HAL
open science

Etude et obtention d'heuristiques et d'algorithmes exacts et approchés pour un problème de partitionnement de maillage sous contraintes mémoire

Sébastien Morais

▶ **To cite this version:**

Sébastien Morais. Etude et obtention d'heuristiques et d'algorithmes exacts et approchés pour un problème de partitionnement de maillage sous contraintes mémoire. Recherche opérationnelle [math.OC]. Université Paris Saclay, 2016. Français. NNT : 2016SACLE038 . tel-01447665

HAL Id: tel-01447665

https://hal.science/tel-01447665v1

Submitted on 27 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

NNT : 2016SACLE038

Thèse de doctorat de l'Université Paris-Saclay préparée à l'Université d'Évry-Val-d'Essonne

Ecole doctorale n°580
Sciences et Technologies de l'Information et de la Communication
Spécialité de doctorat : Informatique

par

M. Sébastien Morais

Étude et obtention d'heuristiques et d'algorithmes exacts et approchés pour un problème de partitionnement de maillage sous contraintes mémoire

Thèse présentée et soutenue à Université d'Évry-Val-d'Essonne, le 23 Novembre 2016.

Composition du Jury :

M.	Christoph Dürr	Directeur de recherche Université Pierre et Marie Curie	(Rapporteur)
M.	François Pellegrini	Professeur Université de Bordeaux	(Rapporteur)
M.	Jean-Christophe Janodet	Professeur Université d'Évry-Val-d'Essonne	(Examinateur)
Mme	Cristina Bazgan	Professeur Université Paris Dauphine	(Présidente du jury)
M.	Ioan Todinca	Professeur Université d'Orléans	(Examinateur)
M.	Eric Angel	Professeur Université d'Évry-Val-d'Essonne	(Directeur de thèse)
M.	Franck Ledoux	Ingénieur chercheur CEA Université d'Évry-Val-d'Essonne	(Co-encadrant de thèse)
M.	Damien Regnault	Maître de conférences Université d'Évry-Val-d'Essonne	(Co-encadrant de thèse)

Remerciements

"En gros, tu ranges des puzzles quoi..."

Romain Bernazzani

Je tiens tout d'abord à remercier les personnes qui ont accepté de rapporter mon manuscrit, Christoph Dürr et François Pellegrini. Je les remercie de leurs remarques judicieuses concernant la pré-version de ce document ainsi que d'avoir été présents lors de ma soutenance. Merci aussi à Cristina Bazgan, Jean-Christophe Janodet et Ioan Todinca d'avoir accepté de participer à mon jury en qualité de présidente du jury et examinateurs.

Ensuite, j'adresse mes remerciements à mon directeur de thèse Eric Angel ainsi qu'à mon encadrant universitaire Damien Regnault. Les discussions que j'ai pu avoir avec eux m'ont grandement ouvert à la culture scientifique et m'ont permis de mieux cerner le monde de la recherche. Je vous remercie pour votre suivi, vos conseils et votre soutien durant ma thèse.

Je tiens aussi à remercier mon encadrant au CEA, Franck Ledoux, pour un nombre de raisons qu'il me paraît difficile d'écrire de manière exhaustive ; parmi cette liste je tiens malgré tout à insister sur tes qualités humaines, ta disponibilité (et cela même avec les 9 000 km de distance pendant une année complète), ta pédagogie,...

Bien que Cédric Chevalier n'apparaisse pas officiellement comme un encadrant de ma thèse, il a été une personne avec qui j'ai énormément travaillé au CEA et qui m'a beaucoup apporté. Merci à toi de m'avoir fait découvrir le monde du partitionnement et de t'être intéressé à mes travaux (même si je ne prenais pas en compte les coûts de communication) !

Pour leur aide pendant mes galères matérielles et administratives, je tiens à remercier Murielle, Pauline, Brigitte, Isabelle V, Isabelle B, Eliane, Thao et le meilleur pour la fin Denis !

Je remercie également tous mes collègues doctorants et stagiaires pour avoir égayé mes journées au CEA et à l'Université. L'atmosphère était parfois pimentée mais cela ne m'a jamais dérangé ; j'adore manger épicé !

Merci donc aux présents Tiégo, Hoby Baby, Rémix, Thomas, Bhugo, Thugo, Arthur, Audrey, aux nouveaux bébés thésards Éloïse et Théo *a.k.a. MC PtiDej* ; aux anciens Javier, Julien mon frère de barrière, Manue, Antonio, Leeroy, Bertrand, DJ Pad, Frédéric, Vincent, Amélie, Yves-Stan, Romain ; et à mes stagiaires préférés Al Baby, Super Saiyan à moustache, Al Exandre et Kiki !

Merci aux bons compagnons, Betty, Bichon, Chaton, Jéjé, Pti Houblon, So et Tud qui ont toujours su me remonter le moral, et cela surtout pendant ma période de rédaction.

Merci aussi à Antoine B, et au meilleur des officiers Romain B, ces trois années n'auraient pas été les mêmes sans vous !

Je suis très reconnaissant à ma famille qui a toujours su me soutenir. Je remercie tout particulièrement mes parents, mon petit frère et ma grand-mère. J'avais hâte d'annoncer la fin de mes études à ma grand-mère ainsi que les résultats de mon étude. Au lieu de cela, terminer cette thèse est un autre rappel douloureux qu'elle n'est plus avec nous. Elle me manque beaucoup.

Enfin, merci à Soraya ; je n'aurais jamais rêvé mieux que de te rencontrer. Tu as su me supporter, m'écouter et m'encourager pendant les moments difficiles et stressants de ma dernière année de thèse. Merci de m'avoir accompagné pendant cette aventure ; je te suis infiniment redevable.

Table des matières

Introduction **7**

1 Définitions et notions générales **11**

1.1 Vocabulaire et notations 11

1.2 Définitions sur les maillages 12

1.3 Définitions sur les graphes 14

2 Formulation et analyse **19**

2.1 Introduction et formalisation des problèmes 20

2.1.1 Partitionnement de graphe 21

2.1.2 Partitionnement d'hypergraphe 24

2.1.3 Partitionnement à séparateur sommets 27

2.2 Partitionnement de maillage sous contraintes mémoire 29

2.3 Programmes linéaires et quadratiques 33

2.3.1 Partitionnement de maillage sous contraintes mémoire 33

2.3.2 k-partitionnement de graphe 34

2.3.3 k-Partitionnement d'hypergraphe 36

2.3.4 Partitionnement à séparateur sommets 37

2.4 Études comparative des problèmes de partitionnement 38

2.4.1 Exemple d'une répartition *pic* 40

2.4.2 Exemple avec un voisinage à distance deux 47

2.4.3 Exemple d'une répartition *unitaire* 48

2.4.4 Exemple d'une répartition *aléatoire* 49

2.5 Analyses et conclusion 51

3 Recherche opérationnelle et ordonnancement **55**

3.1 Problèmes d'ordonnancement 56

3.1.1 Notation pour les problèmes d'ordonnancement 56

3.1.2 Relation entre le problème de partitionnement de maillage sous contraintes mémoire et certains problèmes d'ordonnancement 59

3.1.3 Méthodes de résolution et résultats théoriques 61

3.2 Algorithme approché FPT pour $Rm|G, mem|C_{max}$ 63

3.2.1 Définitions 63

3.2.2 Algorithme exact basé sur la programmation dynamique pour le partitionnement de maillage sous contraintes mémoire 67

3.2.3 Obtention d'un algorithme approché pour le partitionnement de maillage sous contraintes mémoire 72

3.3 Algorithme approché pour $R|G_l, mem|\sum p_{v,l}$ 76

3.3.1 Définitions 77

3.3.2 Problème $R|G_l, mem|\sum p_{v,k}$ et présentation de l'algorithme 78

3.3.3 Construction du graphe biparti et de la fonction d'affectation fractionnaire . 81

3.3.4 Obtention d'un couplage entier 85

3.4 Conclusion 91

4 Heuristiques et schéma multi-niveaux **93**

4.1 Introduction au multi-niveaux 94

4.1.1 Phase de contraction 95

4.1.2 Phase de partitionnement initial 96

4.1.3 Phase d'expansion 99

4.2 Méthode multi-niveaux pour le partitionnement de maillage sous contraintes mémoire 104

4.2.1 Phase de contraction 105

4.2.2 Phase de partitionnement initial 110

4.2.3 Phase d'expansion 113

4.3 Résultats expérimentaux 119

Conclusion et perspectives **129**

Introduction

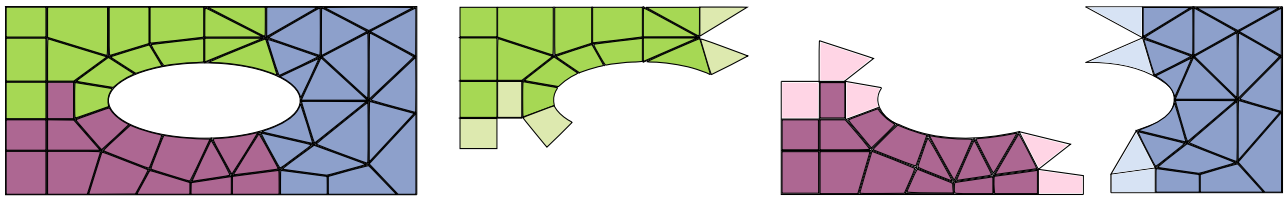
Le contexte général dans lequel s'inscrit notre étude est celui de la simulation numérique. Celle-ci est devenue un outil d'analyse et d'étude incontournable dans plusieurs domaines tels que la mécanique des fluides, l'astrophysique, la physique nucléaire, la climatologie, ... Simuler précisément et fidèlement des phénomènes physiques complexes nécessite de grandes puissances de calcul et le traitement d'importants volumes de données. Cela implique de réaliser ces simulations sur des supercalculateurs à mémoire distribuée où les données et les calculs sont répartis sur différentes unités de calcul.

Dans notre cas, les données des calculs sont attachées à ce que l'on appelle un maillage, c'est-à-dire une discrétisation du domaine d'étude en éléments simples, appelés « mailles ». Dans une simulation parallèle à mémoire distribuée, s'exécutant sur plusieurs unités de calcul, il est nécessaire de répartir le maillage et de distribuer les données qui lui sont attachées. Selon le type de phénomène physique modélisé, ces données peuvent par exemple être un gradient de pression, une densité, une vitesse ou un ensemble de particules. En pratique, ceci se traduit par la distribution des mailles et des données associées sur les différentes unités de calcul. Il est important de maîtriser cette distribution puisqu'elle a un impact sur la mémoire et les temps de calculs. En considérant par exemple une charge unitaire de calcul par maille, on voudra éviter qu'une unité de calcul ait 50% de mailles en plus que les autres. Cela impliquerait au final d'attendre cette unité de calcul et induirait un temps de calcul qui aurait pu être écourté.

En pratique, les approches suivies pour résoudre ce problème représentent le maillage par un graphe ou un hypergraphe pondérés, où le poids associé aux sommets modélise une quantité de calcul, et le poids associé aux arêtes ou hyper-arêtes modélise un coût de communication. Le graphe ou hypergraphe obtenu est ensuite partitionné en équilibrant la charge de calcul par partie, tout en minimisant le volume de communication induit par la partition. C'est en pratique ce que font les algorithmes de partitionnement proposés dans MeTiS [48] et Scotch [67]. À titre d'exemple,

une partition possible d'un maillage sur trois unités de calcul est donnée à la figure 1(a).

Selon le type de schéma numérique choisi pour modéliser le phénomène physique étudié, les calculs effectués au sein d'une maille nécessitent d'accéder aux données portées par des mailles voisines. L'approche standard est de disposer de ce voisinage localement à l'unité de calcul. Certaines mailles sont ainsi dupliquées sur plusieurs unités de calcul. On parle alors de *mailles fantômes*. La figure 1(b) illustre la distribution des données associée à la partition de la figure 1(a) dans le cas d'un voisinage par arête à distance un, c'est-à-dire que deux mailles (ici des triangles et des quadrilatères) partagent une même arête. Sur cette figure, les mailles locales sont en couleur foncée et les mailles fantômes sont en couleur claire. La présence de ces mailles fantômes a un impact sur l'exécution de la simulation en termes de consommation mémoire. En effet, l'espace mémoire de chaque unité de calcul étant limité, ne pas prendre en compte ces mailles peut entraîner des débordements mémoire, ce qui nécessite de tenir compte des mailles locales et des mailles fantômes.



(a) (b)

FIGURE 1 – Exemple d'un maillage dont une partition possible sur trois unités de calcul est présentée en (a). La vue éclatée des mailles distribuées est illustrée en (b) où les mailles fantômes sont présentées en couleurs claires pour chaque unité de calcul.

Partant de ce constat, cette thèse porte sur la définition d'un nouveau problème de partitionnement de maillage et propose différentes approches pour le résoudre. Plus précisément, nous introduisons le problème de partitionnement de maillage sous contraintes mémoire, que nous appelons PMCM, où sont étudiées à la fois la partition des calculs et la distribution des données associées au maillage. Ce problème étant **NP**-difficile, la recherche d'une solution optimale sur des instances de grandes tailles n'est pas envisageable. C'est pourquoi nous proposons d'étudier le problème PMCM sous deux angles :

— l'obtention d'algorithmes approchés avec garantie de performance, c'est-à-dire d'algorithmes garantissant une qualité de solution proche de la solution optimale à un facteur près ;

— la mise en œuvre d'une heuristique basée sur la méthode multi-niveaux.

Dans le but de présenter nos travaux, le présent document est structuré comme suit. Nous rappelons au chapitre 1 les notions et notations qui nous seront nécessaires tout au long de cette

thèse.

Nous définissons au chapitre 2 le problème PMCM que nous étudions et rappelons les différents problèmes de partitionnement classiquement étudiés. Ensuite, afin de comparer ces différents modèles, nous les formalisons tous par des programmes linéaires en nombres entiers. Cette étape de formalisation permettra, d'une part, de montrer que les problèmes classiques de partitionnement sont intrinsèquement différents du problème PMCM, et d'autre part, de comparer les solutions exactes obtenues pour les programmes linéaires en nombres entiers sur des instances de petite taille.

Dans le chapitre 3, nous abordons le problème PMCM comme un problème d'ordonnancement. Après avoir montré que le problème PMCM est une généralisation de problèmes d'ordonnancement connus [4, 31], nous présentons deux résultats obtenus en appliquant des techniques très utilisées en recherche opérationnelle : la programmation dynamique et la programmation linéaire. Le premier résultat est un algorithme FPT approché, en temps polynomial, pour les graphes à largeur linéaire bornée. Le second résultat est un algorithme exact, en temps polynomial, pour un problème voisin de PMCM. Ce problème consiste à minimiser la somme totale des temps de calcul, tout en respectant les contraintes du problème PMCM.

Dans le chapitre 4, nous présentons une heuristique de résolution du problème PMCM. Cette heuristique a été conçue en suivant une approche multi-niveaux qui consiste à réduire la taille du problème, obtenir une solution du problème réduit, puis reporter la solution au problème originel en améliorant la qualité de la solution. Nous détaillons comment nous avons adapté les phases composant cette méthode, c'est-à-dire la phase de contraction, de partitionnement initial et d'expansion. Les heuristiques classiques de partitionnement ne permettant pas toujours d'obtenir des solutions valides au problème PMCM, nous proposons une heuristique originale que nous utilisons lors de la phase de partitionnement initial. Suite au prototypage de cet algorithme multi-niveaux, nous validons celui-ci sur plusieurs instances et confrontons nos résultats avec l'outil de partitionnement Scotch.

Enfin, nous concluons l'ensemble de nos travaux et proposons quelques perspectives.

Chapitre 1

Définitions et notions générales

Préalablement à la présentation des travaux décrits dans ce manuscrit, nous rappelons brièvement quelques notions relatives aux graphes et aux maillages. Avant cela même, nous commençons par préciser quelques notions générales.

1.1 Vocabulaire et notations

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

— Nous distinguerons les expressions « supérieur à »(respectivement « inférieur à ») et « strictement supérieur à »(respectivement « strictement inférieur à »). La première signifiera « supérieur ou égal à », la seconde exclura l'égalité. De la même façon, un nombre sera dit « positif »si et seulement s'il est supérieur à 0 (et donc éventuellement égal à 0).

— Les notations $\mathbb{N}$ et $\mathbb{R}$ désigneront respectivement l'ensemble des nombres entiers naturels et l'ensemble des nombres réels.

— La notation $\lceil x \rceil$ (respectivement $\lfloor x \rfloor$) désignera, pour tout $x \in \mathbb{R}$, la partie entière supérieure (respectivement inférieure) de x.

— La notation $|E|$ désignera le cardinal de l'ensemble E, c'est-à-dire le nombre d'éléments qui composent E.

— La notation $[\![i; j]\!]$ désignera l'intervalle d'entiers entre i et j inclus.

Par la suite, pour évaluer la complexité des algorithmes présentés, nous utiliserons la notation de Landau au voisinage de $+\infty$ avec $\mathcal{O}$ et Θ.

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

$$f = \mathcal{O}(g) \quad \Leftrightarrow \quad \exists k \in \mathbb{R}_+^*, \exists X \in \mathbb{R}, \forall x \in \mathbb{R}, (x > X \Rightarrow |f(x)| \leq k|g(x)|) \ . \tag{1.1}$$

$$f = \Omega(g) \quad \Leftrightarrow \quad \exists k \in \mathbb{R}_+^*, \exists X \in \mathbb{R}, \forall x \in \mathbb{R}, (x > X \Rightarrow |f(x)| \geq k|g(x)|) \ . \tag{1.2}$$

$$f = \Theta(g) \quad \Leftrightarrow \quad f = \mathcal{O}(g) \ \textit{et} \ f = \Omega(g) \ . \tag{1.3}$$

Nous ferons souvent référence à la métrique à minimiser, notée *makespan*, dans notre problème de partitionnement. Celle-ci étant un critère d'optimisation très étudié dans le domaine de l'ordonnancement, nous la définissons selon l'affectation d'un ensemble S de tâches à traiter sur un ensemble M d'unités de calcul.

Définition 2. (Makespan) *Soient S un ensemble de tâches à traiter et M un ensemble d'unités de calcul utilisées pour réaliser ces tâches. Le makespan, induit par l'affectation des tâches de S aux unités de calcul de M, correspond à la date de fin d'exécution de la tâche se terminant en dernier. Minimiser le makespan revient donc à minimiser le temps nécessaire pour le traitement de toutes les tâches de S.*

Notons que la minimisation du makespan peut aussi être vue comme la minimisation du temps de traitement maximum sur une entité de M.

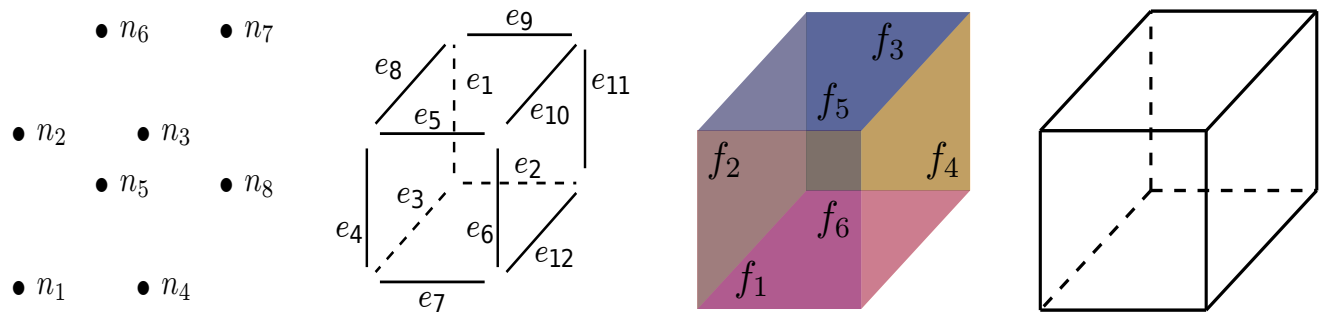
Nous définissons maintenant les objets sur lesquels nous allons principalement travailler, c'est-à-dire les maillages et les graphes.

1.2 Définitions sur les maillages

Dans cette section, nous ne présentons que les notions utiles à la compréhension de ce document. Une étude complète des notions liées au maillage est hors du contexte de notre étude.

Un maillage est une discrétisation d'un domaine Ω à l'aide d'éléments géométriques appelées *cellules*. Les cellules peuvent avoir différentes dimensions : $0D$, $1D$, $2D$ et $3D$. Les cellules $0D$

sont appelées des *nœuds* (figure 1.1(a)), les cellules $1D$ sont appelées des *arêtes* (figure 1.1(b)), les cellules $2D$ sont appelées des *faces* (figure 1.1(c)) et les cellules $3D$ sont appelées des *régions* (figure 1.1(d)).



(a) (b) (c) (d)

FIGURE 1.1 – Élements composant un maillage. En (a) les nœuds composant une région sont présentés ; En (b) les arêtes joignant les noeuds de la région sont présentées ; En (c) les faces formées par les arêtes de la région sont présentées ; En (d) la région est présentée.

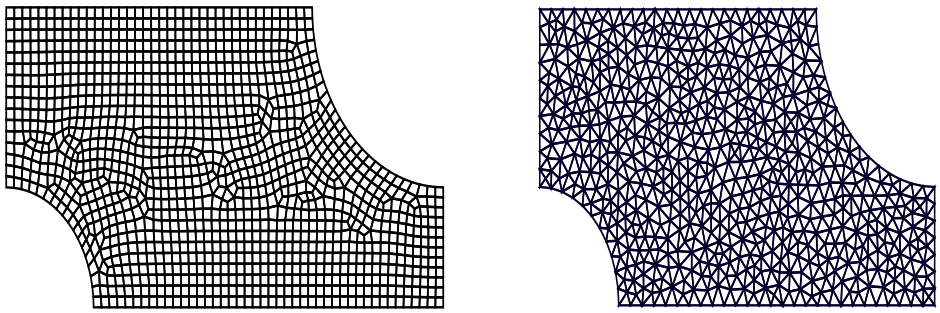
Il existe une décomposition hiérarchique entre les cellules d'un maillage. Une cellule de dimension $i > 0$ est composée de l'union de cellules de dimension j où $0 \leq j \leq i$. Cette relation hiérarchique peut être constatée sur la figure 1.1 où la région, présentée sur la figure 1.1(d), est composée par les faces de la figure 1.1(c), elles-mêmes composées des arêtes de la figure 1.1(b) et elles-mêmes composées des nœuds de la figure 1.1(a). Grâce à cette relation hiérarchique, l'intersection de deux cellules est soit $\emptyset$, soit un ensemble de cellules.

Lorsque la dimension d'une cellule est égale à celle du domaine Ω, on parle alors de *maille*. En dimension 2, les mailles d'un maillage sont généralement des triangles ou des quadrilatères. En dimension 3, ce sont des tétraèdres, des hexaèdres, des prismes ou des pyramides à base carrée. La figure 1.2 présente deux maillages d'un domaine de dimension 2. Le premier (figure 1.2(a)) est composé de mailles quadrangulaires et le second (figure 1.2(b)) est composé de mailles triangulaires.

Nous introduisons maintenant les connectivités entre les mailles et parlerons de relation de voisinage entre cellules.

Définition 3. RELATION DE VOISINAGE *Soient m et m' deux mailles d'un maillage $\mathcal{M}_\Omega$. La maille m est* **voisine** *à la maille m' si et seulement si $m \cap m' \neq \emptyset$.*

Par extension à cette définition, deux mailles seront dites *voisines par arête* signifiera que ces deux mailles partagent une arête commune. À titre d'exemple, considérons le maillage de la



(a) (b)

FIGURE 1.2 – Exemples de deux maillages d'un même domaine. En (a) le maillage est composé de mailles quadrangulaires ; En (b) le maillage est composé de mailles triangulaires.

figure 1.3. Les mailles m_0 et m_1 y sont voisines par arête et par sommet car $m_0 \cap m_1 = \{e_0, n_0, n_1\}$, c'est-à-dire que les mailles m_0 et m_1 partage l'arête e_1 et les nœuds n_0 et n_1.

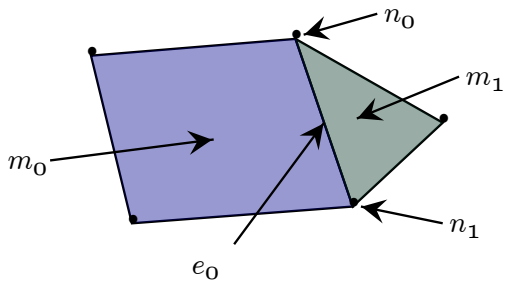


FIGURE 1.3 – Exemple illustrant les relations de voisinage entre cellules.

Définition 4. MAILLAGE DE DIMENSION d *Un maillage $\mathcal{M}_\Omega$ de dimension d, avec $d \geq 0$, est une discrétisation d'un domaine géométrique Ω par des cellules telles que :*

$$\Omega = \bigcup_{c \in \mathcal{M}_\Omega} c,$$

et l'intersection de deux cellules c et c', est soit vide, soit un ensemble de cellules de dimension inférieure.

1.3 Définitions sur les graphes

Nous introduisons maintenant les notions relatives aux graphes.

Définition 5. (GRAPHE NON ORIENTÉ) *Un graphe non orienté $G = (V, E)$ est constitué d'un ensemble V dont les éléments sont appelés **sommets** et d'un ensemble E dont les éléments sont appelés **arêtes**. Une arête $e \in E$ est définie par une paire non ordonnée de sommets de V, appelés*

*les **extrémités** de e. Si l'arête e relie deux sommets v et v', alors ces sommets sont dits **adjacents**, ou **incidents** à e. On dit encore que l'arête e est incidente aux sommets v et v'. Si deux sommets sont incidents à une même arête e, alors ils sont dits **voisins**. Le voisinage d'un sommet $v \in V$ est $\mathcal{N}(v) = \{v' \in V : \{v, v'\} \in E\}$ et le voisinage d'un ensemble de sommets $V' \subseteq V$ est $\mathcal{N}(V') = \bigcup_{v \in V'} \mathcal{N}(v)$.*

À titre d'exemple, la figure 1.4 représente un graphe non orienté $G = (V, E)$ composé de 13 sommets et 15 arêtes. Le sommet v est adjacent aux sommets v_1, v_2 et v_3 car respectivement relié par les arêtes e_1, e_2 et e_3. Les arêtes e_1, e_2 et e_3 sont adjacentes car elles partagent le sommet v.

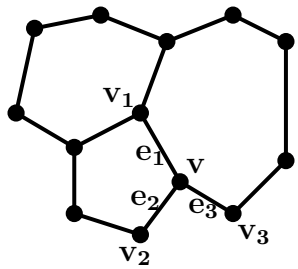


FIGURE 1.4 – Exemple d'un graphe non orienté $G = (V, E)$.

Définition 6. (DEGRÉ D'UN SOMMET) *Soit v un sommet du graphe G. Le degré de v, noté $\delta(v)$, est le nombre d'arêtes de $E(G)$ incidentes à v. Le* **degré minimum** *(respectivement* **maximum***) de G, noté $\delta(G)$ (respectivement $\Delta(G)$), est le minimum (respectivement maximum) des degrés de tous les sommets de G. On a*

$$\delta(G) = \min_{u \in V(G)} \delta(u), \text{ et} \tag{1.4}$$

$$\Delta(G) = \max_{u \in V(G)} \delta(u). \tag{1.5}$$

Définition 7. (GRAPHE ORIENTÉ) *Un graphe orienté $G = (V, A)$ est constitué d'un ensemble V dont les éléments sont appelés sommets et d'un ensemble A dont les éléments sont appelés **arcs**. Un arc $a = (v, v') \in A$ est défini par une paire ordonnée de sommets de V, appelés les **extrémités** de a où v est **l'origine** de a et v' **l'extrémité finale** de a. On dit aussi que v' est un **successeur** de v et que v est un **prédécesseur** de v'. Dans le cas où l'origine et l'extrémité finale d'un arc sont identiques, c'est-à-dire un arc de la forme $a = (v, v) \in A$, on parle de **boucle**.*

À titre d'exemple, la figure 1.5 représente un graphe orienté $G = (V, A)$ composé de 13 sommets et 20 arcs.

Remarquons qu'un graphe non orienté $G = (V, E)$ peut être modélisé par un graphe orienté $G' = (V, A)$ dans lequel, pour toute arête $e = \{v, v'\} \in E$, on crée deux arcs $a = (v, v') \in A$ et $a' = (v', v) \in A$.

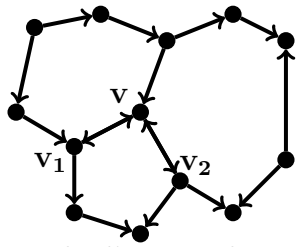
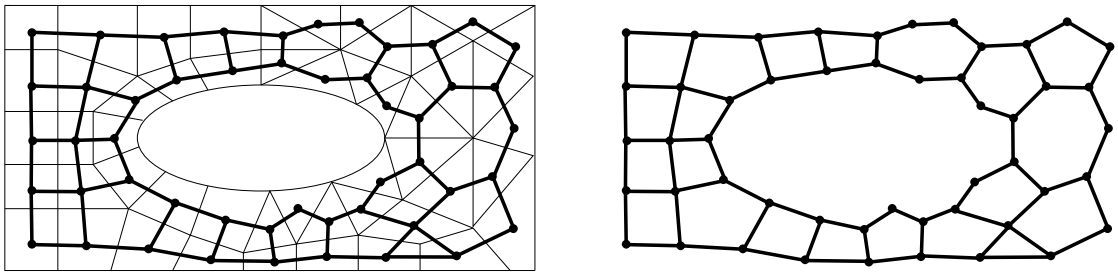


FIGURE 1.5 – Exemple d'un graphe orienté $G = (V, A)$.

Définition 8. (GRAPHE DUAL D'UN MAILLAGE) *On appelle graphe dual d'un maillage $\mathcal{M}_\Omega$ de dimension d, un graphe $G = (V, E)$ que l'on peut définir par la bijection $\psi : \mathcal{M}_\Omega \to G$ qui vérifie :*

- *Pour toute cellule c de dimension d de $\mathcal{M}_\Omega$, $\psi(c)$ est un sommet de G ;*
- *Pour toutes cellules c et c' de dimension d, si c est voisin de c', alors $\psi(c)$ est voisin de $\psi(c')$ dans G.*

Les figures 1.6(a) et 1.6(b) illustrent la modélisation du maillage M_Ω à l'aide de son graphe dual $G = (V, E)$. Chaque sommet de G représente une maille de M_Ω et les arêtes de G modélisent les voisinages par arêtes des mailles de M_Ω.



(a) (b)

FIGURE 1.6 – Exemple de la construction d'un graphe dual d'un maillage M_Ω. En (a) un sommet est associé à chaque maille et ces sommets sont connectés entre eux selon un voisinage par arête à distance 1 ; En (b) le graphe dual résultant est représenté seul.

Définition 9. (GRAPHE BIPARTI) *Un graphe $G = (V, E)$ est dit biparti si et seulement si il existe une partition (V_1, V_2) de V telle que :*

- *$V_1 \neq \emptyset$ et $V_2 \neq \emptyset$;*
- *$\forall (v, w) \in E$, $v \in V_1$ et $w \in V_2$ ou $v \in V_2$ et $w \in V_1$.*

Le figure 1.7 présente un graphe biparti $B = (V_1, V_2, E)$, où V_1 est composé de 5 sommets et V_2 est composé de 4 sommets. Toutes les arêtes $e \in E$ ont une extrémité dans V_1 et l'autre dans V_2.

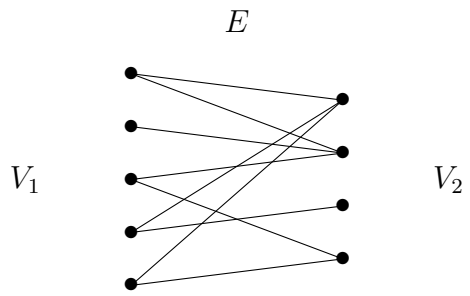


FIGURE 1.7 – Exemple d'un graphe biparti $B = (V_1, V_2, E)$.

Définition 10. (CHEMIN) *Un chemin entre deux sommets u et v est une séquence de sommets $v_1 v_2 \dots v_n$ de $V(G)$ telle que (1) $u = v_1$, (2) $v = v_n$, (3) $\forall i \in [\![1; n-1]\!]$, $\{v_i, v_{i+1}\} \in E(G)$, et (4) $\forall (i,j) \in [\![1; n-1]\!]^2$ si $i \neq j$ alors $v_i \neq v_j$.*

Définition 11. (CYCLE) *Dans un graphe non orienté $G = (V, E)$, un cycle est un chemin dont les extrémités sont confondues.*

Définition 12. (CONNEXITÉ) *Un graphe G est dit* **connexe** *lorsqu'il existe un chemin entre toute paire de sommets.*

Définition 13. (ARBRE) *Un arbre est un graphe acyclique et connexe.*

La figure 1.8 présente un arbre $T = (V, E)$ où r est la racine de T et l_1, l_2, l_3 et l_4 en sont les feuilles.

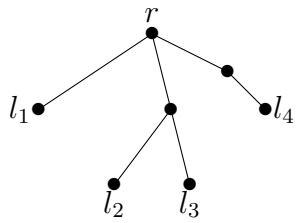


FIGURE 1.8 – Exemple d'un arbre $T = (V, E)$ de racine r et de feuilles l_1, l_2, l_3 et l_4.

Définition 14. (GRAPHE CHEMIN) *Un graphe chemin est un arbre où chaque sommet est de degré au plus deux.*

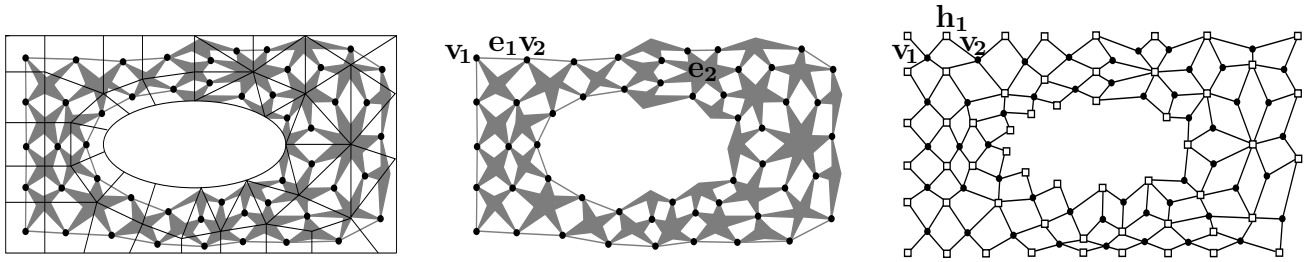
La figure 1.9 présente un exemple de graphe chemin $P = (V, E)$.

FIGURE 1.9 – Exemple d'un graphe chemin $P = (V, E)$.

Définition 15. (HYPERGRAPHE) *Un hypergraphe $H = (V, \mathcal{E})$ est constitué d'un ensemble V dont les éléments sont appelés* ***sommets*** *et d'un ensemble $\mathcal{E}$ dont les éléments sont appelés* ***hyper-arêtes****. Une hyper-arête $e \in \mathcal{E}$ est définie par un ensemble de sommets, appelés les* ***extrémités*** *de e. Si l'hyper-arête e relie deux sommets v et v', alors ces sommets sont dits* ***adjacents****, ou* ***incidents*** *à e. On dit encore que l'hyper-arête e est incidente aux sommets v et v'. Si deux sommets sont incidents à une même hyper-arête e, alors ils sont dits* ***voisins****. Enfin, le nombre de sommets que relie une hyper-arête e est appelé le* ***degré*** *de e.*

On peut aussi définir un hypergraphe comme un graphe biparti $(V, \mathcal{H}, P)$, où V est l'ensemble des sommets, $\mathcal{H}$ l'ensemble des étiquettes des hyper-arêtes, et P l'ensemble des arêtes connectant un élément de V et un élément de $\mathcal{H}$. Une hyper-arête est alors définie comme l'ensemble des arêtes connectées à une étiquette.

La figure 1.10(a) illustre la modélisation du maillage M_Ω à l'aide d'un hypergraphe $H = (V, \mathcal{E})$ où deux mailles sont voisines si elles partagent au moins un nœud. On peut noter que certaines hyper-arêtes sont réduites à deux sommets, comme c'est le cas de l'hyper-arête e_1 sur la figure 1.10(b), et donc correspondent à des arêtes classiques. Ces hyper-arêtes y représentent les relations de voisinage entre des mailles partageant un nœud du bord. On peut aussi noter que les hyper-arêtes peuvent avoir des degrés plus élevés, comme c'est le cas de l'hyper-arête e_2. La figure 1.10(c) fournit une représentation de l'hypergraphe $H' = (V, \mathcal{H}, P)$ associé à $H = (V, \mathcal{E})$. L'hyper-arête e_1 est définie par les sommets v_1, v_2, l'étiquette h_1 et les deux arêtes connectant h_1 aux sommets v_1 et v_2.



(a) (b) (c)

FIGURE 1.10 – Exemple de modélisation d'un maillage à l'aide d'hypergraphes. En (a) est présentée la construction de l'hypergraphe $H = (V, \mathcal{E})$ du maillage M_Ω, où celui-ci est représenté en traits fins ; En (b) est présenté l'hypergraphe $H = (V, \mathcal{E})$, de M_Ω où chaque maille est représentée par un sommet et chaque voisinage par une hyper-arête ; En (c) est présenté l'hypergraphe $H' = (V, \mathcal{H}, P)$ associé à l'hypergraphe $H = (V, \mathcal{E})$.

Chapitre 2

Formulation et analyse

L'objectif de ce chapitre est de présenter les modèles utilisés pour le partitionnement de maillage dans la littérature.

L'objectif de ce chapitre est de montrer que la résolution des problèmes de partitionnement classiques ne permet pas toujours d'obtenir des solutions valides vis-à-vis des contraintes mémoire auxquelles nous nous intéressons. Pour cela, nous rappellerons en section 2.1 les problèmes classiques de partitionnement et présenterons formellement le problème de partitionnement de maillage sous contraintes mémoire en section 2.2.

Dans la section 2.3, nous modéliserons ces problèmes de partitionnement grâce à la programmation linéaire en nombres entiers et à la programmation quadratique en nombres entiers. Ces formalismes nous permettront d'obtenir une modélisation mathématique simple et claire de chaque problème de partitionnement présenté. Nous constaterons ainsi que la nature du problème que nous étudions est intrinsèquement différente de celles des autres problèmes de partitionnement.

Enfin, pour appuyer nos propos, nous résoudrons de manière exacte ces problèmes sur des instances de petites tailles à la section 2.4. Nous pourrons ainsi voir que, même en résolvant de manière exacte les problèmes classiques de partitionnement, les solutions obtenues ne sont pas toujours valides. De plus, nous pourrons comparer la qualité des solutions optimales obtenues pour chaque problème.

2.1 Introduction et formalisation des problèmes

Dans cette section, nous allons introduire et formaliser le problème que nous étudions, ainsi que certains problèmes classiques de partitionnement. Pour cela, nous commençons par définir formellement les notions de k-partition et de distribution. Ces notions nous permettront notamment de distinguer, dans notre problème, le partitionnement des calculs, de la distribution des données.

Définition 16. (k-Partition) *Soient k un entier et P un ensemble quelconque. Une famille finie $\pi = (P_1, \ldots, P_k)$ de sous-ensembles de P est appelée k-partition de P si et seulement si $P = \cup_{i=1}^{k} P_i$ et $\forall (P_i, P_j) \in \pi^2$ avec $i \neq j$, $P_i \cap P_j = \emptyset$. Les sous-ensembles P_i sont alors appelés les* **parties** *de la partition π.*

Notons que la définition de k-partition autorise les parties à être vides, ce qui n'est généralement pas le cas.

Définition 17. (k-Distribution) *Soient P un ensemble quelconque. Une famille finie $\pi = (P_1, \ldots, P_k)$ de sous-ensembles de P est appelée k-distribution de P si et seulement si $P = \cup_{i=1}^{k} P_i$.*

Dans la suite de ce manuscrit, nous utiliserons le terme *partition* à la place de k-partition et de *distribution* à la place de k-distribution.

Remarquons que toute partition est une distribution mais que la réciproque est fausse puisque, par exemple, une distribution peut être composée de sous-ensembles dont l'intersection est non vide. Pour insister sur l'idée qu'un élément p d'un ensemble P peut appartenir à plusieurs sous-ensembles d'une distribution, nous parlerons de distribution avec recouvrement.

Dans le cas du problème de partitionnement de maillage, nous distinguons la partition des calculs, de la distribution des données. Nous entendons par partition du maillage, la partition des calculs associés au maillage et par distribution du maillage, la distribution des données associées au maillage. La distribution induite par cette partition est obtenue avec recouvrement en raison des contraintes de voisinages introduites à la page 8. Partitionner un maillage de dimension n est usuellement interprété comme partitionner les cellules de dimension n composant le maillage [5]. Notons toutefois que certaines approches récentes [56] considèrent toutes les cellules du maillage (sommets, arêtes, faces, régions) comme pouvant être distribuées.

Nous définissons maintenant un problème de partitionnement.

Définition 18. (Problème de partitionnement) *Soit Π l'ensemble des k-partitions d'un ensemble P de taille finie. Un problème de partitionnement consiste à rechercher une k-partition $\pi \in \Pi$ telle que π minimise une fonction objectif, notée f, $f : \Pi \to \mathbb{R}$, tout en respectant un ensemble de contraintes, noté c, $c : \Pi \to \{Vrai, Faux\}$.*

Ces différents ensembles sont illustrés en figure 2.1, où $\mathcal{C}$ est l'ensemble des partitions respectant l'ensemble des contraintes c, et $\mathcal{F}_{\mathcal{C}}$ est l'ensemble des partitions de $\mathcal{C}$ pour lesquelles la fonction objectif f est minimisée.

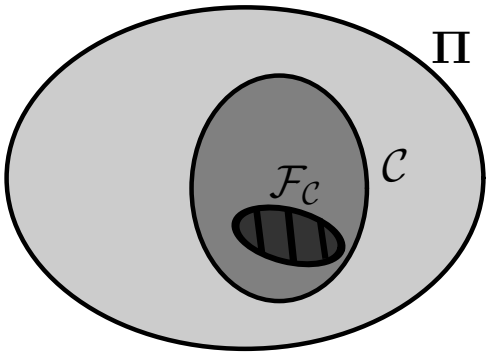


Figure 2.1 – Représentation des partitions Π ainsi que du sous-ensemble des partitions $\mathcal{C}$, respectant l'ensemble des contraintes c, et du sous-ensemble des partitions $\mathcal{F}_{\mathcal{C}} \in \mathcal{C}$ pour lesquelles la fonction objectif f est minimisée.

Nous pouvons alors énoncer de manière générale un problème de partitionnement sous la forme d'un problème d'optimisation :

$$\begin{aligned} \min_{\pi \in \Pi} \quad & f(\pi), \\ \text{contraint par} \quad & c(\pi). \end{aligned} \tag{2.1}$$

Ces diverses notions étant définies, nous introduisons maintenant certains problèmes classiques de partitionnement, avant d'introduire le problème qui nous est spécifique. Pour une présentation détaillée des problèmes de partitionnement de graphe et d'hypergraphe, le lecteur intéressé pourra se référer à [7] où sont aussi présentés des outils et des méthodes pour résoudre ces problèmes.

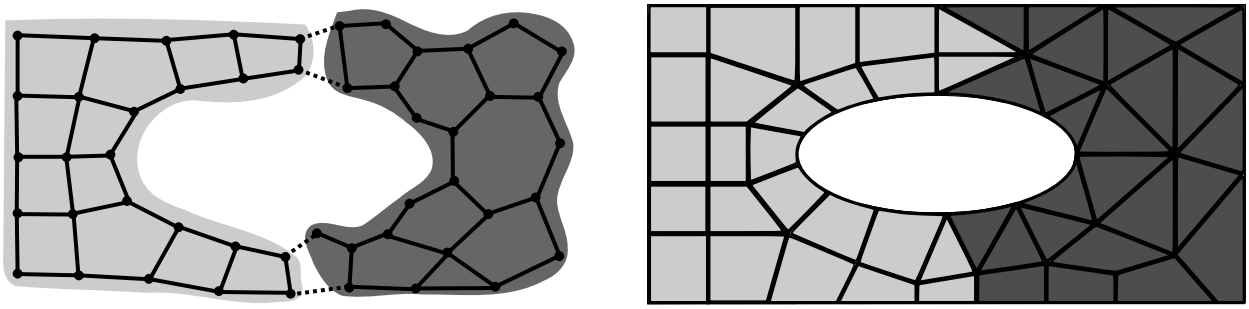
2.1.1 Partitionnement de graphe

Pour commencer, considérons le problème de k-partitionnement de graphe [50]. Ce problème consiste à répartir l'ensemble des sommets d'un graphe pondéré $G = (V, E)$ en k parties $V_1, \ldots, V_k$,

en minimisant le poids total des arêtes reliant les sous-ensembles $V_1, \ldots, V_k$ (arêtes coupées) et en garantissant que ces sous-ensembles aient sensiblement le même poids total de sommets (contrainte d'équilibrage). Ce problème modélise un besoin concret qui survient en VLSI [1] dans la disposition des composants [74], et en calcul distribué où l'on doit répartir des calculs sur un nombre donné d'unités de calcul [38].

Minimiser le poids total des arêtes coupées C, revient alors à réduire le volume des échanges entre parties, et donc le volume des communications entre processus distants. Respecter la contrainte d'équilibrage de charge E_c implique usuellement de minimiser le temps de restitution de la simulation, puisque les processus auront une charge de travail équivalente. En minimisant C et en respectant E_c, on réduit intuitivement à la fois les coûts de communication et les coûts de calcul. C'est l'approche suivie dans des outils tels que MeTiS [47] et Scotch [66].

Manipuler directement un maillage étant difficile, il est fréquent de le modéliser à l'aide de son graphe dual. Réaliser le partitionnement de ce graphe, qui modélise à la fois les mailles du maillage ainsi que leurs relations de voisinage, nous apporte alors une partition du maillage respectant la contrainte d'équilibrage de charge tout en minimisant la somme des poids des arêtes coupées. Un exemple de partition de graphe est présenté sur la figure 2.2(a), où les deux parties ont une charge de calcul égale et où le nombre d'arêtes coupées (arêtes en pointillées) est minimale. La partition du maillage, induite par la partition de son graphe dual, est illustrée en figure 2.2(b).



(a) (b)

Figure 2.2 – Exemple d'une bipartition du graphe dual de M_Ω en (a) où les mailles sont modélisées par les sommets du graphe et les relations de voisinage par les arêtes du graphe. Cette bipartition du maillage est illustrée en (b).

Nous donnons maintenant une définition formelle du problème de k-partitionnement de graphe.

1. Abréviation de l'anglais *Very Large Scale Integration* : technique permettant d'implanter plus de 100 000 composants électroniques sur un circuit intégré.

Définition 19. (k-PARTITIONNEMENT DE GRAPHE) *Soient un graphe non orienté $G = (V, E)$ à arêtes et sommets pondérés, un nombre de parties k et une valeur de déséquilibre maximal autorisé $\mathcal{E}_{eq} \geq 0$. Notons $w_v : v \in V \to \mathbb{R}^+$ et $w_e : e \in E \to \mathbb{R}^+$ les fonctions qui à chaque sommet ou arête associent un poids. Le problème de k-partitionnement de graphe consiste alors à trouver une partition $\pi = (V_i)_{1 \leq i \leq k}$ de V telle que :*

1. *le poids de chaque partie V_i soit sensiblement le même, c'est-à-dire que le poids de chaque partie soit à un facteur au plus $(1+\mathcal{E}_{eq})$ du poids moyen d'une partie. Le prédicat d'équilibre d'une partie V_i s'évalue comme :*

$$\textit{Équilibre}(\pi) = \begin{cases} Vrai & si \quad \forall 1 \leq i \leq k, \sum_{u \in V_i} \omega_v(u) \leq (1 + \mathcal{E}_{eq}) \times \frac{\sum_{u \in V} \omega_v(u)}{k}, \\ Faux & sinon\,; \end{cases}$$

2. *la coupe, c'est-à-dire le poids total des arêtes reliant les sous-ensembles $V_1, \ldots, V_k$, soit minimale. La coupe d'une partition π s'évalue comme :*

$$Coupe(\pi) = \frac{1}{2} \sum_{i=1}^{k} \sum_{\substack{\{u,v\} \in E \\ u \in V_i \\ v \notin V - V_i}} \omega_e(\{u, v\}).$$

Le facteur $\frac{1}{2}$, dans l'expression de la coupe d'une partition, permet de ne pas compter deux fois le poids de chaque arête. En pratique, la valeur de déséquilibre maximal autorisé est souvent dans l'intervalle $[0; 0.05]$ afin d'orienter la recherche vers des solutions équilibrées.

En formalisant ce problème comme un problème d'optimisation (voir page 21), il se définit comme suit :

— la fonction objectif f évalue la coupe induite par la partition ;

— l'ensemble des contraintes c force la partition π à être composée de parties sensiblement de même poids.

Nous énonçons alors ce problème sous la forme :

$$\begin{aligned} \min_{\pi \in \Pi} \quad & Coupe(\pi), \\ \text{contraint par} \quad & \textit{Équilibre}(\pi). \end{aligned} \tag{2.2}$$

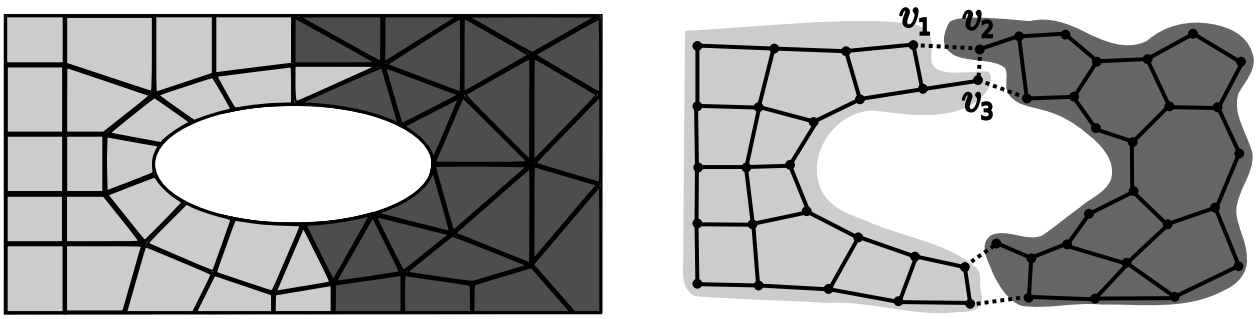
L'idée sous-jacente de ce problème est d'obtenir une partition équilibrant la charge de calcul sur chacune des parties, tout en minimisant le volume total de communications. Le volume total de communications est la métrique la plus populaire mais il en existe d'autres [38] telles que :

— le temps pour envoyer un message, fonction qui peut être plus importante que le volume du message et qui est dépendante de la taille du message ainsi que de la latence ;

— le volume et/ou le nombre maximum de messages traités par toute partie, qui peut ne pas être équilibré même si le coût de calcul l'est ;

— le nombre de commutateurs réseau à travers lesquels un message est acheminé, qui peut induire des conflits de messages et diminuer la qualité du trafic des messages.

Malheureusement, ce modèle de graphe ne minimise pas réellement le volume total de communication [38]. Prenons par exemple le cas de la partition présentée à la figure 2.3(a) et considérons la modélisation associée sous forme de graphe présentée à la figure 2.3(b). Nous pouvons remarquer que le sommet v_2 ayant deux voisins v_1 et v_3 appartenant à une autre partie, le nombre de communications associées à ce sommet v_2 est donc de 2. Or, les sommets v_1 et v_3 appartenant à la même partie, il est donc possible qu'une seule communication de v_2 soit suffisante pour ces deux sommets.



(a) (b)

FIGURE 2.3 – Exemple d'une bipartition d'un maillage M_Ω en (a) et de sa modélisation par un graphe en (b). Les arêtes $\{v_1, v_2\}$ et $\{v_2, v_3\}$ sont coupées, et donc le coût de communication associé au sommet v_2 est de 2.

2.1.2 Partitionnement d'hypergraphe

Dans cette section, nous considérons le problème de k-partitionnement d'hypergraphe [11]. Ce problème consiste à partitionner l'ensemble des sommets d'un hypergraphe $H = (V, \mathcal{E})$ en k parties $V_1, \ldots, V_k$, en minimisant une fonction de coût induite par la partition et en garantissant que ces

parties aient sensiblement le même poids total de sommets (contrainte d'équilibrage). Tout comme le problème de k-partitionnement de graphe, ce problème modélise un besoin concret qui survient en VLSI dans la disposition des composants [2], en algèbre linéaire dans le produit matrice creuse / vecteur [12], en satisfiabilité booléenne [23], ...

La modélisation à l'aide d'un hypergraphe permet de modéliser plus précisément le volume de communications qu'avec le modèle à base de graphes. En effet, un graphe est une structure permettant d'exprimer uniquement des relations entre couples de sommets, alors qu'un hypergraphe permet de modéliser des relations définies sur des ensembles de sommets [2].

Tout comme pour le problème de k-partitionnement de graphe, le k-partitionnement d'hypergraphe consiste à minimiser les échanges entre parties (la coupe), tout en respectant une contrainte d'équilibrage de charge. Une métrique populaire pour évaluer la coupe est celle dite $\lambda - 1$ [20]. Si une hyper-arête connecte λ parties, alors il est nécessaire de communiquer $\lambda - 1$ fois chaque sommet de cette hyper-arête. Notons $w_e : e \in \mathcal{E} \rightarrow \mathbb{R}^+$ la fonction associant un poids à chaque hyper-arête et λ_e le nombre de parties que connecte l'hyper-arête e. Nous définissons alors la coupe comme étant

$$Coupe_{\lambda-1}(\pi) = \sum_{e \in \mathcal{E}} (\lambda_e - 1)\ \omega_{\mathcal{E}}(e).$$

On peut noter que dans le cas d'un graphe, $\lambda \leq 2$, cette expression de la coupe correspond alors à la formule de l'évaluation de la coupe du graphe.

Une telle partition est présentée en figure 2.4(a), où les hyper-arêtes coupées sont identifiables par des traits en pointillés. La partition du maillage induite par la partition de l'hyper-graphe est illustrée sur la figure 2.4(b).

Nous donnons maintenant une définition formelle du problème de k-partitionnement d'hypergraphe.

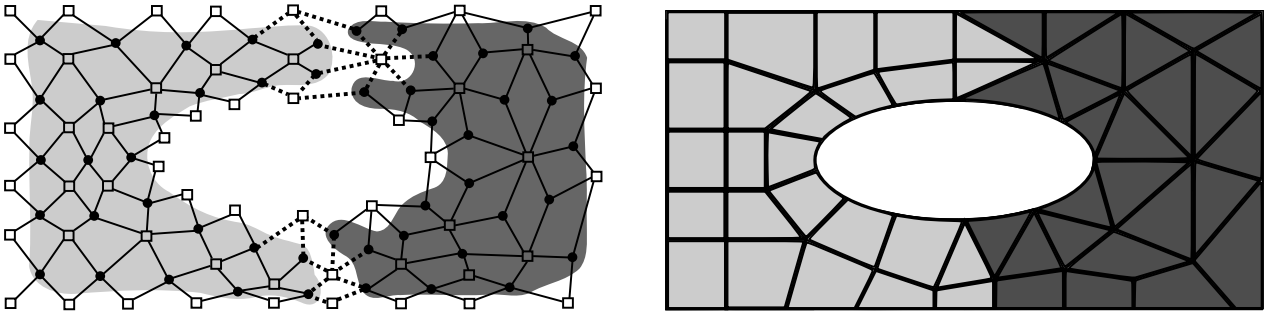
Définition 20. (k-PARTITIONNEMENT D'HYPERGRAPHE) *Soient un hypergraphe non orienté $H = (V, \mathcal{E})$ à hyper-arêtes et sommets pondérés, un nombre de parties k et une valeur de déséquilibre maximal autorisé $\mathcal{E}_{eq} \geq 0$. Les fonctions $w_v : v \in V \rightarrow \mathbb{R}^+$ et $w_e : e \in \mathcal{E} \rightarrow \mathbb{R}^+$ désignent respectivement les applications qui associent un poids à chaque sommet et hyper-arête. Le problème de k-partitionnement d'hypergraphe consiste alors à trouver une partition $\pi = (V_i)_{1 \leq i \leq k}$ de V telle que :*

2. Un graphe est un hypergraphe dont toutes les hyper-arêtes sont d'arité 2.

1. *le poids de chaque partie V_i soit sensiblement le même, c'est-à-dire que le poids de chaque partie soit à un facteur au plus $(1+\mathcal{E}_{eq})$ du poids moyen d'une partie. Le prédicat d'équilibre d'une partie V_i s'évalue comme :*

$$\acute{E}quilibre(\pi) = \begin{cases} Vrai & si \quad \forall 1 \leq i \leq k, \sum_{u \in V_i} \omega_v(u) \leq (1+\mathcal{E}_{eq}) \times \frac{\sum_{u \in V} \omega_v(u)}{k}, \\ Faux & sinon\,; \end{cases}$$

2. *la métrique de coupe, par exemple le poids total des arêtes reliant les sous-ensembles $V_1, \ldots, V_k$ ou la métrique $\lambda - 1$, soit minimale.*



(a) (b)

FIGURE 2.4 – Exemple d'une bipartition d'un hypergraphe modélisant le maillage M_Ω, avec un voisinage par sommet. En (a), les mailles sont modélisées par les sommets de l'hypergraphe, et les relations de voisinage par les hyper-arêtes de l'hypergraphe. En (b), est illustrée la bipartition correspondante du maillage.

En formalisant ce problème comme un problème d'optimisation (voir page 21), il se définit comme suit :

— la fonction objectif f évalue la coupe induite par la partition ;

— l'ensemble des contraintes c force la partition π à être composée de parties sensiblement de même poids.

Nous énonçons également ce problème comme :

$$\begin{aligned} \min_{\pi \in \Pi} \quad & Coupe_{\lambda-1}(\pi), \\ \text{sujet à} \quad & \acute{E}quilibre(\pi). \end{aligned} \tag{2.3}$$

Remarquons que ce problème se formalise de la même façon que le problème de k-partitionnement

de graphe. L'utilisation d'hypergraphe peut s'avérer intéressante puisqu'elle étend les possibilités de modélisation à l'aide de graphe. Elle permet notamment de modéliser plus précisément le volume de communications induit par la partition. Cependant, les hypergraphes ne constituent qu'une extension des graphes et les algorithmes utilisés dans le cadre du partitionnement restent très proches de ceux utilisés pour les graphes, avec très souvent les mêmes limitations.

2.1.3 Partitionnement à séparateur sommets

Dans cette section, nous considérons le problème de partitionnement à séparateur sommets [71]. Ce problème consiste à rechercher, dans un graphe $G = (V, E)$, un ensemble de sommets, appelé séparateur, permettant la séparation des autres parties de la partition. Ce problème est étudié dans le cadre de la génération de numérotations des inconnues de grands systèmes linéaires creux [16]. Tout comme pour les problèmes de k-partitionnement de graphe et d'hypergraphe, ce problème consiste à obtenir k parties respectant une certaine contrainte d'équilibrage de charge. Cependant, contrairement à ceux-ci, la fonction objectif du problème de partitionnement à séparateur sommets n'évalue pas directement la coupe de la partition. En effet, l'objectif de ce problème est d'obtenir un séparateur dont le cardinal soit minimal [24].

Le séparateur sommet est usuellement vu comme une partie supplémentaire, la (k+1)^{ème}, et le problème de partitionnement à séparateur sommets est alors traité comme la recherche d'une partition en $k + 1$ parties, minimisant le cardinal de la (k+1)^{ème} partie où :

— $\forall \{v, v'\} \in E, v \in V_i \Rightarrow v' \in V_i$ ou $v' \in V_{k+1}$;

— la partition respecte la contrainte d'équilibrage de charge.

Une telle partition est présentée en figure 2.5(a), où les deux parties respectent la contrainte d'équilibrage de charge et où le séparateur (ensemble des sommets encerclés par des pointillés) est de taille minimale. La partition du maillage, induite par la partition par séparateur sommets, est illustrée en figure 2.5(b).

Nous donnons maintenant une définition formelle du problème de partitionnement à séparateur sommets.

Définition 21. (Partitionnement à séparateur sommets) *Soient un graphe non orienté $G = (V, E)$ à sommets pondérés, un nombre de parties k une valeur de déséquilibre maximal autorisé $\mathcal{E}_{eq} \geq 0$. Notons $w_v : v \in V \rightarrow \mathbb{R}^+$ la fonction qui associe un poids à chaque sommet. Le problème de partitionnement à séparateur sommets consiste alors à trouver une parti-*

tion $\pi = (V_i)_{1 \leq i \leq k+1}$ *de* V *telle que :*

1. *les parties* 1 *à* k *soient séparées deux à deux par la* $(k+1)^{ème}$ *partie. Le prédicat de séparation d'une partie* V_i *s'évalue comme :*

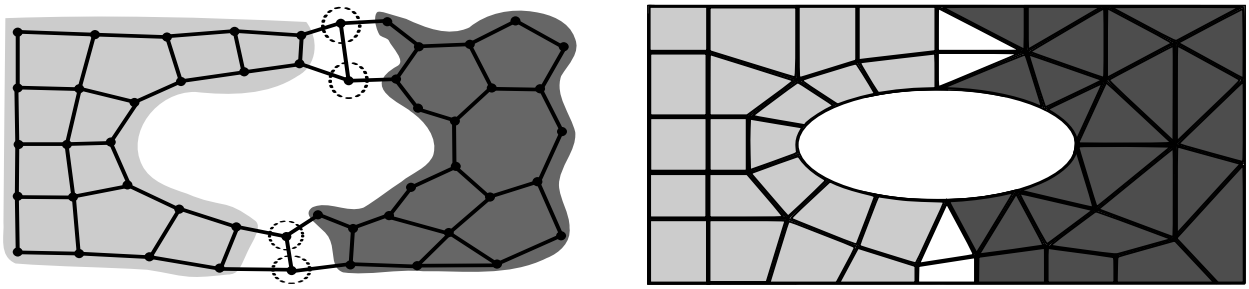
$$Séparation(\pi) = \begin{cases} Vrai & si \quad \forall 1 \leq i \leq k, \forall \{v, v'\} \in E, v \in V_i \Rightarrow v' \in V_i \text{ ou } v' \in V_{k+1}, \\ Faux & sinon\,; \end{cases}$$

2. *le poids de chaque partie* V_i, $1 \leq i \leq k$, *soit sensiblement le même, c'est-à-dire que le poids de chaque partie soit à un facteur au plus* $(1 + \mathcal{E}_{eq})$ *du poids des seuls sommets qui n'appartiennent pas au séparateur. Le prédicat d'équilibre d'une partie* V_i *s'évalue comme :*

$$Équilibre(\pi) = \begin{cases} Vrai & si \quad \sum_{u \in V_i} \omega_v(u) \leq (1 + \mathcal{E}_{eq}) \times \frac{\sum_{u \in V \setminus V_{k+1}} \omega_v(u)}{k}, \\ Faux & sinon\,; \end{cases}$$

3. *la taille du séparateur sommets, c'est à dire le cardinal de la partie* V_{k+1}, *soit minimal. Le séparateur d'une partition* $\pi = (V_i)_{1 \leq i \leq k+1}$ *s'évalue comme :*

$$Séparateur(\pi) = |V_{k+1}|.$$



(a) (b)

FIGURE 2.5 – Exemple d'une bipartition, avec séparateur sommet, du graphe dual modélisant le maillage M_Ω, avec un voisinage par arête. En (a) les mailles sont modélisées par les sommets du graphe, et les relations de voisinage par les arêtes du graphe. En (b) est illustrée la bipartition correspondante du maillage où les mailles en blanc définissent le séparateur.

En formalisant ce problème comme un problème d'optimisation (voir page 21), il se définit comme suit :

— la fonction objectif f évalue la taille du séparateur, c'est-à-dire la $(k+1)^{ème}$ partie ;

— l'ensemble des contraintes c force la partition à être composée de $k+1$ parties où les parties 1 à k sont sensiblement de même poids, et sont séparées deux à deux par la $(k+1)^{ème}$

partie.

Nous énonçons ce problème comme :

$$\begin{aligned} \min_{\pi \in \Pi} \quad & \textit{Séparateur}(\pi), \\ \text{sujet à} \quad & \textit{Équilibre}(\pi), \\ & \textit{Séparation}(\pi). \end{aligned} \tag{2.4}$$

Un intérêt pratique du problème de partitionnement à séparateur sommets, vis-à-vis du problème de partitionnement de maillage que nous nous proposons d'étudier, est que le séparateur sommets peut servir à modéliser les mailles fantômes induites par la partition. Dans le cas où l'on a une seule couche de mailles fantômes, celles-ci seront les éléments duaux des sommets du séparateur sommets. Un tel sommet v sera distribué sur chacune des parties auxquelles appartiennent des sommets adjacents à v et n'appartenant pas au séparateur sommets.

Nous allons maintenant introduire et définir formellement le problème de partitionnement de maillage sous contraintes mémoire.

2.2 Partitionnement de maillage sous contraintes mémoire

Dans cette section, nous introduisons le problème de partitionnement de maillage sous contraintes mémoire. Ce problème consiste à chercher une partition des cellules du maillage, auxquelles sont associés des calculs, de manière à minimiser le makespan tout en respectant la capacité mémoire des unités de calcul. La représentation de cette partition en mémoire est alors une distribution, à cause de la duplication des mailles fantômes. Nous allons donc distinguer dans la formulation du problème, le partitionnement des calculs et la distribution des données. Pour cela, nous introduisons une nouvelle modélisation du maillage nous permettant de modéliser une cellule selon ces deux points de vues.

Contrairement aux problèmes de partitionnement classiques qui utilisent un graphe (ou un hypergraphe) dual associé au maillage, notre modèle représente le maillage à l'aide d'un graphe biparti $B = (C, D, E)$ où :

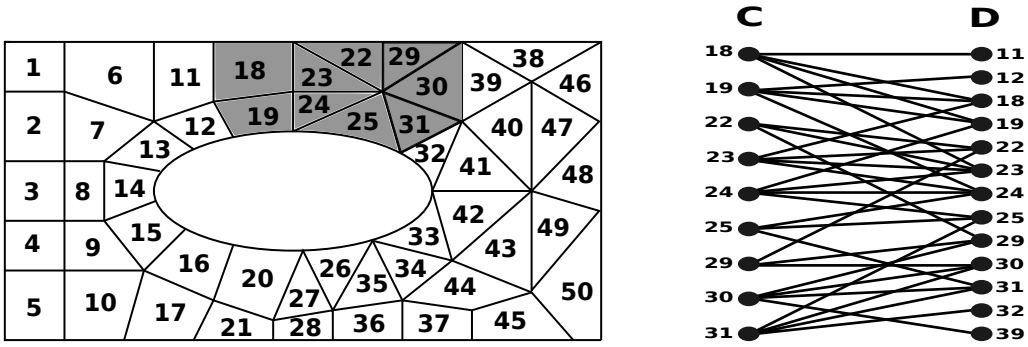
— l'ensemble des sommets $c \in C$, appelés sommets *calcul*, modélise les cellules du maillage dont on désire réaliser les calculs ;

— l'ensemble des sommets $d \in D$, appelés sommets *donnée*, modélise les données des cellules du maillage nécessaires pour la réalisation des calculs ;

— l'ensemble des arêtes $e \in E$ modélise les besoins en informations des sommets de C (sommets *calcul*) vis-à-vis des sommets de D (sommets *donnée*).

Pour des raisons de lisibilité, nous considérons que les mailles sont les seules cellules du maillage auxquelles sont associés un calcul et des données. Ainsi, une maille i du maillage est modélisée par deux sommets c_i et d_i appartenant respectivement à C et D. Puisque nous considérons que tout calcul associé à une maille nécessite les données de celle-ci, il existe alors une arête reliant c_i à d_i.

Nous illustrons cette représentation à l'aide d'un exemple où, pour des raisons de visibilité, nous ne considérons qu'une partie restreinte du maillage M_Ω que nous avons numérotée. La figure 2.6(a) présente le maillage numéroté ainsi que la partie du maillage (couleur grise) concernée. La figure 2.6(b) présente la modélisation de la partie grisée du maillage à l'aide du graphe biparti $B = (C, D, E)$. Dans cet exemple, nous considérons que le calcul d'une maille i nécessite la connaissance des données portées par elle-même et par les mailles avec lesquelles elle partage une arête.



(a) (b)

FIGURE 2.6 – Graphe biparti $B = (C, D, E)$ associé au maillage. Pour des raisons de lisibilité, uniquement la partie grisée du maillage (a) est présentée dans le graphe (b). Ici, nous considérons que la réalisation du calcul associé à une maille i nécessite les données de i et des mailles partageant une arête avec i.

Nous donnons maintenant une définition formelle de problème de partitionnement de maillage sous contraintes mémoire.

Définition 22. (Partitionnement de maillage sous contraintes mémoire) *Soient un graphe biparti $B = (C, D, E)$ à sommets pondérés et M un ensemble d'unités de calcul. Notons $w_c : c \in C \to \mathbb{R}^+$ la fonction qui associe un coût à chaque sommet calcul, $w_d : d \in D \to \mathbb{R}^+$ la fonction qui associe un poids à chaque sommet donnée et $\mathrm{Mem}[i] \in \mathbb{N}$ la capacité mémoire de chaque unité de calcul $i \in M$. Le problème du partitionnement de maillage sous contraintes mémoire consiste alors à trouver une partition $\pi = (C_i)_{1 \leq i \leq |M|}$ de C telle que :*

1. *les données nécessaires à chaque unité de calcul puissent être stockées, c'est-à-dire que la quantité de données associée à une unité de calcul ne dépasse pas sa capacité. Le prédicat de validité mémoire d'une partie C_i s'évalue comme :*

$$Mémoire(\pi) = \begin{cases} Vrai & si \quad \forall 1 \leq i \leq k, \sum_{d \in D_i} w_d(d) \leq \mathrm{Mem}[i], \text{ où } D_i = \bigcup_{c \in C_i} \mathcal{N}(c), \\ Faux & sinon\,; \end{cases}$$

2. *la charge de calcul de l'unité de calcul la plus chargée, notée C_{max}, soit minimisée. Cette charge de calcul s'évalue comme :*

$$C_{max}(\pi) = \max_{i \in M} \sum_{c \in C_i} w_c(c).$$

En utilisant le formalisme précédemment introduit, nous définissons ce problème de partitionnement de la façon suivante :

— la fonction objectif f évalue la charge de calcul de l'unité de calcul la plus chargée ;
— l'ensemble des contraintes c force la partition π à induire une distribution des données ne dépassant pas la capacité mémoire associée à chaque partie.

Nous énonçons ce problème comme :

$$\begin{aligned} \min_{\pi \in \Pi} \quad & C_{max}(\pi), \\ \text{sujet à} \quad & Mémoire(\pi). \end{aligned} \tag{2.5}$$

Récapitulatif des différents problèmes de partitionnement

Nous avons énoncé différents problèmes connus de partitionnement (voir section 2.1) ainsi que le problème de partitionnement de maillage sous contraintes mémoire et nous pouvons remarquer que mis à part le partitionnement de graphe et d'hypergraphe, les objectifs et contraintes de chaque problème diffèrent. Nous rappelons ci-dessous les fonctions objectifs et les contraintes de ces problèmes de partitionnement.

Problème de partitionnement	**Fonction objectif**	**Contraintes**	**Données**
k-Partitionnement de graphe	$Coupe(\pi)$	$Équilibre(V_i)$	Graphe
k-Partitionnement d'hypergraphe	$Coupe(\pi)$	$Équilibre(V_i)$	Hypergraphe
Séparateur sommets	$Séparateur(\pi)$	$Équilibre(V_i)$ $Séparation(V_i)$	Graphe
Partitionnement de maillage sous contraintes mémoires	$Charge_Max(\pi)$	$Mémoire(C_i)$	Graphe Biparti

Au vu des fonctions objectif et des contraintes prises en compte par ces problèmes, nous constatons que les problèmes de partitionnement classiques n'optimisent pas la métrique du partitionnement de maillage sous contraintes mémoire et ne forcent pas la partition à être valide vis-à-vis des contraintes mémoire. En effet, la métrique que nous considérons est le temps de restitution final induit par la partition. Or, les métriques considérées par les problèmes classiques de partitionnement sont la coupe de la partition ou la taille du séparateur. De même, une partition est valide vis-à-vis du problème de partitionnement de maillage sous contraintes mémoire si et seulement si la quantité de données associées à chaque unité de calcul est inférieure à sa capacité. Or, les problèmes de partitionnement classiques sont composés d'une contrainte sur l'équilibrage du poids associé à chaque partie et ne prennent pas en compte la capacité des unités de calcul.

Nous allons maintenant formuler ces problèmes sous la forme de programmes linéaires, et de programmes quadratiques, afin d'étudier leurs solutions optimales sur plusieurs types d'entrées.

2.3 Programmes linéaires et quadratiques

Dans cette section, nous modélisons les problèmes de partitionnement précédemment définis grâce à la programmation linéaire en nombres entiers [19] et à la programmation quadratique en nombres entiers [27]. Ces formalismes permettent une modélisation mathématique simple et claire des problèmes précédents, ainsi que leur résolution exacte. Nous introduirons successivement le problème de partitionnement de maillage sous contraintes mémoire, les problèmes de partitionnement de graphes, d'hypergraphes et enfin à séparateur sommets.

2.3.1 Partitionnement de maillage sous contraintes mémoire

Dans cette section, nous formalisons le problème de partitionnement de maillage sous contraintes mémoire, introduit à la section 2.2, en utilisant la programmation linéaire en nombres entiers. Pour cela, nous introduisons les inconnues $x_{c,i}$ et $y_{d,i}$ telles que :

— $x_{c,i}$ vaut 1 si le sommet *calcul* $c \in C$ est traité par l'unité de calcul $i \in M$, et 0 sinon ;

— $y_{d,i}$ vaut 1 si le sommet *donnée* $d \in D$ est stocké en mémoire par l'unité de calcul $i \in M$ et 0 sinon.

Le problème peut alors s'exprimer à l'aide du programme linéaire en nombres entiers (PLNE) suivant :

Pb 1 : Partitionnement de maillage sous contraintes mémoire

$$
\begin{array}{llll}
\min & \mathbf{C_{max}} & & \\
\text{s.c.} & \sum_{i \in M} x_{c,i} = 1 & \forall c \in C & \text{(Pb 1.1)} \\
& \sum_{c \in C} x_{c,i}\ w_c(c) \leq \mathbf{C_{max}} & \forall i \in M & \text{(Pb 1.2)} \\
& \sum_{d \in D} y_{d,i}\ w_d(d) \leq \mathrm{Mem}[i] & \forall i \in M & \text{(Pb 1.3)} \\
& x_{c,i} \leq y_{d,i} & \forall c \in C, \forall d \in \mathcal{N}(c), \forall i \in M & \text{(Pb 1.4)} \\
& x_{c,i} \in \{0;1\} & \forall c \in C, \forall i \in M & \text{(Pb 1.5)} \\
& y_{d,i} \in \{0;1\} & \forall d \in D, \forall i \in M & \text{(Pb 1.6)}
\end{array}
$$

La première ligne est la fonction objectif du PLNE, à savoir que l'on cherche une distribution des sommets *calcul*, minimisant la charge de calcul de l'unité de calcul la plus chargée. Nous avons ensuite différentes contraintes relatives à la distribution et à la mémoire :

— la contrainte (Pb 1.1) impose que chaque sommet *calcul* soit affecté à une unité de calcul. De plus, puisque la contrainte (Pb 1.5) impose que $x_{c,i}$ soit booléenne, $\forall c \in C$ et $\forall i \in M$, l'affectation se fait sur une et une seule unité de calcul ;

— la contrainte (Pb 1.2) impose que le coût des calculs associés à chaque unité de calcul soit majoré par C_{max} ;

— la contrainte (Pb 1.3) impose que l'occupation mémoire des données nécessaires à chaque unité de calcul i ne dépasse pas sa capacité mémoire $\text{Mem}[i]$;

— la contrainte (Pb 1.4) impose que si une unité de calcul se voit affecter le sommet *calcul* c, alors elle doit disposer localement des données nécessaires aux calculs de c, contenues par les sommets *donnée* voisins de c ;

— les contraintes (Pb 1.5) et (Pb 1.6) imposent que les variables $x_{c,i}$ et $y_{d,i}$ ont pour valeur 0 ou 1.

Une solution de ce PLNE optimise le coût calculatoire grâce aux variables $x_{c,i}$, tout en assurant que la taille mémoire, incluant les mailles fantômes, ne dépasse pas la capacité mémoire de chaque unité de calcul, grâce aux variables auxiliaires $y_{d,i}$.

2.3.2 k-partitionnement de graphe

Dans cette section, nous formalisons le problème de partitionnement de graphe, introduit en section 2.1.1, en utilisant la programmation quadratique en nombres entiers. En effet, puisque la fonction objectif intervenant dans ce problème évalue des relations entre couples de sommets, nous la modélisons comme une fonction quadratique. Nous introduisons l'inconnue $x_{v,i}$ qui vaut 1 si le sommet v appartient à la partie V_i, et 0 sinon. Le programme quadratique en nombres entiers (PQNE) proposé est alors le suivant :

Pb 2 : Partitionnement de graphe (programme quadratique)

$$\min \quad \frac{1}{2} \sum_{i=1}^{k} \sum_{\{v,v'\} \in E} x_{v,i} \, (1 - x_{v',i}) \, \omega_e(\{v, v'\})$$

$$\text{s.c.} \quad \sum_{i=1}^{k} x_{v,i} = 1 \qquad \forall v \in V \qquad \text{(Pb 2.1)}$$

$$\sum_{v \in V} x_{v,i} \, w_v(v) \leq (1 + \mathcal{E}_{eq}) \, \frac{\sum_{v \in V} w_v(v)}{k} \qquad \forall i \in [\![1, k]\!], \; \mathcal{E}_{eq} \in [0, +\infty[\qquad \text{(Pb 2.2)}$$

$$x_{v,i} \in \{0; 1\} \qquad \forall v \in V, \forall i \in [\![1, k]\!] \qquad \text{(Pb 2.3)}$$

La contrainte (Pb 2.1) impose que chaque sommet $v \in V$ appartient à une et une seule partie. La contrainte (Pb 2.2) impose à toute partie $i \in [\![1, k]\!]$, que la somme des poids des sommets qu'elle contient respecte l'équilibre de partitionnement maximal. La contrainte (Pb 2.3) impose que les variables $x_{v,i}$ ont pour valeur 0 ou 1.

Remarquons qu'il est possible de linéariser ce PQNE en modélisant le comportement du produit $x_{v,i}(1 - x_{v',i})$ à l'aide de nouvelles variables et contraintes. L'idée est la suivante : pour modéliser le comportement qu'aurait le produit de deux inconnues booléennes x et y, on introduit une inconnue booléenne z à laquelle on impose un ensemble de contraintes dépendantes des inconnues x et y. Ces contraintes sont par exemple : $x \geq z$, $y \geq z$ et $x + y \leq z + 1$. Les deux premières contraintes imposent que si x ou y sont égaux à 0 alors z l'est aussi. La dernière contrainte impose que lorsque x et y valent 1 alors z aussi (ceci est vérifié dans la Table 2.1).

x	y	$x \geq z$	$y \geq z$	$x + y \leq z + 1$	z	xy
0	0	$\Rightarrow z = 0$	$\Rightarrow z = 0$	$\Rightarrow z = 0$ ou $z = 1$	0	0
0	1	$\Rightarrow z = 0$	$\Rightarrow z = 0$ ou $z = 1$	$\Rightarrow z = 0$ ou $z = 1$	0	0
1	0	$\Rightarrow z = 0$ ou $z = 1$	$\Rightarrow z = 0$	$\Rightarrow z = 0$ ou $z = 1$	0	0
1	1	$\Rightarrow z = 0$ ou $z = 1$	$\Rightarrow z = 0$ ou $z = 1$	$\Rightarrow z = 1$	1	1

TABLE 2.1 – Table de vérité du produit des inconnues x et y, des contraintes de l'inconnue z et de sa valeur finale.

Pour modéliser le comportement de $x_{i,p}\ (1 - x_{i',p})$, nous introduisons l'inconnue $z_{i,i',p}$ ainsi que les contraintes suivantes :

Contraintes supplémentaires

$$z_{v,v',i} \leq x_{v,i} \qquad \forall e = \{v, v'\} \in E,\ \forall i \in [\![1, k]\!] \quad \text{(Pb 2.4)}$$

$$z_{v,v',i} \leq 1 - x_{v',i} \qquad \forall e = \{v, v'\} \in E,\ \forall i \in [\![1, k]\!] \quad \text{(Pb 2.5)}$$

$$z_{v,v',i} \geq x_{v,i} + (1 - x_{v',i}) - 1 \qquad \forall e = \{v, v'\} \in E,\ \forall i \in [\![1, k]\!] \quad \text{(Pb 2.6)}$$

$$z_{v,v',i} \in \{0; 1\} \qquad \forall e = \{v, v'\} \in E,\ \forall i \in [\![1, k]\!] \quad \text{(Pb 2.7)}$$

Ces contraintes étant définies, nous pouvons transcrire le problème de k-partitionnement de graphe sous la forme d'un PLNE. Pour cela, il suffit d'ajouter les contraintes (Pb 2.4) à (Pb 2.7) au programme quadratique précédent, et d'en modifier la fonction objectif en remplaçant $x_{i,p}\ (1 - x_{i',p})$ par $z_{i,i',p}$. Nous obtenons ainsi le PLNE :

Pb 2 : Partitionnement de graphe (programme linéaire)

$$
\begin{array}{llll}
\min & \frac{1}{2}\sum_{i=1}^{k} \sum_{\{v,v'\}\in E} z_{v,v',i}\ \omega_e(\{v,v'\}) & & \\
\text{s.c.} & \sum_{i=1}^{k} x_{v,i} = 1 & \forall v \in V & \text{(Pb 2.1)} \\
 & \sum_{v\in V} x_{v,i}\ w_v(v) \leq (1+\mathcal{E}_{eq})\ \frac{\sum_{v\in V} w_v(v)}{k} & \forall i \in [\![1,k]\!],\ \mathcal{E}_{eq} \in [0,+\infty[& \text{(Pb 2.2)} \\
 & x_{v,i} \in \{0;1\} & \forall v \in V, \forall i \in [\![1,k]\!] & \text{(Pb 2.3)} \\
 & z_{v,v',i} \leq x_{v,i} & \forall e = \{v,v'\} \in E,\ \forall i \in [\![1,k]\!] & \text{(Pb 2.4)} \\
 & z_{v,v',i} \leq 1 - x_{v',i} & \forall e = \{v,v'\} \in E,\ \forall i \in [\![1,k]\!] & \text{(Pb 2.5)} \\
 & z_{v,v',i} \geq x_{v,i} + (1 - x_{v',i}) - 1 & \forall e = \{v,v'\} \in E,\ \forall i \in [\![1,k]\!] & \text{(Pb 2.6)} \\
 & z_{v,v',i} \in \{0;1\} & \forall e = \{v,v'\} \in E,\ \forall i \in [\![1,k]\!] & \text{(Pb 2.7)}
\end{array}
$$

La linéarisation de ce modèle a pour inconvénient d'ajouter un grand nombre de variables booléennes et de contraintes au modèle original. En effet, nous pouvons constater que la linéarisation ajoute $2 \times |E| \times k$ variables ainsi que 4 contraintes par nouvelle variable (voir contraintes (Pb 2.4) à (Pb 2.7)). Nous avons malgré tout voulu présenter cette linéarisation puisque celle-ci permet, entre autres choses, de pouvoir aborder la résolution du PQNE en utilisant une méthode de résolution de programme linéaire.

2.3.3 k-Partitionnement d'hypergraphe

Dans cette section, nous formalisons le problème de partitionnement d'hypergraphe sous la forme d'un PLNE. Pour cela, nous introduisons les deux inconnues $x_{i,p}$ et $z_{e,p}$ telles que :

— $x_{v,i}$ vaut 1 si le sommet v appartient à la partie V_i, et 0 sinon ;

— $z_{e,i}$ est égale à 1 si un sommet connecté à l'hyper-arête e appartient à la partie V_i, et 0 sinon.

Le problème peut alors se modéliser à l'aide du PLNE suivant :

Pb 3 : Partitionnement d'hypergraphe

$$
\begin{array}{llll}
\min & \displaystyle\sum_{e\in\mathcal{E}(H)} \left((\sum_{i=1}^{k} z_{e,i}) - 1 \right) \omega_{\mathcal{E}}(e) & & \\
\text{s.c.} & \displaystyle\sum_{i=1}^{k} x_{v,i} = 1 & \forall v \in V & \text{(Pb 3.1)} \\
 & z_{e,i} \geq x_{v,i} & \forall e \in \mathcal{E}, \forall v \in e, \forall i \in [\![1,k]\!] & \text{(Pb 3.2)} \\
 & \displaystyle\sum_{v\in V} x_{v,i}\ \omega_v(v) \leq (1+\mathcal{E}_{eq}) \frac{\sum_{v\in V} w_v(v)}{k} & \forall i \in [\![1,k]\!], \mathcal{E}_{eq} \in [0,+\infty[& \text{(Pb 3.3)} \\
 & z_{e,i} \in \{0;1\} & \forall e \in \mathcal{E}, \forall i \in [\![1,k]\!] & \text{(Pb 3.4)} \\
 & x_{v,i} \in \{0;1\} & \forall v \in V, \forall i \in [\![1,k]\!] & \text{(Pb 3.5)}
\end{array}
$$

La contrainte (Pb 3.1) impose que chaque sommet $v \in V$ appartient à une et une seule partie. La contrainte (Pb 3.2) impose que si l'un des sommets v de l'hyper-arête $e \in \mathcal{E}$ appartient à la partie $i \in [\![1,k]\!]$, alors $z_{e,i}$ est égale à 1. Cette contrainte n'interdit pas à $z_{e,i}$ d'être égale à 1 dans le cas où aucun des sommets de l'hyper-arête $e \in \mathcal{E}$ n'appartient à la partie i (c'est-à-dire $\sum_{v\in e} x_{v,i} = 0$). L'ajout d'une contrainte de la forme $z_{e,i} \leq \sum_{v\in e} x_{v,i}, \forall e \in \mathcal{E}$, n'est cependant pas nécessaire car elle serait redondante avec la fonction objectif du problème. En effet, puisque la fonction objectif minimise la métrique $(\lambda - 1)$, une solution optimale à ce problème ne sera pas composée d'une variable $z_{e,i}$ égale à 1 dans le cas où aucun des sommets de l'hyper-arête $e \in \mathcal{E}$ n'appartient à la partie i. La contrainte (Pb 3.3) impose à toute partie $i \in [\![1,k]\!]$, que la somme des poids des sommets qu'elle contient respecte la condition d'équilibrage. Finalement, les contraintes (Pb 3.4) et (Pb 3.5) imposent aux variables $x_{v,i}$ et $z_{e,i}$ d'avoir pour valeur 0 ou 1.

Remarquons que nous avons directement exprimé ce problème sous la forme d'un PLNE, contrairement à la section précédente où nous avons eu recours à une étape de linéarisation pour passer d'un PQNE à un PLNE. Ceci est la conséquence de l'évaluation directe de la coupe à partir des hyper-arêtes du graphe et non pas à partir de couples de sommets. Pour cela, nous avons introduit les inconnues $z_{e,i}$ ainsi que la contrainte (Pb 3.2) pour rendre cohérente l'évaluation de la coupe. Nous pouvons d'ailleurs remarquer que le PLNE précédent peut être utilisé pour traduire le problème de partitionnement de graphe.

2.3.4 Partitionnement à séparateur sommets

Finalement, nous énonçons le problème de partitionnement à séparateur sommets sous la forme d'un PLNE. Pour cela, nous introduisons l'inconnue $x_{v,i}$, qui vaut 1 si le sommet v appartient à la

partie V_i et 0 sinon. Le programme linéaire en nombres entiers peut s'écrire comme suit :

Pb 4 : Partitionnement à séparateur sommets

$$\begin{array}{llll}
\min & \sum_{v \in V} x_{v,k+1} & & \\
\text{s.c.} & \sum_{i=1}^{k+1} x_{v,i} = 1 & \forall v \in V & \text{(Pb 4.1)} \\
& x_{v,i} + x_{v',i'} \leq 1 & \forall \{v, v'\} \in E, \forall i \in [\![1,k]\!], & \text{(Pb 4.2)} \\
& & i' \in [\![1,k]\!], i \neq i' & \\
& \sum_{v \in V} x_{v,i}\ \omega_v(v) \leq (1 + \mathcal{E}_{eq})\ (1 - x_{v,k+1})\ \frac{\sum_{v \in V} w_v(v)}{k} & \forall i \in [\![1,k]\!] & \text{(Pb 4.3)} \\
& x_{v,i} \in \{0;1\} & \forall v \in V, \forall i \in [\![1,k]\!] & \text{(Pb 4.4)}
\end{array}$$

La contrainte (Pb 4.1) impose que chaque sommet $v \in V$ appartient à une et une seule partie. La contrainte (Pb 4.2) impose que si deux sommets v et v' sont adjacents et v appartient à la partie $i \in [\![1,k]\!]$ alors v' appartient à la partie i ou $k+1$. La contrainte (Pb 4.3) impose à toute partie $i \in [\![1,k]\!]$, que la somme des poids des sommets qu'elle contient respecte l'équilibre de partitionnement maximal. Enfin, la contrainte (Pb 4.4) impose que les variables $x_{v,i}$ prennent pour valeur 0 ou 1.

Les problèmes classiques de partitionnement et le problème de partitionnement de maillage sous contraintes mémoire ayant été modélisés sous forme de PLNE et PQNE, nous allons maintenant les résoudre de manière exacte sur des instances de petites tailles.

2.4 Études comparative des problèmes de partitionnement

Dans cette section nous étudions les différences entre les solutions optimales des problèmes de partitionnement de graphe, d'hypergraphe, à séparateur sommets, et de maillage sous contraintes mémoire. Pour cela, nous résolvons de manière exacte les programmes linéaires et quadratiques présentés en section 2.3, en utilisant l'outil Cplex [18]. Notons que nous avons choisi d'utiliser cet outil de manière arbitraire et que nous aurions pu utiliser d'autres outils tels que Xpress [33] ou Gurobi [34]. Lors de la résolution des problèmes de partitionnement classiques, nous fixons la valeur de déséquilibre maximal autorisé $\mathcal{E}_{eq}$ à 0.05. Le problème de partitionnement de maillage sous contraintes mémoire étant composé d'une contrainte liée à la capacité des unités de calcul, nous fixons la capacité de celles-ci. Pour cela, nous commençons par déterminer la capacité mémoire minimale Mem_{min}, homogène à chaque unité de calcul et nécessaire à l'existence d'une solution.

Pour déterminer la capacité minimale Mem_{min} nous avons résolu, pour chaque expérience, le PLNE suivant :

$$
\begin{array}{llll}
\min & \text{Mem}_{min} & & \\
\text{s.c.} & \sum_{i \in M} x_{c,i} = 1 & \forall c \in C & (2.6) \\
 & \sum_{d \in D} y_{d,i}\ w_d(d) \leq \text{Mem}_{min} & \forall i \in M & (2.7) \\
 & x_{c,i} \leq y_{d,i} & \forall c \in C, \forall d \in \mathcal{N}(c), \forall i \in M & (2.8) \\
 & x_{c,i} \in \{0;1\} & \forall c \in C, \forall i \in M & (2.9) \\
 & y_{d,i} \in \{0;1\} & \forall d \in D, \forall i \in M & (2.10)
\end{array}
$$

Notons que ce PLNE est similaire au PLNE du problème de partitionnement de maillage sous contraintes mémoire (voir page 33). En effet, nous pouvons remarquer la présence des contraintes 1.1, 1.4, 1.5 et 1.6.

Afin de disposer d'un espace de solutions assez large, nous fixons capacité des unités de calcul à $1.2 \times \text{Mem}_{min}$. Afin de pouvoir analyser la qualité des solutions obtenues, nous introduisons la borne inférieure LB telle que :

$$LB = \lceil \sum_{c \in C} \omega_c(c)\ /|M| \rceil. \tag{2.11}$$

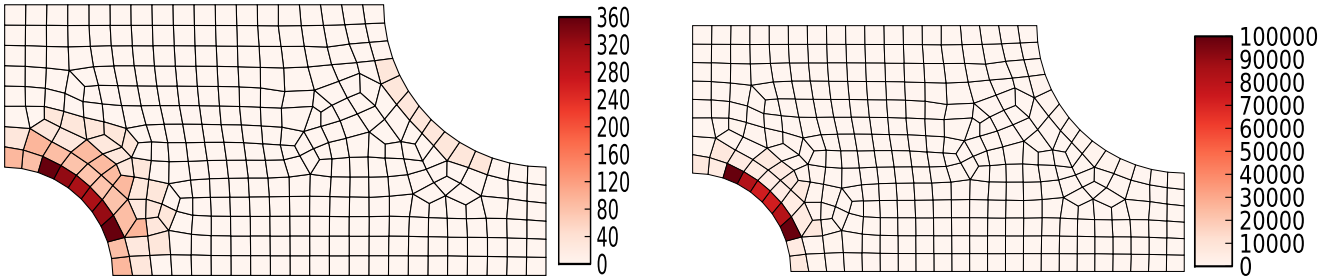
Celle-ci correspond au poids moyen d'une partie arrondi à la valeur entière supérieure, c'est-à-dire la partie entière supérieure de la somme des coûts de calcul, divisée par le nombre d'unités de calcul.

Pour nos expériences, nous considérons une discrétisation, notée $M_{q,\Omega}$, d'un domaine géométrique Ω composé de quadrilatères (voir la figure 2.7). Ce maillage est composé de 312 mailles quadrangulaires et nous étudierons successivement trois types de répartitions mémoires identifiés par les termes *pic* (pour *particle in cell* en anglais), *unitaire* et *aléatoire*. La première consiste en une répartition très hétérogène des poids mémoire (voir section 2.4.1 et 2.4.2), la seconde en une répartition unitaire (voir section 2.4.3) et la dernière en une répartition aléatoire (voir section 2.4.4). Pour chacune de ces répartitions, nous associons au maillage une répartition des calculs où le coût calculatoire d'une maille est égal au carré de son poids mémoire [3].

3. Cette relation entre le coût calculatoire et le poids mémoire est arbitraire. Nous voulions une relation calcul/mémoire non linéaire.

2.4.1 Exemple d'une répartition *pic*

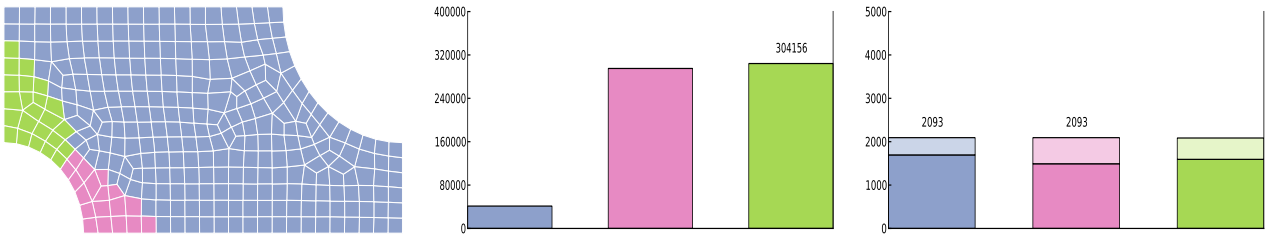
Nous commençons par étudier le maillage $M_{q,\Omega}$ et souhaitons obtenir une partition de celui-ci sur trois unités de calcul. Nous considérons un voisinage par arête à distance un, c'est-à-dire un voisinage où deux mailles sont dites voisines si elles partagent une arête commune. Les répartitions des poids mémoires et des coûts calculatoires associés à $M_{q,\Omega}$ sont respectivement illustrées sur les figures 2.7(a) et 2.7(b).



(a) (b)

FIGURE 2.7 – Maillage $M_{q,\Omega}$ dont la répartition des poids mémoires est présentée en (a) et la répartition des coûts calculatoires est présentée en (b).

Ces répartitions très hétérogènes sont caractéristiques des simulations de physique particulaire [15]. Préalablement à nos comparaisons, nous avons déterminé que la capacité mémoire minimale Mem_{min} était égale à 2093. Pour cela, nous avons utilisé l'outil CPLEX pour résoudre le PLNE introduit en page 39. La capacité mémoire des unités de calcul est ainsi fixée à 2511. Une partition respectant la capacité mémoire minimale Mem_{min} est présentée en figure 2.8(a). Les coûts mémoires et calculatoires de chaque unité de calcul sont présentés en figures 2.8(b) et 2.8(c).



(a) (b) (c)

FIGURE 2.8 – Solution optimale déterminant la capacité mémoire homogène minimale nécessaire à l'existence d'une solution pour $M_{q,\Omega}$. La partition associée au maillage est présentée en (a) et les charges calculatoires et mémoires des unités de calcul sont respectivement présentées en (b) et (c). La charge mémoire induite par les mailles locales est en couleur foncée et la charge mémoire induite par les mailles fantômes est en couleur claire.

Partitionnement de graphe

Commençons par nous intéresser au problème de partitionnement de graphe. Pour cela, nous commençons par modéliser le maillage $M_{q,\Omega}$ à l'aide de son graphe dual $G = (V, E)$. Chaque sommet $i \in V$ modélise une maille $i \in M_{q,\Omega}$ et chaque arête $e = \{i, j\} \in E$ modélise la relation de voisinage entre les mailles i et j. Chaque sommet $i \in V$ est pondéré par un coût calculatoire égal au coût de la maille i qu'il modélise. Chaque arête $e = \{i, j\} \in E$ est pondérée par un poids mémoire égal à la moyenne des poids mémoires des mailles i et j. Une solution optimale obtenue pour le problème de partitionnement de graphe est présentée à la figure 2.9(a) et les coûts mémoires et calculatoires associés à chaque unité de calcul sont respectivement présentés sur les figures 2.9(b) et 2.9(c). Pour une meilleure visualisation des mailles fantômes associées à chaque partie, nous présentons une vue éclatée de la partition à la figure 2.9(d). Les mailles fantômes nécessaires à chaque unité de calcul apparaissent en couleurs plus claires.

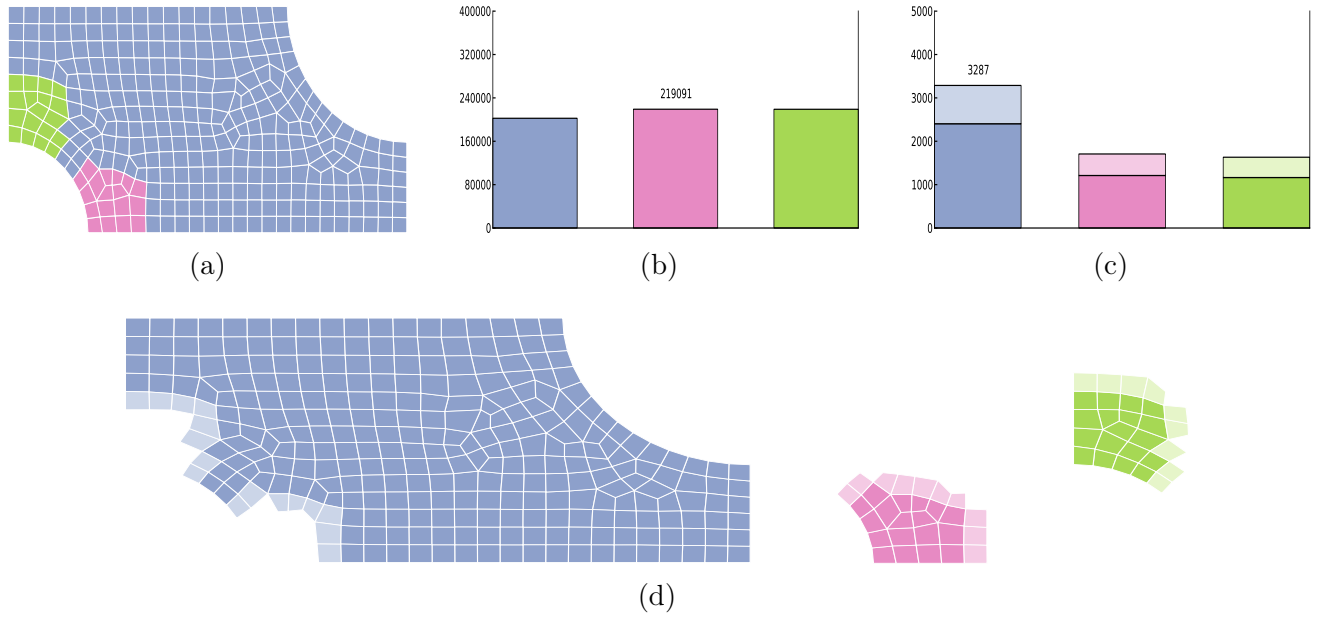


FIGURE 2.9 – Partition induite par la solution du problème de partitionnement de graphe. La partition associée au maillage est présentée en (a) et les charges calculatoires et mémoires des unités de calcul sont respectivement présentées en (b) et (c). La distribution des données est illustrée en (d), où les mailles fantômes sont présentées en couleurs claires pour chaque unité de calcul.

Nous remarquons que les coûts calculatoires associés à chaque partie sont équilibrés mais que la distribution mémoire associée est très irrégulière. Ce constat peut d'ailleurs être fait sans prendre en compte les mailles fantômes. Les unités de calcul ayant une capacité mémoire bornée par 2511, la solution obtenue ne respecte pas la contrainte mémoire.

Partitionnement d'hypergraphe

Intéressons nous maintenant au problème de partitionnement d'hypergraphe. Pour cela, nous commençons par modéliser le maillage par un hypergraphe $H = (V, \mathcal{E})$. Chaque sommet $i \in V$ modélise une maille $i \in M_{q,\Omega}$ et chaque hyper-arête $h_i \in \mathcal{E}$ modélise la relation de voisinage entre la maille i et ses mailles voisines. Chaque sommet $i \in V$ est pondéré par un coût calculatoire égal au coût de la maille i qu'il modélise. Chaque hyper-arête $h_i \in \mathcal{E}$ est pondérée par le poids mémoire de la maille i dont elle modélise le voisinage. La solution optimale obtenue pour le partitionnement d'hypergraphe est présentée en figure 2.10(a) et les coûts mémoires et calculatoires associés à chaque unité de calcul sont respectivement présentés dans les figures 2.10(b) et 2.10(c). Nous présentons une vue éclatée de la partition à la figure 2.10(d).

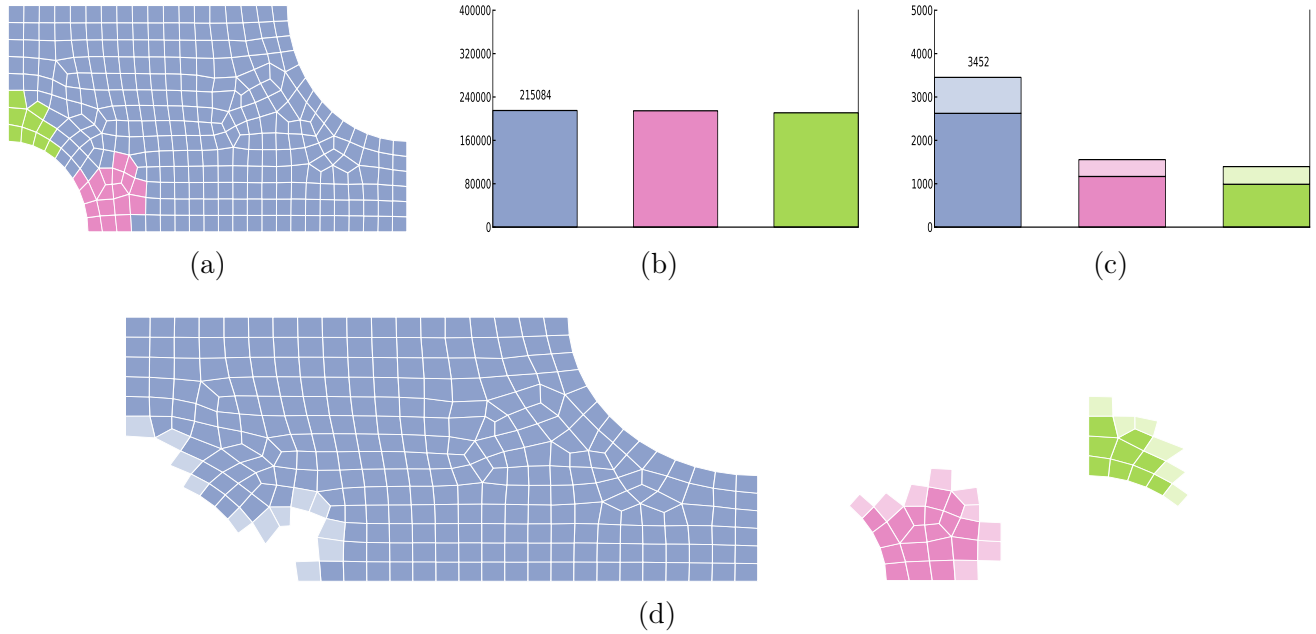
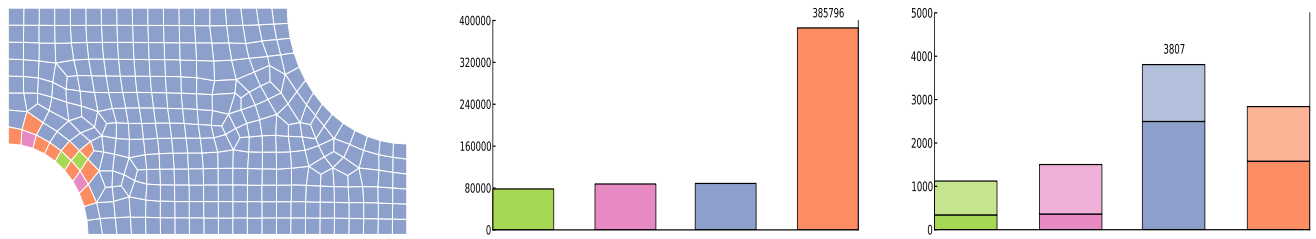


FIGURE 2.10 – Partition induite par la solution du problème de partitionnement d'hypergraphe. La partition associée au maillage est présentée en (a) et les charges calculatoires et mémoires des unités de calcul sont respectivement présentées en (b) et (c). La distribution des données est illustrée en (d), où les mailles fantômes sont présentées en couleurs claires pour chaque unité de calcul.

Comme pour la solution optimale du partitionnement de graphe, la solution obtenue pour le partitionnement d'hypergraphe associe un coût calculatoire par partie bien équilibré. De même, nous constatons que la distribution mémoire est ici aussi très irrégulière.

Partitionnement par séparateur sommets

Intéressons-nous maintenant au partitionnement à séparateur sommets. Rappelons que la fonction objectif de ce problème diffère des problèmes de partitionnement précédents. L'objectif n'est pas de minimiser la coupe induite par la partition mais de minimiser la taille du séparateur. La solution optimale obtenue est présentée en figure 2.11(a) et les coûts mémoires et calculatoires associés à chaque unité de calcul ainsi qu'au séparateur sont repectivement présentés dans les figures 2.11(b) et 2.11(c).

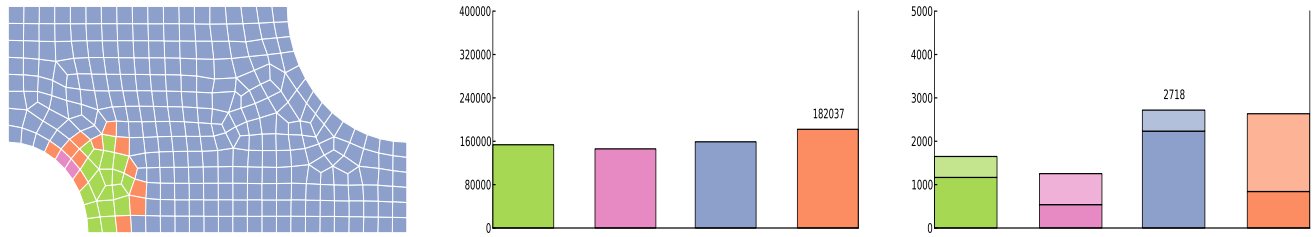


(a) (b) (c)

FIGURE 2.11 – Partition induite par la solution du problème de partitionnement par séparateur sommets. La partition associée au maillage est présentée en (a) et les charges calculatoires et mémoires des unités de calcul et du séparateur sont respectivement présentées en (b) et (c).

Contrairement aux solutions optimales des problèmes de partitionnement de graphe et d'hypergraphe, la solution optimale actuelle est fortement déséquilibrée en termes de poids mémoires. De plus, nous pouvons constater que la charge calculatoire associée au séparateur est plus élevée que celle des unités de calcul. D'ailleurs, la charge mémoire associée au séparateur est elle aussi plus élevée que celle de deux unités de calcul. Partant de cette observation, nous nous intéressons à une variation du partitionnement à séparateur sommets (notée PSP). Celle-ci consiste à introduire la pondération des sommets dans la métrique d'évaluation du séparateur. Ainsi, la fonction objectif du problème n'est plus la minimisation de la taille du séparateur sommets mais la minimisation de la somme des poids mémoires des sommets qui le composent. La solution obtenue pour cette variation est présentée en figure 2.12(a). Les coûts mémoires et calculatoires associés à chaque unité de calcul ainsi qu'au séparateur sont respectivement présentés dans les figures 2.12(b) et 2.12(c).

Nous constatons que cette solution est, elle aussi, déséquilibrée en termes de poids mémoires. De plus, bien que la charge calculatoire du séparateur y soit plus faible que précédemment, celle-ci, ainsi que la charge mémoire du séparateur, sont supérieures à celles d'une des unités de calcul.

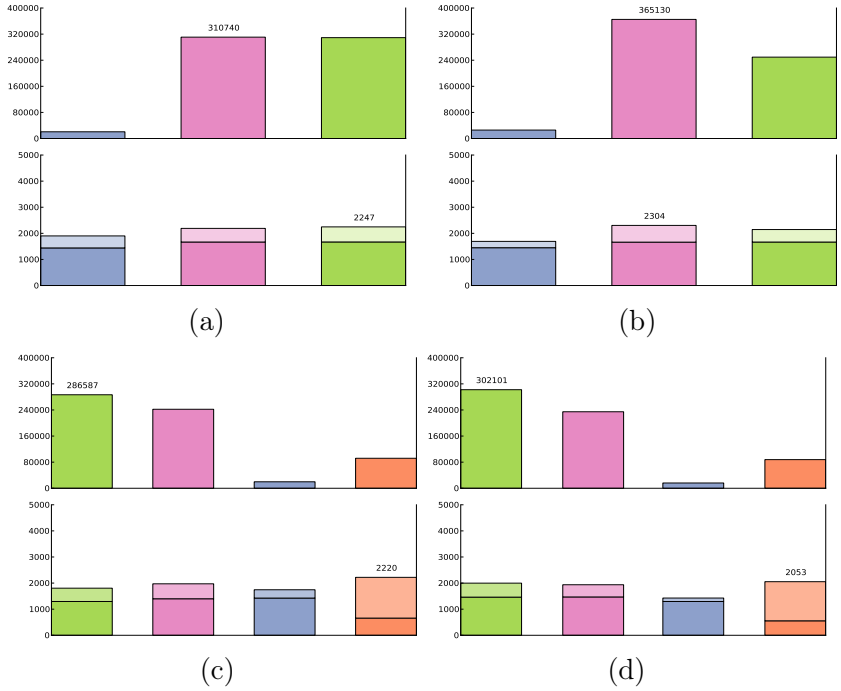


(a) (b) (c)

FIGURE 2.12 – Partition induite par la solution du problème PSP. La partition associée au maillage est présentée en (a) et les charges calculatoires et mémoires des unités de calcul et du séparateur sont respectivement présentées en (b) et (c).

Changement de pondération pour les problèmes précédents

Chaque solution optimale des problèmes de partitionnement étudiés est invalide vis-à-vis de la contrainte mémoire. En pratique, il est possible d'améliorer la gestion de la distribution mémoire pour ces problèmes. Pour cela, les sommets du graphe ou de l'hypergraphe ne sont plus pondérés par le coût calculatoire de la maille qu'ils modélisent, mais par le poids mémoire de celle-ci. Nous opérons cette modification et présentons en figure 2.13 les distributions mémoires et calculatoires des solutions optimales aux problèmes précédents.



(a) (b) (c) (d)

FIGURE 2.13 – Illustration des charges calculatoires et mémoires des solutions optimales, obtenues après modification de la pondération des sommets, pour les problèmes de partitionnement de graphe en (a), d'hypergraphe en (b), par séparateur sommets en (c), PSP en (d). Pour chaque problème, les charges calculatoires (respectivement mémoires) sont présentées sur la figure du haut (respectivement du bas).

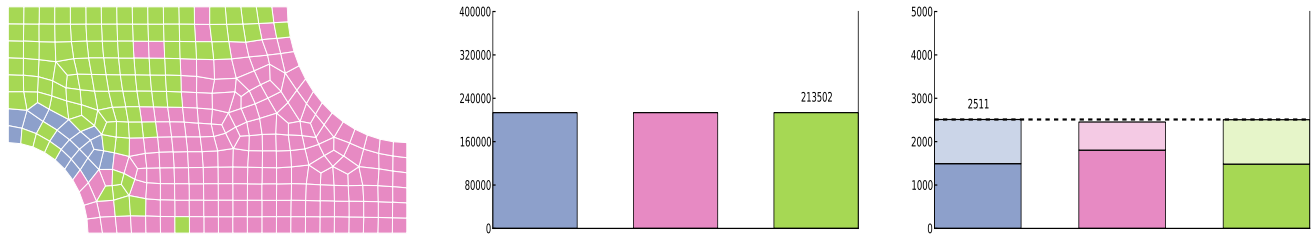
Nous constatons que les solutions des problèmes de partitionnement de graphe et d'hypergraphe sont valides vis-à-vis de la contrainte mémoire. En revanche, elles induisent un coût de restitution supérieur aux solutions obtenues avant la modification des pondérations.

Dans le cas du partitionnement par séparateur sommets, nous remarquons que la distribution mémoire sans prise en compte des mailles fantômes est mieux équilibrée. En les prenant en compte, les solutions du partitionnement par séparateur sommets classique et du PSP sont là aussi mieux équilibrées. Cependant, pour chacun de ces cas les charges calculatoires sont très déséquilibrées.

À ce stade, nous pouvons constater que les solutions obtenues pour les problèmes de partitionnement par séparateur sommets ne sont pas directement comparables avec les solutions des autres problèmes. Il est en effet nécessaire de réaliser une étape de post-traitement afin de distribuer les calculs des sommets du séparateur. Il est donc difficile de dire si la modification des pondérations a permis d'obtenir des solutions valides vis-à-vis du critère mémoire ou non.

Partitionnement de maillage sous contraintes mémoire

Après les résultats obtenus pour les approches classiques de partitionnement, nous présentons maintenant une solution optimale du partitionnement de maillage sous contraintes mémoire. Celle-ci est présentée en figure 2.14(a) et les coûts mémoires et calculatoires associés à chaque unité de calcul sont respectivement présentés dans les figures 2.14(b) et 2.14(b).



(a) (b) (c)

FIGURE 2.14 – Partition induite par la solution du problème de partitionnement de maillage sous contraintes mémoire. La partition associée au maillage est présentée en (a) et les charges calculatoires et mémoires des unités de calcul sont respectivement présentées en (b) et (c).

Cette solution respecte la contrainte mémoire et équilibre la charge calculatoire des unités de calcul. En nous intéressant à l'instance étudiée, nous constatons que la borne inférieure LB est égale à 213 495. Le makespan de la solution étant égal à 213 502, nous pouvons remarquer que la prise en compte des contraintes mémoire n'induit pas forcément une forte détérioration de la qualité du makespan pouvant être obtenu. En effet, celui de la solution actuelle est inférieur à $1.01 \times LB$.

Nous résumons dans la table 2.2 les résultats obtenus pour le makespan et la distribution mémoire, avec et sans prise en compte des mailles fantômes, en résolvant les problèmes de partitionnement de graphe, d'hypergraphe, et de maillage sous contraintes mémoire. Les solutions obtenues pour les problèmes de partitionnement par séparateur sommets n'étant pas directement exploitables, elles ne sont pas rappelées dans ce tableau.

Problème	**Makespan**	**Distribution mémoire sans fantômes**	**Distribution mémoire avec fantômes**
Partitionnement de graphe équilibrage du coût calculatoire	219 091	$(2404, 1211, 1164)$	$(3287, 1707, 1632)$
Partitionnement d'hypergraphe équilibrage du coût calculatoire	215 084	$(2623, 1168, 988)$	$(3452, 1556, 1395)$
Partitionnement de graphe équilibrage du poids mémoire	310 740	$(1441, 1668, 1670)$	$(1901, 2190, 2247)$
Partitionnement d'hypergraphe équilibrage du poids mémoire	365 130	$(1449, 1663, 1667)$	$(1692, 2304, 2145)$
Partitionnement de maillage sous contrainte mémoire	213 502	$(1493, 1804, 1482)$	$(2511, 2453, 2506)$

TABLE 2.2 – Tableau énumérant le makespan et la distribution mémoire avec et sans prise en compte des mailles fantômes, des problèmes de partitionnement de graphe, d'hypergraphe et de maillage sous contraintes mémoire. Dans le cas du makespan, les valeurs en rouge représentent des valeurs que nous considérons comme trop élevées. Dans le cas de la distribution mémoire, les valeurs en rouge représentent des valeurs qui dépassent la capacité des unités de calcul.

Pour cette première expérience, nous pouvons constater que notre modèle permet d'obtenir une solution dont le makespan est inférieur aux solutions des approches classiques. De plus, nous remarquons que ces approches ne respectent pas toujours la contrainte mémoire. Il est cependant possible d'améliorer la manière dont est gérée la mémoire. Pour cela, les sommets du graphe ou de l'hypergraphe ne sont plus pondérés par un coût calculatoire mais par un poids mémoire. En appliquant cette modification, les solutions optimales obtenues pour les problèmes de partitionnement de graphe et d'hypergraphe respectent la contrainte mémoire. Nous constatons cependant que le makespan induit par ces solutions est alors plus élevé qu'avant la modification des pondérations. Les solutions des problèmes de partitionnement à séparateur sommets n'étant pas exploitables en pratique, nous ne les présenterons pas pour les expériences suivantes.

2.4.2 Exemple avec un voisinage à distance deux

Dans cette section nous cherchons à obtenir une partition du maillage $M_{q,\Omega}$ sur 3 unités de calcul, où la répartition est encore de type *pic*, en considérant un voisinage par arête à distance 2. Ce voisinage définit que deux mailles sont voisines si elles partagent une arête commune ou partagent une arête avec la même maille. La capacité mémoire minimale Mem_{min} de cette instance étant égale à 2690, nous fixons la capacité des unités de calcul à $1.2 \times \text{Mem}_{min} = 3228$.

Nous présentons de manière compacte les solutions optimales obtenues en traitant cette nouvelle instance. Ces résultats sont illustrés en figure 2.15 où :

— la première ligne présente les solutions optimales du partitionnement de maillage sous contraintes mémoires, et le partitionnement de graphe et d'hypergraphe lors de l'équilibrage du coût calculatoire des parties ;

— la seconde ligne présente les solutions optimales des problèmes de partitionnement de graphe et d'hypergraphe lors de l'équilibrage des poids mémoires des parties.

Pour cette seconde expérience, nous constatons que notre modèle permet, encore une fois, d'obtenir une solution dont le makespan est inférieur aux solutions des problèmes classiques de partitionnement. De plus, nous remarquons que les solutions optimales à ces problèmes ne respectent pas toujours la contrainte mémoire. La charge mémoire induite par les mailles fantômes y est d'ailleurs plus élevée que dans l'exemple précédent, en allant jusqu'à représenter plus de 40% de la charge mémoire totale par partie. Lors de l'équilibrage de la charge mémoire par partie, la solution obtenue pour le partitionnement de graphe est valide vis-à-vis du critère mémoire. Ce n'est cependant pas le cas de la solution du partitionnement d'hypergraphe dont la première partie (en couleur bleue) a une charge mémoire dépassant la capacité mémoire des unités de calcul. L'absence de prise en compte du coût calculatoire des parties induit à nouveau une charge de calcul très déséquilibrée.

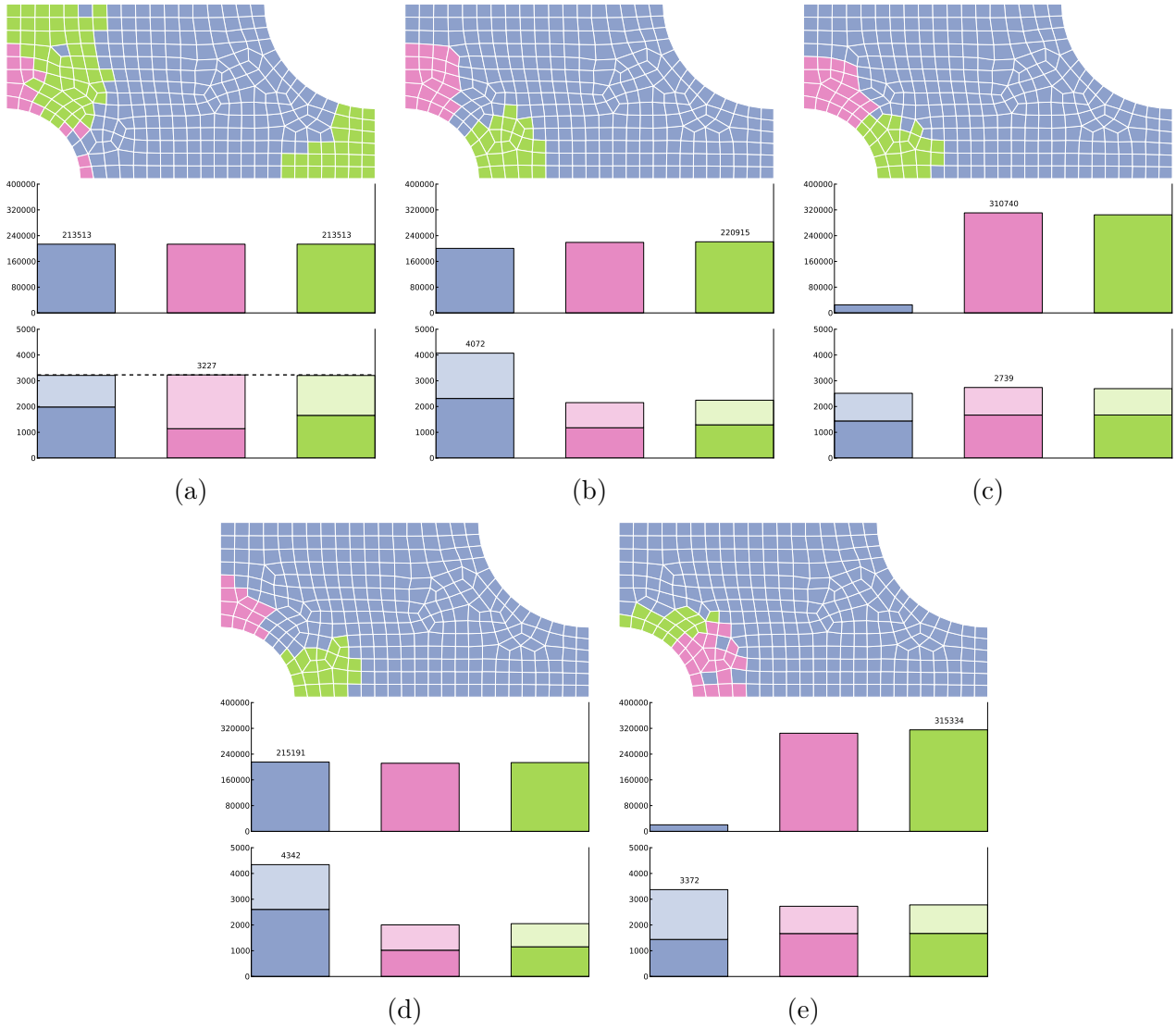


FIGURE 2.15 – Illustration des solutions optimales, et des charges calculatoires et mémoires associées à ses solutions, pour les problèmes de partitionnement : de maillage sous contraintes mémoires en (a) ; de graphes avec équilibrage du coût calculatoire en (b) et de la mémoire en (c) ; d'hypergraphes avec équilibrage du coût calculatoire en (d) et de la mémoire en (e).

2.4.3 Exemple d'une répartition *unitaire*

Dans cette section nous cherchons à obtenir une partition du maillage $M_{q,\Omega}$ sur trois unités de calcul, en considérant un voisinage par arête à distance 1. Contrairement aux sections précédentes, nous supposons maintenant que la répartition calculatoire et la répartition mémoire sont unitaires. Chaque maille se voit donc associer un coût calculatoire $\omega_c = 1$ et un poids mémoire $\omega_d = 1$. La capacité mémoire minimale Mem_{min} de cette instance étant égale à 120, nous fixons la capacité des unités de calcul à $1.2 \times \text{Mem}_{min} = 144$.

Les solutions optimales obtenues pour cette entrée sont présentées en figure 2.16, où appa-

raissent les solutions obtenues lors de l'équilibrage du coût calculatoire. La mémoire associée aux mailles étant égale au coût calculatoire qui leur est associé, les solutions optimales obtenues lors de l'équilibrage de la mémoire sont aussi optimales lors de l'équilibrage du coût.

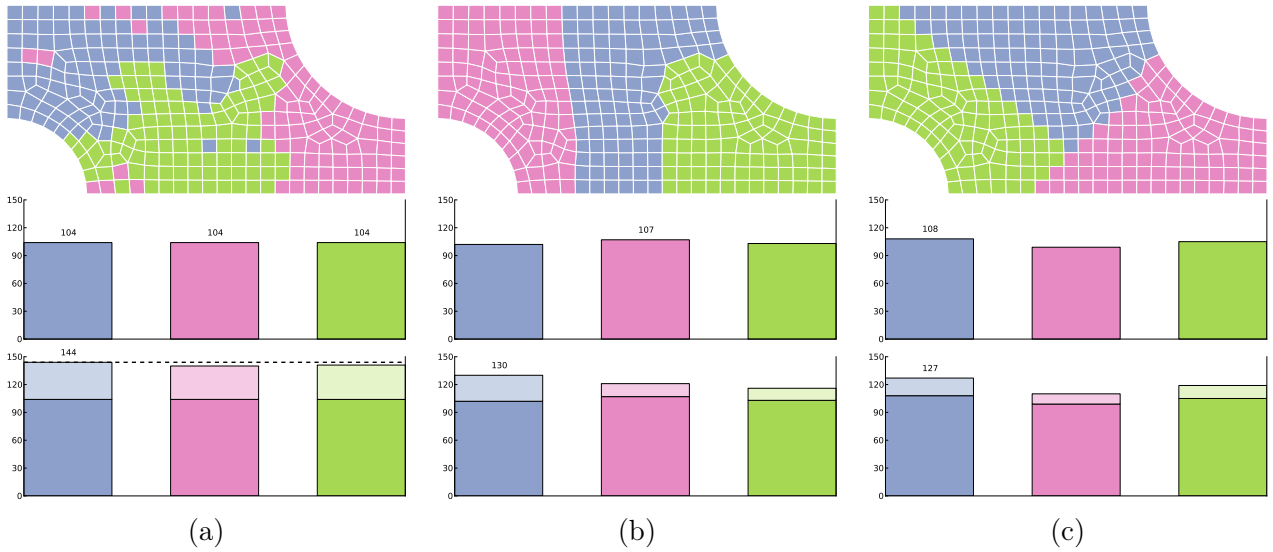
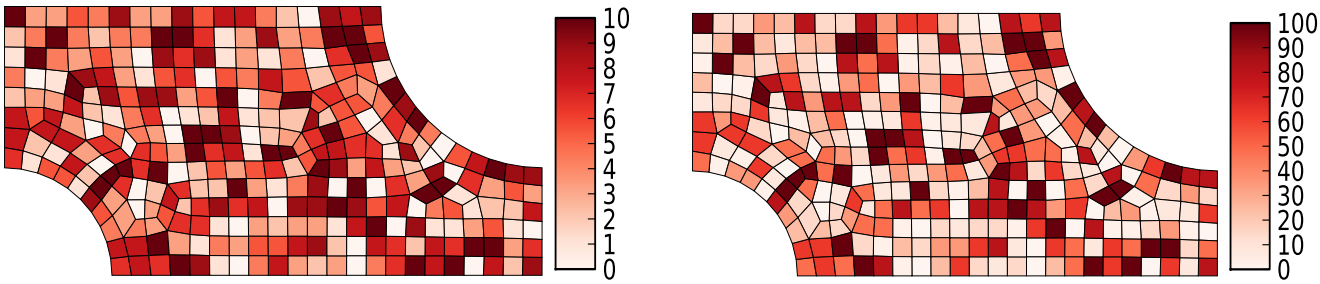


FIGURE 2.16 – Illustration des solutions optimales, et des charges calculatoires et mémoires associées à ces solutions, pour les problèmes de partitionnement : de maillage sous contraintes mémoires en (a) ; de graphes avec équilibrage du coût calculatoire en (b) ; d'hypergraphes avec équilibrage du coût calculatoire en (c).

Dans cette expérience, nous constatons que les solutions optimales des problèmes de partitionnement de graphe et d'hypergraphe sont valides vis-à-vis du critère mémoire. Le makespan induit par ces solutions est d'ailleurs de qualité similaire à celui obtenu par notre approche. Nous pouvons cependant constater que notre approche nous permet d'obtenir le meilleur makespan possible. En effet, la borne inférieure de celui-ci est $LB = (\sum_i \omega_c(i)/3, \max_i \omega_c(i)) = 104$, valeur atteinte par la solution optimale du problème de partitionnement de maillage sous contraintes mémoire.

2.4.4 Exemple d'une répartition *aléatoire*

Dans cette section nous cherchons à obtenir une partition du maillage $M_{q,\Omega}$ sur trois unités de calcul, en nous intéressant à un voisinage par arête à distance 1. Nous étudions une nouvelle répartition des poids mémoires et des coûts calculatoires que nous illustrons respectivement aux figures 2.17(a) et 2.17(b). La répartition mémoire a été obtenue en générant des valeurs entières aléatoires dans l'intervalle $[1; 10]$. La répartition calculatoire correspond au carré de la répartition mémoire.



(a) (b)

FIGURE 2.17 – Maillage $M_{q,\Omega}$ dont la répartition aléatoire des poids mémoire est présentée en (a) et la répartition des coûts calculatoires est présentée en (b).

La capacité mémoire minimale Mem_{min} de cette instance étant égale à 657, nous fixons la capacité des unités de calcul à $1.2 \times \text{Mem}_{min} = 788$. Les solutions optimales obtenues sont présentées en figure 2.18.

Nous constatons que dans cette expérience, les solutions optimales des approches classiques sont valides vis-à-vis du critère mémoire. Ces solutions sont d'ailleurs équilibrées aussi bien concernant la charge de calcul que le poids mémoire (avec ou sans prise en compte des fantômes). Le makespan induit par les approches classiques est meilleur dans le cas de l'équilibrage du coût calculatoire mais n'atteint pas la borne inférieure LB. Celle-ci est égale à 4169, valeur atteinte par la solution optimale du problème de partitionnement de maillage sous contraintes mémoire.

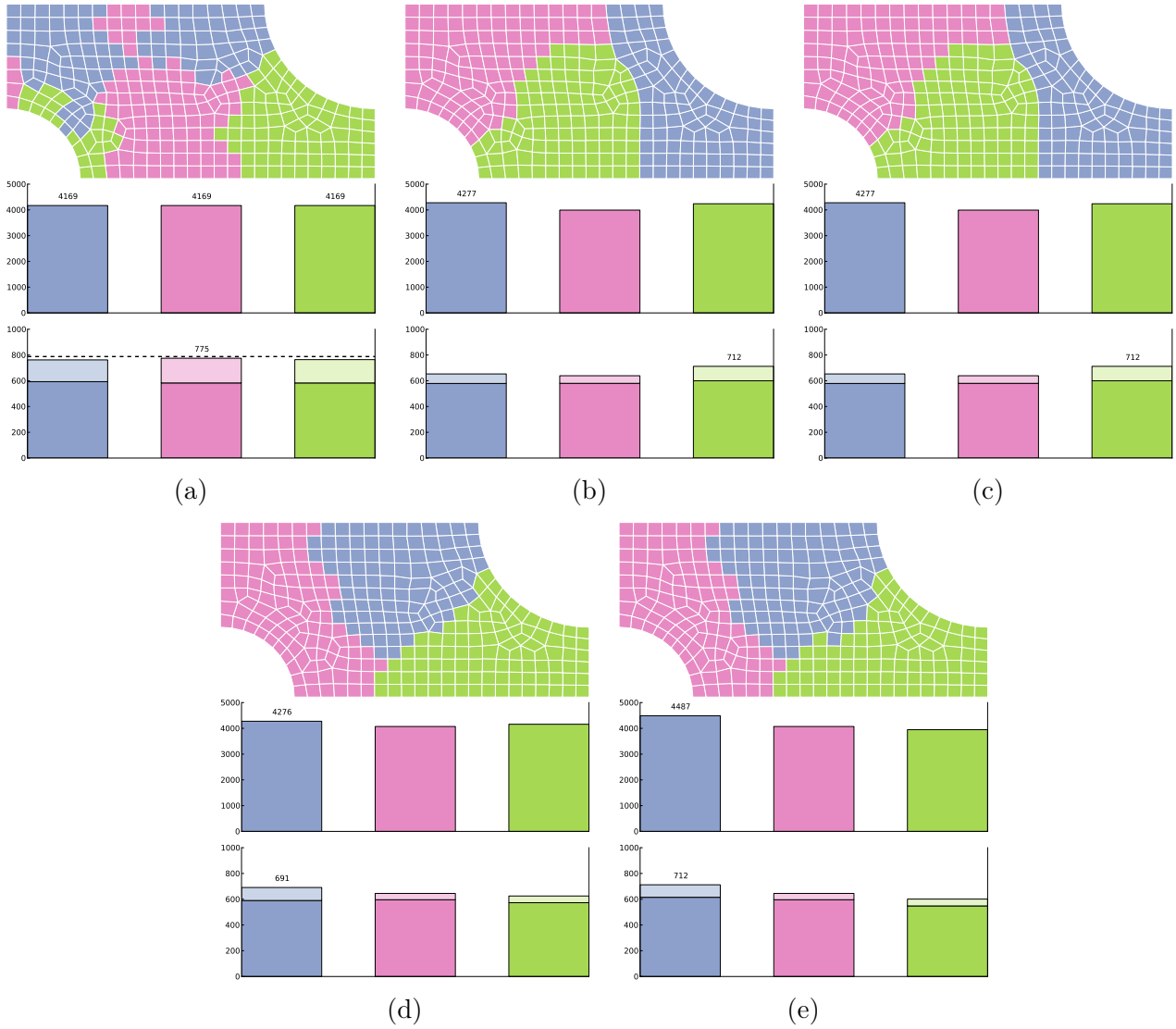


FIGURE 2.18 – Illustration des solutions optimales, et des charges calculatoires et mémoires associées à ces solutions, pour les problèmes de partitionnement : de maillage sous contraintes mémoires en (a) ; de graphes avec équilibrage du coût calculatoire en (b) et de la mémoire en (c) ; d'hypergraphes avec équilibrage du coût calculatoire en (d) et de la mémoire en (e).

2.5 Analyses et conclusion

Analyses

Dans ce chapitre nous avons présenté le problème de partitionnement de maillage sous contraintes mémoire et l'avons modélisé comme un programme linéaire en nombres entiers. Ce problème n'est pas directement étudié dans la littérature et les approches utilisées pour le résoudre répondent à d'autres problèmes de partitionnement. Nous avons rappelé lesdits problèmes et les avons modélisés sous la forme de programmes linéaires et quadratiques en nombres entiers. Cette étape

de modélisation nous a permis de mettre en évidence les différences entre ces problèmes, et leur résolution nous a permis d'en étudier des solutions optimales.

Nous avons ainsi pu constater que des partitions, obtenues après résolution exacte des problèmes de partitionnement de graphe et d'hypergraphe équilibrant la charge calculatoire des parties, ne sont pas automatiquement valides vis-à-vis des contraintes que nous considérons. Ces solutions induisent un makespan de bonne qualité, au sens qu'il est proche de la borne inférieure LB, mais elles peuvent induire des distributions mémoires très irrégulières. C'est en particulier le cas pour les entrées dont la répartition mémoire est très hétérogène (voir les sections 2.4.1 et 2.4.2). La distribution des données conduit alors à dépasser la capacité des unités de calcul lors de la prise en compte des mailles fantômes. Ce constat peut d'ailleurs parfois être fait sans même les prendre en compte !

Dans le cas où ces problèmes sont résolus en équilibrant la charge mémoire des parties, la gestion de la distribution mémoire est mieux maîtrisée. Les solutions ainsi obtenues peuvent donc plus facilement respecter les contraintes mémoire. Cependant, la capacité mémoire des unités de calcul n'étant pas explicitement prise en compte, les solutions ne respectent pas forcément ces contraintes (c'est notamment le cas de la solution illustrée en figure 2.15(e)). De plus, nous pouvons constater que l'amélioration de la gestion mémoire se fait au détriment de la qualité du makespan.

Une des raisons pour lesquelles les solutions obtenues en équilibrant la charge de calcul sont invalides est que la charge mémoire associée à une unité de calcul n'est pas évaluée. En effet, bien qu'il soit naturel de supposer une relation entre la quantité de données associées à une maille et son coût calculatoire, deux parties ayant une charge calculatoire similaire peuvent avoir des charges mémoires très différentes (voir les figures 2.9 et 2.10). Dans le cas des solutions obtenues en équilibrant la charge mémoire, l'une des raisons pour lesquelles celles-ci ne sont pas valides est dû au fait que la coupe est une métrique globale. En effet, bien que la coupe induite par la partition soit minimisée, la quantité de données induite par les mailles fantômes n'est pas évaluée localement à chaque partie.

En analysant les solutions obtenues par les problèmes de partitionnement de graphe et d'hypergraphe, nous pouvons constater que le modèle hypergraphe permet de modéliser plus précisément le volume de communications. En effet, on peut constater que l'affectation des mailles induit une frontière en forme d'escalier qui peut s'expliquer par le fait que les mailles ne sont pas communiquées deux fois mais une seule. Ce comportement se distingue en particulier dans le cas de la

répartition unitaire (voir la figure 2.16).

Conclusion

La résolution exacte des problèmes de partitionnement de graphe, d'hypergraphe, et par séparateur sommets, met en évidence que les solutions obtenues en résolvant ces problèmes ne répondent pas toujours aux contraintes mémoire de notre étude. Nous avons donc introduit un nouveau problème de partitionnement. Celui-ci consiste à minimiser le makespan induit par la partition, tout en assurant que les contraintes mémoire de notre étude soient respectées. La minimisation d'une telle métrique n'est pas prise en compte par les problèmes de partitionnement classiques. Au mieux, un résultat équivalent peut être obtenu, mais de façon indirecte, via les contraintes d'équilibrage et non en tant qu'objectif. La notion de maille fantôme n'est pas non plus directement prise en compte par ces approches, mais de façon indirecte, via la minimisation de la coupe ou la taille du séparateur sommets. Ces deux métriques sont liées à la notion de mailles fantômes, mais ce ne sont que des approximations grossières des mailles fantômes.

Le problème de partitionnement de maillage sous contraintes mémoire ayant des liens avec plusieurs problèmes d'ordonnancement, nous nous sommes intéressés à ces problèmes et aux méthodes de résolution qui leur sont appliquées. Notre objectif a alors été de modifier et d'appliquer ces méthodes afin d'obtenir des algorithmes approchés avec garanties de performances. Dans le chapitre suivant, nous expliquerons les liens entre le problème de partitionnement de maillage sous contraintes mémoire et plusieurs problèmes d'ordonnancement. De plus, nous présenterons deux algorithmes que nous avons pu obtenir.

Chapitre 3

Recherche opérationnelle et ordonnancement

La recherche opérationnelle est une discipline relativement récente ayant des ancrages en mathématiques appliquées, en informatique, en économie, et en sciences de l'ingénieur. Elle propose un ensemble de méthodes scientifiques permettant de prendre des décisions optimales ou proches de l'optimum pour des problèmes complexes. Une partie des problèmes traités par la recherche opérationnelle sont dits d'optimisation combinatoire. Un problème d'optimisation combinatoire consiste à trouver la *meilleure* solution, ou une solution qui s'en approche, parmi un grand nombre (fini) de solutions possibles et pour un critère donné. L'ensemble des solutions candidates, respectant généralement une liste de contraintes, est appelé *l'espace des solutions réalisables*. La qualité d'une solution est déterminée par un critère qui associe un réel ou un entier à toute solution. Ce critère est appelé la fonction objectif. Notons qu'il peut exister plusieurs solutions optimales.

Le problème de partitionnement de maillage sous contraintes mémoire est typiquement un problème d'optimisation combinatoire, puisqu'il consiste à trouver la meilleure solution dans un espace de recherche discret. Afin de mieux s'en rendre compte, nous pouvons remarquer que, dans le programme mathématique donné à la page 31 :

— la minimisation de la charge calculatoire est la fonction objectif du problème ;

— les contraintes de voisinage et de mémoire définissent les solutions réalisables du problème.

Partant de ce constat, il est intéressant de remarquer que ce problème est voisin d'autres problèmes d'optimisation combinatoire connus, notamment de problèmes d'ordonnancement classiques.

Dans ce chapitre, nous exposons les liens qui existent entre notre problème de partitionnement et les problèmes d'ordonnancement $R||Cmax$ et $R|q_k|Cmax$. Ces problèmes ayant été largement étudiés, nous nous sommes intéressés aux méthodes employées pour les résoudre et à leur possible application dans le cadre de notre problème. Parmi celles-ci, nous avons étudié l'utilisation de la programmation dynamique [6] et de la programmation linéaire relaxée [1], qui sont deux grandes méthodes d'optimisation utilisées en recherche opérationnelle.

Ce chapitre est structuré comme suit : en section 3.1, nous présentons les problèmes d'ordonnancement qui nous intéressent et les paramètres qui les composent. Nous expliquons la relation existant entre notre problème et certains de ces problèmes d'ordonnancement. En section 3.2, nous énonçons un résultat obtenu en utilisant la programmation dynamique pour résoudre notre problème, dans le cas où les données d'entrée vérifient certains critères. Enfin, en section 3.3 nous présentons un résultat obtenu à l'aide de la programmation linéaire en nombres entiers dans le cas où la fonction objectif n'est plus de type makespan.

3.1 Problèmes d'ordonnancement

Les problèmes d'ordonnancement font partie de la catégorie de problèmes que l'on regroupe sous le nom d'optimisation combinatoire. Typiquement, les données d'un problème d'ordonnancement se composent d'un ensemble de n tâches $J = \{j_1, \ldots, j_n\}$ où à chaque tâche j_i est associée une quantité de travail p_i, et de m machines sur lesquelles les tâches de J doivent être distribuées, tout en respectant des contraintes liées aux tâches et en optimisant un ou plusieurs critères d'optimisation. Ce type de problème émerge dans différents domaines d'application, comme la productique et l'informatique, dès lors qu'il est nécessaire de distribuer un travail, composé de plusieurs tâches, sur plusieurs machines.

3.1.1 Notation pour les problèmes d'ordonnancement

Les problèmes d'ordonnancement étant extrêmement nombreux et variés, une formalisation de ces problèmes a été fourni par Graham [31]. Ce formalisme permet d'exprimer de manière concise et précise tout problème d'ordonnancement. Selon ce formalisme, un problème d'ordonnancement

1. A l'inverse de la programmation linéaire en nombres entiers utilisée au chapitre 2, les variables ne sont pas discrètes mais continues. La relaxation d'un PLNE où chaque variable a pour valeur 0 ou 1 est un programme linéaire où les variables appartiennent à l'intervalle $[0, 1]$.

correspond à une expression à trois champs de la forme $f_1|f_2|f_3$. Le champ f_1 correspond à l'environnement de traitement, le champ f_2 concerne les caractéristiques des tâches et le champ f_3 spécifie la fonction objectif. Par la suite nous décrivons chacun de ces champs de manière non exhaustive, en nous intéressant aux caractéristiques liées à l'ordonnancement sur machines parallèles.

Environnement de traitement

Commençons par décrire le premier champ f_1 spécifiant l'environnement de traitement. Si celui-ci prend ses valeurs dans $\{\emptyset, P, Q, R\}$, alors étant donnée une tâche j, qui consiste en une opération effectuée sur une machine $k \in [\![1; m]\!]$, le temps de traitement de la tâche j sur la machine k est noté $p_{j,k}$. Nous caractérisons les valeurs $\{\emptyset, P, Q, R\}$ de la manière suivante :

- $f_1 = \emptyset$ signifie qu'il y a une unique machine ;
- $f_1 = P$ signifie que l'on dispose de plusieurs machines parallèles identiques ; le temps de traitement d'une tâche est le même sur toutes les machines, c'est-à-dire $p_{j,k} = p_j, \forall k \in [\![1; m]\!]$;
- $f_1 = Q$ signifie que l'on dispose de plusieurs machines parallèles reliées ; le temps de traitement d'une tâche est proportionnel à la vitesse de la machine sur laquelle elle s'exécute, c'est-à-dire $p_{j,k} = a_k p_j, \forall k \in [\![1; m]\!]$ où a_k est le facteur vitesse de la machine k ;
- $f_1 = R$ signifie que l'on dispose de plusieurs machines parallèles non reliées ; le temps de traitement d'une tâche dépend de la machine sur laquelle elle s'exécute (et ceci de manière arbitraire).

Si l'une de ces valeurs est suivie d'un entier positif l, alors le nombre de machines m est constant et est égal à l.

Caractéristiques des tâches

Décrivons maintenant le second champ caractérisant les tâches. Ce champ est composé d'un ensemble de caractéristiques de tâches dans lequel nous pouvons trouver :

- *pmtn* : *préemption* ; le traitement d'une tâche peut être interrompu entre le début et la fin de son exécution ;
- *prec* : *précédence* ; le début de traitement d'une tâche peut nécessiter que le traitement d'une autre tâche soit complété. Ce genre d'informations est usuellement modélisé à l'aide d'un graphe orienté appelé *graphe de précédence* ;

— r_j : *dates de début au plus tôt* ; le traitement de la tâche j ne peut débuter avant cette date.

— d_j : *dates de fin au plus tard* ; le traitement de la tâche j ne peut terminer après cette date ;

— q : *nombre de tâches* ; le nombre de tâches qui peuvent être traitées sur chaque machine.

Spécification de la fonction objectif

Finalement, nous nous intéressons au dernier champ, spécifiant le critère d'optimisation. Pour cela, supposons un ordonnancement et introduisons les notations suivantes :

— C_j, la date de fin du traitement de la tâche j ;

— T_j, le retard de la tâche j, où $T_j = \max(0, C_j - d_j)$;

— U_j, l'indicateur de retard de la tâche j, où $U_j = 0$ si $C_j \leq d_j$ et $U_j = 1$ sinon.

Le dernier champ se compose alors d'un ou plusieurs critères d'optimisation, dont les plus couramment utilisés impliquent la minimisation de :

— $C_{max} = \max_j C_j$: la durée de l'ordonnancement ;

— $T_{max} = \max_j T_j$: le plus grand retard ;

— $\sum_j w_j T_j$: la somme pondérée des retards ;

— $\sum_j U_j$: le nombre de tâches en retard.

Exemples de notations

À titre d'exemple, nous illustrons comment représenter à l'aide de la notation de Graham certains problèmes dont nous parlerons dans la suite de ce chapitre.

Le problème $R||C_{max}$ [13][31] est un problème d'ordonnancement de tâches dans lequel nous avons plusieurs machines non reliées où l'objectif est de minimiser la date à laquelle toutes les tâches ont été traitées. Plus formellement, ce problème se définit comme un problème d'ordonnancement où :

— chaque tâche $j = [\![1; n]\!]$ doit être traitée par une machine k, $1 \leq k \leq m$, avec un temps de traitement $p_{j,k}$ associé, sans interruption, et chaque tâche peut être traitée au temps 0 ;

— chaque machine ne peut traiter qu'une tâche à la fois ;

— l'objectif est de réaliser toutes les tâches le plus vite possible, c'est-à-dire minimiser C_{max}, le *makespan* de l'ordonnancement.

Le problème $R|q_k|C_{max}$ [4] est une des variations du problème précédent, où chaque machine k, $1 \leq k \leq m$, se voit affecter une capacité maximum q_k. Chaque machine k ne peut alors se voir

affecter qu'au plus q_k tâches.

Le problème $R||\sum p_{j,k}$ [57] est la modélisation du problème d'affectation généralisé sous la forme d'un problème d'ordonnancement. Celui-ci consiste en un problème d'ordonnancement où les machines sont non reliées, ne peuvent traiter qu'une tâche à la fois, et où l'on cherche à minimiser la somme totale des temps de traitement.

3.1.2 Relation entre le problème de partitionnement de maillage sous contraintes mémoire et certains problèmes d'ordonnancement

Dans cette section, nous établissons le lien entre certains problèmes d'ordonnancement et le problème de partitionnement de maillage sous contraintes mémoire. Pour cela, nous commençons par remarquer que ce problème peut être vu comme un problème d'ordonnancement utilisant un *graphe de voisinage*. Pour cela, nous modélisons le maillage sous la forme d'un graphe non orienté $G = (V, E)$. Chaque maille i est modélisée par un sommet $v \in V$ pondéré par une quantité de travail p_v et une quantité mémoire ω_v de données. Chaque relation de voisinage entre deux mailles i et i', respectivement modélisées par les sommets v et v', est modélisée par une arête $\{v, v'\} \in E$. Le problème consiste alors à ordonnancer les sommets de V entre les m unités de calcul, où chaque unité de calcul k a une capacité mémoire $\text{Mem}[k]$, $1 \leq k \leq m$. De plus, pour être traité, chaque sommet v nécessite des données de sommets voisins. Nous définissons le voisinage d'un sous-ensemble de $V' \subseteq V$ comme $\mathcal{N}(V') = \cup_{v \in V'} \mathcal{N}(v)$. Ainsi, lorsqu'un ensemble de sommets est ordonnancé sur une unité de calcul k, celle-ci doit allouer un espace mémoire égal à $\sum_{v \in V' \cup \mathcal{N}(V')} \omega_v$, et le temps de traitement associé est égal à $\sum_{v \in V'} p_{v,k}$. Finalement, l'objectif de ce problème est d'affecter chaque sommet de l'ensemble V à une unité de calcul, tout en minimisant le makespan et en respectant des contraintes sur la mémoire : la quantité de mémoire allouée par chaque machine est plus petite ou égale à sa capacité mémoire. En utilisant le symbolisme de Graham nous notons notre problème $R|G, mem|C_{max}$ où G et mem font respectivement référence au graphe de voisinage et à la contrainte mémoire associée. Notons que que le terme *prec* n'apparaît pas dans le second champ, puisqu'il n'y a pas de contraintes de précédence entre les tâches.

Remarque : Par souci d'homogénéité vis-à-vis des autres chapitres, nous présenterons nos résultats en utilisant les sommets du graphe de voisinage, qui représentent les tâches à ordonnancer, et parlerons d'unité de calcul à la place de machine.

Nous pouvons maintenant remarquer que le problème $R|G, mem|C_{max}$ est lié à d'autres problèmes d'ordonnancement plus classiques. En effet, lorsque chaque sommet v a un poids mémoire $\omega_v = 0$, le problème $R|G, mem|C_{max}$ devient le problème d'ordonnancement **NP**-difficile au sens fort $R||C_{max}$ [58]. De même, lorsque le graphe de voisinage n'a pas d'arêtes, la mémoire sur chaque unité de calcul est bornée, et chaque sommet v a un poids mémoire $\omega_v = 1$, le problème $R|G, mem|C_{max}$ devient le problème d'ordonnancement $R|q_k|C_{max}$ qui est une variation du problème $R||C_{max}$ où chaque unité de calcul k ne peut se voir affecter qu'un nombre de sommets borné par q_k.

Cependant, le problème $R|G, mem|C_{max}$ se démarque des autres problèmes d'ordonnancement par la façon singulière dont est évaluée la mémoire devant être réservée par les unités de calcul. Le graphe de voisinage $G = (V, E)$ en est la caractéristique principale, et affecter dynamiquement les sommets qui le composent peut conduire à des allocations de quantités de mémoire différentes selon l'unité de calcul choisie. Nous illustrons cette caractéristique à l'aide d'une instance à deux unités de calcul, dont le graphe de voisinage associé est présenté en figure 3.1. Supposons que le sous-ensemble $V' = \{v_4, v_5, v_6\}$ est affecté à l'unité de calcul 1 (voir figure 3.1(a)) et que le sous-ensemble $V'' = \{v_7\}$ est affecté à l'unité de calcul 2 (voir figure 3.1(b)). Dans ce cas, l'affectation du sommet v_8 sur l'unité de calcul 1 ou sur l'unité de calcul 2 a un impact différent en termes d'allocation mémoire : affecter v_8 sur l'unité de calcul 1 force cette unité de calcul à réserver une quantité mémoire supplémentaire égale à $\omega_{v_8} + \omega_{v_{10}}$ (voir la figure 3.1(a)), tandis que l'affecter à l'unité de calcul 2 force cette unité de calcul à réserver une quantité mémoire supplémentaire égale à $\omega_{v_{10}}$ seulement (voir la figure 3.1(b)).

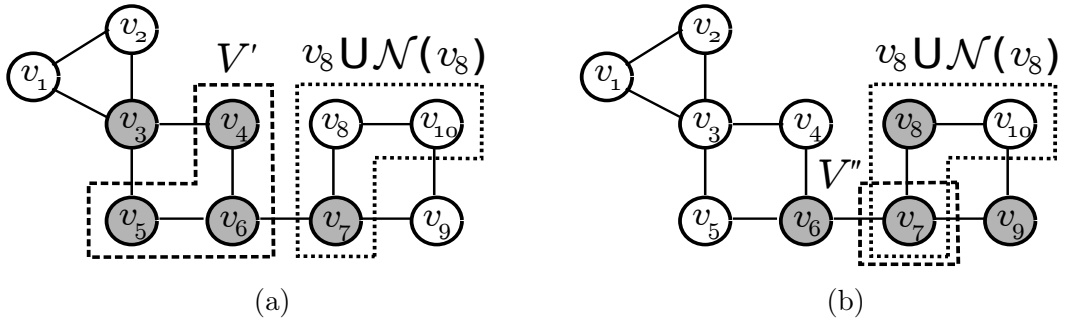


FIGURE 3.1 – Graphe de voisinage $G = (J, E)$, où (a) illustre l'affectation des sommets de $V' = \{v_4, v_5, v_6\} \in V$ et (b) illustre l'affectation des sommets de $V'' = \{v_7\} \in V$. L'affectation de V' (respectivement V'') force l'unité de calcul à allouer une quantité de mémoire pour chaque sommet grisé $v \in V' \cup \mathcal{N}(V') = \{v_3, v_4, v_5, v_6, v_7\}$ (respectivement $v \in V'' \cup \mathcal{N}(V'') = \{v_6, v_7, v_8, v_9\}$). L'affectation de v_8 force l'unité de calcul à réserver une quantité de mémoire additionnelle dépendante des affectations précédemment effectuées : $\omega_8 + \omega_{10}$ en (a), ω_{10} en (b).

Dans la section suivante nous énonçons, de manière non exhaustive, des méthodes issues de

l'état de l'art et utilisées pour résoudre les problèmes d'ordonnancement précédemment introduits. Certaines méthodes ayant permis de résoudre plusieurs de ces problèmes, nous les avons étudiées et avons tenté de les appliquer au problème de partitionnement de maillage sous contraintes mémoire. A ce propos, nous présentons en section 3.2 un résultat obtenu à l'aide d'une de ces méthodes.

3.1.3 Méthodes de résolution et résultats théoriques

Les problèmes d'ordonnancement cités dans la section précédente étant **NP**-difficiles, il est vraisemblablement impossible de trouver des algorithmes polynomiaux capables de les résoudre de manière exacte. Beaucoup d'attention a donc été apportée à la résolution de ces problèmes en relâchant la contrainte d'optimalité. L'objectif n'est alors plus d'obtenir une solution optimale au problème mais une solution « presque » optimale, en temps polynomial. Dans cette section, nous présentons de manière non exhaustive des résultats obtenus pour ces problèmes d'ordonnancement. Pour cela, nous commençons par définir les algorithmes d'approximation avec garantie de performances.

Définition 23. (Algorithme ρ-approché) *Soit x^* la valeur d'une solution optimale d'un problème d'optimisation P. Un algorithme est dit ρ-approché pour un problème de minimisation (respectivement maximisation) P s'il calcule, pour toute instance de P, une solution de valeur x garantissant $x \leq \rho x^*$ avec $\rho > 1$ (respectivement $\rho < 1$). Une telle solution est dite ρ-approchée.*

La famille regroupant ces algorithmes est appelée la famille des algorithmes d'approximation. Elle englobe notamment les schémas d'approximation polynomiaux (en anglais *polynomial-time approximation scheme*, abrégé en $PTAS$) que nous définissons.

Définition 24. (Schéma d'approximation en temps polynomial) *Un schéma d'approximation en temps polynomial est un algorithme prenant en entrée les données d'un problème et une valeur $\varepsilon > 0$ quelconque, renvoyant toujours une solution $(1+\varepsilon)$-approchée, et s'exécutant en temps polynomial en la taille des données. Si le temps d'exécution de l'algorithme est en plus polynomial en $1/\varepsilon$, on parle de* **schéma d'approximation entièrement en temps polynomial**.

Un schéma d'approximation entièrement en temps polynomial (en anglais *fully polynomial-time approximation scheme*, abrégé en $FPTAS$) est typiquement l'un des meilleurs résultats approchés que nous puissions obtenir pour résoudre un problème **NP**-difficile. Parmi les nombreuses techniques existantes, il existe deux méthodes générales bien connues pour obtenir un $FPTAS$:

— Dans la première, les paramètres d'entrée du problème sont arrondis, puis un programme dynamique est appliqué sur l'instance modifiée pour trouver une solution approchée du problème original.
— Dans la seconde, l'idée est de « filtrer » les ensembles d'états d'un programme dynamique, afin que celui-ci soit plus rapide et génère une solution approchée de bonne qualité.

Ces notions étant définies, nous passons maintenant à la présentation des résultats.

Le problème $R||C_{max}$

Parmi les résultats de la littérature pour ce problème, présenté à la page 58, nous retrouvons les algorithmes de *branch and bound* de Van del Velde [80], qui se basent sur la relaxation par substitution et la dualité, et de Martello et al. [61]. Ce problème étant **NP**-difficile au sens fort, une grande attention a été portée à l'une de ses variations, dans laquelle le nombre d'unités de calcul est fixé. Ce problème, noté $Rm||C_{max}$, a notamment été étudié par Lenstra et al. [57] qui ont fourni un algorithme 2-approché et qui ont prouvé qu'il n'existe pas d'algorithme approché en temps polynomial qui puisse obtenir une solution à un facteur plus petit que $\frac{3}{2}$ à moins que $P = NP$. Leur algorithme est basé sur l'application de techniques d'arrondis sur une solution fractionnaire d'un programme linéaire en nombres entiers relaxé. Gairing et al. [28] ont réussi à obtenir un algorithme plus rapide et de même rapport d'approximation en se basant sur des techniques de flots indivisibles. Une certaine attention a été portée sur l'obtention de $FPTAS$. Parmi les deux méthodes permettant l'obtention d'un tel résultat, la première, consistant à utiliser un programme dynamique sur une instance préalablement arrondie, a été utilisée par Sahni [72] pour le problème $Pm||C_{max}$, et par Sahni et Horowitz [41] pour les problèmes $Qm||C_{max}$ et $Rm||C_{max}$. La seconde, consistant à utiliser la méthode *trimming-the-state-space* de Ibarra et Kim [42], a été utilisée et généralisée par Woeginger [84], lui permettant d'obtenir des $FPTAS$ pour plusieurs problèmes d'ordonnancement dont $Pm||C_{max}$, $Qm||C_{max}$ et $Rm||C_{max}$.

Le problème $R|q|C_{max}$

Ce problème, présenté à la page 58, est une des variations de $R||C_{max}$, où chaque machine peut se voir affecter au plus q tâches. La contrainte de capacité mettant en échec plusieurs des méthodes précédentes, des recherches ont été effectuées dans des cas « plus simples » tel que $P2|q|C_{max}$, où le nombre d'unités de calcul est fixé à 2, les unités de calcul sont identiques et une unité de calcul

peut effectuer au plus q tâches. Tsai [79] a développé une heuristique pour ce problème, et a montré qu'elle est asymptotiquement optimale quand les temps de traitement sont indépendants et uniformément distribués. De même, Yang et al. [86] ont proposé un algorithme 1,1626-approché basé sur la programmation semi-définie [82]. Lorsque le nombre d'unités de calcul n'est plus fixé à 2, c'est-à-dire pour le problème $P|q|Cmax$, Babel et al. [3] ont proposé un algorithme 4/3-approché. Dans le cas où il peut y avoir différentes capacités pour les unités de calcul, Zhang et al. [87] proposent un algorithme 3-approché pour le problème $P|q_k|Cmax$. Barna et Aravind [4] présentent un algorithme 2-approché pour $R|q_k|Cmax$ et Kellere et Kotov [49] améliorent ce résultat en donnant un algorithme 3/2-approché pour ce même problème. De même que pour le problème sans contraintes de capacités, Woeginger [85] a conçu un $FPTAS$ pour le problème $Pm|q_k|Cmax$

3.2 Algorithme approché FPT pour $Rm|G, mem|C_{max}$

Le problème $Rm|G, mem|C_{max}$ étant une généralisation des problèmes $Rm||C_{max}$ et $Rm|q|C_{max}$, il est légitime de se demander s'il est possible d'obtenir des algorithmes approchés dépendant potentiellement de certains paramètres du graphe de voisinage. Nous nous proposons de répondre à cette question dans cette section, en fournissant un algorithme qui, soit prouve qu'il n'existe pas de solution valide au problème $Rm|G, mem|C_{max}$, ou bien retourne une solution à ce problème à un facteur au plus $(1 + \varepsilon)$ du makespan optimal, où la capacité des unités de calcul est excédée par un facteur au plus $(1+\varepsilon)$. Par commodité de lecture, nous présentons ce résultat dans le cas à deux unités de calcul identiques ($m = 2$) aux sections 3.2.2 et 3.2.3 et nous expliquerons sa généralisation à tout nombre constant d'unités de calcul m non reliées. Avant cela, nous introduisons en section 3.2.1 les définitions nécessaires.

3.2.1 Définitions

Pour présenter notre algorithme, nous aurons besoin de notions que nous introduisons dans cette section. Nous y définissons notamment les notions de *décomposition linéaire* et de *décomposition arborescente*, ainsi que la *largeur* qui leur est associée. Pour cela, nous utiliserons le graphe $G = (V, E)$ afin de faciliter la compréhension de ces notions vis-à-vis de notre problème. Ces notions sont très utilisées en optimisation combinatoire, et notamment en algorithmique des graphes, puisqu'elles sont à la base de techniques permettant la construction d'algorithmes dont la complexité est poly-

nomiale en la taille de l'entrée mais exponentielle en la taille d'un paramètre fixé. Ces algorithmes sont appelés *Fixed-Parameter Tractable* (FPT), puisque les problèmes qu'ils traitent peuvent être résolus efficacement pour des petites valeurs du paramètre fixé. Plus formellement, un algorithme est dit FPT selon h si sa complexité temporelle est bornée par $f(h).|I|^{\mathcal{O}(1)}$ où $|I|$ est la taille de l'instance et f est une fonction arbitraire dépendant uniquement du paramètre h. L'intérêt de tels algorithmes réside dans le fait qu'en pratique, il peut être raisonnable de supposer que la fonction f ne croît pas trop vite et que le paramètre h est petit. Notons qu'il est possible que ces algorithmes soient paramétrés par un vecteur de paramètres. Parmi ces différents paramètres, nous retrouvons le nombre chromatique [43], les degrés maximum/moyen/minimum, la largeur de coupe [54], la largeur linéaire [69], la largeur arborescente [70], *etc.*

Définition 25. (Décomposition linéaire) *La décomposition linéaire d'un graphe $G = (V, E)$ est un couple (P, X), où $P = (W, F)$ est un graphe chemin, et $X = (X_i)_{i \in W}$ est une famille de sous-ensembles de V satisfaisant :*

1. $\cup_{i \in W} X_i = V$;
2. $\forall (v, v') \in E$, *il existe un* $i \in W$ *tel que* $\{v, v'\} \subseteq X_i$;
3. $\forall i, i', i'' \in W$, *si* i' *se retrouve sur le chemin reliant* i *à* i'' *alors* $X_i \cap X_{i''} \subseteq X_{i'}$.

La *largeur* d'une décomposition linéaire est égale à $\max(|X_i| - 1 : i \in W)$ et la *largeur linéaire* de G, notée $pw(G)$ pour *path-width* en anglais, est la largeur minimale d'une décomposition linéaire de G. Ce paramètre, associé au graphe, est une sorte d'écart mesurant la distance entre le graphe et un graphe chemin : plus la largeur linéaire est petite, plus la structure du graphe est proche d'un graphe chemin.

La construction d'une telle décomposition est illustrée en figure 3.2(a), où les sous-ensembles $(X_i)_{i \in W}$ sont tracés en trait plein, et le résultat est présenté en figure 3.2(b). Pour décomposer le graphe sous la forme d'un arbre plutôt que celle d'un graphe chemin, les définitions précédentes s'étendent directement aux définitions de *décomposition arborescente*, notée (T, X), et de *largeur arborescente*, notée $tw(G)$ pour *tree-width* en anglais. Nous présentons une telle décomposition en figure 3.3(a), où les sous-ensembles $(X_i)_{i \in W}$ sont tracés en trait plein, et le résultat est présenté en figure 3.3(b).

Nous nous intéressons maintenant à une notion équivalente à la largeur linéaire, que l'on appelle

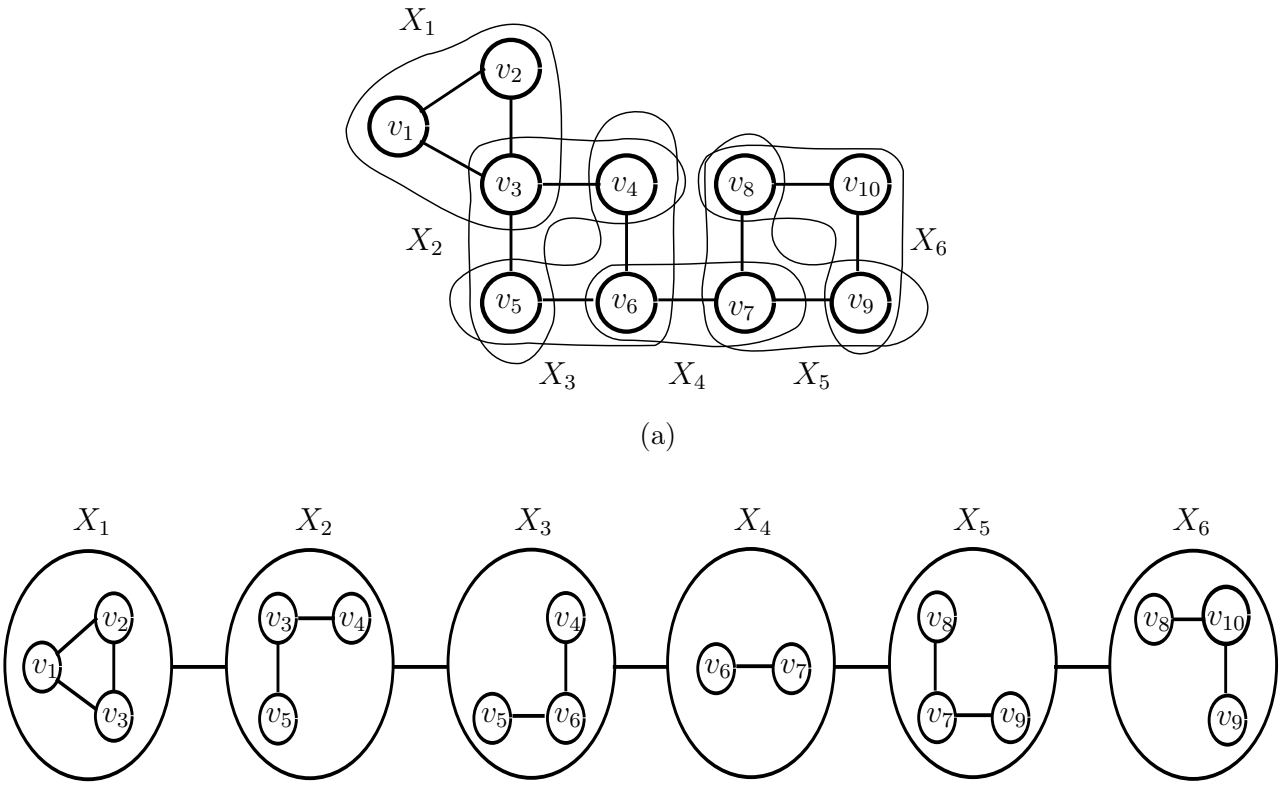


FIGURE 3.2 – Exemple d'une décomposition linéaire (P, X) du graphe de voisinage $G = (V, E)$, où X est composé par les sous-ensembles $X_1 = \{v_1, v_2, v_3\}$, $X_2 = \{v_3, v_4, v_5\}$, $X_3 = \{v_4, v_5, v_6\}$, $X_4 = \{v_6, v_7\}$, $X_5 = \{v_7, v_8, v_9\}$ et $X_6 = \{v_8, v_9, v_{10}\}$ en (a), et (P, X) est présenté en (b). Cette décomposition linéaire est optimale vis-à-vis de la largeur linéaire.

le *nombre de séparation*, noté $vs(G)$ pour *vertex separation number* en anglais. Pour définir cette notion, nous commençons par introduire la notion de *numérotation linéaire* d'un graphe $G = (V, E)$.

Définition 26. (NUMÉROTATION LINÉAIRE) *La numérotation linéaire d'un graphe $G = (V, E)$ est une application bijective $L : V \to \{1, 2, \ldots, |V|\}$.*

Pour toute numérotation L, nous définissons :

$$V_L(i) = \{v \in V | L(v) \leq i \text{ et } \exists v' \in V \text{ tel que } \{v, v'\} \in E \text{ et } L(v') > i\}.$$

Ainsi, $V_L(i)$ est l'ensemble des sommets de $G = (V, E)$ numérotés par un entier inférieur ou égal à i et qui sont connectés à un sommet numéroté par un entier supérieur à i. Nous définissons alors le nombre de séparation de G numéroté par L, $vs_L(G)$, comme le nombre maximum de sommets

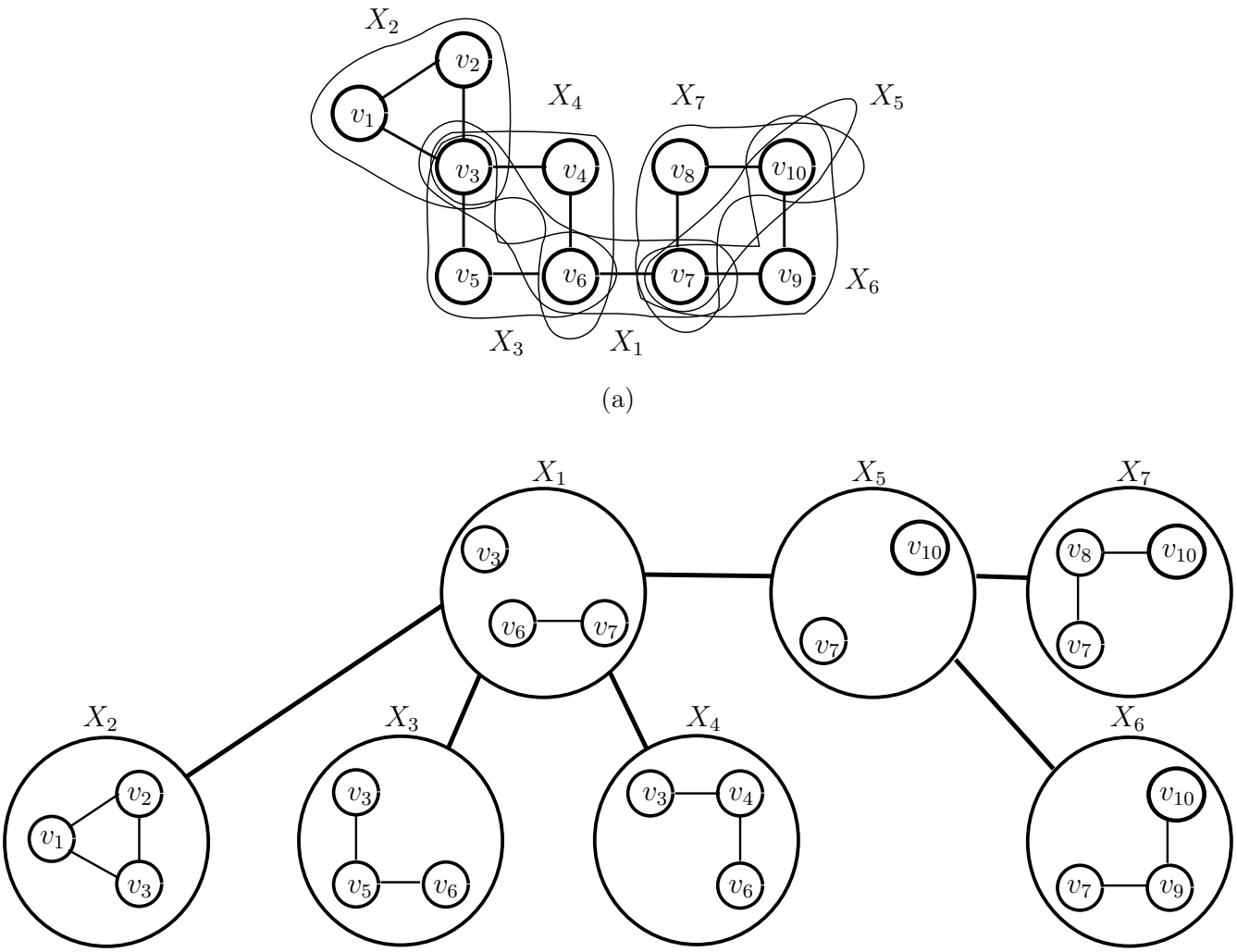


FIGURE 3.3 – Exemple d'une décomposition arborescente (T, X) du graphe de voisinage $G = (V, E)$, où X est composé par les sous-ensembles $X_1 = \{v_3, v_6, v_7\}$, $X_2 = \{v_1, v_2, v_3\}$, $X_3 = \{v_3, v_5, v_6\}$, $X_4 = \{v_3, v_4, v_6\}$, $X_5 = \{v_7, v_{10}\}$, $X_6 = \{v_7, v_9, v_{10}\}$et $X_7 = \{v_7, v_8, v_{10}\}$ en (a), et (P, X) est présenté en (b). Cette décomposition arborescente est optimale vis-à-vis de la largeur linéaire.

pour tout $V_L(i)$, c'est-à-dire

$$vs_L(G) = \max_{1 \leq i \leq |V|} \{|V_L(i)|\}.$$

Définition 27. (NOMBRE DE SÉPARATION) *Le nombre de séparation d'un graphe $G = (V, E)$, $vs(G)$, est le minimum, parmi toutes les numérotations L de G, de $vs_L(G)$. Formellement,*

$$vs(G) = \min\{vs_L(G) | L \textit{ est une numérotation linéaire de } G\}.$$

Par la suite, nous utiliserons la notation L^*, pour faire référence à une numérotation linéaire

optimale selon le nombre de séparation, c'est-à-dire

$$L^* = \arg\min_L vs_L(G).$$

Une telle numérotation des sommets d'un graphe est présentée en figure 3.1. Elle peut être obtenue en temps polynomial dans le cas où le graphe $G = (V, E)$ possède une largeur arborescente bornée par une constante h. Pour obtenir L^*, nous commençons par construire en temps polynomial une décomposition linéaire (P, X) de largeur h. Bodlaender et Kloks [8] ont présenté un algorithme permettant une telle construction en temps polynomial si h est borné. La décomposition linéaire ainsi obtenue est ensuite utilisée pour construire la numérotation linéaire optimale L^*. Kinnersley [51] présente un algorithme permettant une telle construction en utilisant une structure de données adaptée.

3.2.2 Algorithme exact basé sur la programmation dynamique pour le partitionnement de maillage sous contraintes mémoire

Nous rappelons que, par commodité de lecture, nous présentons notre algorithme exact pour le problème $Rm|G, mem|C_{max}$, dans le cas à deux unités de calcul identiques ($m = 2$). Nous supposons que les sommets ont été numérotés de telle sorte que $L^*(v_i) = i$, pour $1 \leq i \leq n$, c'est-à-dire que la numérotation des sommets est optimale vis-à-vis du nombre de séparation.

Le programme dynamique que nous allons présenter se déroule en n phases, où chaque phase i, avec $1 \leq i \leq n$, traite le sommet v_i et produit un ensemble $\mathcal{S}_i$ d'états. Chaque état dans l'ensemble d'états $\mathcal{S}_i$ est un vecteur $S = [\mathcal{P}_1, \mathcal{P}_2, \mathcal{M}_1, \mathcal{M}_2, \mathcal{C}_i] \in \mathcal{S}_i$, qui encode une solution partielle pour les i premiers sommets, c'est-à-dire une affectation des i premiers sommets sur les unités de calcul, où :

1. $\mathcal{P}_1$ (respectivement $\mathcal{P}_2$) est le temps de traitement associé à la première (respectivement seconde) unité de calcul pour l'ordonnancement partiel ;
2. $\mathcal{M}_1$ (respectivement $\mathcal{M}_2$) est la quantité de mémoire allouée par la première (respectivement seconde) unité de calcul pour l'ordonnancement partiel ;
3. $\mathcal{C}_i$ est une structure additionnelle, appelée la *combinaison frontière*. Pour une solution partielle des sommets v_1 à v_i, elle est définie comme $\mathcal{C}_i = (V_L(i), \sigma_i, \sigma'_i)$ avec $\sigma_i : V_L(i) \to \{1, 2\}$ et $\sigma'_i : V_L(i) \to \{0, 1\}$, tels que $\sigma_i(v)$ est l'unité de calcul sur laquelle le sommet v est affecté, et $\sigma'_i(v) = 1$ si et seulement si l'unité de calcul sur laquelle v n'a pas été affectée, c'est-à-dire

l'unité de calcul $3 - \sigma_i(j)$, a déjà mémorisé les données du sommet v.

Nous remarquons que le nombre de combinaisons frontières $\mathcal{C}_i$ distinctes est égal à $4^{|V_L(i)|} \leq 4^{vs(G)}$. L'algorithme 1 présente l'algorithme exact basé sur la programmation dynamique.

Algorithme 1 Algorithme exact basé sur la programmation dynamique.

1: **Fonction** PROGRAMME DYNAMIQUE EXACT($G = (V, E)$)
2: $\mathcal{S}_1 = \{[p_1, 0, \omega_1 + \sum_{v \in \mathcal{N}(v_1)} \omega_v, 0, \mathcal{C}_1^1], [0, p_1, 0, \omega_1 + \sum_{v \in \mathcal{N}(v_1)} \omega_v, \mathcal{C}_1^2]\}$
3: **Pour** $i \leftarrow 2, n$ **Faire**
4: **Pour** $[\mathcal{P}_1, \mathcal{P}_2, \mathcal{M}_1, \mathcal{M}_2, \mathcal{C}_{i-1}] \in \mathcal{S}_{i-1}$ **Faire**
5: Calculer α_i^1 et $\mathcal{C}_i^1$
6: $\mathcal{S}_i \leftarrow \mathcal{S}_i \cup [\mathcal{P}_1 + p_i, \mathcal{P}_2, \mathcal{M}_1 + \alpha_i^1, \mathcal{M}_2, \mathcal{C}_i^1]$
7: Calculer α_i^2 et $\mathcal{C}_i^2$
8: $\mathcal{S}_i \leftarrow \mathcal{S}_i \cup [\mathcal{P}_1, \mathcal{P}_2 + p_i, \mathcal{M}_1, \mathcal{M}_2 + \alpha_i^2, \mathcal{C}_i^2]$
9: **Fin Pour**
10: **Fin Pour**
11: **Retourner** $s \leftarrow [\mathcal{P}_1, \mathcal{P}_2, \mathcal{M}_1, \mathcal{M}_2, \mathcal{C}_n] \in \mathcal{S}_n$ avec $\mathcal{M}_1 \leq M_1$ et $\mathcal{M}_2 \leq M_2$ et tel que $\max\{\mathcal{P}_1, \mathcal{P}_2\}$ est minimal
12: **Fin Fonction**

La ligne 2 est la phase d'initialisation. Elle consiste à créer l'ensemble d'états $\mathcal{S}_1$ qui est composé de deux états, le premier (respectivement second) résultant de l'affectation du sommet 1 sur l'unité de calcul 1 (respectivement 2). $\mathcal{C}_1^1 = (V_L(1), \sigma_1, \sigma_1')$ (respectivement $\mathcal{C}_1^2 = (V_L(1), \sigma_2, \sigma_2')$) est la combinaison frontière obtenue après l'affectation du sommet v_1 sur l'unité de calcul 1 (respectivement 2). Nous avons alors $V_L(1) = \{v_1\}$ dans le cas où le sommet v_1 est voisin à un sommet de numéro supérieur ou égal à 1, et $V_L(1) = \emptyset$ sinon. Dans le cas où $V_L(1) = \{v_1\}$, nous avons $\sigma_1(v_1) = 1$ et $\sigma_1' = 0$. Puis à chaque itération des lignes 5 à 8, pour tout état $S \in \mathcal{S}_{i-1}$ nous ajoutons deux états dans $\mathcal{S}_i$: l'état de la ligne 6 (respectivement 8) correspond au cas où le sommet v_i est affecté à l'unité de calcul 1 (respectivement 2) et α_i^1 (respectivement α_i^2) est la quantité de mémoire utilisée par cette affectation.

Dans le but de montrer comment calculer la valeur α_i^1 utilisée à la ligne 6, nous définissons deux ensembles de sommets, nommés V_1 et V_2, tels que :

$$\begin{aligned} V_1 &= \{v_j \in V_L(i-1) \cap \mathcal{N}(v_i) : \sigma_{i-1}(v_j) \neq 1 \wedge \sigma_{i-1}'(v_j) = 0\}, \\ V_2 &= \{v_j \in \mathcal{N}(v_i) : j > i \wedge \forall v \in V_L(i-1) \cap \mathcal{N}(v_j),\ \sigma_{i-1}(v) \neq 1\}. \end{aligned}$$

V_1 est l'ensemble des sommets, déjà traitées par l'algorithme, qui sont dans le voisinage de v_i, n'ont pas été affectées à l'unité de calcul 1 et n'ont pas été mémorisées par l'unité de calcul 1 non

plus. V_2 est l'ensemble des sommets qui n'ont pas encore été traitées par l'algorithme, sont dans le voisinage de v_i et n'ont pas dans leur voisinage un sommet ayant déjà été affectée à l'unité de calcul 1. Nous avons alors :

$$\alpha_i^1 = \sum_{v \in V_1 \cup V_2} \omega_v + \begin{cases} \omega_{v_i} \text{ si } \forall v \in V_L(i-1) \cap \mathcal{N}(v_i), \sigma_{i-1}(v) \neq 1, \\ 0 \text{ sinon.} \end{cases}$$

La valeur α_i^2, utilisée à la ligne 8, étant calculée d'une manière similaire, nous présentons comment calculer la valeur α_i^k, où $k \in [\![1;2]\!]$, dans l'algorithme 2.

Algorithme 2 Algorithme calculant la quantité de mémoire à allouée par l'affectation de v_i sur k

1: ▷ Fonction pour calculer la valeur α_i^k
2: **Fonction** MÉMOIRE ALLOUÉE$(G = (V, E), i, k)$
3: ▷ Ensembles utilisés pour calculer α_i^k
4: $V_1 \leftarrow \{v_j \in V_L(i-1) \cap \mathcal{N}(v_i) : \sigma_{i-1}(v_j) \neq k \wedge \sigma'_{i-1}(v_j) = 0\}$
5: $V_2 \leftarrow \{v_j \in \mathcal{N}(v_i) : j > i \wedge \forall v \in V_L(i-1) \cap \mathcal{N}(v_j)\ \ \sigma_{i-1}(v) \neq k\}$
6: $\alpha_i^k \leftarrow \sum_{v \in V_1 \cup V_2} \omega_v$
7: **Si** $\forall v \in V_L(i-1) \cap \mathcal{N}(v_i), \sigma_{i-1}(v) \neq k$ **Alors**
8: $\alpha_i^1 \leftarrow \alpha_i^1 + \omega_{v_i}$
9: **Fin Si**
10: **Retourner** α_i^k
11: **Fin Fonction**

Pour illustrer les ensembles V_1 et V_2, considérons l'exemple de la figure 3.4 où nous voulons affecter v_6 à l'unité de calcul 1, alors que $V' = \{v_1, v_2\}$ est déjà affecté à l'unité de calcul 1 et que $V'' = \{v_3, v_4, v_5\}$ est affecté à l'unité de calcul 2. Nous avons $V_L(5) = \{v_4, v_5\}$, $\mathcal{N}(v_6) = \{v_4, v_5, v_7\}$ donc $V_L(5) \cap \mathcal{N}(v_6) = \{v_4, v_5\}$. Comme $V'' = \{v_3, v_4, v_5\}$ est affecté à l'unité de calcul 2, nous avons $\sigma_5(v_4) \neq 1, \sigma_5(v_5) \neq 1$ et $\sigma'_5(v_4) = \sigma'_5(v_5) = 0$. Par conséquent $V_1 = \{v_4, v_5\}$, c'est-à-dire qu'affecter v_6 à l'unité de calcul 1 contraint celle-ci à allouer une quantité mémoire pour v_4 et v_5. De plus, nous avons $v_7 \in \mathcal{N}(v_6)$ tel que $\forall v \in V_L(5) \cap \mathcal{N}(v_7), \sigma_5(v) \neq 1$. Par conséquent, $V_2 = \{v_7\}$, c'est-à-dire qu'affecter v_6 à l'unité de calcul 1 contraint celle-ci à allouer un espace mémoire pour v_7.

Finalement, nous montrons comment obtenir la nouvelle combinaison frontière $\mathcal{C}_i = (V_L(i), \sigma_i, \sigma'_i)$ aux lignes 6 et 8 de l'algorithme 1, dénotée par $\mathcal{C}_i^1$ et $\mathcal{C}_i^2$ respectivement, à partir de $\mathcal{C}_{i-1} = (V_L(i-1), \sigma_{i-1}, \sigma'_{i-1})$. Considérons le premier cas, c'est-à-dire celui où le sommet v_i est affecté à la première unité de calcul, et montrons comment obtenir $\mathcal{C}_i^1$. Si nous avons $v_i \in V_L(i)$, alors $\sigma_i(v_i) = 1$,

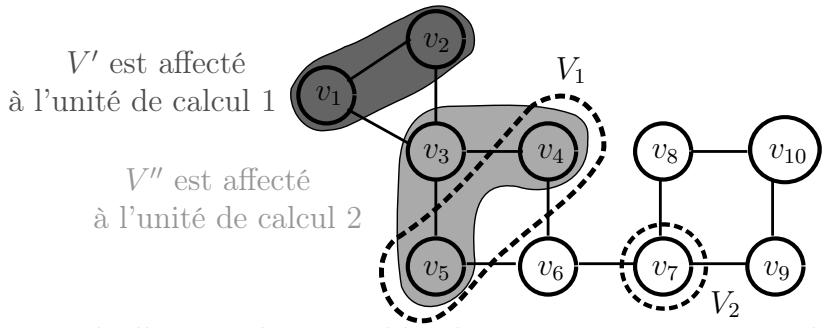


FIGURE 3.4 – Un exemple illustrant les ensembles de sommets V_1 et V_2, quand $i = 6$, $V' = \{v_1, v_2\}$ est affecté à l'unité de calcul 1 et $V'' = \{v_3, v_4, v_5\}$ est affecté à l'unité de calcul 2.

et nous avons $\sigma'_i(v_i) = 1$ si $\exists v \in V_L(i-1) \cap \mathcal{N}(v_i)$ tel que $\sigma_{i-1}(v) = 2$, et $\sigma'_i(v_i) = 0$ sinon. Pour les sommets voisins à v_i, appartenant à $V_L(i)$, c'est-à-dire $v \in \mathcal{N}(v_i) \cap V_L(i)$, nous avons $\sigma_i(v) = \sigma_{i-1}(v)$, et $\sigma'_i(v) = 1$ si $\sigma'_{i-1}(v) \neq 1$, et $\sigma'_i(v) = \sigma'_{i-1}(v)$ sinon. Enfin, pour $v \in V_L(i) \backslash (\{v_i\} \cup \mathcal{N}(v_i))$ nous avons $\sigma_i(v) = \sigma_{i-1}(v)$ et $\sigma'_i(v) = \sigma'_{i-1}(v)$. La combinaison frontière $\mathcal{C}_i^2$ étant calculée d'une manière similaire, nous présentons comment calculer la valeur $\mathcal{C}_i^k$, où $k \in [\![1; 2]\!]$, dans l'algorithme 3.

Algorithme 3 Algorithme calculant la nouvelle combinaison frontière $\mathcal{C}_i^k$

1: **Fonction** NOUVELLE COMBINAISON FRONTIÈRE($G = (V, E), \mathcal{C}_{i-1} = (V_L(i-1), \sigma_{i-1}, \sigma'_{i-1}), k$)
2: Calculer $V_L(i)$
3: **Si** $v_i \in V_L(i)$ **Alors** ▷ Traitement du sommet v_i
4: $\sigma_i(v_i) \leftarrow k$
5: **Si** $\exists v \in V_L(i-1) \cap \mathcal{N}(v_i) : \sigma_{i-1}(v) = 3 - k$ **Alors**
6: $\sigma'_i(v_i) \leftarrow 1$
7: **Sinon**
8: $\sigma'_i(v_i) \leftarrow 0$
9: **Fin Si**
10: **Fin Si**
11: **Pour tout** $v \in V_L(i) \cap \mathcal{N}(v_i)$ **Faire** ▷ Traitement des sommets $v \in \mathcal{N}(v_i)$
12: $\sigma_i(v) \leftarrow \sigma_{i-1}(v)$
13: **Si** $\sigma_{i-1}(v) \neq k$ **Alors**
14: $\sigma'_i(v) \leftarrow 1$
15: **Sinon**
16: $\sigma'_i(v) \leftarrow \sigma'_{i-1}(v)$
17: **Fin Si**
18: **Fin Pour**
19: **Pour tout** $v \in V_L(i) \backslash (\{v_i\} \cup \mathcal{N}(v_i))$ **Faire** ▷ Traitement des sommets $v \notin \mathcal{N}(v_i) \cup \{v_i\}$
20: $\sigma_i(v) \leftarrow \sigma_{i-1}(v)$
21: $\sigma'_i(v) \leftarrow \sigma'_{i-1}(v)$
22: **Fin Pour**
23: **Retourner** $\mathcal{C}_i^k = (V_L(i), \sigma_i, \sigma'_i)$
24: **Fin Fonction**

Pour illustrer la manière dont fonctionne l'algorithme, nous présentons son fonctionnement de manière simplifiée [2] en figure 3.5. Cet exemple illustre les solutions créées par l'algorithme pour les ensembles d'états $\mathcal{S}_1$, $\mathcal{S}_2$ et $\mathcal{S}_3$. Remarquons que, sur le plan, nous avons deux états de $\mathcal{S}_3$ (représentés par les points $\ominus$ et $\blacksquare$) qui se superposent mais qui ne sont pas pour autant considérés comme égaux par l'algorithme. En effet, même si les deux premières composantes de ces vecteurs sont égales, les composantes suivantes ne le sont pas.

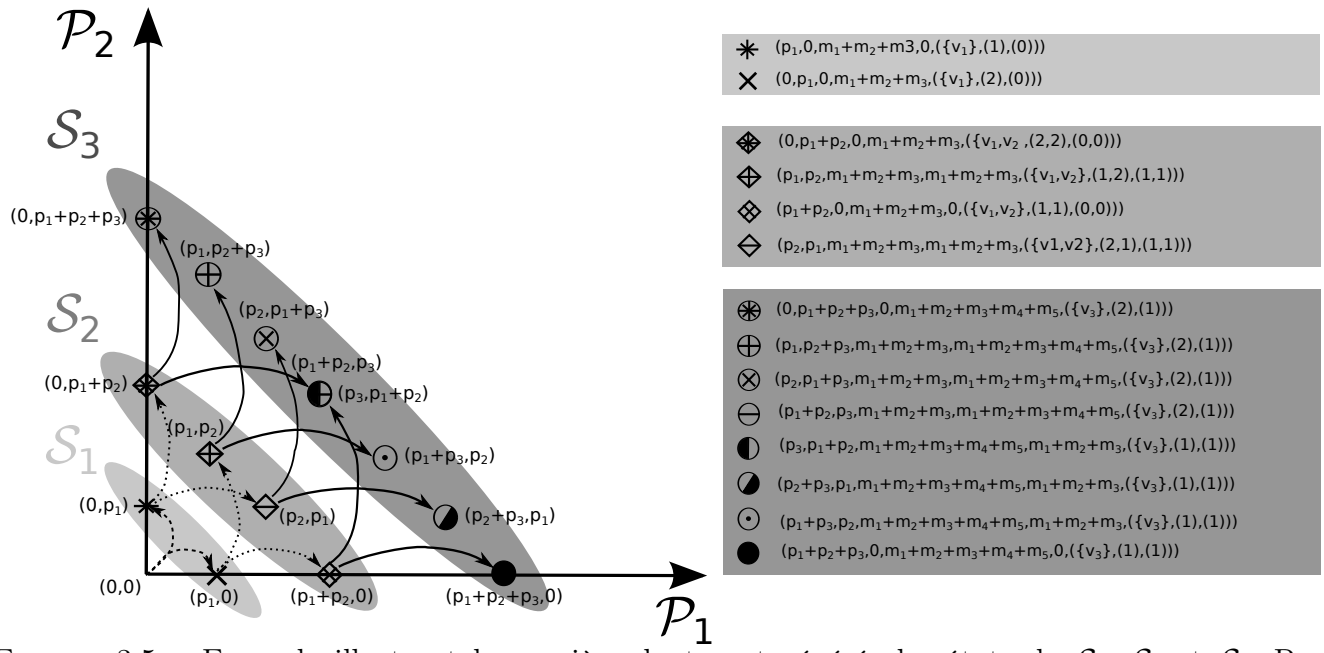


FIGURE 3.5 – Exemple illustrant la manière dont sont générés les états de $\mathcal{S}_1$, $\mathcal{S}_2$ et $\mathcal{S}_3$. Par commodité d'écriture, nous ne présentons que les deux premières dimensions des états.

Nous remarquons que, dans le programme dynamique, lorsque deux états S et S' ont les mêmes composantes, en incluant la même combinaison frontière, alors un seul d'entre eux est conservé dans l'espace d'états. La complexité temporelle pour tester si deux états S et S' sont les mêmes est alors $\mathcal{O}(vs(G))$.

Notons $\mathcal{P}_{sum} = \sum_{i=1}^{n} p_i$ et $\mathcal{M}_{sum} = \sum_{i=1}^{n} \omega_i$. Pour tout vecteur $S = [\mathcal{P}_1, \mathcal{P}_2, \mathcal{M}_1, \mathcal{M}_2, \mathcal{C}_i] \in \mathcal{S}_i$, $\mathcal{P}_1$ et $\mathcal{P}_2$ sont des valeurs entières comprises entre 0 et $\mathcal{P}_{sum}$, $\mathcal{M}_1$ et $\mathcal{M}_2$ sont des valeurs entières comprises entre 0 et $\mathcal{M}_{sum}$, et $|\mathcal{S}_i| = \mathcal{O}(\mathcal{P}_{sum}^2 \times \mathcal{M}_{sum}^2 \times 4^{vs(G)})$. La complexité temporelle de cet algorithme étant proportionnelle à $\sum_{i=1}^{n} |\mathcal{S}_i|$, nous avons donc une complexité globale en $\mathcal{O}(n \times vs(G) \times \mathcal{P}_{sum}^2 \times \mathcal{M}_{sum}^2 \times 4^{vs(G)})$.

2. Nous ne présentons ce résultat que pour les deux premières composantes des vecteurs en raison de la taille de ceux-ci et du fait que les composantes liées à la mémoire feraient se superposer trop de points.

3.2.3 Obtention d'un algorithme approché pour le partitionnement de maillage sous contraintes mémoire

Dans cette section, nous proposons un algorithme approché, dérivé de l'algorithme 1, pour obtenir un algorithme approché FPT. L'idée principale est d'appliquer la méthode *Trimming-of-the-state-space* [42][84] et de supprimer, pendant l'exécution de l'algorithme, des états qui sont proches les uns des autres.

Nous définissons $\Delta = 1+\varepsilon$, avec $\varepsilon > 0$ une constante fixée. Nous commençons par considérer les deux premières coordonnées d'un état $S = [\mathcal{P}_1, \mathcal{P}_2, \mathcal{M}_1, \mathcal{M}_2, \mathcal{C}_i]$. Nous avons $0 \leq \mathcal{P}_1 \leq \mathcal{P}_{sum}$ et $0 \leq \mathcal{P}_2 \leq \mathcal{P}_{sum}$. Nous divisons chacun de ces intervalles en intervalles de la forme $[0]$ et $[\Delta^l, \Delta^{l+1}]$, avec l une valeur entière allant de 0 à $L_1 = \lceil \log_\Delta(\mathcal{P}_{sum}) \rceil = \lceil \ln(\mathcal{P}_{sum}) / \ln(\Delta) \rceil \leq \lceil (1 + \frac{2n}{\varepsilon}) \ln(\mathcal{P}_{sum}) \rceil$. De la même manière, nous divisons les deux prochaines coordonnées en intervalles de la forme $[0]$ et $[\Delta^l, \Delta^{l+1}]$, avec l une valeur entière allant cette fois-ci de 0 à $L_2 = \lceil \log_\Delta(\mathcal{M}_{sum}) \rceil$. L'union de ces intervalles définit un ensemble de boîtes alignées axialement et qui ne se recouvrent pas, dans un espace à quatre dimensions. Si deux états ont la même combinaison frontière et ont leurs quatre premières coordonnées à l'intérieur de la même boîte, alors elles encodent des solutions similaires.

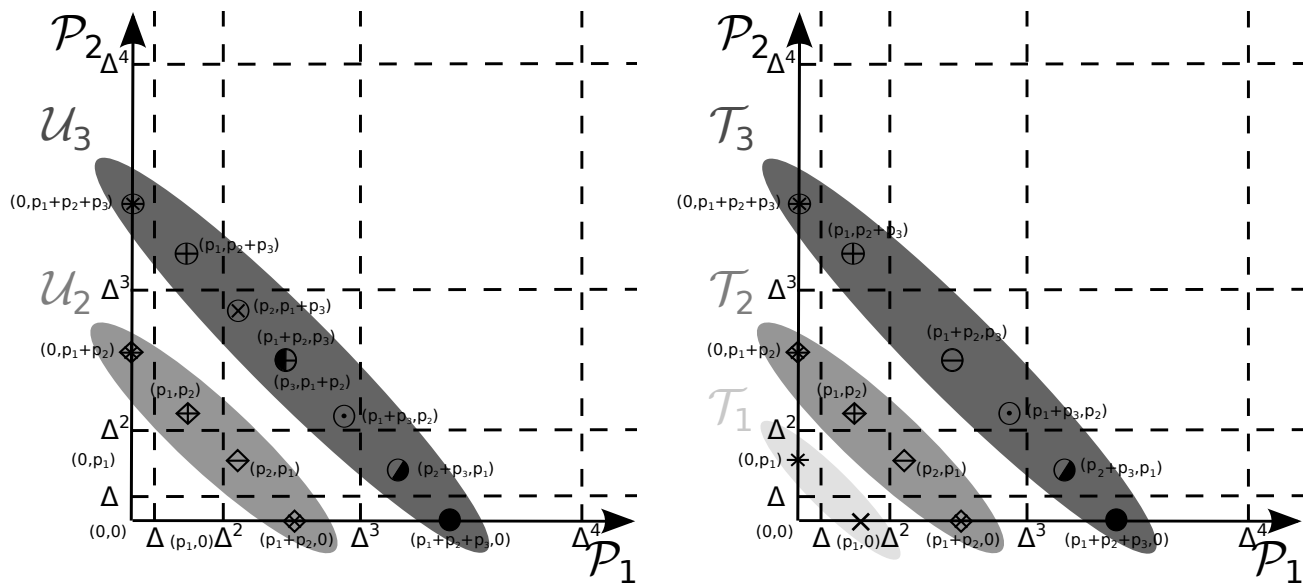
L'algorithme approché fonctionne de la même façon que l'algorithme 1, sauf que nous insérons une phase de filtrage entre les lignes 9 et10 de cet algorithme. La phase de filtrage fonctionne comme suit. S'il y a plusieurs états ayant la même combinaison frontière dans une boîte, alors nous n'en conservons qu'un seul (choisi arbitrairement). Nous notons $\mathcal{U}_i$ l'espace d'états (non filtré) obtenu avant d'effectuer la phase de filtrage à la i-ème phase de l'algorithme, et $\mathcal{T}_i$ l'espace d'états (filtré) obtenu après le filtrage de $\mathcal{U}_i$. L'algorithme 4 décrit pleinement l'algorithme approché basé sur la programmation dynamique.

Nous présentons une version simplifiée de son fonctionnement en figure 3.6 où en 3.6(a) sont présentés les ensemble d'états $\mathcal{U}_i$ et en 3.6(b) les ensembles d'états $\mathcal{T}_i$. Remarquons que l'étape de filtrage de l'algorithme 4 a retiré 2 états lors de la création de $\mathcal{T}_3$ à partir de $\mathcal{U}_3$, et ceci en raison du fait que ces solutions étaient dans la même boîte et encodaient des solutions similaires.

La complexité temporelle de cet algorithme est $\mathcal{O}(n \times vs(G) \times (L_1)^2 \times (L_2)^2 \times 4^{vs(G)})$. Puisque la taille de l'instance est $\Theta(n + |E| + \ln(\mathcal{P}_{sum} + \mathcal{M}_{sum}))$, cet algorithme est par conséquent FPT selon la largeur linéaire. De plus, nous pouvons remarquer que si la largeur arborescente est une constante h, alors la complexité temporelle reste polynomiale. En effet, d'après les travaux effectués

Algorithme 4 Algorithme approché basé sur la programmation dynamique.

1: **Fonction** PROGRAMME DYNAMIQUE APPROCHÉ($G = (J, E)$, k)
2: $\mathcal{S}_1 = \{[p_1, 0, \omega_1 + \sum_{j \in \mathcal{N}(j_1)} \omega_j, 0, \mathcal{C}_1^1], [0, p_1, 0, \omega_1 + \sum_{j \in \mathcal{N}(j_1)} \omega_j, \mathcal{C}_1^2]\}$
3: $\mathcal{T}_1 = \mathcal{S}_1$
4: **Pour** $i \leftarrow 2, n$ **Faire**
5: $\mathcal{U}_i = \emptyset$
6: **Pour** $[\mathcal{P}_1, \mathcal{P}_2, \mathcal{M}_1, \mathcal{M}_2, \mathcal{C}_{i-1}] \in \mathcal{T}_{i-1}$ **Faire**
7: Calculer α_i^1 et $\mathcal{C}_i^1$
8: $\mathcal{U}_i \leftarrow \mathcal{U}_i \cup [\mathcal{P}_1 + p_i, \mathcal{P}_2, \mathcal{M}_1 + \alpha_i^1, \mathcal{M}_2, \mathcal{C}_i^1]$
9: Calculer α_i^2 et $\mathcal{C}_i^2$
10: $\mathcal{U}_i \leftarrow \mathcal{U}_i \cup [\mathcal{P}_1, \mathcal{P}_2 + p_i, \mathcal{M}_1, \mathcal{M}_2 + \alpha_i^2, \mathcal{C}_i^2]$
11: **Fin Pour**
12: Calculer une copie filtrée $\mathcal{T}_i$ de $\mathcal{U}_i$
13: **Fin Pour**
14: **Retourner** $s \leftarrow [\mathcal{P}_1, \mathcal{P}_2, \mathcal{M}_1, \mathcal{M}_2, \mathcal{C}_n] \in \mathcal{T}_n$ avec $\mathcal{M}_1 \leq (1+\varepsilon)M_1$ et $\mathcal{M}_2 \leq (1+\varepsilon)M_2$ et tel que $\max\{\mathcal{P}_1, \mathcal{P}_2\}$ est minimal
15: **Fin Fonction**



(a) (b)

FIGURE 3.6 – Exemple présentant les états de $\mathcal{U}_1$, $\mathcal{U}_2$ et $\mathcal{U}_3$ en (a), et les états de $\mathcal{T}_1$, $\mathcal{T}_2$ et $\mathcal{T}_3$ en (b). Remarquons que les sommets ◐ et $\otimes$ n'apparaissent pas dans $\mathcal{T}_3$ puisqu'ils étaient respectivement dans la même boite que $\odot$ et $\ominus$, et encodaient des solutions similaires à ces deux derniers.

sur les paramètres de graphe, il a été démontré que $pw(G) = \mathcal{O}(\log(n)tw(G))$ pour tout graphe G à n sommets [54], et donc que $vs(G) = \mathcal{O}(\log(n)tw(G))$. Or, la largeur arborescente de G étant une constante h, nous pouvons construire une numérotation linéaire L telle que $vs_L(G) = \mathcal{O}(\log(n)h)$ en temps polynomial en construisant une décomposition arborescente (T, X) de G avec une largeur h [8] et en utilisant les travaux de [77] et [51].

Théorème 1. *Il existe un algorithme FPT selon la largeur linéaire qui, soit prouve qu'il n'existe pas de solution valide au problème $Pm|G, mem|C_{max}$, ou bien retourne une solution à ce problème à un facteur au plus $(1+\varepsilon)$ du makespan optimal, où la capacité mémoire de chaque unité de calcul est excédée par un facteur au plus $(1+\varepsilon)$.*

Comme énoncé précédemment, nous présentons la preuve pour $m = 2$. Nous mentionnons par la suite le cas général où m est une constante fixée quelconque. La preuve de ce théorème se base sur le lemme suivant.

Lemme 1. *Pour tout état $S = [\mathcal{P}_1, \mathcal{P}_2, \mathcal{M}_1, \mathcal{M}_2, \mathcal{C}_i] \in \mathcal{S}_i$, il existe un état $T = [\mathcal{P}_1^\#, \mathcal{P}_2^\#, \mathcal{M}_1^\#, \mathcal{M}_2^\#, \mathcal{C}_i] \in \mathcal{T}_i$ tel que*

$$\mathcal{P}_1^\# \leq \Delta^i \mathcal{P}_1 \quad et \quad \mathcal{P}_2^\# \leq \Delta^i \mathcal{P}_2 \quad et \quad \mathcal{M}_1^\# \leq \Delta^i \mathcal{M}_1 \quad et \quad \mathcal{M}_2^\# \leq \Delta^i \mathcal{M}_2. \tag{3.1}$$

Démonstration. La preuve de ce lemme se fait par récurrence sur i.
Dans le cas où $i = 1$, nous avons par construction $\mathcal{T}_1 = \mathcal{S}_1$. Donc le lemme est vrai pour $i = 1$.
Dans le cas où $i \geq 2$, supposons que les inégalités (3.1) soient valides à l'étape $i - 1$, et considérons un état arbitraire $S = [\mathcal{P}_1, \mathcal{P}_2, \mathcal{M}_1, \mathcal{M}_2, \mathcal{C}_i] \in \mathcal{S}_i$. Cet état S est calculé à partir d'un état $[w, x, y, z, \mathcal{C}_{i-1}] \in \mathcal{S}_{i-1}$.
L'état S vérifie donc l'une des deux égalités suivante : $[\mathcal{P}_1, \mathcal{P}_2, \mathcal{M}_1, \mathcal{M}_2, \mathcal{C}_i] = [w+p_i, x, y+\alpha_i^1, z, \mathcal{C}_i^1]$ ou $[\mathcal{P}_1, \mathcal{P}_2, \mathcal{M}_1, \mathcal{M}_2, \mathcal{C}_i] = [w, x + p_i, y, z + \alpha_i^2, \mathcal{C}_i^2]$. Analysons le premier cas, et notons qu'à l'aide d'arguments similaires, le reste de la preuve est aussi valide pour le second cas. D'après l'hypothèse de récurrence, il existe un vecteur $[w^\#, x^\#, y^\#, z^\#, \mathcal{C}_{i-1}] \in \mathcal{T}_{i-1}$ tel que :

$$w^\# \leq \Delta^{i-1} w \quad \text{et} \quad x^\# \leq \Delta^{i-1} x \quad \text{et} \quad y^\# \leq \Delta^{i-1} y \quad \text{et} \quad z^\# \leq \Delta^{i-1} z. \tag{3.2}$$

L'algorithme filtré génère le vecteur $[w^\# + p_i, x^\#, y^\# + \alpha_i^1, z^\#, \mathcal{C}_i^1] \in \mathcal{U}_i$ et peut le retirer pendant la phase de filtrage, mais il doit laisser un vecteur $[\mathcal{P}_1^\#, \mathcal{P}_2^\#, \mathcal{M}_1^\#, \mathcal{M}_2^\#, \mathcal{C}_i^1] \in \mathcal{T}_i$ qui est dans la même boîte que $[w^\# + p_i, x^\#, y^\# + \alpha_i^1, z^\#, \mathcal{C}_i^1]$. Ce vecteur $[w^\# + p_i, x^\#, y^\# + \alpha_i^1, z^\#, \mathcal{C}_i^1] \in \mathcal{T}_i$ est une approximation de $S = [\mathcal{P}_1, \mathcal{P}_2, \mathcal{M}_1, \mathcal{M}_2, \mathcal{C}_i] \in \mathcal{S}_i$ au sens de l'équation (3.2). En effet, sa première coordonnée $\mathcal{P}_1^\#$ satisfait :

$$\mathcal{P}_1^\# \leq \Delta(w^\# + p_i) \leq \Delta(\Delta^{i-1} w + p_i) \leq \Delta^i w + \Delta p_i \leq \Delta^i (w + p_i) \leq \Delta^i \mathcal{P}_1, \tag{3.3}$$

sa troisième coordonnée $\mathcal{M}_1$ satisfait :

$$\mathcal{M}_1^{\#} \leq \Delta(y^{\#} + \alpha_i^1) \leq \Delta(\Delta^{i-1}y + \alpha_i^1) \leq \Delta^i y + \Delta\alpha_i^1 \leq \Delta^i(y + \alpha_i^1) \leq \Delta^i \mathcal{M}_1, \tag{3.4}$$

et sa dernière coordonnée $\mathcal{C}_i^1$ est égale à $\mathcal{C}_i$.
Par des arguments analogues, nous pouvons montrer que $\mathcal{P}_2^{\#} \leq \Delta^i \mathcal{P}_2$ et $\mathcal{M}_2^{\#} \leq \Delta^i \mathcal{M}_2$. Notre hypothèse de récurrence est donc valide durant la transition de l'étape $i-1$ à l'étape i, ce qui termine notre preuve par récurrence. □

Retournons maintenant à la preuve du théorème 1. A la fin de la n-ième phase, l'algorithme non filtré retourne le vecteur $s = [\mathcal{P}_1, \mathcal{P}_2, \mathcal{M}_1, \mathcal{M}_2, \mathcal{C}_n]$ qui minimise la valeur max $\{\mathcal{P}_1, \mathcal{P}_2\}$, avec $\mathcal{M}_1 \leq M_1$ et $\mathcal{M}_2 \leq M_2$. D'après le lemme 3.2, il existe un vecteur $[\mathcal{P}_1^{\#}, \mathcal{P}_2^{\#}, \mathcal{M}_1^{\#}, \mathcal{M}_2^{\#}, \mathcal{C}_n] \in \mathcal{T}_n$ dont les coordonnées sont au plus à un facteur Δ^n au dessus des coordonnées correspondantes de s. Nous concluons que notre algorithme (algorithme 4) retourne une solution telle que le makespan est au plus Δ^n fois la solution optimale et la quantité de mémoire pour chaque unité de calcul est au plus Δ^n fois sa capacité. De plus, $\Delta^n \leq 1 + \varepsilon$. En effet, si nous considérons les fonctions $f(x) = (1 + x/n)^n$ et $g(x) = 1 + 2x$, avec $(x, n) \in D = [0; 1] \times [1; +\infty[$, nous avons

$$(1 + x/n)^n \leq 1 + 2x, \forall (x, n) \in D, \tag{3.5}$$

puisque f et g sont respectivement une fonction convexe et une fonction linéaire en x et que l'inégalité reste vraie pour $x = 0$ et $x = 1$.

Nous avons donc construit un algorithme qui, s'il existe une solution au problème $Pm|G, mem|C_{max}$, retourne une solution telle que le makespan est au plus $(1 + \varepsilon)$ fois la solution optimale et la quantité de mémoire pour chaque unité de calcul est au plus $(1 + \varepsilon)$ sa capacité. Ceci termine la preuve du théorème 1 dans le cas où le nombre d'unité de calcul m est égal à 2.

Ce résultat peut être étendu à tout nombre constant d'unités de calcul, puisque ajouter des unités de calcul revient simplement à augmenter le nombre de dimensions d'un état. Il suffit alors de redéfinir la combinaison frontière avec :

— $\sigma_i : V_L(i) \to \{1, \ldots, k\}$, c'est-à-dire $\sigma_i(j)$ est l'unité de calcul sur laquelle le sommet v est affecté ;

— $\sigma_i' : V_L(i) \to \mathcal{P}(\{1, \ldots, k\})$, où $\mathcal{P}(E) = \{A | A \subseteq E\}$ est l'ensemble des parties de E, c'est-à-dire $\sigma_i'(j)$ est l'ensemble des unités de calcul qui ne traitent pas le sommet v mais ont déjà

mémorisé les données de ce sommet.

Ceci mène notre algorithme à une complexité temporelle en $\mathcal{O}(n \times m \times vs(G) \times (L_1)^m \times (L_2)^m \times (m \times 2^m)^{vs(G)})$, où n est le nombre de phases, $m \times vs(G)$ est la complexité temporelle pour tester si deux états S et S' sont égaux, $(L_1)^m \times (L_2)^m$ est le nombre de boîtes induites par l'algorithme, et $(m \times 2^m)^{vs(G)}$ est le nombre de combinaisons frontières distinctes. Nous pouvons d'ailleurs remarquer que si le degré maximum du graphe G est borné par une constante d, il nous est possible de diminuer la complexité précédente puisqu'au plus d unités de calcul peuvent mémoriser les données d'un sommet. Ainsi, si $d < m$, la complexité devient alors $\mathcal{O}(n \times m \times vs(G) \times (L_1)^m \times (L_2)^m \times (m \times 2^d)^{vs(G)})$.

Dans cette section, nous avons présenté un algorithme $(1+\varepsilon)$-approché pour notre problème $Rm|G, mem|C_{max}$. Cet algorithme étant FPT selon la largeur linéaire du graphe, il répond à la question sur l'existence d'un algorithme approché dépendant potentiellement de certains paramètres du graphe de voisinage. Ce résultat se limitant à une classe de graphes particuliers, nous avons voulu obtenir un résultat plus générique. Pour cela, nous avons tenté d'appliquer des méthodes basées sur la résolution de programme linéaire relaxé. De plus, nous avons cherché à résoudre un problème voisin de $Rm|G, mem|C_{max}$, noté $R|G_l, mem|\sum p_{v,l}$, où la fonction objectif minimise une somme et non un maximum. Dans la section suivante, nous présentons un résultat approché, en temps polynomial, obtenu pour $R|G_l, mem|\sum p_{v,l}$ en utilisant la résolution de programme linéaire relaxé.

3.3 Algorithme approché pour $R|G_l, mem|\sum p_{v,l}$

Dans cette section, nous nous intéressons au problème $R|G_l, mem|\sum p_{v,k}$, qui est un problème voisin du problème $R|G, mem|C_{max}$ où l'indice l indique que le graphe G est composé de boucles. Ce problème est caractérisé par une fonction objectif minimisant une somme, contrairement au problème $R|G, mem|C_{max}$ dont la fonction objectif minimise un maximum. La raison principale de cet intérêt réside dans l'idée que les problèmes *min-max* semblent être plus difficiles à résoudre que les problèmes *min-sum* [53][62][63]. De plus, obtenir une solution à ce nouveau problème peut s'avérer intéressant, puisque toute solution de celui-ci est aussi solution de $R|G, mem|C_{max}$. La solution pourrait alors être raffinée, en utilisant par exemple des méthodes d'optimisation locale, afin d'améliorer la qualité de la solution.

Pour traiter le problème $R|G_l, mem|\sum p_{v,k}$, nous fournissons un algorithme en temps polynomial qui, étant donné une somme des temps de traitement P, soit prouve qu'il n'existe pas de solution valide de somme des temps de traitement P, soit retourne une solution de temps de traitement au plus P où la capacité de chaque unité de calcul est excédée par un facteur au plus $\left((\Delta+1)\frac{\omega_{max}}{\omega_{min}}+1\right)$, où ω_{max} (respectivement ω_{min}) est le poids mémoire maximal (respectivement minimal). Ainsi, en réalisant une recherche dichotomique sur P, nous pourrons obtenir la somme des temps de traitement minimale.

Nous présenterons le problème $R|G_l, mem|\sum p_{v,k}$ en section 3.3.2 et les résultats obtenus aux sections 3.3.3 et 3.3.4. Cependant avant cela, nous introduisons en section 3.3.1 les définitions nécessaires.

3.3.1 Définitions

Pour présenter l'algorithme, nous aurons besoin de notions que nous introduisons dans cette section. Nous commençons par définir les notions de couplage entier et de couplage fractionnaire. Puis, nous introduisons la notion de combinaison convexe.

Définition 28. (Couplage entier) *Soit $G = (V, E)$ un graphe non orienté. Un couplage entier de $G = (V, E)$ est un sous-ensemble d'arêtes $M \subseteq E$, tel que $\forall (e_1, e_2) \in M \times M, e_1 \cap e_2 = \emptyset$, c'est-à-dire que les arêtes de M n'ont pas d'extrémités en commun. Un sommet $v \in V$ est dit* **saturé** *par un couplage entier M si v appartient à une arête de ce couplage, sinon il est dit insaturé.*

La figure 3.7 représente un graphe $G = (V, E)$ où l'ensemble des arêtes en rouge représente un couplage entier M de G.

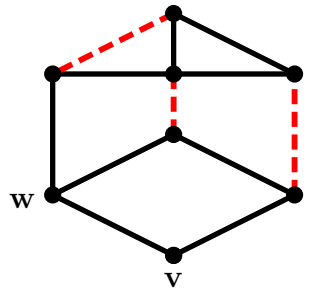
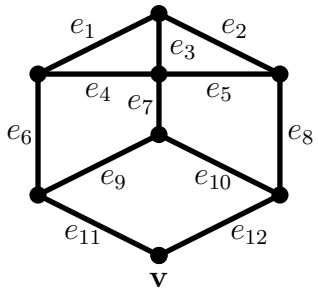


Figure 3.7 – Exemple d'un couplage entier M illustré en tirets rouges, où les sommets v et w ne sont pas saturés.

Nous introduisons maintenant la notion de graphe fractionnaire telle que dans [73].

Définition 29. (Couplage fractionnaire) *Soit $G = (V, E)$ un graphe. On appelle couplage fractionnaire toute fonction f qui associe à chaque arête de E une valeur dans $[0, 1]$ telle que, pour tout sommet $v \in V$, la somme des valeurs associées aux arêtes incidentes à v est au plus égale à 1. Un couplage fractionnaire* ***sature*** *un sommet v si la somme correspondante est égale 1.*

La figure 3.8 représente un graphe $G = (V, E)$ sur lequel le vecteur x sur les arêtes de G est un couplage fractionnaire.



$$f(e_4) = f(e_5) = f(e_{11}) = f(e_{12}) = 0.1$$
$$f(e_1) = f(e_2) = f(e_9) = f(e_{10}) = 0.3$$
$$f(e_3) = f(e_7) = 0.4$$
$$f(e_6) = f(e_8) = 0.6$$

Figure 3.8 – Exemple d'un couplage fractionnaire où tous les sommets, sauf v, sont saturés.

3.3.2 Problème $R|G_l, mem|\sum p_{v,k}$ et présentation de l'algorithme

Dans cette section, nous nous intéressons au problème $R|G_l, mem|\sum p_{v,k}$, qui diffère du problème étudié dans la section précédente sur plusieurs points :

— le premier champ indique que, contrairement au problème $Rm|G, mem|C_{max}$ de la section 3.2, le nombre m d'unités de calcul fait partie de l'entrée ;

— le second champ spécifie que le graphe $G = (V, E)$ est défini comme dans la section précédente, mais à la différence que cette fois-ci chaque sommet $v \in V$ possède une boucle, c'est-à-dire que pour chaque sommet $v \in V$ il y a une arête $(v, v) \in E$;

— le dernier champ indique que la fonction objectif du problème est $\min \sum_{v \in V} \sum_{1 \leq k \leq m} p_{v,k}$.

Le problème $R|G_l, mem|\sum p_{v,k}$ peut s'énoncer sous la forme d'un programme linéaire en nombres entiers (PLNE) en utilisant les variables suivantes :

— La variable $x_{v,k}$ est égale à 1 si et seulement si l'unité de calcul k traite le sommet v, et vaut 0 sinon.

— La variable $y_{v,k}$ est égale à 1 si et seulement si l'unité de calcul k mémorise les données du sommet v, et vaut 0 sinon.

Le PLNE s'écrit alors comme suit :

Pb 5 : Problème d'ordonnancement $R|G_l, mem|\sum p_{v,k}$

$$
\begin{array}{llll}
\min & \sum_{v\in V}\sum_{k=1}^{m} p_{v,k}\ x_{v,k} & & \\
\text{s.c.} & \sum_{k=1}^{m} x_{v,k} = 1 & \forall v \in V & \text{(Pb 5.1)} \\
 & \sum_{v\in V} y_{v,k}\ \omega_v \leq \text{Mem}[k] & \forall k \in [\![1,m]\!] & \text{(Pb 5.2)} \\
 & x_{v,k} \leq y_{v',k} & \forall (v,v') \in E, \forall k \in [\![1,m]\!] & \text{(Pb 5.3)} \\
 & x_{v,k} \in \{0,1\} & \forall v \in V, \forall k \in [\![1,m]\!] & \text{(Pb 5.4)} \\
 & y_{v,k} \in \{0,1\} & v \in V, \forall k \in [\![1,m]\!] & \text{(Pb 5.5)}
\end{array}
$$

La formalisation du problème $R|G_l, mem|\sum p_{v,k}$ est similaire à celle du problème de partitionnement de maillage sous contraintes mémoire. Pour une description des contraintes (Pb 5.1) à (Pb 5.5), nous renvoyons le lecteur aux contraintes (Pb 1.1) à (Pb 1.6) de la section 2.3.1.

Pour trouver une solution au problème $R|G_l, mem|\sum p_{v,k}$, nous nous inspirons des travaux de [75], et nous intéressons à une variante de $R|G_l, mem|\sum p_{v,k}$. Dans celle-ci, au lieu de minimiser la somme totale des temps de traitement, nous cherchons à la borner par une constante P. Nous pourrons ensuite réaliser une recherche dichotomique sur P afin d'obtenir la somme totale des temps de traitement minimale.

Notons Mem le vecteur des capacités mémoires ayant pour composantes $\text{Mem}[k]_{k\in[\![1,m]\!]}$. La variante du problème peut s'énoncer sous la forme du programme linéaire relaxé $PL(P, \text{Mem})$ suivant :

Pb 6 : Problème d'ordonnancement de temps de traitement total borné par P

$$
\begin{array}{llll}
\text{s.c.} & \sum_{v\in V}\sum_{k=1}^{m} p_{v,k}\ x_{v,k} \leq P & & \text{(Pb 6.1)} \\
 & \sum_{k=1}^{m} x_{v,k} = 1 & \forall v \in V & \text{(Pb 6.2)} \\
 & \sum_{v\in V} y_{v,k}\ \omega_v \leq \text{Mem}[k] & \forall k \in [\![1,m]\!] & \text{(Pb 6.3)} \\
 & x_{v,k} \leq y_{v',k} & \forall (v,v') \in E, \forall k \in [\![1,m]\!] & \text{(Pb 6.4)} \\
 & x_{v,k} \in [0,1] & \forall v \in V, \forall k \in [\![1,m]\!] & \text{(Pb 6.5)} \\
 & y_{v,k} \in [0,1] & \forall v \in V, \forall k \in [\![1,m]\!] & \text{(Pb 6.6)} \\
 & x_{v,k} = 0 & \forall v \in V, \forall k \in [\![1,m]\!] \ \text{ tel que } \ \sum_{v'\in\mathcal{N}(v)} \omega_{v'} > \text{Mem}[k] & \text{(Pb 6.7)}
\end{array}
$$

À l'inverse du PLNE modélisant le problème $R|G_l, mem|\sum p_{v,k}$, le programme linéaire $PL(P, \text{Mem})$ n'a pas de fonction objectif mais une borne sur la somme des temps de traitement qui est modélisée grâce à la contrainte (Pb 6.1). Les contraintes (Pb 6.2) à (Pb 6.6) sont les mêmes que pour le

PLNE présenté précédemment, sauf que cette fois-ci les variables $x_{v,k}$ et $y_{v,k}$ prennent leur valeur dans l'intervalle $[0,1]$. La contrainte (Pb 6.7) impose que si la quantité de données nécessaire au sommet v est supérieure à la capacité mémoire de la machine k, alors la variable $x_{v,k}$ est nulle. Cette contrainte nous sera utile pour la preuve du lemme 2 (ci-après) puisqu'elle nous permet de garantir que pour toute solution de $PL(P, \text{Mem})$, si $x_{v,k} \neq 0$, alors l'occupation mémoire nécessaire pour traiter le sommet v est inférieure à $\text{Mem}[k]$.

Notons $P_{opt}(\text{Mem})$ la plus petite somme des temps de traitement possible en fonction de Mem.

Théorème 2. *S'il existe une solution au problème $R|G_l, mem|\sum p_{v,k}$ alors il est possible d'obtenir une solution à ce problème, en temps polynomial, de temps de traitement optimal $P_{opt}(\text{Mem})$, où la capacité de chaque unité de calcul est excédée par un facteur au plus $\left((\Delta+1)\frac{\omega_{max}}{\omega_{min}}+1\right)$.*

Pour prouver ce théorème, nous allons utiliser le lemme suivant :

Lemme 2. *Si une solution du programme linéaire $PL(P, \text{Mem})$ existe, alors on peut trouver en temps polynomial une solution pour le problème $R|G_l, mem|\sum p_{v,k}$ dont la somme totale des temps de traitement est bornée par P, et où chaque unité de calcul k a une occupation mémoire au plus de*

$$\left((\Delta+1)\frac{\omega_{max}}{\omega_{min}}+1\right)\text{Mem}[k].$$

Nous prouvons ce lemme en fournissant un algorithme, se basant sur [75], qui convertit une solution fractionnaire de $PL(P, \text{Mem})$ en une solution entière dont on sait borner le dépassement mémoire. Pour cela l'algorithme se base sur la notion de couplage (entier) dans un graphe biparti mettant en relation les sommets à ordonnancer et les unités de calcul.

L'idée générale de l'algorithme de conversion est la suivante. À partir de x^{opt}, une solution fractionnaire optimale de $PL(P, \text{Mem})$, nous construisons un graphe biparti $B = (V, \mathcal{M}, F)$ où :

1. V est l'ensemble des sommet à affecter ;
2. $\mathcal{M}$ est l'ensemble des unités de calcul, potentiellement dupliquées, où $m_k^i \in \mathcal{M}$ désigne la i-ème duplication de l'unité de calcul k ;
3. F est l'ensemble des couples sommet / unité de calcul, (v, k) tels que $x_{v,k}^{opt}$ est strictement positif.

Nous construisons aussi une fonction f définie sur les arêtes de B tel que :

$$\forall v \in V \text{ et } \forall k = 1, \ldots, m,\ x_{v,k}^{opt} = \sum_{\{v, m_k^i\} \in F} f(\{v, m_k^i\}). \tag{3.6}$$

Pour illustrer la manière dont est construit le graphe $B = (V, \mathcal{M}, F)$ et la fonction f définie sur les arêtes de B, nous introduisons une instance simple sur laquelle nous utiliserons notre algorithme. Considérons l'instance où le nombre d'unité de calcul m est égal à 3, le nombre de sommet n est égal à 7 et dont le graphe de voisinage $G_l = (V, E)$ est donné à la figure 3.9. De plus, nous supposons que :

— la quantité mémoire de chaque sommet est unitaire, c'est-à-dire que $\text{Mem}_v = 1$, $\forall v \in V$;

— la capacité mémoire de chaque unité de calcul est égale à 5, c'est-à-dire $\text{Mem}[k] = 5$, $\forall k \in [\![1, 3]\!]$;

— la première unité de calcul traite un sommet deux fois plus vite que les deux autres, c'est-à-dire

$$p_{v,1} = \frac{p_{v,2}}{2} = \frac{p_{v,3}}{2}.$$

Nous supposons que nous désirons obtenir une distribution de coût au plus égal à 5.5.

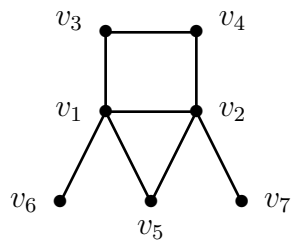


FIGURE 3.9 – Graphe de voisinage $G_l = (V, E)$ sur lequel nous appliquons l'algorithme de conversion.

Dans la section suivante, nous introduisons la manière dont sont construit le graphe $B = (V, \mathcal{M}, F)$ et la fonction f définie sur les arêtes de F.

3.3.3 Construction du graphe biparti et de la fonction d'affectation fractionnaire

Partant d'une solution fractionnaire optimale x^{opt} de $PL(P, \text{Mem})$, nous construisons le graphe biparti $B = (V, \mathcal{M}, F)$ tel que la fonction f satisfasse l'équation (3.6). L'ensemble V représente l'ensemble des sommets de $G_l = (V, E)$ triés par *poids de voisinage* décroissants. Nous notons

$$\mathcal{W}_v = \sum_{v' \in \mathcal{N}(v)} \omega_{v'}$$

le poids du voisinage du sommet v. Notons que disposer de ce poids de voisinage ordonnés de façon décroissante est utile pour la preuve mais n'est pas nécessaire à la construction du vecteur solution. L'ensemble $\mathcal{M} = \{m_k^i : k = 1, \cdots, m,\ i = 1, \cdots, s_k\}$, où $s_k = \lceil \sum_v x_{v,k}^{opt} \rceil$ est le nombre de duplications de l'unité de calcul k, représente l'ensemble des unités de calcul potentiellement dupliquées. Nous notons $m_k^i \in \mathcal{M}$ la i-ème duplication de l'unité de calcul k. L'ensemble F correspond aux arêtes du graphe $B = (V, \mathcal{M}, F)$, où chaque arête représente une affectation sommet-unité de calcul $\{v, k\}$ telle que $x_{v,k}^{opt} \neq 0$. Pour chaque valeur $x_{v,k}^{opt}$ non nulle, il y aura une ou deux arêtes correspondantes dans $B = (V, \mathcal{M}, F)$. Le coût de chaque arête $\{v, m_k^i\} \in F$ est $p_{v,k}$.

Nous décrivons maintenant une manière de construire le graphe $B = (V, \mathcal{M}, F)$ et la fonction f. Pour cela nous traitons chaque unité de calcul k indépendamment et présentons comment traiter une unité de calcul k donnée.

Première duplication

Pour une unité de calcul k donnée, si $\sum_{v \in V} x_{v,k}^{opt} \leq 1$, alors il n'y a qu'une seule instanciation de l'unité de calcul k, notée m_k^1, dans $\mathcal{M}$. Dans ce cas, pour tout $v \in V$ tel que $x_{v,k}^{opt} > 0$, nous ajoutons l'arête $\{v, m_k^1\}$ à F et nous lui associons la valeur $f(\{v, m_k^1\}) = x_{v,k}^{opt}$.
Sinon, nous cherchons l'indice minimal r_1 tel que $\sum_{j=1}^{r_1} x_{v_j,k}^{opt} > 1$. Nous ajoutons alors les arêtes $\{v_j, m_k^1\}$ à F, $\forall j = 1, \cdots, r_1 - 1$ tel que $x_{v_j,k}^{opt} > 0$, auxquelles nous associons la valeur $f(\{v_j, m_k^1\}) = x_{v_j,k}^{opt}$. Ensuite, nous ajoutons l'arête $\{v_{r_1}, m_k^1\}$ à F et lui associons la valeur $f(\{v_{r_1}, m_k^1\}) = 1 - \sum_{j=1}^{r_1-1} f(\{v_j, m_k^1\})$. De cette façon, nous saturons le sommet m_k^1. Puisque $\sum_{j=1}^{r_1} x_{v_j,k}^{opt} > 1$, il reste donc une fraction de la valeur $x_{r_1,k}^{opt}$ qui n'est pas affectée. Nous ajoutons alors l'arête $\{r_1, m_k^2\}$ à F et nous lui associons la valeur $f(\{v_{r_1}, m_k^2\}) = x_{v_{r_1},k}^{opt} - f(\{v_{r_1}, m_k^1\})$.

En résumé, la construction respecte les propriétés suivantes :

— si $\exists r_1$ tel que $\sum_{j=1}^{r_1} x_{v_j,k}^{opt} > 1$ alors :

$$\sum_{j=1}^{r_1} x_{v_j,k}^{opt} = \sum_{j=1}^{r_1} f(\{v_j, m_k^1\})) + f(\{v_{r_1}, m_k^2\}),$$

$$f(\{v_j, m_k^1\}) = x_{v_j,k}^{opt}, \forall j < r_1,$$

et

$$f(\{v_{r_1}, m_k^2\}) = x_{v_{r_1},k}^{opt} - f(\{v_{r_1}, m_k^1\});$$

— sinon,

$$f(\{v, m_k^1\}) = x_{v,k}^{opt}, \forall v \in V, \text{ tel que } x_{v,k}^{opt} > 0.$$

Duplications suivantes

Pour les duplications suivantes, c'est-à-dire $\forall i = 2, \cdots, s_k - 1$, nous cherchons l'indice minimal r_i tel que $\sum_{j=1}^{r_i} x_{v_j,k}^{opt} \geq i$. Nous ajoutons alors les arêtes $\{v_j, m_k^i\}$ à F, pour tous les $j = r_{i-1} + 1, \cdots, r_i - 1$, tels que $x_{v_j,k}^{opt} > 0$, auxquelles nous associons la valeur $f(\{v_j, m_k^i\}) = x_{v_j,k}^{opt}$. Ensuite, nous ajoutons l'arête $\{v_{r_i}, m_k^i\}$ à F et lui associons la valeur $f(\{v_{r_i}, m_k^i\}) = 1 - \sum_{j=r_{i-1}+1}^{r_i-1} f(\{v_j, m_k^i\})$. Si $\sum_{j=1}^{r_i} x_{v_j,k}^{opt} > i$, alors nous ajoutons l'arête $\{v_{r_i}, m_k^{i+1}\}$ à F et lui associons la valeur $f(\{v_{r_i}, m_k^{i+1}\}) = x_{r_i,k}^{opt} - f(\{v_{r_i}, m_k^i\})$. La dernier sommet affecté de cette façon est le sommet $v_{r_{s_k-1}}$. Alors, pour tout sommet v_j tel que $j > r_{s_k-1}$ et $x_{v_j,k}^{opt} > 0$, nous ajoutons l'arête $\{v_j, m_k^{s_k}\}$ et lui associons la valeur $f(\{v_j, m_k^{s_k}\}) = x_{v_j,k}^{opt}$.

En résumé, pour tout $i = 2, \cdots, s_k - 1$, la construction respecte les propriétés suivantes :

$$\sum_{j=1}^{r_i} x_{v_j,k}^{opt} \geq i;$$

$$f(\{v_j, m_k^i\}) = x_{v_j,k}^{opt}, \forall j = r_{i-1} + 1, \cdots, r_i - 1, \text{ tels que } x_{v_j,k}^{opt} > 0;$$

$$f(\{v_{r_i}, m_k^i\}) = 1 - \sum_{j=r_{i-1}+1}^{r_i-1} f(\{v_j, m_k^i\});$$

si $\sum_{j=1}^{r_i} x_{v_j,k}^{opt} > i$, alors

$$\{v_{r_i}, m_k^{i+1}\} \in F \text{ et } f(\{v_{r_i}, m_k^{i+1}\}) = x_{r_i,k}^{opt} - f(\{v_{r_i}, m_k^i\});$$

$$f(\{v_j, m_k^{s_k}\}) = x_{v_j,k}^{opt}, \forall v_j \text{ tel que } j > r_{s_k-1} \text{ et } x_{v_j,k}^{opt} > 0.$$

Cette construction connecte chaque sommet $m_k^i \in \mathcal{M}$ à un ou plusieurs sommets appartenant à V, de façon que, pour tout k, la somme des valeurs associées aux arêtes incidentes au sommet m_k^i pour $i = 1, \cdots, s_k - 1$, soit égale à 1. L'algorithme 5 ci-après fournit un exemple de pseudo-code permettant de construire le graphe $B = (V, \mathcal{M}, F)$ et la fonction f.

Nous illustrons la construction du graphe biparti $B = (J, \mathcal{M}, F)$ et de la fonction d'affectation f sur l'instance présentée en figure 3.9. Pour cela, nous commençons par résoudre le programme linéaire $PL(P, \text{Mem})$ associé à l'instance étudiée. Nous obtenons une solution fractionnelle optimale

Algorithme 5 Algorithme de construction du graphe biparti $B = (V, \mathcal{M}, F)$ et de la fonction f.

1: **Fonction** CONSTRUCTION(x^{opt}, $G_l(V, E)$)
2: Trier les sommets $v \in V$ par ordre de poids de voisinage décroissant
3: **Pour** $k \leftarrow 1, m$ **Faire**
4: $somme \leftarrow 0$
5: $i \leftarrow 1$
6: **Pour** $v \in V$ où $x^{opt}_{v,k} \geq 0$ **Faire**
7: $somme \leftarrow somme + x^{opt}_{v,k}$
8: **Si** $somme \leq 1$ **Alors**
9: Ajouter l'arête (v, m^i_k) à $B = (V, \mathcal{M}, F)$
10: $f(v, m^i_k) \leftarrow x^{opt}_{v,k}$
11: **Si** $somme == 1$ **Alors**
12: $i \leftarrow i + 1$
13: $somme \leftarrow 0$
14: **Fin Si**
15: **Sinon**
16: Ajouter l'arête (v, m^i_k) à $B = (V, \mathcal{M}, F)$
17: $f(v, m^i_k) \leftarrow x^{opt}_{v,k} - (somme - 1)$
18: $somme \leftarrow somme - 1$
19: $i \leftarrow i + 1$
20: Ajouter l'arête (v, m^i_k) à $B = (V, \mathcal{M}, F)$
21: $f(v, m^i_k) \leftarrow somme$
22: **Fin Si**
23: **Fin Pour**
24: **Fin Pour**
25: **Fin Fonction**

x^{opt} dont les valeurs sont présentées dans la table 3.1. Nous construisons alors le graphe biparti $B = (V, \mathcal{M}, F)$ et la fonction f. Cette construction est illustrée dans les figures 3.10(a), 3.10(b) et 3.10(c).

Nous allons maintenant utiliser le graphe $B = (V, \mathcal{M}, F)$ et la fonction f afin d'obtenir un couplage entier définissant l'affectation des sommets aux unités de calcul.

	Sommet						
Unité de calcul	$\frac{1}{3}$	$\frac{1}{3}$	0	0	$\frac{1}{3}$	1	1
	$\frac{2}{3}$	0	0	1	$\frac{2}{3}$	0	0
	0	$\frac{2}{3}$	1	0	0	0	0

TABLE 3.1 – Tableau présentant les valeurs d'une solution fractionnelle optimale x^{opt}. La valeur de $x^{opt}_{v_j,k}$ se lit dans la cellule située sur la $k^{ème}$ ligne et $j^{ème}$ colonne.

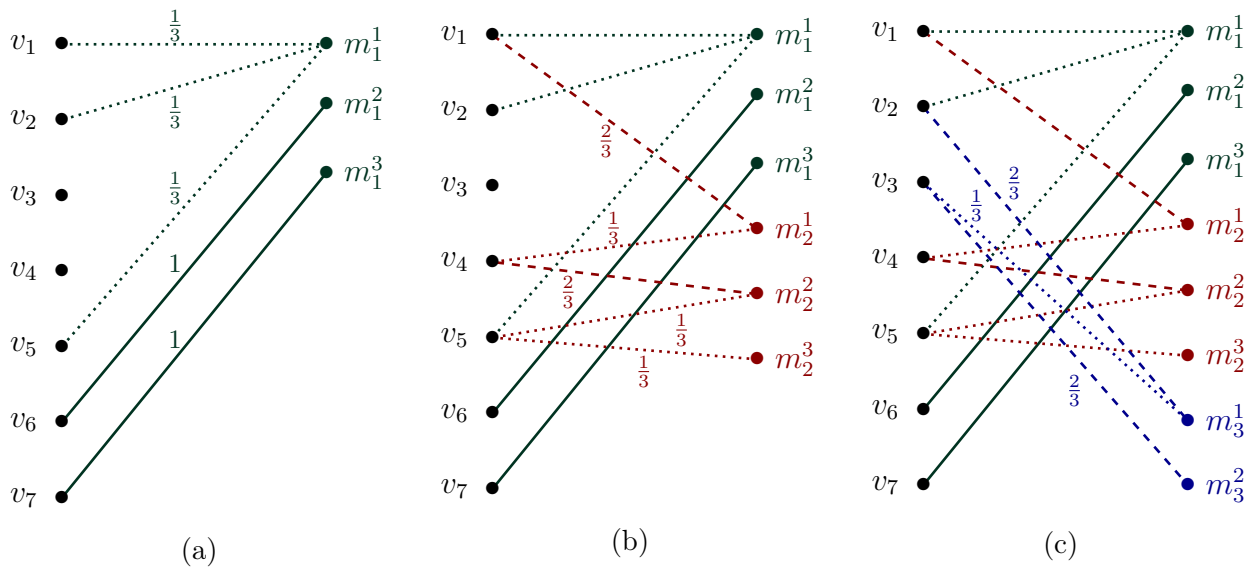


FIGURE 3.10 – Exemple présentant la construction du graphe biparti $B = (V, \mathcal{M}, F)$ et les poids associés aux arêtes de F par la fonction f. Les figures (a), (b) et (c) illustrent les constructions induites par le traitement successif des unités de calcul 1, 2, et 3. Pour chaque figure, les arêtes pleines vérifient $f(\{v, m_k^i\}) = 1$, les arêtes en tirets vérifient $f(\{v, m_k^i\}) = 2/3$ et les arêtes en pointillés vérifient $f(\{v, m_k^i\}) = 1/3$.

3.3.4 Obtention d'un couplage entier

Pour commencer, nous prouvons le lemme suivant.

Lemme 3. *La fonction f est un couplage fractionnaire de $B = (V, \mathcal{M}, F)$ de coût au plus P qui sature exactement chaque sommet $v_j \in V$ et chaque duplication machine m_k^i, pour tout $k = [\![1, m]\!]$ et $i = [\![1, s_k - 1]\!]$.*

Démonstration. Une des caractéristiques de la fonction f est que, de par sa construction, elle respecte la propriété (3.6). Or, puisque x^{opt} est une solution fractionnaire de $PL(P, \text{Mem})$, nous avons :

$$\sum_{k=1}^{|M|} x_{v,k}^{opt} = 1, \ \forall v \in V.$$

Par conséquent, pour tout $v \in V$, la somme des valeurs associées aux arêtes incidentes à v, par la fonction f, est égale à 1. En d'autres termes, f sature exactement chaque sommet $j \in V$. D'autre part, l'algorithme 5 employé pour la construction de $B = (V, \mathcal{M}, F)$ assure que pour chaque duplication m_k^i, avec $k \in [\![1, m]\!]$ et $i \in [\![1, s_k - 1]\!]$, la somme des valeurs associées aux arêtes indicentes à m_k^i par la fonction f vaut 1. Donc, par construction, la fonction f sature exactement chaque duplication m_k^i, $\forall k \in [\![1, m]\!]$, $\forall i \in [\![1, s_k - 1]\!]$. Si, pour une unité de calcul k, ce n'est pas le cas, alors nous avons $\sum_{v \in V} f(\{v, m_k^i\}) \leq 1$. La fonction f est donc un couplage fractionnaire qui

sature exactement chaque sommet $v \in V$ et chaque duplication m_k^i, $\forall k \in [\![1, m]\!]$ et $\forall i \in [\![1, s_k - 1]\!]$. De plus, le coût induit par la fonction f est au plus P puisque f et $B = (V, \mathcal{M}, F)$ sont construits à partir d'une solution x^{opt} de $PL(P, \text{Mem})$ et que f respecte (3.6). □

D'après le lemme 3, f est un couplage fractionnaire de $B = (V, \mathcal{M}, F)$ de somme des temps de traitement au plus P et qui sature exactement chaque sommet $v \in V$. Cela implique qu'il existe un couplage entier sur $B = (V, \mathcal{M}, F)$ de somme des temps de traitement au plus P, qui sature exactement chaque sommet (voir les pages 266 à 270 de [60]). Pour obtenir un tel couplage entier, il suffit de chercher un couplage entier de $B = (V, \mathcal{M}, F)$ saturant exactement chaque sommet $v \in V$ et tel que la somme des temps de traitement soit minimale. Le coût de ce couplage entier étant la même que celui des affectations sommet-unité de calcul induites par celui-ci, l'affectation des sommets de V a donc un coût d'au plus P. Le couplage entier ainsi obtenu pour l'instance étudiée est illustré en figure 3.11.

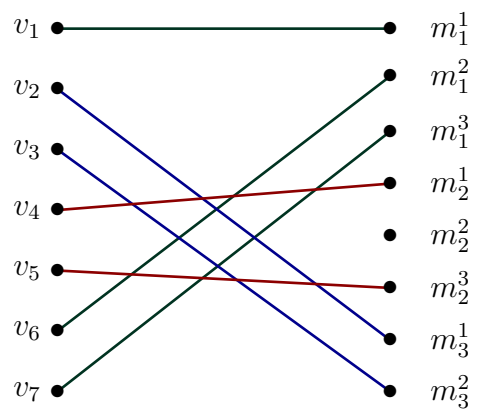


FIGURE 3.11 – Couplage maximum de somme des temps de traitement minimum sur le graphe biparti $B = (V, \mathcal{M}, F)$.

Il ne nous reste plus alors qu'à démontrer que l'occupation mémoire sur chaque machine respecte le rapport d'approximation du lemme 2. Pour cela, nous introduisons deux notations, illustrées en figure 3.12 :

— $\mathcal{W}min_k^i$ est le poids de voisinage minimum dans le graphe de voisinage G_l, parmi les sommets v du graphe biparti B qui partagent une arête avec le sommet m_k^i. Formellement :

$$\forall k = 1, \ldots, m, i \leq s_k, \mathcal{W}min_k^i = \min_{v:(v, m_k^i) \in F} \mathcal{W}_v. \tag{3.7}$$

— $\mathcal{W}max_k^i$ est le poids maximum du voisinage dans le graphe de voisinage G_l, parmi les

sommets v du graphe biparti B qui partagent une arête avec le sommet m_k^i. Formellement :

$$\forall k = 1, \ldots, m, i \leq s_k, \mathcal{W}max_k^i = \max_{v:(v,m_k^i)\in F} \mathcal{W}_v. \tag{3.8}$$

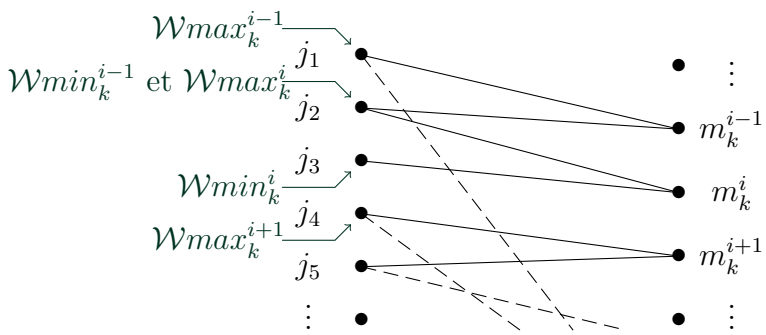


FIGURE 3.12 – Exemple d'une portion d'un graphe biparti $B = (V, \mathcal{M}, F)$ où les arêtes en pointillé représentent des arêtes reliant des sommets de V à des duplications d'unités de calcul différentes de k. Les sommets de V étant triés par poids de voisinage décroissant, on peut visuellement trouver $\mathcal{W}min_k^i$ et $\mathcal{W}max_k^i$.

Lemme 4. *Pour tout $k \in [\![1, m]\!]$ et $i \in [\![1, s_k - 1]\!]$, on a*

$$\mathcal{W}min_k^i \geq \mathcal{W}max_k^{i+1}. \tag{3.9}$$

Démonstration. Étant donné la construction du graphe biparti $B = (V, \mathcal{M}, F)$, la preuve se base sur la distinction de deux cas possibles pour m_k^i et m_k^{i+1} :

— Soit il existe v tel que $\{v, m_k^i\}$ et $\{v, m_k^{i+1}\}$ soient deux arêtes de B, et dans ce cas $\mathcal{W}min_k^i = \mathcal{W}max_k^{i+1}$. Ce cas est illustré par $\mathcal{W}min_k^{i-1}$ et $\mathcal{W}max_k^i$ en figure 3.12.

— Soit il n'existe pas v tel que $\{v, m_k^i\}$ et $\{v, m_k^{i+1}\}$ sont deux arêtes de B, et dans ce cas, les sommets ayant été ordonnés par ordre de poids de voisinage décroissant, on a $\mathcal{W}min_k^i \geq \mathcal{W}max_k^{i+1}$. Ce cas est illustré par $\mathcal{W}min_k^i$ et $\mathcal{W}max_k^{i+1}$ en figure 3.12.

□

Nous montrons maintenant que l'occupation mémoire sur une machine k donnée respecte le rapport d'approximation du lemme 2. Commençons par constater que la charge mémoire d'une unité de calcul k est au plus égale à $\sum_{i=1}^{s_k} \mathcal{W}max_k^i$.

Dans le cas où il n'y a qu'une seule duplication, c'est-à-dire $s_k = 1$, la charge mémoire est au plus égale à $\mathcal{W}max_k^1$. Sachant que le graphe biparti B est construit à partir d'une solution optimale du

programme linéaire $PL(P, \text{Mem})$, qui vérifie (6.7), il n'existe pas d'arête entre un sommet $v \in V$ et une duplication $m \in \mathcal{M}$ telle que le poids de voisinage de v soit plus grand que la mémoire de l'unité de calcul. On a donc

$$\mathcal{W}max_k^1 \leq \text{Mem}[k], \tag{3.10}$$

et donc la charge mémoire de l'unité de calcul k vérifie le lemme 2.

Dans le cas où il y a plusieurs duplications, c'est-à-dire $s_k > 1$, on a

$$\sum_{i=1}^{s_k} \mathcal{W}max_k^i = \mathcal{W}max_k^1 + \sum_{i=2}^{s_k} \mathcal{W}max_k^i. \tag{3.11}$$

Par application du lemme 4, nous avons

$$\sum_{i=2}^{s_k} \mathcal{W}max_k^i \quad \leq \quad \sum_{i=1}^{s_k-1} \mathcal{W}min_k^i. \tag{3.12}$$

Puisque $\mathcal{W}min_k^i$ est le minimum des $\mathcal{W}_v$, pour v tel que $\{v, m_k^i\} \in F$, $\mathcal{W}min_k^i$ est donc inférieur ou égal à n'importe quelle combinaison convexe de ces $\mathcal{W}_v$. Donc

$$\sum_{i=2}^{s_k} \mathcal{W}max_k^i \quad \leq \quad \sum_{i=1}^{s_k-1} \sum_{v:\{v,m_k^i\}\in F} \mathcal{W}_v \ f(\{v, m_k^i\}), \ et \tag{3.13}$$

$$\sum_{i=2}^{s_k} \mathcal{W}max_k^i \quad \leq \quad \sum_{i=1}^{s_k} \sum_{v:\{v,m_k^i\}\in F} \mathcal{W}_v \ f(\{v, m_k^i\}). \tag{3.14}$$

Puisque f est construit de manière à respecter l'équation (3.6), nous avons

$$\sum_{i=2}^{s_k} \mathcal{W}max_k^i \quad \leq \quad \sum_{v\in V} \mathcal{W}_v \ x_{v,k}^{opt}. \tag{3.15}$$

En outre, x^{opt} étant une solution de $LP(P, \text{Mem}[k])$, elle respecte donc l'équation (6.4) et nous pouvons alors écrire

$$\sum_{i=2}^{s_k} \mathcal{W}max_k^i \quad \leq \quad \sum_{v\in V} \mathcal{W}_v \ y_{v,k}^{opt}, \ et \tag{3.16}$$

$$\sum_{i=2}^{s_k} \mathcal{W}max_i^k \quad \leq \quad \sum_{v \in V} \frac{\mathcal{W}_v}{\omega_v} \; y_{v,k}^{opt} \; \omega_v \, . \tag{3.17}$$

Or puisque nous savons que

$$\mathcal{W}_v \leq \Delta \; \omega_{max}, \; \forall v \in V,$$

et que

$$\omega_v \geq \omega_{min}, \; \forall v \in V,$$

nous pouvons écrire

$$\frac{\mathcal{W}_v}{\omega_v} \leq \frac{\Delta \; \omega_{max}}{\omega_{min}}, \; \forall v \in V. \tag{3.18}$$

L'équation (3.17) aboutit donc à

$$\sum_{i=2}^{s_k} \mathcal{W}max_k^i \quad \leq \quad \frac{\Delta \; \omega_{max}}{\omega_{min}} \; \sum_{v \in V} y_{v,k}^{opt} \; \omega_i. \tag{3.19}$$

Et puisque y^{opt} vérifie l'équation (6.3), on obtient finalement

$$\sum_{i=2}^{s_k} \mathcal{W}max_k^i \quad \leq \quad \text{Mem}[k] \; \Big(\Delta \frac{\omega_{max}}{\omega_{min}}\Big). \tag{3.20}$$

Nous avons donc démontré que dans le pire cas, c'est-à-dire dans le cas où les voisinages de chaque sommet affecté à une unité de calcul sont disjoints, l'occupation mémoire sur chaque machine est au plus égale à $\sum_{i=1}^{s_k} \mathcal{W}max_l k^i \leq \Big(1 + \Delta \frac{\omega_{max}}{\omega_{min}}\Big)$ Mem$[k]$. Cette occupation respecte donc le rapport d'approximation du lemme 2 et termine la preuve de celui-ci.

Le lemme 2 étant démontré, nous pouvons maintenant prouver le théorème 2. En effet, s'il existe une solution au problème $R|G_l, mem| \sum p_{j,k}$, il nous est possible de l'obtenir en réalisant une recherche dichotomique sur le paramètre P de $PL(P, \text{Mem})$, et en utilisant l'algorithme introduit dans la preuve du lemme 2. Cette solution approchée est obtenue en temps polynomial et est de coût optimal $P_{opt}(M)$.

□

Remarquons qu'il est possible d'obtenir un résultat complémentaire à celui présenté précédemment. Notons P_{opt} la plus petite somme totale des temps de traitement engendrés parmi toutes les solutions possibles (sans contraintes mémoire). Notons $M_{opt}(P)$ la plus petite occupation mémoire

parmi toutes les solutions (entières) qui ont une somme totale des temps de traitement au plus égale à P.

Théorème 3. *Il existe une approximation bi-critère pour le problème $R|G_l, mem|\sum p_{j,k}$, c'est-à-dire un algorithme en temps polynomial retournant une affectation des sommets dont la somme totale des temps de traitement est P_{opt} avec une occupation mémoire $\left(1+\Delta\frac{\omega_{max}}{\omega_{min}}\right)M_{opt}(P_{opt})$.*

Démonstration. Commençons par obtenir P_{opt} en plaçant chaque sommet sur l'unité de calcul permettant à ce sommet d'obtenir le plus petit temps de traitement possible. Nous réalisons alors une recherche dichotomique sur l'intervalle $[\![0, M_{max}]\!]$, où $M_{max} = \sum_{v\in V}\omega_v$, pour trouver la plus petite taille mémoire M pour laquelle il existe une solution fractionnelle à $PL(P_{opt}, \text{Mem})$. Cela requiert de résoudre le programme linéaire au plus $\log(M_{max})$ fois. Notons $M_{opt}(P_{opt})$ la mémoire de la solution fractionnelle obtenue.

On applique alors l'algorithme ayant permis de construire le graphe biparti B pour la preuve du théorème 2. On obtient ainsi une solution avec un temps de traitement total d'au plus P_{opt} et d'occupation mémoire au plus égale à $\left(1+\Delta\frac{\omega_{max}}{\omega_{min}}\right)M_{opt}(P_{opt})$. La preuve est complétée en observant que P_{opt} ne peut pas être plus grand que le plus petit coût de toute solution, et $M_{opt}(P_{opt})$ ne peut pas être plus grand que la plus petite occupation mémoire de toute distribution ne dépassant pas un coût d'au plus P_{opt}.

L'algorithme décrit ici effectue $O(\log(W_{max}))$ appels à une résolution de programme linéaire, en appliquant l'algorithme du lemme 2. Chacune de ces étapes peut être réalisée en temps polynomial.

□

Dans cette section, nous avons présenté un algorithme en temps polynomial qui, soit prouve qu'il n'existe pas de solution valide au problème $R|G_l, mem|\sum p_{v,k}$, soit retourne une solution dont la somme des temps de traitement est minimale et où la capacité de chaque unité de calcul est excédée par un facteur au plus $\left((\Delta+1)\frac{\omega_{max}}{\omega_{min}}+1\right)$, où ω_{max} (respectivement ω_{min}) est le poids mémoire maximal (respectivement minimal). Contrairement à l'algorithme présenté en section 3.2, l'algorithme présenté ici n'est pas dépendant d'un paramètre du graphe et ne considère pas que le nombre d'unités de calcul est fixé. Cependant, nous pouvons constater que, lorsqu'il retourne une solution, celle-ci peut induire un dépassement borné de la capacité des unités de calcul.

3.4 Conclusion

Dans ce chapitre, nous avons vu comment notre problème de partitionnement peut être énoncé comme un problème d'ordonnancement utilisant un graphe de voisinage. Nous avons pu constater ses liens avec d'autres problèmes **NP**-difficiles et nous avons considéré sa résolution dans le cas de graphes à paramètres fixés. Nous avons ainsi pu obtenir un algorithme approché FPT selon la largeur linéaire du graphe, pour le problème $Rk|G, mem|C_{max}$. De plus, nous nous sommes intéressés au problème $R|G_l, mem|\sum p_{v,l}$, toute solution de ce problème étant une solution valide à $R|G, mem|C_{max}$. Nous avons utilisé la méthode de résolution de [75], qui nous semblait pouvoir être adaptée à notre problème. Le résultat que nous avons alors obtenu n'est pas aussi satisfaisant que nous l'espérions, puisque la borne sur le dépassement maximal de la capacité des machines dépend du poids des tâches. Par conséquent, nous avons étudié d'autres méthodes de résolution classiques, telles que la méthode des arrondis itérés et la relaxation lagrangienne, que nous souhaitions appliquer à ces deux problèmes. Nous nous sommes alors heurtés à différentes difficultés, qui nous ont empêché d'obtenir des résultats satisfaisants à l'aide de ces méthodes.

Nous nous sommes alors tournés vers une des méthodes les plus populaires pour résoudre les problèmes de partitionnement de graphe : la méthode multi-niveaux. L'adaptation de cette méthode à la résolution de notre problème, nécessite une modélisation adaptée du maillage, un algorithme de partitionnement direct et une méthode d'optimisation locale. Dans la section suivante, nous présentons une modélisation du maillage permettant une contraction correcte de celui-ci. Nous présentons une heuristique nous permettant d'obtenir un partitionnement initial valide vis-à-vis des contraintes mémoires. Enfin, nous validons cette heuristique de manière expérimentale.

Chapitre 4

Heuristiques et schéma multi-niveaux

Le problème de partitionnement de maillage sous contraintes mémoire est **NP**-difficile (voir la section 3.1.2), tout comme le k-partitionnement de graphe [29]. Trouver une solution optimale dans le cas d'un graphe $G = (V, E)$ quelconque nécessite un nombre d'opérations proportionnel au nombre de k-partitions de G, c'est-à-dire en $\Omega(k^n)$. Trouver la meilleure meilleure k-partition satisfaisant nos contraintes en parcourant l'ensemble des partitions de G est donc matériellement impossible pour la plupart des graphes. Par conséquent, nous nous intéressons dans ce chapitre à la conception d'heuristiques, c'est-à-dire d'algorithmes efficaces en pratique et fournissant une solution approchée dont on ne sait pas garantir a priori la qualité.

Lorsque nous nous sommes tournés vers l'utilisation d'heuristiques pour résoudre le problème de partitionnement de maillage sous contraintes mémoire, une des difficultés rencontrées a été la taille des instances. Compte tenu du fait que dans le contexte du partitionnement de graphe, ou d'hypergraphe, les algorithmes multi-niveaux permettent d'obtenir des résultats de bonne qualité pour des graphes de très grande taille [14, 17], nous avons cherché à adapter cette technique à notre problème. La technique du multi-niveaux a été introduite au milieu des années 1990 [5, 40, 81] et a pour avantage de réduire la taille du graphe ainsi que l'espace de recherche. Elle est d'ailleurs utilisée par de nombreux outils de partitionnement. C'est notamment le cas des outils de partitionnement de graphes MeTiS [48], Jostle [83] et Scotch [67] ; et des outils de partitionnement d'hypergraphes Zoltan [9] et Par$k$way [78].

Dans ce chapitre, nous commençons par introduire le schéma multi-niveaux dans le cadre du partitionnement de graphe. Nous passerons ensuite à l'adaptation de ce schéma pour notre pro-

blème. Finalement, nous fournirons plusieurs résultats expérimentaux et ferons des comparaisons avec l'outil de partitionnement multi-niveaux SCOTCH.

4.1 Introduction au multi-niveaux

Cette technique est très utilisée pour le partitionnement de graphe et d'hypergraphe, et permet aux algorithmes combinatoires utilisés pour résoudre ces problèmes, d'avoir un aperçu global du graphe. L'idée principale est de traiter un graphe réduit ayant les mêmes propriétés topologiques que le graphe originel. Pour cela, le schéma multi-niveaux se décompose en trois phases :

1. la *phase de contraction*, qui correspond à un processus itératif où certains sommets d'un graphe $G_l = (V_l, E_l)$ sont agrégés pour définir un graphe plus petit (dit grossier) $G_{l+1} = (V_{l+1}, E_{l+1})$. Ainsi, après un nombre N d'itérations de ce processus, on obtient une famille de graphes $\{G_0, \ldots, G_N\}$, où G_0 est le graphe originel. Chaque graphe G_l correspond au *niveau* l du schéma multi-niveaux ;

2. la *phase de partitionnement initial*, qui consiste à créer une partition du graphe le plus grossier $G_n = (V_n, E_n)$. La solution de cette phase est une application $S_n : V_n \to [\![1; k]\!]$, qui associe un numéro de partie à chaque sommet du graphe G_n ;

3. la *phase d'expansion*, qui correspond à un processus itératif où la solution S_i du graphe $G_i = (V_i, E_i)$ est prolongée sur le graphe de niveau inférieur $G_{i-1} = (V_{i-1}, E_{i-1})$ puis affinée, c'est-à-dire que la valeur de la fonction objectif est optimisée. Ce raffinement est effectué à l'aide d'un algorithme d'optimisation locale dont le résultat est la solution $S_{i-1} : V_{i-1} \to [\![1; k]\!]$. Ainsi, en partant de la solution S_n du graphe le plus grossier, nous obtenons une solution S_0 du graphe originel à la fin des n itérations de ce processus.

Le fonctionnement de ce procédé est présenté en figure 4.1 dans le cas d'un 4-partitionnement.

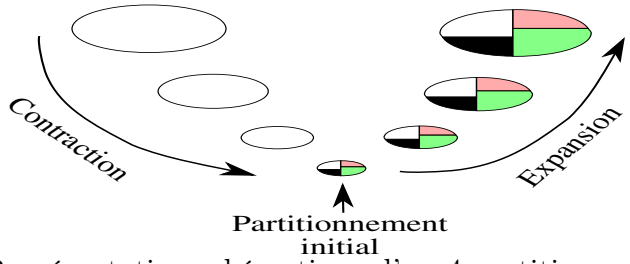


FIGURE 4.1 – Représentation schématique d'un 4-partitionnement multi-niveaux.

L'algorithme 6 présente le schéma multi-niveaux lors de son application au k-partitionnement de graphe. Cet algorithme est très général et son efficacité dépend fortement des méthodes utilisées lors des phases de contraction, de partitionnement initial et d'expansion.

Algorithme 6 Algorithme multi-niveaux pour le k-partitionnement de graphe.

1: **Procédure** MULTI-NIVEAUX(G, k)
2: $i \leftarrow 0$
3: $G_i \leftarrow G$
4: ▷ Contraction du graphe
5: **Tant que** G_i trop grand ou contraction importante **Faire**
6: $G_{i+1} \leftarrow$ CONTRACTER(G_i)
7: $i \leftarrow i + 1$
8: **Fin Tant que**
9: ▷ Partitionnement du graphe le plus grossier
10: $S_i \leftarrow$ PARTITIONNER(G_i, k)
11: ▷ Projection et affinement de la solution
12: **Tant que** $i > 0$ **Faire**
13: $i \leftarrow i - 1$
14: $S_i \leftarrow$ PROJETER(S_{i+1}, G_i)
15: $S_i \leftarrow$ AFFINER(S_i)
16: **Fin Tant que**
17: **Retourner** S_i
18: **Fin Procédure**

Comme le nombre de partitions possibles croît exponentiellement avec le nombre de sommets du graphe et que la contraction diminue le nombre de sommets de celui-ci, il est très avantageux de contracter le graphe. En particulier, il est avantageux d'utiliser un algorithme d'optimisation globale sur un graphe réduit, car il sera moins coûteux de le faire fonctionner. En contrepartie, seules les solutions du graphe contracté sont étudiées par l'algorithme de partitionnement initial. Cependant, en travaillant sur plusieurs niveaux, l'algorithme de raffinement local permet d'améliorer la qualité de la solution vis-à-vis de la fonction objectif. Ainsi, bien que la partition du graphe contracté puisse ne pas induire une bonne partition du graphe originel, l'application répétée de l'algorithme de raffinement local tend à résoudre ce problème.

4.1.1 Phase de contraction

Le but de cette phase est de produire une famille $(G_i)_{1 \leq i \leq n}$ de versions de plus en plus grossières du graphe originel G, telle que leur topologie soit proche de celle de G et que leur nombre de sommets soit strictement décroissant. Pour cela, la phase de contraction repose essentiellement sur

des fusions d'ensembles de sommets de G, conduisant ainsi à la réduction du nombre de sommets du graphe.

Généralement, l'opération de fusion de sommets s'effectue sur des sommets appariés, c'est-à-dire regroupés deux par deux, pour former un sommet grossier. Il existe différentes stratégies pour apparier les sommets :

— la méthode aléatoire, qui consiste à apparier un sommet $v \in G$ à l'un de ses voisins $v' \in \mathcal{N}(v)$ choisi aléatoirement ; c'est l'approche qui a été utilisée dans les premières implantations du schéma multi-niveaux [10][40].

— l'heuristique *Heavy Edge Matching* (HEM) [45], qui consiste à apparier un sommet $v \in G$ à son voisin $v' \in \mathcal{N}(v)$ dont l'arête $e = (v, v')$ est de poids le plus élevé possible.

— l'heuristique *Sorted Heavy Edge Matching* (SHEM), qui est une version légèrement modifiée de l'heuristique HEM. Dans celle-ci, les sommets sont triés en fonction de leur degré avant d'appliquer la méthode HEM sur cette liste triée.

La méthode HEM est utilisée par les outils de partitionnement de graphe SCOTCH et JOSTLE, car elle semble bien conserver la topologie du graphe. Dans le cas de METIS, c'est la méthode SHEM qui est utilisée [46] par défaut, car il semble que le tri des sommets améliore la qualité de la partition finale trouvée par le schéma multi-niveaux.

Tout sommet v'' résultant de la contraction des sommets $v \in V$ et $v' \in V$ a alors pour caractéristiques d'être pondéré par la somme des poids des sommets v et v', et d'être adjacent aux sommets de $\mathcal{N}(v) \cup \mathcal{N}(v') \backslash \{v, v'\}$.

Notons que l'opération de fusion peut s'effectuer sur des ensembles de sommets, et non pas uniquement des appariements. La réduction de la taille du graphe peut alors être plus rapide que dans le cas de l'appariement. Cela permet notamment de limiter le nombre de niveaux dans le schéma multi-niveaux.

4.1.2 Phase de partitionnement initial

Une fois la phase de contraction terminée, c'est-à-dire quand le nombre de sommets du graphe contracté G_n est suffisamment petit, une heuristique de partitionnement est appliquée sur G_n. L'objectif de cette phase est de trouver une partition S_n du graphe contracté G_n. Celle-ci sera ensuite utilisée comme partition initiale de la phase d'expansion.

Il existe différents types d'heuristiques pour obtenir une solution initiale et le type utilisé varie

selon les applications. Dans cette section, nous présenterons les deux approches les plus utilisées par les algorithmes multi-niveaux : l'approche spectrale [40, 5] et l'approche de grossissement de bulles [47].

L'approche spectrale

L'utilisation de l'approche spectrale pour la résolution du problème de partitionnement de graphe est ancienne et fut introduite dans [21, 22]. Avant d'être utilisée lors de la phase de partitionnement initial d'un algorithme multi-niveaux [39], c'était la méthode multi-niveaux qui était utilisée pour simplifier le calcul nécessaire à l'approche spectrale [5]. L'approche spectrale consistant à rechercher des valeurs propres de la matrice de Laplace M_{Lap} associée au graphe $G = (V, E)$, nous définissons cette matrice.

Définition 30. (Matrice de Laplace) *Soit $G = (V, E)$ un graphe. La matrice de Laplace M_{Lap} associée à G est définie pour tout $(i, j) \in [\![1; |V|]\!]^2$ par :*

$$(M_{Lap})_{i,j} = \begin{cases} \delta(v_i) & \textit{si } i = j, \\ \textit{-1} & \textit{si } i \neq j \textit{ et } \{i, j\} \in E, \\ \textit{0} & \textit{sinon.} \end{cases} \tag{4.1}$$

La matrice de Laplace M_{Lap} du graphe $G = (V, E)$ est semi-définie positive, donc ses valeurs propres sont positives. Elles peuvent donc être ordonnées de la façon suivante : $0 \leq \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$. Le vecteur unité $X_1 = (1, \ldots, 1)^T$ étant un vecteur propre trivial dont la valeur propre est $\lambda = 0$, nous avons $\lambda_1 = 0$. Cette famille de vecteurs ayant été intensivement étudiée par Fiedler [26], celle-ci est souvent appelée la famille des vecteurs de Fiedler et le vecteur propre X_2 associé à la valeur propre λ_2 est appelé *vecteur de Fiedler*. Ce vecteur propre possède des propriétés permettant d'ordonner les sommets de G sur la droite des réels en associant à chaque sommet v_i une position x_i correspondant à sa $i^{\text{ème}}$ composante dans le vecteur de Fiedler X_2.

Hall [36] a montré que la solution optimale du placement des sommets sur une droite est donnée par cette deuxième plus petite valeur propre λ_2. Ceci implique que si deux sommets v_i et v_j sont connectés par une arête de E, la distance $|x_i - x_j|$ est petite. Les sommets fortement connectés sont donc proches dans l'ordonnancement des sommets. Nous pouvons donc déduire une partition $\Pi = (P_0, P_1)$ de G en choisissant un réel ρ et en posant $P_0 = \{v_i | x_i \leq \rho\}$ et $P_1 = \{v_i | v_i > \rho\}$. Les autres p vecteurs de Fiedler peuvent être utilisés par ordre croissant pour obtenir une 2^p-partition.

Cependant, à cause des contraintes d'équilibre, le nombre de vecteurs de Fiedler pouvant être utilisés est limité [39].

Afin d'obtenir une k-partition du graphe, avec $k > 2$, un entier quelconque, une technique de bissections récursives peut être utilisée. Dans le cadre du partitionnement de graphe, cette technique consiste à réaliser récursivement des bipartitions du graphe. Ainsi, on commence tout d'abord par obtenir une bipartition d'un graphe $G = (V, E)$, et ensuite on obtient récursivement une bipartition des deux parties générées. Après $\log k$ étapes, le graphe $G = (V, E)$ est partitionné en k parties. L'approche spectrale récursive [68, 76] consiste à réaliser récursivement des bipartitions du graphe en utilisant l'approche spectrale.

Un des inconvénients de l'approche spectrale est qu'elle nécessite des algorithmes de recherche de vecteurs propres qui sont coûteux en terme de calculs [5]. Cependant, il est raisonnable de l'utiliser lors de la phase de partitionnement initial où le graphe est de petite taille (typiquement 100 sommets).

L'approche de grossissement de bulles

L'approche de grossissement de bulles est une approche gloutonne, faisant partie de la classe des algorithmes de diffusion. Elle regroupe un ensemble de méthodes généralement employées de manière récursive, simples à mettre en œuvre et efficaces sur des graphes de petite taille (au plus quelques centaines de sommets). Le fonctionnement d'une telle méthode est présenté dans l'algorithme 7. Elle consiste à partir de plusieurs sommets du graphe qui vont constituer le centre des bulles, c'est-à-dire les points d'où la diffusion est initiée. L'étape principale de l'algorithme consiste alors à faire grossir les bulles depuis leur centre par une méthode de propagation, c'est-à-dire rassembler les sommets du graphe dans chacun de ces ensembles. Cette étape est réalisée jusqu'à ce que les bulles se rencontrent en recouvrant tous les sommets du graphe. La nouvelle partition est alors définie en prenant les bulles comme parties, et les centres $\mathcal{C}$ sont recalculés. On itère ce procédé glouton jusqu'à obtenir une partition d'une certain qualité, ou quand le résultat obtenu n'évolue plus.

Une des versions les plus simples de cette approche, dans laquelle la phase de redéfinition des centres n'est pas réalisée, est la méthode du *Graph Growing Partitioning* (GGP) [47] . La solution retournée par l'algorithme de grossissement de bulles étant dépendante des sommets choisis au départ, il est fréquent d'appeler plusieurs fois l'algorithme sur des centres choisis aléatoirement et

Algorithme 7 Algorithme de grossissement de bulles

1: **Fonction** BULLES(G, k)
2: ▷ $\mathcal{C}$: ensemble des centres des bulles
3: $\mathcal{C} \leftarrow k$ sommets choisis aléatoirement
4: **Tant que** la coupe est supérieure à un certain seuil **Faire**
5: Faire grossir les bulles depuis leur centre $\mathcal{C}$: obtention de Π
6: Mettre à jour les centres $\mathcal{C}$ des parties de Π
7: **Fin Tant que**
8: **Retourner** Π
9: **Fin Fonction**

de conserver la meilleure partition obtenue.

Notons que pour obtenir une partition initiale, en plus de l'utilisation des heuristiques précédemment citées, il est possible d'utiliser des méthodes exactes ou des méthodes ne prenant pas en compte la coupe.

4.1.3 Phase d'expansion

Le but de cette phase est d'utiliser la partition S_n, obtenue suite à la phase de partitionnement initial, en la projetant jusqu'au graphe originel G_0. Pour cela, la partition initiale est projetée de niveau en niveau, c'est-à-dire du niveau $i+1$ au niveau i avec $0 \leq i \leq n-1$. Ceci peut être effectué par un simple algorithme de projection qui procède de la manière suivante. Chaque sommet de $G_{i+1} = (V_{i+1}, E_{i+1})$ est traversé, et pour chaque sommet grossier $v \in V_{i+1}$, l'ensemble des sommets de V_i, dont il est la fusion, est attribué à $S_{i+1}(v)$.

À chaque niveau successif, la solution S_i peut être raffinée. Pour cela, il existe plusieurs algorithmes d'optimisation locale qui, partant d'une partition Π_0 de G valide et bien équilibrée, se déplacent dans l'espace des solutions en sélectionnant une solution voisine la plus à même de réduire la coupe de la partition. Parmi les algorithmes d'optimisation locale, nous pouvons citer l'algorithme de Kernighan-Lin (KL), ainsi que l'algorithme de Fiduccia et Mattheyses (FM) que nous présentons dans cette section.

L'algorithme de Kernighan-Lin (KL)

Cet algorithme fut introduit par Kernighan et Lin dans [50]. Il consiste à effectuer des échanges de sommets entre les différentes parties. Ainsi, n'importe quel sommet d'une paire $(u, v) \in V^2$, tel que $u \in P_i$, $v \in P_j$ et $P_i \neq P_j$, peut être échangé. Pour choisir les sommets à échanger, les auteurs

introduisent les notions de coût extérieur et de coût intérieur. Le coût extérieur, noté $E(u)$ pour le sommet u affecté à la partie P_i, est défini comme la somme des poids des arêtes $e = \{u, v\}$ telles que $v \notin P_i$:

$$E(u) = \sum_{\substack{e=\{u,v\} \\ v \notin P_i}} \omega(e). \tag{4.2}$$

Le coût intérieur, noté $I(u)$ pour le sommet u affecté à la partie P_i, est défini comme la somme des poids des arêtes $e = \{u, v\}$ telles que $v \in P_i$:

$$I(u) = \sum_{\substack{e=\{u,v\} \\ v \in P_i}} \omega(e). \tag{4.3}$$

Soit $D(u)$ la différence entre le coût extérieur et le coût intérieur du sommet u :

$$D(u) = E(u) - I(u). \tag{4.4}$$

Le gain résultant de l'échange de $u \in P_1$ et $v \in P_2$ est alors :

$$gain(u, v) = D(u) + D(v) - 2 \times \omega(u, v). \tag{4.5}$$

L'algorithme KL est présenté dans l'algorithme 8. Celui-ci fonctionne par *passes*, c'est-à-dire qu'il effectue plusieurs itérations, comme nous pouvons le voir à la ligne 4 de l'algorithme. Ces passes font appel aux gains associés aux échanges des sommets (voir équation 4.5 dans le cas d'un bipartitionnement de graphe dont les sommets sont pondérés par un poids unitaire). Ceux-ci permettent d'évaluer la variation de la qualité du partitionnement vis-à-vis de la fonction objectif. Ainsi, une valeur de gain positive indique une amélioration du partitionnement.

Le cœur de l'algorithme est d'effectuer le meilleur échange possible parmi les échanges disponibles, puis de verrouiller les deux sommets déplacés. La coupe est alors mise à jour et l'échange effectué est mémorisé. Une fois qu'il n'existe plus de sommets non verrouillés à échanger, l'algorithme parcourt l'ensemble des mouvements mémorisés à la recherche de la meilleure valeur de coupe, c'est-à-dire la valeur maximale de la somme des gains. L'algorithme revient alors à cette configuration et déverrouille les sommets. Ce processus est itéré jusqu'à ce qu'il ne soit plus possible d'améliorer la qualité de la partition.

Nous remarquons que les échanges sont effectués dans l'ordre donné par les gains. Ainsi, l'al-

gorithme peut sortir des extrema locaux puisqu'il autorise les échanges dégradant temporairement la qualité de la partition (à la ligne 9 de l'algorithme, le gain peut être négatif s'il n'existe pas d'autres échanges possibles). Néanmoins, le choix de l'échange à effectuer lorsque plusieurs échanges de même gain sont disponibles peut avoir d'importantes conséquences sur la partition finale. C'est pourquoi il est intéressant de réaliser plusieurs exécutions de l'algorithme KL et de conserver la meilleure solution obtenue.

Algorithme 8 Algorithme de Kernighan-Lin

1: **Procédure** KL(G, Π)
2: i : rang de l'itération interne
3: SG_i : somme des gains jusqu'au rang i
4: **Répéter** ▷ Passe sur le graphe G
5: $i \leftarrow 0$ ▷ Initialisation des variables pour une itération
6: $SG_i \leftarrow 0$
7: ▷ Boucle interne de l'algorithme KL
8: **Tant que** (il existe un échange de deux sommets non verrouillés) **Faire**
9: Faire le meilleur échange possible
10: Verrouiller les deux sommets déplacés
11: Enregistrer le gain g_i de l'échange
12: $SG_{i+1} \leftarrow SG_i + g_i$
13: $i \leftarrow i + 1$
14: **Fin Tant que**
15: Trouver x telle que la somme partielle SG_x des gains soit maximale
16: **Si** $SG_x < 0$ **Alors**
17: Annuler tous les échanges effectués
18: **Sinon**
19: Annuler les mouvements de x à i
20: **Fin Si**
21: **Jusqu'à ce que** $S_x < 0$
22: **Fin Procédure**

En utilisant une liste triée selon les gains, l'étape de sélection du meilleur échange possible peut s'effectuer en $\Theta(n \log n)$; comme la boucle est parcourue au plus n fois, la complexité en temps de la boucle interne peut être ramenée à $\Theta(n^2 \log n)$. Le nombre de passes, c'est-à-dire le nombre d'itérations de la boucle externe est quant à lui borné par le nombre d'arêtes.

L'algorithme de Fiduccia-Mattheyses (FM)

Cet algorithme, introduit dans [25], est inspiré par l'algorithme KL, mais effectue des mouvements de sommets d'une partie vers une autre et non des échanges. Cet algorithme réutilise la notion de gain introduite dans [50]. Le gain évalue la variation de la qualité du partitionne-

ment, vis-à-vis de la fonction objectif, lors de la migration d'un sommet d'une partie à l'autre. Les sommets sont classés selon leur gain et un tableau de listes chaînées de sommets, indexé par les gains, est utilisé ; chaque liste contient les sommets de gain correspondant à l'indice de la case du tableau qui la contient. Ce tableau est borné par une valeur maximale et une valeur minimale du gain (souvent l'opposée de la maximale). En pratique, un tableau de gains est utilisé pour chaque partie comme cela est illustré par la figure 4.2.

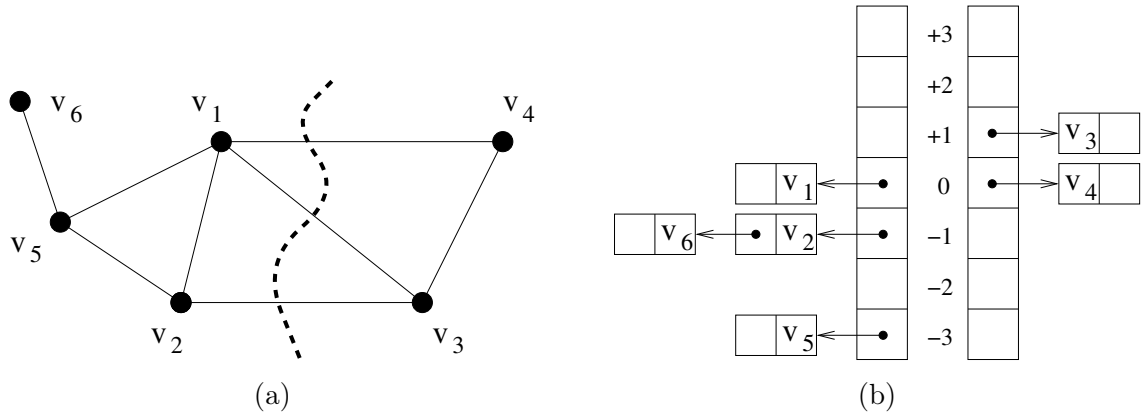


FIGURE 4.2 – Structures de données utilisées par l'heuristique de Fiduccia-Mattheyses.

L'heuristique de Fiduccia-Mattheyses est présentée dans l'algorithme 9. Elle consiste à sélectionner le sommet u induisant le meilleur gain parmi les migrations possibles, à verrouiller u et à mettre à jour les gains des sommets voisins de u, c'est-à-dire les sommets $v \in \mathcal{N}(u)$, n'ayant pas déjà été déplacés. La position de ces voisins dans la table des gains est alors mise à jour et le procédé réitéré. Nous constatons que la plus grosse différence avec l'algorithme KL repose sur le fait que l'on travaille sur les sommets du graphe et non sur les arêtes. En effet, l'algorithme FM déplace des sommets d'une partie à une autre tandis que KL réalise des échanges de sommets. Ces deux algorithmes diffèrent aussi sur la capacité de FM à trouver le sommet de plus grand gain en temps quasi-constant. Une étude plus complète de cette caractéristique peut être trouvée dans la thèse de François Pellegrini [65]. Le temps quasi-constant est obtenu grâce à la structure d'ordonnancement des gains, qui associe à chaque valeur possible pour les gains la liste des sommets correspondant à ce gain. Le temps nécessaire pour sélectionner un sommet de meilleur gain est donc proportionnel au nombre de valeurs possibles, qui est constant lors du déroulement de l'algorithme. La mise à jour de cette structure étant effectuée uniquement pour les sommets non verrouillés, elle devient de moins en moins coûteuse lorsque le nombre de boucles effectuées augmente. La complexité en espace de l'algorithme est en $\Theta(n + m)$ et celle en temps est aussi en $\Theta(n + m)$ [65].

Comme pour l'algorithme KL nous noterons qu'en général plusieurs sommets correspondent au gain maximum et que le choix d'un sommet peut grandement influencer la solution finale obtenue par l'algorithme [35][55]. Par conséquent, comme pour l'algorithme KL, de nombreuses implantations de l'algorithme FM effectuent plusieurs exécutions et conservent le meilleur résultat obtenu.

Algorithme 9 Algorithme de Fiduccia-Mattheyses

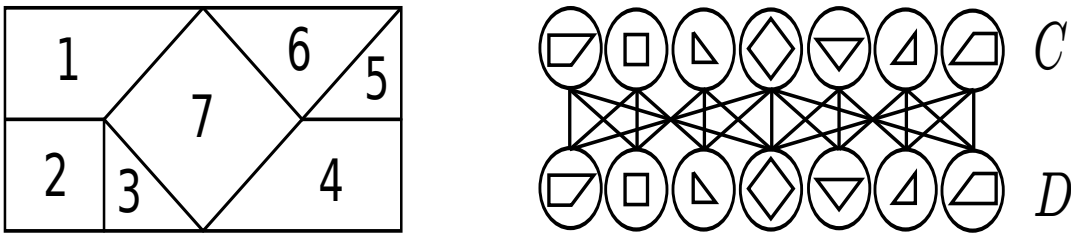
```
 1: Procédure FM(G, Π )
 2:     i : rang de l'itération interne
 3:     SG_i : somme des gains jusqu'au rang i
 4:     Répéter                                          ▷ On itère sur le graphe G
 5:         i ← 0                                        ▷ Initialisation des variables pour une itération
 6:         SG_i ← 0
 7:         Calculer les gains des sommets
 8:         Ordonner les gains des sommets
 9:         ▷ Boucle interne de l'algorithme FM
10:         Tant que Il existe un sommet à déplacer Faire
11:             Sélectionner le sommet u ∈ V correspondant au meilleur déplacement possible
12:             Déplacer u
13:             Verrouiller u
14:             Pour v non verrouillé tel que v ∈ N(u) Faire
15:                 Mettre à jour le gain pour v
16:             Fin Pour
17:             Ordonner les gains des sommets non verrouillés
18:             g_i ← le gain du déplacement
19:             SG_{i+1} ← SG_i + g_i
20:             i ← i + 1
21:         Fin Tant que
22:         Trouver x telle que la somme partielle SG_x des gains soit maximale
23:         Si SG_x < 0 Alors
24:             Annuler tous les déplacements effectués
25:         Sinon
26:             Annuler les déplacements de x à i
27:         Fin Si
28:     Jusqu'à ce que SG_x < 0
29: Fin Procédure
```

Maintenant que nous avons passé en revue les techniques couramment utilisées pour le partitionnement classique, nous allons décrire une adaptation de l'approche multi-niveaux pour notre problème de partitionnement de maillage.

4.2 Méthode multi-niveaux pour le partitionnement de maillage sous contraintes mémoire

La difficulté d'obtention d'une solution valide pour le problème que nous étudions nous a amené à adapter l'approche multi-niveaux classique. En effet, l'un des objectifs principaux étant d'assurer que les contraintes mémoire sont respectées, nous allons consacrer les phases de contraction et de partitionnement initial à cet effet. Contrairement aux problèmes de partitionnement de graphe et d'hypergraphe, où l'obtention d'une solution valide peut être réalisée à l'aide de méthodes simples [37][44], de telles méthodes ne permettent pas d'obtenir facilement une solution initiale au problème de partitionnement de maillage sous contraintes mémoire. Pour cette raison, nous proposons une heuristique de partitionnement basée sur le tri des sommets *calcul* du graphe $B = (C, D, E)$. Cette heuristique ayant pour but principal de fournir une partition respectant les contraintes mémoire, elle se concentre uniquement sur les données mémoire de l'instance et fait abstraction des coûts calculatoires. La solution ainsi obtenue étant non optimisée vis-à-vis du makespan, nous consacrons la phase d'expansion à l'amélioration de celui-ci, tout en conservant le respect des contraintes mémoire.

Nous détaillons maintenant ces différentes phases dans le cadre de notre modèle de graphe biparti (voir section 2.2, page 29). La figure 4.3 illustre ce modèle, où (a) est un maillage $2D$ composé de triangles et de quadrangles et (b) est le graphe biparti $B = (C, D, E)$ associé lorsque l'on considère que deux mailles sont voisines si elles partagent une arête. Nous réutiliserons ce maillage simple par la suite, notamment pour illustrer la phase de contraction.



(a) (b)

FIGURE 4.3 – Un maillage en deux dimensions (a) et son modèle biparti associé (b) où deux mailles sont adjacentes si elles partagent une même arête.

4.2.1 Phase de contraction

Le but de cette phase est de produire une version grossière $B_n = (C_n, D_n, E_n)$ du graphe biparti $B = (C, D, E)$ originel. Nous utiliserons alors ce graphe biparti pour obtenir une première partition valide vis-à-vis des contraintes mémoire. Cette phase de contraction repose essentiellement sur une opération qui correspond à la fusion d'ensembles de sommets *calcul* de B. Nous commencerons par détailler cette opération, avant d'expliquer comment nous effectuons un niveau de contraction puis décrirons la stratégie de contraction globale. Par souci de lisibilité, l'opération de fusion de sommets *calcul* est décrite dans le cas d'appariements, mais peut être étendue à n'importe quel type de fusion de sommets *calcul*.

Fusion de deux sommets *calcul*

Soit $F = \{c_0, c_1\}$ un ensemble de sommets *calcul* de $B = (C, D, E)$ que nous souhaitons fusionner en un sommet grossier c'. Nous réalisons le procédé suivant, illustré en figure 4.4 :

1. Tout d'abord, nous commençons par identifier l'ensemble des sommets *donnée* nécessaires à la réalisation des calculs des sommets de F. Cet ensemble de sommets correspond aux sommets de $\bigcup_{c \in F} \mathcal{N}(c)$ (voir la figure 4.4(a)).
2. Ensuite, nous supprimons chaque arête $e = \{c, d\} \in E$, où $c \in F$ et $d \in \mathcal{N}(c)$ (voir la figure 4.4(b)).
3. Nous remplaçons alors les sommets de F par le sommet c' que nous pondérons par un coût calculatoire $\mu(c') = \sum_{c \in F} \mu(c)$ (voir la figure 4.4(c)).
4. Finalement, nous ajoutons toutes les arêtes $e = \{c', d\}$, où $d \in \bigcup_{c \in F} \mathcal{N}(c)$ (voir la figure 4.4(d)).

Cette opération a la propriété fondamentale que l'affectation à une partie du nouveau sommet *calcul* grossier équivaut à l'affectation des anciens sommets *calcul* à cette même partie. En effet, le sommet *calcul* grossier satisfait le lemme suivant :

Lemme 5. *Affecter à l'unité de calcul $m \in M$ le sommet calcul grossier c', résultant de la fusion d'un ensemble F de sommets calcul fins, modélise exactement l'affectation des sommets fins de F à l'unité de calcul m.*

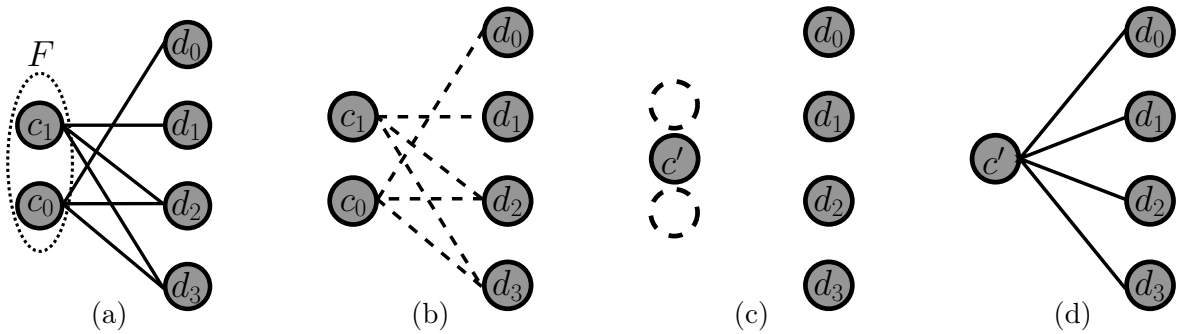


FIGURE 4.4 – *Fusion de $F = \{c_0, c_1\}$ en c'. En (a), les sommets donnée d_0, d_1, d_2, d_3 sont identifiés car ils sont les éléments de $\cup_{c \in F} \mathcal{N}(c)$. En (b), les arêtes incidentes aux sommets de F sont supprimées. En (c), les sommets de F sont remplacés par le sommet c'. En (d), le nouveau sommet c' est connecté aux sommets d_0, d_1, d_2, d_3 (initialement les éléments de $\cup_{c \in F} \mathcal{N}(c)$).*

Pour prouver le lemme précédent, nous allons porter notre attention sur les données nécessaires au sommet grossier et à son coût calculatoire.

Démonstration. Affecter les sommets calcul de F sur l'unité de calcul m implique que celle-ci doit stocker localement les sommets *donnée* adjacents à au moins un sommet de F. Or, lors de la création du sommet grossier c', nous avons généré les arêtes $e = (c', d)$, où $d \in \bigcup_{c \in F} \mathcal{N}(c)$. Nous avons donc conservé les besoins en informations induits par l'affectation des sommets de F sur l'unité de calcul m.

De plus, puisque le sommet c' a été pondéré par un coût calculatoire $\mu(c') = \sum_{c \in F} \mu(c)$, l'affectation de c' sur une unité de calcul m induit un coût calculatoire égal à l'affectation des sommets de F sur m. □

Puisque notre opération de fusion vérifie le lemme 5, la prolongation d'une solution obtenue à n'importe quel niveau est aussi une solution valide du problème.

Fusion de sommets *donnée*

L'opération précédemment décrite s'appliquant sur les arêtes et les sommets *calcul* du graphe biparti $B = (C, D, E)$, les ensembles C et E voient leur taille décroître au cours d'une contraction. Ainsi, l'étape de contraction permettant de passer d'un niveau à l'autre va réduire la taille du graphe par au plus $|C|/2$ sommets dans le cas d'appariements. Il est alors légitime de se demander s'il est possible de réduire encore plus la taille du graphe. Pour cela, nous nous intéressons à la réduction de la taille de l'ensemble D des sommets *donnée*. En effet, lorsque la fusion de sommets calcul engendre un sous-ensemble D' de sommets *donnée* qui sont tous adjacents à un même sous-

ensemble C' de sommets *calcul*, nous pouvons fusionner les sommets de D' en un nouveau sommet *donnée* d'. Ce nouveau sommet *donnée* est alors pondéré par un poids mémoire $\omega(d') = \sum_{d \in D'} \omega(d)$ et est connecté aux sommets *calcul* de C'. Cette opération, qui réduit la taille de l'ensemble D, peut être réalisée à n'importe quel moment de la phase de contraction. Nous avons évalué l'évolution du nombre de sommets *donnée* contractés en fonction du niveau pour le maillage $M_{quad,1211}$ (voir la figure 4.5). Les résultats obtenus sont résumés dans la table 4.1, où sont présentés pour chaque niveau : le nombre de sommets *calcul*, le nombre de sommets *donnée* et, pour chaque type de sommet, le rapport entre le nombre de ces sommets et le nombre de sommets originel. Nous pouvons ainsi constater que la méthode de contraction utilisée fait décroître rapidement le nombre de sommets *calcul*. Le nombre de sommets *donnée* décroît lui aussi, mais moins rapidement.

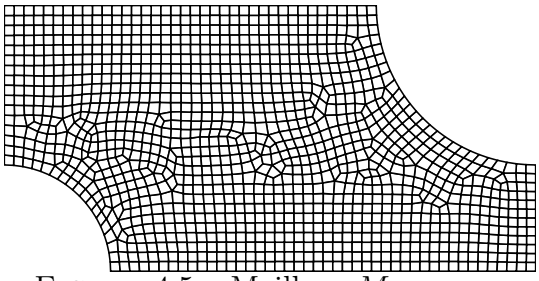


FIGURE 4.5 – Maillage $M_{quad,1211}$.

Niveau	Nombre de sommets *calcul*	Rapport	Nombre de sommets *donnée*	Rapport
0	1211	100 %	1211	100 %
1	690	56.9%	1197	98.8%
2	410	33.8%	1098	90.6%
3	262	21.6%	886	73.1%
4	182	15%	639	52.7%

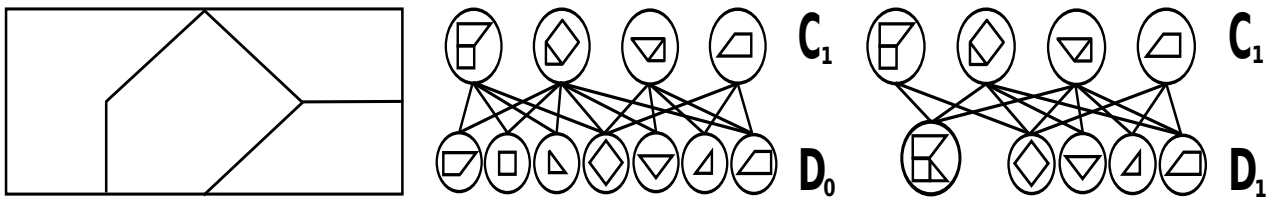
TABLE 4.1 – Évaluation du nombre de sommets *calcul* et *donnée* obtenus après chaque niveau de contraction.

La fusion de sommets ayant été présentée, nous expliquons maintenant comment un niveau de contraction est réalisé (voir l'algorithme 10). En fait, nous exécutons un niveau de contraction comme la plupart des schémas multi-niveaux pour le partitionnement de graphe ou d'hypergraphe. Plus précisément, nous commençons par rechercher une liste d'appariements des sommets *calcul* fins et nous fusionnons chacun de ces couples en un unique sommet *calcul* grossier.

Algorithme 10 Un niveau de contraction du graphe biparti : formation de B_{i+1} à partir de B_i

Fonction CONTRACTION($B_i = (C_i, D_i, E_i)$)
  $B_{i+1} \leftarrow (C_{i+1} = \emptyset, D_{i+1} = D_i, E_{i+1} = \emptyset)$
  couplage $\leftarrow$ COUPLAGESOMMETSCALCUL($B_i = (C_i, D_i, E_i)$)
  Pour tout $(c_0, c_1) \in$ *couplage* **Faire**
    Créer le sommet *calcul* c' dans C_{i+1}
    $\mu(c') \leftarrow \mu(c_0) + \mu(c_1)$
    Pour $d \in \mathcal{N}(c_0) \cup \mathcal{N}(c_1)$ **Faire**
      Créer l'arête $e = (c', d)$ dans E_{i+1}
    Fin Pour
  Fin Pour
  $B_{i+1} \leftarrow$ FUSIONSOMMETSDONNÉE(B_{i+1})
  Retourner $\{B_{i+1}, couplage\}$
Fin Fonction

La figure 4.6 illustre une telle contraction à la fois pour le maillage et pour le graphe biparti de la figure 4.3. La liste d'appariement associée à cette étape de contraction est $Couplage = [(1,2),(3,7),(5,6)]$. Le maillage et le graphe biparti résultant de la contraction sont respectivement illustrés en figures 4.6(a) et 4.6(b), où C_1 est l'ensemble des sommets *calcul* résultant d'un niveau de contraction et D_0 l'ensemble des sommets *donnée* originel. Suite à cette étape de contraction, puisque certains sommets *donnée* ont le même voisinage, le nombre de sommets *donnée* peut être réduit comme expliqué précédemment (voir figure 4.6(c)).



(a) (b) (c)

FIGURE 4.6 – *Exemple d'une étape de contraction effectuée sur le maillage de la figure 4.3(a) et le graphe biparti correspondant de la figure 4.3(b). Le maillage en (a) et le graphe biparti en (b) sont les résultats de l'étape de contraction induite par la liste d'appariement Couplage =* $[(1,2),(3,7),(5,6)]$. *L'ensemble de sommets* donnée *du graphe biparti est réduit en (c) puisque certains sommets* donnée *peuvent être fusionnés.*

Puisque nous souhaitons contracter le graphe biparti de manière à pouvoir facilement obtenir une partition du graphe contracté final, nous allons réaliser des appariements de sommets en ce sens. Pour cela, nous allons parcourir l'ensemble des sommets *calcul* C et choisir aléatoirement un sommet c que nous allons apparier. Ainsi, nous récupérons l'ensemble $C_{adj} = \{c_1, c_2, \ldots, c_j\}$

composé des sommets non appariés et nécessitant des données commune à c, c'est-à-dire ;

$$C_{adj} = \bigcup_{d \in \mathcal{N}(c), c' \in \mathcal{N}(d) \setminus c} c'.$$

Nous apparions alors un sommet $c'' \in C_{adj}$ au sommet c de telle façon que soit vérifiée :

$$\sum_{d \in \mathcal{N}(c) \cup \mathcal{N}(c'')} \omega(d) \leq \text{Mem}[k] \quad , \forall k \in M \tag{4.6}$$

et :

$$\arg\max_{c'' \in C_{adj}} \sum_{d \in \mathcal{N}(c) \cap \mathcal{N}(c'')} \omega(d). \tag{4.7}$$

L'inégalité 4.6 nous assure que le sommet issu de la contraction est stockable sur n'importe quelle unité de calcul $k \in M$. L'expression 4.7 contraint la phase de contraction à apparier des sommets dont la quantité des données mémoire communes est maximale. Le but est fusionner ensemble des sommets qu'il serait intéressant de placer sur la même partie. Nous présentons cette méthode d'appariement dans l'algorithme 11.

La phase de contraction globale est alors réalisée en calculant itérativement les graphes bipartis grossiers jusqu'à ce que :

— soit le niveau de contraction l est dépassé ;

— soit le nombre de sommets du graphe contracté est inférieur à un seuil ciblé N ;

— soit il n'existe plus d'appariements possibles, c'est-à-dire qu'il n'existe plus deux sommets *calcul* c et c' dont la fusion induit un sommet *calcul* stockable sur n'importe quelle unité de calcul $k \in M$.

Les valeurs l et N sont des paramètres de l'algorithme, au même titre que la borne mémoire régissant la validité d'un appariement, qui peuvent être dérivés de l'architecture ciblée.

Algorithme 11 Algorithme d'appariement de sommets calcul

Fonction CouplageSommetsCalcul($B = (C, D, E)$)
  $C_{couplage} \leftarrow \emptyset$ ▷ Ensemble des sommets appariés
  $Pairs \leftarrow \emptyset$ ▷ Liste des appariements
  $M_{min} \leftarrow \min_{k \in M} \text{Mem}[k]$ ▷ Capacité minimale parmi les unités de calcul
  Tant que Size($C_{couplage}$) $\neq |C|$ **Faire**
    $c \leftarrow$ SélectionAléatoire($C \backslash C_{couplage}$)
    $C_{adj} \leftarrow \emptyset$
    Pour $c'' \in \bigcup_{d \in \mathcal{N}(c), c' \in \mathcal{N}(d) \backslash c} c'$ **Faire**
      Si $c'' \notin C_{couplage}$ et $\sum_{d \in \mathcal{N}(c) \cup \mathcal{N}(c'')} \omega(d) \leq M_{min}$ **Alors**
        $C_{adj} \leftarrow C_{adj} \cup \{c'\}$
      Fin Si
    Fin Pour
    Si $C_{adj} = \emptyset$ **Alors**
      $C_{couplage} \leftarrow C_{couplage} \cup \{c\}$
    Sinon
      $c'' \leftarrow \arg\max_{c' \in C_{adj}} \sum_{d \in \mathcal{N}(c) \cap \mathcal{N}(c')} \omega(d)$
      $C_{couplage} \leftarrow C_{couplage} \cup \{c, c''\}$
      $Pairs \leftarrow Pairs \cup (c, c'')$
    Fin Si
  Fin Tant que
  Retourner $Pairs$
Fin Fonction

4.2.2 Phase de partitionnement initial

L'objectif de la phase de partitionnement initial est de générer une partition du graphe grossier, telle que la quantité de données nécessaire sur chaque unité de calcul ne dépasse pas sa capacité mémoire. Dans ce but, nous avons développé un algorithme glouton basé sur une étape de prétraitement qui va trier les sommets *calcul* du graphe biparti. Ce type de technique est très répandu dans les algorithmes d'ordonnancement et permet notamment d'obtenir des algorithmes dont le rapport d'approximation est borné. À titre d'exemple, on peut citer l'algorithme LPT (pour *Longest Processing Time first* en anglais) présenté par R. L. Graham dans [32], où les tâches sont triées par temps de traitement décroissant. Dans notre cas, la liste triée des sommets *calcul* est parcourue par notre algorithme glouton qui affecte les sommets au fur et à mesure. Le choix de l'unité de calcul sur laquelle est affecté un sommet *calcul* dépend des affectations précédemment réalisées, et est basé sur plusieurs critères que nous développons dans le reste de la présente section.

Tri des sommets *calcul*

En s'inspirant de techniques d'ordonnancement, notre heuristique de partitionnement se base sur un tri des sommets *calcul* par besoins mémoire décroissants. Le besoin mémoire d'un sommet *calcul* c_i, que nous notons $\omega(\mathcal{N}(c_i))$, est évalué comme la somme des coûts mémoire des sommets *donnée* adjacents à c_i, c'est-à-dire $\omega(\mathcal{N}(c_i)) = \sum_{(c_i,d)\in E} \omega(d)$. L'idée de ce tri est de faciliter l'obtention d'une solution valide vis-à-vis des capacités mémoire des unités de calcul. Pour cela, nous souhaitons favoriser l'affectation de sommets *calcul* ayant besoin de données déjà distribuées.

Le principe de notre algorithme de tri est le suivant. Tout d'abord, les sommets *calcul* sont triés par besoins mémoire décroissants. Nous considérons ce tri comme partiel, puisque de nombreux sommets *calcul* peuvent nécessiter la même quantité de mémoire. Ainsi, pour améliorer le tri, nous réordonnons chaque sous-ensemble de sommets *calcul* nécessitant la même quantité de mémoire. Pour cela, nous simulons l'affectation des sommets *calcul* ayant déjà été parcourus et en déduisons leur impact sur l'affectation des sommets *calcul* ayant un besoin mémoire plus faible.

Plus précisément, soit $L = \{c_1, \ldots, c_n\}$ la liste triée partielle des sommets *calcul*. Nous traversons alors L du premier au dernier élément. Tant que nous ne rencontrons pas un sommet c_k tel que c_{k+1} vérifie $\omega(\mathcal{N}(c_k)) = \omega(\mathcal{N}(c_{k+1}))$, nous visitons les sommets *donnée* du graphe biparti $B = (C, D, E)$ qui sont connectés à un sommet *calcul* traversé. Nous utilisons la notion de visite d'un sommet *donnée* pour modéliser le fait qu'il ait déjà été distribué ou non. Lorsque nous rencontrons un ensemble $C' = \{c_k, \ldots, c_l\}$ de sommets *calcul* de même besoin mémoire, cet ensemble est localement réordonné en considérant les sommets *donnée* déjà visités. En d'autre termes, nous plaçons en premier le sommet $c_i \in C'$ qui maximise [1] $\omega(\mathcal{N}(c_i) \cap \mathcal{N}(C''))$, où $C'' = \{c_1, \ldots, c_{k-1}\}$. Puis nous visitons les sommets *donnée* connectés à c_i, ordonnons c_i comme le $k^{\text{ième}}$ sommet ainsi que c_k comme le $i^{\text{ième}}$ sommet, et réordonnons les sommets *calcul* $\{c_{k+1}, \ldots, c_l\}$. Cet algorithme de tri est présenté dans l'algorithme 12.

1. Nous étendons les notations ω et μ à des ensembles de sommets, tel que $\omega(C) = \sum_{c\in C} \omega(c)$ et $\mu(C) = \sum_{c\in C} \mu(c)$.

Algorithme 12 Tri des sommets *calcul*

Fonction TRISOMMETSCALCUL($B = (C, D, E)$) ;
  ▷ Tri (partiel) des sommets *calcul* par quantité de besoins mémoire décroissants.
  $C_{tri} \leftarrow$ TRIE(C)
  ▷ Deuxième passe complétant le tri des sommets *calcul*
  $D' \leftarrow \emptyset$ ▷ Sommets *donnée* déjà visités
  Pour $k = 1..|C|$ **Faire**
    ▷ $C_{tri}[k]$ définit le $k^{\text{ième}}$ sommet *calcul*.
    $l \leftarrow$ indice le plus élevé vérifiant $\omega(\mathcal{N}(C_{tri}[l])) = \omega(\mathcal{N}(C_{tri}[k]))$
    $C' \leftarrow \{c_k, \ldots, c_l\}$
    ▷ Sélection du sommet *calcul* $c_i \in C'$ qui maximise $\omega(\mathcal{N}(c_i) \cap D')$
    $k' \leftarrow \arg\max_{k \leq i \leq l} \omega(\mathcal{N}(c_i) \cap D')$
    ECHANGE($C_{tri}[k]$, $C_{tri}[k']$)
    $D' \leftarrow D' \cup \mathcal{N}(C_{tri}[k])$
  Fin Pour
  Retourner C_{tri}
Fin Fonction

Affectation gloutonne

Les sommets *calcul* ayant été triés comme décrit précédemment, nous parcourons alors les sommets *calcul* selon ce tri et les affectons aux unités de calcul suivant un algorithme glouton. Soit c un sommet *calcul*, nous procédons alors de la manière suivante :

1. Nous calculons l'ensemble $M_0 \subseteq M$ composé des unités de calcul encore capables de stocker les données nécessaires à c, c'est-à-dire les unités de calcul $m \in M$ qui satisfont :

$$\text{Mem}[m] \geq \sum_{d \in \mathcal{N}(c) \cup D[m]} \omega(d),$$

   où $D[m]$ est l'ensemble des sommets *donnée* déjà stockés par l'unité de calcul m.

2. Nous calculons le sous-ensemble $M_1 \subseteq M_0$ composé des unités de calcul pour lesquelles les données supplémentaires devant être stockées seront minimales, c'est-à-dire les unités de calcul $m \in M_0$ qui minimisent $\sum_{d \in \mathcal{N}(c) \setminus D[m]} \omega(d)$.

3. Nous calculons le sous-ensemble $M_2 \subseteq M_1$ composé des unités de calcul pour lesquelles la capacité mémoire disponible sera maximale, c'est-à-dire les unités de calcul $m \in M_1$ qui maximisent $\text{Mem}[m] - \sum_{d \in D[m]} \omega(d)$.

4. Nous choisissons aléatoirement une unité de calcul $m_t \in M_2$ et affectons c à m_t.

L'algorithme 13 présente l'algorithme glouton utilisé pour obtenir une partition initiale.

Algorithme 13 Affectation gloutonne des sommets calcul

Fonction GreedyInitialPartitioning($B = (C, D, E)$, $\text{Mem}[m]_{m \in M}$)
  $C_{tri} \leftarrow$ SortCVertices(B)
  ▷ $C[m]$ définit l'ensemble des sommets *calcul* affectés à l'unité de calcul $m \in M$.
  $C[m]_{m \in M} \leftarrow \emptyset$
  ▷ $D[m]$ définit l'ensemble des sommets *donnée* stockés par l'unité de calcul $m \in M$.
  $D[m]_{m \in M} \leftarrow \emptyset$
  Pour tout $c \in C_{tri}$ **Faire**
    $M_0 \leftarrow \{m \in M, \text{Mem}[m] \geq \sum_{d \in \mathcal{N}(c) \cup D[m]} \omega(d)\}$
    $M_1 \leftarrow \{\arg\min_{m \in M_0} \sum_{d \in \mathcal{N}(c) \setminus D[m]} \omega(d)\}$
    $M_2 \leftarrow \{\arg\max_{m \in M_1} \text{Mem}[m] - \sum_{d \in D[m]} \omega(d)\}$
    $m_t \leftarrow$ Random(M_2)
    ▷ Affectation de c à l'unité de calcul m_t
    $C[m_t] \leftarrow C[m_t] \cup \{c\}$
    $D[m_t] \leftarrow D[m_t] \cup \mathcal{N}(c)$
  Fin Pour
  Retourner $\{C[m]_{m \in M}\}$
Fin Fonction

4.2.3 Phase d'expansion

À ce moment du schéma multi-niveaux, les phases de contraction et de partitionnement initial se sont toutes deux focalisées sur les données mémoire, en ne considérant pas les coûts calculatoires. Par conséquent, la partition obtenue a un makespan sujet à amélioration. L'objectif de la phase de prolongation et raffinement est donc d'améliorer le makespan, tout en conservant le respect des contraintes mémoire. Pour cela, nous allons recourir à une méthode de recherche locale [1]. Les méthodes de recherche locale consistent à explorer l'espace de recherche afin d'obtenir une « meilleure » solution.

Ce type de méthode est fondé sur une relation de voisinage entre les solutions et sur une procédure exploitant ce voisinage. Cette relation de voisinage est une application qui associe à toute solution de l'espace de recherche un ensemble de solutions appelées voisines. Dans la section 4.1.3 nous avons introduit les algorithmes KL et FM et avons défini la relation de voisinage entre deux solutions. Dans KL deux solutions sont voisines si l'on peut passer de l'une à l'autre en échangeant deux sommets de parties. Dans FM deux solutions sont voisines si l'on peut passer de l'une à l'autre en déplaçant un sommet d'une partie à l'autre. Il existe un grand nombre de méthodes de recherche locale, parmi lesquelles on peut citer la recherche tabou [30], le recuit simulé [52], la recherche locale itérée [59] ou les méthodes de descentes [64].

Dans notre cas, nous définissons que deux solutions sont voisines si l'on peut passer de l'une à l'autre en changeant l'affectation d'un sommet *calcul*. De plus, contrairement aux fonctions objectif des problèmes de partitionnement de graphe et d'hypergraphe, la fonction objectif du problème de partitionnement de maillage sous contraintes mémoire n'est pas basée sur la minimisation d'une somme (voir les définitions 19 et 20 aux pages 22 et 25) mais sur la minimisation d'un maximum (voir la définition 22 à la page 31). Cette différence a un impact important lors du raffinement et tout particulièrement lors du calcul du gain induit par la migration d'un sommet d'une unité de calcul vers une autre.

Ainsi, nous commençons par décrire la manière dont est calculé le gain dans notre approche. Puis, nous décrivons l'heuristique d'optimisation locale utilisée pour raffiner itérativement la partition. Cette heuristique est basée sur le déplacement d'un sommet *calcul* d'une unité de calcul à l'autre, avec l'objectif de réduire le makespan.

Calcul du gain

Dans le cas du partitionnement de graphe, le gain induit par le déplacement d'un sommet c de $m_1 \in M$ à $m_2 \in M$ est usuellement calculé comme la différence entre la somme des coûts associés aux arêtes reliant c à des sommets de m_2 (coût extérieur) et la somme des coûts associés aux arêtes reliant c à des sommets de m_1 (coût intérieur). Le gain associé au changement de partie d'un sommet peut donc être calculé de manière locale. Dans notre cas, puisque la fonction objectif est une minimisation d'un maximum, nous ne pouvons pas calculer le gain d'une manière aussi locale. En effet, nous avons besoin de calculer le gain en considérant toutes les unités de calcul et non uniquement les unités de calcul source et cible m_1 et m_2.

Pour illustrer cette idée, considérons l'exemple de la figure 4.7 où six sommets avec différents coûts calculatoires dans l'intervalle $[\![1;5]\!]$ sont distribués sur trois unités de calcul (voir la figure 4.7(a) qui illustre les charges calculatoires initiales des unités de calcul). Maintenant, déplaçons un sommet de m_1 vers m_2. Les charges calculatoires induites par ces mouvements sont représentées respectivement en figures 4.7(b), 4.7(c) et 4.7(d). Nous pouvons alors constater que le meilleur makespan est obtenu pour la configuration 4.7(d), où l'unité de calcul la plus chargée est m_3, qui n'était pas impliquée dans le mouvement du sommet. En fait, déplacer un sommet *calcul* de m_1 à m_2 peut uniquement mener à l'un des cas suivants :

1. l'unité de calcul avec la charge de calcul la plus importante est toujours m_1 (voir la fi-

gure 4.7(b)) ;

2. l'unité de calcul avec la charge de calcul la plus importante devient la machine m_2 (voir la figure 4.7(c)) ;

3. l'unité de calcul avec la charge de calcul la plus importante devient m_3, où $m_3 \in M$ et telle que $m_3 \neq m_1$ et $m_3 \neq m_2$ (voir la figure 4.7(d)).

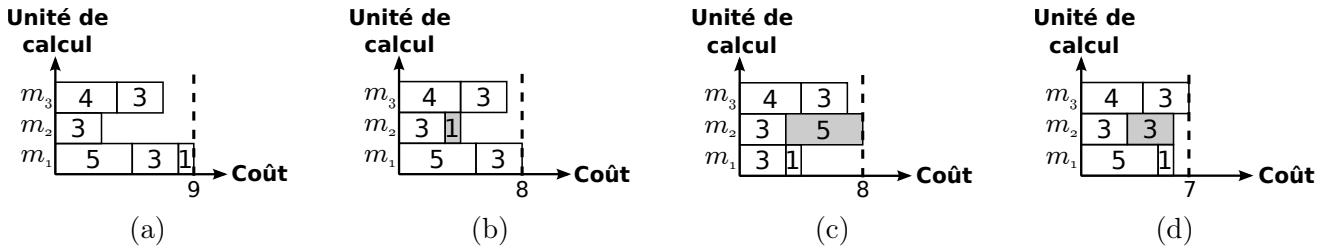


FIGURE 4.7 – *Charge calculatoire initiale de trois unités de calcul m_1, m_2 et m_3 sur (a). Les charges calculatoires induites par le déplacement d'un sommet (identifié en gris) de m_1 à m_2, sont successivement illustrées. En (b) le sommet de coût* 1 *est déplacé, en (c) le sommet de coût* 5 *est déplacé et en (d) le sommet de coût* 3 *est déplacé.*

Par conséquent, un mouvement d'une unité de calcul vers une autre peut induire un gain qui n'est pas calculable en ne considérant uniquement que les unités de calcul source et cible. Nous évaluons le gain associé au déplacement d'un sommet c, depuis une unité de calcul source m_S vers une unité de calcul cible m_T, par la fonction GAIN de l'algorithme 14.

Algorithme 14 Calcul du gain induit par le déplacement du sommet c de m_S à m_T

Requiert: $B = (C, D, E)$, $C[m]_{m \in M}$
  Fonction GAIN$((c, m_S, m_T))$
    $C_{\text{max-avant}} = max_{m \in M}\, \mu(C[m])$
    ▷ Calcul de la charge de calcul de m_S après le déplacement de c.
    $C_S = \mu(m_S) - \mu(c)$
    ▷ Calcul de la charge de calcul de m_T après le déplacement de c.
    $C_T = \mu(m_T) + \mu(c)$
    ▷ Calcul de la charge de calcul maximale parmi les autres unités de calcul.
    $C_{other} = \max_{m \in M \setminus \{m_S, m_T\}} \mu(m)$
    $C_{\text{max-après}} = \max(C_S, C_T, C_{other})$
    Retourner $C_{\text{max-après}} - C_{\text{max-avant}}$
  Fin Fonction

Heuristique d'optimisation locale

En partant de $C[m]_{m \in M}$, une partition des sommets *calcul*, et de $iter_{max}$, un nombre maximum de déplacements, nous raffinons itérativement la partition $C[m]_{m \in M}$ (voir la procédure LocalOpt de l'algorithme 15). Pour cela, nous commençons par calculer l'ensemble M_S composé des unités de calcul dont la charge calculatoire est la plus élevée. Nous nous intéressons à cet ensemble puisque calculer un déplacement depuis une unité de calcul $m \notin M_S$ ne peut pas induire de gain strictement positif. Puis, pour toute unité de calcul $m_S \in M_S$, nous évaluons l'ensemble des déplacements possibles $\mathcal{MP}$ depuis m_S vers $m_T \in M \backslash M_S$ et calculons leur gain correspondant. En effet, certains déplacements ne sont pas possibles car ils entraîneraient un non respect des contraintes mémoires. Nous sélectionnons alors le mouvement de gain maximum et l'effectuons tant que le gain est positif.

Dans le cas où le gain maximum n'est plus strictement positif, nous n'avons pas forcément abouti à une répartition optimale des charges calculatoires. En effet, le fait qu'aucun déplacement de gain strictement positif n'est possible peut être du à la distribution des données. Par conséquent, lorsqu'aucun mouvement strictement positif n'est possible depuis $m_S \in M_S$, nous effectuons aléatoirement un déplacement de gain nul. En autorisant des déplacements depuis des unités de calcul $m'_S \notin M_S$, nous souhaitons modifier la charge mémoire des unités de calcul de telle sorte que de nouveaux mouvements strictement positifs soient réalisables.

Nous illustrons ce raisonnement à l'aide de la figure 4.8, où 4.8(a) présente le maillage M_{Grid} étudié. Le coût calculatoire et le poids mémoire associés aux mailles de M_{Grid} varient selon la couleur de celles-ci : une maille m_i à fond rouge est pondérée par $\mu[m_i] = \omega[m_i] = 2$; une maille à fond blanc est pondérée par $\mu[m_i] = \omega[m_i] = 1$. Nous cherchons à partitionner ce maillage sur trois unités de calcul m_1, m_2 et m_3 telles que $\text{Mem}[m_i]_{1 \leq i \leq 3} = 14$, en considérant un voisinage par arêtes à distance 1. Une solution non optimale à ce problème est présentée en figure 4.8(b), qui illustre la charge calculatoire et la charge mémoire induites par la solution et où les unités de calcul m_1, m_2 et m_3 sont respectivement identifiées en couleur bleu, rose et vert. Même si cette solution n'est pas optimale, nous pouvons constater qu'il n'existe pas de déplacement depuis l'unité de calcul la plus chargée vers une des autres unités de calcul. En effet, tout déplacement depuis celle-ci engendrerait un dépassement de la capacité mémoire des autres unités de calcul ($M_S \cap \mathcal{MP} = \emptyset$). Il existe cependant des déplacements de gain nul qui peuvent nous permettre de converger vers une solution optimale du problème. En effet, en réalisant deux déplacements depuis m_3 vers m_1 (voir les figures 4.8(c) et 4.8(d)), nous modifions suffisamment la distribution des données pour

permettre un déplacement depuis m_2 vers m_3 puis un déplacement depuis m_1 vers m_2 (voir les figures 4.8(e) et 4.8(f)). La solution résultante est optimale pour le problème considéré et a été obtenue en réalisant une suite de trois mouvements de gain nul qui ont permis par la suite un mouvement de gain strictement positif.

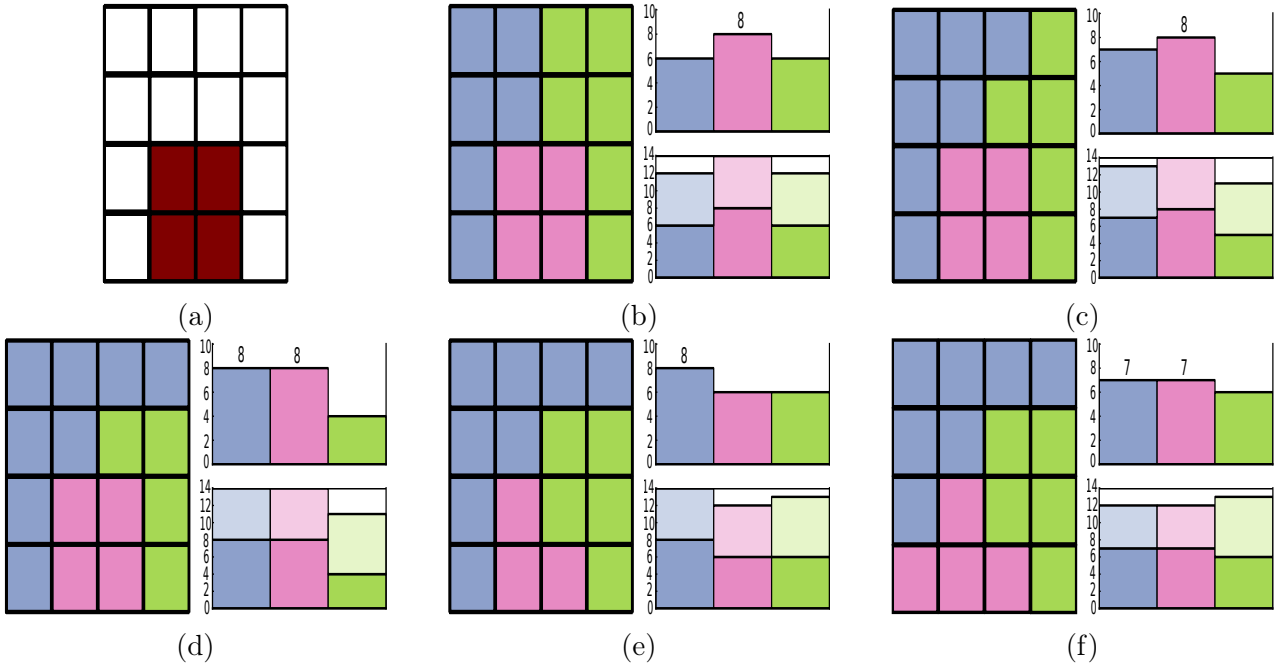


FIGURE 4.8 – Illustration d'une suite de mouvements de gain nul permettant d'obtenir la réalisation d'un mouvement de gain strictement positif. La figure (a) présente le maillage M_{Grid} considéré où les mailles en fond rouge sont pondérées par $\mu[m_i] = \omega[m_i] = 2$ et les mailles en fond blanc sont pondérées par $\mu[m_i] = \omega[m_i] = 1$. La figure (b) présente la partition initiale ; les figures (c), (d) et (e) présentent des partitions obtenues en effectuant des mouvements de gain nul ; la figure (f) présente une partition optimale, obtenue suite à un mouvement de gain strictement positif.

Si le nombre de déplacements effectués est égal à $iter_{max}$, ou bien s'il n'existe plus que des mouvements de gain négatif, nous interrompons notre algorithme. N'ayant effectué que des mouvements valides, l'optimisation locale est effectuée en garantissant le respect de la capacité mémoire des unités de calcul. L'algorithme 15 présente l'heuristique utilisée à chaque niveau lors de la phase d'expansion.

Algorithme 15 Heuristique d'optimisation locale

Requiert: $B = (C, D, E)$, $C[m]_{m \in M}$, $\text{Mem}[m]_{m \in M}$
  Procédure LocalOpt($iter_{max}$)
    $iter \leftarrow 0$
    Tant que $iter \leq iter_{max}$ **Faire**
      ▷ M_S est l'ensemble des unités de calcul dont la charge calculatoire est la plus élevée.
      $M_S \leftarrow \arg\max_{m \in M} \mu(C[m])$
      ▷ Calcul de $\mathcal{MP}_S$ l'ensemble des déplacements possibles depuis M_S.
      $\mathcal{MP}_S \leftarrow \emptyset$
      Pour tout $c \in \bigcup_{m_S \in M_S} C[m_S]$ **Faire**
        Pour tout $m_T \in M \backslash M_S$ **Faire**
          Si $\omega(\mathcal{N}(C[m_T] \cup \{c\}) \leq \text{Mem}[m_T]$ **Alors**
            $m_S \leftarrow$ Partie(c)
            $\mathcal{MP}_S \leftarrow \mathcal{MP}_S \cup \{(c, m_S, m_T)\}$
          Fin Si
        Fin Pour
      Fin Pour
      ▷ Sélection du meilleur déplacement de $\mathcal{MP}_S$.
      $(c, m_S, m_T) \leftarrow \arg\max_{move \in \mathcal{MP}_S}$ Gain($move$)
      Si Gain((c, m_S, m_T)) > 0 **Alors**
        $C[m_S] \leftarrow C[m_S] \backslash \{c\}$ ▷ On effectue le mouvement
        $C[m_T] \leftarrow C[m_T] \cup \{c\}$
        $iter \leftarrow iter + 1$
      Sinon
        ▷ Calcul de $\mathcal{MP}_{\overline{M_S}}$ l'ensemble des déplacements possibles depuis $M \backslash M_S$.
        $\mathcal{MP}_{\overline{M_S}} \leftarrow \emptyset$
        Pour tout $c \in \bigcup_{m'_S \in \{M \backslash M_S\}} C[m'_S]$ **Faire**
          Pour tout $m_T \in M \backslash M_S$ **Faire**
            Si $\omega(\mathcal{N}(C[m_T] \cup \{c\}) \leq \text{Mem}[m_T]$ **Alors**
              $m'_S \leftarrow$ Part(c)
              $\mathcal{MP}_{\overline{M_S}} \leftarrow \mathcal{MP}_{\overline{M_S}} \cup \{(c, m'_S, m_T)\}$
            Fin Si
          Fin Pour
        Fin Pour
        ▷ Sélection d'un déplacement de $\mathcal{MP}_{\overline{M_S}}$ dont le gain est nul.
        $(c, m'_S, m_T) \leftarrow \arg\max_{move \in \mathcal{MP}_{\overline{M_S}}}$ Gain($move$)
        Si Gain((c, m'_S, m_T)) $== 0$ **Alors**
          $C[m'_S] \leftarrow C[m'_S] \backslash \{c\}$
          $C[m_T] \leftarrow C[m_T] \cup \{c\}$
          $iter \leftarrow iter + 1$
        Sinon
          Retourner $C[m]_{m \in M}$
        Fin Si
      Fin Si
    Fin Tant que
  Fin Procédure

En résumé, notre approche multi-niveaux consiste tout d'abord à obtenir une partition respectant les contraintes du problème de partitionnement de maillage sous contraintes mémoire, puis à améliorer la qualité de la solution obtenue en explorant l'espace de recherche. À cet effet, nous avons orienté la phase contraction à fusionner des ensembles de sommets *calcul* qu'il serait intéressant de placer sur la même partie. Pour obtenir une partition initiale du graphe grossier, nous avons développé un algorithme glouton basé sur une étape de prétraitement des sommets *calcul*. Enfin, la partition initiale ayant été obtenue sans prendre en compte les coûts de calcul, nous améliorons sa qualité lors de la phase d'expansion. Pour cela, nous avons introduit la notion de gain, associée au déplacement d'un sommet d'une partie à l'autre, dans notre problème de partitionnement.

Notons que cette approche a le défaut de pouvoir retourner des solutions valides de mauvaise qualité. En effet, il est par exemple possible que la solution initiale ne soit pas sujette à des améliorations. Dans la section suivante, nous allons étudier le taux de solutions valides retournées par notre approche ainsi que la qualité des solutions obtenues sur plusieurs expériences.

4.3 Résultats expérimentaux

Dans cette section, nous utilisons notre approche multi-niveaux sur différents problèmes et commentons les résultats obtenus par rapport aux résultats de l'approche classique, retournés par l'outil de partitionnement Scotch. Dans la section 2.4, nous avons pu constater que les solutions optimales aux problèmes de partitionnement de graphe et d'hypergraphe peuvent induire une distribution des données dépassant les capacités mémoire des unités de calcul. Ceci ayant été mis en évidence dans le cas de répartition mémoire et calculatoire hétérogène (voir les sections 2.4.1 et 2.4.2), nous nous intéresserons particulièrement à ce type d'entrées. Les maillages étudiés sont présentés en table 4.2. Cette table fournit le nom associé au maillage, sa dimension, le type et le nombre de cellules qui le composent, et la répartition calculatoire de type *pic* qui lui est associée. Notons que, comme en section 2.4, nous considérons que le poids mémoire d'une maille est égal à la racine de son coût calculatoire. Nous étudierons aussi une répartition *unitaire* et une répartition aléatoire dans le cas du maillage $\mathcal{M}_{quad}$. Celles-ci ne figurent pas dans la table 4.2 mais la répartition *unitaire* est intuitive et la répartition *aléatoire* a été introduite en figure 2.17.

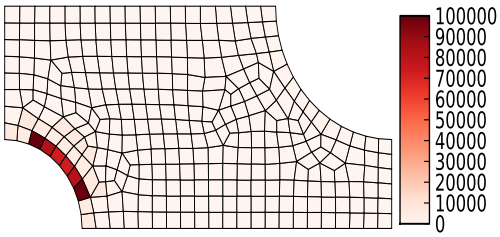
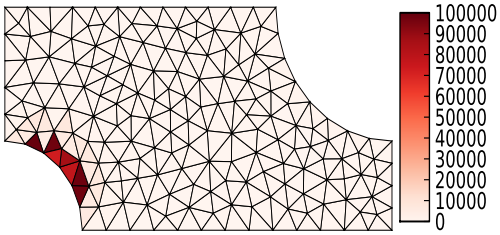
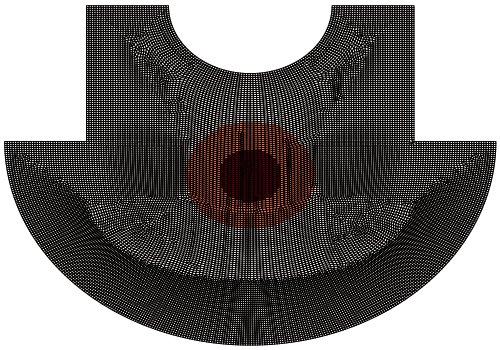
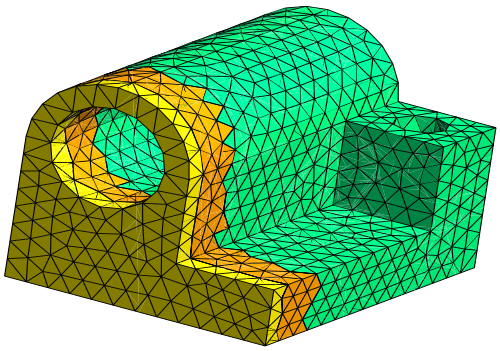
Nom	Dimension	Type de cellule	Nombre de cellule	Répartition
$\mathcal{M}_{quad}$	$2D$	Quadrilatère	312	
$\mathcal{M}_{tri}$	$2D$	Triangle	310	
$\mathcal{M}_{mush}$	$2D$	Quadrilatère	22 800	
$\mathcal{M}_{cad}$	$3D$	Tétraèdre	10 788	

TABLE 4.2 – Présentation des maillages étudiés lors des expériences. Chaque maillage est identifié par un nom et défini par sa dimension, son type et nombre de cellules, ainsi que la répartition calculatoire de type *pic* qui lui est associée. Dans le cas du maillage $\mathcal{M}_{mush}$, les mailles en blanc ont un poids calculatoire unitaire, les mailles en rose ont un poids calculatoire de 100 et les mailles en rouge ont un poids calculatoire de 22500. Dans le cas du maillage $\mathcal{M}_{cad}$, les mailles en vert ont un poids calculatoire unitaire, les mailles en orange ont un poids calculatoire de 100 et les mailles en jaune ont un poids de 2500.

Les deux premières expériences que nous allons réaliser portent sur le maillage $\mathcal{M}_{quad}$, en considérant un voisinage par arête à distance 1. Nous distinguons ces deux expériences selon la répartition des poids mémoires et des coûts calculatoires associés au maillage. La première est une répartition *unitaire* (voir section 2.4.3) et la seconde est une répartition *aléatoire* (voir section 2.4.4). Les expériences suivantes sont réalisées en considérant chaque maillage. Pour chacune de ces expériences, nous étudions une répartition de type *pic* et nous considérons successivement le cas d'un voisinage par arête à distance 1 et 2. Pour évaluer la qualité des solutions de SCOTCH et de notre algorithme multi-niveaux, nous allons utiliser la borne inférieur LB (voir équation 2.11 page 39).

Pour chaque expérience, nous procédons de la manière suivante. Nous utilisons l'outil SCOTCH et notre algorithme multi-niveaux de manière non-déterministe et générons cent solutions par expérience. Nous paramétrons SCOTCH afin d'autoriser un déséquilibre maximal de 5% et conservons les autres paramètres standard (par exemple, la taille cible du graphe contracté est de 120 sommets). Notre algorithme est quant à lui paramétré par :

— un nombre de niveaux fixé à 5, ce qui permet d'obtenir un graphe contracté composé d'un millier de sommets pour l'instance la plus grande ;
— le nombre maximum de mouvement de raffinement par niveau est borné à 100, afin d'éviter de boucler sur des solutions parcourues ;

La quantité de données maximale induite lors de la contraction de deux sommets *calcul* pouvant perturber l'efficacité de l'algorithme glouton, nous étudions les cas où celle-ci est bornée par la capacité des unités de calcul ainsi que la moitié de cette valeur.

Lors de notre utilisation de l'outil SCOTCH, nous équilibrons successivement la charge de calcul puis la charge mémoire. Afin de distinguer les deux ensembles de solutions générées, nous notons S_{scotch}^{cout} l'ensemble des solutions obtenues en équilibrant la charge de calcul et S_{scotch}^{mem} l'ensemble des solutions obtenues en équilibrant la charge mémoire. De même, nous distinguons les solutions retournées par notre algorithme multi-niveaux selon la borne maximale sur la quantité de données, induite lors de la contraction. Nous notons S_{multi} l'ensemble des solutions obtenues avec une borne fixée à la capacité des unités de calcul et S_{multi}^{half} l'ensemble des solutions obtenues avec une borne fixée à la moitié de la capacité des unités de calcul.

Expériences à répartition *unitaire* et *aléatoire*

Pour ces expériences, nous souhaitons partitionner le maillage sur trois unités de calcul. Nous fixons la capacité des unités de calcul aux valeurs M_{20p} obtenues lors de la résolution exacte de ces problèmes aux sections 2.4.3 et 2.4.4. Nous rappelons que $M_{20p} = 1.2 \times M_{min}$, où M_{min} est la capacité minimale, homogène à chaque unité de calcul, nécessaire à l'existence d'une solution.

Nous avons alors généré les ensembles de solutions S^{cout}_{scotch}, S^{mem}_{scotch}, S_{multi} et S^{half}_{multi}. Puis, nous avons restreint chacun de ces ensembles aux solutions valides qui le composent. Pour cela, nous avons évalué la quantité de données maximale associée à une unité de calcul de chaque solution. Nous avons ensuite calculé le makespan de chaque solution valide. Afin d'évaluer la qualité d'une solution, nous calculons le rapport entre le makespan d'une solution et la borne inférieure LB. Ainsi, plus ce rapport est proche de la valeur un, plus le makespan de la solution est proche de la borne inférieure.

La distribution des valeurs ainsi obtenues est illustrée en table 4.3, où les expériences 1 et 2 concernent respectivement les répartitions *unitaire* et *aléatoire*. Sur ces figures, le makespan normalisé correspond à la valeur du makespan divisé par LB. La longueur d'une barre horizontale d'ordonnée y représente le nombre de fois qu'une heuristique a retourné une solution dont la valeur du makespan divisée par LB est y. Enfin, pour chaque ensemble de solutions, nous avons ajouté une graduation permettant d'évaluer visuellement le nombre de solutions à une valeur de makespan donnée.

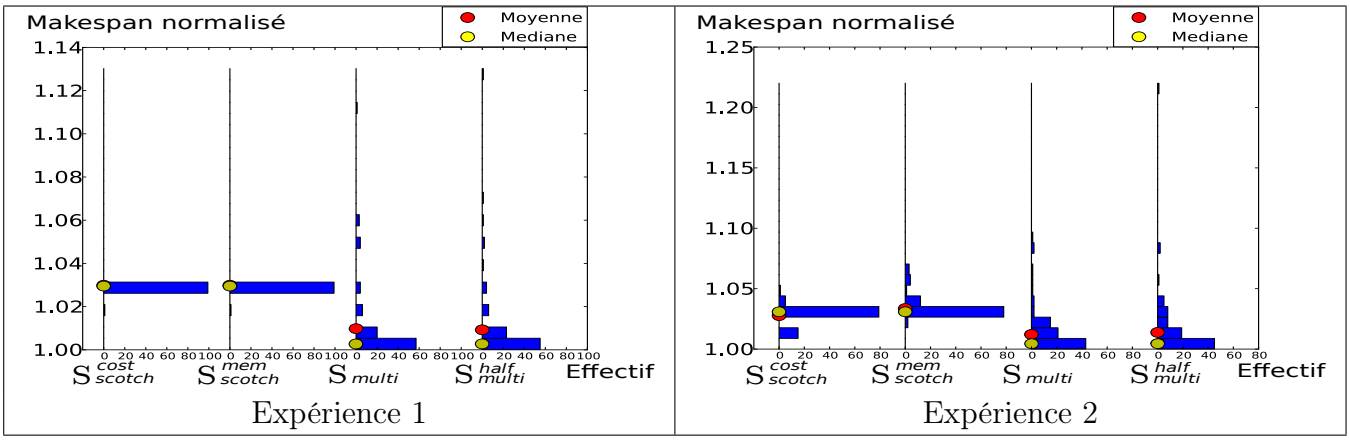


TABLE 4.3 – Évaluation du rapport entre le makespan des solutions de S^{cout}_{scotch}, S^{mem}_{scotch}, S_{multi} et S^{half}_{multi}, par la borne inférieure LB. Les expériences 1 et 2 présentent respectivement cette évaluation pour la répartition *unitaire* et la répartition *aléatoire*. Sur chaque figure, la moyenne et la médiane des rapports sont respectivement illustrées par un cercle rouge et un cercle jaune.

Dans le cas de la répartition *unitaire*, nous pouvons constater que le makespan induit par les solutions de SCOTCH est groupé. À l'inverse, notre algorithme retourne une plus grande dispersion de makespan dont le maximum est de mauvaise qualité. Cependant, nous pouvons constater que notre algorithme est en moyenne meilleur que SCOTCH. Le makespan médian est d'ailleurs égal à la borne inférieure LB.

Dans le cas de la répartition *aléatoire*, la dispersion du makespan de chaque approche est étalée. Nous pouvons à nouveau constater que notre algorithme est en moyenne meilleur que SCOTCH, et que le makespan maximal retourné est de mauvaise qualité. La makespan médian de l'ensemble S_{multi}^{half}est égal à LB et, bien que ce ne soit pas aussi le cas de l'ensemble S_{multi}, celui-ci a un makespan médian proche de LB.

Expériences à répartition *pic*

Comme nous avons pu le constater aux sections 2.4.1 et 2.4.2, les solutions optimales aux problèmes de partitionnement de graphe et d'hypergraphe, pour des instances à répartition *pic*, peuvent induire une distribution des données non valide. Par conséquent, nous allons nous intéresser à ce type de répartitions pour les prochaines expériences. Pour cela, nous allons successivement étudier les maillages $\mathcal{M}_{quad}$, $\mathcal{M}_{tri}$, $\mathcal{M}_{mush}$, et $\mathcal{M}_{cad}$. Pour chaque maillage étudié, nous allons considérer successivement un voisinage par arête à distance 1 et 2.

Lors des expériences sur les maillages $\mathcal{M}_{quad}$ et $\mathcal{M}_{tri}$, nous souhaitons partitionner le maillage sur trois unités de calcul. Nous fixons la capacité des unités de calcul aux valeurs M_{20p} obtenues lors de la résolution exacte de ces problèmes (voir sections 2.4.1 et 2.4.2 pour le maillage $\mathcal{M}_{quad}$). Lors des expériences sur les maillages $\mathcal{M}_{mush}$ et $\mathcal{M}_{cad}$, nous souhaitons partitionner le maillage sur trente-deux unités de calcul. Les tailles de ces problèmes étant trop grandes, nous ne pouvons pas utiliser l'outil CPLEX pour déterminer la capacité des unités de calcul. Nous bornons donc de manière arbitraire la capacité des unités de calcul à 8 000 chacune, dans le cas du maillage $\mathcal{M}_{mush}$, et à 4 000 chacune, dans le cas du maillage $\mathcal{M}_{cad}$.

Après avoir généré les ensembles de solutions S_{scotch}^{cout}, S_{scotch}^{mem}, S_{multi} et S_{multi}^{half} associés à chaque expérience, nous les avons restreints aux solutions valides qui les composent. Puis, nous avons évalué le makespan de chaque solution et avons calculé le rapport entre ce makespan et la borne inférieure LB. La distribution des valeurs obtenues est illustrée en table 4.4, où chaque expérience est identifiée grâce au maillage et à la relation de voisinage considérée.

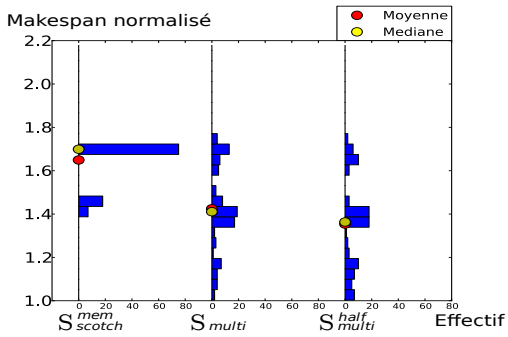
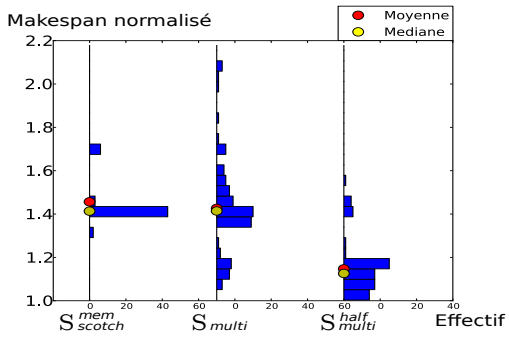
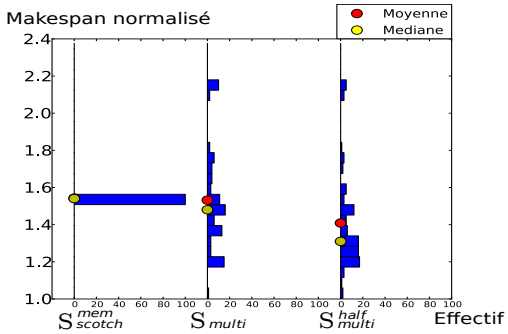
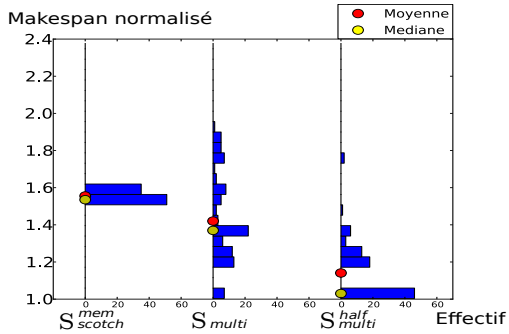
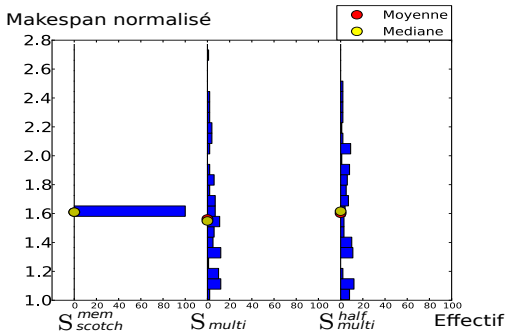
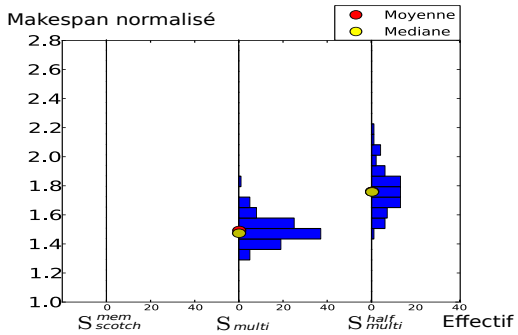
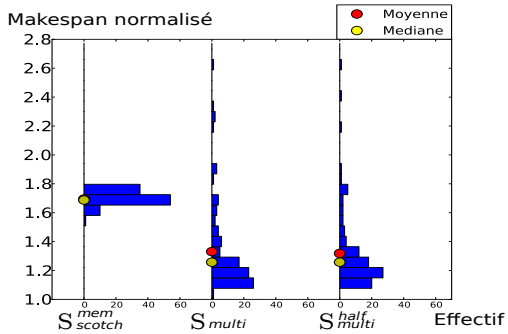
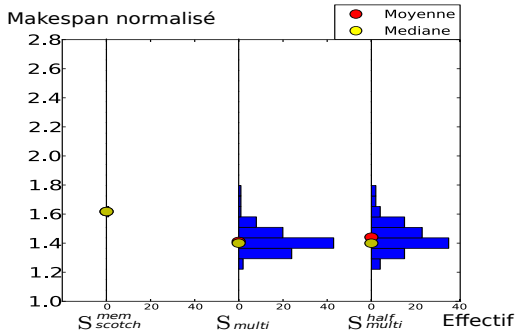
Maillage	Voisinage par arête à distance 1	Voisinage par arête à distance 2
$\mathcal{M}_{quad}$	 Expérience 3	 Expérience 4
$\mathcal{M}_{tri}$	 Expérience 5	 Expérience 6
$\mathcal{M}_{mush}$	 Expérience 7	 Expérience 8
$\mathcal{M}_{cad}$	 Expérience 9	 Expérience 10

TABLE 4.4 – Évaluation du rapport entre le makespan des solutions de S_{scotch}^{mem}, S_{multi}et S_{multi}^{half}, et la borne inférieure LB. Chaque case du tableau correspond à une expérience qui peut être identifiée selon le maillage et la relation de voisinage considérée.

Intéressons-nous maintenant aux résultats des expériences. Pour commencer, nous pouvons remarquer l'absence de l'ensemble S^{cout}_{scotch} sur chaque figure de la table 4.4. Ceci est dû au fait qu'aucunes des solutions de S^{cout}_{scotch} n'est valide vis-à-vis des contraintes mémoire. Nous pouvons d'ailleurs remarquer que l'ensemble S^{mem}_{scotch} peut lui aussi ne pas être composé de solutions valides. C'est le cas de l'expérience considérant $\mathcal{M}_{mush}$ et un voisinage par arête à distance 2. Notons d'ailleurs que dans l'expérience considérant $\mathcal{M}_{cad}$ et un voisinage par arête à distance 2, l'ensemble S^{mem}_{scotch} est composé d'une unique solution valide.

Afin d'illustrer ces observations, nous considérons l'expérience sur le maillage $\mathcal{M}_{mush}$ avec un voisinage par arête à distance 2, et présentons en figure 4.9 la distribution des données des solutions induisant respectivement la plus petite et la plus grande charge mémoire sur une unité de calcul. Les figures 4.9(a) et 4.9(b) (respectivement 4.9(c) et 4.9(d)) concernent l'ensemble S^{cout}_{scotch} (respectivement S^{mem}_{scotch}). Sur chaque figure, la capacité des unités de calcul est illustrée par une droite en tirets. Ainsi, si la quantité de donnée associée à une unité de calcul dépasse cette ligne, la solution n'est pas valide. Nous pouvons constater, et c'est le cas pour toutes les expériences à répartition de type *pic*, que les solutions de S^{cout}_{scotch} ne sont pas valides, et ceci même en ignorant l'impact mémoire des mailles fantômes. De plus, contrairement aux solutions de S^{cout}_{scotch}, c'est la prise en compte de la charge mémoire induite par les mailles fantômes qui peut rendre les solutions de S^{mem}_{scotch} non valides. Nous pouvons d'ailleurs constater que la charge mémoire résultante des mailles fantômes peut dépasser la charge mémoire induite par les mailles que traite une unité de calcul.

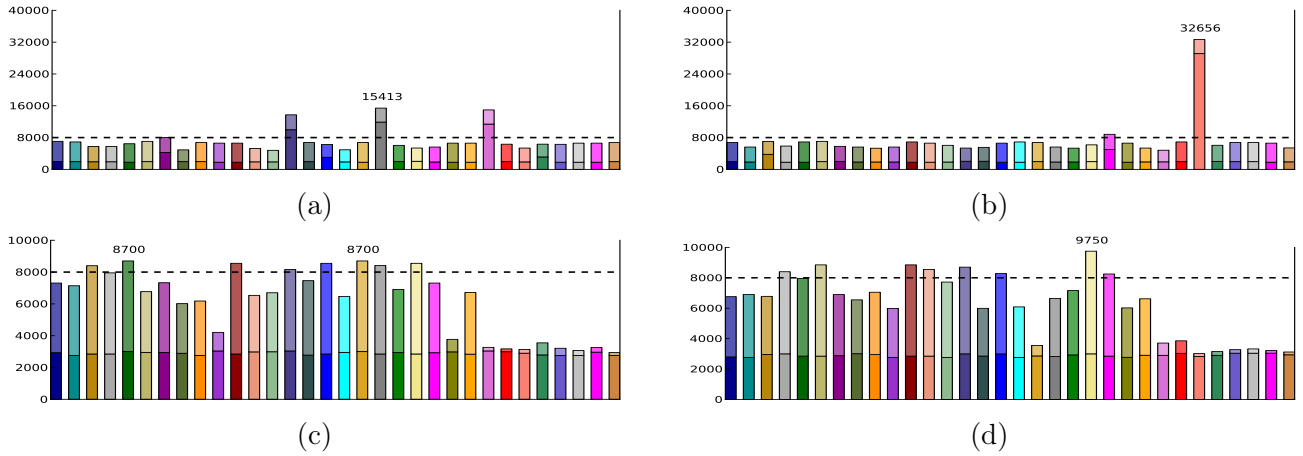


FIGURE 4.9 – Évaluation de la distribution des données des solutions de S^{cout}_{scotch} et S^{mem}_{scotch}, pour l'expérience considérant $\mathcal{M}_{mush}$ et un voisinage par arête à distance 1. La figure (a) (respectivement (b)) présente la distribution mémoire de la solution de charge mémoire minimale (respectivement maximale) de S^{cout}_{scotch}. La figure (c) (respectivement (d)) présente la distribution mémoire de la solution de charge mémoire minimale (respectivement maximale) de S^{mem}_{scotch}.

Ensuite, nous pouvons remarquer que, comme dans le cas des répartitions *unitaire* et *aléatoire*, notre algorithme retourne des solutions valides dont la dispersion est plus importante que celle de SCOTCH. Nous pouvons constater que la dispersion du makespan des solutions de S_{multi} est similaire à celle des solutions de S_{multi}^{half} dans le cas d'un voisinage à distance 1. Cependant, lorsque nous considérons un voisinage à distance 2, les différences entre les dispersions s'accentuent.

En termes de meilleure qualité de solution, nous pouvons constater que, dans le cas du voisinage par arête à distance 1, notre algorithme permet d'obtenir des solutions à moins d'un écart de 10% de LB. Ce constat peut d'ailleurs être étendu au voisinage par arête à distance 2 pour les maillages $\mathcal{M}_{quad}$, et $\mathcal{M}_{tri}$. Bien que ce ne soit pas le cas dans le cas des maillages $\mathcal{M}_{mush}$ et $\mathcal{M}_{cad}$, l'incapacité de notre algorithme à retourner des solutions à moins d'un écart de 10% de LB peut s'expliquer par la difficulté d'obtenir de telles solutions. De plus, il est aussi possible qu'il n'existe pas de solution valide à moins d'un écart de 10% de LB.

Enfin, nous pouvons remarquer qu'en moyenne, notre algorithme retourne des solutions valides dont le makespan est inférieur à celui des solutions valides de SCOTCH. De plus, dans le cas de notre algorithme, le makespan médian est toujours proche ou inférieur au makespan moyen.

Bilan des expériences

Nous avons réalisé des expériences sur plusieurs types d'instances en faisant varier à la fois le maillage, la relation de voisinage entre les mailles, et le type de répartition des coûts calculatoires et des poids mémoires. La liste des expériences réalisées est présentée dans la table 4.5, où chaque expérience est identifiée par son maillage, sa relation de voisinage et son type de répartition. Les résultats obtenus pour l'ensemble des expériences sont ensuite résumés dans la table 4.6.

Expérience	Maillage	Voisinage	Répartition
1	$\mathcal{M}_{quad}$	Arête à distance 1	*unitaire*
2	$\mathcal{M}_{quad}$	Arête à distance 1	*random*
3	$\mathcal{M}_{quad}$	Arête à distance 1	*pic*
4	$\mathcal{M}_{quad}$	Arête à distance 2	*pic*
5	$\mathcal{M}_{tri}$	Arête à distance 1	*pic*
6	$\mathcal{M}_{tri}$	Arête à distance 2	*pic*
7	$\mathcal{M}_{mush,.}$	Arête à distance 1	*pic*
8	$\mathcal{M}_{mush,.}$	Arête à distance 2	*pic*
9	$\mathcal{M}_{cad_liu,.}$	Arête à distance 1	*pic*
10	$\mathcal{M}_{cad_liu,.}$	Arête à distance 2	*pic*

TABLE 4.5 – Liste des expériences réalisées.

Exp.	Opt	S^{cout}_{scotch}				S^{mem}_{scotch}				S_{multi}				S^{half}_{multi}			
		$\|.\|$	min	med	moy	$\|.\|$	min	med	moy	$\|.\|$	min	med	moy	$\|.\|$	min	med	moy
1	104	100	106	107	107	100	106	107	107	95	104	104	105	94	104	104	105
2	4 169	100	4 204	4 299	4 287	100	4 266	4 277	4 297	89	4 169	4 192	4 222	89	4 169	4 189	4 227
3	213 502	0	-	-	-	100	296 936	366 459	351 469	95	221 280	297 710	304 076	98	218 127	294 540	289 012
4	213 513	0	-	-	-	54	277 656	305 946	311 128	97	225 404	301 343	304 452	85	217 466	240 962	244 858
5	250 609	0	-	-	-	100	381 815	388 421	386 902	99	252 521	372 538	384 039	100	260 612	328 288	353 085
6	250 618	0	-	-	-	100	386 589	387 923	389 659	99	251 089	342 592	356 071	89	250 754	261 767	286 011
7	279 645	0	-	-	-	100	450 000	450 400	450 312	100	293 899	427 533	437 046	100	294 030	450 000	447 907
8	279 645	0	-	-	-	0	-	-	-	100	360 051	407 003	413 759	67	405 200	496 300	496 253
9	45 952	0	-	-	-	100	71 000	78 200	77 808	100	48 900	56 209	61 141	100	50 155	56 239	60 552
10	45 952	0	-	-	-	1	75 500	75 500	75 500	100	57 009	64 217	64 849	100	57 929	65 451	66 241

TABLE 4.6 – Évaluation des ensembles de solutions S^{cout}_{scotch}, S^{mem}_{scotch}, S_{multi} et S^{half}_{multi}. Pour chaque expérience, nous rappelons la valeur minimale du makespan, c'est-à-dire C^*_{20p} ou LB. Puis, pour chaque ensemble de solutions, nous évaluons son nombre de solutions valides ainsi que son makespan minimal, médian et moyen. Les cellules en fond vert correspondent aux valeurs les plus faibles du makespan minimal, médian et moyen parmi les ensembles S^{cout}_{scotch}, S^{mem}_{scotch}, S_{multi} et S^{half}_{multi}, restreints à leurs solutions valides.

Au vu des résultats obtenus, nous remarquons que notre approche permet d'obtenir des solutions valides pour notre problème, et cela même sur des instances où un outil classique de partitionnement tel que SCOTCH est mis en défaut. Dans le cas où SCOTCH permet d'obtenir des solutions valides, notre algorithme retourne une plus grande dispersion de makespan dont le maximum est de mauvaise qualité. Cependant, nous pouvons constater qu'en moyenne le makespan des solutions retourné par notre algorithme est meilleur que celui des solutions de SCOTCH. Ce résultat est d'autant plus intéressant que, pour l'ensemble des expériences, le makespan médian de notre algorithme est inférieur ou proche du makespan moyen. En pratique, nous pouvons donc nous attendre à obtenir des solutions dont le makespan est proche ou inférieur au makespan moyen. De plus, il est intéressant de constater que, pour les expériences à répartitions *unitaire* et *aléatoire*, notre algorithme retourne un makespan médian égal à la borne inférieure LB.

En conclusion, les expériences réalisées nous ont permis de valider notre approche multi-niveaux. En effet, celle-ci nous a permis d'obtenir des solutions valides pour notre problème, dans le cas d'instances ou un outil de partitionnement classique tel que SCOTCH est mis en défaut. De plus, bien que les coûts de calcul ne soient pris en compte que pendant la phase d'expansion, notre approche nous a permis d'obtenir des résultats de bonne qualité.

Conclusion et perspectives

Conclusion

L'objectif de cette thèse était d'apporter une solution pour le partitionnement de maillage pour les simulations utilisant des mailles fantômes. Pour cela, nous avons défini et formalisé le problème de partitionnement de maillage sous contrainte mémoire.

Dans le chapitre 2, nous avons montré, à l'aide de la programmation linéaire en nombres entiers, que ce problème est intrinsèquement différent des modèles usuels de partitionnement de graphe ou d'hypergraphe avec minimisation des communications. À l'aide de résolutions exactes sur des problèmes de petites tailles, nous avons illustré la nécessité de prendre explicitement en compte les mailles fantômes comme nous le proposons. Les résultats obtenus corroborent le fait que les approches classiques de partitionnement ne sont pas adaptées aux respect des contraintes mémoire, ce qui les rend inadaptées dans nos cas d'utilisation.

Nous avons ensuite proposé deux axes de résolution du problème de partitionnement de maillage sous contrainte mémoire :

1. la résolution avec des algorithmes avec garantie de performance ;
2. l'utilisation d'heuristiques pour une résolution plus rapide mais sans garantie de performance.

Concernant les algorithmes avec garantie de performance, nous nous sommes appliqués à concevoir un algorithme de programmation dynamique inspiré par des algorithmes d'ordonnancement. Nous avons ainsi proposé un algorithme FPT selon la largeur linéaire du graphe dual associé au maillage, dans le cas où le nombre m de machines est fixé. Cet algorithme permet de déterminer s'il n'existe pas de solution au problème de partitionnement sous contrainte mémoire et, dans le cas où il en existe une, de retourner une solution à un facteur au plus $1 + \varepsilon$ du makespan optimal.

Nous avons également étudié des méthodes basées sur la relaxation d'un programme linéaire en nombres entiers. Nous avons obtenu un algorithme pour un problème voisin du problème de partitionnement de maillage sous contraintes mémoire, où la métrique minimisée est la somme totale des coûts de calcul. Cet algorithme permet de déterminer s'il n'existe pas de solution à ce problème voisin et, dans le cas où il en existe une, de retourner une solution de coût optimal.

Ces deux résultats montrent qu'il est possible d'obtenir des solutions de bonne qualité en temps polynomial, mais leurs complexités calculatoires sont trop élevées pour une utilisation directe sur des maillages et des machines de grande taille.

Pour les cas pratiques, nous avons, dans le chapitre 3, conçu et développé une méthode multi-niveaux pour le partitionnement de maillage sous contraintes mémoire. Nous nous sommes inspirés des algorithmes utilisés dans Scotch ou MeTiS, que nous avons adaptés aux spécificités de notre problème :

1. l'utilisation d'un graphe biparti permettant de modéliser les calculs, les données et les relations les liant ;

2. l'optimisation d'une métrique minimisant un maximum au lieu de la minimisation d'une somme.

Concernant ce dernier point, les phases de contraction et de partitionnement initial que nous proposons ont pour but d'obtenir une solution valide, sans prise en compte du coût calculatoire, alors que la phase d'expansion est destinée à améliorer le makespan de la solution valide.

Des expérimentations sur des maillages de nature différente nous ont permis de mettre en évidence la pertinence de la méthode. En particulier, elle nous a permis d'obtenir des solutions valides et de bonne qualité, pour des problèmes où les méthodes classique, comme l'outil Scotch, ne retournent aucune ou peu de solutions valides.

Ces premiers résultats montrent qu'il est possible d'obtenir des partitionnements de bonne qualité, même dans des contextes où la mémoire est fortement contrainte. Cela est particulièrement encourageant pour les applications qui seront amenées à être exécutées sur des machines où la quantité de mémoire par unité de calcul est de plus en plus faible.

Perspectives

Les algorithmes présentés dans cette thèse constituent une première réponse pour le partitionnement de maillage sous contrainte mémoire. Ils sont certainement améliorables, et nous présentons quelques pistes d'amélioration qui pourraient être explorées à court terme.

Parmi les perspectives immédiates, s'inscrivant dans la poursuite à court terme de nos travaux, nous pouvons citer le besoin de valider notre approche multi-niveaux sur des instances de tailles plus importantes, sur des unités de calcul hétérogènes, et sur un plus grand nombre d'unités de calcul.

Concernant notre approche multi-niveaux, il serait judicieux de réaliser une étude théorique et expérimentale sur les méthodes d'optimisation locale pouvant être utilisées lors de la phase d'expansion. L'intérêt d'une telle étude est de pouvoir déterminer des méthodes d'optimisation locale plus performantes pour le cas d'une métrique de minimisation d'un maximum. De plus, il serait intéressant d'obtenir une alternative à notre algorithme glouton utilisé pendant la phase de partitionnement initial. L'une des alternatives possibles est l'utilisation d'algorithmes avec garantie de performance.

En ce qui concerne les algorithmes à garantie de performance, il serait intéressant d'obtenir des algorithmes approchés FPT selon des paramètres de graphe plus génériques que la largeur linéaire. Par exemple, la largeur arborescente locale nous permettrait de traiter les graphes planaires, puisque ceux-ci ont une largeur arborescente locale faible. Un autre axe pouvant être profitable serait l'obtention d'autres algorithmes approchés pour le problème voisin de PMCM. En effet, les solutions de ce problème étant valides, de tels algorithmes pourraient être utilisés dans la phase de partitionnement initial de notre approche multi-niveaux.

Une autre piste d'amélioration concerne le repartitionnement. Par exemple, être capable de repartitionner une solution non valide obtenue rapidement par un outil classique de partitionnement. Ou encore, repartitionner une solution valide dont la charge de calcul est très déséquilibrée, afin de minimiser le makespan. Le contexte dans lequel les codes de calcul s'exécutent étant parallèle, il serait intéressant de pouvoir réaliser ce repartitionnement en parallèle.

Enfin, une perspective à long terme serait l'ajout à notre problème de la prise en compte des coûts de communications et de l'architecture matérielle.

Bibliographie

[1] E. Aarts et J. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, Inc., 1997.

[2] C. J. Alpert et A. B. Kahng. Recent directions in netlist partitioning : a survey. *Integration, the VLSI journal*, 19(1) :1–81, 1995.

[3] L. Babel, H. Kellerer, et V. Kotov. The k-partitioning problem. *Mathematical Methods of Operations Research*, 47(1) :59 – 82, 1998.

[4] S. Barna et S. Aravind. A new approximation technique for resource-allocation problems. In *Proc. Innovations in Computer Science (ICS), 342-–357*, 2010.

[5] S. T. Barnard et H. D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency : Practice and Experience*, 6(2) :101–117, 1994.

[6] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[7] C.-E. Bichot et P. Siarry. *Partitionnement de graphe : Optimisation et applications*. Traité IC2, Série Informatique et système d'information. Hermes Science Publications, Lavoisier, 2010.

[8] H. L. Bodlaender et T. Kloks. Better algorithms for the pathwidth and treewidth of graphs. In J. L. Albert, B. Monien, and M. R. Artalejo, editors, *Automata, Languages and Programming*, volume 510 of *Lecture Notes in Computer Science*, pages 544 – 555. Springer, 1991.

[9] E. G. Boman, K. D. Devine, L. A. Fisk, R. Heaphy, B. Hendrickson, V. Leung, C. Vaughan, U. Catalyurek, D. Bozdag, et W. Mitchell. Zoltan home page. `http://www.cs.sandia.gov/Zoltan`, 1999.

[10] T. Bui, C. Heigham, C. Jones, et T. Leighton. Improving the performance of the Kernighan-Lin and simulated annealing graph bisection algorithms. In *DAC '89 : Proceedings of the 26th*

ACM/IEEE conference on Design automation, pages 775–778, New York, NY, USA, 1989. ACM Press.

[11] Ü. V. Çatalyürek et C. Aykanat. Hypergraph model for mapping repeated sparse matrix-vector product computations onto multicomputers. *Proc. International Conference on High Performance Computing*, 1995.

[12] Ü. V. Çatalyürek et C. Aykanat. Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans Parallel Distrib. Syst.*, 10(7) :673–693, 1999.

[13] B. Chen, C. Potts, et G. Woeginger. *A review of machine scheduling : Complexity, algorithms and approximability*, pages 1493 – 1641. Springer US, 1999.

[14] C. Chevalier. *Conception et mise en oeuvre d'outils efficaces pour le partitionnement et la distribution parallèles de problème numériques de très grande taille.* Thèse de doctorat, Université Bordeaux I, 2007.

[15] C. Chevalier, G. Grospellier, F. Ledoux, et J. Weill. Load balancing for mesh based multi-physics simulations in the Arcane framework. In *Proceedings of the Eighth International Conference on Engineering Computational Technology*, 2012.

[16] C. Chevalier and F. Pellegrini. PT-Scotch : A tool for efficient parallel graph ordering. *Parallel Comput.*, 34(6-8) : 318–331, 2008.

[17] C. Chevalier and I. Safro. Comparison of coarsening schemes for multilevel graph partitioning. *Learning and Intelligent Optimization*, pages 191–205, 2009.

[18] I. I. CPLEX. V12.1 : User's manual for Cplex. *International Business Machines Corporation*, 46(53) :157, 2009.

[19] G. Dantzig. *Linear Programming and Extensions*, Princeton University Press, 1963.

[20] K. Devine, E. Boman, R. Heaphy, B. Hendrickson, et C. Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 2002.

[21] W. E. Donath et A. J. Hoffman. Algorithms for partitioning of graphs and computer logic based on eigenvectors of connection matrices. *IBM Technical Disclosure Bulletin*, 15 :938–944, 1972.

[22] W. E. Donath et A. J. Hoffman. Lower bounds for the partitioning of graphs. *IBM Journal of Research and Development*, 17(5) :420–425, 1973.

[23] V. Duraira et P. Kalla. Exploiting hypergraph partitioning for efficient boolean satisfiability. In *High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International*, pages 141–146, 2004.

[24] C. Evrendilek. Vertex separators for partitioning a graph, 2008.

[25] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of the 19th Design Automation Conference*, pages 175–181. IEEE, 1982.

[26] M. Fiedler. A property of eigenvectors of non-negative symmetric matrices and its application to graph theory. *Czechoslovak Math. J.*, 25 :619–633, 1975.

[27] M. Frank and P. Wolfe. An algorithm for quadratic programming. *Naval Research Logistics Quarterly*, 3(1-2) :95–110, 1956.

[28] M. Gairing, B. Monien, and A. Woclaw. A faster combinatorial approximation algorithm for scheduling unrelated parallel machines. *Theor. Comput. Sci.*, 380 :87–99, 2007.

[29] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified np-complete graph problems. *Theoretical Computer Science*, 1(3) :237 – 267, 1976.

[30] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[31] R. Graham, E. Lawler, J. Lenstra, and A. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling : a survey. volume 5 of *Annals of Discrete Mathematics*, pages 287 – 326. Elsevier, 1979.

[32] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, 17 :416–429, 1969.

[33] C. Guéret, C. Prins, et M. Sevaux. *Applications of optimization with Xpress-MP*. Dash Optimization Ltd., 2000.

[34] I. Gurobi Optimization. Gurobi optimizer reference manual, 2015.

[35] L. Hagen, D. Huang, et A. Kahng. On implementation choices for iterative improvement partitioning algorithms, 1997.

[36] K. M. Hall. An r-dimensional quadratic placement algorithm. *Management Science*, 17(3) :219–229, 1970.

[37] S. Hauck et G. Borriello. An evaluation of bipartitioning techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(8) :849–866, 1997.

[38] B. Hendrickson et T. G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing 26*, 2000.

[39] B. Hendrickson et R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Sci. Compu.*, 16(2) :452–469, 1995.

[40] B. Hendrickson et R. Leland. A multilevel algorithm for partitioning graphs. In *Proceedings of Supercomputing*, 1995.

[41] A. Horowitz et S. Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *J. ACM*, 23(2) :317 – 327, 1976.

[42] O. H. Ibarra et C. E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *J. ACM*, 22(4) :463 – 468, 1975.

[43] R. M. Karp. *Reducibility among Combinatorial Problems*, pages 85 – 103. Springer US, 1972.

[44] G. Karypis, R. Aggarwal, V. Kumar, et S. Shekhar. Multilevel hypergraph partitioning : applications in VLSI domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1) :69–79, 1999.

[45] G. Karypis et V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. Technical Report 95-035, University of Minnesota, June 1995.

[46] G. Karypis et V. Kumar. METIS *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*. University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN 55455, U.S.A., 4 edition, Sept. 1998.

[47] G. Karypis et V. Kumar. A fast and highly quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing, Vol. 20, No. 1, pp. 359—392*, 1999.

[48] METIS : Family of multilevel partitioning algorithms. `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[49] H. Kellerer et V. Kotov. A 3/2-approximation algorithm for 3/2-partitioning. *Oper. Res. Lett.*, 39(5) :359 – 362, 2011.

[50] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *BELL System Technical Journal*, pages 291–307, 1970.

[51] N. G. Kinnersley. The vertex separation number of a graph equals its path-width. *Inf. Process. Lett.*, 42(6) :345 – 350, 1992.

[52] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220 :671–680, 1983.

[53] Y. Kobayashi and C. Sommer. On shortest disjoint paths in planar graphs. *Discrete Optimization*, 7(4) :234 – 245, 2010.

[54] E. Korach and N. Solel. Tree-width, path-width, and cutwidth. *Discrete Applied Mathematics*, 43(1) :97 – 101, 1993.

[55] B. Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Trans. Computers*, 33(5) :438–446, 1984.

[56] C. Lachat, F. Pellegrini, and D. C. PaMPA : Parallel Mesh Partitioning and Adaptation. In *21st International Conference on Domain Decomposition Methods (DD21)*, 2012.

[57] J. Lenstra, D. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46(1) :259 271, 1990.

[58] J. Leung, L. Kelly, and J. Anderson. *Handbook of Scheduling : Algorithms, Models, and Performance Analysis*. CRC Press, Inc., 2004.

[59] H. R. Lourenço, O. C. Martin, and T. Stützle. *Iterated Local Search*. Handbook of Metaheuristics, pages 320–353, 2003.

[60] L. Lovaász and M. Plummer. *Matching Theory*, 1986.

[61] S. Martello, F. Soumis, and P. Toth. Exact and approximation algorithms for makespan minimization on unrelated parallel machines. *Discrete Applied Mathematics*, 75(2) :169 – 188, 1997.

[62] K. Neumann. *Min-sum and min-max single-machine scheduling with stochastic tree-like precedence constraints : Complexity and algorithms*. Springer Berlin Heidelberg, 1990.

[63] K. Neumann. *Nonpreemptive Scheduling with Stochastic Precedence Constraints*, pages 139 – 147. 1991.

[64] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization : Algorithms and Complexity*. Prentice-Hall, Inc., 1982.

[65] F. Pellegrini. *Application de méthodes de partition à la résolution de problèmes de graphes issus du parallélisme*. Thèse de Doctorat, Université Bordeaux I, 1995.

[66] F. Pellegrini et J. Roman. Scotch : A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. *Proceedings of HPCN'96, Brussels, Belgium. LNCS 1067, pages 493-498*, 1996.

[67] Scotch : Static mapping, graph partitioning, and sparse matrix block ordering package. `http://www.labri.fr/~pelegrin/scotch/`.

[68] A. Pothen, H. D. Simon, et K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11 :430–452, 1990.

[69] N. Robertson et P. Seymour. Graph minors. i. excluding a forest. *Journal of Combinatorial Theory, Series B*, 35(1) :39 – 61, 1983.

[70] N. Robertson et P. Seymour. Graph minors. ii. algorithmic aspect of tree-width. *Journal of Algorithms*, 7(3) :309 – 322, 1986.

[71] A. L. Rosenberg et L. S. Health. Graph separators, with applications. *Frontiers of Computer Science*, 2001.

[72] S. K. Sahni. Algorithms for scheduling independent tasks. *J. ACM*, 23(1) :116 – 127, 1976.

[73] E. R. Scheinerman et D. H. Ullman. *Fractional graph theory : a rational approach to the theory of graphs*. Courier Corporation, 2011.

[74] A. Sen, H. Deng, et S. Guha. On a graph partitioning problem with applications to VLSI layout. In *Circuits and Systems, 1991., IEEE Internation Symposium on*, volume 5, pages 2846–2849, 1991.

[75] D. B. Shmoy et E. Tardos. An approximation algorithm for the generalized assignment problem. *Math Programming*, 1993.

[76] H. D. Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 14 :135–148, 1991.

[77] K. Skodinis. Construction of linear tree-layouts which are optimal with respect to vertex separation in linear time. *Journal of Algorithms*, 47(1) :40 – 59, 2003.

[78] A. Trifunovic et W. J. Knottenbelt. *Computer and Information Sciences - ISCIS 2004 : 19th International Symposium, Kemer-Antalya, Turkey, October 27-29, 2004. Proceedings*, chapter Parkway 2.0 : A Parallel Multilevel Hypergraph Partitioning Tool, pages 789–800. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[79] L.-H. Tsai. Asymptotic analysis of an algorithm for balanced parallel processor scheduling. *SIAM Journal on Computing*, 21(1) :59 – 64, 1992.

[80] S. L. Van de Velde. Duality-based algorithms for scheduling unrelated parallel machines. *ORSA Journal on Computing*, 5(2) :192–205, 1993.

[81] R. van Driessche et D. Roose. A graph contraction algorithm for the calculation of eigenvectors of the laplacian matrix of a graph with a multilevel method. Technical Report TW 209, Katholieke Universiteit Leuven, May 1994.

[82] L. Vandenberghe et S. Boyd. Semidefinite programming. *SIAM Review*, 38(1) :49–95, 1996.

[83] Jostle : Graph partitioning software. `http://staffweb.cms.gre.ac.uk/~c.walshaw/jostle/`.

[84] G. J. Woeginger. When does a dynamic programming formulation guarantee the existence of a fully polynomial time approximation scheme (FPTAS) ? *INFORMS Journal on Computing*, 12(1) :57 – 74, 2000.

[85] G. J. Woeginger. A comment on scheduling two parallel machines with capacity constraints. *Discret. Optim.*, 2(3) :269 – 272, 2005.

[86] H. Yang, Y. Ye, et J. Zhang. An approximation algorithm for scheduling two parallel machines with capacity constraints. *Discrete Applied Mathematics*, 130(3) :449 – 467, 2003.

[87] C. Zhang, G. Wang, X. Liu, et J. Liu. *Approximating Scheduling Machines with Capacity Constraints*, pages 283 – 292. Frontiers in Algorithmics : Third International Workshop. Proceedings, 2009.

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

Titre : Étude et obtention d'heuristiques et d'algorithmes exacts et approchés pour un problème de partitionnement de maillage sous contraintes mémoire

Mots clefs : Partitionnement, Maillage, Algorithme approché, Heuristique, Multi-niveaux

Résumé : Dans de nombreux domaines scientifiques, la taille et la complexité des simulations numériques sont si importantes qu'il est souvent nécessaire d'utiliser des supercalculateurs à mémoire distribuée pour les réaliser. Les données de la simulation ainsi que les traitements sont alors répartis sur différentes unités de calculs, en tenant compte de nombreux paramètres. En effet, cette répartition est cruciale et doit minimiser le coût de calcul des traitements à effectuer tout en assurant que les données nécessaires à chaque unité de calcul puissent être stockées localement en mémoire. Pour la plupart des simulations numériques menées, les données des calculs sont attachées à un maillage, c'est-à-dire une discrétisation du domaine géométrique d'étude en éléments géométriques simples, les mailles. Les calculs à effectuer sont alors le plus souvent effectués au sein de chaque maille et la distribution des calculs correspond alors à un partitionnement du maillage. Dans un contexte de simulation numérique, où les méthodes mathématiques utilisées sont de types éléments ou volumes finis, la réalisation du calcul associé à une maille peut nécessiter des informations portées par des mailles voisines. L'approche standard est alors de disposer de ce voisinage localement à l'unité de calcul. Le problème à résoudre n'est donc pas uniquement de partitionner un maillage sur k parties en plaçant chaque maille sur une et une seule partie et en tenant compte de la charge de calcul attribuée à chaque partie. Il faut ajouter à cela le fait de prendre en compte l'occupation mémoire des cellules où les calculs sont effectués et leurs voisines. Ceci amène à partitionner les calculs tandis que le maillage est distribué avec recouvrement. Prendre explicitement ce recouvrement de données est le problème que nous proposons d'étudier.

Title : Study and obtention of exact, and approximation, algorithms and heuristics for a mesh partitioning problem under memory constraints

Keywords : Partitioning, Mesh, Approximation algorithm, Heuristic, Multi-level

Abstract : In many scientific areas, the size and the complexity of numerical simulations lead to make intensive use of massively parallel runs on High Performance Computing (HPC) architectures. Such computers consist in a set of processing units (PU) where memory is distributed. Distribution of simulation data is therefore crucial: it has to minimize the computation time of the simulation while ensuring that the data allocated to every PU can be locally stored in memory. For most of the numerical simulations, the physical and numerical data are based on a mesh. The computations are then performed at the cell level (for example within triangles and quadrilaterals in 2D, or within tetrahedrons and hexahedrons in 3D). More specifically, computing and memory cost can be associated to each cell. In our context, where the mathematical methods used are finite elements or finite volumes, the realization of the computations associated with a cell may require information carried by neighboring cells. The standard implementation relies to locally store useful data of this neighborhood on the PU, even if cells of this neighborhood are not locally computed. Such non computed but stored cells are called ghost cells, and can have a significant impact on the memory consumption of a PU. The problem to solve is thus not only to partition a mesh on several parts by affecting each cell to one and only one part while minimizing the computational load assigned to each part. It is also necessary to keep into account that the memory load of both the cells where the computations are performed and their neighbors has to fit into PU memory. This leads to partition the computations while the mesh is distributed with overlaps. Explicitly taking these data overlaps into account is the problem that we propose to study.