d smaller than the radius of this peak (so, in the mountain corresponding to this peak) increase their weight for the current criterion of $g(d)$. Cells that are not included in any mountain for the current criterion are assigned for this criterion a random weight between $w_{valley_{min}}$ and $w_{valley_{max}}$. Note that the function `FindNewPeak`, which is not implemented here, simply selects a cell which is not already in a mountain for this criterion.

Table 9.1.2 sums-up the parameters used to generate the computation weights associated with the cells of the meshes. For the first and second criteria, cells are given computation weights using the `VWgtsMountains` function. In each case, 3 random peaks (peaks cannot be part of another mountain) are generated, of radius chosen randomly between one eighth and one fourth of the length of the mesh.

The function used to compute the weight of a cell at a distance d from the peak is, given a radius r, $g : d \mapsto \max\left(w_{valley_{min}}, int\left((w_{peak}(1 - \frac{d}{r}))^2\right)\right)$. Moreover, we chose $w_{peak} = 50$, and $(w_{valley_{min}}, w_{valley_{max}}) = (10, 11)$. The weight of the peak is thus $g(w_{peak}) = 50^2 = 2500$, and the weights of cells outside the mountains are either 10 or 11. For the third criterion, all cells are

Algorithm 51 Generation of the Vertex Weights

function `VWgtsUnit`(M)
  $n \leftarrow$ `length`(M)
  return $[1 \mathinner{..} n \mathinner{..} 1]$
end function

function `VWgtsMountains`(M)
Require: $peaks \in M^{npeaks}$: the cells considered as peaks
  $radius \in \mathbb{R}^{npeaks}$: the radius of each peak
  $g : \mathbb{R} \to \mathbb{R}_+$: given a distance, returns a weight
  $w_{valley_{min}}, w_{valley_{max}}, w_{peak} \in (\mathbb{R}+)^3$: bounds for vertex weights
  $n \leftarrow$ `length`(M)
  $W_c \leftarrow [0 \mathinner{..} n \mathinner{..} 0]$
  for $peak, r \in (peaks, radius)$ **do** # Weights for the cells on the mountains
    for $i \leftarrow 1, n$ **do**
      $d \leftarrow$ `dist`$(M[i], M[peak])$ # Euclidean distance
      if $d \leq r$ **then**
        $W_c[i] \leftarrow W_c[i] + g(d)$
      end if
    end for
  end for
  for $i \leftarrow 1, n$ **do** # Weights for the cells outside from the mountains
    if $W_c[i] = 0$ **then**
      $W_c \leftarrow$ `random`$(w_{valley_{min}}, w_{valley_{max}})$
    end if
  end for
  return W_c
end function

Table 9.1.2 – Generating the computation weight function W

`VWgtsFct`	$W_{c_1}, W_{c_2} \leftarrow$ `Mountains`				$W_{c_3} \leftarrow$ `Unit`
parameters	$npeaks$	$g(d)$	$w_{valley_{min/max}}$	w_{peak}	$\varnothing$
	3	$\propto int((1-\frac{d}{r})^2)$	10/11	50	

Table 9.1.3 – Model used for our tests

Representation	f	$W_{com}((v_i, v_j))$
Graph	$edgecut$	$\sum_{v \in \{v_i, v_j\}} int\left(\sum_{c \in \{1,2\}} \sqrt{W(v)[c]}\right)$

given a unit weight.

Therefore, the two first criteria correspond to a simulation modeling two physical models relying on the particle-in-cell model, while the third criterion corresponds to the fact that we want each unit to receive roughly the same number of cells.

Then, Table 9.1.2 defines the model that we will use to partition the meshes: we will rely on the graph model introduced previously in Section 2.4. Each edge is given a weight that depends on the weights of its ends for the first and second criteria. Indeed, in particle-in-cell simulations, the communication cost is often linked to the number of particles present in a cell, and as the weight of a cell for a criterion is the square of the number of particles on it, the number of particles for this criterion is the square root of the weight of a cell.

For every mesh, we will generate 3 weight distributions corresponding to the description given in Table 9.1.2. Therefore, this makes 5 meshes $\times$ 3 weights distributions $\times$ 3 tolerance thresholds $\times$ 1 number of parts $=$ 45 instances. In the next section, we will show that for every instance, one run per heuristic is not sufficient to compare them. Indeed, given a heuristic, there is a high discrepancy on the communication cost of the returned partitions.

9.2 Highlighting the Discrepancy on the Communication Cost of the Returned Partitions

This section examines first the proportion of solutions returned by existing partitioning tools, and then analyze the communication cost of these solutions for each tool. The aim is to show that comparing these algorithms will require to run every algorithm many times, and will demonstrate that comparing

Table 9.2.1 – State-of-the-art partitioning tool options

Tool	version	options	remarks
MeTiS	5.1.0	random seed	$f = edgecut$
PaToH	3.2	random seed	$f = cut_{\lambda-1}$
		`PQ=Q` (enhance quality over speed)	
		`BO=C` (balance cell-weights)	
Scotch	6.0.4	random seed	$f = edgecut$
		mono-criterion version:	
		$W(m) = W(m)[1]$	

partitioning algorithms is not obvious.

The partitioning tools that we will use in this section are MeTiS-5.1.0 and PaToH-3.2, which handle multi-criteria partitioning, and Scotch-6.0.4, which does not handle multi-criteria yet, but in which we will implement our algorithms. The characteristics of each tool are summarized in Table 9.2.1.

In particular, note that PaToH, which relies on the hypergraph model described in Section 2.3, uses as communication cost function the $cut_{\lambda-1}$ of a partition, which slightly differs from the *edgecut* of a partition. Besides, as Scotch does not handle multi-criteria graph partitioning, we will use it as a mono-criterion graph partitioning tool in this section. Therefore, Scotch will balance only the first criterion in each weight distribution.

Each partitioning tool takes as argument a random seed. In order to test the robustness of every tool, we will run it 100 times on every instance with a different seed for every run.

Proportion of Solutions Returned

Table 9.2.2 reports the percentage of solutions returned by each partitioning tool.

Scotch returns in each case 100% of solutions. However, we cannot compare its results to those of MeTiS and PaToH, since they do not solve the same problem: for Scotch, which is here a mono-criterion partitioning tool, a solution is a partition balanced for the first criterion.

PaToH fails to return solutions for two instances. In particular, it does not manage to return a single solution for the *Shock* mesh when the tolerance is very tight, though some solutions exist since MeTiS manages to find some.

MeTiS returns 100% of solutions for only four instances. For the eleven others, the proportion is very low: usually below 50%, and for five instances 20% or less. An interesting fact is that, in many cases, MeTiS returns in many cases more solutions when the tolerance is tighter.

Therefore, the existing multi-criteria partitioning tools sometimes fail to

Table 9.2.2 – Percentage of partitions respecting the constraints returned by each partitioning tool, for 100 runs of the first weight distribution of each instance

instance mesh	t	Scotch (%)	MeTiS (%)	PaToH (%)
Mushroom	5%	✓	34	✓
	1%	✓	68	✓
	0.2%	✓	58	✓
Onera	5%	✓	✓	✓
	1%	✓	✓	✓
	0.2%	✓	41	✓
Wave	5%	✓	✓	✓
	1%	✓	2	✓
	0.2%	✓	9	✓
Linkrodsok1	5%	✓	✓	✓
	1%	✓	✓	✓
	0.2%	✓	12	✓
Shock	5%	✓	20	✓
	1%	✓	46	37
	0.2%	✓	18	0

return a solution, although one exists. This only could justify the fact that, in order to compare algorithms, one should perform many runs to conclude on the tendency of an algorithm to return a solution when one exists. Nevertheless, we will see in the following that comparing the communication cost of the returned solutions is not straightforward.

Bar Plots

In order to show the distribution of the communication cost of the solutions returned by each algorithm, we will use a bar plot. A bar plot represents the distribution of the communication cost of the partitions returned by some tool, for one instance. The length of a bar at ordinate y is the number of times that the tool returned a partition of communication cost y for this instance.

Since the objective is to minimize the communication cost, lower bars are better. A perfect partitioning tool would only have one bar at the lowest possible ordinate, meaning that it always returns a partition of minimum communication cost.

For MeTiS, which returned many partitions that are not solutions, we also included a bar plot of the imbalance of the partitions returned. Regarding balance, the objective is to return partitions of imbalance lower than the

tolerance. Hence, bars above t represent invalid partitions.

Finally, we recall that this section *does not aim at comparing partitioning tools*, which is not possible in the first place, because:

- Scotch balances only the first criterion of each instance;
- for PaToH, which relies on the hypergraph model, we reported the $cut_{\lambda-1}$ (defined in Section 2.3), which is the same communication cost that it optimizes.

However, a comparison between multi-criteria partitioning algorithms, including MeTiS, PaToH and a beta version of Scotch, will be performed in Section 10.2.

Results Analysis

Figure 9.2.1 displays in each line the bar plots of a partitioning tool, and on each column the bar plots for a given tolerance. The tolerance is relaxed in the left column (5%), tight in the middle column (1%), and very tight in the right column (0.2%). Each plot displays results for the first weight distribution of the *Mushroom* mesh introduced previously in Table 9.1.1, and the plots for the first weight distribution of the other meshes are given in Appendix A.2. In each plot, the mean is displayed with a red triangle pointing down, and the median with a yellow triangle pointing up.

Scotch. The first line displays the results for Scotch-6.0.4, which shows an extended range of communication costs for the returned solutions when the tolerance becomes tighter. When $t = 0.2\%$, Scotch sometimes returns a partition whose communication cost is more than 15 000, which is more than three times the communication cost of the best partitions that it returns. When $t = 1\%$ and $t = 5\%$, there is also a factor of two between the minimum and maximum communication costs obtained. Nevertheless, we observe that the communication costs of the optimal solution for each tolerance are close.

Implementation – Scotch-6.0.4

By default, when Scotch-6.0.4 is called, it runs two independent multilevel algorithms and returns the best partition of the two. For Scotch, between two partitions, the "best" one is, if both are not solution, the one of minimal imbalance, else if both are solutions, the one of minimal communication cost, else the one that is a solution.

Therefore, a thorough study of Scotch algorithm would require making Scotch run the multilevel only once when called. This is what we will do when testing the multi-criteria version of Scotch that we implemented, in Section 10.2.

The first line of Figure 9.2.1 shows that, in this case, the communication cost of the solutions returned by Scotch varies by a factor of two. Besides, it is quite interesting that the smallest communication cost obtained is nearly the

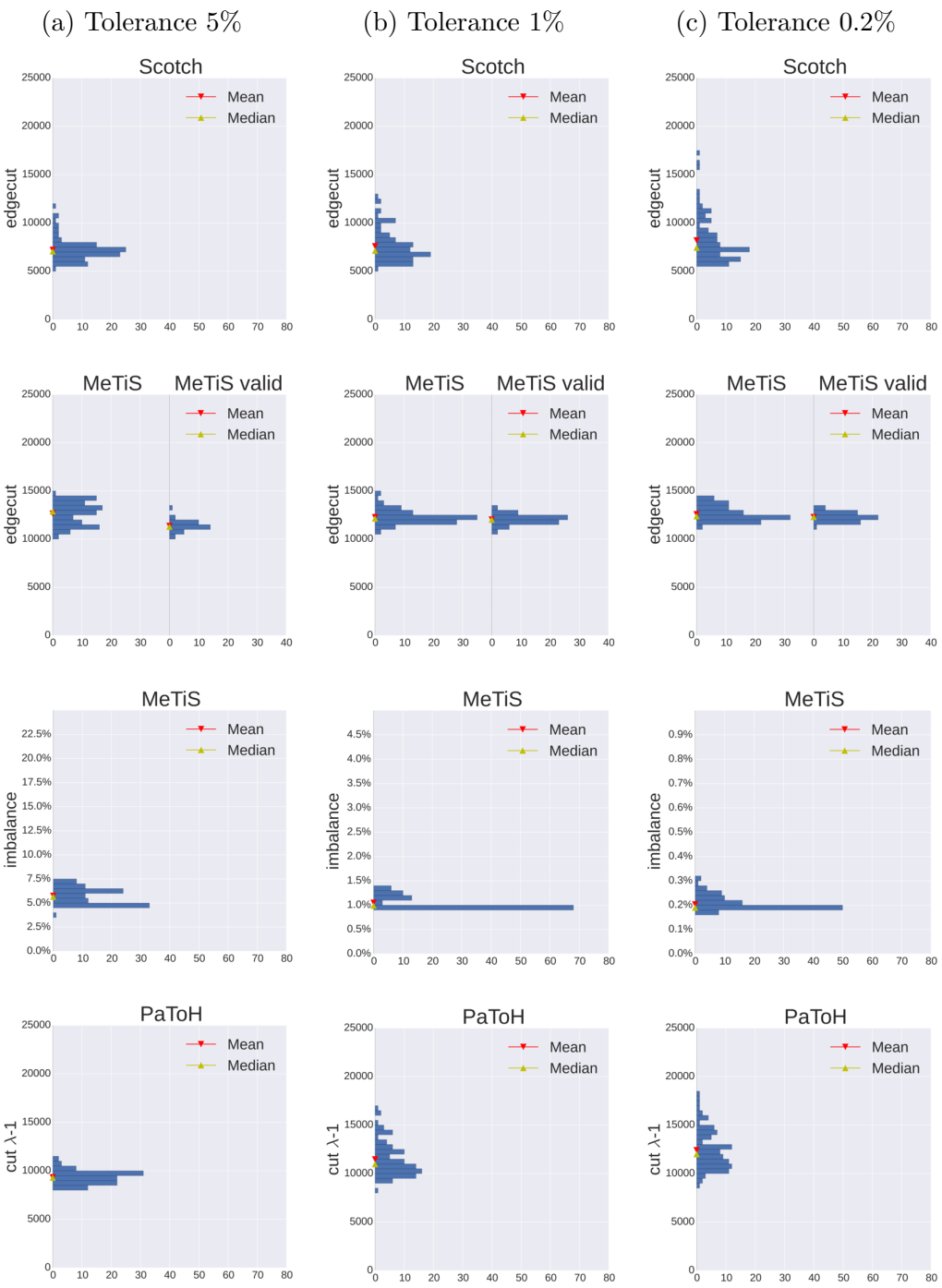


Figure 9.2.1 – Bar plots of the *edgecut*, $cut_{\lambda-1}$ and *imbalance* of the partitions returned by Scotch, MeTiS and PaToH over 100 runs on the mesh *Mushroom*

same even when the tolerance is very tight, but that the highest communication cost obtained increases a lot.

MeTiS Edgecut. The second line displays the bar plots for the communication cost of the partitions returned by MeTiS-5.1.0. In this case, for each tolerance, two distributions are displayed. The distribution on the left considers all the partitions returned by MeTiS, while the distribution on the right displays only the partitions that respect the constraints (hence the name "MeTiS valid").

The proportion of solutions was displayed previously in Table 9.2.2, which shows that for *Mushroom*, MeTiS returned more solutions when the imbalance was tighter. When looking at the communication costs of all partitions returned by MeTiS for this mesh, it also seems that it returns fewer solutions of high communication cost when the tolerance is tighter. This phenomenon is surprising: paradoxically, it is more profitable to run MeTiS with a tighter tolerance to get more solutions, or to get more partitions of smaller communication cost in general.

Implementation – MeTiS-5.1.0

In the multi-criteria case, if n is the original number of vertices, MeTiS coarsens the graph until its number of vertices becomes smaller than $N = \max\left(\frac{n}{20\log_2(k)}, 30 \times k\right)$. Then, it will run 4 to 5 independent multilevel algorithms on this coarsened graph, and take the best of them. Then, it will run its expansion phase normally using this best partition.

`Source:` variables `ctrl->CoarsenTo` and `ctrl->nIparts` set in function `METIS_PartGraphKway`.

Therefore, like Scotch does, MeTiS attempts to return solutions of smaller communication cost by running its algorithm multiple times. However, unlike Scotch, MeTiS only performs several runs on an already coarsened graph. For example, for *Mushroom*, the size of the coarsest graph is $N = \frac{22\,800}{20} = 1\,140$ vertices, and for *Shock*, it is $N = \frac{1\,196\,352}{20} = 59\,817$ vertices.

MeTiS Imbalance. In the third line, we have plotted the distribution of the imbalance of the 100 partitions returned by MeTiS. As one can see, the partitions that do not respect the constraints are not very imbalanced; the imbalance is higher than the tolerance by most one half. This is due to MeTiS' policy to allow a relaxed tolerance at coarser levels, as seen in Section 8.3. Partitions that are not solutions may be caused by the fact that, when reaching the finest level, MeTiS either does not manage to decrease the imbalance enough, or does not want to decrease the imbalance because it would increase the communication cost very much.

PaToH. Finally, the last line displays the distribution of the communication cost for PaToH. For PaToH, the communication cost is $cut_{\lambda-1}$. As for Scotch, the communication cost varies a lot between the returned partitions, and this variation increases when the imbalance is tighter.

Conclusion

From the analysis of the proportion of solutions returned by each software and of the bar plots of the communication cost, we can formulate some qualitative observations:

- the partition returned by a partitioning tool does not always respect the balance constraints;
- there are some variations in the communication cost of the returned solutions, even for mono-criterion partitioning, and up to a factor of 2 here. The variations seem to intensify when the imbalance tolerance becomes tighter;
- for the first weight distribution of *Mushroom*, the minimal communication cost of a partition is very close for each imbalance tolerance.

Therefore, considering that a heuristic sometimes return partitions that are not solutions and due to the variety of communication costs yielded for the same instance, how can we compare two heuristics? This question will be addressed in the next section.

9.3 Definition of the Method Used to Compare Algorithms

In the previous section, we have shown that heuristics return various partitions of the same instance, of various imbalance and communication cost. In this section, we examine several ways to compare heuristics, and we will define which one we will use in this document. Using the defined method, we will be able to analyze the characteristics of a heuristic.

Indeed, there is no uniform metric to compare heuristics. For example, Walshaw *et al.* [2000] run each algorithm once on each instance, before comparing the average on the instances. Deveci *et al.* [2015b] run each algorithm five times on each instance and, for each instance, compares the average on the five runs. LaSalle and Karypis [2016] run each algorithm 25 times on each instance and compare their geometric average.

Section 9.3.1 will start by defining a metric that we already considered implicitly in the previous section, which is the probability for a heuristic to return a solution for a given instance. Then, Section 9.3.2 will show that in order to compare heuristics based on the communication cost of the returned solutions, classic metrics such as the *mean* or the *median* are not forcibly

pertinent. Therefore, Section 9.3.3 will define how to compare heuristics as well as provide a way to find properties of a heuristic.

9.3.1 Probability to Return a Partition Respecting the Constraints

This first metric estimates the probability that a heuristic will return a partition that respects the constraints. Indeed, in the mesh partitioning Problem 2, in order to be able to compare the communication cost of two partitions, they must respect the constraints.

Definition 34 (Probability to Return a Solution)
We will denote by $P(valid)$ the probability that a heuristic returns a partition respecting the balance constraints.

Given a heuristic and an instance, in order to estimate $P(valid)$, we will perform N_{runs} independent runs of this heuristic on this instance. We denote by N_{valid} the number of partitions returned that respect the constraints. Then,

$$P(valid) = \frac{N_{valid}}{N_{runs}} \ . \tag{9.1}$$

Example
Table 9.2.2 showed in the previous section is an estimation of $P(valid)$ for Scotch, MeTiS and PaToH, for the first weight distribution of each mesh and tolerance.

In our experiments, we only run each heuristic 100 times on each instance, so one may wonder if it is legitimate to say that $P(valid)$ is accurate. The purpose of this study is not to compute accurately $P(valid)$, but to give a rough estimate of it. Moreover, we have observed experimentally that $P(valid)$ does not change very much with 200 runs.

$P(valid)$ estimates the probability that a heuristic returns a solution. In the following sections, we will introduce metrics to compare heuristics based on the communication cost of the solutions that they return.

9.3.2 Classic Metrics

A simple way to compare heuristics based on the communication cost of the returned solutions is to use classic statistical indicators. Nevertheless, we will see that they are not sufficient to understand whether some algorithm is better than another, or why.

In this section, we will denote by $\mathbb{S}$ the set of the solutions returned by a heuristic that was run N_{runs} times. Besides, we will consider:

- the *mean* $= \frac{\sum_{\Pi \in \mathbb{S}} f(\Pi)}{|\mathbb{S}|}$ (if $|\mathbb{S}| \geq 1$);

- the *median*, which is the communication cost of the returned solution such that half of the other solutions has a higher communication cost, and the other half has a smaller communication cost;
- the standard deviation, *SD*, which quantifies the dispersion of a set of data values. If we denote by $\mathbb{S}$ the set of solutions returned by the heuristic, then $SD = \sqrt{\frac{\sum_{\Pi \in \mathbb{S}} (f(\Pi) - mean)^2}{|\mathbb{S}| - 1}}$ (provided that $|\mathbb{S}| \geq 2$);
- the minimum and maximum communication costs, $min = \min_{\Pi \in \mathbb{S}} f(\Pi)$ and $max = \max_{\Pi \in \mathbb{S}} f(\Pi)$.

Example

In this example, we will use the classic statistical indicators to compare the communication cost of the solutions returned by MeTiS when it is called with various imbalance tolerances. We define $metis_t$ to refer to MeTiS used with an imbalance tolerance of $t/100$, with $t \in \{5\%, 1\%, 0.2\%\}$. Normally, one could expect that $metis_5$ will get better results than $metis_1$, which itself will get better results than $metis_{0.2}$.

Table 9.3.1 sums up the values for these statistics for the returned solutions of MeTiS, when partitioning the mesh *Linkrodsok1* for the first weight distribution.

Table 9.3.1 – Classic statistics on the communication cost of the solutions returned by MeTiS when partitioning the mesh *Linkrodsok1* for the first weight distribution

Tolerance	*mean*	*SD*	*min*	*median*	*max*
5%	7613	2282	5676	6588	14340
1%	8187	454	6996	8160	9384
0.2%	7479	202	7248	7416	7944

Firstly, since these statistics only take into account the solutions, $metis_{0.2}$ gets the smallest *mean*, standard deviation and *max*. However, as seen in Table 9.2.2, $metis_{0.2}$ only returned 12 solutions out of 100 runs. Therefore, it is difficult to compare its statistics with those of $metis_1$ and $metis_5$ which both returned 100 solutions for this mesh.

Secondly, $metis_5$ gets a smaller *mean*, *min* and *median* than $metis_1$. However, $metis_5$ standard deviation and *max* far exceed those of $metis_1$. Therefore, ordering $metis_1$ and $metis_5$ would mean to prioritize a statistical indicator over the others. If the preferred indicator is the *mean*, as it is usually, then $metis_5$ would be considered better than $metis_1$ for this mesh. However, as shown by the high standard deviation, $metis_5$ may return solutions of very high communication cost, which is not visible using only the *mean*.

As illustrated by this example, comparing heuristics requires us to prioritize

one metric over the others. Moreover, these metrics are difficult to combine with $P(valid)$, the probability to return a partition respecting the constraints. Between a partitioning method that returns more solutions and one that returns solutions with a smallest communication cost on average, how do we choose?

Therefore, in the next section, we will define a method to represent the results. This method will help us compare and analyze heuristics more easily.

9.3.3 Cumulative Plots and Probability to Be x Times as Good as the Optimal Solution Found

In this section, we introduce a representation of the distribution of the communication cost of the solutions returned by a heuristic. The idea is, for some heuristic $\mathbb{A}$, to use a cumulative plot function, that we call *cumul* in this document, associating with each $x \in \mathbb{R}_+$ the number of times that $\mathbb{A}$ returned a solution of communication cost smaller than or equal to x.

As claimed by Dolan and Moré [2002], cumulative plots (that they name performance profiles) help us compare more easily heuristics and understand some of their properties. Indeed, in our case, a cumulative plot conveys all the information on the communication cost distribution of the solutions returned by a heuristic, as well as the proportion of solutions returned. Dolan and Moré used the example of optimization softwares, which search for an optimal solution in a minimum amount of time. This is analogous to our aim, which is to find a solution as close as possible to an optimum, considering that the compared algorithms run in a comparable amount of time because their complexities are equivalent.

The *cumul* function displays the distribution of the communication cost of the solution returned by a heuristic differently than the bar plots introduced previously in Section 9.2. Using the *cumul* function, we compare more easily two heuristics: the curve on the top left is better, as illustrated in the following example.

Example

Figure 9.3.1 plots the *cumul* function for MeTiS, when applied to the mesh *Linkrodsok1* with a tolerance in the set $\{5\%, 1\%, 0.2\%\}$. As previously, we define $metis_t$ to refer to MeTiS used with an imbalance tolerance of $t/100$.

A first observation is that the *cumul* function for MeTiS when the tolerance is 0.2% does not reach 100, because MeTiS returned only 12 partitions of imbalance smaller than the tolerance when the tolerance was set to 0.2%.

A second observation confirms what we mentioned in the previous section: it is not obvious to compare $metis_5$ and $metis_1$. Indeed, the *cumul* functions of $metis_1$ and $metis_5$ intersect, for a normalized edgecut of approximately 1.5.

Nevertheless, the behaviors of $metis_1$ and $metis_5$ may be analyzed easily

using their *cumul* function. Until approximately a normalized edgecut of 1.5, $metis_5$ is on the left of $metis_1$, which means that $metis_5$ returns more solutions of normalized edgecut smaller than 1.5 than $metis_1$. However, after 1.5, we observe the opposite, and the communication cost of the solutions returned by $metis_5$ raises quickly, whereas for $metis_1$, the communication cost is very stable. Therefore, for this instance, despite returning in general solutions of higher communication cost, $metis_1$ seems more reliable than $metis_5$.

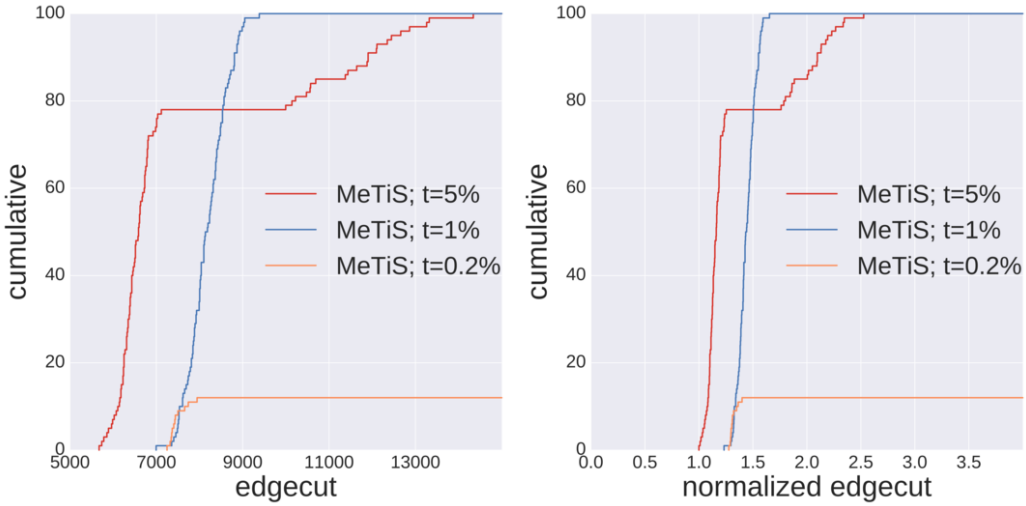


Figure 9.3.1 – *cumul* function for MeTiS on the mesh *Linkrodsok1*, for the first weight distribution. $cumul(x)$ is the number of times that MeTiS returned a partition whose edgecut was smaller than x. Unlike on the left, the *edgecut* on the right plot is normalized with the minimal communication cost found by $metis_5$, $metis_1$ and $metis_{0.2}$, respectively.

Thus, using the *cumul* function, one can qualitatively compare heuristics. One heuristic whose *cumul* function remains on the left of that of another heuristic is clearly better (irrespectively of the partitioning time). Moreover, a curve at the top of the plot indicates that the heuristic returned a large proportion of solutions.

Besides, the *cumul* function helps one to choose the heuristic fitting the best its needs. For example, being able to run a heuristic several times will in most cases reduce greatly the probability to get a partition of very bad quality, in which case a heuristic that returns more solutions of small communication cost (such as $metis_5$) may be preferred. On the opposite, if the heuristic can only be run once, then one may prefer a robust heuristic, that guarantees not to return solutions of high communication cost (like $metis_1$).

In the next chapter, we will therefore rely on the *cumul* function to compare

heuristics, but also to analyze their respective results. Nevertheless, before reporting the results, the next section will introduce the software testbed, named Crack, that we implemented in order to test various algorithms.

9.4 Crack: a Flexible Python Mesh Partitioning Tool

In order to compare the various heuristics that have been described in this document, we have implemented a Python tool, that we named Crack.

Model Used for the Experiments

In Crack, it is possible to choose between the mesh, graph and hypergraph models. We have carried out some experiments on the effects of using the hypergraph model over the graph model. We have observed qualitatively that using the graph model, the *edgecut* of the returned solutions is on average smaller than when using the hypergraph model, but the $cut_{\lambda-1}$ of the returned solutions is greater.

Nevertheless, though the hypergraph model is usually considered modeling better the communication time induced by a partition, we were not able to determine the impact of the model on the run time of a real simulation. For the experiments reported in this document, we will use the graph model.

Multilevel Implementation

The multilevel implementation in Crack uses an original model. Indeed, it models the multilevel algorithm as a finite-state machine whose states are coarsening, partitioning or prolongation phases. The user defines the states and the transitions between states. Modeling the multilevel algorithm as a finite-state machine enables the user to define easily variations on the multilevel scheme.

Example

For example, the default multilevel algorithm of MeTiS, which has been described throughout this document, is not as simple as the abstract multilevel Algorithm 26. For example, MeTiS modifies the imbalance tolerance depending on the current level, or uses a `Rebalancing` function in addition to the `FM` refinement algorithm. These modifications can be visualized simply using a finite-state machine, as displayed in Figure A.1.1 on page 238.

Another use of a finite-state machine to represent easily a multilevel algorithm is when defining variations of the V-cycle (which is the common multilevel algorithm), such as the W-cycle. The W-cycle has been introduced by Ouyang *et al.* [2000] and Walshaw [2004], and studied in particular by

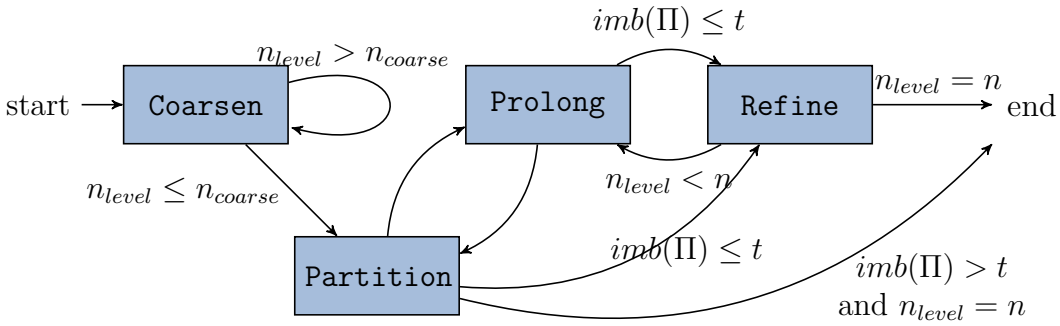


Figure 9.4.1 – The multilevel algorithm used for the tests. n_{level} is the number of vertices of the coarsened graph, and t is the imbalance tolerance.

> Sanders and Schulz [2010]. It consists in applying V-cycles (the default multilevel algorithm) on coarsened graphs.

For our tests, we use the classic multilevel algorithm (the V-cycle), defined in Algorithm 26. If we define it using a finite-state machine, as displayed on Figure 9.4.1, it is composed of four states:

- the `Coarsen` state, which is repeated until the number of vertices becomes smaller than n_{coarse};
- the initial `Partitioning` state, which alternates with the `Prolong` state as long as it does not find a solution;
- the `Prolong` state, which prolongs the graph and the partition (if it is solution) onto the upper level;
- the `Refine` state, which is a partitioning state, called alternatively with the `Prolong` state as long as a solution has been found, until a partition of the original graph has been computed.

Note that this scheme was designed knowing that the `Refine` function will always return a partition respecting the balance constraints.

Algorithm Specification File

Crack was designed to be flexible, which means that one should be able to set precisely and easily every algorithm and parameter used. We chose to use YAML specification files, that are described accurately by Ben-Kiki *et al.* [2009]. YAML is a human-friendly language that can be used among others with Python, C++ or Java. It transforms a file written in the YAML language into a data structure using a parser. Note in particular that indentation matters in YAML, and that a `#` comments the rest of the line.

For Python, the parser is called PyYAML, and it outputs a Python data structure, which in our case will be a list of algorithms. We will use the example

```
# ********************************************************** #
# YAML specification: VNBest+FM initial partitioning algorithm #
- algo: random                      # Random partitioning       #
- algo: number_best                 # Imbalance refinement      #
- algo: fm                          # Com_cost  refinement      #
  args:
    model: graph                    # Will minimize the edgecut #
    stop_inn:                       # Inner loop stop condition #
        algo: consecutive_negative_gain
        args:
            nmoves: 120
# ********************************************************** #
```

Figure 9.4.2 – Example of a YAML specification file

file displayed in Figure 9.4.2 to explain how such a list is encoded in Crack.

First, the file starts with two commented lines, that are not interpreted by PyYAML. Then, the first character is a `-`, which introduces the first element of a list. This element will be composed of the Python dictionary `{"algo": "random"}`, which has one key (the string `algo`) with which one value is associated (the string `random`). This means that, in the first place, Crack will call the `RandomPart` Algorithm 10 on page 69.

Then, on the second line, the first character is once again a `-`, which introduces the second element of the list. This line means that after that `RandomPart` finishes, the partition will be refined with the `VNBest` Algorithm 43 on page 168.

Finally, starting from the third line, the lines all exhibits the same level of indent, which means that they all relate to the third element in the list. This element is a dictionary of two keys, that are `algo` and `args`. In Crack, the value of the `algo` key should be a string corresponding to an algorithm name, and the value of the `args` key are a dictionary specifying the arguments of this algorithm. In this case, we will use the `FM` Algorithm 44 on page 176, minimizing the *edgecut*, and stopping an inner loop after that 120 moves of negative gain have been performed in a row.

Besides, we have given in Figure 9.4.2 the YAML specification file for an algorithm relying on the multilevel framework.

Remark

The specification format of Crack is quite similar to Scotch strategy strings. However, YAML files are more flexible (for example, data types are given automatically) and are more easily understood by human beings (for example, thanks to the indentation and the use of comments). In Scotch, algorithm names are defined by one character only (though it is not the case for the argument names), and the strategy strings are quite difficult to

handle.

Conclusion

In this chapter, we have introduced in Section 9.1 the set of instances that we will use to evaluate the possible heuristics that we defined in Chapters 6, 7 and 8. The instances comprise 6 meshes, of which one is an industrial test case. Except for the industrial test case, which has only one weight distribution, we generate for all meshes 3 particular weight distributions. The imbalance tolerance belongs to the set $\{5\%, 1\%, 0.2\%\}$, and the number of parts requested is $\{2, 32, 128\}$. Each heuristic will be run 100 times on each instance, except for the industrial case for which we were not able to perform as many runs.

Section 9.1 also defined an algorithm which, given a mesh, generates a multi-criteria weight distributions that mimic those of particle-in-cell simulations. The weight distributions for each mesh rely on this method.

Then, Section 9.2 showed that one needs to run an algorithm many times in order to be able to analyze its results. However, representing the result distributions is far from obvious, and as seen in Section 9.3.2, classic indicators are not sufficient to compare heuristics. Consequently, we will rely on the *cumul* function defined in Section 9.3.3, that helps one to compare quite easily heuristics (a curve on the top left is better) and also shows on the same plot the proportion of solutions returned.

Finally, we have introduced in Section 9.4 the framework that we will use to compare the various algorithms that were defined in this document.

Chapter 10

Comparison of Heuristics

Contents

This chapter performs an experimental analysis of the various algorithms studied throughout this document. The instances used were defined in Section 9.1, and include an industrial test case named *LMJ*. Comparisons will be performed by studying the proportion of the solutions returned using the *cumul* function of the distribution of the communication cost of the solution returned, defined in Section 9.3.3.

In particular, using Crack, we will compare in Section 10.1 the different initial partitioning algorithms, and the different weight restriction policies. Among the studied algorithms, some will be implemented in a multi-criteria version of Scotch. In Section 10.2, we will compare this multi-criteria version of Scotch with existing multi-criteria partitioning tools and with the algorithms of Crack, in the bipartitioning case.

Section 10.3 will study if our algorithms can be improved thanks to a

different ordering policy when computing the matchings. Finally, we will show in Section 10.4 that our algorithms are also efficient for k-partitioning.

10.1 A "Crack Analysis" of Initial Partitioning Algorithms and Restriction Policies on the Ability to Find a Solution at Coarse Levels

This section compares the imbalance of the partition returned by two kinds of algorithms. Firstly, we compare the restriction policies introduced in Section 6.3. These restrictions stem from the theoretical results of Chapter 5, which settled that bounding the vertex weights more would simplify the search for a solution. Secondly, we compare initial partitioning algorithms. This is a critical issue because our multilevel algorithm relies on the fact as long as that the initial partitioning algorithm finds a solution, we are guaranteed to return a solution at the end of the algorithm.

Section 10.1.1 will first show that the run time of our initial partitioning algorithms is suitable for our requirements. Then, Section 10.1.2 will focus on the imbalance of the partitions produced at the coarsest level. The coarsest level is reached when $n_{level} \leq n_{coarse} = 120$. Yet, one has also to consider the case when n_{coarse} is higher, or when the initial partitioning algorithm fails to return a solution at the coarsest level. Section 10.1.3 will study the evolution of the imbalance of the returned partitions at coarsen levels close to the coarsest level. Finally, Section 10.1.4 will demonstrate and analyze the impact of the restriction policy and of the initial partitioning algorithm on the communication cost of the returned partitions.

10.1.1 Run Time of the Initial Partitioning Algorithms Implemented in Crack

Table 10.1.1 reports the average run time of each initial partitioning algorithm on the tested instances. Times are given in ms. We can see that `VNBest` is only about 5 times slower than `VNFirst`, thanks to the implementation described in Section 7.2. In spite of its seemingly higher complexity, `VNBest` is even faster than `GGG`.

As we will see in Section 10.2.1, the run time of the initial partitioning algorithms is negligible with respect to the run time of the whole multilevel algorithm. Moreover, the run time of `VNFirst` and `VNBest`, which address a vector-of-numbers partitioning problem, are far shorter than the run time reported in Section 3.4.3 for existing vector-of-number partitioning problems.

Table 10.1.1 – Average run times of each initial partitioning algorithm, in ms

level	`Randomize`	`VNFirst`	`VNBest`	`GGG`
coarsest ($n \sim 90$)	0.504	1.25	5.59	7.39
coarsest + 3 ($n \sim 500$)	2.52	6.48	38.3	74.5

10.1.2 Comparison of the Imbalance at the Coarsest Level

This section compares the imbalance of the partitions returned by the initial partitioning algorithms, when they are run on the coarsest graph produced by the different `Restriction` policies.

Implementation – Crack

The coarsest graph is produced using different `Restriction` policies. However, we always use the `OrderRandom` scheme and the `HEM` algorithm to compute the matching.

`VNBest` and `VNFisrt` are given a random partition computed by the `RandomPart` algorithm as input.

Finally, the `GGG` algorithm is not used as a stochastic algorithm, unlike in MeTiS, because it is an experimental study. The random seeds are computed once per execution of the `GGG` algorithm.

Comparison of Initial Partitioning Algorithms

We will use a representation of the *cumul* function for the imbalance. We will start by examining the results for each initial partitioning algorithms, displayed on Figure 10.1.1. For each initial partitioning algorithm, we display the area in which its *cumul* functions lie (there are four *cumul* functions per initial partitioning algorithm, because we tested four `Restriction` policies).

The objective of each initial partitioning algorithm, whatever the tolerance, is to minimize the imbalance of the returned partition. This means that, on the figure, algorithms whose areas span the most on the top left corner are better. This is the case of `VNBest`, which returned partitions of imbalance smaller than 1% for half of the executions.

The area of `VNFirst` is quite close to that of `VNBest`, but always a little lower. This means that the partitions returned by `VNFirst` have in general a higher imbalance than those returned by `VNBest`. When searching for a balanced partition of a vector-of-numbers partitioning problem, for these instances, performing the move that leads to the smallest imbalance therefore leads to better local optima.

`GGG` rarely manages to return partitions of very small imbalance, but its

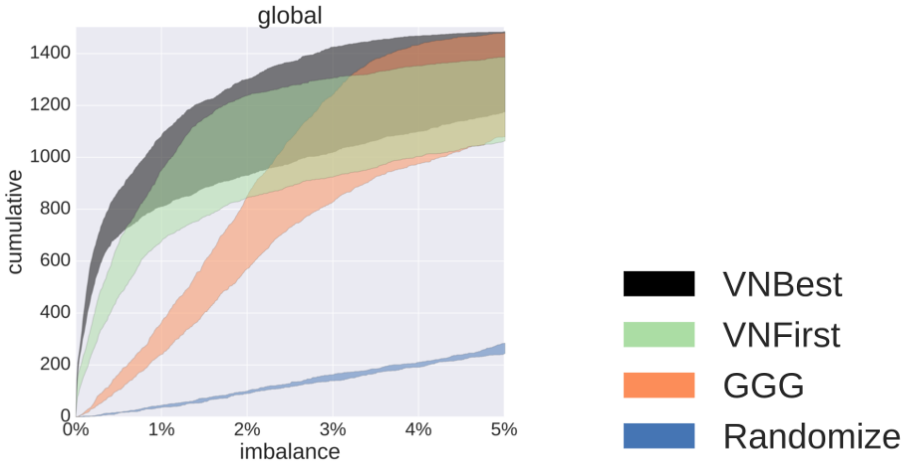


Figure 10.1.1 – For a given algorithm, the *cumul* function introduced in Section 9.3.3 counts, given an imbalance x, the number of times that the algorithm returned a partition of imbalance smaller than or equal to x (Therefore, top left is better). The *cumul* functions for one initial partitioning algorithm and several restriction policies are grouped; the figure displays the areas in which their *cumul* functions lie.

area almost matches that of `VNBest` when considering imbalance higher than 4%.

Finally, the `Randomize` algorithm manages to return about 13% of partitions of imbalance smaller than 5%.

Whereas `Randomize`'s area is thin, the areas of `VNFirst`, `VNBest` and `GGG` rapidly enlarge. This means that the restriction policy impacts their performance. In order to compare each scheme (initial algorithm and restriction policy), we will use the average, which is simpler to represent and analyze.

Comparison Using the Average

Figure 10.1.2 displays the average imbalance of the partitions returned by each initial partitioning algorithm, when partitioning the coarsest graph for each `Restriction` policy. We only displayed the averages for the meshes *Wave* and *Shock*, because they are quite similar to the other meshes, as displayed in Figure A.3.3 on page 246.

For each initial partitioning algorithm, the results are quite similar; more restrictions decrease in general the average imbalance. Besides, for *Shock* and *Mushroom*, a tight `Restriction` policy enables `VNFirst` and `GGG` to return on average partitions of imbalance as small as that of `VNBest`. This shows that restricting the vertex weights during the coarsening phase helps the initial partitioning algorithms to find partitions of smaller imbalance, which raises the probability to find a solution at the coarsest level.

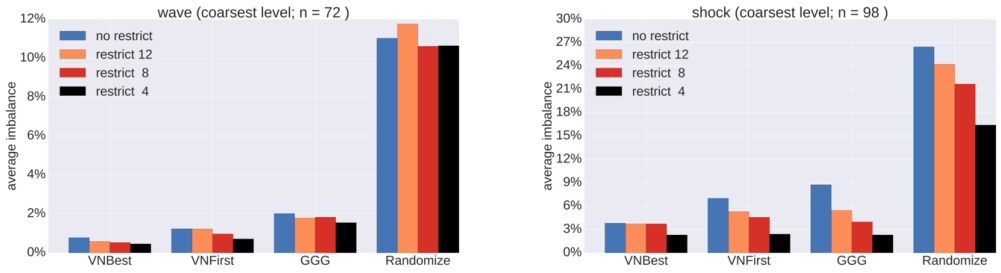


Figure 10.1.2 – Averages of the imbalance of the partitions returned by each algorithm for each restriction policy and each initial partitioning algorithm, at the coarsest level, over 300 runs.

Numerical values of the average of the imbalance of the partitions found show that the initial partitioning algorithm is not able to find a balanced partition at the coarsest level for each execution. In this case, the global algorithm specifies that another initial partitioning will be tried at the next level. The next section will therefore study the performance of the initial algorithms at less coarse levels.

10.1.3 Evolution of the Imbalance at Coarse Levels

This section studies the imbalance of the partitions returned by the initial partitioning algorithms at different coarse levels, which amounts to changing the value of n_{coarse}. Thus, as we will see further, understanding the evolution of the performance of initial partitioning algorithm may give information on how to set n_{coarse}.

Figure 10.1.3 plots the average imbalance of the partitions returned by each algorithm for the coarsest level and the three previous levels (the average number of vertices at each level, n, is displayed above each plot) for the mesh *Wave*. The trends are quite similar for the other instances, whose plots are displayed in Figures A.3.4 to A.3.7 on page 248.

For each initial partitioning algorithm, the average imbalance of the returned partitions decreases when n_{coarse} (and therefore n) increases. Indeed, it becomes easier to balance the partition (this generalizes the analysis of the number partitioning problem performed in Section 3.1 to the vector-of-numbers partitioning problem). In particular, the decrease is spectacular for `GGG`, which returns partitions of imbalance smaller than `VNFirst` at finer levels, for three meshes out of five.

In our case, the goal of the initial partitioning phase is to find a balanced partition. If $n_{coarse} = 120$, the probability to find a balanced partition may be quite small for many instances, depending on the tolerance. Therefore, when the tolerance is tight, n_{coarse} may be increased, or the initial partitioning

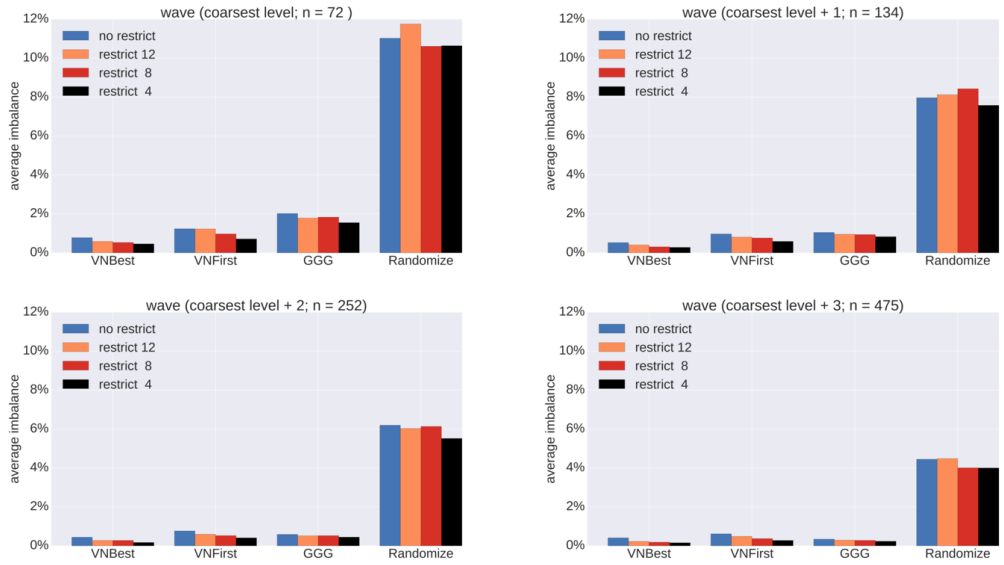


Figure 10.1.3 – Average of the imbalance of the partitions returned for the mesh *Wave* at different coarse levels. The average was computed over 100 runs on each of the three weight distributions.

algorithm may need to be run additional times to increase the probability to find a balanced partition.

Conclusion

So far, we have studied the imbalance of the partitions of the coarsest graph, when using different initial partitioning algorithms and different `Restriction` policies on the vertex weights. The main results are:

- the initial partitioning algorithm `VNBest` returns in general partitions of smaller imbalance than the other initial partitioning algorithms, for this set of instances. This result is valid regardless of the `Restriction` policy;
- establishing tighter bounds on the vertex weights leads each initial partitioning algorithm to return, on average, partitions of smaller imbalance. This result supports a conclusion drawn in Chapter 5, that bounding the vertex weights helps the algorithms to find balanced partitions more easily;
- increasing n_{coarse} leads the initial partitioning algorithms to find more easily partitions of smaller imbalance. This is particularly the case for `GGG`, for which the average of the imbalance of the partitions returned reaches the average of the imbalance of the partitions returned by `VNBest` when n_{coarse} increases.

In the next section, we will study the communication cost of the partitions found using `VNBest` and `GGG`, and the different `Restriction` policies. Indeed, the `VNFirst` algorithm is based on the same principle as `VNBest`, so their behavior regarding the communication cost should be the same.

10.1.4 Impact of the Restriction policy and of the Initial Partitioning Algorithm on the Communication Cost

Figure 10.1.4 displays the *cumul* function of the communication cost of the partition for the meshes *Onera*, *Shock* and *LMJ*, which are representative of the *cumul* functions obtained for all meshes. The tested algorithms rely on two initial partitioning algorithms used (`VNBest` with plain lines, `GGG` with dashed lines), and various `Restriction` policies. We will first consider results for the meshes *Onera* and *Shock*.

Comparison of the Initial Partitioning Algorithms. This amounts to comparing plain lines (`VNBest`) with dashed lines (`GGG`), for a given color. For all meshes, when the tolerance is 5%, `VNBest` and `GGG` are equivalent. In general, using `GGG` yields solutions of smaller communication cost than `VNBest`. Indeed, the performance of `VNBest` worsens when the tolerance tightens. A possible explanation is that, when the tolerance is tight, the solution space is less connected, and the refinement algorithm (`FM`) does not manage to reach the optimal solution.

Comparison of the `Restriction` Policies This amounts to comparing, for a given linestyle, the lines of different colors. For all meshes, more restrictions help to return partitions of smaller communication cost. In particular, for the meshes *Mushroom* and *Shock*, the gap between restriction policies is large when using `VNBest`. Once again, this result supports a conclusion of Chapter 5, but here on the connection of the solution space. Therefore, we claim that more restrictions are likely to increase the connection of the solution space, enabling the refinement algorithm to reach the optimal solution more easily.

Results for the *LMJ* Mesh. We chose to discuss results for *LMJ* separately, firstly because they are more difficult to analyze due to the number of runs that we were able to perform (only 18 runs per algorithm, for $t \in \{5\%, 1\%\}$), and secondly because the findings regarding the `Restriction` policies is quite different. Indeed, the tightest restriction policy, `Restrict 04`, returns in general solutions of higher communication cost. As the vertex weight distribution was not generated as for the other meshes, this shows that a very tight `Restriction` policy is not suited for every instance. Therefore, it would be worth to study

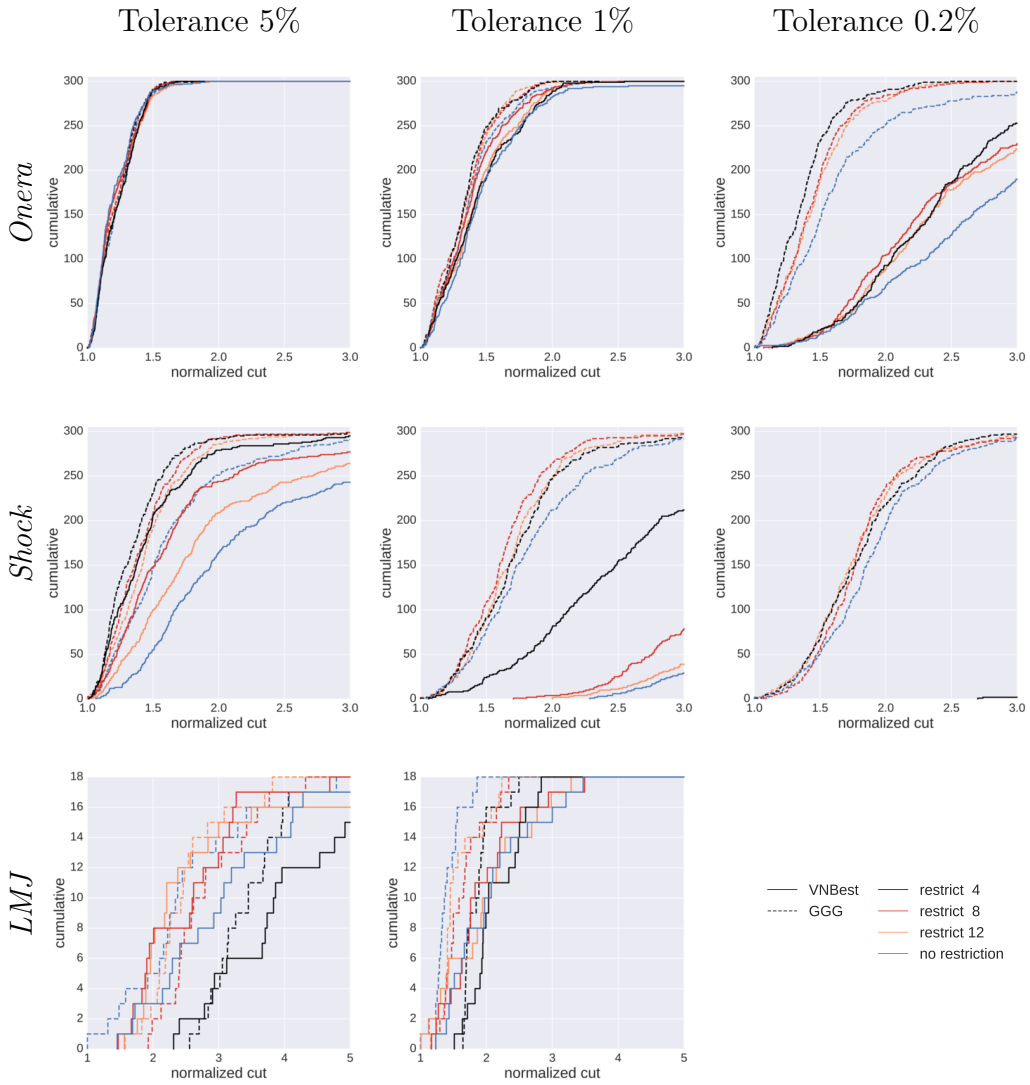


Figure 10.1.4 – Cumulative plots of the communication cost of the partitions returned when using either `VNBest` or `GGG` for the initial partitioning phase, and when enforcing various restriction policies for the coarsening phase.

how to set the maximum weight of a merged vertex during the coarsening phase.

Conclusion

In this section, we have studied the impact on the communication cost of using the initial partitioning algorithms `VNBest` and `GGG`, and of using different `Restriction` policies. The main results are:

- when the tolerance tightens, algorithms relying on `VNBest` do not manage to reach a solution near the optimal, unlike algorithms using `GGG`;
- in general, restricting the vertex weights enables one to return solutions of smaller communication cost, which supports our assumption that bounding the vertex weights increases the connection of the solution space in the multi-criteria case;
- however, determining the maximum threshold value for the vertex weights will need further investigation, since a too tight bound leads to an increase in the communication cost of the returned partition.

In the following, we will restrict our study to the algorithms relying on `VNBest` and `GGG` for the initial partitioning phase, and on the `Restrict 8` policy during the coarsening phase.

10.2 Implementation in Scotch and Comparison with Crack and Existing Multi-criteria Partitioning Tools

We have implemented a multi-criteria version of Scotch-6.0.4, and will compare in this section the performance of this new multi-criteria version of Scotch with MeTiS and PaToH. We will consider the results of two versions of Crack, one relying on `VNBest` and the other using `GGG`.

Implementation – Experimental Multi-criteria Version of Scotch

For the experiments, we prevented Scotch to run two independent multi-levels per execution (which is the default behavior in Scotch-6.0.4). Moreover, we also removed the relaxation of the imbalance tolerance at coarse levels, since we focus on returning balanced partitions. Finally, we implemented the `VNBest` initial partitioning algorithm in Scotch because, as shown in Section 10.1, it yields the highest probability to return a balanced partition.

The remaining differences between Scotch and Crack are its ordering of the vertices when determining the matchings during the coarsening phase, and the tie-breaking policy of its `FM` algorithm during the refinement phase, which, as explained by Alpert and Kahng [1995], can highly change the behaviour of the algorithm.

For matching ordering, Scotch adds a random tie-breaking policy to the increasing degree order. For `FM`, the gain table structure in Scotch has a Last-In-First-Out policy which defines the tie-breaking policy.

Implementation – MeTiS-5.1.0

We used MeTiS-5.1.0 with its default settings, whose implementation has been described throughout this document. One main difference of MeTiS over our approach is that it relaxes the imbalance tolerance at lower levels, and uses rebalancing mechanisms to improve the partition imbalance.

Another notable point is that, whereas Karypis and Kumar [1998b] introduced the `GGG` algorithm that we use with Crack in this section, the default settings for multi-criteria graph partitioning with MeTiS do not rely on this initial partitioning algorithm. As displayed on Figure A.1.1 on page 238, it uses a `Random` partitioning algorithm, followed by MeTiS's `FM` again, the rebalancing algorithm again, `FM`, the rebalancing algorithm, and a third call to `FM` (see file `initpart.c`, function `McRandomBisection`).

The final notable point is that, by default, MeTiS computes 5 partitions of the coarse graph when its number of vertices becomes smaller than $\max(\frac{n}{20\times\log_2(k)}, 30\times k) = \frac{n}{20}$ in our case ($k = 2$ and $\frac{n}{20} > 60$).

Implementation – PaToH-3.2

Recall that PaToH does not optimize the *edgecut*, but the $cut_{\lambda-1}$. We will nevertheless report its results in terms of *edgecut*, in order to compare them with those of Crack, MeTiS and Scotch.

10.2.1 Run Time

Figure 10.2.1 displays the average run time of each partitioning tool. Each point corresponds to the average on the 300 runs performed (100 per weight distribution) for a given tolerance. Standard deviation *std* is indicated as an error bar (the length of the bar is $\frac{std}{\sqrt{300}}$). Because PaToH is not an open-source software, we were not able to run it on our industrial test case *LMJ*.

A first observation is that Scotch run time is comparable to the run time of MeTiS and PaToH, which shows the viability of our algorithm. A second observation is that the standard deviation is negligible. A third observation is that the average run time is rather independent from the prescribed imbalance tolerance.

We did not include the running time of Crack in the figure, because its run time is not of the same order of magnitude. Crack run time ranges from on average 6s for bipartitioning the mesh *Mushroom*, to 2min for the industrial test case *LMJ* and up to 5min for the mesh *Shock*.

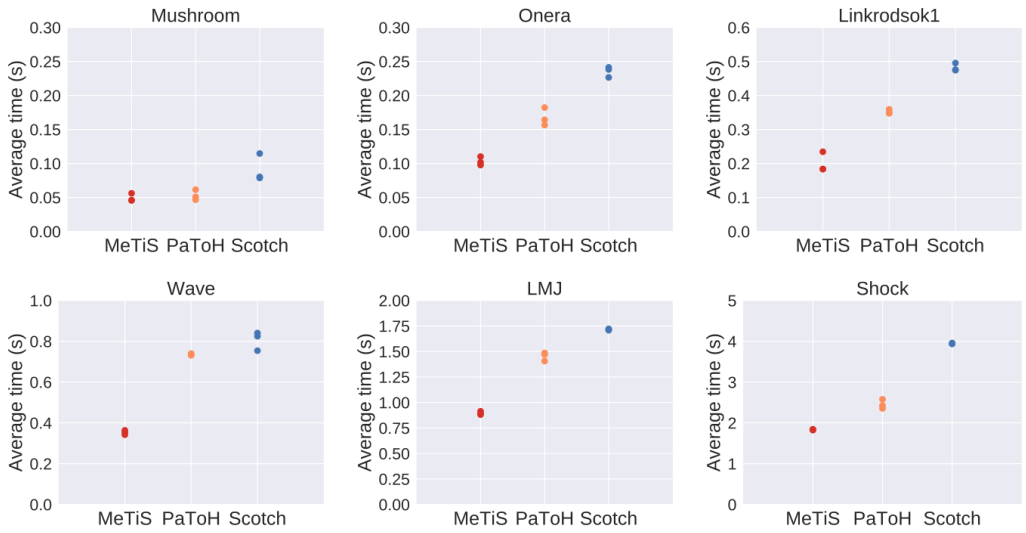


Figure 10.2.1 – Average time of Scotch, MeTiS and PaToH on each instance. Error bars are also displayed. For each algorithm, three points are plotted, corresponding to the imbalance tolerance of 5%, 1%, and 0.2%.

10.2.2 Ability to Find a Solution

Table 10.2.1 displays the proportion of solutions returned for Crack (using `GGG` or `VNBest`, results are the same), MeTiS, PaToH and Scotch. A check mark indicates that all partitions returned were solutions. Otherwise, the rounded percentage of solutions returned is indicated (so a 100% indicates that out of 300 runs, almost all partitions returned were solutions). We also provide, in parentheses, the average imbalance of the partitions that are not solutions.

Remark

Note that we accepted an error of 1% on the imbalance of the solution returned to consider it as a solution. Therefore, for an imbalance tolerance of 1%, we accepted as solutions the partitions of imbalance below 1.01%. Indeed, MeTiS, Scotch and PaToH returned some partitions of imbalance very close to the tolerance.

MeTiS fails to return solutions in many cases, and the imbalance of the partitions that are not solutions are on average imbalanced half more than the tolerance. The reason why MeTiS returns imbalanced partitions surely lies in the fact that it relaxes the tolerance at coarse levels. Note that, on the industrial test case *LMJ*, MeTiS does not find any solution when the tolerance is of 5%. However, it finds some when the tolerance is of 1%. This counter-intuitive behavior shows a lack of robustness in the MeTiS approach.

PaToH manages to return a solution at every execution for every mesh but *Shock*, and *LMJ* when the tolerance is very tight. Finally, Crack and Scotch

Table 10.2.1 – Percentage of solutions returned by each partitioning tool, for 100 runs on each of the 3 weight distributions generated on each instance, except for *LMJ* which has only 1 weight distribution. A check mark means that all partitions returned were solutions. When it was not the case, the average imbalance of the partitions that were not solution is indicated in parentheses.

instance mesh	t	Crack (%)	Scotch (%)	MeTiS (%)	PaToH (%)
Mushroom	5%	✓	✓	39 (6.21)	✓
	1%	✓	✓	25 (1.33)	✓
	0.2%	✓	✓	29 (0.25)	✓
Onera	5%	✓	✓	92 (6.82)	✓
	1%	✓	✓	73 (1.34)	✓
	0.2%	✓	✓	45 (0.22)	✓
Wave	5%	✓	✓	67 (6.50)	✓
	1%	✓	✓	54 (1.28)	✓
	0.2%	✓	✓	14 (0.24)	✓
Linkrodsok1	5%	✓	✓	✓	✓
	1%	✓	✓	✓	✓
	0.2%	✓	✓	11 (0.20)	✓
Shock	5%	✓	✓	13 (6.25)	100 (5.14)
	1%	✓	✓	35 (1.26)	56 (1.30)
	0.2%	✓	✓	11 (0.24)	2.0 (0.60)
LMJ	5%	✓	✓	0 (7.14)	✓
	1%	✓	✓	15 (1.39)	✓
	0.2%	✓	✓	1.0 (0.26)	2.0 (0.39)

return 100% of solutions for all meshes.

10.2.3 Comparison of the Communication Cost Distributions

Figure 10.2.2 shows the *cumul* function of the communication cost for the meshes *Onera*, *Wave* and *LMJ*. The *cumul* function for the other three meshes is displayed in Figure A.4.1 on page 250.

Results show that, for a tolerance of 5%, Crack return solutions of communication cost slightly smaller than for MeTiS and PaToH, whatever the initial partitioning algorithm. The performance of Crack degrades when the tolerance tightens, especially for the variant relying on `VNBest`, as we saw previously. Nevertheless, the version of Crack relying on `GGG` obtains a high proportion of solutions of small communication cost, whereas MeTiS algorithm seems inconsistent due to the high proportion of invalid solutions that it returns.

The results of PaToH are quite average. However, the comparison is quite difficult because it does not minimize the *edgecut*.

Finally, Scotch yields good results when the tolerance is in $\{5\%, 1\%\}$. When the tolerance is tight, indeed, we have shown in Section 10.1.4 that `VNBest` performance degrades. Surprisingly, the performance of Scotch is quite different from that of Crack_`VNBest`. For example, it performs very well on the industrial test case, *LMJ*. This shows that little differences of implementation can lead to big differences in the results produced.

Conclusion

In this section, we have compared a multi-criteria implementation of Scotch with two versions of Crack and with two partitioning tools, MeTiS and PaToH. A first result is that our algorithms, unlike the tested partitioning tools, return solutions at each execution. A second result is that, when the tolerance is of 5%, our algorithms return partitions of smaller communication cost than existing tools.

However, the performance seems to degrade when the tolerance tightens. The next section will study a last variation of the multilevel algorithm, which is the order of the vertices before computing the matchings.

10.3 Analysis of the Impact of the Matching Order Using Crack

This section aims at determining if the matching order impacts the distribution of the communication cost of the returned solution and, if this

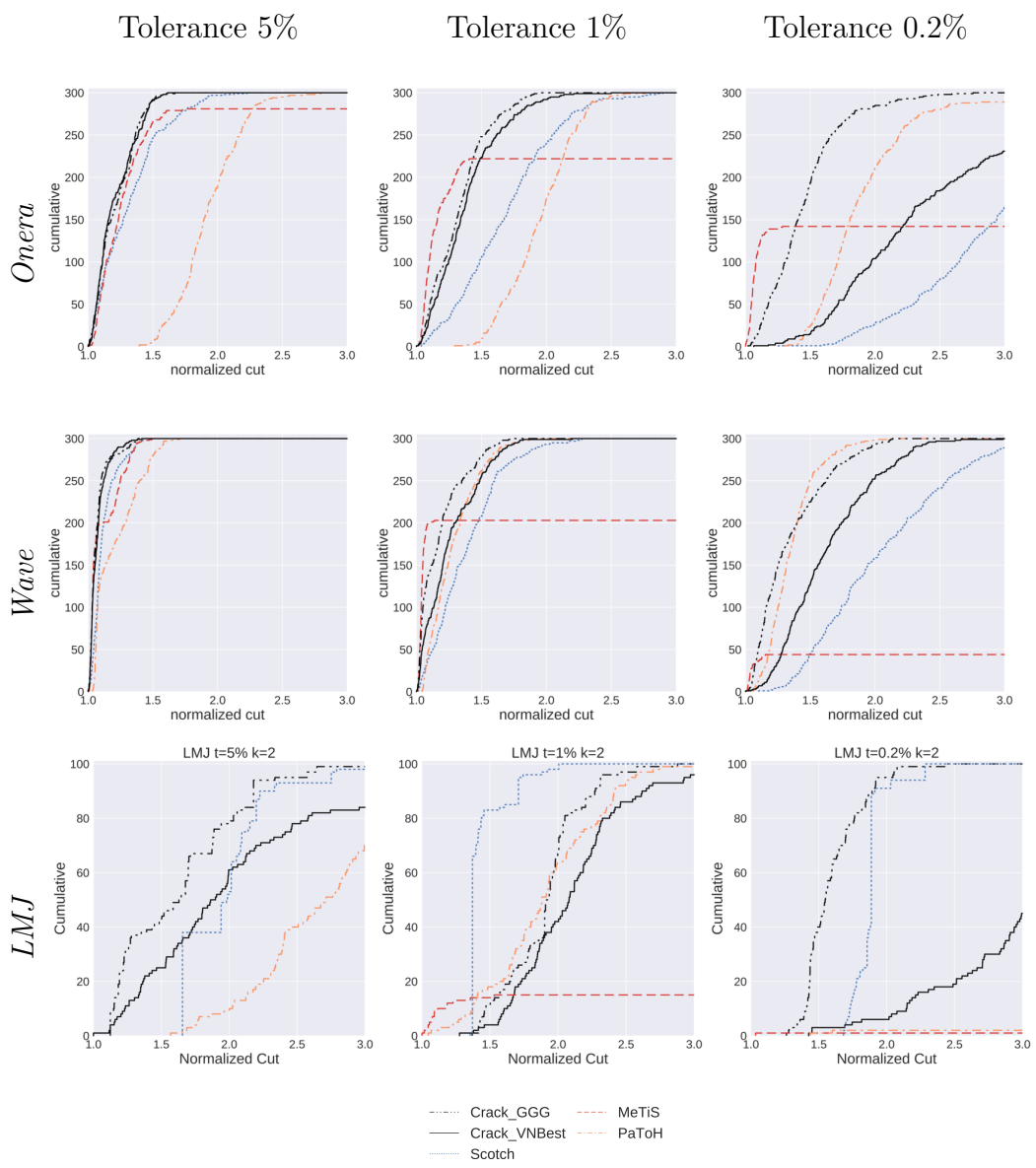


Figure 10.2.2 – Comparison of Crack, Scotch, MeTiS and PaToH on the three meshes, using the *cumul* function. For a given algorithm, $cumul(x)$ is the number of solutions of communication cost smaller than x returned by the algorithm. Since we aim at minimizing the communication cost, top left is better.
Each algorithm was run 100 times on each of the 3 weight distributions per mesh. Note that only the partitions that are valid are counted in the *cumul* function. The cut was normalized by the minimal communication cost found among all the algorithms.

is the case, which scheme is the best suited. The tested schemes were described in Section 6.2; they are `OrderPriority`, `OrderDegrees`, `OrderFirst` and `OrderRandom`.

Figure 10.3.1 reports the *cumul* function for the communication cost of the returned partitions for the meshes *Wave*, *Shock* and *LMJ*. The algorithms all used the `Restrict 8` policy during the coarsening phase, and different ordering schemes whose *cumul* functions are displayed in different colors. Algorithms relying on `VNBest` are displayed with plain lines, while those relying on `GGG` are displayed with dashed lines. Figure A.5.1 on page 251 reports the results for the other three meshes.

First, we observe that ordering the vertices using different strategies before applying the `HEM` scheme may considerably impact the *cumul* function. This is particularly visible when using the `OrderFirst` scheme (in yellow), which performs in general rather poorly compared with the other ordering schemes. As we explained in Section 6.2, this scheme will always consider the same vertices first for matching. Results thus show that matching schemes must avoid considering vertices in the same order.

Then, each of the other ordering strategies has at least one instance for which it returns partitions of smallest communication cost. Nevertheless, given an initial partitioning algorithm, the `OrderDegrees` scheme always manages to return partitions of small communication cost. This indicates that ordering strategies must also take care of the graph topology.

Conclusion

In this section, we have shown that ordering the vertices before matching has a heavy impact on the communication cost of the solutions returned. Moreover, ordering schemes must avoid to always consider vertices in the same order, but instead need to take into account the graph topology.

10.4 Comparison for k-partitioning ($k \in \{32, 128\}$)

This section reports in Figure 10.4.1 the results obtained by our multi-criteria version of Scotch, Crack using `OrderDegrees`, `Restrict8` and `VNBest` or `GGG`, MeTiS and PaToH for k-partitioning, with a tolerance of 5%.

Implementation – Crack

For k-partitioning with Crack, we used the recursive bisection algorithm `RB`, defined in Section 4.1. When applying a bipartitioning algorithm on a part, the tolerance is adapted considering the remaining bisection levels and the current imbalance of this part.

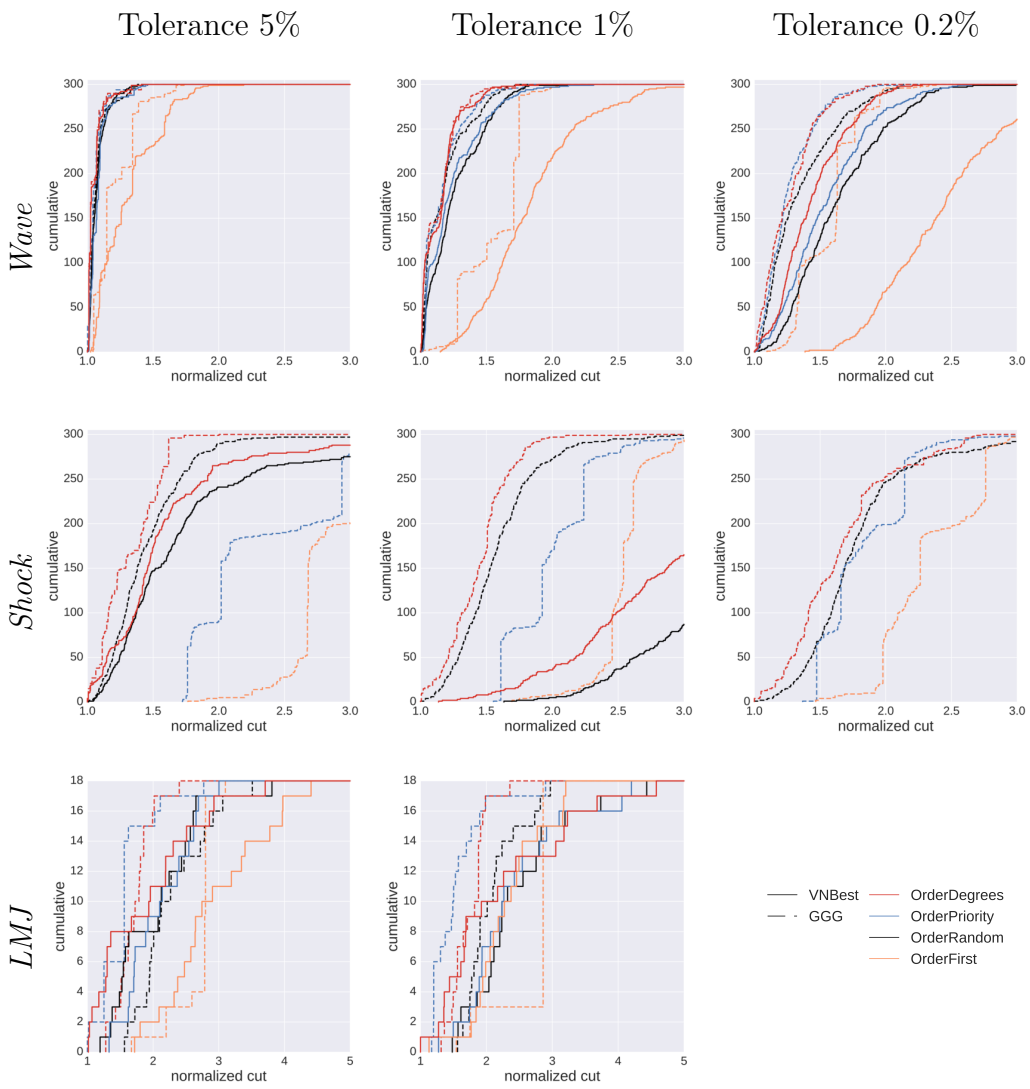


Figure 10.3.1 – Comparison of using different ordering policies on the communication cost of the partitions for three meshes, using the *cumul* function. For a given algorithm, $cumul(x)$ is the number of solutions of communication cost smaller than x returned by the algorithm. Since we aim at minimizing the communication cost, top left is better.
Each algorithm was run 100 times on each of the 3 weight distributions per mesh (except for *LMJ*, which has only 1 weight distribution). Note that only the partitions that are valid are counted in the *cumul* function. The cut was normalized by the minimal communication cost found among all the algorithms.

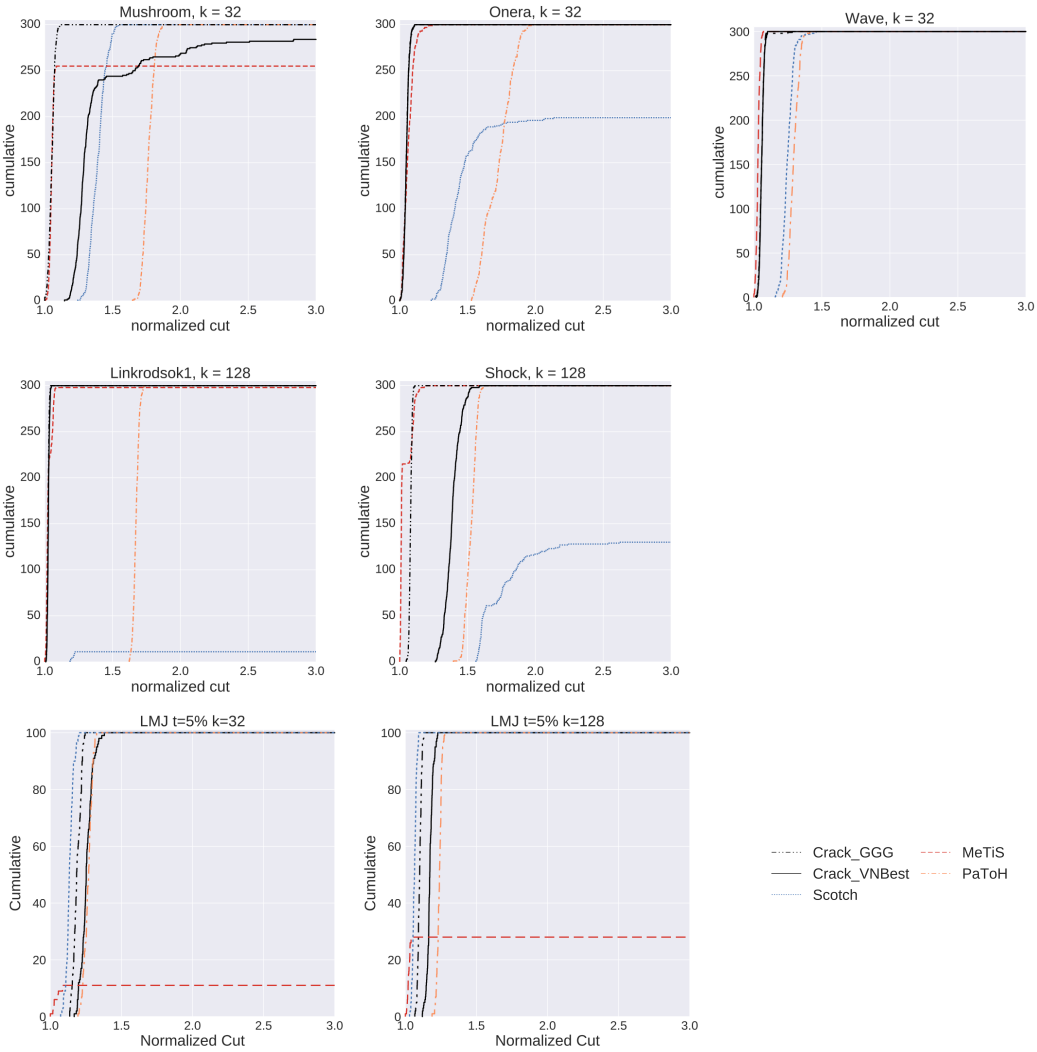


Figure 10.4.1 – Comparison of the communication cost of the solution returned by Crack, Scotch, MeTiS and PaToH for k-partitioning.

Results show that, despite using the recursive bisection scheme, which is said to be less effective than direct k-partitioning (see Section 4.1), the versions of Crack relying on `GGG` perform very well, since they return partitions with the smallest communication cost.

Moreover, on the industrial test case *LMJ*, Crack and Scotch manage to return a solution at each execution, unlike MeTiS, and the communication cost of the solutions of Crack and Scotch is very close to the minimal communication cost found by MeTiS.

Conclusion

In this chapter, we have performed an experimental analysis of the performance of the algorithms that have been defined throughout this document. To do so, we have run them several times on different instances, including an industrial test case. This performance was measured using the percentage of solutions returned, and the *cumul* function that represents the distribution of the communication cost of the solutions returned.

The main results are:

- our initial partitioning algorithm `VNBest` returns partitions of the coarsest graph with the smallest imbalance on average. Nevertheless, the evolution of the average imbalance showed that `GGG` was able to catch up with `VNBest` at finer levels;
- for a tolerance of 5%, using `GGG` or `VNBest` does not change much the distribution of the communication cost of the returned solutions. However, when the tolerance tightens, algorithms relying on `VNBest` do not manage to return as many solutions close to an optimal solution;
- enforcing restrictions on the vertex weights during the coarsening phase leads each initial partitioning algorithm to return more balanced solutions. This result supports our claim that bounding the vertex weights simplifies the search for a solution;
- restrictions also decreased the communication cost of the returned solutions, which sides with our claim that bounding the vertex weights simplifies the search for an optimal solution. However, how to find the optimal bound remains an open question. In our experiments, we relied on the `Restrict 8` policy;
- ordering the vertices before computing the matching is of great importance. In particular, the order must not always schedule the vertices in the same order, and must take into account the graph topology;
- comparison with other partitioning tools showed that, unlike the others, our approach always returned a solution. In particular, MeTiS policy to relax the imbalance tolerance leads to counter-intuitive results, when MeTiS finds more solutions for a tighter tolerance. This was particularly

true on the industrial case, for which MeTiS returned 0 solutions for a tolerance of 5%, but found some when the tolerance was of 1%, whereas Crack and Scotch managed to always return a solution;

- in addition to always returning a solution, our approach achieves to return partitions of small communication cost, even clearly smaller in some cases;
- our approach is also efficient for k-partitioning using the recursive bisection scheme. The communication cost is as small as when using other partitioning tools, and on the industrial test case, it manages, unlike MeTiS, to always return a solution.

Conclusion

The work presented in this thesis aimed at the reduction of the run time of mesh-based multiphysics simulations running on distributed memory architectures. The main characteristic of multiphysics simulation is that they couple several computation phases. In Chapter 1, we described such simulations and showed that, in order to reduce their run time, one should balance the workload between computation units for each phase, minimize the communication time induced by the data distribution, and minimize the time needed to partition the mesh.

Then, we explained in Chapter 2 how these objectives can be modeled, making several approximations. These approximations resulted in the formulation of the classic multi-criteria mesh/hypergraph/graph partitioning problems, which search for a partition respecting balance constraints and minimizing a communication cost function. Moreover, we also defined the vector-of-numbers partitioning problem, which can be seen as a generalization of the classic number partitioning problem, or as a subproblem of the multi-criteria mesh partitioning problem. Finally, we introduced the notion of fitness landscape, which defines how a partitioning algorithm explores the set of all partitions in order to find an optimal solution.

We showed in Chapter 3 that the number partitioning problem is a hard problem, which has been well studied. Nevertheless, few algorithms can be directly applied to the vector-of-numbers partitioning problem. The situation is quite similar for the multi-criteria mesh/hypergraph/graph partitioning problems: while many algorithms have been designed for the mono-criterion case, few exist in the multi-criteria case, as evidenced in Chapter 4.

Our main contributions include a theoretical analysis of the solution space in Chapter 5. In particular, in the mono-criterion case, we stated a bound on the vertex weights that guarantees the existence of a solution. The same bound guarantees that the solution space is connected when using a refinement algorithm passing from one solution to another by switching the part of a single vertex at a time. Therefore, we conjectured that low vertex weights are likely to increase the number of solutions and the probability that the solution space is connected.

Based on this conjecture, we analyzed variations of the multilevel framework. Existing partitioning tools implement some of these variations, but they do not always report about them, so we highlighted the diversity of implementations and analyzed their consequences by referencing their source code. The different algorithms were compared in Chapter 10, when run on a set of instances described in Chapter 9. The set of instances comprises one industrial test case, and five meshes for which three fictitious weight distributions per mesh were generated. The generation method introduced reflects weight distributions used in Particles-In-Cells simulations.

In order to compare the algorithms, we showed that each algorithm needed to be run many times, and we defined the *cumul* function that helps compare on the same plot both the number of solutions returned and the distribution of the communication cost of the returned solutions. We claim that this method also enables one to evaluate algorithms more easily.

Chapter 9 also introduced the framework, named Crack, that we implemented to perform our experiments. Crack is a flexible partitioning tool that uses user-friendly YAML specification files, and relies on a finite-state automaton to run multiple variations of the multilevel algorithm.

In Chapter 6, we discussed restricting the vertex weights, in order to avoid creating heavy weights in the coarsened graphs. In Section 10.1, we experimentally verified that restrictions enable indeed initial partitioning algorithms to find solutions more easily, and also that algorithms relying on restrictions find solutions closer to the optimal solution.

We also considered in Chapter 6 coarsening schemes relying on various ordering algorithms that influence the behavior of the Heavy-Edge Matching algorithm. Results of Section 10.3 showed first that using the default order at each step led to solutions of higher communication cost. Then, among the other policies, the gap was small, but ordering using the mesh topology was the most robust approach.

We proposed two initial partitioning algorithms in Chapter 7. These algorithms, that are local optimization algorithms, tackle the multi-criteria mesh partitioning problem as a vector-of-numbers partitioning problem. The first one, `VNFirst`, moves any vertex if it decreases the imbalance, the second one, `VNBest`, moves a vertex so that the imbalance decreases the most. For `VNBest`, as shown in Section 10.1.1, our data structure enabled it to take less time than the classic greedy-graph-growing algorithm `GGG`.

As reported in Sections 10.1.2 and 10.1.3, `VNBest` returns partitions of the coarsest graph of smaller imbalance than `VNFirst` and `GGG`. Besides, as seen in Section 10.1.4, when the tolerance is of 5%, algorithms relying on `VNBest` return solutions as good as those returned by algorithms relying on `GGG`. Nevertheless, its performance degrades when the tolerance tightens, because optimal solutions become more difficult to reach.

In Chapter 8. we argued that relaxing the tolerance during the refinement phase is not needed, and can lead algorithms to return imbalanced partitions. When comparing with the existing multi-criteria partitioning tools MeTiS and PaToH in Section 10.2.2, we showed that our approach, be it with Crack or with the multi-criteria version of Scotch that we implemented, was the only one to have returned a solution after each call. Moreover, in Sections 10.2.3 and 10.4, we showed that the communication cost of the solutions of Crack and Scotch was equivalent or better than the one of the other partitioning tools.

Future Works

Models. Chapter 2 has defined many models to reflect the minimization of the run time of multiphysics simulations run on memory distributed architectures. Despite many claims on the efficiency of a model over the others, the influence of the model has not been reported for industrial cases, which may explain that the simplest one, the graph model, remains popular. A study or reports on the accuracy of each model in practice would be very useful, both for the users and the developers. Such study could also help the user to set the imbalance tolerance.

Instances. In this document, we were able to compare our algorithms on one industrial test case. It is also possible to find meshes or graphs used in practice on the DIMACS [2012] website or on the database provided by Soper *et al.* [2004], but no weight distribution is provided. This is particularly problematic when experimenting on multi-criteria partitioning algorithms. Indeed, as we showed in this document, the weight distribution is preponderant, so it can change the performance of an algorithm. Though we proposed a method to generate plausible multi-criteria weight distributions, they correspond only to Particles-in-Cells simulations. Therefore, a database gathering both meshes and (multi-criteria) weight distributions used in practice would be highly profitable both for the algorithm designers and the users, as the algorithms designed would tackle the exact problems of the users.

Algorithms. The bar plots displayed in Section 9.2 and the *cumul* functions in Sections 10.2.3 and 10.4 show that current partitioning algorithms lack robustness, as for a given instance, changing the random seed can lead to a difference of a factor of more than two in the communication cost of the returned solution. Improving this robustness is a major challenge, since the multi-criteria graph partitioning problem is NP-Hard. Nevertheless, it is possible, and using simple mechanisms as restricting the weights in the coarsened graphs. However, how to set the maximum allowed weight remains an open question. A starting point could be to consider the bound that appears in our theorems: half the tolerance (when the weights are normalized).

In order to improve robustness, another possibility is to run the algorithm

several times, keeping only the best solution found. This is the policy of Scotch-6.0.4 (that searches for two solutions). How many executions are needed? This appears to be a statistical question, but in practice, we also need to take into account the fact that algorithms are tested on a finite set of meshes, and that the robustness of an algorithm can change for a mesh not included in the experimental set. Therefore, one also needs to understand the properties of a mesh that will make that an algorithm will have a high robustness or not on the input mesh.

Another technique used by MeTiS-5.1.0 is to run the algorithm several times on the coarsened graph. We also studied this scheme, but could not find a correlation between the communication cost of partition of the coarsest graph and the communication cost of the partition of the original graph (good partitions of the coarsest graph did not always lead to good partitions of the original graph, and reversely). Characterizing such a correlation would be definitely worthy.

Fitness Landscapes. We have analyzed the fitness landscape issued from FM-like local optimization algorithms. We have discovered that such fitness landscapes comprise large "plateaux", that are sets of neighboring solutions of same communication cost. We also attempted to define the "ruggedness" of a fitness landscape. The ruggedness of a landscape characterizes its shape; basically, mountains are more rugged than hills, which are more rugged than plains. The FM algorithm is hill-climbing, because it allows selecting partitions of higher communication cost. Such a feature is mandatory when the landscape is rugged, in order to bypass local optima.

However, the FM algorithm always selects a neighboring solution of smallest communication cost. In a very rugged landscape, this means that `FM` would descend in every valley, descending at the bottom before attempting to escape from them. Other policies, such as selecting any neighboring solution that decreases the communication cost, or even selecting a neighboring solution that decreases the least the communication cost (when it really decreases the communication cost), could avoid local optima to descend directly in the lowest valley.

We studied the performance of such algorithms, that we called `FMFirst` and `FMWorst`. Their distributions of the communication cost, especially for `FMFirst`, were more spread than the one of `FM`, and usually not as good. Nevertheless, our tests were not performed using the multilevel framework, and studies on whether `FMFirst` and `FMWorst` can be useful would be very interesting.

Implementation. We presented in this document Crack, a flexible graph partitioning tool. Indeed, though many graph partitioning tools exist, they mostly focus on performance. However, being able to implement, test, and compare rapidly new and exotic algorithms is very difficult using such tools.

Distributing and improving our work in Crack, or proposing a similar partitioning tool, would highly benefit to researchers in partitioning algorithms. In particular, such a framework should enable the use of various models (such as mesh, graph, hypergraph), provide an interface to design benchmarks, and help the researcher to compare the algorithms by detailing the results using various methods.

Such a tool would also benefit to users, as they could test and compare many partitioning algorithms on their own test cases, and thus select the algorithm which is best suited for their applications.

Other Applications of Multi-criteria Partitioning. If during a simulation, the weight distribution changes in a predictable way, multi-criteria partitioning algorithm can be used instead of repartitioning algorithms, by adding the weight distributions at selected time steps as extra criteria.

Other Applications of Our Work. We proposed a methodology to study local optimization algorithms. This methodology can apply to similar problems than the multi-criteria partitioning problem. For example, Morais [2016] considers partitioning a mesh under memory constraints. His work takes into account the duplication of bordering cells, which in some cases leads to memory overflows.

Appendix

Contents

A.1 MeTiS-5.1.0 Default Multi-criteria Partitioning Algorithm

We will describe MeTiS default algorithm using a finite-state machine, as defined in Section 9.4 in order to represent Crack multilevel algorithm. Figure A.1.1 displays the finite-state machine for MeTiS default algorithm. Therefore, this is the algorithm that we compare with in the experiments of Sections 9.2 and 10.2.

MeTiS first coarsens the graph using the `HEM` policy (details about MeTiS coarsening phase were explained in Chapter 6). When the number of vertices of the coarsened graph (noted n_{level}) becomes smaller than $n_{coarse_1} = \max(0.05\frac{n}{\ln k}, 30k)$, MeTiS will run $n_{runs} \in \{4, 5\}$ times the following on the coarsened graph:

- continue the coarsening phase until the number of vertices becomes smaller than $n_{coarse_2} = 100$;
- perform an initial partitioning algorithm, which means calling successively the `RandomPart` algorithm, `FM`, MeTiS' `Rebalance` algorithm, `FM` once again, another time the `Rebalance` algorithm, and a third call to `FM` (MeTiS implementation of these algorithms was defined in Chapter 8);
- uncoarsen the graph, which means first to `Prolong` the partition to the finer level, then refine it using `Rebalance` and `FM`, and restart from the

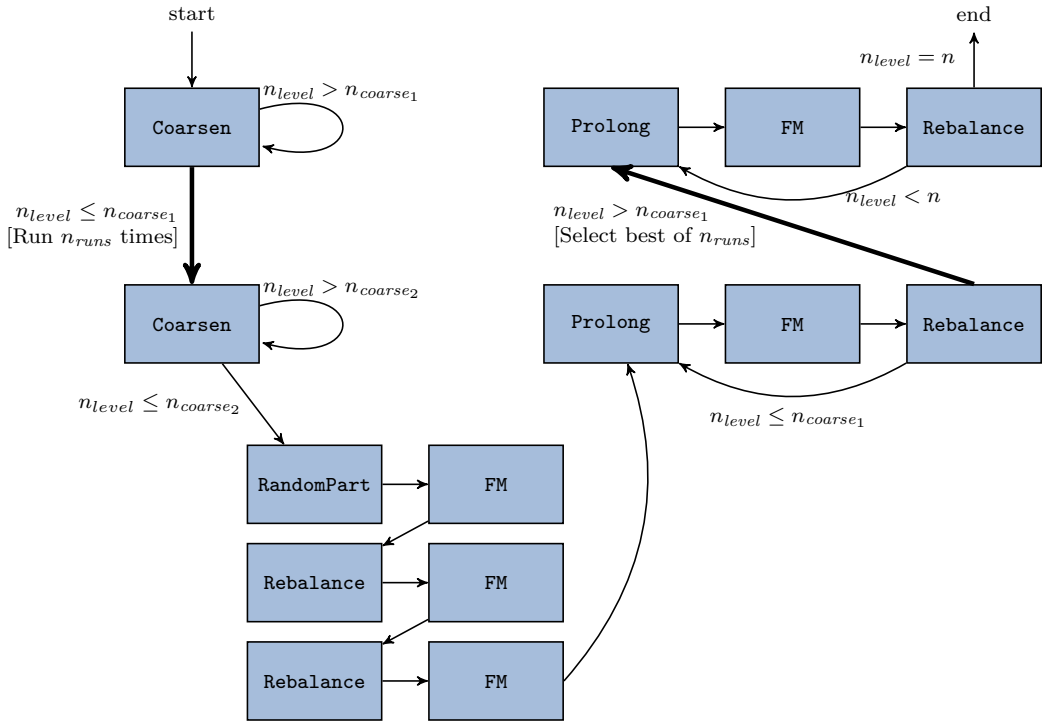


Figure A.1.1 – The default multilevel algorithm used by MeTiS

`Prolong` state as long as the number of vertices is smaller than n_{coarse_1}.

MeTiS thus computes 4 to 5 partitions of a coarsened graph. Then, the best partition is selected, and only this one is prolonged and refined until a partition of the original graph is obtained.

A.2 Bar Plots Showing the Discrepancy of Communication Cost of the Solutions Returned by Scotch-6.0.4, MeTiS-5.1.0 and PaToH-3.2

This section reports the distribution of the communication cost of the partitions returned by Scotch, MeTiS and PaToH. Bar plots were defined in Section 9.2 in order to show the discrepancy of the communication cost of the returned solutions. Figures A.2.1 on the next page to A.2.4 on page 242 report the results for each mesh (the meshes were defined in Section 9.1). On each page, the results for one tolerance are displayed on the same column, and the results for one partitioning tool on the same line.

As the objective is to minimize the communication cost, lower bars are better. Moreover, *the results can not be compared between partitioning tools*,

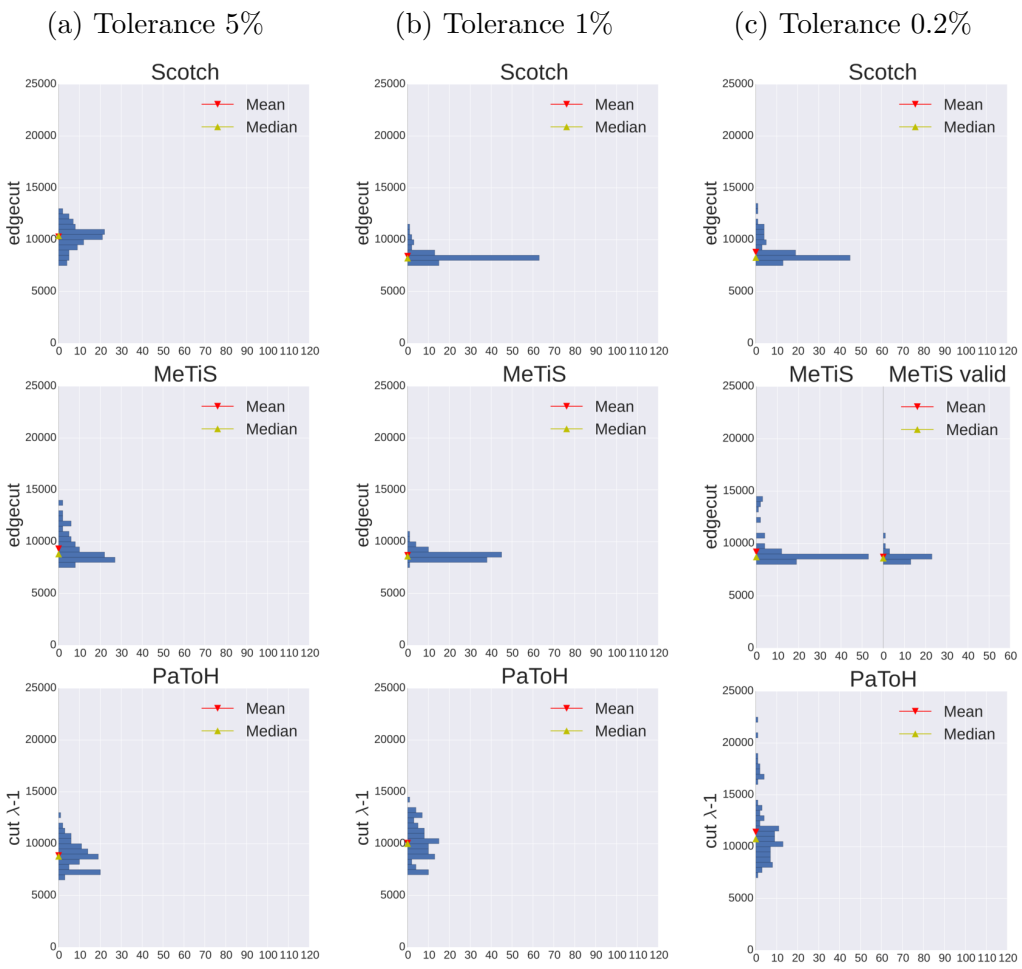


Figure A.2.1 – Bar plots of the communication cost of the 100 partitions of the mesh *Onera*

because the mono-criterion version of Scotch was used, and PaToH does not minimize the *edgecut* but the $cut_{\lambda-1}$. Whenever a tool did not return 100% of solutions over the 100 runs, a column "valid" displays the bar plots for the solutions.

Note that the discrepancy depends on the mesh, and is particularly visible for the meshes *Onera* and *Shock*. Moreover, for these two meshes, MeTiS and Scotch surprisingly obtain better results when the tolerance is tighter.

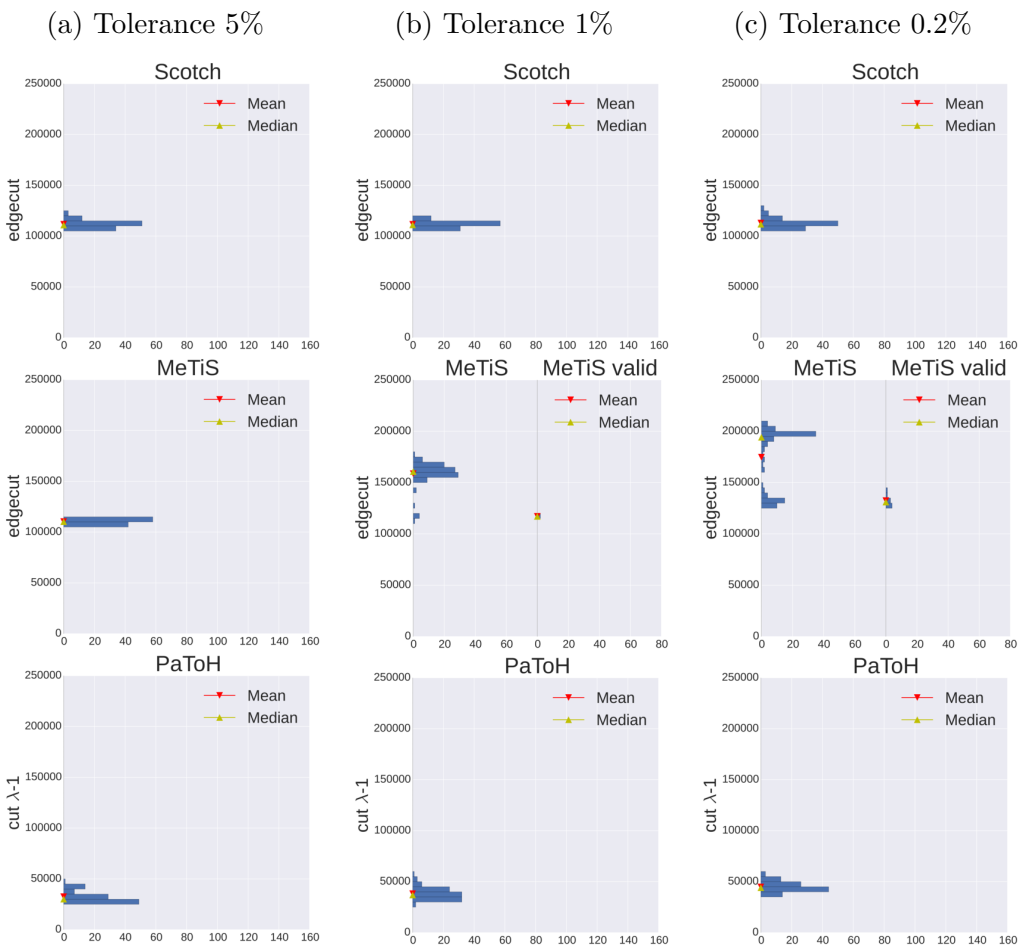


Figure A.2.2 – Bar plots of the communication cost of the 100 partitions of the mesh *Wave*

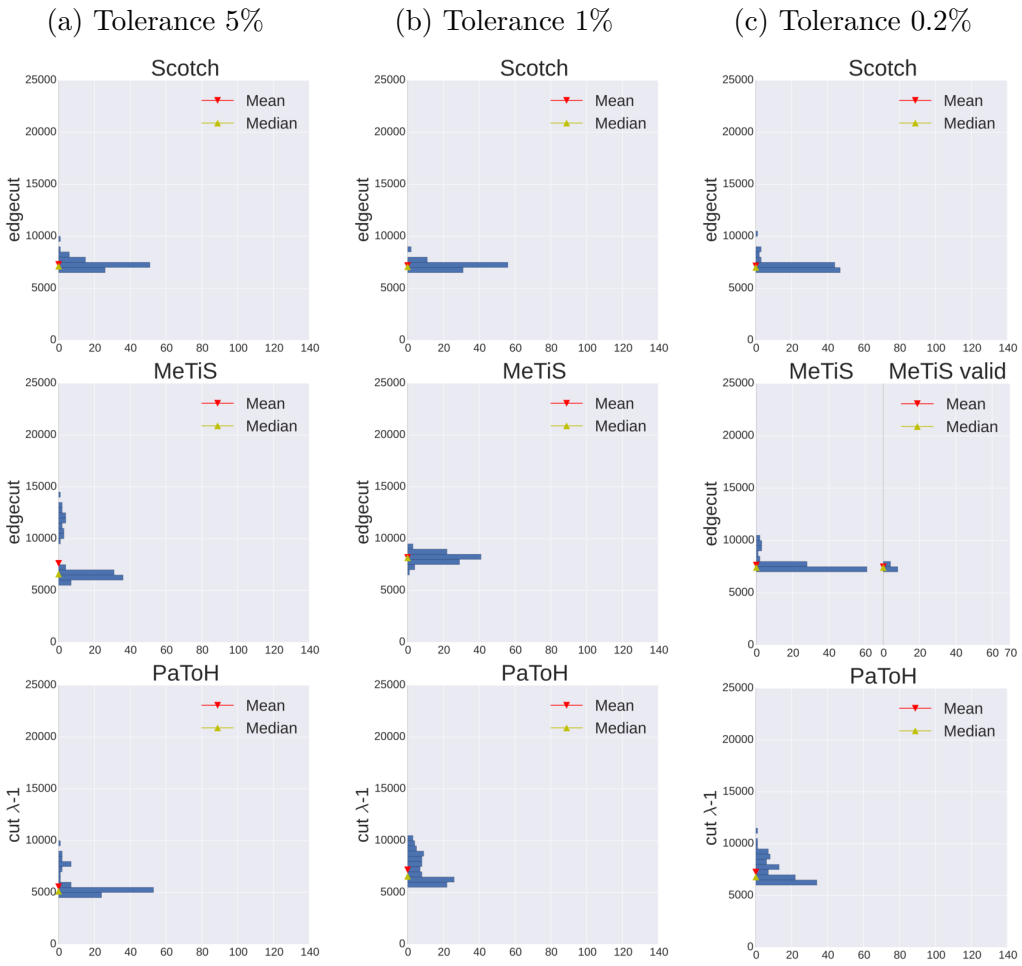


Figure A.2.3 – Bar plots of the communication cost of the 100 partitions of the mesh *Linkrodsok1*

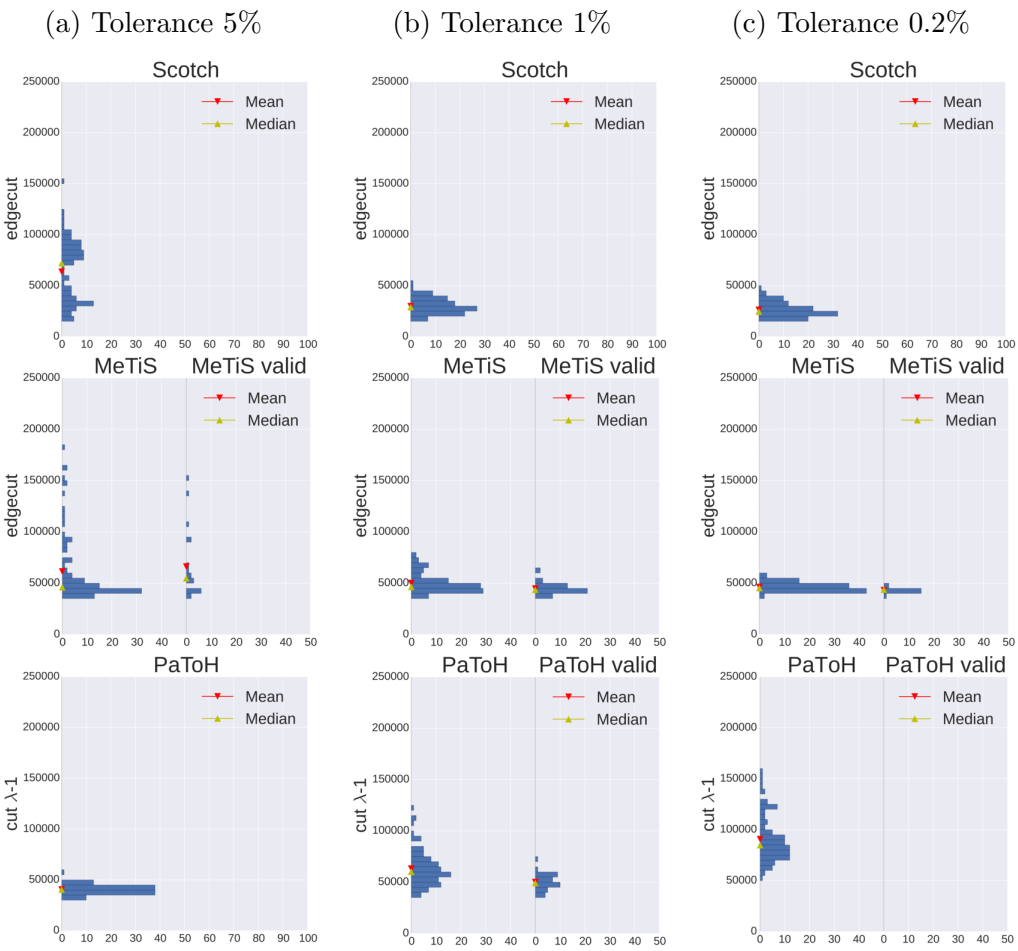


Figure A.2.4 – Bar plots of the communication cost of the 100 partitions of the mesh *Shock*

A.3 Imbalance at the Coarsest Level

Section 10.1.2 studied the imbalance of the partitions returned by each initial partitioning algorithm, for the coarsest graph originated from each restriction policy. The experiments did not include the *LMJ* mesh.

Figure A.3.1 on the next page uses the same representation as in Section 10.1.2: the *cumul* functions are given as areas, in order to study the results of each partitioning algorithm, independently from the `Restriction` policy.

Conclusions on the performance of the initial partitioning algorithms are quite similar for all instances. Thus, `VNBest` returns in general partitions of smallest imbalance, followed by `VNFirst` and then `GGG`.

Nevertheless, the variability to the `Restriction` policy differs between the instances (the space occupation of the area of one initial partitioning algorithm changes from one mesh to another).

Then, Figure A.3.2 on page 245 details the results of each initial partitioning algorithm, for each restriction policy, on each mesh. Without regrouping the `Restriction` policies, analysis of the *cumul* functions is more complex, because the curves intertwine. Nevertheless, in general, for a given initial partitioning algorithm, more restrictions lead to partitions of smaller imbalance.

Finally, Figure A.3.3 on page 246 displays the average imbalance of the partitions returned by each initial partitioning algorithms at the coarsest level. Then, Figures A.3.4 on page 247 to A.3.7 on page 248 display the average imbalance of the partitions returned by each initial partitioning algorithm, for each mesh (but *LMJ*), at different coarsened levels (which amounts to changing the value of n_{coarse}).

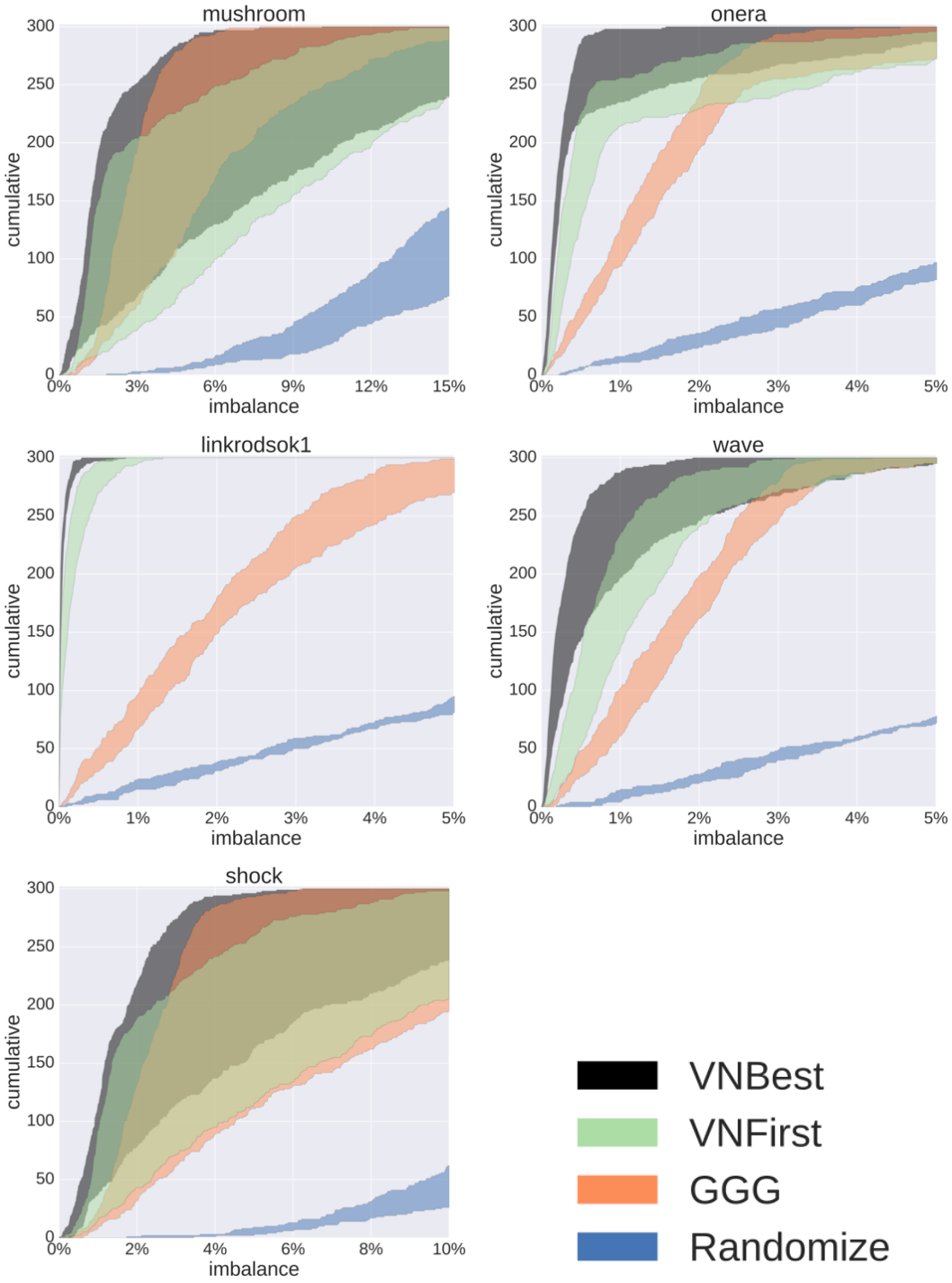


Figure A.3.1 – For a given algorithm, the *cumul* function introduced in Section 9.3.3 counts, given an imbalance x, the number of times that the algorithm returned a partition of imbalance smaller than or equal to x (therefore, left is better).
The figure represents the area in which lies the *cumul* function of each initial partitioning algorithm, when partitioning the coarsest graph produced by different `Restriction` policies.

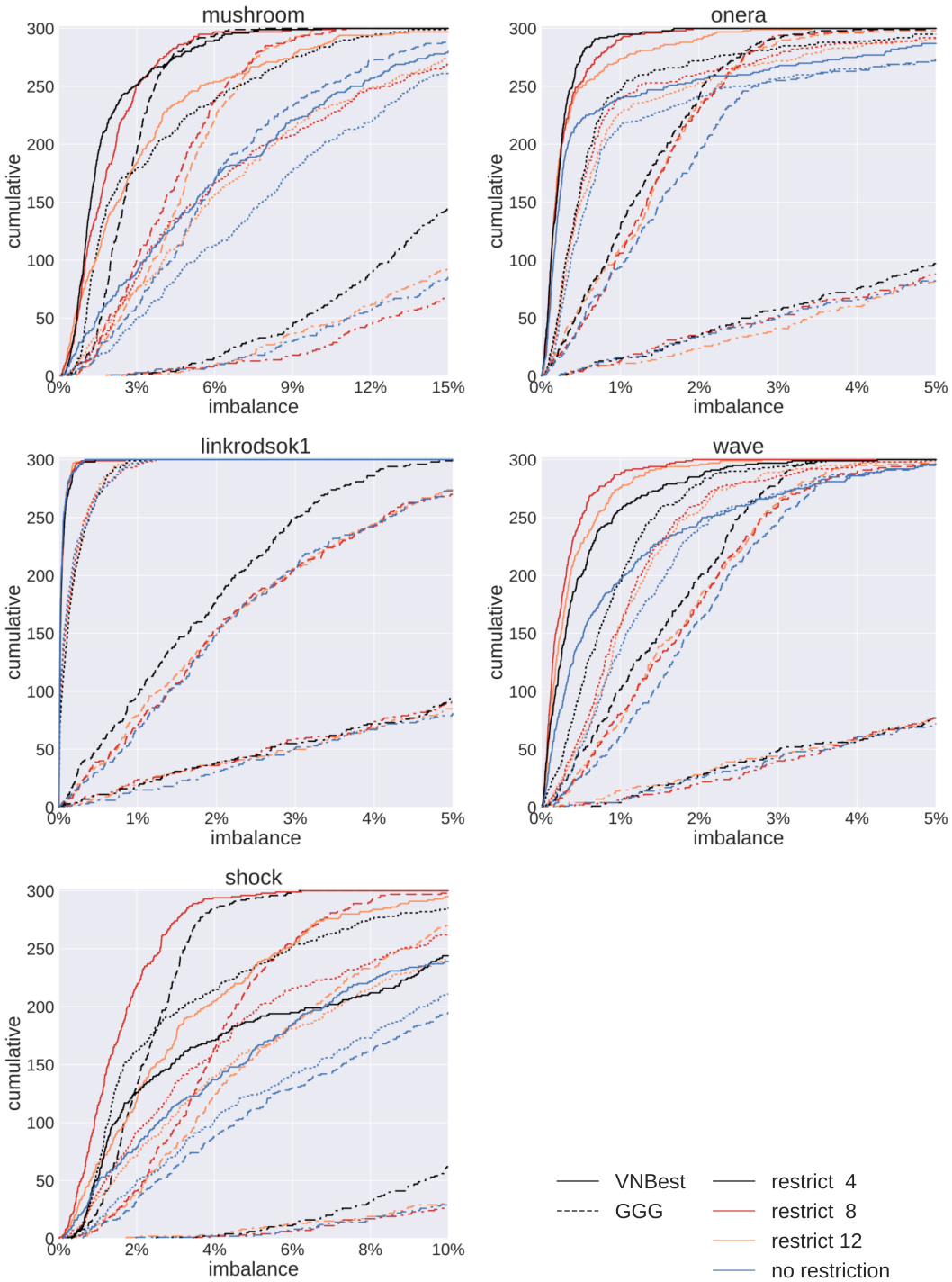


Figure A.3.2 – For a given algorithm, the *cumul* function introduced in Section 9.3.3 counts, given an imbalance x, the number of times that the algorithm returned a partition of imbalance smaller than or equal to x (Therefore, left is better).
Dash types indicate the initial partitioning algorithm used, and colors indicate the `Restriction` policy.

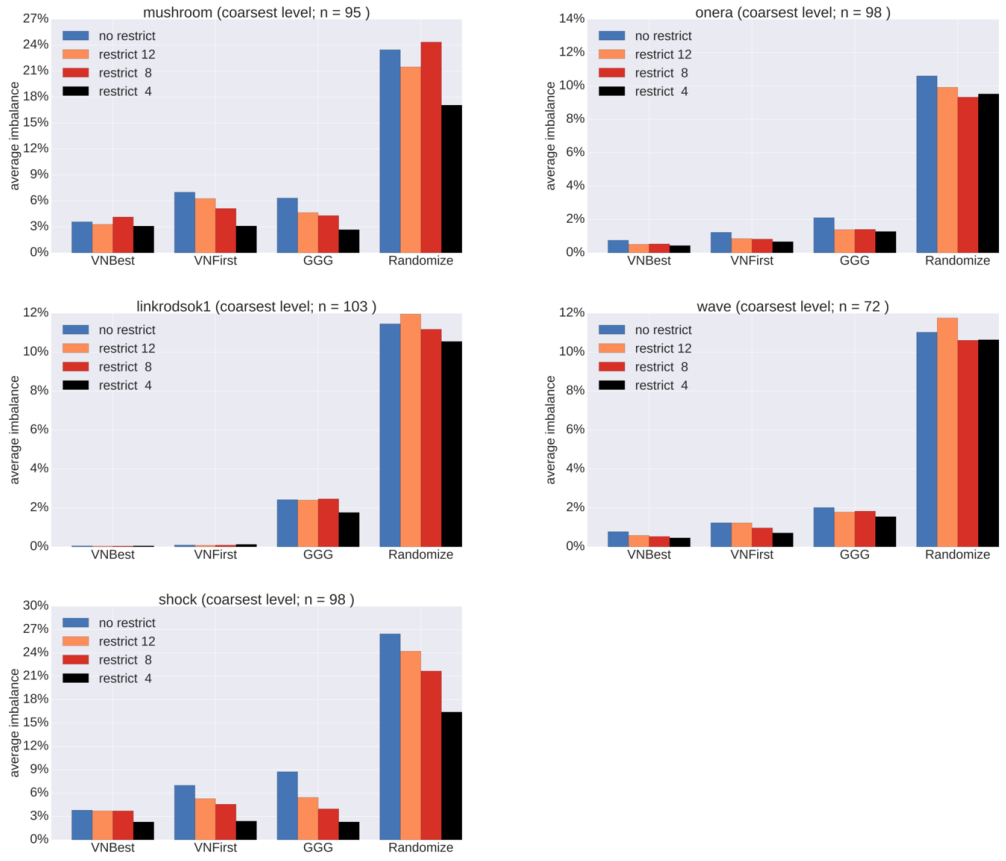


Figure A.3.3 – Averages of the imbalance of the partitions returned by each algorithm for each restriction policy and each initial partitioning algorithm, at the coarsest level

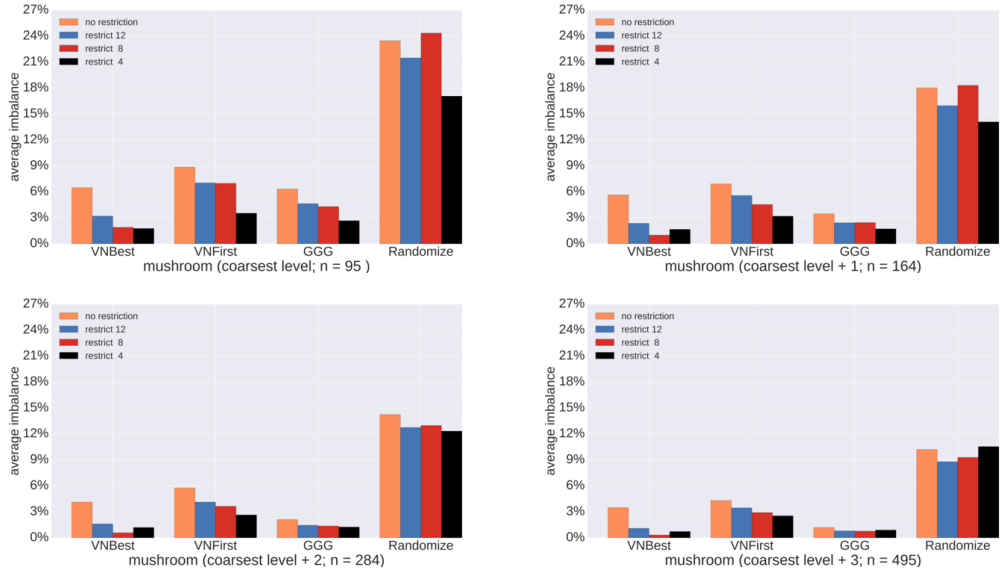


Figure A.3.4 – Average of the imbalance of the partitions returned for the mesh *Mushroom* at different coarse levels. The average was computed over 100 runs on each of the three weight distributions.

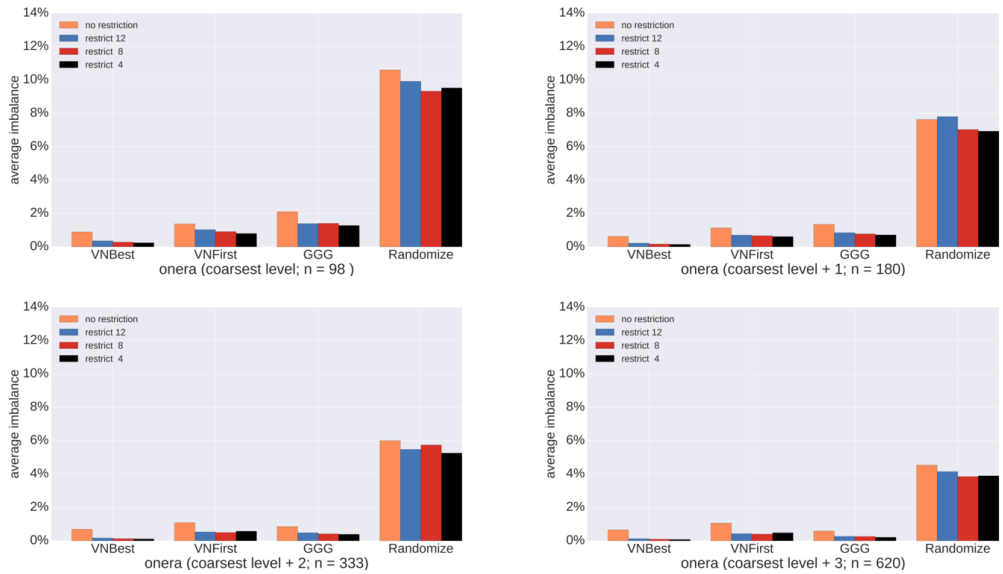


Figure A.3.5 – Average of the imbalance of the partitions returned for the mesh *Onera* at different coarse levels. The average was computed over 100 runs on each of the three weight distributions.

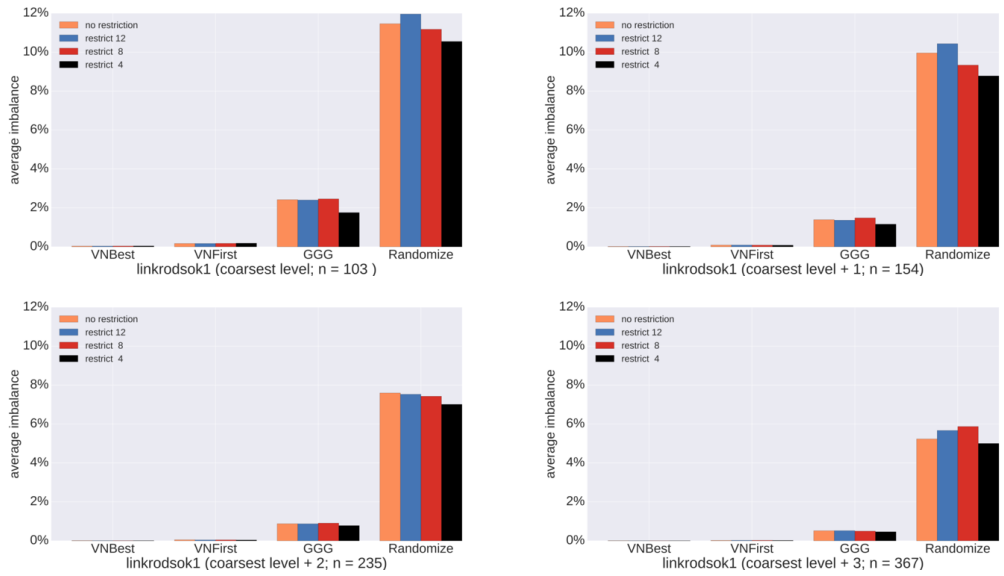


Figure A.3.6 – Average of the imbalance of the partitions returned for the mesh *Linkrodsok1* at different coarse levels. The average was computed over 100 runs on each of the three weight distributions.

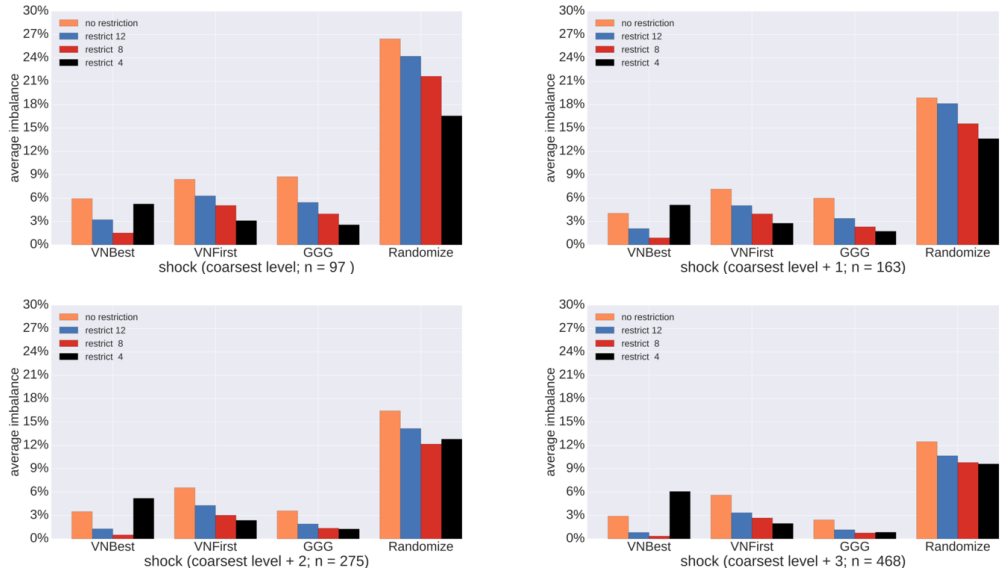


Figure A.3.7 – Average of the imbalance of the partitions returned for the mesh *Shock* at different coarse levels. The average was computed over 100 runs on each of the three weight distributions.

A.4 Implementation in Scotch and Comparison with Crack and Existing Multi-criteria Partitioning Tools

Figure A.4.1 on the following page displays the *cumul* functions for our multi-criteria Scotch implementation, for Crack, and for the partitioning tools MeTiS and PaToH, for the meshes *Mushroom*, *Linkrodsok1* and *Shock*. The results were discussed in Section 10.2.3.

A.5 Comparison of Ordering Schemes before Applying `HEM`

Figure A.5.1 on page 251 shows the impact of using different ordering schemes before applying the `HEM` algorithm, for the meshes *Mushroom*, *Onera* and *Linkrodsok1*. The results were commented in Section 10.3.

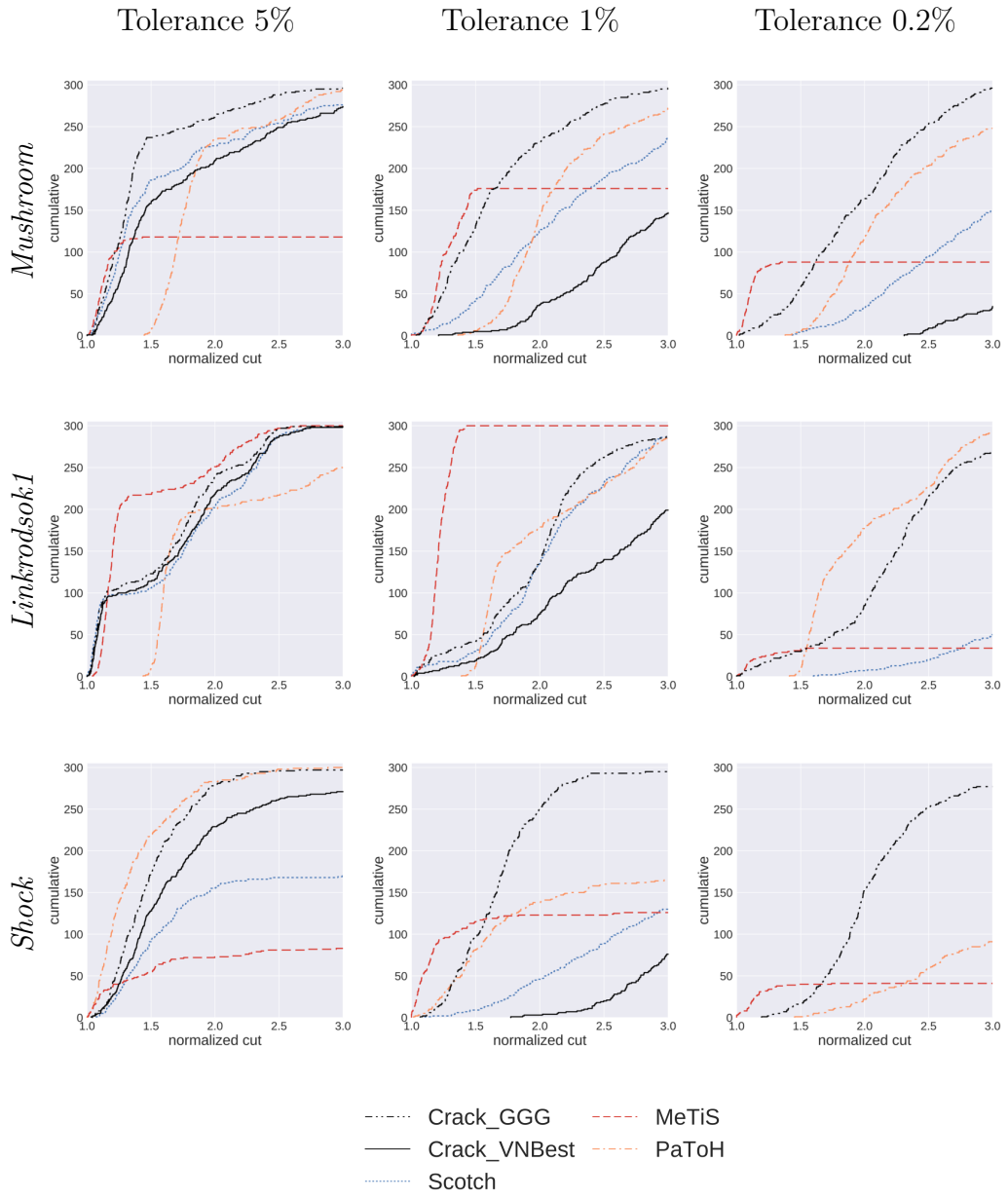


Figure A.4.1 – Comparison of Crack, Scotch, MeTiS and PaToH on the first three meshes, using the *cumul* function. For a given algorithm, $cumul(x)$ is the number of solutions of communication cost smaller than x returned by the algorithm. Since we aim at minimizing the communication cost, left is better. Each algorithm was run 100 times on each of the 3 weight distributions per mesh, and note that only the partitions that are valid are counted in the *cumul* function. The cut was normalized by the minimal communication cost found among all the algorithms.

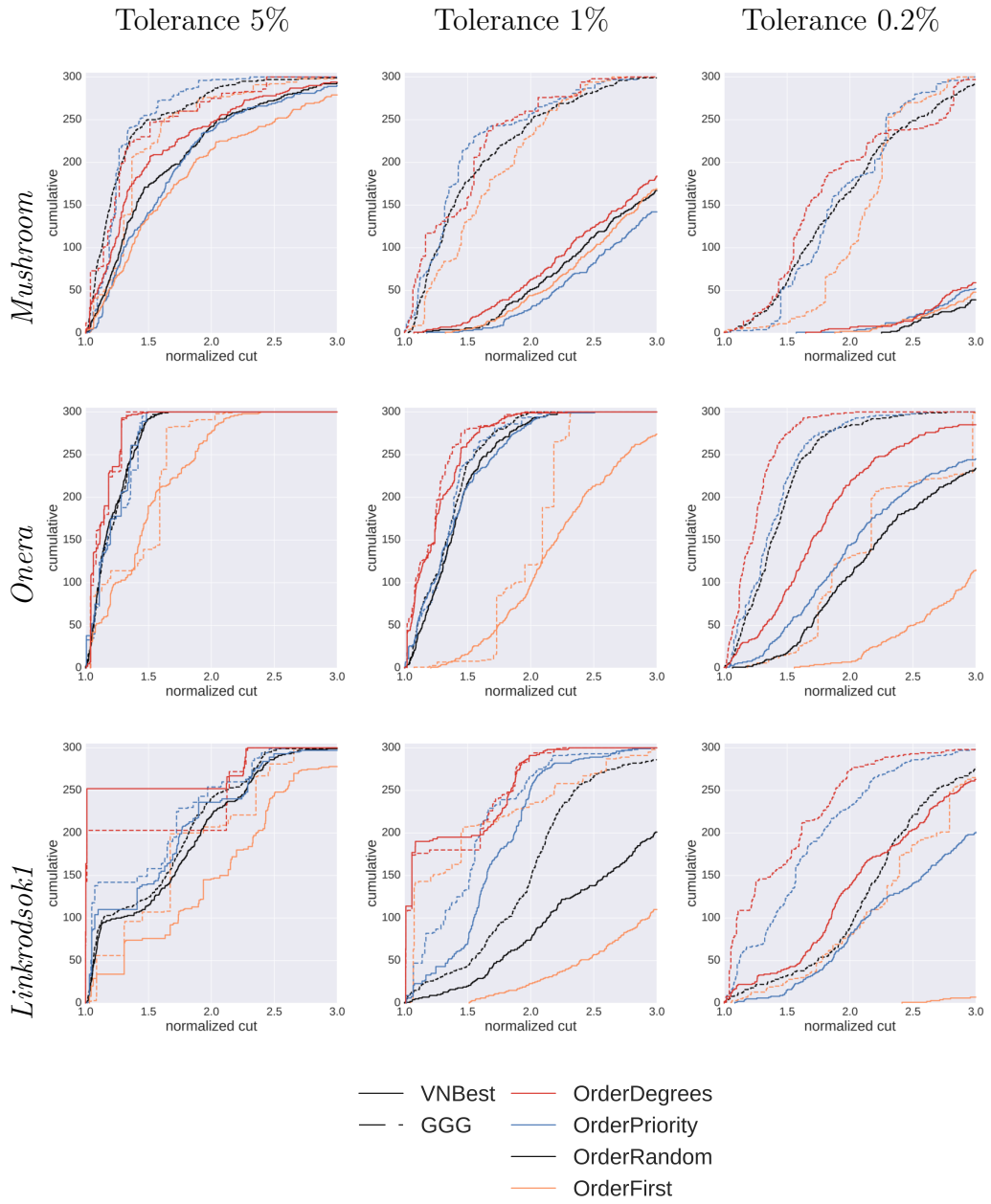


Figure A.5.1 – Comparison of different ordering policies (indicated by the different colors) before applying the `HEM` scheme, during the coarsening phase, using Crack. All algorithms use the same `Restrict 8` policy on the vertex weights, and either the `VNBest` algorithm (plain lines) or the `GGG` algorithm (dashed lines) for initial partitioning.
Each algorithm was run 100 times on each of the 3 weight distributions per mesh, and note that only the partitions that are valid are counted in the *cumul* function. The cut was normalized by the minimal communication cost found among all the algorithms.

Bibliography

2015. *MPI: A Message-passing Interface Standard, Version 3.1*.
URL http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

Alpert, Charles J. and Kahng, Andrew B., 1995. Recent directions in netlist partitioning: A survey. *Integr. VLSI J.*, 19(1-2):1–81. doi:10.1016/0167-9260(95)00008-4.
URL http://dx.doi.org/10.1016/0167-9260(95)00008-4

Aykanat, C., Cambazoglu, B. and Uçar, B., 2008. Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *J. Parallel Distrib. Comput.*, 68(5):609–625. doi:10.1016/j.jpdc.2007.09.006.
URL http://dx.doi.org/10.1016/j.jpdc.2007.09.006

Bader, David A., Meyerhenke, Henning, Sanders, Peter and Wagner, Dorothea, editors, 2013. *Graph Partitioning and Graph Clustering - 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, USA, February 13-14, 2012. Proceedings*, volume 588 of *Contemporary Mathematics*. American Mathematical Society. doi:10.1090/conm/588.
URL http://dx.doi.org/10.1090/conm/588

Barat, Rémi, Chevalier, Cédric and Pellegrini, François, 2016. Multi-constraints graph partitioning for load balancing of multi-physics simulations. In *Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS)*. Lorient, France.
URL https://hal.inria.fr/hal-01417532

Barnard, S. T. and Simon, H. D., 1994. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice and Experience*, 6(2):101–117.

Ben-Kiki, Oren, Evans, Clark and döt Net, Ingy, 2009. Yaml ain't markup language (yaml™) version 1.2.
URL http://www.yaml.org/spec/1.2/spec.html

Berger, M. J. and Bokhari, S. H., 1987. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, C-36(5):570–580. doi:10.1109/TC.1987.1676942.

Beyer, W. A. and Zardecki, Andrew, 2004. The early history of the ham sandwich theorem. *The American Mathematical Monthly*, 111(1):58–61. URL http://www.jstor.org/stable/4145019

Bichot, C.-H. and Siarry, P., 2010. *Partitionnement de graphe*. Lavoisier, Paris.

Bisseling, Rob H. and Meesen, Wouter, 2005. Communication balancing in parallel sparse matrix-vector multiplication. *ETNA. Electronic Transactions on Numerical Analysis [electronic only]*, 21:47–65. URL http://eudml.org/doc/128024

Blackburn, Steven, Diwan, Amer, Hauswirth, Matthias, Sweeney, Peter, Amaral, José Nelson, Babka, Vlastimil, Binder, Walter, Brecht, Tim, Bulej, Lubomír, Eeckhout, Lieven, Sebastian, Fisch, Frampton, Daniel, Garner, Robin, Georges, Andy, Hendren, Laurie, Hind, Michael, Hosking, Antony, Jones, Richard, Kalibera, Tomas, Moret, Philippe, Nystrom, Nathaniel, Pankratius, Victor and Tuma, Petr, 2012. Can you trust your experimental results? Technical report. URL http://sape.inf.usi.ch/sites/default/files/publication/EvaluateCollaboratoryTR1.pdf

Boman, Erik, Devine, Karen, Fisk, Lee Ann, Heaphy, Robert, Hendrickson, Bruce, Vaughan, Courtenay, Catalyurek, Umit, Bozdag, Doruk, Mitchell, William and Teresco, James, 2007. *Zoltan 3.0: Parallel Partitioning, Load-balancing, and Data Management Services; User's Guide*. Sandia National Laboratories, Albuquerque, NM. Tech. Report SAND2007-4748W http://www.cs.sandia.gov/Zoltan/ug_html/ug.html.

Bui, T.N. and Jones, C., 1993. *A heuristic for reducing fill-in in sparse matrix factorization*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA (United States).

Buluç, A, Meyerhenke, H., Safro, I., Sanders, P. and Schulz, C., 2015. Recent advances in graph partitioning. In *Algorithm Engineering: Selected Results and Surveys, LNCS 9220*. Springer-Verlag.

Cai, Xing and Bouhmala, Noureddine, 2007. A unified framework of multi-objective cost functions for partitioning unstructured finite element meshes. *Applied Mathematical Modelling*, 31(9):1711 – 1728. doi:https://doi.org/10.1016/j.apm.2006.06.007. URL http://www.sciencedirect.com/science/article/pii/S0307904X06001375

Catalyurek, U. and Aykanat, C., 1999. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans.*

Parallel Distrib. Syst., 10(7):673–693. doi:10.1109/71.780863.
URL http://dx.doi.org/10.1109/71.780863

Catalyurek, U. V. and Aykanat, C., 2011. Patoh: Partitioning tool for hypergraphs. User's manual.
URL https://hal.archives-ouvertes.fr/hal-00410327

Catalyurek, U. V., Boman, E. G., Devine, K. D., Bozdag, D., Heaphy, R. and Riesen, Lee Ann, 2007. Hypergraph-based dynamic load balancing for adaptive scientific computations. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–11. doi:10.1109/IPDPS.2007.370258.

Catalyurek, Umit and Aykanat, Cevdet, 1996. Decomposing irregularly sparse matrices for parallel matrix-vector multiplication. In *Proceedings of the Third International Workshop on Parallel Algorithms for Irregularly Structured Problems*, IRREGULAR '96, pages 75–86. Springer-Verlag, London, UK, UK. ISBN 3-540-61549-0.
URL http://dl.acm.org/citation.cfm?id=646010.676990

CEA, 2015. Le Laser Mégajoule. http://www-lmj.cea.fr/fr/lmj/index.htm. Accessed: 2017-11-08.

Chekuri, C. and Khanna, S., 1999. On multi-dimensional packing problems. In *Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '99, pages 185–194. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. ISBN 0-89871-434-6.
URL http://dl.acm.org/citation.cfm?id=314500.314555

Chevalier, C., Grospellier, G., Ledoux, F. and Weill, J.C., 2012. Load balancing for mesh based multi-physics simulations in the arcane framework. In *Proceedings of 8th International Conference on Engineering Computational Technology*, pages 47–62.

Chevalier, Cédric and Safro, Ilya, 2009. *Comparison of Coarsening Schemes for Multilevel Graph Partitioning*, pages 191–205. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-642-11169-3. doi:10.1007/978-3-642-11169-3_14.
URL http://dx.doi.org/10.1007/978-3-642-11169-3_14

Deveci, M., Kaya, K., Uçar, B. and Çatalyürek, Ü. V., 2015a. Fast and high quality topology-aware task mapping. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 197–206. doi:10.1109/IPDPS.2015.93.

Deveci, Mehmet, Kaya, Kamer, Uçar, Bora and Ümit V. Çatalyürek, 2015b. Hypergraph partitioning for multiple communication cost metrics: Model and methods. *Journal of Parallel and Distributed Computing*, 77:69 – 83. doi:http://dx.doi.org/10.1016/j.jpdc.2014.12.002.
URL http://www.sciencedirect.com/science/article/pii/S0743731514002275

Devine, K., Boman, E., Heaphy, R., Hendrickson, B. and Vaughan, C., 2002. Zoltan data management services for parallel dynamic applications. *Computing in Science Engineering*, 4(2):90–96. doi:10.1109/5992.988653.

Devine, K. D., Boman, E. G., Heaphy, R. T., Bisseling, R. H. and Catalyurek, U. V., 2006. Parallel hypergraph partitioning for scientific computing. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pages 10 pp.–. doi:10.1109/IPDPS.2006.1639359.

DIMACS, 2012. Graphs from Numerical Simulations. https://www.cc.gatech.edu/dimacs10/archive/numerical.shtml. Accessed: 2017-11-13.

Dolan, Elizabeth D. and Moré, Jorge J., 2002. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213. doi:10.1007/s101070100263.
URL https://doi.org/10.1007/s101070100263

Donath, W. E. and Hoffman, A. J., 1973. Lower bounds for the partitioning of graphs. *IBM Journal of Research and Development*, 17(5):420–425. doi:10.1147/rd.175.0420.

Fiduccia, C. M. and Mattheyses, R. M., 1982. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181. IEEE Press, Piscataway, NJ, USA. ISBN 0-89791-020-6.
URL http://dl.acm.org/citation.cfm?id=800263.809204

Fiedler, Miroslav, 1975. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, 25(4):619–633.
URL http://eudml.org/doc/12900

Forouzan, Behrouz A., 2007. *Data Communications and Networking*. McGraw-Hill, Inc., New York, NY, USA, 4 edition.

Fortmeier, O., Bücker, H.M., Fagginger Auer, B.O. and Bisseling, R.H., 2013. A new metric enabling an exact hypergraph model for the communication volume in distributed-memory parallel applications. *Parallel Computing*, 39(8):319 – 335. doi:https://doi.org/10.1016/j.parco.2013.05.003.

URL http://www.sciencedirect.com/science/article/pii/S0167819113000690

Frey, P.J. and George, P.L., 1999. *Maillages: applications aux éléments finis*. Hermès Science Publications. ISBN 9782746200241.
URL https://books.google.fr/books?id=RLJ9AAAACAAJ

Garey, M. R. and Johnson, D. S., 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA. ISBN 0716710447.

Garey, M. R., Johnson, D. S. and Stockmeyer, L., 1974. Some Simplified NP-Complete Problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, STOC '74, pages 47–63. ACM, New York, NY, USA. doi:10.1145/800119.803884.
URL http://doi.acm.org/10.1145/800119.803884

Gent, Ian P. and Walsh, Toby, 1998. Analysis of heuristics for number partitioning. *Computational Intelligence*, 14(3):430–451. doi:10.1111/0824-7935.00069.
URL http://dx.doi.org/10.1111/0824-7935.00069

George, Alan and Liu, Joseph W., 1981. *Computer Solution of Large Sparse Positive Definite*. Prentice Hall Professional Technical Reference. ISBN 0131652745.

Goehring, T. and Saad, Y., 1995. Heuristic algorithms for automatic graph partitioning.

Hager, W. W. and Krylyuk, Y., 1999. Graph partitioning and continuous quadratic programming. *SIAM J. Discret. Math.*, 12(4):500–523. doi:10.1137/S0895480199335829.
URL http://dx.doi.org/10.1137/S0895480199335829

Hendrickson, B., 1998. Graph partitioning and parallel solvers: Has the emperor no clothes? In *In Proc. Irregular'98*, pages 218–225. Springer-Verlag.

Hendrickson, B. and Kolda, T. G., 2000. Graph partitioning models for parallel computing. *Parallel Comput.*, 26(12):1519–1534. doi:10.1016/S0167-8191(00)00048-X.
URL http://dx.doi.org/10.1016/S0167-8191(00)00048-X

Hendrickson, B. and Leland, R., 1995. A multi-level algorithm for partitioning graphs.

HOLTGREWE, Manuel, SANDERS, Peter and SCHULZ, Christian, 2009. Engineering a scalable high quality graph partitioner. *CoRR*, abs/0910.2004.
URL http://arxiv.org/abs/0910.2004

HOROWITZ, E. and SAHNI, S., 1978. *Fundamentals of Computer Algorithms*. Computer software engineering series. Computer Science Press. ISBN 9783540120353.
URL https://books.google.fr/books?id=k1p0AAAACAAJ

HOROWITZ, Ellis and SAHNI, Sartaj, 1974. Computing partitions with applications to the knapsack problem. *J. ACM*, 21(2):277–292. doi:10.1145/321812.321823.
URL http://doi.acm.org/10.1145/321812.321823

HYAFIL, L. and RIVEST, R. L., 1973. Graph partitioning and constructing optimal decision trees are polynomial complete problems. Technical report Rapport de Recherche no. 33, IRIA – Laboratoire de Recherche en Informatique et Automatique.

HYAFIL, Laurent and RIVEST, Ronald L., 1976. Constructing optimal binary decision trees is np-complete. *Information Processing Letters*, 5(1):15 – 17. doi:http://dx.doi.org/10.1016/0020-0190(76)90095-8.
URL http://www.sciencedirect.com/science/article/pii/0020019076900958

JAIN, Sachin, SWAMY, Chaitanya and BALAJI, K, 1998. Greedy algorithms for k-way graph partitioning. In *the 6th international conference on advanced computing*.

KALOS, Malvin H. and WHITLOCK, Paula A., 2009. *What is Monte Carlo?*, pages 1–5. Wiley-VCH Verlag GmbH & Co. KGaA. ISBN 9783527626212. doi:10.1002/9783527626212.ch1.
URL http://dx.doi.org/10.1002/9783527626212.ch1

KARMARKAR, Narenda and KARP, Richard M., 1983. The differencing method of set partitioning. Technical report, Berkeley, CA, USA.

KARYPIS, G., 2003. Multi-constraint mesh partitioning for contact/impact computations. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 56–. ACM, New York, NY, USA. ISBN 1-58113-695-1. doi:10.1145/1048935.1050206.
URL http://doi.acm.org/10.1145/1048935.1050206

KARYPIS, G., 2013. *METIS - A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*.

KARYPIS, G., AGGARWAL, R., KUMAR, V. and SHEKHAR, S., 1997. Multilevel Hypergraph Partitioning: Application in VLSI Domain. In *Proceedings of the 34th Annual Design Automation Conference*, DAC '97, pages 526–529. ACM, New York, NY, USA. ISBN 0-89791-920-3. doi:10.1145/266021.266273.
URL http://doi.acm.org/10.1145/266021.266273

KARYPIS, G. and KUMAR, V., 1998a. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392. doi:10.1137/S1064827595287997.
URL http://dx.doi.org/10.1137/S1064827595287997

KARYPIS, G. and KUMAR, V., 1998b. Multilevel algorithms for multi-constraint graph partitioning. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–13. IEEE Computer Society, Washington, DC, USA. ISBN 0-89791-984-X.
URL http://dl.acm.org/citation.cfm?id=509058.509086

KARYPIS, G. and KUMAR, V., 1998c. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *J. Parallel Distrib. Comput.*, 48(1):71–95. doi:10.1006/jpdc.1997.1403.
URL http://dx.doi.org/10.1006/jpdc.1997.1403

KARYPIS, George and KUMAR, Vipin, 1995. Analysis of multilevel graph partitioning. In *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, Supercomputing '95. ACM, New York, NY, USA. ISBN 0-89791-816-9. doi:10.1145/224170.224229.
URL http://doi.acm.org/10.1145/224170.224229

KARYPIS, George and KUMAR, Vipin, 1998d. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel Distrib. Comput.*, 48(1):96–129. doi:10.1006/jpdc.1997.1404.
URL http://dx.doi.org/10.1006/jpdc.1997.1404

KELLERER, Hans, PFERSCHY, Ulrich and PISINGER, David, 2004. *Knapsack problems*. Springer. ISBN 978-3-540-40286-2.

KERNIGHAN, B. W. and LIN, S., 1970. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307. doi:10.1002/j.1538-7305.1970.tb01770.x.

KIRKPATRICK, S., GELATT, C. D. and VECCHI, M. P., 1983. Optimization by simulated annealing. *SCIENCE*, 220(4598):671–680.

KOJIĆ, Jelena, 2010. Integer linear programming model for multidimensional two-way number partitioning problem. *Computers & Mathematics with Applications*, 60(8):2302 – 2308. doi:https://doi.org/10.1016/j.camwa.2010.08.024.

URL http://www.sciencedirect.com/science/article/pii/S0898122110005882

Korf, Richard E., 1995. From approximate to optimal solutions: A case study of number partitioning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'95, pages 266–272. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1-55860-363-8, 978-1-558-60363-9.
URL http://dl.acm.org/citation.cfm?id=1625855.1625890

Korf, Richard E., 1998. A complete anytime algorithm for number partitioning. *Artificial Intelligence*, 106(2):181 – 203. doi:http://dx.doi.org/10.1016/S0004-3702(98)00086-1.
URL http://www.sciencedirect.com/science/article/pii/S0004370298000861

Korf, Richard E., 2009. Multi-way number partitioning. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pages 538–543. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.
URL http://dl.acm.org/citation.cfm?id=1661445.1661531

Korf, Richard E. and Schreiber, Ethan L., 2013. Optimally scheduling small numbers of identical parallel machines. In *ICAPS*. AAAI.

Kratica, Jozef, Kojić, Jelena and Savić, Aleksandar, 2014. Two metaheuristic approaches for solving multidimensional two-way number partitioning problem. *Computers & Operations Research*, 46(Supplement C):59 – 68. doi:https://doi.org/10.1016/j.cor.2014.01.003.
URL http://www.sciencedirect.com/science/article/pii/S0305054814000045

LaSalle, Dominique and Karypis, George, 2016. A parallel hill-climbing refinement algorithm for graph partitioning. *2016 45th International Conference on Parallel Processing (ICPP)*, 00:236–241. doi:doi.ieeecomputersociety.org/10.1109/ICPP.2016.34.

Leinberger, W., Karypis, G. and Kumar, V., 1999. Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints. In *Parallel Processing, 1999. Proceedings. 1999 International Conference on*, pages 404–412. doi:10.1109/ICPP.1999.797428.

Malan, Katherine and Engelbrecht, Andries Petrus, 2013. A survey of techniques for characterising fitness landscapes and some possible ways forward. *Inf. Sci.*, 241:148–163.
URL http://dblp.uni-trier.de/db/journals/isci/isci241.html#MalanE13

MERTENS, S, 2003. The easiest hard problem: Number partitioning, inst. f. *Theor. Physik, University of Magdeburg, Magdeburg, Germany*.

MICHIELS, Wil, KORST, Jan H. M., AARTS, Emile H. L. and VAN LEEUWEN, Jan, 2003. Performance ratios for the differencing method applied to the balanced number partitioning problem. In *STACS*, volume 2607 of *Lecture Notes in Computer Science*, pages 583–595. Springer.

MORAIS, Sébastien, 2016. *Study and obtention of exact, and approximation, algorithms and heuristics for a mesh partitioning problem under memory constraints*. Theses, Université Paris Saclay.
URL https://hal.archives-ouvertes.fr/tel-01447665

OUYANG, Min, TOULOUSE, Michel, THULASIRAMAN, Krishnaiyan, GLOVER, Fred and DEOGUN, Jitender S., 2000. Multilevel cooperative search: Application to the circuit/hypergraph partitioning problem. In *Proceedings of the 2000 International Symposium on Physical Design*, ISPD '00, pages 192–198. ACM, New York, NY, USA. ISBN 1-58113-191-7. doi:10.1145/332357.332399.
URL http://doi.acm.org/10.1145/332357.332399

PELLEGRINI, F., 2008. Scotch and libScotch 5.1 User's Guide. User's manual.
URL https://hal.archives-ouvertes.fr/hal-00410327

PELLEGRINI, François, 2007. A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. In T. Priol A.-M. Kermarrec, L. Bougé, editor, *EuroPar*, volume 4641 of *Lecture Notes in Computer Science*, pages 195–204. Springer, Rennes, France. doi:10.1007/978-3-540-74466-5_22.
URL https://hal.archives-ouvertes.fr/hal-00301427

PELLEGRINI, François and ROMAN, Jean, 1996a. *Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs*, pages 493–498. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-540-49955-8. doi:10.1007/3-540-61142-8_588.
URL https://doi.org/10.1007/3-540-61142-8_588

PELLEGRINI, François and ROMAN, Jean, 1996b. Experimental analysis of the dual recursive bipartitioning algorithm for static mapping. Technical report, TR 1038-96, LaBRI, URA CNRS 1304, Univ. Bordeaux I.

PELT, D. M. and BISSELING, R. H., 2014. A medium-grain method for fast 2d bipartitioning of sparse matrices. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 529–539. doi:10.1109/IPDPS.2014.62.

Pilkington, John R., Pilkington, John R., Baden, Scott B. and Baden, Scott B., 1994. Partitioning with spacefilling curves. Technical report.

Pop, P. C. and Matei, O., 2013. *A Genetic Algorithm Approach for the Multidimensional Two-Way Number Partitioning Problem*, pages 81–86. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-642-44973-4. doi:10.1007/978-3-642-44973-4_10.
URL https://doi.org/10.1007/978-3-642-44973-4_10

Richter, Hendrik and Engelbrecht, Andries, 2013. *Recent Advances in the Theory and Application of Fitness Landscapes*. Springer Publishing Company, Incorporated. ISBN 3642418872, 9783642418877.

Rodríguez, Francisco J., Glover, Fred, García-Martínez, Carlos, Martí, Rafael and Lozano, Manuel, 2017. GRASP with exterior path-relinking and restricted local search for the multidimensional two-way number partitioning problem. *Computers & OR*, 78:243–254.

Ruml, W., Ngo, J. T., Marks, J. and Shieber, S. M., 1996. Easily searched encodings for number partitioning. *Journal of Optimization Theory and Applications*, 89(2):251–291. doi:10.1007/BF02192530.
URL http://dx.doi.org/10.1007/BF02192530

Sanchis, L. A., 1989. Multiple-way network partitioning. *IEEE Trans. Comput.*, 38(1):62–81. doi:10.1109/12.8730.
URL http://dx.doi.org/10.1109/12.8730

Sanders, Peter and Schulz, Christian, 2010. Engineering multilevel graph partitioning algorithms. *CoRR*, abs/1012.0006.
URL http://arxiv.org/abs/1012.0006

Schlag, Sebastian, Henne, Vitali, Heuer, Tobias, Meyerhenke, Henning, Sanders, Peter and Schulz, Christian, 2015. k-way hypergraph partitioning via n-level recursive bisection. *CoRR*, abs/1511.03137.
URL http://arxiv.org/abs/1511.03137

Schreiber, Ethan L., 2014. *Optimal Multi-Way Number Partitioning*. PhD. Thesis.
URL http://escholarship.org/uc/item/30g6n09q

Schroeppel, Richard and Shamir, Adi, 1981. A t=o(2n/2), s=o(2n/4) algorithm for certain np-complete problems. *SIAM J. Comput.*, 10:456–464.

Selvakkumaran, N. and Karypis, G., 2006. Multiobjective hypergraph-partitioning algorithms for cut and maximum subdomain-degree minimization. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 25(3):504–517. doi:10.1109/TCAD.

2005.854637.
URL http://dx.doi.org/10.1109/TCAD.2005.854637

Shi, Jianbo and Malik, J., 1998. Motion segmentation and tracking using normalized cuts. In *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*, pages 1154–1160. doi:10.1109/ICCV.1998.710861.

Simon, Horst D. and Teng, Shang-Hua, 1997. How good is recursive bisection? *SIAM Journal on Scientific Computing*, 18(5):1436–1445. doi:10.1137/S1064827593255135.
URL https://doi.org/10.1137/S1064827593255135

Slininger, Brian, 2013. Fiedler's theory of spectral graph partitioning.

Soper, A.J., Walshaw, C. and Cross, M., 2004. A combined evolutionary search and multilevel optimisation approach to graph-partitioning. *Journal of Global Optimization*, 29(2):225–241. doi:10.1023/B:JOGO.0000042115.44455.f3.
URL https://doi.org/10.1023/B:JOGO.0000042115.44455.f3

van Driessche, R. and Roose, D., 1994. Dynamic load balancing with an improved spectral bisection algorithm. In *Proceedings of SHPCC'94, Knoxville*, pages 494–500. IEEE.

von Luxburg, Ulrike, 2007. A tutorial on spectral clustering. *CoRR*, abs/0711.0189.
URL http://arxiv.org/abs/0711.0189

Walshaw, C., Cross, M. and McManus, K., 2000. Multiphase mesh partitioning. *Applied Mathematical Modelling*, 25(2):123 – 140. doi:http://dx.doi.org/10.1016/S0307-904X(00)00041-X.
URL http://www.sciencedirect.com/science/article/pii/S0307904X0000041X

Walshaw, Chris, 2004. Multilevel refinement for combinatorial optimisation problems. *Annals of Operations Research*, 131(1):325–372. doi:10.1023/B:ANOR.0000039525.80601.15.
URL https://doi.org/10.1023/B:ANOR.0000039525.80601.15

Wei, Y. C. and Cheng, C. K., 2006. Ratio cut partitioning for hierarchical designs. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 10(7):911–921. doi:10.1109/43.87601.
URL http://dx.doi.org/10.1109/43.87601

Wei, Yen-Chuen and Cheng, Chung-Kuan, 1989. Towards efficient hierarchical designs by ratio cut partitioning. In *1989 IEEE International Conference on Computer-Aided Design. Digest of Technical Papers*, pages 298–301. doi:10.1109/ICCAD.1989.76957.

Williams, Roy D., 1991. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency: Practice and Experience*, 3(5):457–481. doi:10.1002/cpe.4330030502.
URL http://dx.doi.org/10.1002/cpe.4330030502

Yakir, Benjamin, 1996. The Differencing Algorithm LDM for Partitioning: A Proof of a Conjecture of Karmarkar and Karp. *Math. Oper. Res.*, 21(1):85–99. doi:10.1287/moor.21.1.85.
URL http://dx.doi.org/10.1287/moor.21.1.85