Équilibrage de charge dynamique avec un nombre variable de processeurs basé sur des méthodes de partitionnement de graphe

Clément Vuchener

▶ **To cite this version:**

Clément Vuchener. Équilibrage de charge dynamique avec un nombre variable de processeurs basé sur des méthodes de partitionnement de graphe. Calcul parallèle, distribué et partagé [cs.DC]. Université de Bordeaux, 2014. Français. NNT : . tel-00952777v1

HAL Id: tel-00952777

https://theses.hal.science/tel-00952777v1

Submitted on 27 Feb 2014 (v1), last revised 26 May 2015 (v2)

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.

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Clément VUCHENER**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Équilibrage de charge dynamique avec un nombre variable de processeurs basé sur des méthodes de partitionnement de graphe

---

Soutenue le : 7 février 2014

Après avis des rapporteurs :

Pierre Manneback	Professeur
Bora Uçar	Chargé de Recherche

Devant la commission d'examen composée de :

Florent Duchaine .	Senior Researcher	Examinateur
Aurélien Esnard ..	Maître de Conférences	Directeur de Thèse
Pierre Manneback	Professeur	Président du Jury & Rapporteur
François Pellegrini	Professeur	Examinateur
Jean Roman	Professeur	Directeur de Thèse
Bora Uçar	Chargé de Recherches	Rapporteur

– 2014 –

Remerciements

Avant tout, je remercie mes directeurs de thèse Jean Roman et Aurélien Esnard qui ont accepté d'encadrer cette thèse. Et plus particulièrement Aurélien pour tout le temps qu'il m'a accordé, ces longues discussions qui ont permis à cette thèse d'avancer et tous les conseils donnés.

Je remercie Pierre Manneback et Bora Uçar qui ont accepté de rapporter cette thèse malgré des délais très courts, ainsi que les autres membres du jury : François Pellegrini et Florent Duchaine pour l'intérêt porté à mes travaux.

Je remercie également Sébastien Fourestier et, encore une fois, François Pellegrini pour toute leur aide sur *Scotch* et comment y ajouter ma méthode de partitionnement.

Je remercie toute l'équipe INRIA HiePACS au sein de laquelle j'ai effectué cette thèse et plus particulièrement les autres doctorants de l'équipe qui m'ont tenu compagnie pendant près de trois ans pour certains.

Équilibrage de charge dynamique avec un nombre variable de processeurs basé sur des méthodes de partitionnement de graphe

Résumé

L'équilibrage de charge est une étape importante conditionnant les performances des applications parallèles. Dans le cas où la charge varie au cours de la simulation, il est important de redistribuer régulièrement la charge entre les différents processeurs. Dans ce contexte, il peut s'avérer pertinent d'adapter le nombre de processeurs au cours d'une simulation afin d'obtenir une meilleure efficacité, ou de continuer l'exécution quand toute la mémoire des ressources courantes est utilisée. Contrairement au cas où le nombre de processeurs ne varie pas, le rééquilibrage dynamique avec un nombre variable de processeurs est un problème peu étudié que nous abordons ici.

Cette thèse propose différentes méthodes basées sur le repartitionnement de graphe pour rééquilibrer la charge tout en changeant le nombre de processeurs. Nous appelons ce problème « repartitionnement $M \times N$ ». Ces méthodes se décomposent en deux grandes étapes. Dans un premier temps, nous étudions la phase de migration et nous construisons une « bonne » matrice de migration minimisant plusieurs critères objectifs comme le volume total de migration et le nombre total de messages échangés. Puis, dans un second temps, nous utilisons des heuristiques de partitionnement de graphe pour calculer une nouvelle distribution optimisant la migration en s'appuyant sur les résultats de l'étape précédente. En outre, nous proposons un algorithme de partitionnement k-aire direct permettant d'améliorer le partitionnement biaisé. Finalement, nous validons cette thèse par une étude expérimentale en comparant nos méthodes aux partitionneurs actuels.

Mots-clés

Simulation numérique, parallélisme, équilibrage de charge dynamique, redistribution, partitionnement de graphe, repartitionnement.

Dynamic Load-Balancing with Variable Number of Processors based on Graph Partitioning

Abstract

Load balancing is an important step conditioning the performance of parallel programs. If the workload varies drastically during the simulation, the load must be redistributed regularly among the processors. Dynamic load balancing is a well studied subject but most studies are limited to an initially fixed number of processors. Adjusting the number of processors at runtime allows to preserve the parallel code efficiency or to keep running the simulation when the memory of the current resources is exceeded.

In this thesis, we propose some methods based on graph repartitioning in order to rebalance the load while changing the number of processors. We call this problem "$M \times N$ repartitioning". These methods are split in two main steps. Firstly, we study the migration phase and we build a "good" migration matrix minimizing several metrics like the migration volume or the number of exchanged messages. Secondly, we use graph partitioning heuristics to compute a new distribution optimizing the migration according to the previous step results. Besides, we propose a direct k-way partitioning algorithm that allows us to improve our biased partitioning. Finally, an experimental study validates our algorithms against state-of-the-art partitioning tools.

Keywords

Numerical simulation, parallelism, dynamic load-balancing, redistribution, graph partitioning, repartitioning.

Table des matières

1 Introduction **1**

2 État de l'art **5**
2.1 Contexte 5
2.2 Partitionnement 6
2.2.1 Modélisation 6
2.2.2 Méthodes de partitionnement de graphe 9
2.2.3 État de l'art des outils de partitionnement 11
2.3 Repartitionnement pour l'équilibrage dynamique 13
2.3.1 Modélisation du problème de repartitionnement 13
2.3.2 Méthodes de repartitionnement de graphe 14
2.4 Positionnement 20

3 Modèle de migration pour le repartitionnement $M \times N$ **25**
3.1 Notations et définitions 25
3.1.1 Généralités 26
3.1.2 Définitions pour le repartitionnement $M \times N$ 27
3.2 Matrices de migration optimales dans le cas équilibré 28
3.3 Construction des matrices de migration 35
3.3.1 Méthode basée sur la chaîne (*1D*) 35
3.3.2 Méthode d'appariement (*Matching*) 42
3.3.3 Méthode gloutonne (*Greedy*) 47
3.3.4 Programme linéaire (*LP*) 58
3.4 Évaluation des méthodes et conclusion 62

4 Repartitionnement $M \times N$ **67**
4.1 Repartitionnement $M \times N$ basé sur le partitionnement biaisé (*BIASED*) 67
4.1.1 Méthodologie 68
4.1.2 Limitations des partitionneurs 71
4.2 Partitionnement k-aire direct (*KGGGP*) 73
4.2.1 Description de l'algorithme 73
4.2.2 Critères de sélection 75
4.2.3 Complexité 76
4.2.4 Améliorations 77
4.2.5 Évaluation 78
4.3 Repartitionnement $M \times N$ basé sur la diffusion (*DIFF*) 85
4.4 Partitionnement biaisé à l'aide d'hyper-arêtes de repartitionnement 89
4.5 Conclusion 90

5 Résultats **91**
5.1 Méthodologie expérimentale 91
5.2 Influence du coût de migration et du facteur de repartitionnement 94
5.3 Influence du nouveau nombre de processeurs 97
5.4 Comparaison sur des graphes complexes 101
5.5 Étude de la complexité en temps 110

6 Conclusion et Perspectives **115**
6.1 Conclusion 115
6.2 Perspectives 116

A Programmes linéaires pour optimiser les différentes métriques **119**
A.1 Minimisation de TotalV 119
A.2 Minimisation de MaxV 120
A.3 Minimisation de TotalZ 120
A.4 Minimisation de MaxZ 121

B Documentation du KGGGP dans *Scotch* **123**

C Résultats supplémentaires **125**
C.1 Influence du coût de migration et du facteur de repartitionnement 125
C.2 Comparaison sur des graphes complexes 126

Bibliographie **133**

Liste des publications **137**

Chapitre 1

Introduction

Une simulation numérique modélise généralement l'évolution en temps d'un phénomène physique dans un domaine de l'espace. Souvent, on utilise un schéma itératif en temps et une discrétisation de l'espace via un maillage d'éléments géométriques (triangles, carrés, tétraèdres, etc.). Des simulations de plus en plus complexes demandent une plus grande puissance de calcul. Pour cela, ces applications doivent fonctionner en parallèle sur plusieurs nœuds de calcul. Les données sont alors réparties sur les différentes unités de traitement ou « processeurs ». Les processeurs ne pouvant pas accéder directement et rapidement aux données des autres processeurs, ce parallélisme induit un surcout dû aux communications sur le réseau. La mémoire étant distribuée, la répartition des calculs implique une distribution des données.

Dans ce contexte, la distribution des données est une étape importante conditionnant les performances d'une simulation numérique et elle est faite selon deux objectifs :

— la charge de calcul affectée aux différents processeurs doit être équilibrée pour minimiser le temps de calcul de chacun ;
— les communications inter-processeurs induites par la distribution doivent être minimisées.

Certaines simulations numériques complexes ont une charge de calcul qui peut évoluer de façon imprévisible ; c'est le cas par exemple des simulations de type AMR (*Adaptive Mesh Refinement*) où la discrétisation de l'espace évolue pour être plus fine autour de points d'intérêt. La charge de certains processeurs peut augmenter plus rapidement que d'autres et il n'est alors plus possible de garder une même distribution tout le long de la simulation. De nouvelles distributions doivent être recalculées pour rééquilibrer la charge lorsque le déséquilibre devient trop important. On parle alors d'*équilibrage dynamique*. En plus des deux objectifs précédents, la nouvelle distribution doit répondre à deux nouvelles contraintes :

— la nouvelle partition doit être calculée rapidement en parallèle ;
— la migration des données nécessaire pour passer de l'ancienne distribution à la nouvelle doit être la plus rapide possible, c'est-à-dire qu'il faut que le moins possible de données ne change de processeurs.

Le problème de calcul de telles distributions peut être résolu à l'aide du *partitionnement* d'un graphe non-orienté en k parties. Les sommets du graphe représentent les calculs et les arêtes représentent les dépendances entre ces calculs. Chaque partie représente l'ensemble des calculs affectés à un processeur. Deux sommets connectés qui se trouvent dans deux parties différentes (on dira que l'arête est coupée) induisent un besoin de communication entre ces deux processeurs. L'objectif de la distribution devient alors de construire une partition dont les parties sont de même taille et coupant un minimum d'arêtes. Dans le cas de l'équilibrage dynamique, on parlera de « repartitionnement ». Le partitionnement de graphe est un problème déjà bien étudié, que

ce soit dans le cas *statique* où une seule partition est calculée, ou *dynamique* où la partition est recalculée régulièrement.

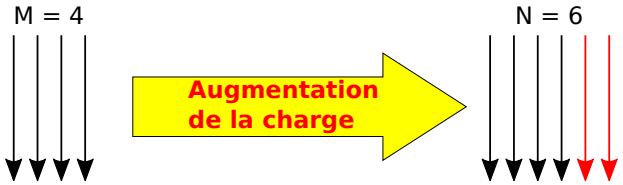


FIGURE 1.1 – Allocation dynamique de processeurs pour un code parallèle dont la charge augmente.

Lorsque la charge de calcul varie de façon importante, il peut devenir intéressant de changer le nombre de processeurs alloués. Cela peut permettre de continuer la simulation quand toute la mémoire des processeurs est utilisée, ou ainsi d'améliorer la consommation des ressources pour garder une bonne efficacité au cours de la simulation. Par exemple, un code AMR utilise initialement un maillage très simple avec peu d'éléments, qu'il ne serait pas efficace de distribuer sur un grand nombre de processeurs. Avec le raffinement du maillage, la quantité de calcul augmente de façon importante, et la parallélisation sur un plus grande nombre de processeurs devient plus intéressante, voire nécessaire si la taille du problème dépasse la capacité mémoire des processeurs déjà alloués. Il faut alors être capable de calculer une nouvelle distribution pour un plus grand nombre de processeurs, comme illustré en figure 1.1.

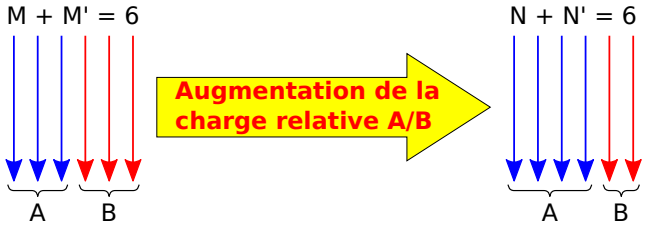


FIGURE 1.2 – Allocation dynamique de processeurs pour deux codes couplés A et B dont la charge relative de A par rapport à B augmente.

L'allocation dynamique de processeurs est également utile dans le cas des codes couplés. Ce sont différents codes s'exécutant en parallèle et collaborant pour réaliser une simulation plus complexe. Ces codes doivent régulièrement échanger des données entre eux durant la « phase de couplage » qui induit une synchronisation plus ou moins explicite. Il est donc nécessaire que les différents codes avancent à la même vitesse pour ne pas se ralentir entre eux. Cet équilibrage entre les codes peut s'avérer très difficile à cause des natures très différentes de ces codes, ou dans le cas où au moins un des codes a une charge de calcul qui évolue dynamiquement. La figure 1.2 présente un exemple avec deux codes couplés A et B utilisant initialement 3 processeurs chacun. Si le code B devient trop rapide par rapport à A, on peut transférer un processeur du code B vers A.

Cette thèse propose un ensemble de méthodes permettant de résoudre le problème d'équilibrage dynamique de la charge avec changement du nombre de processeurs, et ce en se basant sur

le repartitionnement de graphe. Nous appellerons ce problème, le repartitionnement $M \times N$. Ce repartitionnement doit réaliser les différents objectifs déjà cités :

— équilibrage de la charge ;
— minimisation des communications dues aux dépendances de calculs ;
— rapidité de calcul de la nouvelle partition ;
— minimisation des communications dues à la redistribution.

Ceci devra être fait en prenant en compte à la fois une charge variable et un nombre de processeurs variable.

Ces méthodes de repartitionnement se divisent en deux étapes : la construction d'une *matrice de migration* puis le repartitionnement basé sur cette matrice de migration. Plusieurs méthodes de construction de matrices de migration permettent d'optimiser différentes métriques relatives à la migration des données. Deux méthodes de repartitionnement sont présentées : l'une de partitionnement biaisé et l'autre diffusive. De plus, des limitations des méthodes de bissections récursives utilisées habituellement sont mises en évidence dans le cadre du partitionnement biaisé et nous proposons une méthode de partitionnement k-aire direct pour y remédier.

Nous commencerons par présenter un état de l'art du partitionnement et repartitionnement de graphe (chapitre 2). Puis, dans le chapitre 3, nous définirons et construirons des modèles pour la migration. Le chapitre 4 présentera des méthodes de repartitionnement s'appuyant sur les modèles de migration introduits au chapitre précédent. Puis, nous évaluerons nos méthodes au travers d'une série d'expériences comparatives avec d'autres outils de repartitionnement de graphe (chapitre 5). Finalement, nous conclurons et présenterons quelques pistes d'améliorations de ces méthodes dans le chapitre 6. Plusieurs annexes complètent ce manuscrit.

Chapitre 2

État de l'art

Sommaire

2.1 Contexte 5
2.2 Partitionnement 6
2.2.1 Modélisation 6
2.2.2 Méthodes de partitionnement de graphe 9
2.2.3 État de l'art des outils de partitionnement 11
2.3 Repartitionnement pour l'équilibrage dynamique 13
2.3.1 Modélisation du problème de repartitionnement 13
2.3.2 Méthodes de repartitionnement de graphe 14
2.4 Positionnement 20

2.1 Contexte

Une simulation numérique modélise l'évolution d'un phénomène physique dans le temps sur un domaine de l'espace. Ces simulations s'appuient sur une discrétisation en temps et en espace. Le domaine de l'espace simulé est discrétisé à l'aide d'un maillage d'éléments géométriques : triangles, carrés, tétraèdres, ... La discrétisation en temps permet de calculer l'état du système à un instant donné à partir de l'état précédent. Plus précisément, l'état d'un élément est calculé à partir de son état précédent et de l'état de quelques autres éléments en fonction du modèle physique ; généralement, ce sont les éléments voisins dans le maillage.

Ces simulations, de plus en plus complexes, nécessitent d'être exécutées en parallèle. Le domaine doit donc être distribué parmi plusieurs unités de calcul. Le maillage est donc découpé en autant de parties que de processeurs utilisés. Le calcul de l'état d'un élément dépendant de l'état d'autres éléments, le plus souvent dans son voisinage, cette distribution induit des communications entre processeurs quand l'un d'entre eux nécessite les données possédées par un autre. Ces communications sont synchronisantes et tous les processeurs doivent donc faire progresser la simulation au même rythme. Une bonne distribution ou partition des données doit donc fournir des parties équilibrées et minimisant le nombre de dépendances de données entre processeurs, minimisant ainsi le temps de calcul et de communication. Par exemple, un code simulant la diffusion de la chaleur sur un maillage calcule la chaleur de chaque élément à partir de sa propre chaleur et de celle des éléments voisins à l'itération précédente. Une unité de calcul travaillant

sur un domaine doit donc recevoir les températures des éléments sur la frontière des domaines voisins.

La distribution des données peut être *statique*, c'est-à-dire fixée au début de la simulation, ou *dynamique* qui évolue au cours de la simulation. Certaines simulations utilisent des maillages dynamiques : le domaine de simulation peut s'étendre, par exemple pour la simulation d'une explosion, ou le maillage peut être raffiné autour de certains points d'intérêt permettant ainsi d'utiliser peu d'éléments tout en gardant une bonne précision là où c'est nécessaire. Cette dernière méthode est appelée *Adaptive Mesh Refinment* (AMR). Au cours de ces simulations dynamiques, la charge de calcul des processeurs évolue de façon hétérogène. Il est donc nécessaire de rééquilibrer régulièrement cette charge en calculant une nouvelle partition des données. Une fois cette nouvelle partition réalisée, les données doivent être migrées vers leur nouveau propriétaire. La nouvelle partition doit optimiser cette phase de migration en plus des critères précédents.

Il existe une grande variété de méthodes pour partitionner des maillages. Les plus courantes sont les méthodes géométriques, s'appuyant uniquement sur les coordonnées des éléments dans l'espace, et les méthodes basées sur les graphes utilisant les dépendances de calcul entre éléments et ne s'intéressant plus qu'à la topologie et non à la géométrie du problème.

2.2 Partitionnement

Le partitionnement de maillages à l'aide de graphes utilise les dépendances de calcul entre les éléments pour construire un graphe. Certains partitionneurs utilisent un modèle d'hypergraphe, plus général que le modèle de graphe. Un partitionneur de graphe ou d'hypergraphe calcule des partitions en prenant en compte deux objectifs :

— les parties doivent être équilibrées, c'est-à-dire de même taille. Les partitionneurs sont généralement tolérants et se contentent de créer des parties à peu près équilibrées à un « facteur de déséquilibre » près. Un facteur de déséquilibre de 5% veut dire qu'une partie pourra atteindre jusqu'à $1,05$ fois sa taille idéale. Ce premier objectif permet d'optimiser le temps de calcul de la simulation.

— une fonction de coût, appelée coupe du graphe, est minimisée. Cette fonction de coût varie suivant les modèles mais représente le temps de communication entre les processeurs.

2.2.1 Modélisation

Pour pouvoir appliquer le partitionnement de graphe ou d'hypergraphe à un problème donné, il est nécessaire de modéliser celui-ci. Il est important de choisir une modélisation adaptée au problème pour que le partitionneur puisse créer une partition optimisant les bons critères pour cette simulation.

a) Modélisation à l'aide de graphes

L'approche la plus courante consiste à modéliser les éléments à distribuer sous forme d'un graphe. Le graphe construit doit modéliser les contraintes du calcul distribué : équilibrage de charge et réduction de communications.

Un graphe est défini par un ensemble de sommets et un ensemble d'arêtes connectant les sommets par paires. Le graphe est donc construit avec un sommet pour représenter chaque élément. Les arêtes de ce graphe servent à représenter les dépendances entre les éléments. Par exemple, le calcul sur un élément peut nécessiter les données des éléments voisins dans le maillage. Un sommet est donc connecté à tous les sommets représentant les éléments voisins dans le maillage.

Si des éléments sont associés à des charges de calcul différentes, il est possible d'associer des poids aux sommets représentant ces charges de calcul. De la même façon, si certaines communications sont plus coûteuses que d'autres, un poids peut être affecté aux arêtes.

Un partitionneur de graphe optimise deux objectifs : l'équilibrage et la coupe. La taille d'une partie est le nombre de sommets dans cette partie ou, s'ils sont pondérés, la somme des poids de ceux-ci. La fonction de coût la plus généralement utilisée dans les partitionneurs de graphe est le nombre d'arêtes coupées (ou la somme de leurs poids). Une arête est dite « coupée » lorsque ses deux extrémités sont affectées à des parties différentes. On vise ainsi à réduire le volume de données à communiquer entre processeurs.

La figure 2.1 présente la modélisation d'un maillage : une grille de 2×3 carrés sous forme de graphe. Chaque sommet du graphe correspond à un carré de la grille et est relié aux sommets correspondant à ses voisins dans le maillage.

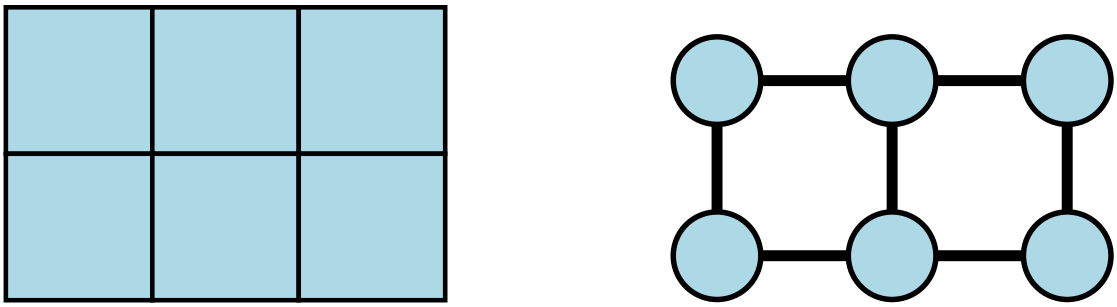


FIGURE 2.1 – Un maillage de 6 éléments et le graphe associé modélisant les dépendances.

Cette modélisation, bien que très répandue, ne modélise pas toujours parfaitement le volume de communication nécessaire pour la distribution induite. On pourrait penser que comme une arête relie deux sommets, chaque arête coupée induit deux sommets devant communiquer. Mais un sommet peut avoir plusieurs arêtes coupées par la même partie et la coupe du graphe peut surévaluer le volume de communication induit par cette partition. Sur la figure 2.2, un maillage de 6 éléments est partitionné en deux : une partie bleue et une partie verte. Il y a 4 éléments qui ont au moins un voisin dans l'autre partie, ce sont les éléments qui ont besoin d'être communiqués avec l'autre processeur. Mais 3 arêtes sont coupées par la partition.

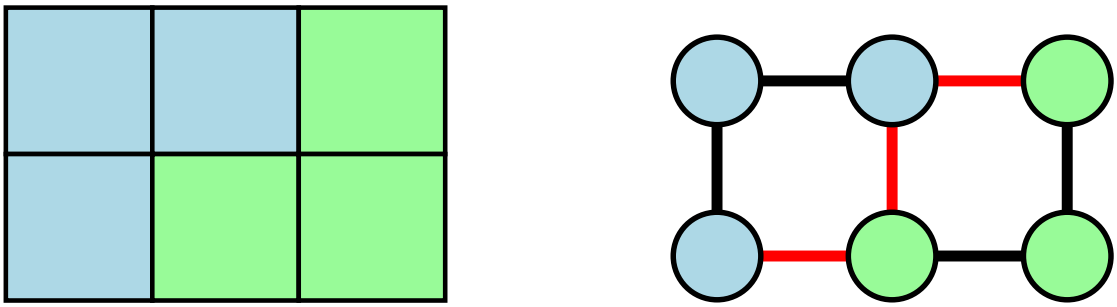


FIGURE 2.2 – Maillage et graphe partitionnés en deux.

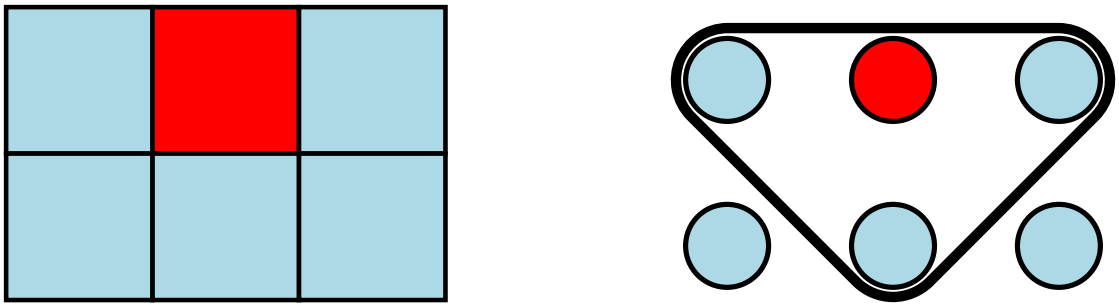


FIGURE 2.3 – Un maillage de 6 éléments et l'hypergraphe associé modélisant les dépendances. Pour ne pas surcharger le dessin, seule l'hyper-arête correspondant à l'élément rouge est représentée.

b) Modélisation à l'aide d'hypergraphes

Il est également possible d'utiliser une modélisation à partir d'hypergraphe. Le partitionnement d'hypergraphe est depuis longtemps utilisé pour le partitionnement de VLSI [33, 59], mais il est également intéressant dans le cas du partitionnement de matrices ou de maillages [63].

Les hypergraphes sont des extensions des graphes, où une arête appelée hyper-arête, ne connecte plus deux sommets mais un nombre quelconque de sommets. Un graphe peut être considéré comme un hypergraphe particulier contenant uniquement des hyper-arêtes de taille 2.

Ces hyper-arêtes plus complexes rendent la représentation graphique des hypergraphe plus difficile. Il existe plusieurs façons de représenter une hyper-arête ; par exemple, on peut dessiner une courbe entourant les sommets inclus dans l'hyper-arête. Il est aussi possible de représenter une hyper-arête par un sommet spécial relié aux sommets qu'elle connecte. Cette représentation est en fait celle d'un graphe biparti dont les parties sont l'ensemble des sommets et l'ensemble des hyper-arêtes.

Comme avec les graphes, chaque élément d'un maillage est représenté par un sommet de l'hypergraphe. Mais les hyper-arêtes permettent de modéliser différemment les dépendances. Une hyper-arête est associée à chaque sommet, représentant toutes les dépendances de cet élément. Une hyper-arête contient donc le sommet auquel elle est associée, ainsi que tous les sommets qui dépendent de celui-ci.

La figure 2.3 montre une hyper-arête de l'hypergraphe modélisant la grille 2×3. Cette hyper-arête est associée au sommet rouge et contient tous les sommets voisins.

Cette nouvelle modélisation des dépendances permet d'utiliser des modèles de coupe différents représentant mieux les communications induites par le partitionnement. La coupe connectivité-1, aussi appélée $\lambda-1$, permet de représenter fidèlement les communications [63]. Ce modèle de coupe prend en compte le nombre de parties coupant une arête. Si une hyper-arête est coupée par λ parties, son poids est compté $\lambda-1$ fois dans la coupe finale. En effet, une hyper-arête est associée à un élément et λ parties possèdent des éléments appartenant à cette hyper-arête. Ces éléments sont soit ceux qui ont une dépendance sur l'élément associé à l'hyper-arête, soit l'élément lui-même. Parmi ces λ parties, une possède l'élément et les $\lambda-1$ autres devront recevoir les données associées à cet élément.

La figure 2.4 montre la même partition que la figure 2.2 mais en utilisant un modèle d'hypergraphe. Parmi les 6 hyper-arêtes de cet hypergraphe, 4 sont coupées (en rouge). Ce sont les hyper-arêtes associées aux sommets se trouvant sur la frontière.

La modélisation à l'aide d'hypergraphe permet de modéliser exactement la quantité de données se trouvant sur la frontière des domaines contrairement à la modélisation à l'aide de graphe.

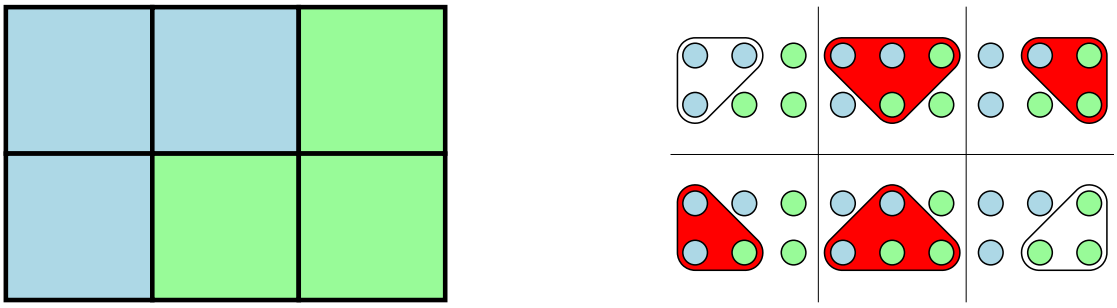


FIGURE 2.4 – Maillage et hypergraphe partitionné en deux. Les hyper-arêtes en rouge sont coupées par les deux parties.

c) Sommets fixes

Certains partitionneurs permettent d'utiliser des *sommets fixes*. Ces sommets sont initialement fixés dans une partie et le partitionneur ne remettra pas en cause cette affectation, mais en tiendra compte dans l'équilibrage et l'optimisation de la coupe. Les sommets fixes ont été largement utilisés dans le cadre du partitionnement de VLSI [7] mais ont également trouvé une utilisation dans le cadre du rééquilibrage dynamique [2, 9]. Le problème de partitionnement à sommets fixes est une version plus contrainte du partitionnement de graphe ou hypergraphe complètement libre. Bien que le problème puisse paraitre plus simple, comme il y a moins de sommets à placer dans les différentes parties, les heuristiques élaborées pour le partitionnement libre peuvent avoir du mal à prendre en compte ces nouvelles contraintes [3, 7].

2.2.2 Méthodes de partitionnement de graphe

Le partitionnement d'un graphe ou d'un hypergraphe est un problème NP-complet [20, GRAPH PARTITIONING]. On utilise donc diverses heuristiques pour pouvoir calculer une partition en un temps raisonnable. La plupart des méthodes utilisées s'appliquent aussi bien sur des graphes que sur des hypergraphes.

La plupart des partitionneurs de graphe ou d'hypergraphe utilisent des méthodes appelées « multi-niveaux » [4, 22, 36]. L'approche multi-niveaux permet d'accélérer les méthodes de partitionnement classiques tout en gardant une bonne qualité. Cette approche se décompose en trois étapes, comme indiqué en figure 2.5.

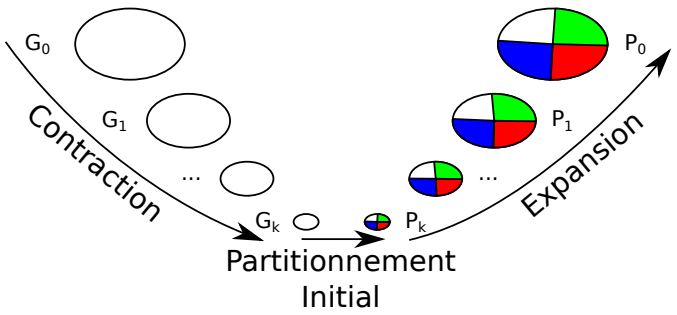


FIGURE 2.5 – Phases d'un partitionnement multi-niveaux.

La phase de contraction : Dans un premier temps, la taille du graphe est réduite en fusionnant des sommets. Cela est répété pendant plusieurs itérations, jusqu'à obtenir un graphe suffisamment petit. Une suite de graphes de taille décroissante a ainsi été créée : $(G_0, G_1, \ldots, G_k)$.

Le partitionnement initial : Une fois que le graphe a été suffisamment contracté, on applique une heuristique de partitionnement pour calculer la partition P_k du graphe G_k. N'importe quelle stratégie de partitionnement peut être appliquée ici. Certaines seront détaillées par la suite.

La phase d'expansion : La suite des différents graphes construits pendant l'étape de contraction est alors « remontée ». La partition P_{i+1} du graphe G_{i+1} est prolongée sur le graphe G_i puis cette nouvelle partition P_i est raffinée à l'aide d'une heuristique améliorant localement la coupe.

a) Stratégies de contraction

Il existe plusieurs heuristiques pour sélectionner les sommets à fusionner. La plus connue, appelée *Heavy Edge Matching* (HEM), recherche les arêtes de poids les plus élevés et en fusionne les deux extrémités. Cette opération est répétée jusqu'à obtenir un graphe suffisamment petit. Dans un partitionneur supportant les sommets fixes, ces derniers doivent être traités de manière particulière pendant la phase de contraction. En effet, le graphe contracté doit garder les mêmes informations sur les sommets fixes. Dans certains partitionneurs, il peut être décidé de ne jamais fusionner ces sommets fixes. Dans d'autres, il peut être décidé de les fusionner seulement avec des sommets non fixés ou fixés dans la même partie. Dans ce cas, le sommet obtenu après fusion est considéré comme un sommet fixe dans le graphe contracté.

b) Stratégies de partitionnement initial

La stratégie de partitionnement utilisée au niveau le plus bas dans un partitionneur multi-niveaux peut être n'importe quelle stratégie de partitionnement, y compris un autre algorithme multi-niveaux. Il existe une très grande variété de méthodes de partitionnement ou de bissection comme par exemple des algorithmes gloutons ajoutant les sommets un à un dans les parties [5, 6, 29], ou des algorithmes spectraux utilisant des propriétés d'algèbre linéaire pour obtenir une partition [53].

La stratégie la plus utilisée est celle des bissections récursives. Le graphe est coupé récursivement en deux parties jusqu'à obtenir le nombre de parties souhaité. Il est possible d'obtenir un nombre de parties qui n'est pas une puissance de 2 en créant des parties déséquilibrées. Par exemple pour obtenir 3 parties, on crée deux parties avec les proportions 1/3 et 2/3, la partie de taille 2/3 est ensuite coupée en deux parties égales de 1/3 chacune. Son intérêt réside dans la grande variété et qualité des heuristiques de bissection souvent mieux maîtrisées que les heuristiques de partitionnement k-aires [35].

Le graphe peut également être partitionné directement en k parties ; on parlera de partitionnement k-aire direct. Ces stratégies peuvent être plus rapides que les bissections récursives car une seule partition est calculée. Théoriquement, le partitionnement k-aire direct permet une meilleure qualité car il profite d'une vision globale du problème que les bissections récursives n'ont pas [60]. Mais en pratique, il est plus difficile de calculer une bonne partition k-aire directement et la méthode des bissections récursives donne souvent de meilleurs résultats [35].

c) Stratégies de raffinement utilisées dans la phase d'expansion

Les stratégies de raffinement modifient une partition déjà existante en changeant quelques sommets de partie pour améliorer la coupe.

L'algorithme de Kernighan et Lin [39] (KL) raffine une bissection en cherchant des paires de sommets dont l'échange améliore le plus la coupe (ou la dégrade le moins). Un sommet déplacé ne sera plus considéré pour un échange dans la même passe. Ces échanges sont considérés même s'ils dégradent la coupe, cela permettant de ne pas rester bloqué sur une solution localement optimale mais qui pourrait être améliorée en s'en éloignant plus. À la fin de la passe, seuls les échanges menant à la meilleure coupe sont conservés. L'échange par paires permet de conserver l'équilibre mais rend l'algorithme complexe. Plusieurs passes peuvent être appliquées pour améliorer encore plus la coupe.

Fiduccia et Mattheyses [17] (FM) s'inspirent de cet algorithme pour décrire un raffinement linéaire en temps en ne déplaçant les sommets que un par un et grâce à une structure de données adaptée. L'équilibre est conservé en se refusant d'effectuer des déplacements qui déséquilibreraient trop la partition, c'est-à-dire que la partition dépasserait la taille maximale donnée par le facteur de déséquilibre. Cet algorithme, initialement conçu pour les bissections, a également l'avantage de s'étendre facilement aux k-partitions. Le principe général reste le même, mais il y a plusieurs déplacements à considérer pour chaque sommet.

Les sommets fixes s'intègrent facilement aux méthodes de types KL/FM, car il suffit de ne pas considérer ces sommets parmi les déplacements possibles.

Comme un algorithme de raffinement se contente d'améliorer une partition déjà existante en ne déplaçant souvent que les sommets le long de la frontière, il n'est pas toujours nécessaire de l'appliquer sur le graphe entier. Ces stratégies sont souvent appliquées sur un graphe « bande » ne contenant que les sommets à une certaine distance de la frontière entre les parties, les autres sommets étant réunis en un seul sommet fixe pour chaque partie appelé « ancre ». La taille du problème est ainsi largement réduite [10].

D'autres méthodes de raffinement s'inspirent de la diffusion d'un liquide dans le graphe [42,49]. Ces méthodes se basent sur des graphes bandes : un type liquide différent pour chaque partie est injecté dans chaque sommet ancre. Le liquide se diffuse itérativement dans le graphe le long des arêtes et est détruit lorsqu'il rencontre un autre type de liquide. Quand un sommet reçoit plusieurs type de liquide, il choisit la partie dont il en reçoit le plus. Cette heuristique tend à créer des parties aux formes très lisses mais la coupe n'est pas toujours optimale. Elle aussi pour avantage d'être locale et donc d'être plus efficace en parallèle que les heuristiques de type KL/FM.

2.2.3 État de l'art des outils de partitionnement

Il existe une grande variété d'outils de partitionnement. Le tableau 2.1 présente quelques partitionneurs de graphe ou d'hypergraphe et quelques unes de leurs fonctionnalités. Les sommets fixes sont surtout supportés par les partitionneurs d'hypergraphe car ils sont utilisés depuis longtemps pour le partitionnement de circuits intégrés (VLSI). Beaucoup de partitionneurs offrent le choix entre un partitionnement multi-niveaux récursif ou directement k-aire, mais le partitionnement initial du multi-niveaux direct utilise le plus souvent des bissections récursives.

Metis [31] est un des partitionneurs les plus connus et utilisés. Il possède deux méthode de partitionnement : récursif ou k-aire mais la méthode k-aire se base sur des bissections récursives pour son partitionnement initial. Il existe également une version parallèle : *ParMetis* permettant de créer une partition en parallèle sur au moins 2 processus. *ParMetis* propose également une méthode de repartitionnement permettant de rééquilibrer une partition existante [57]. *hMetis* est un partitionneur d'hypergraphe également disponible en version récursive ou k-aire. Il supporte

Outil	Type	Sommets fixes	Parallèle	Type de multi-niveaux	Partitionnement initial
Metis [31]	graphe	non	non	bissections [34]	—
				k-aire [35]	bissections
ParMetis [32]	graphe	non	oui	k-aire [37]	bissections
hMetis [30]	hypergraphe	oui	non	bissections [33]	—
		non		k-aire [38]	bissections
PaToH [64]	hypergraphe	oui	non	bissections	—
kPaToH [3]	hypergraphe	oui	non	k-aire	bissections + *matching*
Zoltan [1]	hypergraphe	oui	oui	bissections	—
Scotch [48]	graphe	oui	non	bissections	—
				k-aire	bissections
PT-Scotch [48]	graphe	non	oui	bissections	
RM-Metis [2]	graphe	oui (exactement k)	non	k-aire	k-aire direct ($GGGP$)
Mondriaan [62]	hypergraphe	non	non	bissections	—
ML-Part [47]	hypergraphe	oui	non	bissections	—
Parkway [61]	hypergraphe	non	oui	k-aire	bissections
Chaco [24]	graphe	non	non	bissections	—

TABLE 2.1 – Partitionneurs de graphe et d'hypergraphe.

les sommets fixes mais ne peut optimiser que le nombre d'hyper-arêtes coupées et non la coupe $\lambda - 1$.

Scotch [48] est un partitionneur de graphe et de maillage sous licence libre. Il est très configurable et modulaire. Il propose une grande variété de méthodes de partitionnement, mais seulement des méthodes de bissections sont disponibles pour le partitionnement initial. Depuis sa dernière version (6.0), il permet d'utiliser des sommets fixes et de rééquilibrer une partition existante. *PT-Scotch* est la version parallèle multi-thread et multi-processus de *Scotch*.

Zoltan [1] est une bibliothèque permettant de gérer la distribution des données pour des applications parallèles. Il offre en particulier des fonctionnalités de partitionneur. Il est possible d'utiliser *ParMetis* ou *Scotch*, mais il contient également son propre partitionneur d'hypergraphe. Le partitionneur d'hypergraphe de *Zoltan* utilise des bissections récursives et supporte les sommets fixes. Il est également capable de rééquilibrer la distribution des données lorsque celles-ci évoluent dynamiquement [8, 9].

2.3 Repartitionnement pour l'équilibrage dynamique

Dans certaines simulations numériques, la charge varie dynamiquement. C'est par exemple le cas des simulations AMR (Adaptive Mesh Refinement) dans lesquelles la résolution du maillage s'adapte dynamiquement pour réduire la quantité de calcul tout en gardant une précision suffisante là où elle est nécessaire.

La charge ne varie pas de façon homogène et il devient nécessaire après un certain temps de simulation de rééquilibrer la charge entre les différents processeurs. Le graphe représentant les données doit alors être « repartitionné ». Les objectifs du repartitionnement de graphe sont :

- la charge de calcul doit être équilibrée entre tous les processeurs ;
- le volume de données à communiquer entre les processeurs doit être minimal ;
- le temps de calcul de la nouvelle partition doit être minimal ;
- la migration des données, c'est à dire l'envoi depuis l'ancien propriétaire des sommets vers le nouveau, doit être le plus rapide possible.

Aux deux premiers objectifs communs avec le partitionnement classique, on ajoute la minimisation du temps de repartitionnement (calcul de la nouvelle partition et migration induite par celle-ci). Notons que pour obtenir une partition de qualité (d'après les deux premiers critères), il est souvent nécessaire de migrer beaucoup de sommets.

2.3.1 Modélisation du problème de repartitionnement

On peut exprimer le temps total de la simulation (T_{total}) à minimiser sous la forme [8, 57] :

$$T_{total} = \alpha \times (T_{calcul} + T_{comm}) + T_{repart} + T_{mig}$$

avec :

- α le nombre d'itérations avant que la partition devienne trop déséquilibrée et qu'un repartitionnement devienne nécessaire ;
- T_{calcul} le temps d'une itération de calcul, pris sur le processeur le plus lent et donc minimisé par l'équilibrage ;
- T_{comm} le temps des communications entre processeurs, correspondant à la coupe du graphe ;
- T_{repart} le temps passé à calculer la nouvelle partition ;
- T_{mig} le temps de migration des sommets vers leur nouvelle partie.

Ce modèle d'une simulation itérative est représenté en figure 2.6. Après chaque itération de calcul, les processeurs effectuent une phase de communication collective qui les synchronise ; le temps de calcul en parallèle est donc celui du processeur le plus lent. Après $\alpha = 3$ itérations, la partition devient trop déséquilibrée ; une nouvelle partition est calculée lors du repartitionnement puis les sommets sont migrés pour respecter cette nouvelle partition.

En pratique, le repartitionnement n'est pas effectué après un nombre fixe d'itérations mais quand le déséquilibre dépasse un certain seuil. On peut alors considérer que α est le nombre moyen d'itérations avant que ce seuil soit dépassé.

Obtenir une nouvelle partition de qualité (c'est-à-dire avec T_{calcul} et T_{comm} faibles), impose un repartitionnement plus complexe et une nouvelle partition plus éloignée de l'ancienne (T_{repart} et T_{mig} plus élevés). Il est donc nécessaire de réaliser un compromis suivant la valeur de α entre la qualité de la nouvelle partition et la rapidité de sa construction et mise en place. Si le besoin de repartitionnement est rare (α élevé), il faut privilégier la qualité de la partition avant celle de la migration. Mais pour les applications très dynamiques qui nécessitent des repartitionnements fréquents (α faible), il faut mieux privilégier la migration devant la qualité de la partition qui se dégradera de toute façon rapidement.

Concernant la migration, différents critères peuvent être optimisés [44]. L'objectif le plus souvent optimisé est le volume total des données migrées, appelé *volume total de migration*. Mais il peut aussi être intéressant de minimiser le volume de migration par processeur, ou encore le nombre de messages nécessaires à la migration d'un point de vue global ou par processeur.

Les communications de migration peuvent être modélisées par une matrice que nous appellerons *matrice de migration* dans laquelle chaque élément représente le volume de données d'un message. Chaque ligne est associée à un processeur (et à son ancienne partie) et chaque colonne est associée à une nouvelle partie. L'élément (i, j) représente le volume de données que le processeur i enverra au processeur associé à la nouvelle partie j.

La figure 2.7 présente deux exemples de repartitionnement. La partition initiale sur la figure 2.7a est déséquilibrée : la partie 1 possède quatre sommets alors que la partie 3 n'en a que deux. Les figures 2.7b et 2.7c présentent deux nouvelles partitions et les matrices de migrations associées. Les sommets migrants (en rouge) correspondent aux éléments hors diagonale alors que les sommets ne migrant pas correspondent aux éléments de la diagonale (les messages qu'une partie « s'envoie » à elle-même). Par exemple, sur la figure 2.7c, la nouvelle partie 4 contient deux sommets de l'ancienne partie 4 et un de l'ancienne partie 1 : la quatrième colonne de la matrice contient donc un 1 sur la première ligne et un 2 sur la quatrième.

2.3.2 Méthodes de repartitionnement de graphe

Pour équilibrer la charge, il existe différentes approches basées sur le repartitionnement de graphe dont les plus courantes sont le *Scratch-Remap*, les méthodes diffusives et le partitionnement biaisé. Il existe d'autres approches basées des méthodes géométriques (bissections récursives ou SFC [52]) ou encore des méthodes spectrales modifiées pour prendre en compte la migration [23]. Nous allors maintenant présenter les trois approches les plus courantes.

a) Scratch-Remap

La méthode la plus simple et la plus évidente est le « Scratch-Remap ». Cette méthode se décompose en deux étapes. Dans un premier un temps, le graphe est partitionné « from scratch », c'est-à-dire sans prendre en compte l'ancienne partition. Ensuite les parties obtenues sont renumérotées pour optimiser la migration.

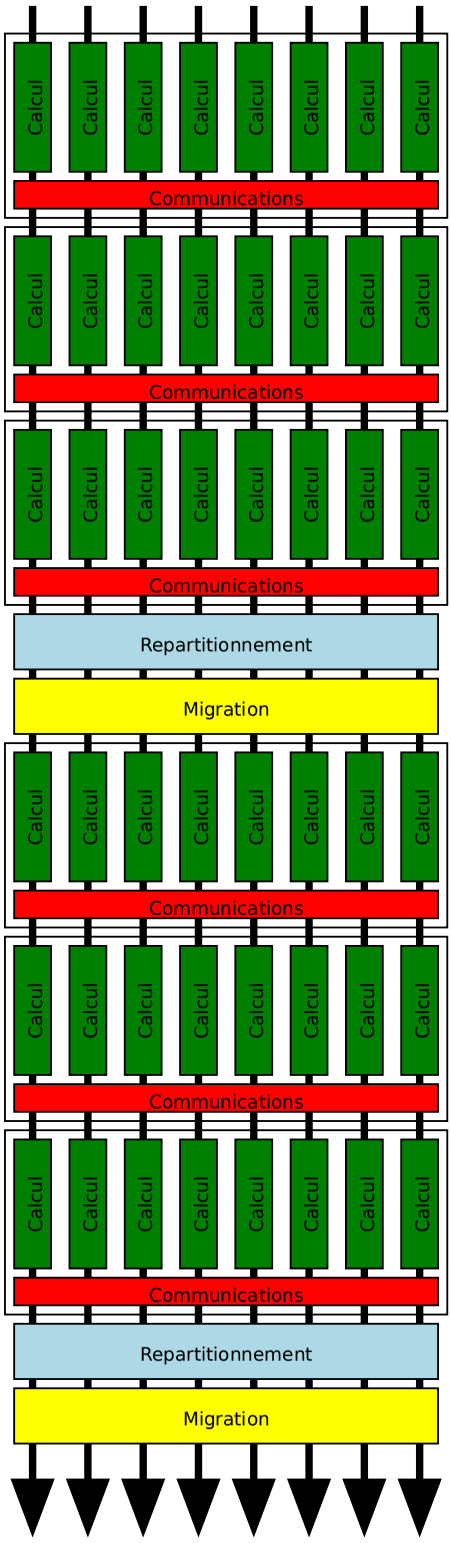
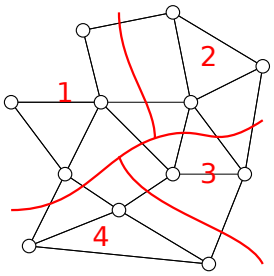
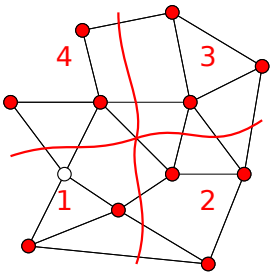


FIGURE 2.6 – Étapes d'un code se rééquilibrant toutes les $\alpha = 3$ itérations.

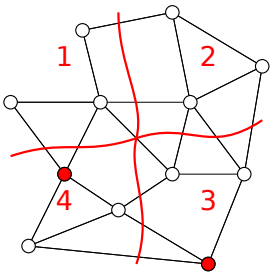


(a) Partition initiale déséquilibrée en 4 parties.



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

(b) Partition en 4 parties avec une mauvaise correspondance des parties.



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

(c) Même partition que précédemment mais avec une bonne correspondance.

FIGURE 2.7 – Exemples de repartitionnement. Les sommets migrants sont dessinés en rouge.

La figure 2.7 présente deux exemples de repartitionnement produisant la même nouvelle partition, mais avec une numérotation différente des nouvelles parties. Un repartitionnement « from scratch » peut tout à fait donner la partition de la figure 2.7b où tous les sommets sauf un sont migrés, alors qu'en renumérotant les nouvelles parties par rapport à l'ancienne partition comme sur la figure 2.7c, seulement deux sommets doivent être migrés.

Dans PLUM, Oliker et Biswas [45] proposent une heuristique gloutonne pour résoudre le problème d'association des nouvelles parties aux processeurs (*remap*). Cette heuristique est décrite par l'algorithme 1. Il parcourt les éléments $C_{i,j}$ de la matrice de migration du plus grand au plus petit élément. La partie j est associée au processeur i dès que c'est possible, c'est-à-dire si le processeur i est libre (*free*$[i]$ est vrai) et si la partie j n'est pas déjà associée à un processeur (*unassigned*$[j]$ est vrai). L'algorithme se termine lorsque toutes les parties ont été affectées à un processeur.

Comme la méthode *Scratch-Remap* recalcule une partition sans prendre en compte la migration, celle-ci possède généralement une bonne coupe car c'est le seul critère optimisé lors du partitionnement. En revanche, la migration n'est minimisée que dans un second temps et peut être très supérieure à l'optimal.

Il est possible d'apporter quelques améliorations au *Scratch-Remap*. Pour obtenir un volume de migration plus faible, Oliker et Biswas [45] repartitionne en kM parties au lieu de M. Lors de la phase de *remapping*, k nouvelles parties sont associées à chaque processeur. Ce grain plus fin permet un volume de migration moindre mais une coupe un peu plus élevée. Schloegel et al. [58] proposent une autre variante du *Scratch-Remap* appelée LMSR (*Locally Matched Scratch Remap*) dans laquelle la phase de contraction du partitionnement multi-niveaux est contrainte pour ne fusionner que des sommets appartenant à une même ancienne partie en espérant créer une nouvelle partition avec des frontières plus proches de l'ancienne partition.

b) Diffusion

Les méthodes diffusives s'inspirent du phénomène physique de diffusion de la chaleur pour rééquilibrer la charge. L'idée de base des méthodes diffusives est que chaque processeur échange

Algorithme 1 Réaffectation des nouvelles parties aux anciens processeurs (*remap*)

Entrée : Matrice de migration C de taille $M \times M$
map : vecteur d'association des nouvelles parties
unassigned : vecteur de booléens initialement vrais
free : vecteur de booléens initialement vrais
$n \leftarrow M$
$L \leftarrow$ liste triée des $C_{i,j}$ par ordre décroissant
tant que $n > 0$ **faire**
  Prendre et retirer le premier élément $C_{i,j}$ de L
  si *free*$[i] \wedge$ *unassigned*$[j]$ **alors**
    map$[j] \leftarrow i$
    free$[i] \leftarrow$ faux
    unassigned$[j] \leftarrow$ faux
    $n \leftarrow n - 1$
  fin si
fin tant que
retourner *map*

des sommets avec ses voisins selon le déséquilibre de charge entre les deux parties. Ces échanges locaux de sommets à la frontière des parties concernées permettent de rééquilibrer la charge.

Dans la version itérative de cette méthode, un rééquilibrage partiel est répété à chaque itération. Lors de ce rééquilibrage partiel, deux processeurs voisins échangent un nombre de sommets proportionnel à leur différence de charge. Cybenko [12] calcule la nouvelle charge $w_i^{(t+1)}$ d'une partie i à l'itération $t+1$, à l'aide la formule :

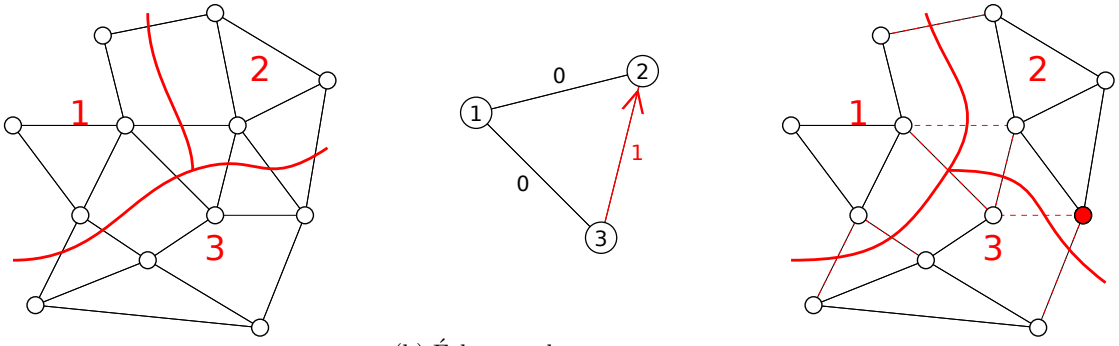
$$w_i^{(t+1)} = w_i^{(t)} + \sum_j \alpha_{ij}(w_j^{(t)} - w_i^{(t)}). \tag{2.1}$$

Les α_{ij} sont des coefficients positifs ou nuls et tels que $\forall i, \sum_j \alpha_{ij} \leq 1$ (chaque partie ne peut pas donner plus de charge qu'elle n'en possède), permettant de choisir si les parties i et j échangeront facilement des sommets. Cela permet de favoriser les échanges entre parties proches dans le graphe ou entre processeurs proches sur le réseau. Après plusieurs itérations, cette méthode tend à homogénéiser la charge, conformément au principe de diffusion de la chaleur.

Le rééquilibrage peut également être réalisé directement en cherchant la solution finale du problème de diffusion à l'aide d'un solveur. Hu et Blake [26] montrent que chercher la solution optimisant la norme 2 de la migration (la racine carré de la somme des carrés de la taille de chaque message) est équivalent au problème de diffusion. La solution est calculée à l'aide de la méthode du gradient conjugué.

Ou et Ranka [46] expriment le problème sous forme d'un programme linéaire pour minimiser la norme 1 de la migration (la somme des tailles de chaque message). Ce programme est résolu à l'aide de la méthode du *simplexe* [43].

Une fois que la charge à échanger entre chaque processeur a été décidée, il faut choisir quels seront les sommets à migrer. Ce choix est fait de façon à optimiser la coupe de la partition finale et peut être réalisé à l'aide d'un algorithme de type FM. Comme les déplacements de sommets s'effectuent entre des pairs de parties, il est nécessaire de trouver un bon ordonnancement de ses migrations [58]. En effet un processeur peut avoir besoin de transmettre plus de sommets qu'il n'en possède initialement et il a donc besoin de recevoir des sommets avant de pouvoir en envoyer.



(a) Partition déséquilibrée.

(b) Échanges de sommets entre parties.

(c) Partition finale rééquilibrée.

FIGURE 2.8 – Exemple de diffusion.

Un exemple de repartitionnement par diffusion est illustré sur la figure 2.8. Tous les sommets sont de poids unitaires. À partir de la partition initialement déséquilibrée (figure 2.8a), on calcule la migration optimale (figure 2.8b). Il faut alors déplacer un sommet de la partie 3 vers la partie 2. Ce sommet est choisi de façon à obtenir la meilleure coupe possible et ainsi obtenir la partition finale équilibrée de la figure 2.8c.

Les méthodes diffusives permettent d'obtenir de faibles volumes de migration. Mais les heuristiques locales utilisées se comportent mal si il y a de grandes variations de charge.

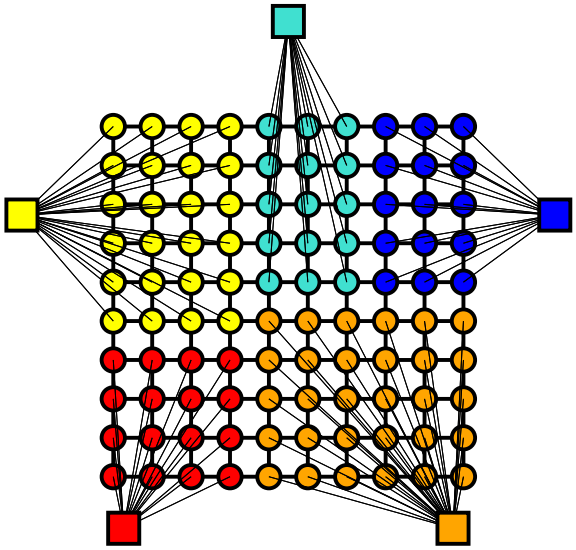
c) Partitionnement biaisé

Le partitionnement peut être biaisé pour optimiser la migration. Les heuristiques habituelles sont modifiées pour ne plus optimiser seulement la coupe mais un compromis entre la coupe et le volume total de migration.

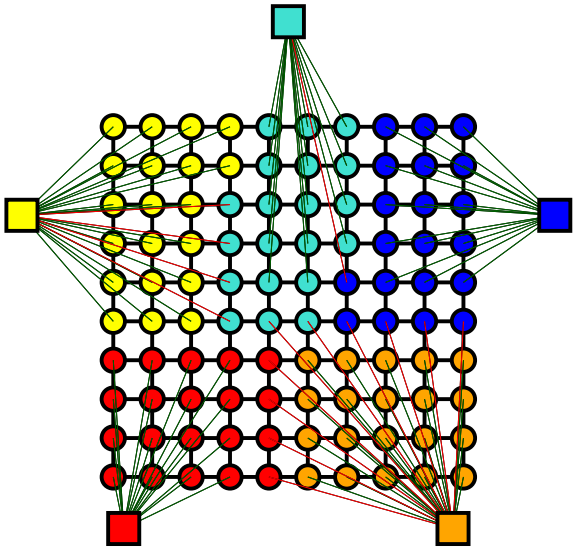
Au lieu de modifier les heuristiques, il est également possible de modifier le graphe à partitionner pour biaiser le partitionnement. Plusieurs études [2, 8, 9] montrent qu'il est possible de modéliser le problème de repartitionnement à l'aide de sommets fixes. Le graphe à partitionner est enrichi pour modéliser les objectifs du repartitionnement : la coupe des arêtes ajoutées représente la migration des sommets. Cette méthode est utilisée par le partitionneur d'hypergraphe *Zoltan* [9] et le partitionneur de graphe *RM-Metis* [2].

Des sommets sont ajoutés dans ce graphe pour modéliser les processeurs. Les processeurs ne sont pas des calculs à distribuer ; le poids de ces sommets est donc nul et ils sont fixés dans la partie du processeur qu'il modélise, c'est-à-dire que le partitionneur ne pourra pas les placer dans une autre partie. Ces sommets fixes sont ensuite connectés à tous les sommets contenus dans l'ancienne partie associée au sommet fixe. Ces nouvelles arêtes permettent de modéliser la migration et sont donc appelées *arêtes de migration*. En effet, si un sommet est migré, l'arête le connectant au sommet fixe de son ancienne partie sera coupée, ajoutant ainsi un coût de migration dans la taille de la coupe de la nouvelle partition. Le poids de l'arête de migration permet donc de représenter la taille des données à migrer.

Sur la figure 2.9, on peut voir un exemple de repartitionnement qui illustre cette méthode. La partition en 5 parties est initialement déséquilibrée ; 5 sommets fixes (carrés) sont ajoutés et reliés par des arêtes de migration aux sommets de leur partie respective (figure 2.9a). Après l'utilisation d'un partitionneur à sommets fixes, une nouvelle partition équilibrée est obtenue (figure 2.9b). Les arêtes de migration associées aux sommets ayant migré sont coupées (en rouge). La plus



(a) Graphe colorié selon l'ancienne partition auquel les sommets fixes ont été ajoutés.



(b) Graphe colorié selon la nouvelle partition. Les arêtes de migration coupées sont coloriées en rouge.

FIGURE 2.9 – Exemple de repartitionnement basé sur des sommets fixes.

grande partie des arêtes n'est pas coupée (en vert). Le partitionneur a donc créé une nouvelle partition avec peu de migration (15 sommets migrent correspondant aux 15 arêtes de migration coupées).

Scotch [18,19] utilise une méthode de repartitionnement biaisé : sa méthode de raffinement par « diffusion » est modifiée dans le cas du repartitionnement. Pour prendre en compte la migration des sommets, une nouvelle source de « liquide » est ajoutée pour chaque sommet l'incitant à rester dans sa partie d'origine.

Hendrikson et al. [23] proposent une autre approche du repartitionnement biaisé où chaque sommet u a un désir $d_k(u)$ d'être dans un partie k. Au lieu de minimiser le coût de migration, le but est de maximiser les désirs de chaque sommet (si $p(u)$ est la partie de u, on maximise la somme $\sum_u d_{p(u)}(u)$). Deux approches sont présentées : un raffinement de type FM modifié et une méthode de partitionnement spectrale prenant en compte les désirs des sommets.

Le partitionnement biaisé permet un compromis entre qualité de la nouvelle partition et volume de migration grâce à un coût de migration introduit dans la coupe à optimiser, par exemple, à l'aide du poids des arêtes de migration. Mais le repartitionnement est plus complexe car une nouvelle partition est complètement recalculée avec des heuristiques modifiées.

Le partitionnement biaisé peut également être hybride [57]. Dans le cadre d'un partitionnement multi-niveaux, le partitionnement initial est réalisé à l'aide d'une méthode *Scratch-Remap* ou diffusive, mais le raffinement appliqué lors de la phase d'expansion est biaisé pour optimiser la migration.

2.4 Positionnement

Toutes les méthodes de repartitionnement pour l'équilibrage dynamique présentées ici ne s'intéressent qu'au cas où le nombre de processeurs reste fixe. Le problème de l'équilibrage dynamique avec variation du nombre de processeurs est peu étudié. Iqbal et Carey [27,28] ont pourtant montré que faire varier le nombre de processeurs au cours de la simulation permet d'optimiser la consommation de ressources et même, dans certains cas, le temps d'exécution.

Bien choisir le nombre de processeurs alloués à une simulation est essentiel pour avoir une bonne performance ou efficacité. Si une simulation est lancée en parallèle sur un trop grand nombre de processeurs, le temps passé à communiquer peut devenir trop important par rapport au temps de calcul. Utiliser le maximum de processeurs possible n'est pas toujours un bon choix suivant la taille du problème. La taille du problème pouvant varier au cours de la simulation, le nombre de processeurs devrait varier en conséquence.

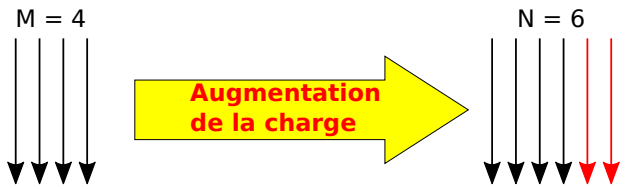


FIGURE 2.10 – Allocation dynamique de processeurs pour un seul code dont la charge augmente.

Par exemple, un code AMR (*Adaptive Mesh Refinement*) commence sa simulation sur un maillage grossier contenant peu d'éléments. Puis, au cours de la simulation, le maillage est raf-

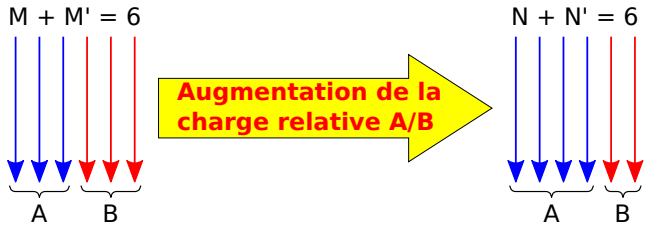


FIGURE 2.11 – Allocation dynamique de processeurs pour deux codes couplés A et B dont la charge relative de A par rapport à B augmente.

finé là où c'est nécessaire. La charge de calcul, initialement très faible, croît avec le temps. La simulation très légère au début finit par demander beaucoup de ressources, que ce soit en puissance de calcul ou en mémoire. En commençant le calcul sur très peu de processeurs puis en l'augmentant quand la limite mémoire est atteinte, il est possible de garder une meilleure efficacité tout le long de la simulation. Les surcoûts dus au parallélisme inutile en début de simulation sont évités quand c'est possible. C'est la cas présenté sur la figure 2.10 : après une augmentation significative de la charge, deux processeurs supplémentaires sont alloués, et il faut donc effectuer un repartitionnement de 4 processeurs vers 6.

L'allocation dynamique de processeurs peut aussi être utile dans le cas du couplage de codes. Dans un couplage de codes, plusieurs codes parallèles s'exécutent simultanément et doivent régulièrement échanger des données. Cette phase d'échange est synchronisante. Il est donc important que tous les codes concernés progressent à la même vitesse pour minimiser le temps d'attente lors de cette synchronisation. Le choix du nombre de processeurs utilisés par chaque code doit être fait en prenant en compte les charges relatives de chacun. Cette équilibrage entre plusieurs codes peut être difficile, plus particulièrement si la charge de ces codes peut varier dynamiquement. Pour rééquilibrer ces codes, une solution serait donc de réallouer des ressources d'un code vers un autre. Le nombre idéal de processeurs alloués à chaque code peut donc être approché expérimentalement en corrigeant au cours de la simulation les déséquilibre qui surviendraient. Par exemple, sur la figure 2.11, le code A est devenu trop lent par rapport à B, un processeur est donc réalloué du code B vers le code A. Il faut repartitionner chacun des deux codes : de 3 processeurs vers 4 pour A, et de 3 vers 2 pour B.

Pour réaliser ce changement du nombre de ressources, il est nécessaire de mettre en place une méthode repartitionnement prenant en compte le changement du nombre de parties. Nous appelons ce problème, le repartitionnement $M \times N$. Les objectifs du repartitionnement sont similaires à ceux du repartitionnement classiques :

— équilibrer la charge parmi les N processeurs ;
— minimiser la coupe du graphe ;
— calculer rapidement la nouvelle partition ;
— minimiser la migration (différentes métriques seront présentées).

Les outils de repartitionnement suivent déjà ces objectifs mais ne sont pas prévus pour le changement du nombre de processeurs. La méthode de *Scratch-Remap* est facilement adaptable au repartitionnement $M \times N$. Lors du partitionnement *from scratch*, l'ancienne partition, et donc l'ancien nombre de processeurs, n'a pas d'influence. L'algorithme de *Remapping* peut être facilement adapté en prenant en compte le fait que certaines parties ne seront pas associées à un ancien processeur (lors de l'ajout de processeurs), ou que des anciens processeurs ne recevront pas de nouvelles parties (lors de la suppression de processeurs).

Nous proposons donc une nouvelle méthode de repartitionnement adaptée au cas où le nombre de processeurs varie. La méthode proposée est présentée sur la figure 2.12. Elle se décompose en deux étapes.

Dans un premier temps, nous cherchons à construire des matrices de migrations adaptées à ce problème. Après une étude de quelques matrices optimales, plusieurs méthodes seront présentées dans le chapitre 3. La construction de la matrice de migration ne s'intéresse pas aux sommets du graphe individuellement mais seulement aux parties, à leurs tailles et à leurs positions respectives. Ainsi, la construction de la matrice s'effectue à partir du graphe quotient des parties.

Ensuite, la matrice de migration construite est utilisée pour guider un repartitionnement du graphe remplissant les objectifs habituels du partitionnement de graphe : l'équilibre et la coupe tout en fournissant une migration telle que prévue par la matrice construite à l'étape précédente. Plusieurs approches sont possibles : un partitionnement biaisé utilisant des sommets fixes pour imposer une matrice de migration lors du repartitionnement ; une approche diffusive adaptée pour le repartitionnement $M \times N$; et un partitionnement biaisé utilisant des hyper-arêtes pour optimiser les messages de migration sans utiliser de matrice de migration calculée à l'avance.

Les méthodes de partitionnement biaisé mettent en évidence les limites des bissections récursives utilisées dans les partitionneurs usuels. Un module de partitionnement k-aire utilisant une heuristique de *greedy graph growing* est ajoutée dans le partitionneur *Scotch* pour améliorer la qualité de nos partitionnements.

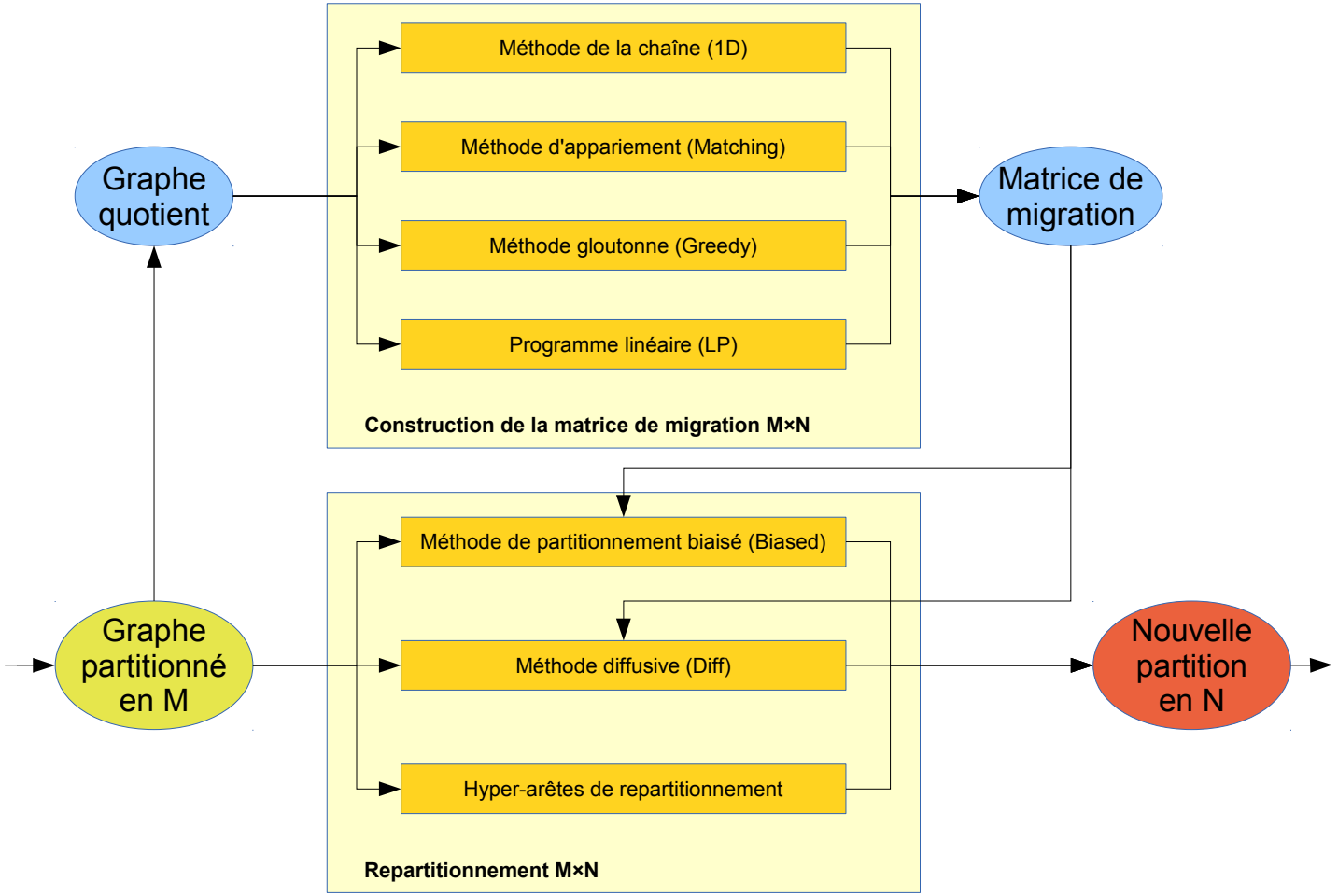


FIGURE 2.12 – Vue d'ensemble de nos méthodes de repartitionnement $M \times N$.

Chapitre 3

Modèle de migration pour le repartitionnement $M \times N$

Sommaire

3.1 Notations et définitions 25
3.1.1 Généralités 26
3.1.2 Définitions pour le repartitionnement $M \times N$ 27
3.2 Matrices de migration optimales dans le cas équilibré 28
3.3 Construction des matrices de migration 35
3.3.1 Méthode basée sur la chaîne (*1D*) 35
3.3.2 Méthode d'appariement (*Matching*) 42
3.3.3 Méthode gloutonne (*Greedy*) 47
3.3.4 Programme linéaire (*LP*) 58
3.4 Évaluation des méthodes et conclusion 62

La première étape de notre méthode de repartitionnement consiste à construire une matrice de migration optimisée. Pour réaliser cela, il n'est pas nécessaire de disposer de toute l'information sur le graphe partitionné mais seulement du *graphe quotient*. Nous proposons plusieurs algorithmes permettant de construire des matrices de migration optimisant différents critères. Cette étape est résumée en figure 3.1.

Dans un premier temps, nous énoncerons quelques définitions générales sur le graphe et le partitionnement, puis des définitions particulières pour nos algorithmes de repartitionnement. Ensuite, nous étudierons plus en détails les matrices de migration dans le cas où l'ancienne partition est déjà équilibrée. Enfin, nous décrirons nos algorithmes de construction de matrice de migration, avant de conclure sur une comparaison de ces différents méthodes.

3.1 Notations et définitions

Dans cette section, nous définissons les notations et termes qui seront utilisés dans cette thèse. Nous rappelons d'abord les définitions relatives au partitionnement, puis nous introduisons les nouvelles notations que nous utiliserons pour le repartitionnement $M \times N$.

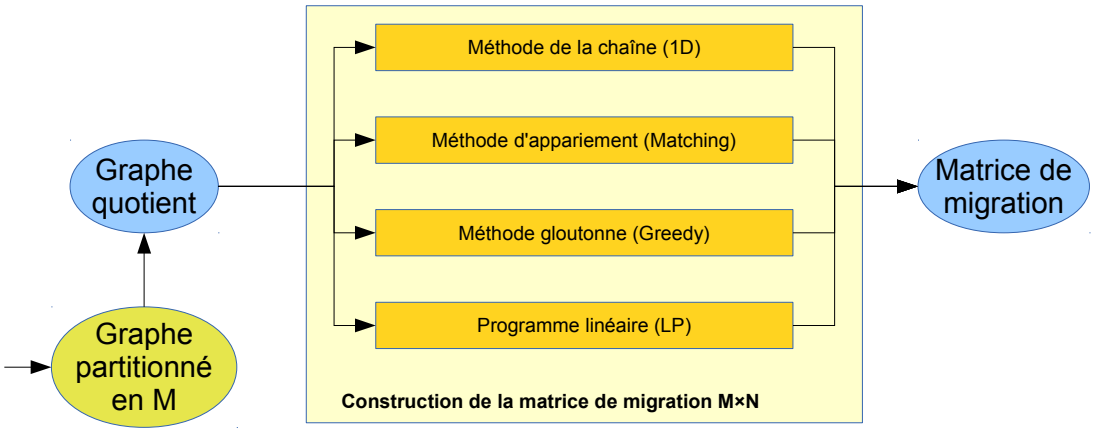


FIGURE 3.1 – Vue d'ensemble de nos algorithmes de construction des matrices de migration.

3.1.1 Généralités

Soit $G = (V, E)$, un graphe où V est l'ensemble des sommets et E l'ensemble des arêtes. À chaque sommet v est associé un poids w_v représentant la charge de calcul associé à ce sommet. Une arête e peut avoir un poids w_e représentant le coût de la communication. On notera $W = \sum_{v \in V} w_v$ le poids total du graphe.

Définition 1 (Partition)**.** Une k-partition P de V est un ensemble de k sous-ensembles deux à deux disjoints $(P_i)_{i \in [\![1,k]\!]}$ de V tels que l'union de tous les sous-ensembles couvre V.

Définition 2 (Poids d'une partie)**.** Le poids d'une partie P_i noté $w(P_i)$ est la somme des poids des tous les sommets de la partie : $w(P_i) = \sum_{v \in P_i} w_v$.

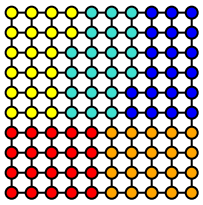
Définition 3 (Partition équilibrée)**.** La k-partition P est dite équilibrée au facteur de déséquilibre $\epsilon \geq 0$ près si $\forall P_i \in P, w(P_i) \leq (1 + \epsilon)\frac{W}{k}$.

En pratique, le facteur de déséquilibre souvent utilisé est de l'ordre de quelques pour cent. Avec un facteur de 5 % ($\epsilon = 0,05$), une partie qui aurait une taille idéale de 1000 éléments, ne devra pas dépasser 1050.

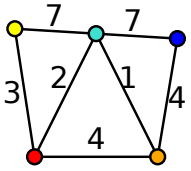
Définition 4 (Coupe d'un graphe)**.** On appelle taille de la coupe (ou par abus de langage, coupe) d'un graphe pour une partition donnée, le nombre d'arêtes coupées ou la somme de leurs poids si elles sont pondérées. Les arêtes coupées sont les arêtes dont les deux extrémités sont dans des parties différentes.

Définition 5 (Graphe quotient)**.** La graphe quotient $Q = G/P$ d'un graphe G par rapport à la partition P est obtenu à partir de G en fusionnant les sommets appartenant à une même partie de P. Les arêtes devenues identiques après la fusion des sommets sont également fusionnées et leurs poids sont additionnés. Si les arêtes n'étaient pas pondérées, le poids des arêtes du graphe quotient est le nombre d'arêtes fusionnées.

Le graphe quotient $Q = G/P$ possède autant de sommets qu'il y a de parties dans P. Le poids de chaque sommet est le poids de la partie correspondante (la somme des poids des sommets de cette partie). Les sommets sont connectés aux sommets correspondant aux parties voisines et le poids des arêtes est la somme des poids des arêtes coupées entre ces deux parties.



(a) Partition d'un graphe en 5 parties.



(b) Graphe quotient correspondant à la partition.

FIGURE 3.2 – Exemple de construction d'un graphe quotient pour une grille 10×10.

La figure 3.2 montre la partition d'un graphe issu d'une grille 2D de taille 10×10 et le graphe quotient associé. La partition est en 5 parties et le graphe quotient possède 5 sommets coloriés de la même couleur que la partie à laquelle il est associé. La partition étant parfaitement équilibrée, le poids de chaque sommet est égal, et vaut 20. Le poids des arêtes est plus variable : par exemple, l'arête entre le sommet jaunes et cyan a un poids 7, comme il y a 7 arêtes dans le graphe partitionné connectant des sommets jaunes et cyan.

3.1.2 Définitions pour le repartitionnement $M \times N$

Dans le cas du repartitionnement, on distinguera deux partitions : la partition initiale P en M parties et une partition finale désirée P' en N parties.

Définition 6 (Matrice de migration)**.** On appelle matrice de migration, la matrice C de dimension $M \times N$ décrivant les volumes de données à migrer entre les différents parties. L'élément $C_{i,j}$ correspond au nombre d'éléments en commun entre l'ancienne partie i et la nouvelle partie j. C'est aussi la quantité de données envoyée par le processeur i au processeur j si $i \neq j$; si $i = j$, $C_{i,i}$ est la quantité de données ne migrant pas, c'est-à-dire restant « sur place ».

Chaque ligne représente une ancienne partie, donc la somme des éléments sur cette ligne est égale à sa taille. De la même façon, une colonne représente une nouvelle partie et la somme des éléments de chaque colonne est égale à sa taille.

Pour évaluer la qualité des matrices de migration, nous définissons plusieurs métriques : TOTALV et MAXV sont déjà introduits par Oliker et al. [45] ; nous y ajoutons deux nouvelles métriques TOTALZ et MAXZ.

Le volume total de migration, noté TOTALV, est la quantité de sommets qui migrent (ou la somme de leurs poids s'il sont pondérés). Cela correspond donc à la somme des éléments non-diagonaux de la matrice de migration. Le volume maximum par processeur, noté MAXV, correspond à la plus grande quantité de sommets (ou somme des poids) reçus et envoyés par un même partie. Il correspond à la plus grande somme sur une ligne et une colonne de mêmes numéros des éléments hors diagonale de la matrice de migration.

Le nombre total de messages nécessaires à la migration, noté TotalZ, correspond au nombre de non-zéros hors diagonale. De la même façon, on définit le nombre maximal de messages (envoyés et reçus) par processeur, noté MaxZ.

TotalZ vaut au plus $M \times N - \min(M, N)$ dans le cas où tous les éléments non-diagonaux de C sont non nuls.

La figure 3.3 montre plusieurs cas de repartitionnement de la partition initiale de la figure 3.3a. Pour chaque repartitionnement, la matrice de migration et les différentes métriques sont données. Les figures 3.3b et 3.3c montrent la même nouvelle partition mais avec une numérotation différente. Il est important de bien faire correspondre les anciennes et nouvelles parties pour avoir une faible migration. En effet, le volume total de migration TotalV est bien plus important dans le cas de la figure 3.3b. La figure 3.3d montre une partition différente ayant une meilleure coupe mais une migration plus élevée.

Définition 7 (Graphe biparti de migration)**.** Le graphe biparti de migration est le graphe biparti $\mathcal{G} = ((A, B), E)$ où A est l'ensemble des M anciennes parties et B l'ensemble des N nouvelles parties. Il existe une arête dans E entre une ancienne et une nouvelle partie si elles partagent des sommets. Le poids de cette arête est la somme des poids des sommets partagés.

La matrice d'adjacence de ce graphe biparti est la matrice de migration C. Il existe une arête entre les sommets i et j si et seulement si $C_{i,j}$ est non nul et si le poids de cette arête vaut $C_{i,j}$. Des exemples de graphes biparti de migration sont présentés sur la figure 3.3. Chaque arête du graphe représente soit un message de migration, soit les données qui restent sur place lorsque l'arête relie une ancienne partie à sa nouvelle partie associée.

Définition 8 (Hypergraphe de repartitionnement)**.** L'hypergraphe de repartitionnement est l'hypergraphe comportant M sommets correspondant aux anciennes parties et N hyper-arêtes correspondant aux nouvelles parties. Une hyper-arête contient tous les sommets correspondant aux anciennes parties avec lesquelles la nouvelle partie échange des sommets. Les hyper-arêtes ne sont pas pondérées.

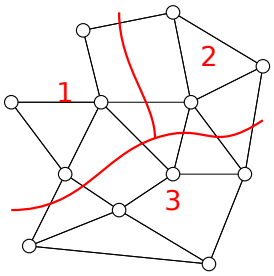
La matrice d'adjacence de cet hypergraphe est la matrice de migration sans prendre en compte les valeurs : on ne s'intéresse qu'aux zéros et non-zéros. Le sommet v appartient à l'hyper-arête e si et seulement si $C_{v,e}$ est non nul. Chaque ligne de la matrice ou ancienne partie correspond à un sommet de l'hypergraphe et chaque colonne ou nouvelle partie correspond à une hyper-arête. Une hyper-arête contient tous les sommets dont l'élément dans la matrice est non nul.

Les hypergraphes de repartitionnement correspondant à chaque matrice de migration sont donnés dans la figure 3.3. La renumérotation des parties entre les figures 3.3b et 3.3c ne change pas l'hypergraphe de repartitionnement mais seulement les numéros des hyper-arêtes.

L'utilisation de ces graphes bipartis et hypergraphes permet de représenter le schéma de communication utilisé lors de la migration. Les hypergraphes de repartitionnement permettent une représentation plus graphique des relations entre une ancienne et une nouvelle partition : c'est ce que l'on constate par exemple sur la figure 3.4 en le superposant avec le graphe quotient de l'ancienne partition.

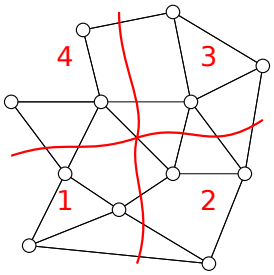
3.2 Matrices de migration optimales dans le cas équilibré

Dans un premier temps, nous allons étudier les matrices de migration dans le cas particulier où la partition initiale est déjà parfaitement équilibrée. Plus précisément, on considère que les partitions initiales et finales sont parfaitement équilibrées. Dans ce cas, W doit être un multiple commun de M et de N. On a donc $W = k \times \text{ppcm}(M, N)$ avec k un entier strictement positif.



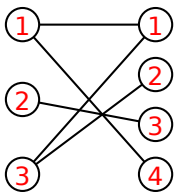
Coupe : 8

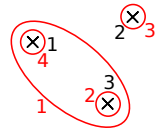
(a) Partition initiale déséquilibrée en 3 parties.



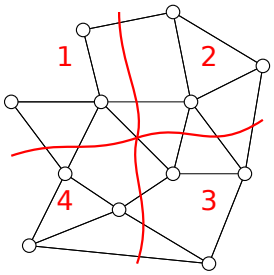
Coupe : 11
TotalV : 11
MaxV : 8
TotalZ : 4
MaxZ : 3

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



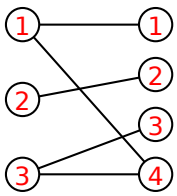


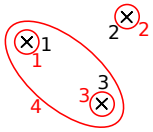
(b) Partition en 4 parties avec une mauvaise correspondance des parties.



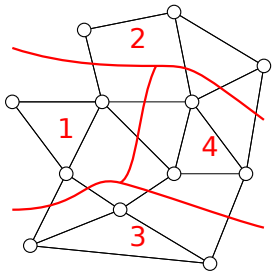
Coupe : 11
TotalV : 3
MaxV : 3
TotalZ : 2
MaxZ : 2

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



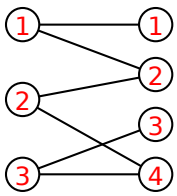


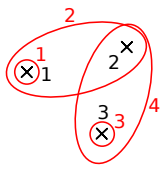
(c) Même partition que précédemment mais avec une bonne correspondance.



Coupe : 10
TotalV : 4
MaxV : 3
TotalZ : 3
MaxZ : 2

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





(d) Autre exemple de repartitionnement en 4 parties.

Figure 3.3 – Exemples de repartitionnement avec les matrices de migration, graphes bipartis de migration, hypergraphes de repartitionnement et mesures associées.

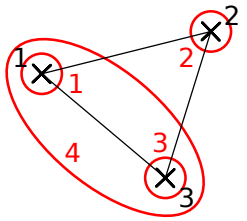


FIGURE 3.4 – Superposition de l'hypergraphe de repartitionnement de la figure 3.3c et du graphe quotient de la partition de la figure 3.3a.

Pour simplifier notre discussion, on supposera que les poids des sommets sont unitaires. Ainsi le problème de la partition du graphe est équivalent à la partition de l'entier correspondant à son poids.

Définition 9 (Matrice de migration optimale)**.** Nous appelons matrice de migration optimale, une matrice de migration C minimisant à la fois le volume de total de migration (TOTALV) et le nombre de messages (TOTALZ).

Lemme 1. *Le volume total de migration optimal vérifie :*

$$\text{TOTALV}_{opt} \geq W\left(1 - \frac{\min(M,N)}{\max(M,N)}\right).$$

Démonstration. Il y a $\min(M,N)$ parties communes entre les partitions initiale et finale et $|M-N|$ parties différentes. Comme les partitions sont équilibrées, les tailles des parties initiales et finales sont respectivement $\frac{W}{M}$ et $\frac{W}{N}$. Donc au plus, $\min(\frac{W}{M},\frac{W}{N}) = \frac{W}{\max(M,N)}$ données restent en place pour chaque partie commune. Le reste des données migre, donc $\text{TOTALV}_{opt} \geq W - \min(M,N) \times \frac{W}{\max(M,N)} = W \times (1 - \frac{\min(M,N)}{\max(M,N)})$. □

Lemme 2. *Une matrice de migration minimisant le nombre de messages possède au moins* $M + N - \text{pgcd}(M,N)$ *coefficients non nuls. Le nombre minimal de messages vérifie :*

$$\text{TOTALZ}_{opt} \geq \max(M,N) - \text{pgcd}(M,N).$$

Démonstration. Le volume total de données est $W = k \times \text{ppcm}(M,N) = kaM = kbN$ avec a et b deux entiers strictement positifs premiers entre eux. Chaque partie possède initialement ka sommets et finalement kb sommets.

Soit $\mathcal{G} = ((A,B),E)$ le graphe biparti de migration. Pour dénombrer les éléments non nuls dans la matrice de migration, il suffit de compter le nombre d'arêtes dans le graphe biparti de migration.

On note $(\mathcal{G}_i = ((A_i,B_i),E_i))_{i\in[\![1,K]\!]}$, les K composantes connexes de $\mathcal{G}$. Pour chaque composante $\mathcal{G}_i$, $M_i = |A_i|$ processeurs envoient $M_i ka$ sommets à $N_i = |B_i|$ processeurs qui reçoivent $N_i kb$ sommets. Dans $\mathcal{G}_i$, on a donc $W_i = M_i ka = N_i kb$ sommets qui sont échangés, avec M_i et N_i non nuls. Comme W_i est un multiple commun de ka et kb et comme a et b sont premiers entre eux, le plus petit multiple commun de ka et kb est kab. Donc $W_i \geq kab$. Le volume total échangé $W = \sum_{i=1}^{K} W_i$ est donc supérieur ou égal à $Kkab$. Autrement dit, $K \leq \frac{W}{kab} = \frac{M}{b}$.

D'autre part, on sait que $MN = \text{pgcd}(M,N) \times \text{ppcm}(M,N)$ et $\text{ppcm}(M,N) = bN$. On a donc $\frac{M}{b} = \text{pgcd}(M,N)$. On obtient alors $K \leq \text{pgcd}(M,N)$.

Comme $\mathcal{G}_i$ est un graphe connexe, il possède au moins $M_i + N_i - 1$ arêtes. Donc le nombre total d'arêtes $|E| = \sum_{i=1}^{K} |E_i|$ est au moins $\sum_{i=1}^{K} M_i + \sum_{i=1}^{K} N_i - K = M + N - K$. Comme $K \leq \text{pgcd}(M,N)$, on obtient $|E| \geq M + N - \text{pgcd}(M,N)$. Le nombre de coefficients non nuls dans la matrice de migration est donc au moins $M + N - \text{pgcd}(M,N)$.

TotalZ est le nombre de coefficients non nuls hors de la diagonale. Il y a au plus $\min(M,N)$ coefficients non nuls sur la diagonale. Donc $\text{TotalZ} \geq M + N - \text{pgcd}(M,N) - \min(M,N) = \max(M,N) - \text{pgcd}(M,N)$. □

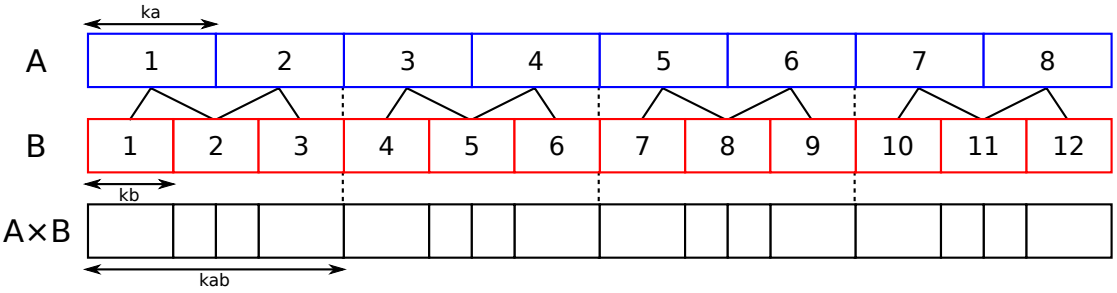


Figure 3.5 – Partitionnement d'un graphe chaîne en 8 et 12 parties, le motif de l'intersection $A \times B$ permet de construire la matrice de migration « en escalier ».

Il existe différentes méthodes pour obtenir de telles matrices avec un nombre minimal d'éléments non nuls.

Une première méthode s'inspire du partitionnement d'un graphe « chaîne » ou d'un tableau, comme présenté sur la figure 3.5. Pour compter le nombre de messages, il faut compter le nombre d'arêtes du graphe biparti. En reprenant les notations de la démonstration du lemme 2, la taille des parties de A est ka et celle des parties de B est kb. Tous les $\text{ppcm}(ka,kb) = kab$ sommets, les séparateurs des anciennes parties et des nouvelles parties sont juxtaposés. Le graphe biparti de migration est coupé en sous-graphes (composantes connexes), chacun avec b anciennes parties de taille ka et a nouvelles parties de taille kb. Ce motif de longueur kab est donc répété $\frac{W}{kab} = \frac{M}{b} = \frac{N}{a} = \text{pgcd}(M,N)$ fois. Chacun de ces sous-graphes contient $b + a - 1$ arêtes. Le nombre total d'arêtes est donc $(b + a - 1) \times \text{pgcd}(M,N) = M + N - \text{pgcd}(M,N)$. Sur l'exemple de la figure 3.5, on a $\text{ppcm}(8,12) = 24$ donc $a = 3$ et $b = 2$. Chacune des $\text{pgcd}(8,12) = 4$ composantes connexes contient $3 + 2 - 1 = 4$ arêtes. Le graphe biparti contient 16 arêtes. Cette construction donne une matrice « en escalier » comme montré sur la figure 3.7a.

Il est également possible de construire une telle matrice récursivement. On remplit la matrice avec des blocs qui sont des matrices carrées diagonales de taille $\min(M,N)$, et le bloc restant est rempli en répétant la même méthode sur le nouveau sous-problème. Pour que la matrice soit une matrice de migration, il faut que les sommes sur les lignes ou colonnes soient respectivement les tailles des anciennes et nouvelles parties. Les valeurs diagonales sont donc $\frac{W}{\max(M,N)}$. La figure 3.6 donne un exemple de telle construction dans le cas 7×10 avec $W = 7 \times 10$ en trois étapes. Calculer le nombre de non-zéros avec cette méthode, revient à appliquer l'algorithme d'Euclide. La division euclidienne de M par N donne $M = qN + r$: on commence à remplir q blocs carrés de taille $N \times N$ et on recommence récursivement sur le bloc restant de taille $r \times N$

avec $r < N$. L'algorithme d'Euclide donne une suite d'équations :

$$M = q_2 N + r_2 \tag{3.1}$$
$$\vdots \tag{3.2}$$
$$r_{i-2} = q_i r_{i-1} + r_i \tag{3.3}$$
$$\vdots \tag{3.4}$$
$$r_{n-3} = q_{n-1} r_{n-2} + r_{n-1} \tag{3.5}$$
$$r_{n-2} = q_n r_{n-1} \tag{3.6}$$

avec $r_0 = M$, $r_1 = N$ et $r_{n-1} = \text{pgcd}(M, N)$. À chaque étape de l'algorithme, $q_i r_{i-1}$ éléments non nuls sont ajoutés dans la matrice : ce sont les éléments diagonaux des q_i blocs carrés. Le nombre total d'éléments non nuls est donc $\sum_{i=2}^{n} q_i r_{i-1}$. En faisant la somme de toutes les équations précédentes, on obtient :

$$\sum_{i=0}^{n-2} r_i = \sum_{i=2}^{n} q_i r_{i-1} + \sum_{i=2}^{n} r_i \tag{3.7}$$
$$\sum_{i=2}^{n} q_i r_{i-1} = r_0 + r_1 - r_{n-1} - r_n \tag{3.8}$$
$$\sum_{i=2}^{n} q_i r_{i-1} = M + N - \text{pgcd}(M, N). \tag{3.9}$$

D'après l'équation 3.9, cette méthode construit bien une matrice avec un nombre minimal de non-zéros. La figure 3.7d présente un exemple d'une telle matrice.

Théorème 3. *Les bornes inférieures données dans les lemmes 1 et 2 sont atteintes. C'est-à-dire :*

$$\textsc{TotalV}_{opt} = W\left(1 - \frac{\min(M, N)}{\max(M, N)}\right) \tag{3.10}$$
$$\textsc{TotalZ}_{opt} = \max(M, N) - \text{pgcd}(M, N). \tag{3.11}$$

Démonstration. La méthode de construction basée sur l'algorithme d'Euclide construit une matrice de migration avec $M + N - \text{pgcd}(M, N)$ coefficients non nuls. Les $\min(M, N)$ éléments sur la diagonale sont maximisés et valent $\frac{W}{\max(M,N)}$. On a donc $\textsc{TotalV} = W\left(1 - \frac{\min(M,N)}{\max(M,N)}\right)$ et $\textsc{TotalZ} = \max(M, N) - \text{pgcd}(M, N)$. □

Proposition 4. *Soient M et N deux entiers tels que $M < N$. Soient D une matrice diagonale de dimension $M \times M$ dont les éléments diagonaux sont $\frac{W}{N}$ et A une matrice de migration de dimension $M \times (N - M)$ minimisant le nombre de messages. Alors, la matrice $\begin{pmatrix} D & A \end{pmatrix}$ est une matrice de migration optimale au sens de la définition 9.*

De même, dans le cas où $M > N$, si D est de taille $N \times N$ avec des éléments diagonaux valant $\frac{W}{M}$ et si A est de taille $(M - N) \times N$, alors $\begin{pmatrix} D \\ A \end{pmatrix}$ est une matrice de migration optimale.

Démonstration. La démonstration est faite dans le cas $M < N$. La démonstration du cas $M > N$ est similaire.

La quantité de données qui reste sur place correspond à la première diagonale de la matrice, soit D. Donc, on a $W\frac{M}{N}$ données qui restent sur place, ce qui est bien le cas optimal.

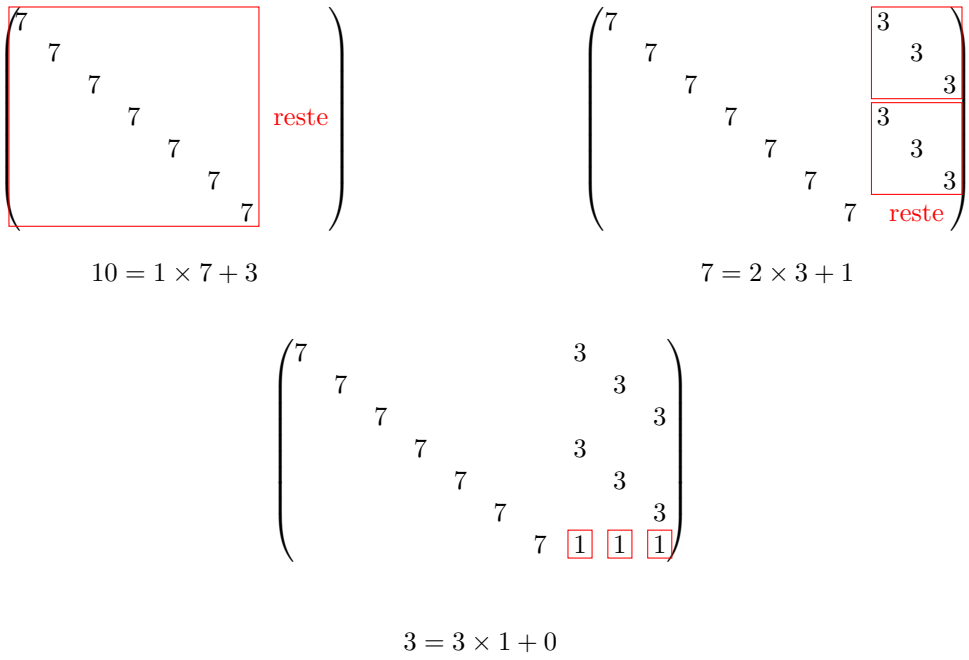


FIGURE 3.6 – Exemple de construction récursive (avec les divisions euclidiennes associées à chaque étape) d'une matrice de migration avec un nombre minimal de communications.

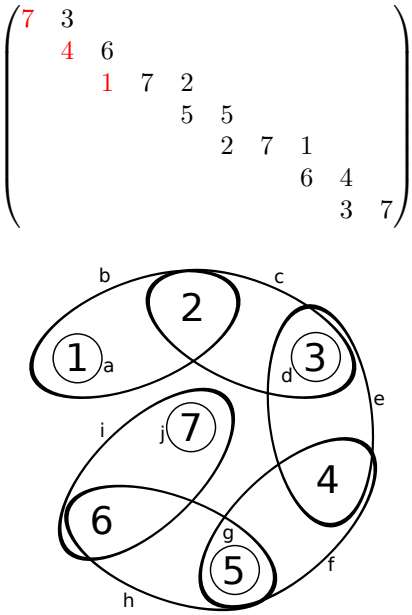
D contient M éléments non nuls et A contient par hypothèse un nombre minimal d'éléments non nuls pour sa taille : $M + (N - M) - \text{pgcd}(M, N - M) = N - \text{pgcd}(M, N)$. Le nombre total d'éléments non nuls est donc $M + N - \text{pgcd}(M, N)$ qui est bien le nombre minimal.

La matrice $\begin{pmatrix} D & A \end{pmatrix}$ remplit bien les deux critères pour être optimale. □

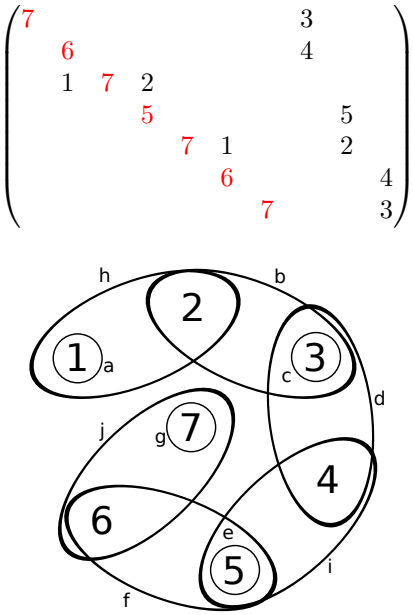
La figure 3.7 présente quelques exemples de matrices de migration. La matrice de la figure 3.7a a été obtenue par la méthode inspirée du repartitionnement d'une chaîne qui donne une forme en escalier. Le nombre de messages est optimal mais pas le volume de migration. Seules 12 données restent sur place (les nombres en rouges) contre $\left(\frac{W}{M} \times N\right) = 49$ dans le cas d'une migration optimale. Le volume de migration peut être amélioré à l'aide d'une simple renumérotation des parties similaires au « remapping » de la méthode *Scratch-Remap* (cf. section 2.3). La matrice obtenue est présentée sur le figure 3.7b, 45 données restent alors sur place, ce qui est très proche de l'optimal.

D'après le théorème précédent, il nous suffit de combiner une matrice diagonale et une matrice minimisant le nombre de message pour obtenir une matrice optimale. La figure 3.7c montre la combinaison d'une matrice diagonale et d'une matrice en escalier. Dans le cas de la figure 3.7d, on construit récursivement une matrice optimale à l'aide de matrices diagonales en se basant sur l'algorithme d'Euclide. Bien que ces deux matrices soient optimales pour les métriques désirées (TOTALV et TOTALZ), la matrice construite à partir de la matrice en escalier donne un nombre de messages par processeur (MAXZ) plus faible que la méthode récursive.

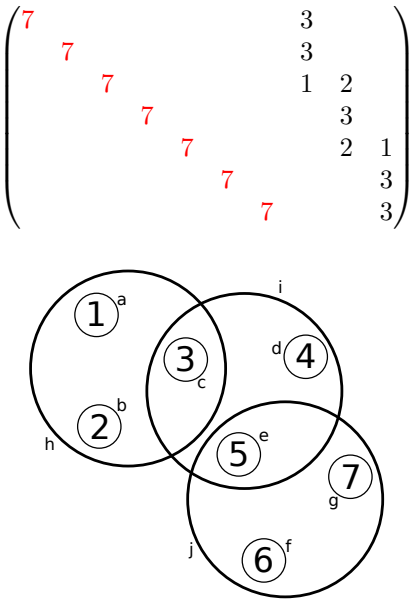
En pratique, les partitions ne sont pas parfaitement équilibrées car cela n'est pas nécessaire : on se contente d'une partition à peu près équilibrée à un facteur de déséquilibre près. Cette équilibre parfait peut même être impossible lorsque W n'est pas divisible par M ou N ou que le



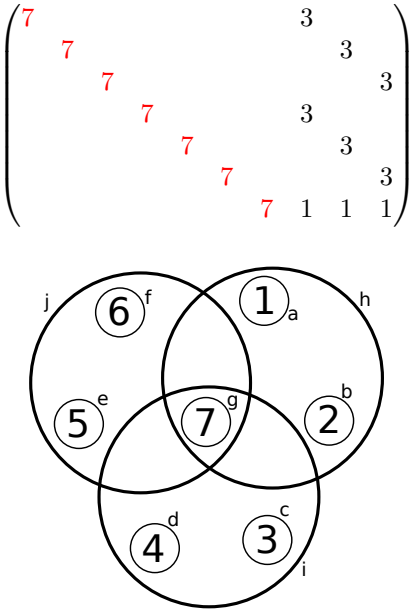
(a) Matrice « en escalier ».



(b) Matrice « en escalier » avec permutation des colonnes pour minimiser TotalV.



(c) Matrice optimale utilisant une sous-matrice en escalier.



(d) Exemple de matrice optimale construite par ajout de blocs diagonaux récursivement.

FIGURE 3.7 – Exemples de matrices de migration pour le cas 7×10 et leurs représentations en hypergraphe de repartitionnement. Les éléments nuls ne sont pas montrés. Les lignes numérotées de 1 à 7 correspondent aux sommets et les colonnes numérotées de a à j correspondent aux hyper-arêtes. Les éléments en rouge montrent les données qui restent en place. Les éléments en noir sont ceux qui migrent, la somme de ces éléments donne le volume de migration.

poids des sommets de permet pas un découpage en parties rigoureusement égales. Les résultats présentés dans cette section sur le nombre de messages restent vrais pour des déséquilibres faibles. En effet, il est possible de profiter de la tolérance au déséquilibre des nouvelles parties pour mieux s'adapter aux anciennes parties légèrement déséquilibrées et économiser des messages. Le volume de migration optimal reste proche.

Dans le cas où la partition initiale est fortement déséquilibrée, il est difficile de donner des valeurs optimales pour tous les cas. Il est toujours possible d'effectuer la migration en $\max(M, N)-1$ messages (le nombre minimal d'arêtes d'un graphe biparti connexe étant $M + N - 1$), mais ce nombre n'est pas toujours minimal.

3.3 Construction des matrices de migration

3.3.1 Méthode basée sur la chaîne (*1D*)

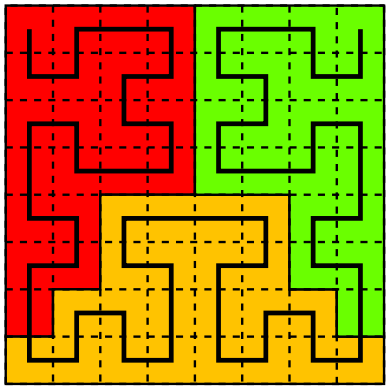
a) Cas équilibré

Dans le cas d'une partition initiale équilibrée, la méthode la plus simple pour construire une matrice de migration permettant à la fois une bonne migration (métriques TotalV, MaxV, TotalZ et MaxZ) et une bonne coupe, consiste à réutiliser le principe du partitionnement de la chaîne présentée dans la section 3.2.

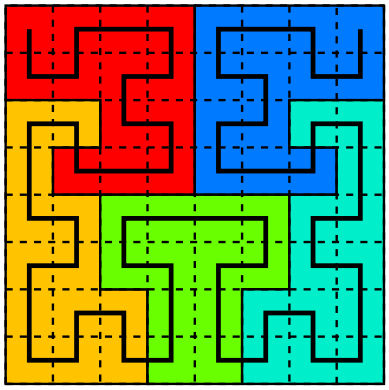
De telles matrices de migration peuvent être obtenues à l'aide d'un repartitionnement basé sur des courbes de remplissage ou *space filling curve* (SFC) [52,54]. Ce repartitionnement géométrique construit une courbe remplissant l'espace à partitionner, comme par exemple à l'aide d'une courbe de Hilbert [54] (cf. figure 3.8). On peut facilement obtenir une partition en M parties en coupant cette courbe en M sections égales. Si on construit une nouvelle partition en N parties en utilisant la même courbe de remplissage, on obtient un repartitionnement basé sur la chaîne (la chaîne étant la courbe de remplissage). Cette méthode a l'inconvénient de nécessiter des informations géométriques sur les données à partitionner et donne une coupe assez élevée à cause de la nature fractale des courbes utilisées.

Pour appliquer le principe du partitionnement de la chaîne sur un graphe quelconque, il suffit de trouver un chemin dans le graphe quotient. L'utilisation du graphe quotient au lieu du graphe à partitionner permet de laisser à une heuristique de partitionnement classique le choix de la distribution des sommets. Ainsi, le problème de coupe des courbes de remplissage est réglé tout en gardant le même schéma de communication. Le chemin de M sommets dans le graphe quotient est alors découpé en N nouvelles parties, un sommet (correspondant à une ancienne partie) pouvant être partagé entre plusieurs nouvelles parties. Le choix de ce chemin influence directement la coupe de la partition finale. En effet, comme une nouvelle partie peut prendre des sommets de plusieurs anciennes parties consécutives dans ce chemin, il est préférable que ces parties soient bien connectées entre elles. Il faudra donc trouver un chemin dont les arêtes ont des poids élevés dans le graphe quotient.

Pour optimiser la migration, une renumérotation des nouvelles parties est nécessaire, car en numérotant les parties dans l'ordre du chemin, les numéros sont décalés par rapport à la partition initiale comme on peut le voir sur la figure 3.9. L'utilisation d'un chemin peut aussi être vue comme une renumérotation des anciennes parties après laquelle on applique le repartitionnement de la chaîne sur le nouvel ordre des parties. Donc, deux renumérotations sont en fait nécessaires : l'une permutant les anciennes parties (ou lignes de la matrice de migration) en fonction du chemin, l'autre permutant les nouvelles parties (ou colonnes de la matrice de migration) pour optimiser la migration.



(a) Partitionnement en 3.

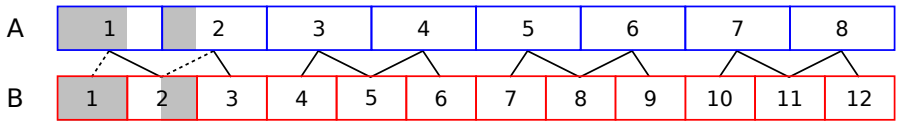


(b) Partitionnement en 5.

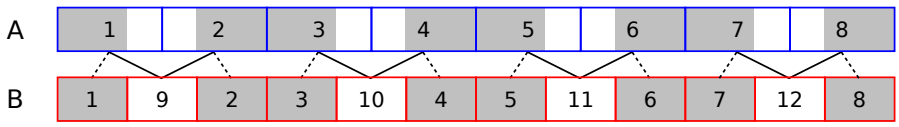
$$\begin{pmatrix} 13 & 8 & & & \\ & 5 & 12 & 5 & \\ & & & 8 & 13 \end{pmatrix}$$

(c) Matrice de migration.

FIGURE 3.8 – Partitionnement et repartitionnemnent d'une grille 8×8 basés sur une courbe de remplissage de Hilbert.



(a) Numérotation des nouvelles parties dans l'ordre du chemin.



(b) Renumérotation pour optimiser la migration.

FIGURE 3.9 – Exemple de numérotations des nouvelles parties. Les arêtes en pointillés correspondent aux « communications » d'un processeur vers lui-même. Les données qui ne migrent pas sont en gris clair.

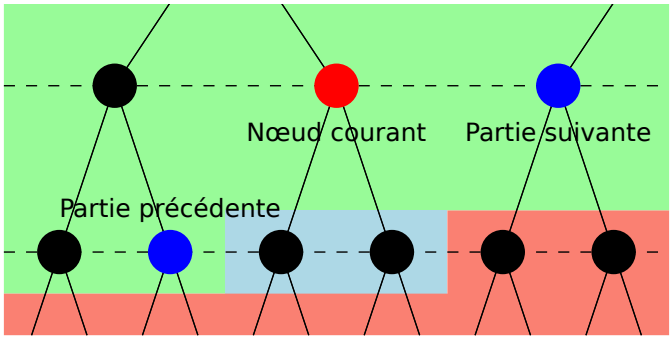


FIGURE 3.10 – Exemple d'étape de l'algorithme de recherche de chemin à partir de l'arbre des bissections. La zone verte contient les nœuds dont l'ordre est déjà fixé, la zone rouge ceux dont on ne connait pas encore l'ordre. Le choix de l'ordre est à faire entre les deux sommets de la zone bleue claire.

La recherche d'un tel chemin dans le graphe quotient est un problème analogue à celui du voyageur de commerce, qui est NP-complet [20, TRAVELING SALESMAN].

Il existe de nombreuses méthodes pour obtenir des chemins non optimaux mais assez proches de l'optimal, comme par exemple des algorithmes probabilistes. Mais certains « bons » chemins peuvent créer des problèmes dans le cas d'un partitionnement par bissections récursives. En effet, une bissection devra couper ce chemin en deux en même temps que le graphe. La bissection du chemin est forcée par la numérotation des nouvelles parties : elle sépare les plus petits numéros des plus grands. Une bissection du chemin par le milieu doit donc laisser la possibilité d'une bonne bissection du graphe.

Dans le cas où la partition initiale a été réalisée par un partitionneur utilisant des bissections récursives, l'arbre des bissections peut être utilisé pour trouver rapidement un chemin. Les parties issues d'une bissection récursive sont numérotées dans l'ordre des feuilles. Les nœuds internes représentent l'union des parties dans le sous-arbre. À l'aide d'un parcours en largeur de l'arbre, chaque nœud interne est visité. L'ordre de ses deux fils est choisi de façon à maximiser les connexions avec les parties (ou unions de parties) précédentes et suivantes. Comme tout l'arbre n'a pas encore été visité, l'ordre de toutes les parties n'a pas encore été décidé. On prend donc la partie précédente au même niveau dans l'arbre et la partie suivante au niveau supérieur comme indiqué sur la figure 3.10.

La figure 3.12 présente un exemple d'exécution de cet algorithme sur le graphe quotient de la figure 3.11. L'arbre initial est celui de la figure 3.12a. Au niveau de la racine, il n'y a pas de choix à faire car il n'y a ni partie précédente ni suivante. Le premier choix arrive avec le nœud unissant 0 et 1. La partie suivante est 2–4 dont la connexion avec 0 est $39 + 14 = 53$ et celle avec 1 est $17 + 34 = 51$. La partie 0 est donc placée en second comme indiqué sur la figure 3.12b. Sur la figure 3.12c, les connexions des parties 2 et 3–4 avec la partie précédente 0 sont comparées. Enfin, les connexions des parties 3 et 4 avec les parties précédente 0 et suivante 2 sont comparées, et l'ordre est conservé (figure 3.12d).

En pratique, pour améliorer la qualité du chemin en gardant un algorithme rapide, toutes les solutions possibles sont recherchées quand on arrive sur un sous-arbre suffisamment petit.

Une fois la nouvelle partition obtenue, il faut renuméroter les parties pour optimiser la migration comme montré sur la figure 3.9. De la même façon que le *Scratch-Remap*, il faut chercher la meilleure association entre anciennes et nouvelles parties. Mais dans le cas du partitionnement

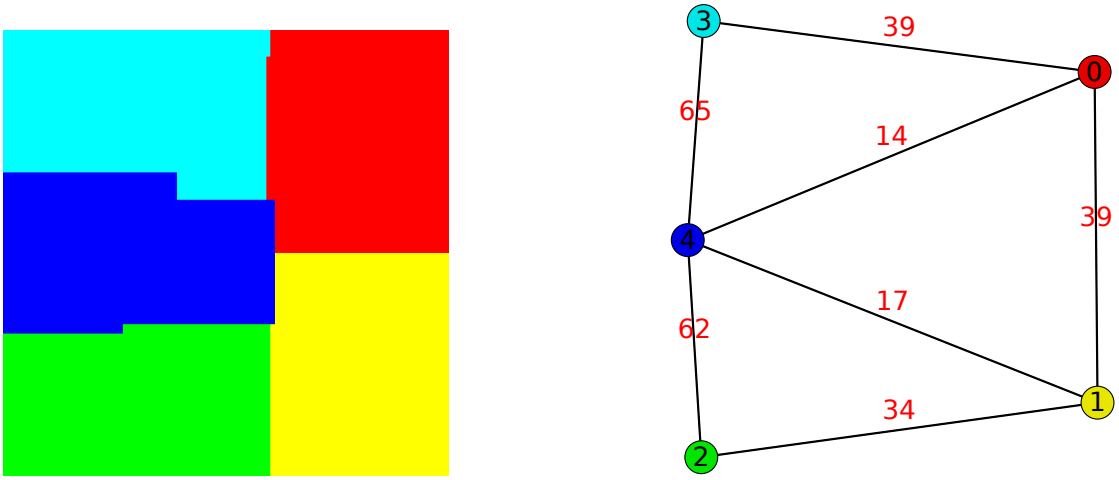


FIGURE 3.11 – Exemple de partition initiale en 5 partie et du graphe quotient associé.

de la chaîne, le choix est plus simple :

— si $M < N$, pour chaque ancienne partie, on choisit la nouvelle partie qui reçoit le plus de données ;

— si $M > N$, pour chaque nouvelle partie, on choisit l'ancienne partie qui lui en envoie le plus.

Les nouvelles parties qui ne sont pas associées à une ancienne prennent alors de nouveaux numéros qui n'existaient pas dans l'ancienne partition : ce sont les $N - M$ plus grands.

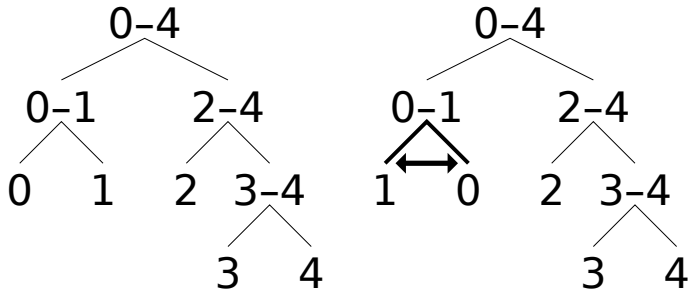
On peut vérifier par l'absurde que ces choix sont possibles dans le cas $M < N$. Cette méthode de renumérotation poserait problème si et seulement si une même nouvelle partie est le meilleur choix pour deux anciennes. Si une ancienne partie partage des données avec trois ou plus nouvelles parties, une de ces nouvelles parties ne reçoit des données que de cette ancienne partie et ne serait le meilleur choix que pour celle-ci. Ces deux anciennes parties n'ont donc chacune des données partagées qu'avec deux nouvelles : les nouvelles parties étant plus petites, il y en a au moins deux. La nouvelle partie qui serait le meilleur choix pour les deux anciennes doit donc recevoir plus de la moitié des données de chacune des deux anciennes parties. Elle serait donc plus grande que les anciennes, ce qui est en contradiction avec l'hypothèse $M < N$.

On peut faire un raisonnement similaire dans le cas $M > N$, en inversant les rôles des anciennes et nouvelles parties.

b) Illustration dans le cas équilibré

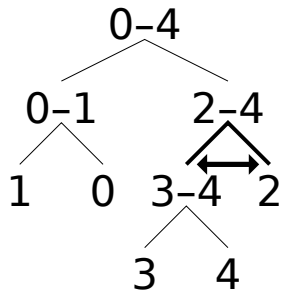
La figure 3.13 présente un exemple de repartitionnement dans le cas où il y a 5 parties initiales équilibrées (1400 éléments chacune) et 7 parties finales qui doivent contenir 1000 éléments chacune pour être équilibrées. On commence par construire une matrice correspondant au repartitionnement de la chaîne comme présenté dans la section 3.2 (matrice en escalier) :

$$\begin{pmatrix} 1000 & 400 & 0 & 0 & 0 & 0 & 0 \\ 0 & 600 & 800 & 0 & 0 & 0 & 0 \\ 0 & 0 & 200 & 1000 & 200 & 0 & 0 \\ 0 & 0 & 0 & 0 & 800 & 600 & 0 \\ 0 & 0 & 0 & 0 & 0 & 400 & 1000 \end{pmatrix}$$

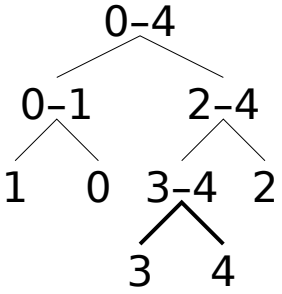


(a) Arbre de bissections initial.

(b) Permutation de 0 et 1.



(c) Permutation de 2 et 3–4.



(d) L'ordre entre 3 et 4 est préservé.

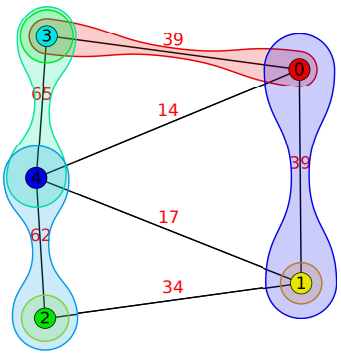
FIGURE 3.12 – Exemple de recherche de chemin par permutation de l'arbre des bissections.

Ensuite, un chemin est recherché dans le graphe, maximisant la connexion entre les sommets successifs (les anciennes parties). Ici, le chemin choisi est 1, 0, 3, 4 et 2. Les lignes de la matrice sont permutées en conséquence. Enfin, il faut renuméroter les nouvelles parties pour minimiser le volume de migration. On choisit les nouvelles parties recevant le plus d'éléments des anciennes parties. Les nouvelles parties correspondant aux anciennes 1, 0, 3, 4 et 2, sont respectivement 0, 2, 3, 4 et 6. Renuméroter les nouvelles parties revient à permuter les colonnes de la matrice de migration. Les nouvelles parties non associées aux anciennes (1 et 5) prennent de nouveaux numéros, ici 5 et 6. Les colonnes sont permutées en conséquence, les colonnes supplémentaires (5 et 6) peuvent être dans un ordre quelconque.

La figure 3.14 présente les hypergraphes de repartitionnement dans trois cas de repartitionnement. Dans les cas ou les nombres de parties initial et final ne sont pas premiers entre eux (8×12 et 12×14), les chemins sont coupés en plusieurs parties : ce sont les composantes connexes utilisées dans la démonstration du lemme 2. On remarque également que les hyper-arêtes sont de petites tailles et regroupent des anciennes parties voisines comme on le souhaitait.

c) Généralisation au cas déséquilibré

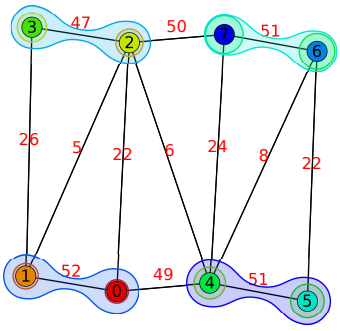
Cette méthode peut être facilement adaptée dans le cas où la partition initiale est déséquilibrée. Il faut d'abord trouver un chemin dans le graphe quotient. Ensuite, connaissant cet ordre des parties, il suffit de le redécouper en parties de tailles égales. Dans le cas déséquilibré, toutes



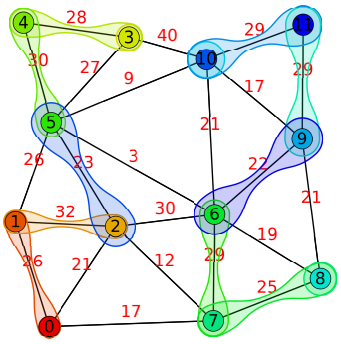
$$\begin{pmatrix} 800 & 0 & 0 & 0 & 0 & 0 & 600 \\ 0 & 1000 & 0 & 0 & 0 & 0 & 400 \\ 0 & 0 & 1000 & 0 & 0 & 400 & 0 \\ 200 & 0 & 0 & 1000 & 200 & 0 & 0 \\ 0 & 0 & 0 & 0 & 800 & 600 & 0 \end{pmatrix}$$

$$\begin{aligned} \text{TotalV} &= 2400 \\ \text{MaxV} &= 1000 \\ \text{TotalZ} &= 6 \\ \text{MaxZ} &= 2 \end{aligned}$$

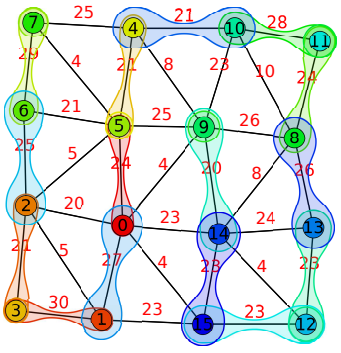
FIGURE 3.13 – Exemple de repartitionnement basé sur la chaîne dans le cas 5×7 (cas équilibré).



(a) Hypergraphe de repartitionnement dans le cas 8×12.

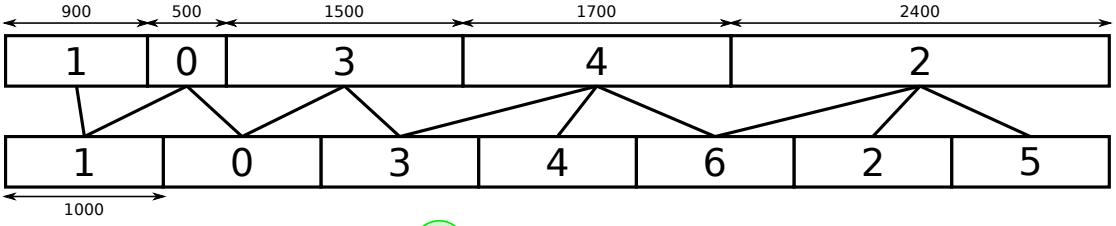


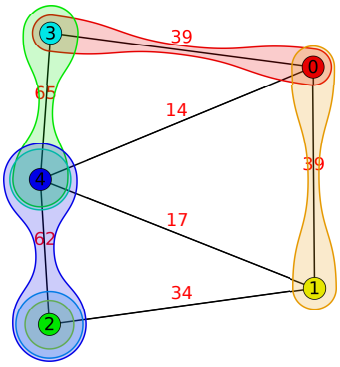
(b) Hypergraphe de repartitionnement dans le cas 12×14.



(c) Hypergraphe de repartitionnement dans le cas 16×21.

FIGURE 3.14 – Applications du repartitionnement basé sur la chaîne dans le cas équilibré.





$$\begin{pmatrix} 400 & 100 & 0 & 0 & 0 & 0 & 0 \\ 0 & 900 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1000 & 0 & 0 & 1000 & 400 \\ 600 & 0 & 0 & 900 & 0 & 0 & 0 \\ 0 & 0 & 0 & 100 & 1000 & 0 & 600 \end{pmatrix}$$

$$\begin{aligned} \text{TotalV} &= 2800 \\ \text{MaxV} &= 1400 \\ \text{TotalZ} &= 6 \\ \text{MaxZ} &= 2 \end{aligned}$$

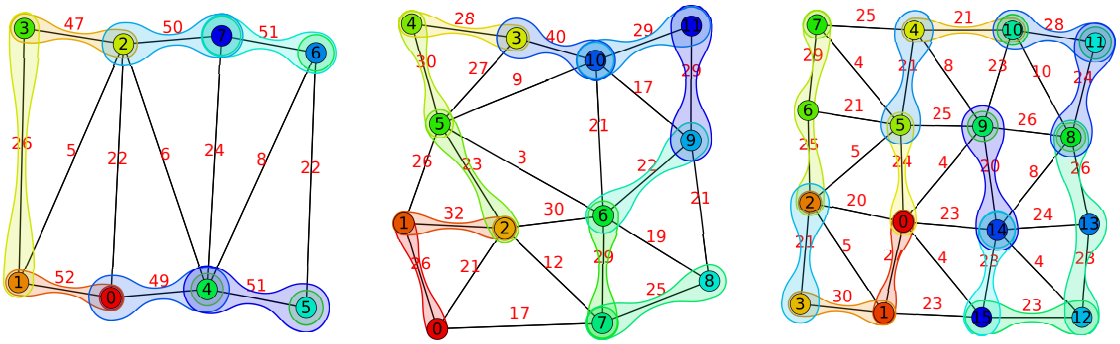
FIGURE 3.15 – Exemple de repartitionnement basé sur la chaîne dans le cas 5×7 (cas déséquilibré).

les parties ne se valent pas. La forme de l'hypergraphe de repartitionnement utilisé peut varier suivant l'ordre choisi.

Par exemple, considérons le cas 5×7 où les parties sont initialement déséquilibrées avec comme poids dans l'ordre de 0 à 4 : 500, 900, 2400, 1500 et 1700. En reprenant le même chemin [1] que dans le cas équilibré (figure 3.12), l'ordre des poids devient 900, 500, 1500, 1700 et 2400, et on obtient le repartitionnement présenté sur la figure 3.15.

La figure 3.16 présente alors les hypergraphes de repartitionnement obtenus avec la méthode de la chaîne dans le cas déséquilibré. Contrairement au cas équilibré, il n'est plus garanti que certaines séparations entre parties soient communes entre l'ancienne partition et la nouvelle. Le nombre de messages peut donc atteindre, au pire, $M+N-1$ messages. On peut voir que les hyper-arêtes forment les mêmes chemins que ceux de la figure 3.14 mais « sans être coupées ». Dans le cas équilibré, le chemin était découpé en $\text{pgcd}(M, N)$ parties correspondant aux composantes connexes du graphe biparti de repartitionnement utilisées dans la section 3.2.

1. Le chemin ne dépend que de la connectivité du graphe quotient, pas du poids de ses sommets.



(a) Hypergraphe de repartitionnement dans le cas 8×12.

(b) Hypergraphe de repartitionnement dans le cas 12×14.

(c) Hypergraphe de repartitionnement dans le cas 16×21.

FIGURE 3.16 – Applications du repartitionnement basé sur la chaîne dans le cas déséquilibré.

3.3.2 Méthode d'appariement (*Matching*)

Dans le cas équilibré, nous avons vu en section 3.2 qu'il est possible de construire des matrices de migration optimales, c'est-à-dire optimisant à la fois le volume de migration et le nombre de messages. Mais l'optimisation de la coupe n'est pas considérée. Comme on l'a vu avec la méthode de la chaîne, il peut être nécessaire de permuter la matrice de migration pour obtenir une bonne coupe. Mais une matrice optimale ne correspond pas toujours à une chaîne : il ne suffit plus de trouver un bon chemin dans le graphe quotient. On cherche donc à apparier correctement la matrice de migration avec le graphe quotient pour que la partition finale ait une bonne coupe. Comme les valeurs des éléments de la matrice de migration n'influencent pas cet appariement, nous utiliserons l'hypergraphe de repartitionnement dans cette section.

Le choix de la matrice de migration ou de l'hypergraphe repartitionnement conditionne la qualité de la migration. Nous décidons de prendre une matrice de migration optimale (d'après la définition 9), plus précisément la matrice optimale composée d'une diagonale et complétée avec une sous-matrice « chaîne » (comme celle de la figure 3.7c). En plus de minimiser $TotalV$ et $TotalZ$, cette matrice a l'avantage de garder un $MaxZ$ plus faible que les autres matrices optimales.

a) Problème d'appariement entre le graphe quotient et l'hypergraphe de repartitionnement

Une bonne matrice de migration ne suffit donc pas pour obtenir un bon repartitionnement. Elle ne permet que d'optimiser la migration et l'équilibre. Un bon repartitionnement doit aussi fournir une coupe basse. La coupe sera principalement décidée lors de l'étape de repartitionnement, mais la matrice de migration a une influence sur la coupe. Une matrice de migration indique l'emplacement des nouvelles parties : elles seront là où se trouvent les anciennes parties dont elles prennent les sommets. Pour permettre au partitionneur de bien optimiser la coupe, il faut que les anciennes parties communiquant avec une même nouvelle partie soient proches.

Pour obtenir ce résultat, on apparie l'hypergraphe de repartitionnement avec le graphe quotient, c'est-à-dire qu'on cherche une correspondance entre les sommets de l'hypergraphe et ceux du graphe quotient comme présenté sur la figure 3.17. En effet, les hyper-arêtes regroupent les anciennes parties donnant des sommets à une même nouvelle partie, alors que les arêtes du graphe quotient connectent les anciennes parties proches.

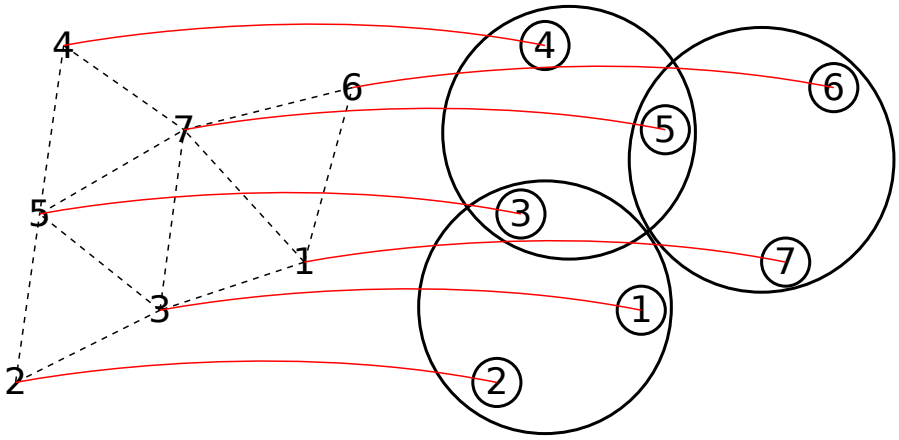


FIGURE 3.17 – Appariement d'un graphe quotient et d'un hypergraphe de repartitionnement.

Par exemple, il y a plusieurs façons d'apparier l'hypergraphe de repartitionnement pour repartitionner la partition initiale d'un maillage en grille visible sur la figure 3.18a. Les figures 3.18b et 3.18c présentent deux possibilités d'appariement de l'hypergraphe de repartitionnement en le superposant au graphe quotient. Dans le premier cas, les hyper-arêtes regroupant plusieurs sommets contiennent des sommets « proches » organisés en cliques. En conséquence la partition finale sur la figure 3.18d a des parties bien formées, alors que, dans le second cas, les hyper-arêtes regroupent des sommets « éloignés » et la partition finale sur la figure 3.18e a des parties déconnectées sur le maillage. Les métriques pour évaluer la migration (TOTALV, MAXV, TOTALZ et MAXZ) sont les mêmes dans ces deux cas, mais la coupe du graphe est bien meilleure dans le premier cas.

Pour éviter de tels scénarios, une métrique supplémentaire est introduite pour estimer la qualité de la coupe finale. Cette métrique doit favoriser les nouvelles parties prenant des données à un ensemble d'anciennes parties bien connectées, autrement dit des quasi-cliques. Cette métrique s'exprime à l'aide du graphe quotient de l'ancienne partition et de l'hypergraphe de repartitionnement.

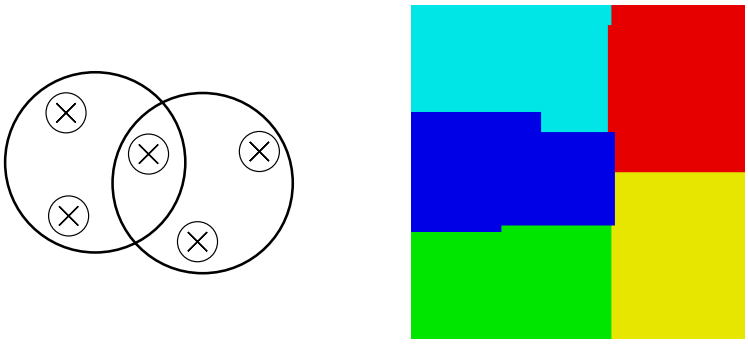
b) Définition d'un score d'appariement

Une fois la matrice choisie, il faut trouver une bonne correspondance entre les sommets du graphe quotient et ceux de l'hypergraphe de repartitionnement associé à la matrice. Cet appariement nécessite la définition d'un score.

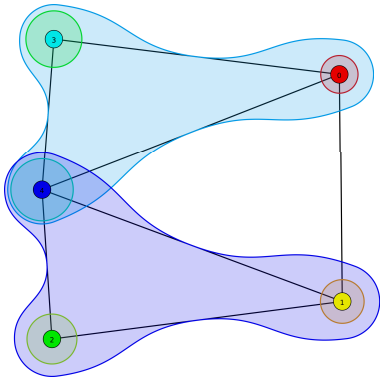
Notons $Q = (V_Q, E_Q)$ le graphe quotient par rapport à l'ancienne partition, $H = (V_H, E_H)$ l'hypergraphe de repartitionnement déjà apparié (on considère que les sommets de Q et de H sont les mêmes, on pose $V = V_Q = V_H$) et $E_S(e)$ l'ensemble des arêtes du sous-graphe de Q induit par l'hyper-arête e. Une première version simple de ce score compte la somme des poids des arêtes des sous-graphes induits par chaque hyper-arête e :

$$\text{Score} = \sum_{e \in E_H} \left(\sum_{e' \in E_S(e)} w_{e'} \right) \tag{3.12}$$

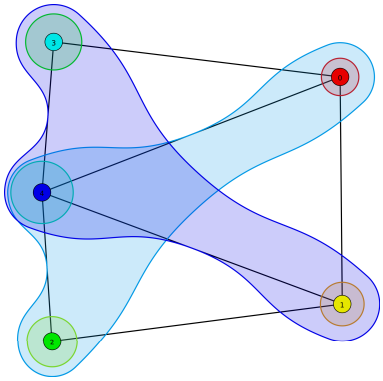
$$\text{où} \quad E_S(e) = \{(i,j) \in E_Q : i \in e \wedge j \in e\}\,.$$



(a) Hypergraphe de repartitionnement et partition initiale.

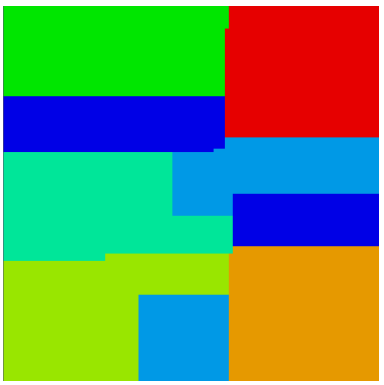


(b) Exemple de bonne correspondance.



(c) Exemple de mauvaise correspondance.

(d) La partition est bien formée.



(e) Certaines parties sont déconnectées.

FIGURE 3.18 – Deux cas d'application d'un même hypergraphe de repartitionnement.

Il est possible d'écrire ce score sous forme matricielle. Notons X la *matrice d'appariement* (ou de permutation) de taille $M \times M$ à trouver : $X_{i,i'}$ vaut 1 si et seulement si le sommet i de l'hypergraphe de repartitionnement est associé au sommet i' du graphe quotient, sinon 0. La matrice H représente l'hypergraphe de repartitionnement [2]. Elle est de taille $M \times N$ et $H_{i,j}$ vaut 1 si est seulement si le sommet i est inclus dans l'hyper-arête j. Q est la matrice d'adjacence du graphe quotient de taille $M \times M$, $Q_{i,j}$ est le poids de l'arête entre les sommets i et j. Ces notations sont résumées sur la figure 3.19.

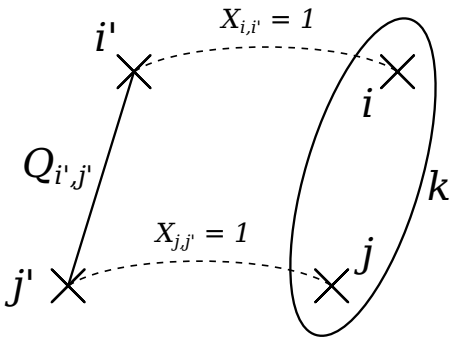


FIGURE 3.19 – Illustration des notations utilisées dans la formule de score avec une arête (i', j') du graphe quotient Q à gauche et une hyper-arête k à droite.

Grâce aux matrices à valeurs binaires X et H, on peut écrire la somme du poids des arêtes $e_{i',j'}$ dont les extrémités i' et j' sont respectivement associées à deux sommets i et j dans la même hyper-arête k :

$$\text{Score} = \sum_{i,j,i',j',k} X_{i,i'} X_{j,j'} H_{i,k} H_{j,k} Q_{i',j'} \quad \text{soit} \tag{3.13}$$

$$\text{Score} = \sum_{i,i'} \left(X_{i,i'} \sum_{j,j'} \left(X_{j,j'} \left(\sum_{k} H_{i,k} H_{j,k} \right) Q_{i',j'} \right) \right). \tag{3.14}$$

Définition 10 (Produit de Kronecker)**.** Le produit de Kronecker de deux matrices $A \otimes B$, A de taille $m \times n$ et B de taille $m' \times n'$, est une matrice de taille $mm' \times nn'$, décrite par la matrice bloc suivante :

$$\begin{pmatrix} A_{1,1}B & \cdots & A_{1,n}B \\ \vdots & \ddots & \vdots \\ A_{m,1}B & \cdots & A_{m,n}B \end{pmatrix}.$$

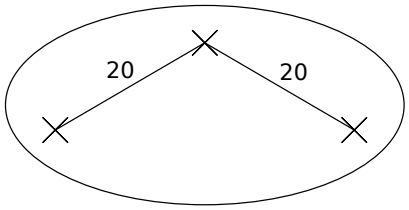
En notant $A = HH^T \otimes Q$, où $\otimes$ est le produit de Kronecker, et x le vecteur colonne de taille M^2 tel que $x_{iM+i'} = X_{i,i'}$, le score s'écrit comme la forme quadratique :

$$\text{Score} = x^T A x. \tag{3.15}$$

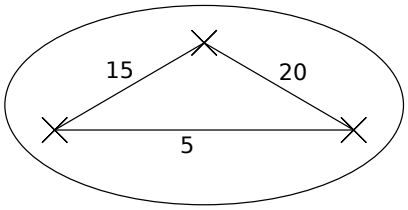
Dans l'équation 3.14, on reconnait le coefficient de la matrice $A = HH^T \otimes Q$:

$$A_{iM+i',jM+j'} = \left(\sum_{k} H_{i,k} H_{j,k} \right) Q_{i',j'},$$

2. Par abus de langage, on confond les graphes et hypergraphes avec leurs matrices d'adjacence.



(a) Chaine de 3 parties réunies dans une même hyper-arête. Les deux parties aux extrémités sont éloignées l'une de l'autre.



(b) Clique de 3 parties réunies dans une même hyper-arête.

FIGURE 3.20 – Deux exemples de sous-graphes quotients appariés avec une hyper-arête.

$\sum_k H_{i,k}H_{j,k}$ correspondant au coefficient (i,j) de la matrice HH^T.

Cette métrique favorise les hyper-arêtes regroupant des sommets fortement connectés. Mais elle est loin d'être parfaite, car elle ne prend pas en compte le nombre d'arêtes du graphe quotient associées à une hyper-arête. À score égal, il apparait plus avantageux de favoriser trois sommets connectés par trois arêtes plutôt que par deux.

Sur l'exemple de la figure 3.20, les deux appariements d'une hyper-arête avec un graphe de 3 sommets ont le même score : 40. Pourtant, dans le cas de la figure 3.20a, les deux parties aux extrémités de la chaîne peuvent être éloignées et la nouvelle partie créée devra alors être étirée, ce qui n'est pas optimal pour la coupe. On préférerait favoriser les cas de cliques comme sur la figure 3.20b, où les 3 parties sont bien regroupées.

Pour corriger ces défauts, nous allons modifier le score pour prendre en compte la quantité d'arêtes et la taille des hyper-arêtes [3]. On propose la nouvelle formule de score suivante. Soient $E_S(e)$ l'ensemble des arêtes du sous-graphe induit par l'hyper-arête e et $K(e)$ l'ensemble des arêtes d'une clique entre les sommets de l'hyper-arête e. La métrique s'exprime :

$$\text{Score}^{\text{R}} = \sum_{e \in E_H} \left(\frac{|E_S(e)|}{|K(e)|} \sum_{e' \in E_S(e)} w_{e'} \right) \tag{3.16}$$

$$\text{où} \quad \begin{cases} E_S(e) = \{(i,j) \in E_Q : i \in e \wedge j \in e\} \\ K(e) = \left\{(i,j) \in V^2 : i \neq j \wedge i \in e \wedge j \in e\right\}. \end{cases}$$

En reprenant l'exemple 3.20, les anciens scores sont multipliés par 2/3 dans le cas de la chaîne (fig. 3.20a) et par 1 dans le cas de la clique (fig. 3.20b), ce qui donne les scores 26 et 40 : la clique est maintenant favorisée par rapport à la chaîne.

c) Choix d'un appariement par une méthode d'optimisation du score

Afin de choisir un bon appariement, nous allons utiliser une méthode d'optimisation du score défini précédemment.

La maximisation de l'équation 3.15 est un problème d'optimisation quadratique à valeurs binaires. C'est un problème NP-difficile [20, QUADRATIC PROGRAMMING] mais déjà beaucoup étudié, en particulier dans le domaine de la vision numérique [15,41]. Une première solution consiste à utiliser une *méthode spectrale*. Comme $x^T x = M$ est une constante, maximiser revient

3. On rappelle que la taille d'une hyper-arête est le nombre de sommets qu'elle regroupe.

à maximiser le quotient de Rayleigh-Ritz [25] $\frac{x^T A x}{x^T x}$. Dans le cas où x est à coefficient réels, ce quotient atteint son maximum pour un vecteur propre associé à la plus grande valeur propre. Mais ici x est à valeur binaire, et la relaxation du problème ne donne pas de solution convenable. En effet, approcher le vecteur propre avec un vecteur correspondant à une permutation est une approximation trop importante.

D'autres heuristiques pour trouver un bon appariement sont possibles, comme par exemple des algorithmes probabilistes, combinatoires de type *branch & bound*, ... Nous avons choisi d'utiliser un algorithme de *recuit simulé* [40] qui donne des résultats satisfaisants. En partant d'une correspondance quelconque, des permutations de deux sommets aléatoires sont appliquées. Si la nouvelle correspondance a un meilleur score, elle est conservée. Sinon, elle est conservée ou non avec une probabilité qui décroit exponentiellement avec le nombre d'itérations. Après un nombre d'itérations fixé, l'algorithme s'arrête. Cette méthode peut s'appliquer à n'importe quelle forme de score et notamment à la formule du score 3.16.

d) Illustration de la méthode de matching

La figure 3.21 présente un exemple de repartitionnement basé sur le *matching*. Dans un premier temps, une matrice de migration optimale est choisie. Ici, on choisit une matrice combinant une sous-matrice diagonale avec une sous-matrice en escalier dont l'hypergraphe de repartitionnement associé est représenté dans la figure 3.21a. Cette classe de matrice a l'avantage de fournir un nombre de messages par partie souvent plus faible que les matrices construites récursivement (cf. section 3.2). Ensuite, cet hypergraphe de repartitionnement est mis en correspondance avec le graphe quotient à l'aide de l'heuristique de recuit simulé optimisant la fonction Score^{R} (équation 3.16). Le résultat est visible sur la figure 3.21b. Cette appariement correspond à une permutation des lignes de la matrice mais aussi des 5 premières colonnes pour garder une diagonale optimisant le volume de migration.

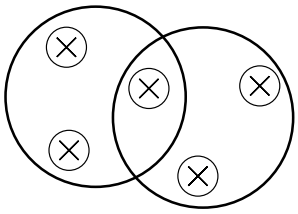
La figure 3.22 présente des résultats d'application de la méthode d'appariement dans différents cas. On remarque que dans le cas 8×12 (figure 3.22a), le résultat est le même qu'avec la méthode de la chaîne. En effet, les matrices 8×12 choisies dans les deux méthodes sont équivalentes à une permutation près dans ce cas précis. Dans le cas 12×14 (figure 3.22b), il y a de très grandes hyper-arêtes. Cela risque de poser problème pour la construction des parties associées à ces grandes hyper-arêtes. Ce problème vient du choix de la matrice optimale qui concentre les non-zéros dans les dernières colonnes de la matrice. Cela aurait pu être évité en choisissant un autre type de matrice, mais le volume total de migration n'aurait pas été optimal. Le recuit simulé ne trouve pas toujours de bonne solution : sur la figure 3.22c, l'hyper-arête contenant les sommets 0, 4, 7 et 15 ne contient que deux arêtes (4–7 de poids 25 et 0–15 de poids 4), mais les autres hyper-arêtes sont assez bien placées.

Notons que cette méthode de repartitionnement ne peut pas être appliquée dans le cas où les parties sont initialement déséquilibrées. La mise en correspondance de l'hypergraphe de repartitionnement et du graphe quotient nécessite que les parties initiales soient interchangeables, ce qui n'est le cas que quand les parties initiales ont toutes le même poids.

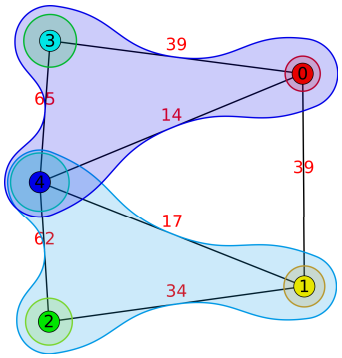
3.3.3 Méthode gloutonne (*Greedy*)

Dans le cas général, il n'est pas possible de choisir une matrice avant de l'apparier au graphe quotient : les anciennes parties ont chacune un poids différent et elles ne sont plus interchangeables. Il faut construire la matrice de migration directement en tenant compte de la connectivité entre les parties et de leurs différents poids.

$$\begin{pmatrix} 1000 & 0 & 0 & 0 & 0 & 400 & 0 \\ 0 & 1000 & 0 & 0 & 0 & 400 & 0 \\ 0 & 0 & 1000 & 0 & 0 & 200 & 200 \\ 0 & 0 & 0 & 1000 & 0 & 0 & 400 \\ 0 & 0 & 0 & 0 & 1000 & 0 & 400 \end{pmatrix}$$



(a) Matrice de migration optimale et hypergraphe de repartitionnement associé pour le cas 5×7.

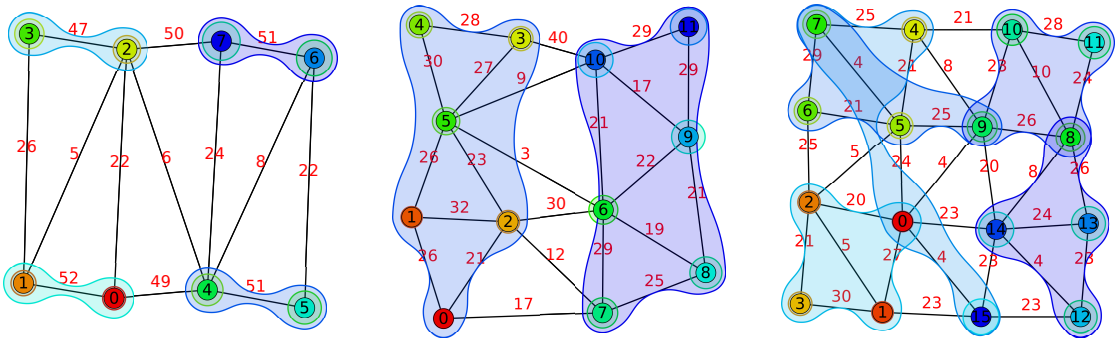


$$\begin{pmatrix} 1000 & 0 & 0 & 0 & 0 & 0 & 400 \\ 0 & 1000 & 0 & 0 & 0 & 400 & 0 \\ 0 & 0 & 1000 & 0 & 0 & 400 & 0 \\ 0 & 0 & 0 & 1000 & 0 & 0 & 400 \\ 0 & 0 & 0 & 0 & 1000 & 200 & 200 \end{pmatrix}$$

$$\begin{aligned} \text{TotalV} &= 2000 \\ \text{MaxV} &= 1000 \\ \text{TotalZ} &= 6 \\ \text{MaxZ} &= 3 \end{aligned}$$

(b) Hypergraphe apparié au graphe quotient et matrice de migration finale.

FIGURE 3.21 – Exemple de repartitionnement basé sur le *matching* dans le cas 5×7.



(a) Hypergraphe de repartitionnement dans le cas 8×12.

(b) Hypergraphe de repartitionnement dans le cas 12×14.

(c) Hypergraphe de repartitionnement dans le cas 16×21.

FIGURE 3.22 – Application du repartitionnement basé sur le *matching*.

a) Construction de la matrice de migration par une méthode gloutonne

La matrice de migration C est construite par l'algorithme 2 selon une méthode gloutonne en prenant en entrée le graphe quotient $Q = (V, E)$ et le nouveau nombre de parties N. On note $(P_i)_{i \in [\![0, M-1]\!]}$ les anciennes parties et $(P'_j)_{j \in [\![0, N-1]\!]}$ les nouvelles parties. L'algorithme commence avec une matrice vide, puis les colonnes de la matrice (correspondant aux hyper-arêtes de l'hypergraphe de repartitionnement) sont remplies une par une. Ce remplissage s'effectue de façon à respecter les contraintes sur les sommes des lignes et des colonnes qui correspondent respectivement aux poids des anciennes parties ($w(P_i)$) et des nouvelles parties ($w(P'_j)$). La migration peut être minimisée (*optdiag* = vrai en entrée de l'algorithme) en construisant une hyper-arête de taille 1 pour chaque sommet du graphe quotient quand cela est possible, c'est-à-dire quand l'ancienne partie est plus grosse que la nouvelle ($w(P_i) > w(P'_j)$). Les hyper-arêtes suivantes sont construites de façon gloutonne en commençant par une sommet pseudo-périphérique (voir définition 12) de faible degré, puis en ajoutant des sommets v un à un dans l'hyper-arête h en cherchant à maximiser un score inspiré de celui présenté dans la section 3.3.2 :

$$score\,(v,h) = \frac{|E_S(h) \cap E_V(v)|}{|E_V(v)|} \times |E_S(h)| \times \frac{\sum_{e \in E_S(h) \cap E_V(v)} w_e}{\sum_{e \in E_V(v)} w_e} \times \sum_{e \in E_S(h)} w_e$$
$$\text{avec} \quad \begin{cases} E_S(h) = \{(u,v) \in E : u \in h \wedge v \in h\} \\ E_V(v) = \{e \in E : v \in e\}\,. \end{cases} \tag{3.17}$$

$E_S(h)$ est l'ensemble des arêtes du sous-graphe défini par l'hyper-arête h. $E_V(v)$ est l'ensemble des arêtes dont une extrémité est v (son voisinage). Le score de l'équation 3.17 reprend du score 3.16 le produit du nombre d'arêtes et de leur poids permettant d'obtenir des groupes d'anciennes parties bien connectées et le multiplie par la proportion des arêtes autour du nouveau sommet v qui sont dans l'hyper-arête h, en nombre et en poids. Cette modification du score permet de mieux guider l'algorithme glouton qui préférera alors choisir des sommets sur le bord ou encerclés par l'hyper-arête, laissant de meilleurs choix pour la construction des hyper-arêtes suivantes.

Pour chaque sommet ajouté dans l'hyper-arête, on remplit le coefficient correspondant dans la matrice avec la valeur maximale permettant de respecter les contraintes de poids sur les lignes et colonnes. Comme les coefficients de la matrice doivent être positifs, la somme des valeurs sur la ligne (et respectivement la colonne) ne doit pas dépasser la taille d'une partie initiale (et respectivement finale). À une itération de notre algorithme, la valeur choisie pour l'élément $C_{i,j}$

est donc $\min(w(P_i) - \sum_k C_{i,k}, w(P'_j) - \sum_k C_{k,j})$. Comme on veut obtenir une partition finale équilibrée, on pose $\forall j, w(P'_j) = w_{final} = \frac{W}{N}$.

Le volume total de migration peut être minimisé en commençant par remplir la diagonale de la matrice de migration avec les plus grandes valeurs possibles. Le reste de la matrice est alors rempli normalement selon l'heuristique gloutonne.

Algorithme 2 Construction de la matrice de migration C (*Greedy1*)

Entrée : Nombre d'anciennes parties M
Entrée : Nombre de nouvelles parties N
Entrée : Graphe quotient de l'ancienne partition $Q = (V, E)$
Entrée : Booléen *optdiag* indiquant l'optimisation de la diagonale
  $C \leftarrow$ matrice nulle de dimension $M \times N$
  $j \leftarrow 0$ /* Indice de l'hyper-arête en cours de construction. */
  /* Construction d'hyper-arêtes de taille 1. */
  si *optdiag* **alors**
    pour tout $v \in V$ correspondant à l'ancienne partie P_i **faire**
      si $w_v \geq w_{final}$ et $j < N$ **alors**
        $C_{i,j} \leftarrow w_{final}$
        $w_v \leftarrow w_v - w_{final}$
        $j \leftarrow j + 1$
      fin si
    fin pour
  fin si
  /* Construction gloutonne des hyper-arêtes */
  tant que $j < N$ **faire**
    $v \leftarrow$ sommet pseudo-périphérique de Q d'indice i et de poids w_i non nul
    $h \leftarrow \{v\}$ /* Hyper-arête d'indice j en cours de construction. */
    $C_{i,j} \leftarrow \min(w_i, w_{final})$
    $w_i \leftarrow w_i - C_{i,j}$
    tant que $\sum_k C_{k,j} < w_{final}$ **faire**
      $v \leftarrow$ sommet de V d'indice i, de poids w_i non nul tel que $score(v, h \cup \{v\})$ soit maximal
      $h \leftarrow h \cup \{v\}$
      $C_{i,j} \leftarrow \min(w_i, w_{final} - \sum_k C_{k,j})$
      $w_i \leftarrow w_i - C_{i,j}$
    fin tant que
    $j \leftarrow j + 1$
  fin tant que
  retourner C

Notre algorithme se base sur la notion de sommet pseudo-périphérique pour trouver un sommet sur le « bord » du graphe. Rechercher un sommet périphérique [4] est plus complexe et n'est pas nécessaire pour l'application de notre heuristique.

Définition 11 (Excentricité)**.** L'excentricité d'un sommet v dans un graphe, notée $\epsilon(v)$, est la plus grande distance entre v et n'importe quel autre sommet du graphe.

4. Un sommet périphérique est un sommet d'excentricité maximale.

Définition 12 (Sommet pseudo-périphérique)**.** Un sommet v est un sommet pseudo-périphérique si pour tous les sommets u les plus éloignés de v, on a $\epsilon(v) = \epsilon(u) = d(u, v)$. Plus formellement v est pseudo-périphérique si $\forall u \in V,\ d(u, v) = \epsilon(v) \implies d(u, v) = \epsilon(u)$.

Un sommet pseudo-périphérique peut être facilement trouvé comme le montre l'algorithme 3. En partant d'un sommet quelconque, on cherche, à l'aide d'un parcours en largeur, un sommet le plus loin possible (calculant ainsi l'excentricité du sommet de départ) et de faible degré. Cette recherche est répétée à partir du nouveau sommet jusqu'à trouver un sommet dont l'excentricité est égale à celle du sommet précédent. En pratique cet algorithme termine en très peu d'itérations [21] (2 ou 3 itérations lors de nos tests).

Algorithme 3 Recherche d'un sommet pseudo-périphérique de degré faible

Entrée : Graphe G
  $u \leftarrow$ sommet quelconque de G
  $\epsilon_{nouveau} \leftarrow 0$
  répéter
    Effectuer un parcours en largeur du graphe G en partant de u
    $v \leftarrow$ sommet avec une distance maximale et un degré minimal
    $\epsilon_{courant} \leftarrow \epsilon_{nouveau}$
    $\epsilon_{nouveau} \leftarrow d(u, v)$ /* correspond à l'excentricité de u */
    $u \leftarrow v$
  jusqu'à $\epsilon_{nouveau} = \epsilon_{courant}$
  retourner u

b) Optimisation du nombre de messages par une recherche de sous-ensembles

L'algorithme 2 construit une matrice de migration avec $M + N - 1$ messages. À chaque itération, la valeur de l'élément est choisie pour satisfaire la contrainte de poids sur la ligne ou la colonne, sauf à la dernière itération où les contraintes sur la ligne *et* la colonne de l'élément sont satisfaites. Comme il y a M lignes et N colonnes, il y a au plus $M + N - 1$ itérations et éléments non nuls.

Il est possible de diminuer ce nombre de messages en décomposant le problème de construction de la matrice de migration en plusieurs sous-problèmes de repartitionnement sur des sous-ensembles disjoints de parties, comme le montre la démonstration du lemme 2. Pour décomposer le problème en K sous-problèmes, l'ensemble des parties $(P_i)_{i \in [\![1,M]\!]}$ est partitionné en K sous-ensembles $(S_j)_{j \in [\![1,K]\!]}$. Pour chaque sous ensemble S_j, la somme des poids des parties qu'il contient est à peu près un multiple de la taille idéale d'une partie finale. Plus formellement, avec un facteur de déséquilibre ϵ, un sous-ensemble S_j est de « bonne taille » si il existe un entier N_j tel que $\sum_j N_j = N$ et

$$(1 - \epsilon) \times N_j \frac{W}{N} < \sum_{P_i \in S_j} w(P_i) < (1 + \epsilon) \times N_j \frac{W}{N}.$$

On note $M_j = |S_j|$ la taille de chaque sous-ensemble. Comme les sous-ensembles sont une partition de l'ensemble des parties, on a $\sum_j M_j = M$. Chaque sous problème est donc de taille $M_j \times N_j$ et construit une sous-matrice de migration avec $M_j + N_j - 1$ messages. Le nombre de messages total est donc $M + N - K$. Pour minimiser le nombre de messages, il faut donc trouver le plus possible de sous-ensembles. Dans le cas équilibré, on trouvait $\text{pgcd}(M, N)$ tels sous-ensembles de chacun $M/\text{pgcd}(M, N)$ parties.

La recherche de tels sous-ensembles est une version plus générale du problème de la somme d'un sous-ensemble [5] qui est NP-complet [20, SUBSET SUM]. Ces sous-ensembles peuvent être construits avec un algorithme glouton similaire à celui de construction de la matrice de migration. Dans l'algorithme 4, on commence par trouver un sommet pseudo-périphérique dans le graphe quotient, puis on cherche parmi ses voisins si il existe un sommet permettant de créer un sous-ensemble de la bonne taille, sinon on en choisit un qui maximise le score du sous-ensemble comme dans l'algorithme 2. Puis on recommence à chercher parmi les voisins du sous-ensemble jusqu'à construire un sous-ensemble de bonne taille. Au pire, l'algorithme termine en sélectionnant tous les sommets du graphe quotient dans un seul sous-ensemble, puisque par définition de la taille idéale, le poids total du graphe vaut N fois celle-ci.

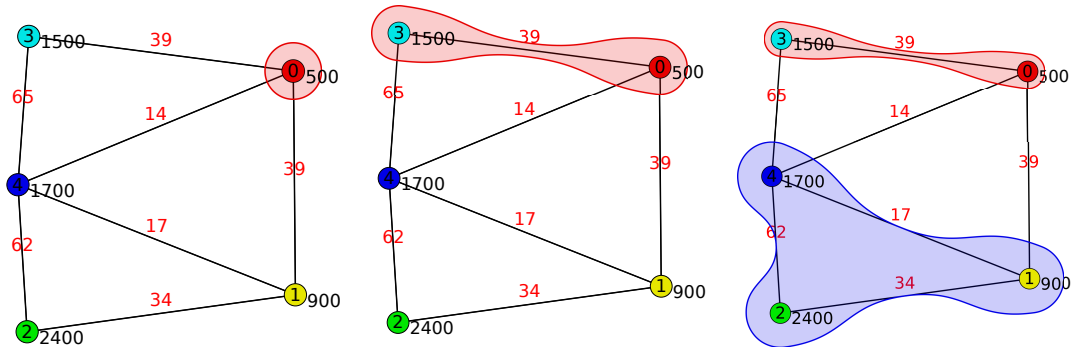
Algorithme 4 Construction de la matrice de migration avec une recherche de sous-ensembles (*Greedy2*)

Entrée : Nombre d'anciennes parties M
Entrée : Nombre de nouvelles parties N
Entrée : Graphe quotient $Q = (V, E)$
  $R \leftarrow V$ /* Ensemble des sommets restants */
  tant que $R \neq \emptyset$ **faire**
    $u \leftarrow$ un sommet pseudo-périphérique du sous-graphe de Q restreint à R
    $S \leftarrow \{u\}$
    $R \leftarrow R \setminus \{u\}$
    tant que S n'est pas un sous-ensemble de bonne taille pour N' nouvelles parties **faire**
      si il existe $v \in R$ tel que v est connecté à S et $S \cup \{v\}$ est de bonne taille **alors**
        $S \leftarrow S \cup \{v\}$ /* Sélectionner v */
        $R \leftarrow R \setminus \{v\}$
      sinon
        $u \leftarrow$ sommet maximisant le score du sous-ensemble $S \cup \{u\}$
        $S \leftarrow S \cup \{u\}$ /* Sélectionner u */
        $R \leftarrow R \setminus \{u\}$
      fin si
    fin tant que
    Construire la sous-matrice $|S| \times N'$ associée au sous-ensemble S (appel de l'algorithme 2 (*Greedy1*) sur le sous-graphe de Q restreint à S)
  fin tant que

Un exemple d'application de ces algorithmes est présenté sur les figures 3.23 à 3.25 avec le cas 5×7 déséquilibré. La figure 3.23 présente la recherche des sous-ensembles, alors que les figures 3.24 et 3.25 présentent respectivement les constructions des sous-matrices pour chacun des sous-ensembles. Les poids des parties initiales sont dans l'ordre 500, 900, 2400, 1500 et 1700. Les parties finales devront être de poids 1000. Les valeurs indiquées à coté des sommets correspondent aux données des anciennes parties qui n'ont pas encore été affectées dans une nouvelle. Les éléments de la matrice ne faisant pas partie de la sous-matrice concernée sont grisés.

L'algorithme commence par chercher un sous-ensemble en partant d'un sommet pseudo-périphérique (figure 3.23a). Parmi les voisins, la partie 3 permet d'avoir un sous-ensemble adapté

5. Étant donné un ensemble d'entiers, existe-t-il un sous-ensemble non vide dont la somme vaut une valeur donnée ?



(a) Première étape de la construction des sous-ensembles.

(b) Seconde étape de la construction des sous-ensembles : le premier sous-ensemble est terminé.

(c) Cinquième et dernière étape de la construction des sous-ensembles.

FIGURE 3.23 – Construction des sous-ensembles.

au repartitionnement en 2 parties (500+1500 = 2×1000) (figure 3.23b). L'algorithme de construction de la matrice de migration est alors appliqué sur le sous-ensemble $\{0, 3\}$. La partie 0 est trop petite pour créer une hyper-arête de taille 1 (figure 3.24a). La partie 3 a un poids suffisant donc 1000 est ajouté sur la diagonale (figure 3.24b). Il reste alors une hyper-arête à construire : elle part du sommet 0 (figure 3.24c) et s'étend sur le seul sommet voisin 3 (figure 3.24d). Les deux hyper-arêtes nécessaires sont donc construites et toutes les données de 0 et 3 ont été redistribuées.

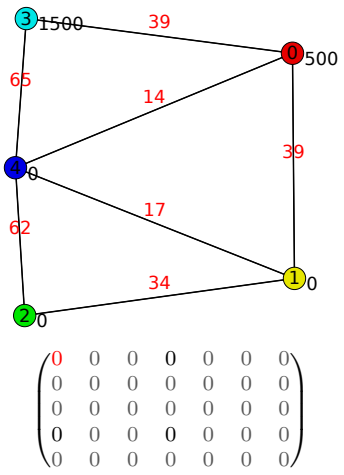
Le second sous-ensemble est construit avec le reste du graphe quotient (figure 3.23c). Les parties 1, 2 et 4 seront repartitionnées vers 5 parties ($900 + 2400 + 1700 = 5 \times 1000$). On commence par construire les hyper-arêtes de taille 1 (figure 3.25a) pour optimiser la diagonale de la matrice, ici 2 et 4 sont de poids suffisants. L'hyper-arête suivante est construite à partir de 1 (figure 3.25b) et s'étend sur le voisin « libre » le plus connecté à 1 : 2 (figure 3.25c). La suivante est construite à partir de 2 qui a encore assez de données (1300) pour construire une hyper-arête de taille 1 (figure 3.25d). La dernière hyper-arête part de 2 (figure 3.25e) et s'étend sur le seul sommet restant, la partie 4 (figure 3.25f).

Par la suite, nous utiliserons systématiquement la recherche des sous-ensembles (*Greedy2*) pour notre algorithme glouton (*Greedy*).

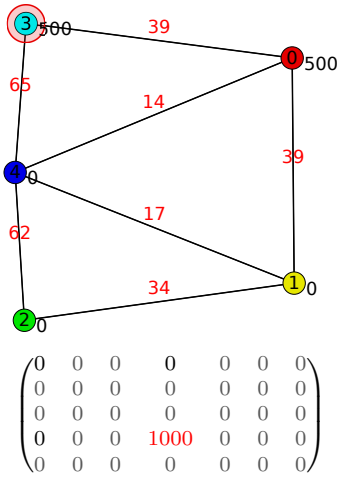
c) Illustration de la méthode gloutonne

Les figures 3.26 et 3.27 présentent des résultats de la méthode gloutonne avec optimisation de la diagonale dans les cas équilibrés et déséquilibrés. Comme avec la méthode d'appariement, de grandes hyper-arêtes sont créées. Cela est dû au remplissage de la diagonale qui laisse de petits restes pour créer les parties supplémentaires. Dans le cas déséquilibré, cet effet est atténué : la diagonale ne peut pas toujours être remplie et les restes peuvent être plus importants.

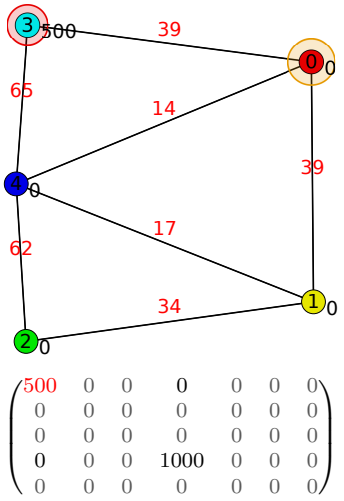
Les figures 3.28 et 3.29 montrent les résultats dans les mêmes cas, mais sans l'optimisation de la diagonale. Les tailles des hyper-arêtes sont bien plus raisonnables. Comme avec la méthode de la chaîne, les nouvelles parties sont souvent placées entre deux anciennes parties voisines.



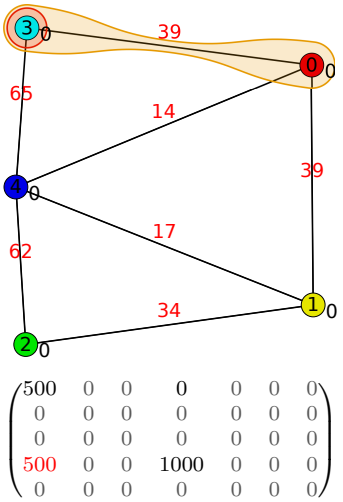
(a) Première étape du remplissage de la diagonale.



(b) Seconde étape du remplissage de la diagonale.

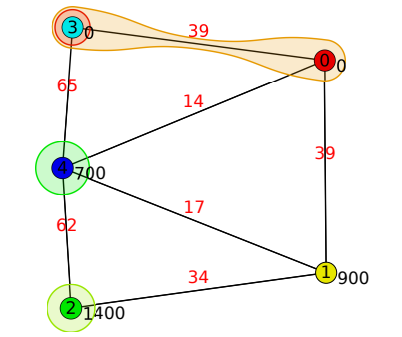


(c) Première étape de la construction de la seconde hyper-arête.



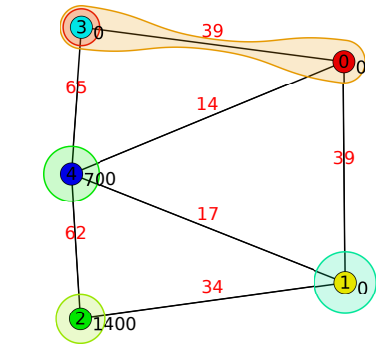
(d) L'hyper-arête est construite.

FIGURE 3.24 – Construction des sous-ensembles et de la sous-matrice correspondant au premier sous-ensemble $\{0, 3\}$.



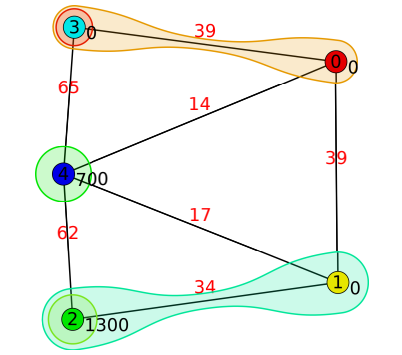
$$\begin{pmatrix} 500 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1000 & 0 & 0 & 0 & 0 \\ 500 & 0 & 0 & 1000 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1000 & 0 & 0 \end{pmatrix}$$

(a) Remplissage de la diagonale.



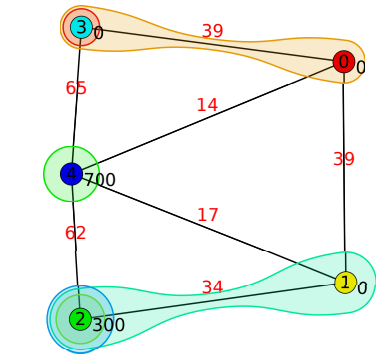
$$\begin{pmatrix} 500 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 900 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1000 & 0 & 0 & 0 & 0 \\ 500 & 0 & 0 & 1000 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1000 & 0 & 0 \end{pmatrix}$$

(b) Première étape de la construction de la troisième hyper-arête.



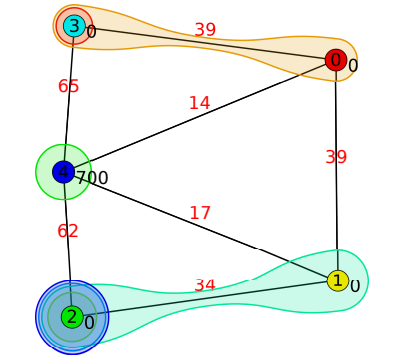
$$\begin{pmatrix} 500 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 900 & 0 & 0 & 0 & 0 & 0 \\ 0 & 100 & 1000 & 0 & 0 & 0 & 0 \\ 500 & 0 & 0 & 1000 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1000 & 0 & 0 \end{pmatrix}$$

(c) Seconde étape de la construction de la troisième hyper-arête.



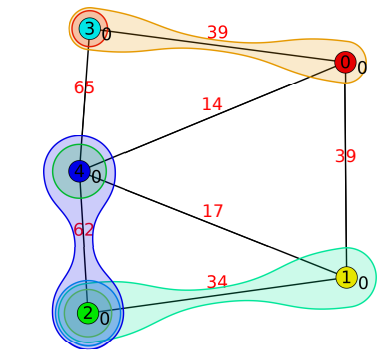
$$\begin{pmatrix} 500 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 900 & 0 & 0 & 0 & 0 & 0 \\ 0 & 100 & 1000 & 0 & 0 & 1000 & 0 \\ 500 & 0 & 0 & 1000 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1000 & 0 & 0 \end{pmatrix}$$

(d) Construction de la quatrième hyper-arête.



$$\begin{pmatrix} 500 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 900 & 0 & 0 & 0 & 0 & 0 \\ 0 & 100 & 1000 & 0 & 0 & 1000 & 300 \\ 500 & 0 & 0 & 1000 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1000 & 0 & 0 \end{pmatrix}$$

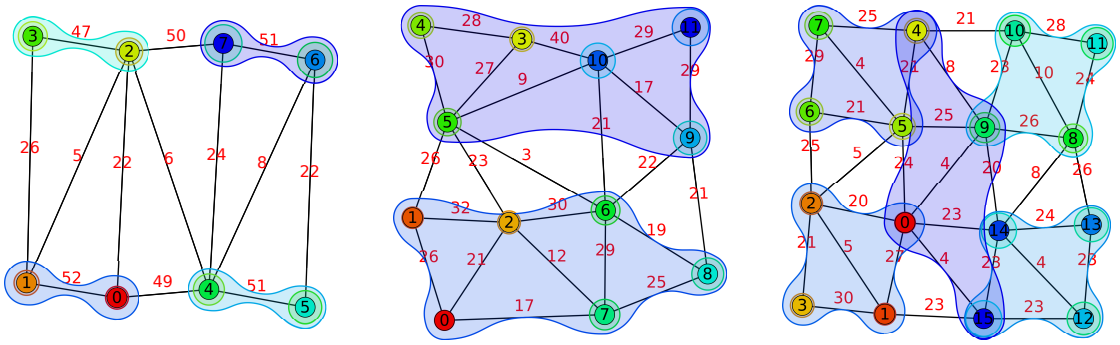
(e) Première étape de la construction de la cinquième hyper-arête.



$$\begin{pmatrix} 500 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 900 & 0 & 0 & 0 & 0 & 0 \\ 0 & 100 & 1000 & 0 & 0 & 1000 & 300 \\ 500 & 0 & 0 & 1000 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1000 & 0 & 700 \end{pmatrix}$$

(f) Seconde étape de la construction de la cinquième hyper-arête et fin de l'algorithme.

FIGURE 3.25 – Construction de la sous-matrice correspondant au second sous-ensemble $\{1, 2, 4\}$.

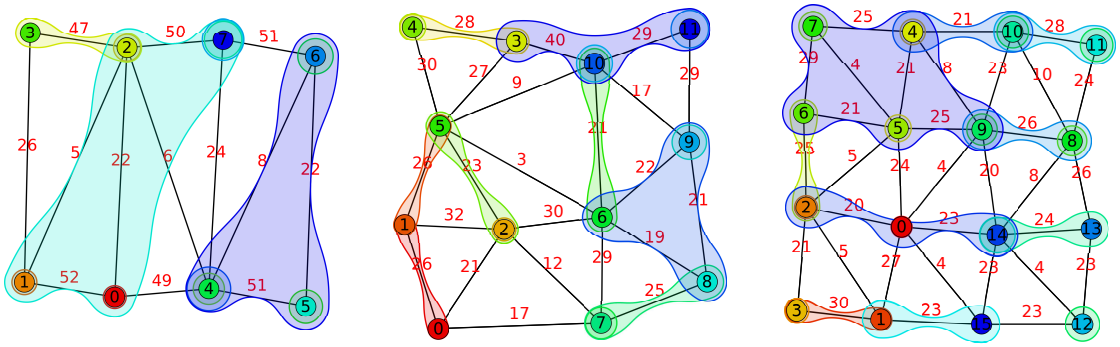


(a) Hypergraphe de repartitionnement dans le cas 8×12 équilibré.

(b) Hypergraphe de repartitionnement dans le cas 12×14 équilibré.

(c) Hypergraphe de repartitionnement dans le cas 16×21 équilibré.

FIGURE 3.26 – Évaluation du repartitionnement à l'aide de l'algorithme glouton avec optimisation de la diagonale dans des cas équilibrés.

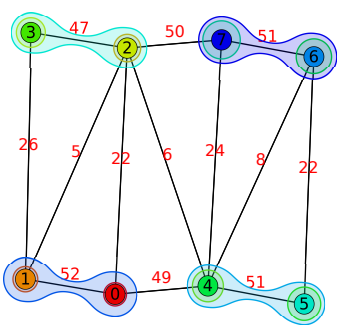


(a) Hypergraphe de repartitionnement dans le cas 8×12 déséquilibré.

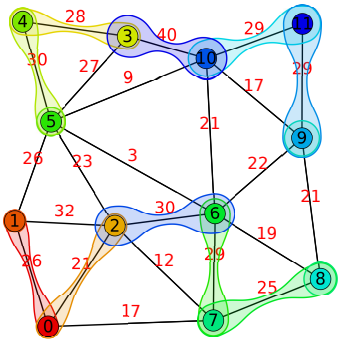
(b) Hypergraphe de repartitionnement dans le cas 12×14 déséquilibré.

(c) Hypergraphe de repartitionnement dans le cas 16×21 déséquilibré.

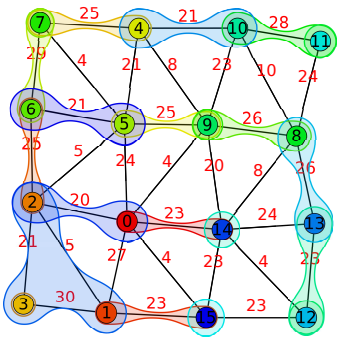
FIGURE 3.27 – Évaluation du repartitionnement à l'aide de l'algorithme glouton avec optimisation de la diagonale dans des cas déséquilibrés.



(a) Hypergraphe de repartitionnement dans le cas 8×12 équilibré.

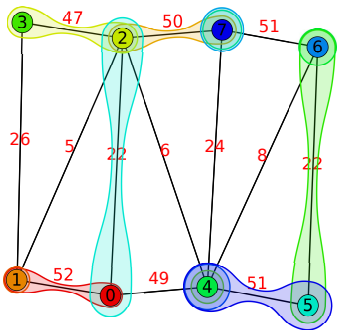


(b) Hypergraphe de repartitionnement dans le cas 12×14 équilibré.

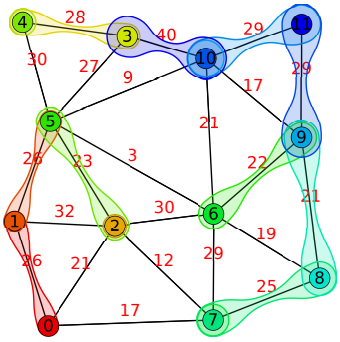


(c) Hypergraphe de repartitionnement dans le cas 16×21 équilibré.

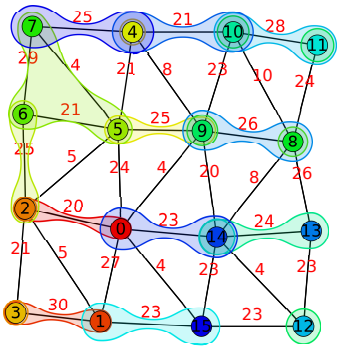
FIGURE 3.28 – Évaluation du repartitionnement à l'aide de l'algorithme glouton sans optimisation de la diagonale dans des cas équilibrés.



(a) Hypergraphe de repartitionnement dans le cas 8×12 déséquilibré.



(b) Hypergraphe de repartitionnement dans le cas 12×14 déséquilibré.



(c) Hypergraphe de repartitionnement dans le cas 16×21 déséquilibré.

FIGURE 3.29 – Évaluation du repartitionnement à l'aide de l'algorithme glouton sans optimisation de la diagonale dans des cas déséquilibrés.

3.3.4 Programme linéaire (*LP*)

Les méthodes de repartitionnement diffusives [26,46] optimisent le volume de migration en effectuant des communications entre les parties voisines sur le graphe quotient (cf. section 2.3.2 b)). Nous proposons d'étendre la méthode en utilisant un programme linéaire dans le cas du repartitionnement avec changement dynamique du nombre de processeurs et pour optimiser les différentes métriques TotalV, MaxV, TotalZ et MaxZ.

Le principal problème de cette adaptation est que le graphe quotient ne prend en compte que les anciennes parties et non les nouvelles. Il faut donc adapter celui-ci pour prendre en compte ce changement de nombre de parties. Pour un repartitionnement de M vers N parties, on utilise un graphe décrivant les messages possibles entre $\max(M, N)$ parties. Le poids de certaines parties peut être initialement nul, si elles n'existaient pas dans l'ancienne partition (cas où $M < N$) ; on peut aussi vouloir un poids final nul pour les parties qui n'existent pas dans la nouvelle partition (cas où $M > N$).

Dans le cas où le nombre de parties augmente ($M < N$), il faut ajouter des nouveaux sommets dans le graphe quotient et, pour que la migration soit possible, connecter ces nouveaux sommets avec le reste du graphe. En effet, Hu *et al.* [26] montrent que pour qu'une solution soit possible, le graphe doit être connexe. On veut donc construire à partir du graphe quotient Q, un nouveau graphe quotient enrichi $\tilde{Q}$. La façon dont sont connectés ces nouveaux sommets influence grandement la migration optimale possible, mais aussi la coupe de la partition finale. Pour obtenir une bonne migration, il faut placer les nouvelles parties proches des anciennes parties capables de fournir les sommets nécessaires. Pour obtenir une bonne coupe, il faut connecter les nouvelles parties avec des anciennes proches entre elles. Une méthode naïve pour trouver de tels placements est de rechercher dans le graphe quotient de petites cliques de parties bien connectées entre elles et possédant un poids élevé.

Dans le cas où le nombre de parties diminue, le graphe quotient n'a pas besoin d'être modifié mais il faut choisir les parties qui seront supprimées, c'est-à-dire dont le poids final visé est nul.

a) Optimisation du volume total de migration TotalV

Une fois le graphe $\tilde{Q}$ des messages autorisés construit, il faut décider du volume de données à communiquer sur chaque arête. Pour minimiser le volume total de migration, il est possible d'utiliser le programme linéaire suivant :

$$\text{minimiser} \quad \sum_{i,j} e_{ij}$$

. Contraintes linéaires :

$$\forall i \in V, \quad v_i = \sum_j (e_{ji} - e_{ij}). \tag{3.18}$$

Bornes :

$$\forall i \in V, \qquad v_i \ \leq d_i + ub. \tag{3.19}$$

$$\forall (i,j) \in E, \quad 0 \leq e_{ij}. \tag{3.20}$$

On note e_{ij} la quantité de données envoyées par la partie i vers la partie j. Cette variable est toujours positive ou nulle (équation 3.20) ; si le message se passe dans l'autre sens, e_{ij} est nul et e_{ji} contient le volume de données échangées. v_i est la variation de la quantité de données de la partie i et est égale à la différence des données reçues et envoyées (équation 3.18). d_i est

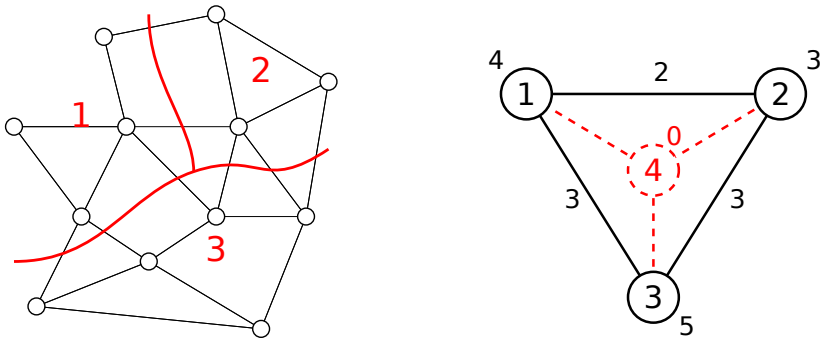


FIGURE 3.30 – Exemple de partition initiale et de son graphe quotient enrichi.

la variation souhaitée de la quantité de données de la partie i, c'est-à-dire la différence entre le poids initial et le poids souhaité. v_i doit être à peu près égal à d_i à une tolérance de déséquilibre ub près (équation 3.19).

Par exemple, en prenant la partition en trois parties de la figure 3.30, on construit un graphe quotient de trois sommets auquel on ajoute un quatrième pour repartitionner en 4 parties. Ici, le nouveau sommet est connecté aux trois autres.

L'application des contraintes 3.18 donne :

$$\begin{aligned} v_1 &= e_{21} - e_{12} + e_{31} - e_{13} + e_{41} - e_{14} \\ v_2 &= e_{12} - e_{21} + e_{21} - e_{23} + e_{42} - e_{24} \\ v_3 &= e_{13} - e_{31} + e_{23} - e_{32} + e_{43} - e_{34} \\ v_4 &= e_{14} - e_{41} + e_{24} - e_{42} + e_{34} - e_{43} \end{aligned}$$

et les bornes 3.19 :

$$\begin{aligned} v_1 &\leq -1 \\ v_2 &\leq 0 \\ v_3 &\leq -2 \\ v_4 &\leq 3. \end{aligned}$$

De plus tous les e_{ij} sont positifs ou nuls. La solution de ce problème est :

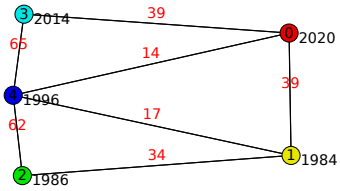
$$\begin{aligned} e_{41} &= 1 \\ e_{43} &= 2 \end{aligned}$$

et tous les autres e_{ij} sont nuls. Ce qui veut dire que la partie 1 devra envoyer un sommet à la partie 4 et la partie 3 devra en envoyer deux. Il n'y a pas d'autre communication.

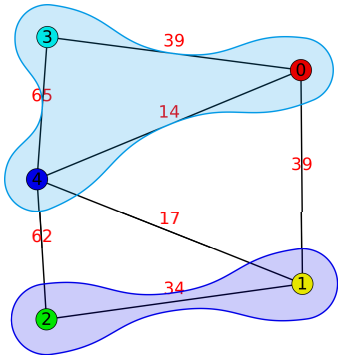
b) Extension à d'autres métriques

Il est possible d'étendre le programme linéaire pour d'autres objectifs, tels que : MAXV, TOTALZ, MAXZ ou n'importe quelle combinaison linéaire de ceux-ci.

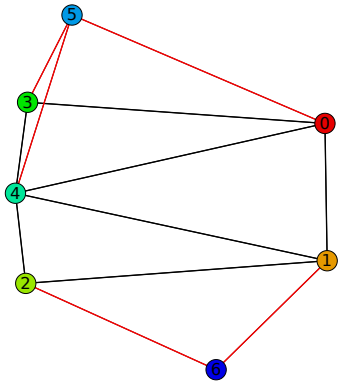
Pour optimiser MAXV, il suffit d'ajouter une variable v représentant le volume maximal de migration par partie. On cherche donc à minimiser $v = \max_{i \in V} \left(\sum_j (e_{ij} + e_{ji}) \right)$. Cela se traduit par les contraintes supplémentaires :



(a) Graphe quotient.



(b) Groupes de sommets retenus pour les nouvelles parties ajoutées.



(c) Graphe quotient enrichi avec les nouveaux sommets.

$$\begin{pmatrix} 1500 & 0 & 0 & 0 & 0 & 520 & 0 \\ 0 & 1184 & 0 & 0 & 0 & 0 & 800 \\ 0 & 0 & 1428 & 0 & 0 & 0 & 558 \\ 0 & 0 & 0 & 1500 & 0 & 514 & 0 \\ 0 & 172 & 0 & 0 & 1500 & 324 & 0 \end{pmatrix}$$

(d) Matrice de migration obtenue pour l'optimisation de TotalV.

FIGURE 3.31 – Enrichissement du graphe quotient puis construction de la matrice de migration pour le cas 5×7.

$$\forall i \in V, \sum_j (e_{ij} + e_{ji}) \leq v. \tag{3.21}$$

v étant minimisé, il vaut bien la plus grande des valeurs $\sum_j (e_{ij} + e_{ji})$.

Lorsque les valeurs e_{ij} sont entières, on peut ajouter des variables binaires x_{ij} pour chaque arête donnant l'existence ou non d'un message sur cette arête grâce aux contraintes :

$$x_{ij} \leq e_{ij} \leq W x_{ij}. \tag{3.22}$$

W est le poids total du graphe, aucun message ne peut dépasser cette taille. En étudiant les deux cas possibles suivant la valeur de x_{ij}, on obtient :

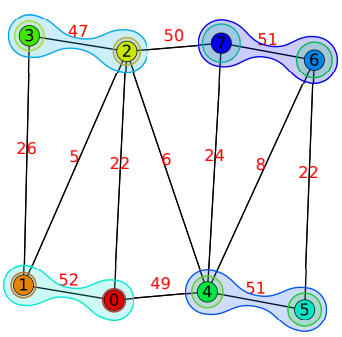
$$\begin{aligned} e_{ij} &= 0 \quad \text{si } x_{ij} = 0 \\ 1 \leq e_{ij} &\leq W \quad \text{si } x_{ij} = 1. \end{aligned} \tag{3.23}$$

x_{ij} correspond donc bien à l'existence d'un message de la partie i vers la partie j. Il est alors possible de minimiser TotalZ en minimisant la somme des x_{ij} et MaxZ en utilisant une méthode similaire à celle pour MaxV.

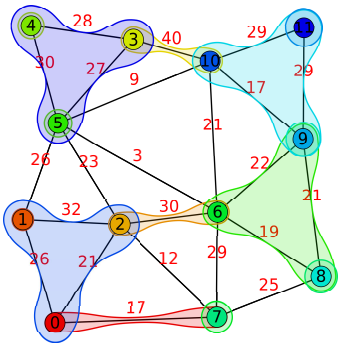
Les programmes linéaires complets pour les différentes métriques sont décrits dans l'annexe A.

c) Application du programme linéaire

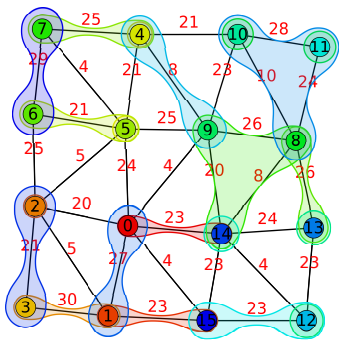
Pour enrichir le graphe quotient, un algorithme recherche, pour chaque nouvelle partie supplémentaire, de petits groupes de sommets (de taille 1, 2 ou 3) bien connectés et de poids importants.



(a) Hypergraphe de repartitionnement dans le cas 8×12 équilibré.



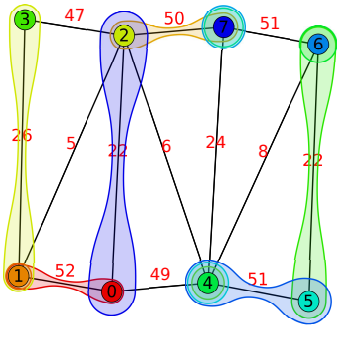
(b) Hypergraphe de repartitionnement dans le cas 12×14 équilibré.



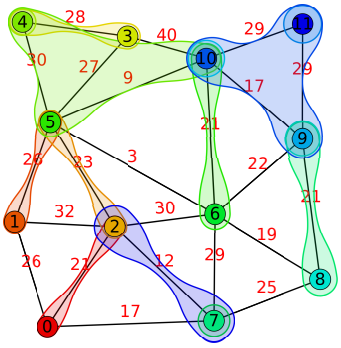
(c) Hypergraphe de repartitionnement dans le cas 16×21 équilibré.

FIGURE 3.32 – Application de la construction de la matrice de migration à l'aide du programme linéaire dans des cas équilibrés.

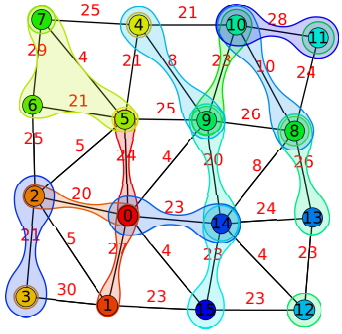
Nous ne détaillerons pas ici cet algorithme qui utilise une approche gloutonne selon un critère comparable à celui de la méthode *Greedy*, mais en bornant la taille des hyper-arêtes (entre 1 et 3). Sur l'exemple de la figure 3.31, pour passer de 5 à 7 parties, il faut ajouter deux nouvelles parties : un groupe de trois sommets et un groupe de deux sont choisis (fig. 3.31b). Les deux nouveaux sommets sont connectés aux anciens sommets d'après ces ensembles (fig. 3.31c). En appliquant le programme linéaire pour l'optimisation de TOTALV sur ce graphe quotient enrichi, on obtient 6 messages : 5 messages correspondant aux arêtes ajoutées pour créer les nouvelles parties supplémentaires et un message entre les parties 4 et 1. Présentés sous forme de matrice de migration, ces résultats donnent la matrice sur la figure 3.31d. Les valeurs sur la diagonale (en gris) ne sont pas obtenues directement par le programme linéaire mais calculées indirectement d'après les tailles des messages et les tailles des anciennes parties.



(a) Hypergraphe de repartitionnement dans le cas 8×12 déséquilibré.



(b) Hypergraphe de repartitionnement dans le cas 12×14 déséquilibré.



(c) Hypergraphe de repartitionnement dans le cas 16×21 déséquilibré.

FIGURE 3.33 – Application de la construction de la matrice de migration à l'aide du programme linéaire dans des cas déséquilibrés.

Les résultats obtenus pour différents cas avec l'optimisation de TOTALV sont présentés sur

les figures 3.32 et 3.33. Les tailles des hyper-arêtes finales correspondant aux nouvelles parties sont effectivement limitées à un maximum de 3 et les autres communications se font uniquement le long des arêtes du graphe quotient.

Cet algorithme optimise la métrique souhaitée, mais relativement à un graphe quotient enrichi donné dont la construction ne permet pas toujours une migration optimale.

3.4 Évaluation des méthodes et conclusion

Nous comparons les méthodes de construction de matrice de migration présentées dans ce chapitre : la méthode basée sur la chaîne (*1D*), la méthode d'appariement (*Matching*, seulement dans le cas équilibré), les deux variantes de l'algorithme glouton avec optimisation de la diagonale (*GreedyD*) et sans (*Greedy*) et le programme linéaire optimisant TotalV (*LP*).

Les méthodes présentées sont évaluées sur six cas équilibrés et déséquilibrés. Les trois cas équilibrés sont :

- 8×12 avec des anciennes parties de taille 1500 et donc des nouvelles parties de taille 1000 ;
- 12×14 avec des anciennes parties de taille 1000 et donc des nouvelles parties de taille 857 ;
- 16×21 avec des anciennes parties de taille 1000 et donc des nouvelles parties de taille 761.

Les trois cas déséquilibrés utilisent les mêmes nombres de parties :

- 8×12 avec un déséquilibre de 49 % et des nouvelles parties de taille 1509 ;
- 12×14 avec un déséquilibre de 58 % et des nouvelles parties de taille 1311 ;
- 16×21 avec un déséquilibre de 50 % et des nouvelles parties de taille 922.

Ces déséquilibres ont été réalisés avec la méthode utilisée par l'article [56] : pour chaque partie, un nombre aléatoire de sommets sont sélectionnés pour avoir leur poids triplés, conduisant à un déséquilibre d'environ 50 %.

Les tableaux 3.1 et 3.2 présentent les métriques associées aux applications présentées dans les sections précédentes. Les quatre métriques présentées ne permettent pas d'évaluer la qualité de la coupe finale mais seulement la qualité de la migration. La coupe ne pourra être évaluée qu'après le repartitionnement dans le chapitre 5.

En dehors du repartitionnement 8×12 dans le cas équilibré, on remarque que les optimisations de TotalV et MaxZ sont contradictoires. En effet, comme expliqué avec la méthode d'appariement, l'optimisation de la diagonale oblige les nouvelles parties supplémentaires à prendre les données depuis de nombreuses anciennes parties, ce qui favorise des hyper-arêtes de grandes tailles et donc l'augmentation de MaxZ. Les méthodes *1D* et *Greedy* donnent un MaxZ bas mais un TotalV plus élevé, contrairement aux méthodes *Matching* (avec choix d'une matrice diagonale) et *GreedyD* qui optimisent en priorité TotalV. Le cas du programme linéaire est particulier : bien que le programme linéaire minimise TotalV, la construction du graphe quotient enrichi impose un degré faible. Malgré cette minimisation, TotalV peut être plus élevé qu'avec d'autres méthodes mais MaxZ est maintenu bas. TotalV aurait pu être plus bas avec un choix de graphe quotient enrichi différent. Il est également possible d'utiliser le programme linéaire pour optimiser d'autres critères.

Dans le cas 8×12 équilibré, il existe une solution donnant à la fois un TotalV bas et un MaxZ bas que toutes les méthodes trouvent. Le programme linéaire est capable de donner un TotalV plus bas que l'optimal (4000) en profitant de la tolérance au déséquilibre.

Le MaxV varie peu d'une méthode à l'autre et il est généralement proche de la taille d'une nouvelle partie. Une nouvelle partie devant recevoir l'intégralité de ses données, il n'est pas

possible d'avoir un MaxV inférieur à la taille idéale d'une nouvelle partie.

Dans le cas équilibré, le TotalZ optimal est bien atteint pour toutes les méthodes sauf le programme linéaire qui n'optimise pas ce critère. Dans le cas déséquilibré, l'optimal n'est pas connu. La méthode de la chaîne atteint $\max(M, N) - 1$ messages et les algorithmes gloutons donne un nombre de messages inférieur grâce à la recherche des sous-ensembles. Bien que le programme linéaire n'optimise pas le nombre de messages, celui-ci reste très bas. C'est un effet de bord de la minimisation de TotalV.

Dans la suite de cette thèse, nous retiendrons les méthodes utilisant l'algorithme glouton (*Greedy* et *GreedyD*) et le programme linéaire (*LP*). L'algorithme glouton permet de minimiser TotalZ et offre deux variantes permettant au choix de minimiser fortement TotalV ou de garder MaxZ bas. La méthode d'appariement (*Matching*) n'est pas utilisable dans le cas général et la méthode *1D* donne des résultats comparables à la méthode *Greedy* (MaxZ faible mais TotalV plus élevé) mais avec de moins bons résultats.

La construction de matrices de migration à partir du graphe quotient est la première étape de notre méthode de repartitionnement dont la démarche globale est rappelée en figure 3.34. Les matrices de migration ainsi construites vont être utilisées par les algorithmes de repartitionnement de graphe présentés dans le chapitre suivant.

		1D (fig. 3.14)	Matching (fig. 3.22)	GreedyD (fig. 3.26)	Greedy (fig. 3.28)	LP (fig. 3.32)
Cas 8×12	TotalV	4000	4000	4000	4000	3960
	MaxV	1000	1000	1000	1000	900
	TotalZ	8	8	8	8	8
	MaxZ	2	2	2	2	2
Cas 12×14	TotalV	3427	1714	1714	3426	2900
	MaxV	857	857	857	857	939
	TotalZ	12	12	12	12	13
	MaxZ	2	6	6	2	3
Cas 16×21	TotalV	5239	3808	3808	5335	5606
	MaxV	762	762	762	762	1150
	TotalZ	20	20	20	20	20
	MaxZ	2	4	4	3	3

Table 3.1 – Résultats des différentes méthodes dans le cas équilibré.

		1D (fig. 3.16)	GreedyD (fig. 3.27)	Greedy (fig. 3.29)	LP (fig. 3.33)
Cas 8×12	TotalV	7453	6266	7127	6299
	MaxV	1871	1869	1869	1862
	TotalZ	11	10	10	10
	MaxZ	2	4	3	2
Cas 12×14	TotalV	5064	5960	5769	5261
	MaxV	1311	1372	1372	2013
	TotalZ	13	12	12	13
	MaxZ	2	3	3	4
Cas 16×21	TotalV	6200	5504	6138	5858
	MaxV	922	922	923	1103
	TotalZ	20	18	18	19
	MaxZ	2	5	3	4

Table 3.2 – Résultats des différentes méthodes dans le cas déséquilibré.

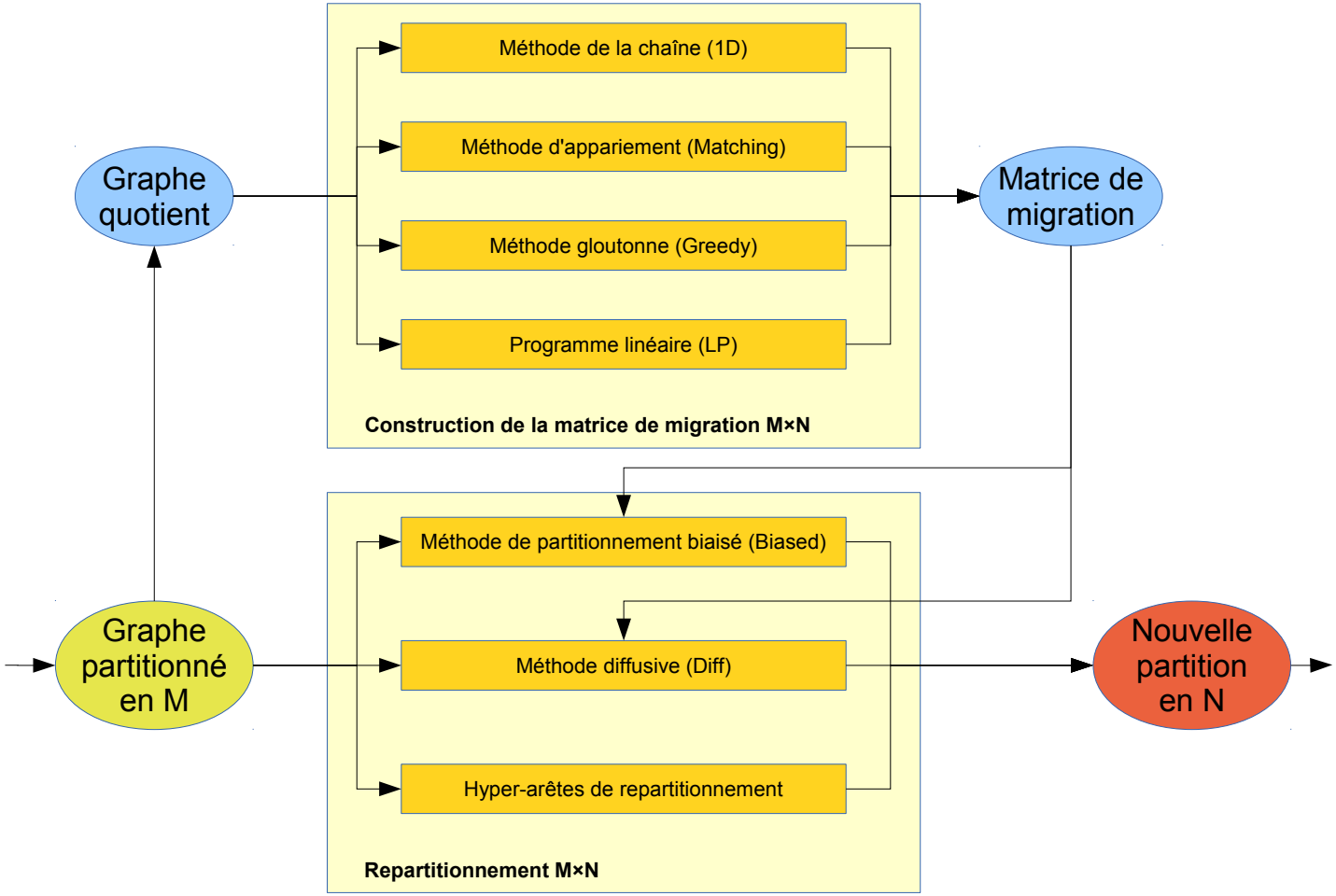


FIGURE 3.34 – Vue d'ensemble.

Chapitre 4

Repartitionnement $M \times N$

Sommaire

4.1 Repartitionnement $M \times N$ basé sur le partitionnement biaisé (*BIASED*) 67
4.1.1 Méthodologie 68
4.1.2 Limitations des partitionneurs 71
4.2 Partitionnement k-aire direct (*KGGGP*) 73
4.2.1 Description de l'algorithme 73
4.2.2 Critères de sélection 75
4.2.3 Complexité 76
4.2.4 Améliorations 77
4.2.5 Évaluation 78
4.3 Repartitionnement $M \times N$ basé sur la diffusion (*DIFF*) 85
4.4 Partitionnement biaisé à l'aide d'hyper-arêtes de repartitionnement 89
4.5 Conclusion 90

Dans ce chapitre, nous présentons nos algorithmes de repartitionnement $M \times N$, résumés en figure 4.1. Nos méthodes de repartitionnement biaisé (*BIASED*) et diffusive (*DIFF*) s'inspirent des méthodes de repartitionnement classiques du mêmes noms mais sont étendues dans le cas du repartitionnement avec un nombre de processeurs variable. Ces deux méthodes se basent sur une matrice de migration obtenue à l'aide d'un des algorithmes présentés dans le chapitre précédent. De plus, dans le cadre de notre partitionnement biaisé, nous mettons en évidence des limitations de la méthode des bissections récursives largement utilisée par les différents outils de partitionnement actuels, et nous proposons une méthode de partitionnement k-aire direct (*KGGGP*) qui surmonte ces limitations.

4.1 Repartitionnement $M \times N$ basé sur une méthode de partitionnement biaisé (*BIASED*)

Nous présentons dans cette section une méthode partitionnement biaisé permettant de réaliser un repartitionnement $M \times N$ en respectant un modèle de migration donné par une matrice C. Cette méthode n'impose que les messages donnés dans le modèle de migration et non leur volume. Néanmoins, les volumes sont respectés lorsque le nombre de messages de C est minimal, comme

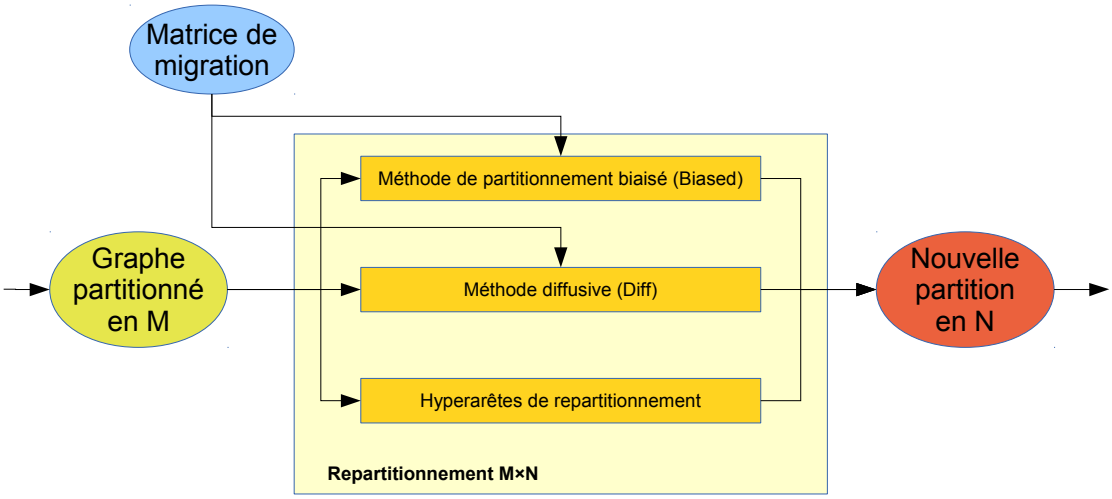


FIGURE 4.1 – Vue d'ensemble de nos algorithmes de repartitionnement $M \times N$ d'un graphe.

nous le verrons dans la section suivante. Cette méthode est donc déconseillée pour les modèles de migration n'optimisant pas TOTALZ, en particulier ceux obtenus à l'aide du programme linéaire optimisant seulement TOTALV, MAXV ou MAXZ.

Cette méthode de partitionnement biaisé s'inspire des méthodes de repartitionnement utilisant des sommets fixes [2,8,9] (cf. section 2.3.2). Le graphe est enrichi avec des sommets fixes et de nouvelles arêtes puis il est partitionné avec un partitionneur acceptant les sommets fixes.

4.1.1 Méthodologie

Pour créer un nouvelle partition respectant le modèle de communication construit grâce aux méthodes présentées dans le chapitre 3, on utilise une méthode de repartitionnement à sommet fixes. De la même façon que dans le cas du repartitionnement sans changement du nombre de processeurs, on ajoute N sommets fixes de poids nuls représentant les processeurs. Mais au lieu de les connecter aux sommets de leur ancienne partie, ils sont connectés aux sommets des parties dont on accepte qu'ils reçoivent des données, conformément à la matrice C.

Plus précisément, pour repartitionner un graphe G de M parties vers N et obtenir une nouvelle partition P', étant donné une ancienne partition $P = (P_i)_{i \in [\![1,M]\!]}$ et une matrice de migration C, il faut :

1. construire un graphe enrichi $\tilde{G} = (\tilde{V}, \tilde{E})$ à partir de G avec :
   — N sommets fixes $(f_j)_{j \in [\![1,N]\!]}$ de poids nuls, chacun fixé dans une nouvelle partie différente j ;
   — des arêtes supplémentaires dont le poids est appelé « coût de migration », telles que pour chaque élément $C_{i,j} > 0$ de la matrice de migration, les sommets de l'ancienne partie i soient connectés au sommet fixe de la nouvelle partie j. Plus formellement,

$$\forall i \in [\![1, M]\!],\ \forall j \in [\![1, N]\!],\ \forall v \in P_i,\ \left(C_{i,j} \neq 0 \iff (v, f_j) \in \tilde{E}\right).$$

2. partitionner le nouveau graphe enrichi $\tilde{G}$ en N parties avec un partitionneur acceptant les sommets fixes ;

3. restreindre la partition de $\tilde{G}$ au graphe original G pour obtenir P'.

Le rôle des arêtes ajoutées, dites « de migration », n'est pas exactement le même que dans les méthodes classiques. Elles ne servent pas, ici, à minimiser la migration directement mais à imposer le schéma de communication choisi. Un sommet est souvent relié à plusieurs sommets fixes et au plus une de ses arêtes de migration peut ne pas être coupée. Avec un poids suffisamment élevé, le partitionneur laissera exactement une arête de migration par sommet non-coupée, pour minimiser la coupe. Ainsi, en cherchant à remplir ses objectifs habituels, le partitionneur impose le schéma de migration choisi.

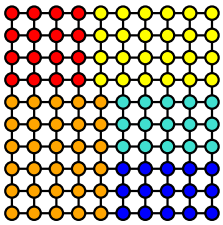
Un exemple de repartitionnement est proposé en figure 4.2. En partant de l'ancienne partition en $M = 5$ parties (figure 4.2a), on veut construire une nouvelle partition en $N = 7$ parties. À l'aide du graphe quotient associé (figure 4.2b), on construit une matrice de migration (ici représentée par l'hypergraphe de repartitionnement) comme indiqué dans le chapitre précédent (figure 4.2c). Les arêtes de migration sont ajoutées d'après cet hypergraphe sur la figure 4.2d : par exemple, le sommet fixe de la nouvelle partie 3 est relié à tous les sommets des parties contenues dans l'hyper-arête correspondante (les anciennes parties 3 et 4). Le partitionneur calcule ensuite une nouvelle partition de ce graphe enrichi. La figure 4.2e montre les arêtes de migration coupées (en rouge) et internes (en vert). Enfin, la partition de $\tilde{G}$ est restreinte graphe G pour obtenir la partition finale (figure 4.2f).

Cette méthode utilise n'importe quel outil de partitionnement classique capable de prendre en compte le cas des sommets fixes [1]. Il est aussi possible d'utiliser des partitionneurs d'hypergraphe, les arêtes de migration deviennent alors des hyper-arêtes de taille 2, la coupe de telles hyper-arêtes étant équivalente à celle des arêtes du graphe.

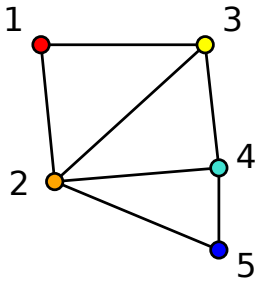
Les sommets fixes et les arêtes de migration imposent le schéma de communication mais pas le volume des messages échangés. Affecter un sommet à une partie de façon à engendrer un message non autorisé correspond à une arête de migration coupée en plus. Un sommet est relié par des arêtes de migration à un ou plusieurs sommets fixes chacun dans une partie différente. Au mieux une seule de ces arêtes de migration n'est pas coupée (le sommet ne peut être que dans une seule partie), alors la migration de ce sommet est autorisé par la matrice de migration. Si le sommet est dans une partie différente de celles des sommets fixes auxquels il est connecté, la migration de ce sommet n'est pas autorisée et toutes les arêtes de migration de ce sommets sont coupées. Le partitionneur choisira donc de placer un sommet dans l'une des parties des sommets fixes auxquels il est connecté. La contrainte sur la taille des parties n'empêche pas ce choix car, par construction de la matrice de migration, il y a toujours assez de sommets reliés au sommet fixe pour créer la partie de la taille souhaitée parmi ces derniers.

Bien que seul le schéma de communication soit imposé, les tailles des messages sont respectées lorsque le nombre de message (TotalZ) est minimal. Le partitionneur est libre de créer n'importe quelle partition induisant une matrice de migration avec les non-zéros souhaités. On cherche le nombre de matrices de migration C' qu'il est possible d'obtenir après le partitionnement. L'espace des matrices de migration possibles peut être défini par un système d'équations linéaires avec autant d'inconnues, notées $C'_{i,j}$, que de messages (d'une ancienne partie i vers une nouvelle partie j). Les équations sont les contraintes de sommes sur les lignes et les colonnes ; il y en a $M+N$. Il y a autant d'inconnues $C'_{i,j}$ que d'éléments $C_{i,j}$ non nuls dans la matrice de migration C souhaitée. En notant V_i le poids de l'ancienne partie i, V'_j la poids de la nouvelle partie j et W le poids

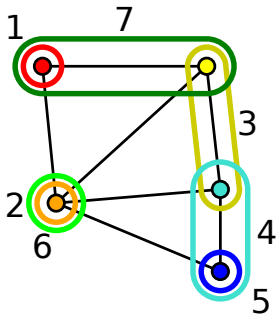
1. Il est en fait possible d'obtenir un partitionnement équivalent sans sommets fixes : il suffit de donner aux sommets « fixes » un poids suffisamment grand, pour qu'il ne puisse pas y en avoir deux dans une même partie. Les nouvelles parties sont ensuite renumérotées pour que les sommets fixes soient dans les bonnes parties. Le défaut de cette méthode est que le déséquilibre toléré est plus important à cause du poids de ces sommets.



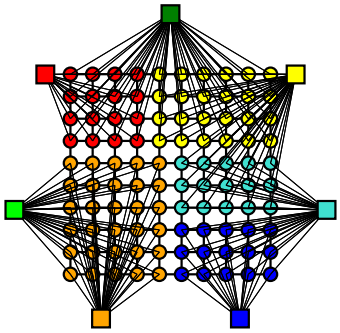
(a) Partition initiale de G en 5 parties.



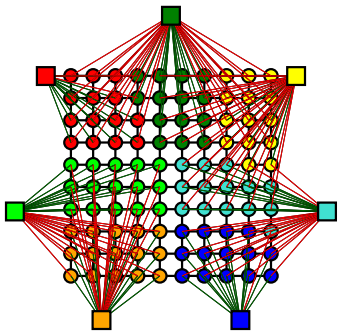
(b) Graphe quotient Q de la partition initiale.



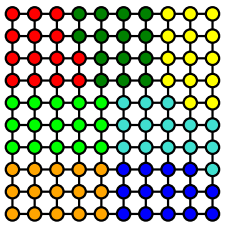
(c) Hypergraphe de repartitionnement dessiné par dessus le graphe quotient.



(d) Graphe $\tilde{G}$ auquel est ajouté les sommets fixes et arêtes de migration.



(e) Partition de $\tilde{G}$ en 7 parties. Les arêtes de migration coupées sont en rouge, les arêtes de migration non-coupées en vert.



(f) Partition finale de G en 7 parties.

FIGURE 4.2 – Repartitionnement de 5 vers 7 parties.

total du graphe ($W = \sum_i V_i = \sum_j V'_j$), on a :

$$\forall i \in [\![1, M]\!], \sum_{\substack{j \in [\![1,N]\!] \\ C_{i,j}>0}} C'_{i,j} = V_i \tag{4.1}$$

$$\forall j \in [\![1, N]\!], \sum_{\substack{i \in [\![1,M]\!] \\ C_{i,j}>0}} C'_{i,j} = V'_j \tag{4.2}$$

La somme des équations sur les lignes (équations 4.1) est équivalente à la somme des équations sur les colonnes (équations 4.2) et donnent :

$$\sum_{\substack{i \in [\![1,M]\!] \\ j \in [\![1,N]\!] \\ C_{i,j}>0}} C'_{i,j} = W$$

Ce système possède donc $M + N - 1$ équations indépendantes [2].

On sait que l'ensemble des solutions n'est pas vide car la matrice de migration C est une solution. La dimension de l'espace solution est donc égale à la différence entre le nombre d'inconnues (le nombre de messages dans la matrice C) et le rang du système. Si le nombre de messages dans C n'est pas minimal (il n'atteint pas le rang du système), il existe une infinité de solutions C' possibles. Le partitionneur peut donner une de ces matrice C' (éventuellement différente de C) en minimisant la coupe du graphe enrichi. Plus le nombre de messages dans C est important plus il y a de liberté dans le partitionnement. La matrice de migration C' effectivement obtenue après repartitionnement peut ne pas être aussi bonne que la matrice C choisie. On préférera donc utiliser ce partitionnement biaisé avec des matrices de migration donnant un faible nombre de messages (TotalZ).

4.1.2 Limitations des partitionneurs

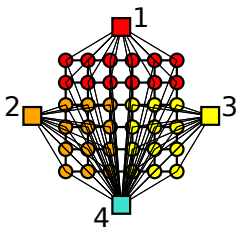
Cette méthode partitionnement à sommets fixes peut poser quelques problèmes avec certains partitionneurs, plus particulièrement ceux utilisant des bissections récursives [3].

Le premier problème est que le placement et la numérotation des nouvelles parties peut ne pas être favorable aux bissections récursives. Comme expliqué dans la section 3.3.1, lors d'une bissection, chaque moitié regroupera plusieurs parties suivant leurs numéros : généralement les parties avec les plus petits numéros sont regroupées dans une moitié et les plus grand numéros dans l'autre. Il est possible qu'une moitié regroupe des parties qu'on souhaite éloignées. Les heuristiques peuvent avoir plus de mal à créer des parties et devoir travailler avec des graphes non connexes.

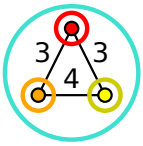
Mais, certains cas sont encore plus graves. Même avec des bissections parfaites et un poids très important sur les arêtes de migration, un partitionneur à bissections récursives peut ne pas

2. Il est possible de réduire encore le nombre d'équations indépendantes lorsqu'il existe une somme partielle de poids d'anciennes parties égale à une somme partielle de poids de nouvelles parties. C'est-à-dire que s'il existe $A \subset [\![1, M]\!]$ et $B \subset [\![1, N]\!]$ tels que $\sum_{i \in A} V_i = \sum_{j \in B} V'_j$ et que pour tout $(i, j) \in [\![1, M]\!] \times [\![1, N]\!]$ tels que $(i \in A \wedge j \notin B) \vee (i \notin A \wedge j \in B)$, $C_{i,j} = 0$, alors la somme des équations sur les lignes correspondant aux anciennes parties de A est équivalente à la somme des équations sur les colonnes correspondant aux nouvelles parties de B :

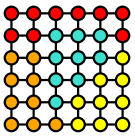
$$\sum_{\substack{i \in A \\ j \in B \\ C_{i,j}>0}} C'_{i,j} = \sum_{i \in A} V_i = \sum_{j \in B} V'_j$$



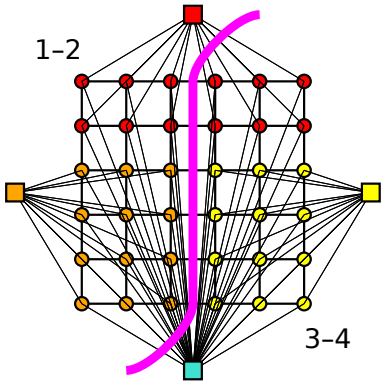
(a) Partition intiale et sommets fixes.



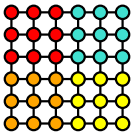
(b) Graphe quotient et hypergraphe de repartitionnement.



(c) Partition finale k-aire direct.



(d) Première bissection.



(e) Partition finale avec bissections récursives.

FIGURE 4.3 – Exemple d'échec de partitionnement avec bissections récursives.

respecter le schéma de communication imposé alors qu'un partitionneur k-aire direct trouverait une solution [3, 60].

Par exemple, la figure 4.3 présente un cas où la méthode des bissections récursives échoue. La grille est initialement partitionnée en 3 (figure 4.3a) et on souhaite la repartitionner en 4 en utilisant l'hypergraphe de repartitionnement de la figure 4.3b. Un exemple de partition finale respectant ce schéma est présenté sur la figure 4.3c. On essaye d'appliquer les bissections récursives sur ce graphe. Lors de la première bissection (figure 4.3d), le partitionneur regroupe les parties 1 et 2 d'un coté, et 3 et 4 de l'autre. La meilleure bissection possible est de coupée le graphe verticalement par le milieu. Les parties 1 et 2 seront donc dans la partie gauche et les parties 3 et 4 dans la partie droite. Il n'est donc pas possible pour la partie 4 de reprendre des sommets de l'ancienne partie 2. Chaque moitié est ensuite à nouveau bipartitionnée pour obtenir la partition finale présentée sur la figure 4.3e. Sa matrice de migration est :

$$\begin{pmatrix} 6 & & & 6 \\ 3 & 9 & & \\ & & 9 & 3 \end{pmatrix},$$

alors que celle attendue, qui est celle de la partition de la figure 4.3c, est :

$$\begin{pmatrix} 9 & & & 3 \\ & 9 & & 3 \\ & & 9 & 3 \end{pmatrix}.$$

La partition de bissections récursives coupent 39 arêtes de migration alors qu'il est possible de n'en couper seulement 36, même si on a supposé que les bissections étaient parfaites et le poids des arêtes de migration bien plus important que celui des arêtes internes.

En pratique, la plupart des outils utilisent un multi-niveaux k-aire avec une partition initiale du graphe contracté calculée par bissections récursives. Dans le cas où les bissections récursives échouent à calculer une bonne partition initiale, le raffinement k-aire appliqué pendant la phase d'expansion doit alors grandement modifier la partition initiale pour optimiser la coupe. Quand l'heuristique de raffinement réussit à corriger la partition initiale, ce qui est difficile comme ces heuristiques sont souvent incrémentales, le partitionneur est alors très ralenti par la phase d'expansion.

Pour cette raison, nous proposons une heuristique de partitionnement k-aire direct pouvant être utilisée à la place des bissections récursives en tant que méthode de partitionnement initiale dans un partitionneur.

4.2 Partitionnement k-aire direct (*KGGGP*)

L'heuristique du *Greedy Graph Growing Partitioning* (GGGP) est une heuristique gloutonne largement utilisée dans le cadre du bipartitionnement [5, 11, 29]. En partant de deux sommets « graines », les autres sommets sont ajoutés un à un dans chacune des parties. Nous proposons d'étendre cette méthode dans le cas du partitionnement k-aire. Nous l'appellerons dans ce cas KGGGP (*k-way greedy graph growing partitioning*).

4.2.1 Description de l'algorithme

Cette heuristique s'inspire en grande partie de celle de Fiduccia et Mattheyses [17] (abrégée FM), à la différence que les sommets ne sont initialement dans aucune partie et sont à distribuer dans k parties. Cette méthode peut être vue comme un raffinement FM $k+1$-aire où la partie supplémentaire (que l'on numérotera -1) contient initialement tous les sommets et doit être vide dans la partition finale.

Pour réaliser cet algorithme, on considère tous les déplacements possibles des sommets sans parties vers une des k parties. Chaque itération de l'algorithme 5 place un sommet sans partie dans une partie p. À chaque itération, on sélectionne le meilleur déplacement, d'après un *critère de sélection* basé sur la coupe, qui respecte l'équilibre final (c'est-à-dire que le poids W_p de la partie p ne dépassera pas la taille maximum permise avec un facteur de déséquilibre ϵ). Il existe plusieurs critères de sélection qui seront détaillés plus loin dans cette section. Il est possible qu'aucun sommet ne respecte la contrainte d'équilibre, particulièrement dans le cas d'un graphe contracté dans le cadre d'un algorithme multi-niveaux, où des sommets peuvent avoir un poids très important par rapport au poids total du graphe. Dans ce cas, l'équilibre ne pourra pas être respecté et le meilleur déplacement est choisi indépendamment du poids du sommet et de la partie cible. Une fois un sommet sélectionné, il est alors déplacé et verrouillé, c'est-à-dire qu'il ne sera plus considéré pour d'autres déplacements.

Le critère de sélection calcule le score $s_i(u)$ d'un déplacement à partir des parties de ses voisins (cf. section suivante). Ainsi lorsqu'un sommet est déplacé, le critère de sélection de chacun de ses voisins doit être mis à jour.

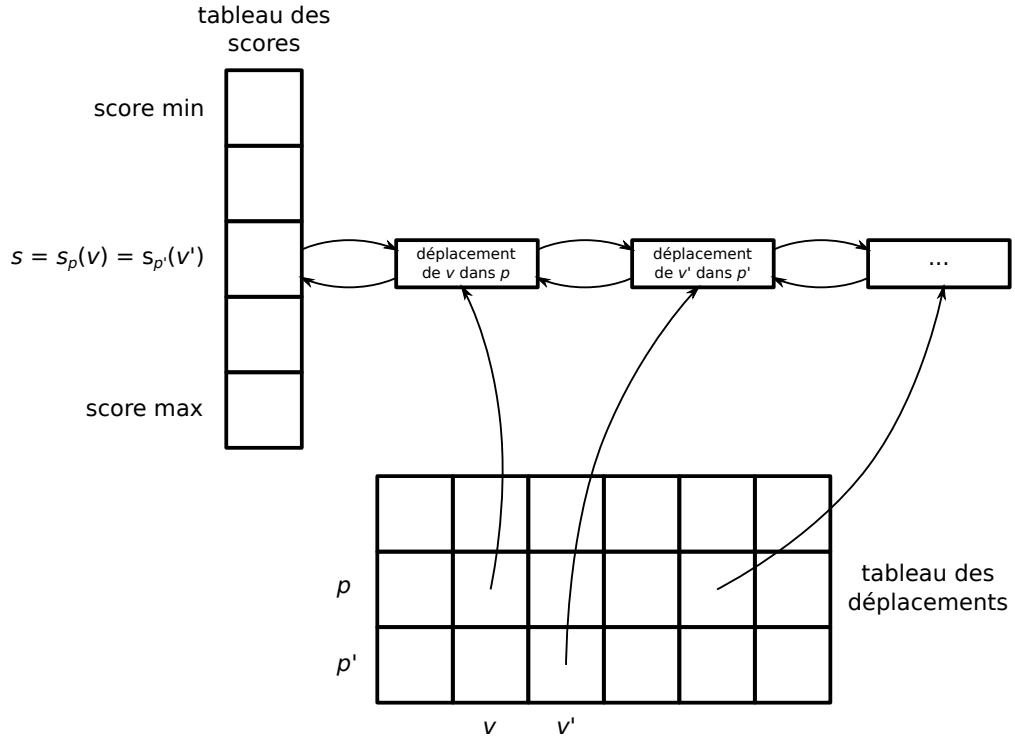


FIGURE 4.4 – Structure de données inspirée de Fiduccia et Mattheyses, illustrant le cas des déplacements de v dans p et de v' dans p' ayant un même score s.

L'algorithme 5 sélectionne le meilleur déplacement globalement. Mais il existe d'autres variantes possibles pour sélectionner un déplacement afin d'obtenir une partition équilibrée [29]. Par exemple, la partie de destination du déplacement est choisie l'une après l'autre (méthode appelée *round-robin*), ou elle peut être la partie dont le poids courant est le plus faible. Dans ces deux cas, on choisit le meilleur déplacement seulement vers la partie choisie.

Pour trouver rapidement le meilleur déplacement d'après le critère choisi, la structure de données présentée par Fiduccia et Mattheyses est utilisée. Cette structure de données (résumée sur la figure 4.4) utilise un tableau à chaque case duquel est associée une valeur du score utilisée comme critère de sélection (le gain de coupe du déplacement dans le cas de FM). Chaque case de ce tableau donne la liste doublement chainée des déplacements correspondant à ce score. Ce tableau des scores permet de sélectionner rapidement un déplacement avec un gain donné (ici, le plus grand). Un autre tableau référençant les éléments des listes chainées par leur numéro de sommet permet de les retrouver rapidement pour effectuer une mise à jour du gain. Dans notre algorithme k-aire ce tableau est étendu avec une seconde dimension correspondant à la partie du déplacement. Le double chainage permet d'effectuer en temps constant le changement de liste d'un élément après une mise à jour ou son retrait quand le sommet est placé dans une partie.

En pratique, notre mise en œuvre de cet algorithme utilise deux tableaux de score pour classer les déplacements. Initialement, le premier contient tous les déplacements et le second est vide. Les déplacements sont recherchés d'abord dans la première structure, lorsque qu'un déplacement ne respectant pas la contrainte d'équilibre est trouvé, il est déplacé dans la seconde. En effet, le poids des parties ne faisant que croître, il ne respectera pas la contrainte d'équilibre dans le futur, il est inutile de le considérer à nouveau pour être déplacer. Quand la première structure

est vide, il n'y a plus de déplacements respectant la contrainte d'équilibre, la seconde structure est alors utilisée jusqu'à la fin de l'algorithme et le poids de la partie n'est plus vérifié.

Il est très simple de gérer le cas des sommets fixes avec cette méthode. En effet, les sommets fixes sont simplement placés dans leur parties respectives initialement et ne sont pas considérés dans les déplacements possibles.

Algorithme 5 Heuristique de partitionnement gloutonne (*KGGGP*)

Entrée : Graphe G à partitionner
Entrée : k parties ne contenant que les sommets fixes
Entrée : Partition où tout les sommets fixes sont dans leur partie respective, et les autres sommets sont dans la partie -1
  tant que il existe des sommets dans la partie -1 **faire**
    Sélectionner un sommet v sans partie dont le déplacement vers une partie p a un critère de sélection $s_p(v)$ maximal et tel que $W_p + w_v \leq (1+\epsilon) \times W/k$
    si un tel sommet n'existe pas **alors**
      Sélectionner un sommet v sans partie dont le déplacement vers une partie p a un critère de sélection $s_p(v)$ maximal
    fin si
    Placer le sommet v dans la partie p
    $W_p \leftarrow W_p + w_v$
    pour tout voisin u de v **faire**
      pour tout partie i **faire**
        Mettre à jour le critère de sélection $s_i(u)$
      fin pour
    fin pour
  fin tant que
  retourner Partition de G en k parties

4.2.2 Critères de sélection

Il existe plusieurs critères pour évaluer la qualité des déplacements. On note $s_p(v)$, le score du déplacement du sommet v vers la partie p. Les déplacements sont évalués en fonction des arêtes les connectant aux différentes parties. On notera $N_i(u)$ le poids des arêtes connectant le sommet u à des voisins dans la partie i, et $N_{-1}(u)$ le poids des arêtes le connectant à des sommets sans partie. On appelle arête interne, une arête dont les deux extrémités sont dans la même partie et on appelle arête externe (ou coupée), une arête dont les deux extrémités sont dans des parties différentes.

Comme avec la méthode FM, il est possible d'utiliser la formule du gain de coupe en considérant que les sommets sans parties sont dans une partie numérotée -1. Le gain d'un sommet u déplacé de la partie i vers j se calcule par la différence entre les arêtes vers la nouvelle partie j, qui étaient coupées mais ne le seront plus, et les arêtes vers l'ancienne partie i, qui étaient internes et seront coupées. En reprenant la formule du gain : $g_{i\to j}(u) = N_j(u) - N_i(u)$, le critère de sélection basé sur le gain pour déplacer u dans i est $s_i(u) = g_{-1\to i}(u) = N_i(u) - N_{-1}(u)$.

Battiti et Bertossi [5] proposent d'utiliser un autre critère, appelé *diff*. Bien que conçu pour le bipartitionnement, ce critère peut également s'appliquer dans le cas du partitionnement k-aire. Ce critère ne s'intéresse plus aux voisins sans partie, mais utilise la différence entre les nouvelles arêtes internes (voisins dans la partie visée) et les nouvelles arêtes coupées (voisins dans les autres

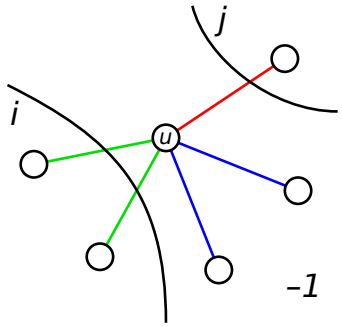


FIGURE 4.5 – Différentes arêtes autour du sommet sans partie u qui se trouve à la frontière des parties i et j.

parties) : $s_i(u) = \mathit{diff}_i(u) = N_i(u) - \sum_{j \neq i} N_j(u)$. En effet, minimiser la coupe avec la partie -1 qui finira par disparaitre peut paraitre inutile alors que les arêtes vers les autres parties font partie de la coupe finale. D'autre part, en maximisant le nombre d'arêtes internes ($N_i(u)$), on espère réduire le nombre d'arêtes qui devront être coupées plus tard.

La figure 4.5 présente les différentes arêtes autour du sommet u : en vert, les arêtes correspondant à $N_i(u)$; en rouge à $N_j(u)$; et en bleu à $N_{-1}(u)$. On suppose que toutes les arêtes de cet exemple ont un poids unitaire. Si on considère le déplacement de u dans la partie i, le gain est $g_{-1 \to i}(u) = 0$: initialement 3 arêtes sont coupées (en vert et rouge), après déplacement 3 arêtes sont toujours coupées (cette fois en bleu et rouge). Le critère *diff* ne s'intéresse pas aux arêtes avec les sommets sans partie (en bleu) mais seulement à la différence entre les futures arêtes internes (en vert) et les arêtes externes avec les « vraies » parties (en rouge). On a donc $\mathit{diff}_i(u) = 2 - 1 = 1$.

4.2.3 Complexité

L'algorithme de partitionnement déplace une fois chaque sommet libre. Il y a donc autant d'itérations de la boucle principale que de sommets non fixes (au plus $|V|$). La boucle principale comporte les étapes de sélection, déplacement et mise à jour des structures de données.

Lors de la sélection, chaque déplacement n'est considéré qu'une ou deux fois : s'il n'est pas accepté la première fois, il est retiré de la première liste des déplacements et placé dans la seconde ; dans la seconde liste, un déplacement n'est jamais rejeté. La complexité en temps totale de la sélection est donc au pire proportionnelle au nombre de déplacements possibles, c'est-à-dire $O(k|V|)$.

Le déplacement est une opération simple en temps constant. Cela n'est que le changement d'une valeur dans le tableau représentant la partition.

La mise à jour des structures de données après le déplacement d'un sommet, nécessite de visiter tous les voisins du sommet déplacé. Pour chaque voisin, il est nécessaire de mettre à jour le critère de sélection vers chacune des k parties. Comme ces opérations sont effectuées pour chaque sommet du graphe, la complexité en temps totale des mises à jour est au pire $O(k|E|)$.

La complexité en temps de l'algorithme est donc au pire $O(k|E|)$, en considérant que $|V|$ est dominé par $|E|$. La complexité en mémoire est principalement due au stockage des déplacements possibles, elle est donc au pire $O(k|V|)$.

4.2.4 Améliorations

Plusieurs variantes peuvent être utilisées pour améliorer la qualité des partitions produites par l'algorithme.

a) Sélection aléatoire et répétition

Lors de la sélection des sommets, plusieurs peuvent avoir le même score. Si le premier sommet trouvé est choisi, la sélection est biaisée par l'ordre de parcours des sommets ou par le dernier déplacement. En effet, lors de la mise à jour, les voisins dont le score s'est amélioré se retrouvent en début de liste, ce qui tend à faire grossir la partie toujours dans la même direction. En sélectionnant un sommet aléatoire parmi les meilleurs, on évite ce biais. Cela permet d'obtenir des partitions différentes avec différentes exécutions de l'algorithme. Comme cet algorithme est très rapide lorsqu'il est appliqué dans le cadre d'un partitionnement multi-niveaux, il devient intéressant de le répéter plusieurs fois et de garder la meilleure partition. Cela permet aussi une version parallèle triviale où chaque nœud calcule une partition puis partage la meilleure.

b) Graines

Bien que l'algorithme puisse s'appliquer sans aucun sommet initialement dans une partie, les critères de sélection ne seront pas très efficaces pour le choix des premiers sommets. Il est donc intéressant de choisir avant le début de l'algorithme certains sommets qui initieront les parties. Il existe plusieurs possibilités pour choisir ces sommets qu'on appelle « graines ».

Graines aléatoires. k sommets sont choisis aléatoirement dans le graphe pour servir de graines à chacune des parties. Ces techniques se combinent bien avec la répétition de l'heuristique et permettent encore plus de variété dans les tentatives pour trouver la meilleure partition possible.

Graines au centre des parties. La répétition de l'heuristique peut également être utilisée pour trouver des graines de façon plus intelligente. Après une première passe à l'aide de graines choisies d'une manière quelconque, les centres [3] ou des « pseudo-centres [4] » des parties sont calculés et servent de graine à l'itération suivante. Cette méthode est proposée par Diekmann *et al.* dans le cadre du *Bubble Partitioning* [14].

Graines éloignées. Diekmann *et al.* [14] proposent également un algorithme pour optimiser l'ensemble des graines initiales. L'algorithme 6 permet de construire un ensemble de k graines bien éloignées les unes des autres. En partant d'un sommet quelconque, on recherche le sommet le plus éloigné à l'aide d'un parcours en largeur. Les graines suivantes sont trouvées de façon similaire en initialisant le parcours en largeur avec les n graines déjà trouvées. La seconde boucle permet d'améliorer cet ensemble. On retire une à une les graines de l'ensemble et on en cherche une nouvelle, toujours à l'aide d'un parcours en largeur. Cette dernière boucle peut être répétée jusqu'à ce que l'ensemble ne change plus ou répétée un nombre limité de fois. La complexité d'un parcours en largeur étant $O(|E|)$, la complexité de la création de l'ensemble initiale ou d'une passe d'optimisation est $O(k|E|)$.

3. Le centre d'un graphe est un sommet avec une excentricité minimale.
4. La recherche du centre d'un graphe est complexe. Nous préférons donc chercher un « pseudo-centre » à l'aide d'un parcours en largeur à partir de la frontière de la partie. Le dernier sommet parcouru est le plus éloigné de la frontière, il est donc sélectionné en tant que « pseudo-centre »

Algorithme 6 Création d'un ensemble de k graines éloignées

Entrée : Graphe G
Entrée : Nombre de parties k
  $v \leftarrow$ sommet quelconque de G
  $graines \leftarrow$ {sommet le plus éloigné de v dans G}
  $n \leftarrow 1$
  tant que $n < k$ **faire**
    $v \leftarrow$ sommet le plus éloigné des sommets dans $graines$
    Ajouter v dans $graines$
    $n \leftarrow n + 1$
  fin tant que
  répéter
    pour tout $v \in graines$ **faire**
      $u \leftarrow$ sommet le plus éloigné des sommets dans $graines \setminus \{v\}$
      si u est plus loin que v **alors**
        Remplacer v par u dans $graines$
      fin si
    fin pour
  jusqu'à Aucune graine n'a été changée
  retourner $graines$

c) Raffinement à la fin

Les derniers choix de déplacement lors du KGGGP sont souvent très limités et les derniers sommets se retrouvent placés dans les parties qui pouvaient encore accepter des sommets malgré une très mauvaise coupe. Pour corriger ces défauts, il est nécessaire d'appliquer un léger raffinement de type FM à la partition obtenue.

Bien que très léger, ces raffinements peuvent grandement améliorer la coupe, il est donc intéressant de le prendre en compte dans la répétition de l'algorithme glouton, que ce soit pour décider la partition à garder ou calculer de nouvelles graines à l'aide des centres des parties.

4.2.5 Évaluation

Cette méthode de partitionnement a été mise en œuvre dans un module *Scotch* [48]. L'avantage de *Scotch* est d'être un logiciel libre très modulaire. Cela permet de réutiliser les autres méthodes déjà présentes dans *Scotch*, en particulier l'approche multi-niveaux et les méthodes de raffinement. Pour combiner ces différentes stratégies de partitionnement, *Scotch* utilise des *chaînes de stratégies* permettant de configurer l'enchainement, l'imbrication et les paramètres des différentes stratégies. Notre stratégie de KGGGP est documentée dans l'annexe B.

Pour évaluer le KGGGP, notre méthode est comparée à la stratégie de bissections récursives de *Scotch*. La stratégie utilisée pour les bissections récursives (RB) dans les expériences suivantes est celle utilisée par défaut dans *Scotch*, elle est donnée par la chaîne de stratégie suivante :

```
r{job=t,map=t,poli=S,bal=0.01,sep=
  m{vert=120,low=h{pass=10}f{bal=0.01,move=120},
    asc=b{bnd=f{bal=0.01,move=120},org=f{bal=0.01,move=120}}}|
  m{vert=120,low=h{pass=10}f{bal=0.01,move=120},
    asc=b{bnd=f{bal=0.01,move=120},org=f{bal=0.01,move=120}}}|
  m{vert=120,low=h{pass=10}f{bal=0.01,move=120},
```

Graphe	Description	$\|V\|$	$\|E\|$	d
grid3d32	Grille 3D régulière	32 768	95 232	5,81
grid3d100	Grille 3D régulière	1 000 000	2 970 000	5,94
cfd2	Mécanique des fluides numériques (*CFD*)	123 440	1 482 229	24,02
crankseg_2	Problème structurel	63 838	7 042 510	220,64
thermal2	Problème thermique	1 228 045	3 676 134	5,99
brack2	Problème 2D/3D	62 631	366 559	11,71
wave	Problème 2D/3D	156 317	1 059 331	13,55
cage12	Électrophorèse d'ADN	130 228	951 154	14,61

TABLE 4.1 – Description des graphes utilisés pour nos expériences. d est le degré moyen du graphe qui vaut $\frac{2\times|E|}{|V|}$.

```
    asc=b{bnd=f{bal=0.01,move=120},org=f{bal=0.01,move=120}}}
}
```

C'est-à-dire que, pour chaque bissection, trois partitionnements multi-niveaux (`m`) sont calculés et la meilleure partition est retenue. Ces partitionnements multi-niveaux pour les bissections sont utilisés en plus du multi-niveaux principal dans lequel cette stratégie de bissections récursives est éventuellement utilisée. Ils utilisent comme stratégie de partitionnement initial une méthode de *greedy graph growing* (`h`) déjà présente dans *Scotch* mais seulement pour les bissections, et comme raffinement un algorithme FM (`f`).

Le tableau 4.1 présente les différents graphes utilisés dans les expériences. Les graphes *grid3d32* et *grid3d100* sont des grilles cubiques de tailles respectives $32 \times 32 \times 32$ et $100 \times 100 \times 100$. Les autres graphes sont issus de la collection de matrices creuses de l'Université de Floride [13].

a) Cas du partitionnement classique

Plusieurs configurations du KGGGP sont évaluées. Les deux critères de sélection basés sur le gain de coupe (`type=g`) et *diff* (`type=d`) sont utilisés pour chaque cas :

— Une seule passe et sans raffinement :

g-noref-1p `g{bal=0.01,type=g,pass=1,ref=}`

d-noref-1p `g{bal=0.01,type=d,pass=1,ref=}`

— Quatre passes avec deux passes de raffinement FM chacune :

g-ref-4p `g{bal=0.01,type=g,pass=4,ref=`
`f{bal=0.01,pass=2,move=200}}`

d-ref-4p `g{bal=0.01,type=d,pass=4,ref=`
`f{bal=0.01,pass=2,move=200}}`

— Quatre passes avec deux passes de raffinement FM chacune et l'utilisation de graines aléatoires (`seeds=r`) :

g-ref-4p-rand `g{bal=0.01,type=g,pass=4,seeds=r,center=n,ref=`
`f{bal=0.01,pass=2,move=200}}`

d-ref-4p-rand `g{bal=0.01,type=d,pass=4,seeds=r,center=n,ref=`
`f{bal=0.01,pass=2,move=200}}`

— Quatre passes avec deux passes de raffinement FM chacune et l'utilisation de graines éloignées (`seeds=o`) pour la première passe puis des « centres » des parties (`center=y`) pour les suivantes :

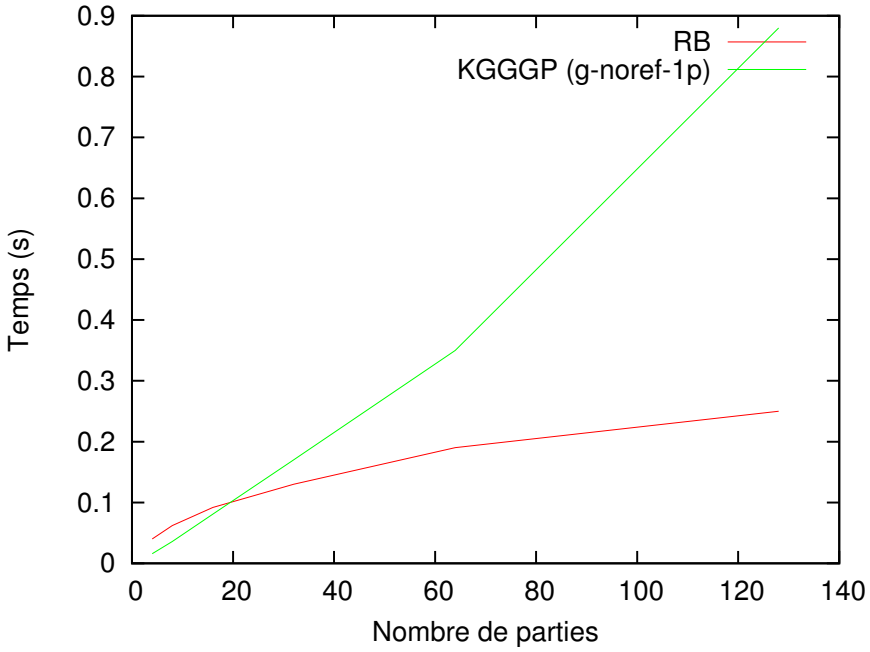


FIGURE 4.6 – Temps du partitionnement de la grille $32 \times 32 \times 32$ sans multi-niveaux, avec KGGGP (une passe et sans raffinement) en comparaison avec RB suivant le nombre de parties.

g-ref-4p-opt `g{bal=0.01,type=g,pass=4,seeds=o,center=y,ref=f{bal=0.01,pass=2,move=200}}`

g-ref-4p-opt `g{bal=0.01,type=d,pass=4,seeds=o,center=y,ref=f{bal=0.01,pass=2,move=200}}`

La figure 4.7 présente la coupe et la figure 4.8 présente le temps obtenus pour le partitionnement des différents graphes avec la méthode KGGGP relativement à la méthode des bissections récursives (RB), toutes sans utiliser de multi-niveaux. Les deux premières stratégies ne faisant qu'une passe et n'utilisant pas de raffinement donnent une coupe beaucoup plus élevée. En effet, le KGGGP a généralement du mal à bien placer les derniers sommets, le raffinement FM permet de corriger les mauvais choix faits par l'algorithme glouton. Le temps d'exécution de ces stratégies est évidement plus rapide que celles faisant plusieurs passes et utilisant un raffinement.

On remarque également que, parmi les deux types de critère de sélection utilisés, celui basé sur le gain de coupe donne toujours de meilleurs résultats que le *diff*. L'utilisation de graines a généralement peu d'influence sur la coupe, bien qu'on remarque une légère amélioration sur quelques cas.

La coupe des meilleures stratégies KGGGP est égale ou supérieure (jusqu'à 20 % plus élevée) à celle des bissections. Le KGGGP est également plus lent : en effet, sa complexité en temps est $O(k|E|)$ alors que celle des bissections récursives est $O(log(k)|E|)$. La figure 4.6 donne les temps d'exécution des bissections récursives et du KGGGP (avec 1 passe et sans raffinement). Les courbes sans multi-niveaux correspondent bien aux complexités attendues.

Mais, en pratique, ces stratégies sont utilisées systématiquement avec une méthode multi-niveaux. Les figures 4.9 et 4.10 montrent les résultats dans le même cas que les figures précédentes mais en utilisant les stratégies (RB et KGGGP) dans une approche multi-niveaux utilisant un

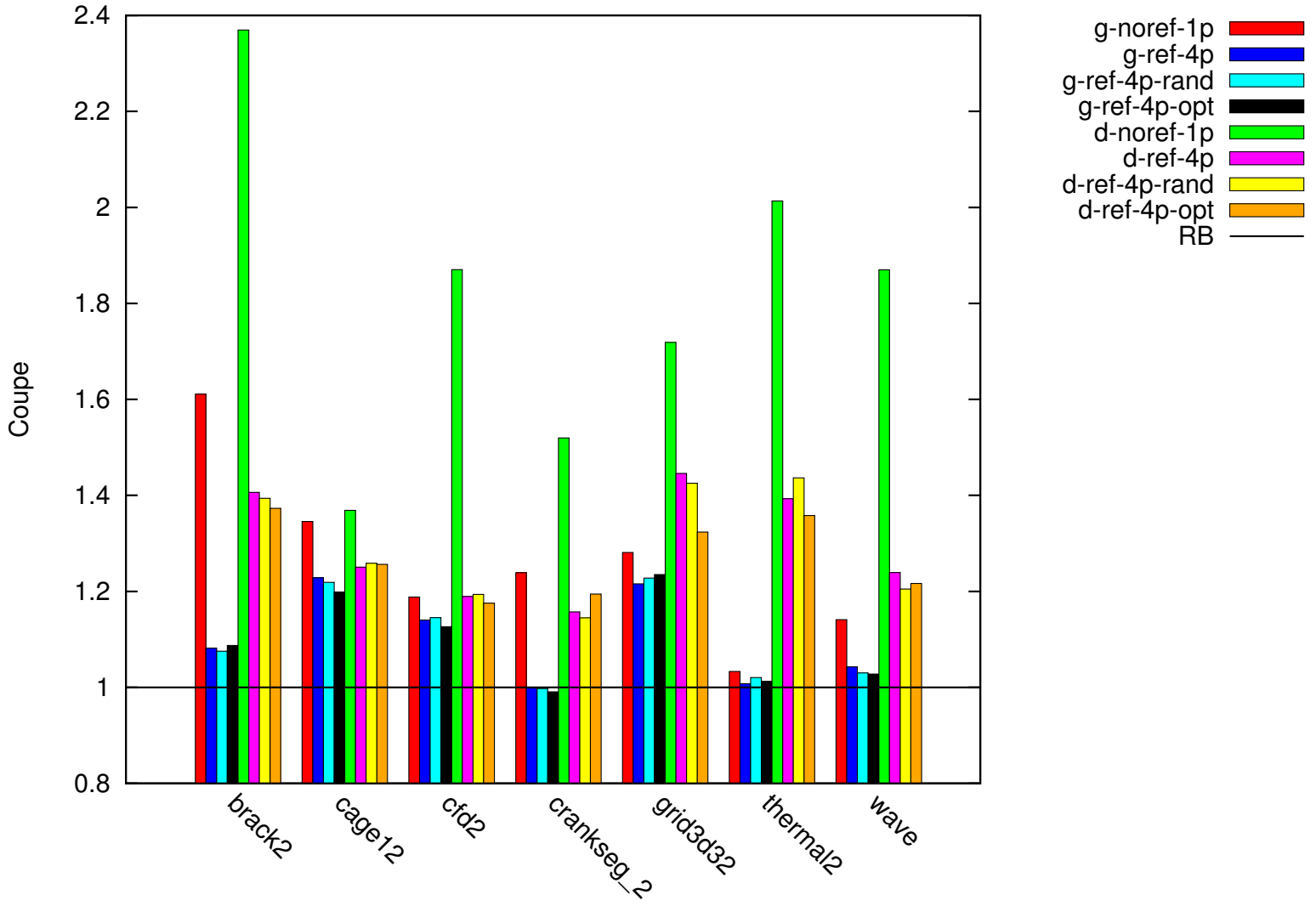


FIGURE 4.7 – Coupe de partitions en 32 parties sans multi-niveaux.

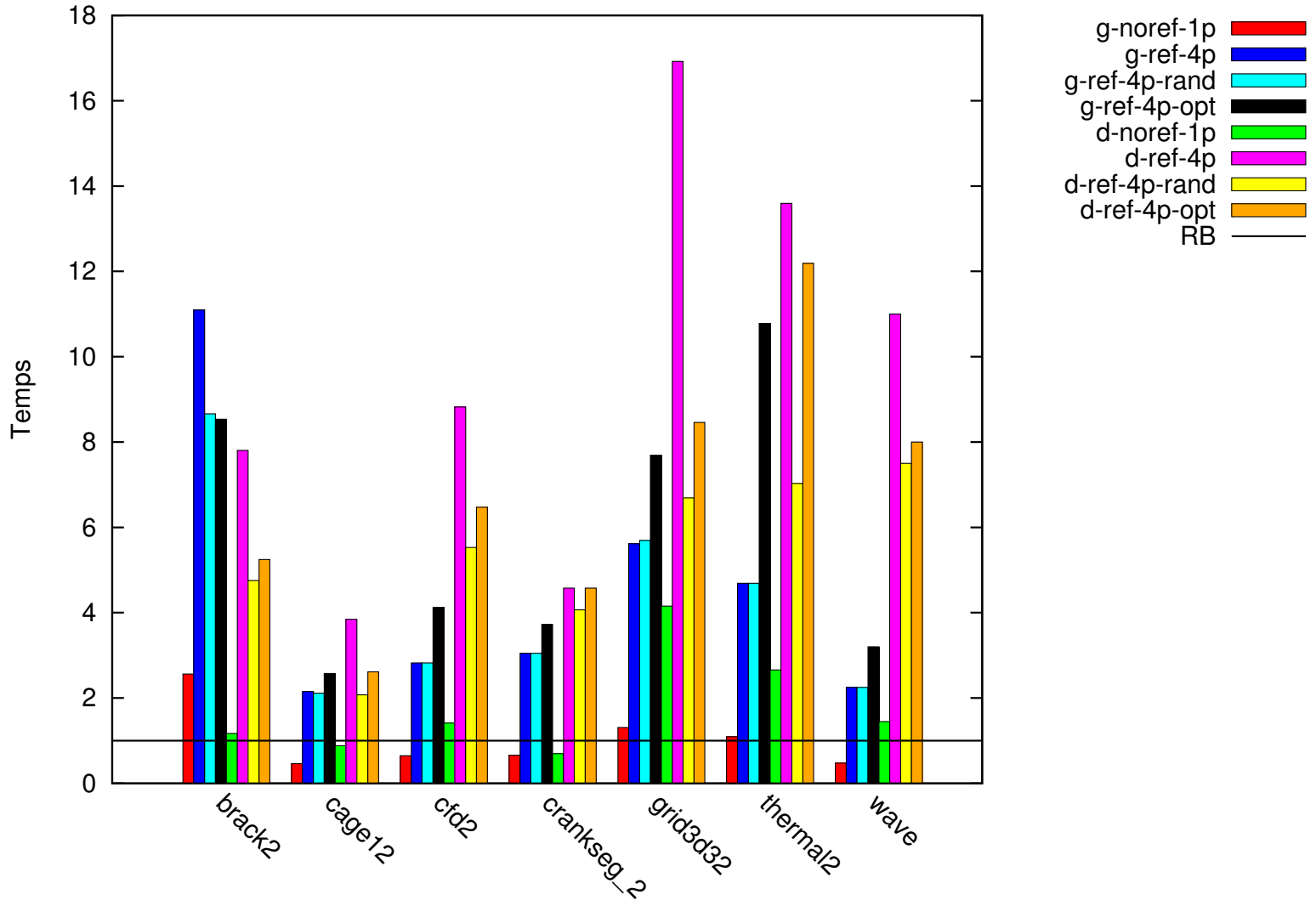


FIGURE 4.8 – Temps de partitionnement en 32 parties sans multi-niveaux.

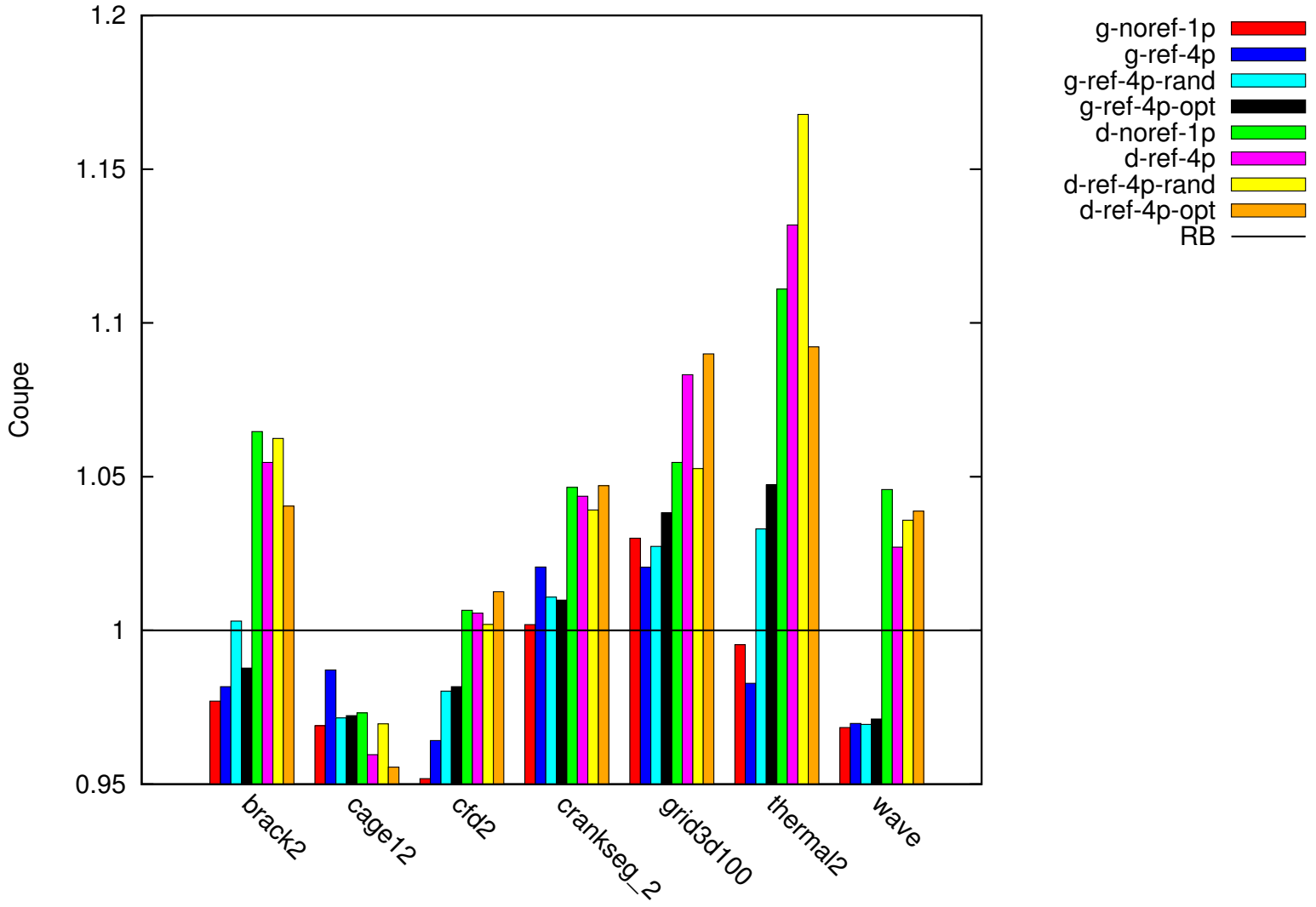


FIGURE 4.9 – Coupe de partitions en 32 parties avec un méthode multi-niveaux.

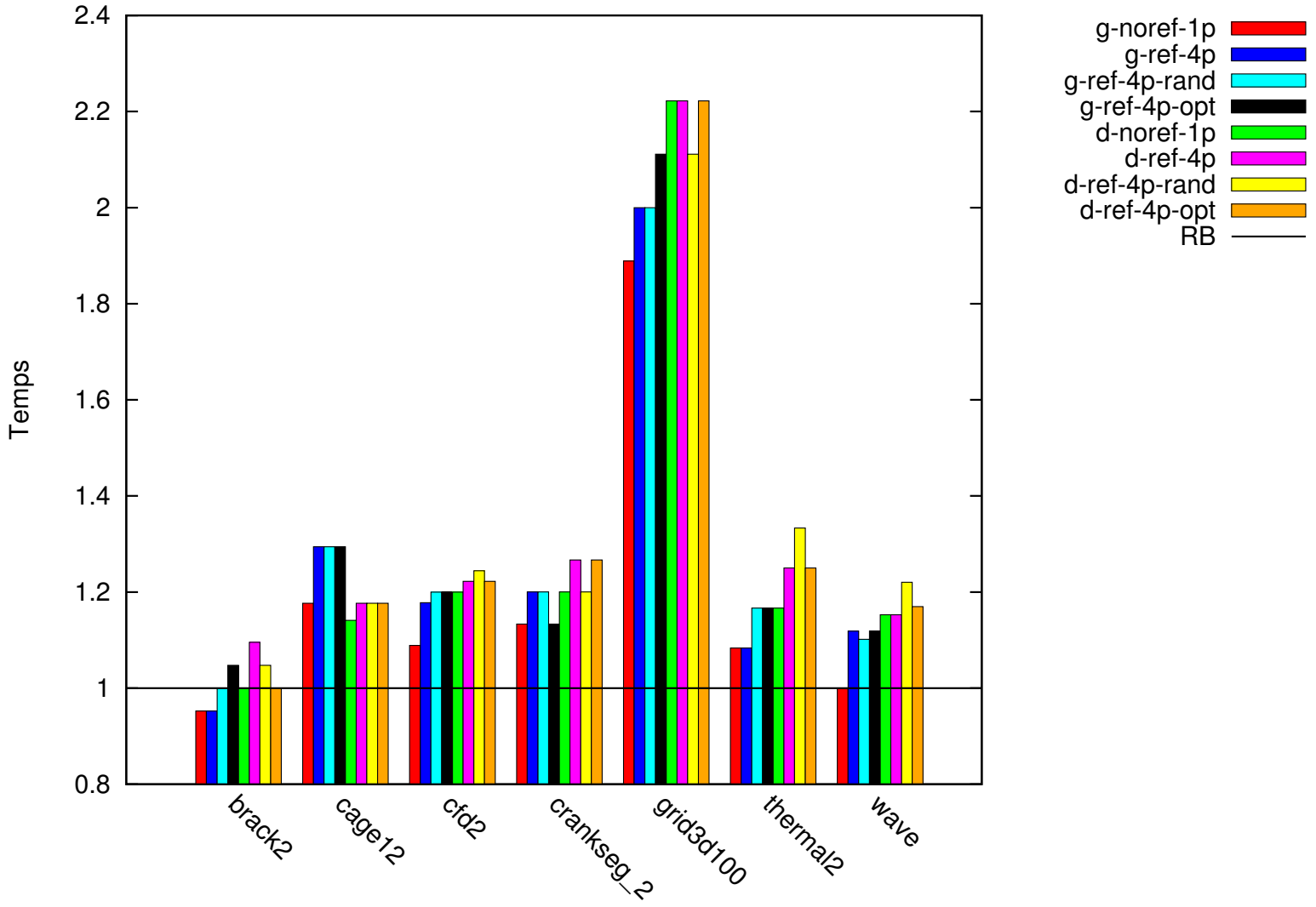


FIGURE 4.10 – Temps de partitionnement en 32 parties avec un méthode multi-niveaux.

raffinement FM k-aire. Les résultats en coupe et temps sont beaucoup plus proches : les phases de contraction et de raffinement prennent le plus de temps alors que la le partitionnement initial est plus rapide comme il s'applique sur un graphe de petite taille ; la coupe finale est largement due au raffinement du multi-niveaux, le partitionnement initial ne lui sert que d'initialisation. L'absence de raffinement n'a plus autant d'importance, le raffinement utilisé lors de la phase d'expansion corrige déjà la plupart des défauts du partitionnement initial. Le type de critère de sélection a toujours un impact significatif sur la coupe finale. Dans le cas de *grid3d100*, les bissections récursives sont capables de trouver une bonne partition directement grâce à la nature régulière du graphe. La phase de raffinement est donc plus plus courte. Le partitionnement RB est donc dans ce cas beaucoup plus rapide que le KGGGP qui profite moins de la régularité du graphe. La coupe obtenue avec les stratégies KGGGP (en utilisant le critère de sélection basé sur le gain) est très proche des bissections récursives : entre $-5\,\%$ et $+5\,\%$.

Notre algorithme de partitionnement KGGGP offre des partitions de qualité similaire au partitionnement utilisant des bissections récursives. Mais cette méthode de partitionnement souffre d'une complexité plus importante par rapport au nombre de parties.

b) Partitionnement avec sommets fixes

Dans la section 4.1.2, nous avons mis en évidence les difficultés des partitionneurs utilisant des bissections récursives à partitionner les graphes construits dans le cadre de notre partitionnement biaisé avec sommets fixes. Nous évaluons donc notre méthode de partitionnement KGGGP appliquée sur les graphes déjà présentés mais modifiés selon la méthode de la section 4.1. Les graphes sont enrichis pour un repartitionnement 16×21 avec un poids 1 pour les arêtes du graphe original et un poids 10 pour les arêtes de migration.

Les figures 4.11 et 4.12 donnent les coupes et temps obtenus pour partitionner ces graphes enrichis. Nous utilisons un partitionnement multi-niveaux avec KGGGP en 4 passes avec raffinement relativement aux bissections récursives. Dans tous les cas, la coupe du KGGGP est inférieure à celle des bissections récursives, parfois de façon très importante (plus de 40% de réduction dans les meilleurs cas). Comme les arêtes de migration ont un poids beaucoup plus important, les valeurs de coupes données concernent principalement les arêtes des migration. Les temps de partitionnement sont généralement plus élevés qu'avec les bissections récursives mais restent du même ordre de grandeur.

Bien que notre algorithme KGGGP ne donnait pas des meilleurs résultats que les bissections récursives pour des partitionnements de graphes classiques, les résultats sont bien meilleurs dans le cas de graphes modifiés avec des sommets fixes et des arêtes de migration. Cela confirme les inconvénients d'utiliser des bissections récursives pour partitionner des tels graphes.

4.3 Repartitionnement $M \times N$ basé sur la diffusion (*DIFF*)

Nous proposons dans cette section une autre approche pour repartitionner un graphe en se basant sur une matrice de migration donnée. Cette approche s'inspire des méthodes de repartitionnement diffusives déjà utilisées dans le cas du repartitionnement classique.

La méthode de repartitionnement biaisé n'impose pas correctement les messages choisis lorsque ceux-ci sont trop nombreux. Cela peut poser problème dans le cas où on veut utiliser une matrice de migration avec un nombre de messages plus élevé, en particulier les matrices de migration obtenues à l'aide de programmes linéaires ne minimisant pas TotalZ. Le repartitionnement basé sur la diffusion règle ce problème en migrant exactement le nombre de sommets souhaité.

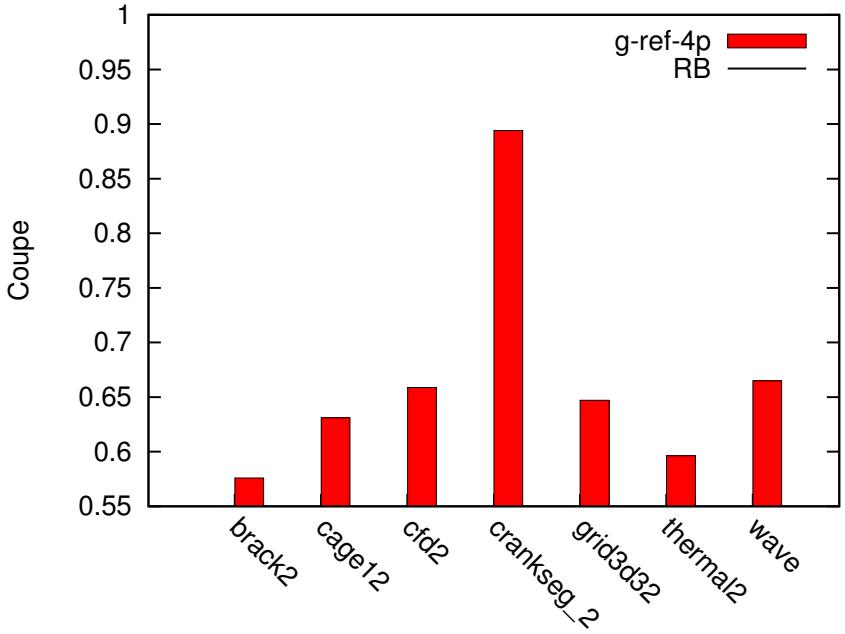


FIGURE 4.11 – Coupe de partitionnement en 21 parties (avec une méthode multi-niveaux) des graphes enrichis pour un repartitionnement 16×21.

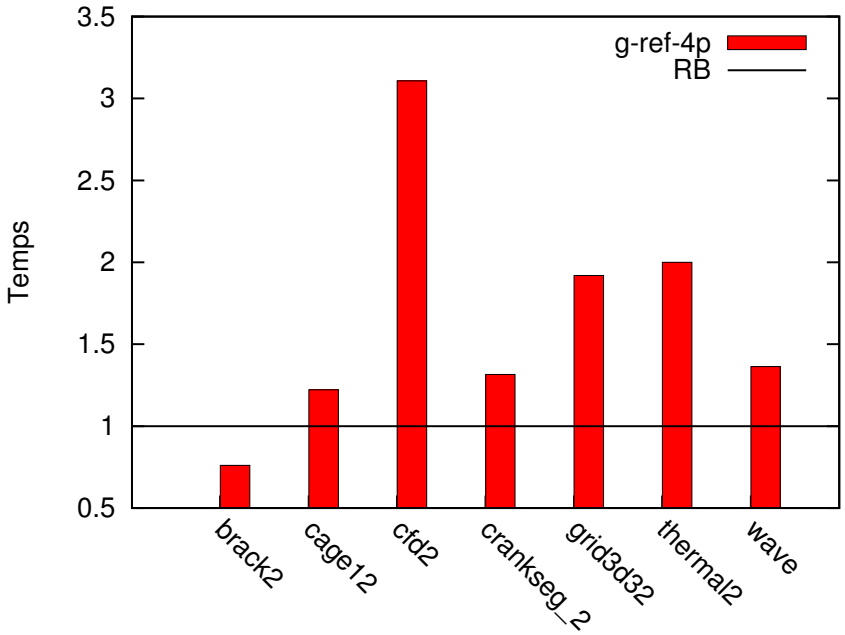


FIGURE 4.12 – Temps de partitionnement en 21 parties (avec une méthode multi-niveaux) des graphes enrichis pour un repartitionnement 16×21.

Le repartitionnement classique basé sur la diffusion [55] modifie l'ancienne partition en migrant des sommets d'une partie à l'autre en se basant sur une matrice de migration C donnée. Pour chaque élément $C_{i,j}$ de la matrice migration, exactement $C_{i,j}$ sommets de la partie i sont déplacés dans la partie j. Les sommets à migrer sont choisis d'une façon similaire à l'heuristique FM mais en arrêtant les déplacements quand le poids souhaité (d'après la matrice de migration) a été déplacé, au lieu de s'arrêter lorsque la coupe optimale est atteinte. Cette heuristique fonctionne bien quand il est nécessaire de déplacer peu de sommets le long d'une frontière séparant deux parties.

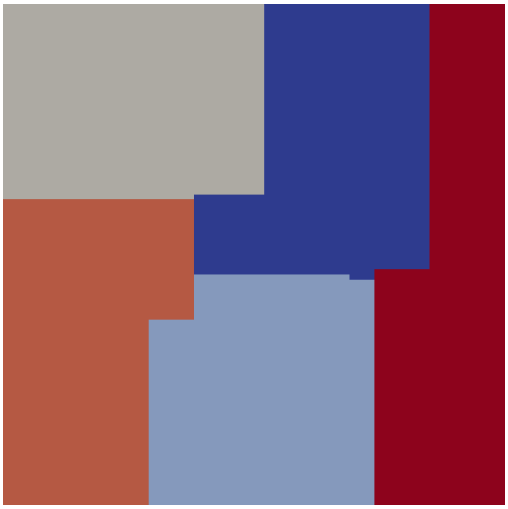
Dans le cas du repartitionnement avec changement du nombre de parties, cette heuristique rencontre quelques problèmes. D'une part, certaines parties connaissent une migration importante : les parties crées ou supprimées au cours du repartitionnement. D'autre part, certaines migrations de sommets doivent être décidées alors qu'il n'existe initialement aucune frontière. C'est le cas des parties initialement vides lorsque le nombre de parties augmente ($N > M$).

Si une partie ne possède initialement aucun sommet, une heuristique choisissant les sommets suivant leur gain de coupe, commencera par sélectionner les sommets de plus petit degré parmi ceux des parties qui doivent donner des sommets à la nouvelle. Ces sommets peuvent être éloignés les uns des autres, ce qui va créer une partie en petits morceaux dispersés. Un tel cas est visible sur la figure 4.13. La partition initiale en 5 parties (figure 4.13a) est équilibrée permettant ainsi de se concentrer sur l'ajout des nouvelles parties. Dans ce cas, il n'y a pas d'échanges de sommets entre les anciennes parties. Les deux nouvelles parties prennent chacune des sommets depuis trois anciennes parties : bleu foncé, gris clair et orange pour la partie vert foncé et bleu foncé, bleu clair et rouge pour la partie vert clair. La figure 4.13b montre un exemple de repartitionnement où l'heuristique a construit les nouvelles parties aux mauvais endroits : les parties ajoutées (en vert) sont découpées en plusieurs bouts, augmentant ainsi la taille de la frontière.

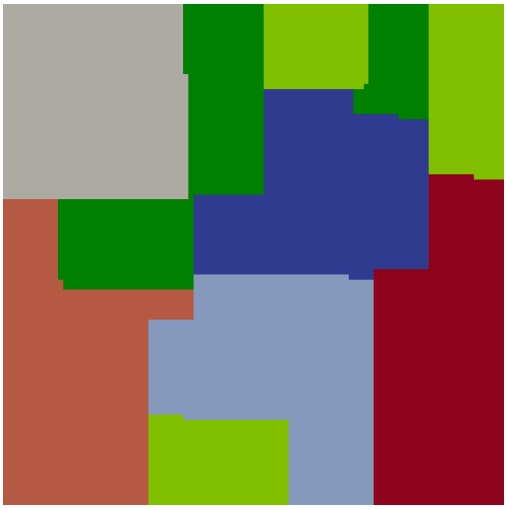
Pour régler ce problème de parties initialement vides, il est nécessaire d'ajouter quelques sommets qui permettront d'initier la création de la nouvelle partie. Le choix de ces graines doit se faire à l'aide d'une méthode plus globale, idéalement au centre de la future partie et ayant des voisins parmi toutes les anciennes parties intervenant dans la migration de la nouvelle partie à créer. Trouver de telles sommets peut demander des algorithmes complexes. Une méthode naïve est de rechercher les sommets ayant des voisins parmi les autres anciennes parties voulues et d'en prendre un au hasard. De telles sommets n'existent pas toujours comme il n'est pas toujours possibles de créer des parties connexes pour certaines matrices de migration choisies. La figure 4.13d montre un repartitionnement obtenu à l'aide de graines ajoutées à la frontière des trois anciennes parties donnant des sommets à une même nouvelle partie comme indiqué sur la figure 4.13c.

Que la migration soit faite par paire de parties émettrice–réceptrice ou globalement, l'ordre dans lequel sont effectués les déplacements de sommets a une grande influence sur le résultat final. En effet, les sommets peuvent généralement être déplacés vers plusieurs autres parties et il faut donc choisir dans quel ordre considérer ces parties. Schloegel *et al.* [58] étudie se problème dans le cas de repartitionnement classique. Les solutions les plus simples sont un ordre aléatoire ou de considérer en priorité les parties émettrices qui ont déjà reçu tous les sommets dont elles ont besoin. Si ce dernier ordonnancement est choisi, cela peut poser problème lors de la création de nouvelles parties. Ces nouvelles parties seraient alors traiter en dernier car elles ne sont que réceptrices. Mais les déplacements précédents changent les frontières et éventuellement la forme du graphe quotient qui a été utilisé pour choisir l'emplacement de ces nouvelles parties. Les anciennes parties donnant à une même nouvelle partie peuvent ne plus être proches quand arrive le moment de construire cette nouvelle partie.

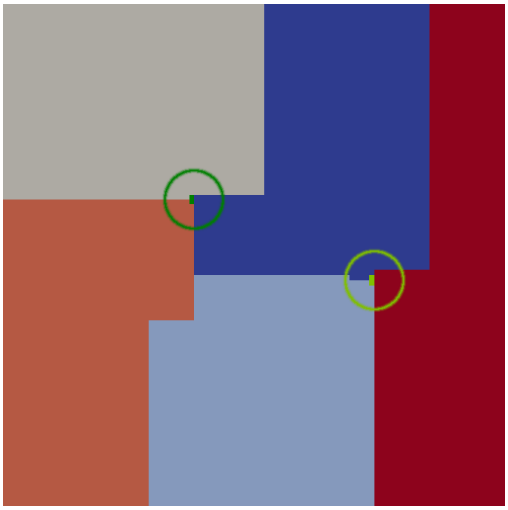
Notre réalisation préliminaire du repartitionnement diffusif utilise le graphe entier sans approche multi-niveaux et sans utiliser de raffinement, ce qui limite la qualité de la coupe finalement



(a) Ancienne partition en 5 parties.



(b) Nouvelle partition en 7 parties sans utiliser de graines.



(c) Ajout de graines sur l'ancienne partition.

(d) Nouvelle partition en 7 parties en utilisant des graines.

FIGURE 4.13 – Exemple d'ajout de deux nouvelles parties prenant des sommets depuis trois anciennes chacune à l'aide de la diffusion.

obtenue. Les migrations sont effectuées par paires de parties en commençant par les parties initialement vides (les processeurs ajoutés) puis les paires d'anciennes parties en commençant par celles où les parties émettrices ont déjà reçu tous leurs sommets.

4.4 Partitionnement biaisé à l'aide d'hyper-arêtes de repartitionnement

En utilisant un modèle hypergraphe (plutôt que graphe) et la coupe connectivité − 1 (notée $\lambda - 1$), il est possible d'optimiser le nombre de messages lors du repartitionnement avec une nouvelle approche. Cette méthode optimise directement la migration lors du repartitionnement et ne nécessite pas de matrice de migration en entrée, comme le suggère la figure 4.1.

En ajoutant, pour chaque ancienne partie, une hyper-arête regroupant tous les sommets de cette partie, le critère de coupe sera modifié pour prendre en compte le nombre de messages. En effet, ces hyper-arêtes seront coupées par λ parties qui prendront des sommets de cette ancienne partie. Chaque nouvelle partie qui coupe une de ces hyper-arêtes induit un message de migration avec l'ancienne partie associée à cette hyper-arête. On note λ_i, le nombre de parties qui coupent l'hyper-arête correspondant à l'ancienne partie i et w_{msg} le poids de ces hyper-arêtes. λ_i est égale au nombre de non-zéros sur la ligne i de la matrice de migration C. Comme le poids de chaque hyper-arête est compté $\lambda_i - 1$ fois, la coupe totale de toutes ces hyper-arêtes ajoutées est :

$$\begin{aligned}\sum_{i\in[\![1,M]\!]} (\lambda_i - 1) \times w_{msg} &= \left(\sum_{i\in[\![1,M]\!]} \lambda_i - M\right) \times w_{msg} \\ &= (\mathrm{nnz}(C) - M) \times w_{msg}\end{aligned}$$

où $\mathrm{nnz}(C)$ est le nombre de non-zéros dans la matrice de migration C.

En notant G l'hypergraphe initial et G' l'hypergraphe enrichi des hyper-arêtes de poids w_{msg}, on obtient :

$$\begin{aligned}cut(G') &= cut(G) + w_{msg} \times (\mathrm{nnz}(C) - M) \\ &\approx cut(G) + w_{msg} \times \textsc{TotalZ}\end{aligned}$$

Remarquons que $\mathrm{nnz}(C) - M$ est à peu près le nombre total de messages. TotalZ est le nombre d'éléments non nuls hors diagonale de la matrice de migration. Dans le cas où la diagonale ne possède aucun élément nul, on a $\textsc{TotalZ} = \mathrm{nnz}(C) - \min(M, N)$.

Ainsi en minimisant la coupe du graphe enrichi, le partitionneur minimise donc un compromis entre la coupe du graphe initial et le nombre de messages. Si le poids w_{msg} de ces arêtes est suffisamment grand, cela revient à minimiser le nombre de messages dans un premier temps puis à minimiser la coupe dans un second temps.

Par ailleurs, cette méthode optimisant TotalZ peut être combinée avec une stratégie de partitionnement biaisé optimisant le volume de migration TotalV. Pour cela, on reprend la méthode de repartitionnement à sommets fixes [2, 9] : on construit l'hypergraphe enrichi G'' en ajoutant un sommet fixe pour chaque ancienne partie, et des hyper-arêtes de taille 2 reliant les sommets fixes aux autres sommets appartenant à l'ancienne partie correspondante avec un poids w_{mig}. La coupe du nouveau graphe G'' devient alors :

$$\begin{aligned}cut(G'') &= cut(G') + w_{mig} \times \textsc{TotalV} \\ &\approx cut(G) + w_{msg} \times \textsc{TotalZ} + w_{mig} \times \textsc{TotalV}\end{aligned}$$

Bien que cette méthode soit capable de bien modéliser les objectifs du repartitionnement, l'application à l'aide d'un partitionneur d'hypergraphe ne donne pas de bons résultats.

Partitionner un tel graphe enrichi à la fois avec de grandes hyper-arêtes et les sommets fixes connectés à de nombreux sommets est effectivement très complexe. Comme il a déjà été expliqué dans la section 4.1.2, les stratégies de bissections récursives couramment utilisées par les partitionneurs peuvent avoir du mal à trouver de bonnes partitions pour certains types de graphes ou d'hypergraphes. Dans le cas des hypergraphes, en plus des problèmes avec les sommets fixes, les bissections optimisent moins bien la coupe connectivité -1 des grandes hyper-arêtes. Il serait néanmoins possible d'améliorer le partitionnement à l'aide d'un algorithme k-aire direct adapté aux hypergraphes [3] [5]. Mais même avec un partitionnement k-aire, la phase de contraction dans un partitionnement multi-niveaux peut aussi être perturbée par ces grandes hyper-arêtes : il peut être décidé de fusionner des sommets très éloignés s'ils appartiennent à la même hyper-arête. Il donc important d'utiliser un algorithme de contraction adapté qui donnerait moins d'importance, voire ignorerait, ces grandes hyper-arêtes malgré leur poids important.

4.5 Conclusion

Dans ce chapitre, nous avons présenté trois méthodes de repartitionnement $M \times N$. Notre partitionnement biaisé (*BIASED*) permet d'obtenir un bon partitionnement (équilibre et coupe faible) tout en imposant un schéma de communication donné, mais cette méthode n'impose pas toujours la matrice de migration souhaitée. Notre partitionnement diffusif (*DIFF*) permet d'appliquer strictement une matrice de communication mais la coupe est moins bien optimisée, en particulier pour les nouvelles parties, qui induisent d'importantes migrations. Nous avons aussi présenté une méthode de partitionnement biaisé basé sur des hyper-arêtes permettant de minimiser directement TotalZ et TotalV sans utiliser de matrice de migration en entrée. Mais cette dernière méthode ne donne pas de bonne partition avec les outils actuels et ne sera pas évaluée par la suite.

5. Le partitionneur *kPaToH* présenté par Aykanat et al. n'est malheureusement pas disponible.

Chapitre 5

Résultats

Sommaire

5.1 Méthodologie expérimentale 91
5.2 Influence du coût de migration et du facteur de repartitionnement 94
5.3 Influence du nouveau nombre de processeurs 97
5.4 Comparaison sur des graphes complexes 101
5.5 Étude de la complexité en temps 110

Dans ce chapitre, nous présentons plusieurs expériences permettant d'évaluer nos méthodes de repartitionnement en les comparant aux partitionneurs actuels.

5.1 Méthodologie expérimentale

Pour rappel, les objectifs du repartitionnement $M \times N$ sont :
— équilibrer la charge des différentes parties ;
— minimiser la coupe du graphe ;
— calculer rapidement la nouvelle partition ;
— optimiser la migration.
Le premier objectif, l'équilibrage, est toujours atteint par les partitionneurs : dans les évaluations suivantes la tolérance au déséquilibre est de 1%, c'est-à-dire que la plus grosse partie ne dépassera pas de plus de 1% la taille moyenne des parties. Pour les trois autres objectifs, des métriques sont utilisés : la coupe ; TotalV, MaxV, TotalZ, MaxZ pour la migration ; et le temps d'exécution du partitionneur.

Dans ce chapitre, nous évaluons les méthodes de repartitionnement $M \times N$ *Biased* et *Diff* que nous avons proposés, dans deux variantes chacunes.

La première méthode combine la construction de la matrice de migration à l'aide de l'algorithme glouton (plus précisément l'algorithme *Greedy2* optimisant le nombre de messages) et le partitionnement biaisé utilisant les sommets fixes. L'algorithme glouton peut optimiser ou non la diagonale de la matrice de migration. Dans le chapitre 3, il a été montré que ces deux variantes permettent d'obtenir un TotalZ bas. L'optimisation de la diagonale permet de minimiser TotalV mais en augmentant MaxZ. Ces deux stratégies seront appelées *BiasedGreedyD* pour la variante optimisant la diagonale, *BiasedGreedy* sinon. Le partitionnement biaisé est réalisé à l'aide de *Scotch* et de la méthode de partitionnement k-aire direct présentées dans la section 4.2

et intégrée à *Scotch*. Notons que le partitionnement est biaisé en modifiant le graphe fourni au partitionneur et non en utilisant les fonctionnalités de repartitionnement disponibles dans *Scotch*.

La seconde méthode construit la matrice de migration à l'aide du programme linéaire minimisant soit TotalV, soit MaxV. Cette construction de la matrice de migration ne donne pas un TotalZ minimal (mais garantit un MaxZ faible), il est donc préférable d'utiliser la méthode diffusive pour repartitionner le graphe d'après ces matrices de migration. On nommera *DiffTV* la stratégie minimisant TotalV et *DiffMV* la stratégie minimisant MaxV.

Ces méthodes seront comparées à des méthodes de repartitionnement classiques :

— une méthode de *Scratch-Remap* ;
— la méthode de repartitionnement de *Zoltan* ;
— celle de *ParMetis* ;
— et celle de *Scotch*.

La méthode de *Scratch-Remap* est mise en œuvre à l'aide du partitionnement *from scratch* (sans repartitionnement) de *Scotch* et de l'algorithme 1 de *remapping*.

Il est important de noter que les partitionneurs *Zoltan*, *ParMetis* et *Scotch* n'ont pas été développés pour le repartitionnement $M \times N$ mais il est tout de même possible des les utiliser dans ce cas. Cela revient à un repartitionnement $N \times N$ où certaines parties sont initialement vides lorsque le nombre de processeurs augmente ($M < N$) ou certains sommets sont sans partie lorsque le nombre de processeurs diminue ($M > N$).

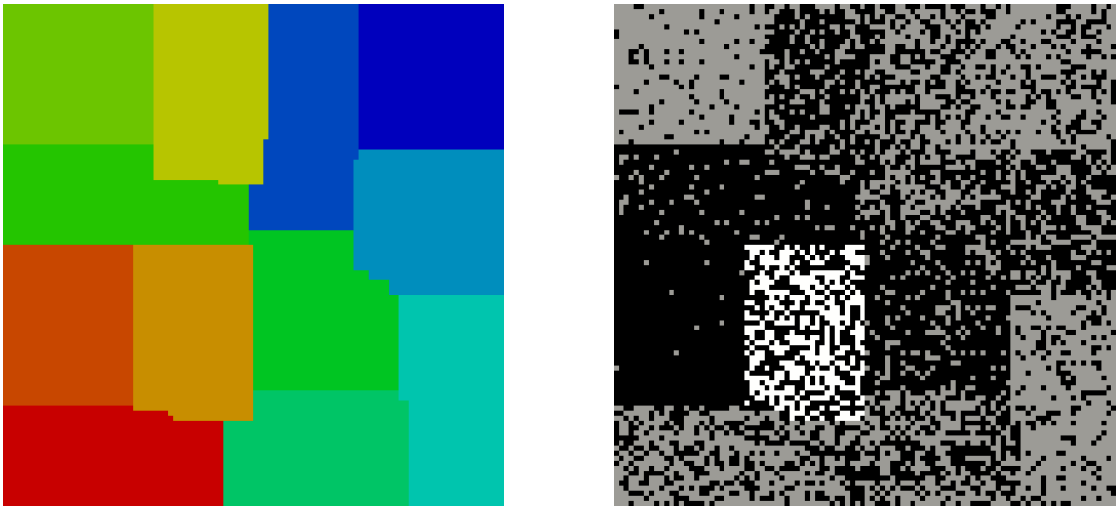
Zoltan est un partitionneur d'hypergraphe [1] possédant une méthode de repartitionnement biaisée à l'aide de sommets fixes. Le poids des arêtes de migration choisi est 1 comme celui des arêtes du graphe à partitionner. Cela permet d'obtenir un bon compromis entre coupe et migration. Bien que *Zoltan* soit un partitionneur parallèle, il est utilisé avec un seul processus dans nos expériences.

ParMetis utilise une méthode de repartitionnement hybride choisissant le meilleur résultat entre un repartitionnement diffusif et un *Scratch-Remap* pour le partitionnement initial d'un repartitionnement multi-niveaux [58]. Le raffinement utilisé est biaisé pour prendre en compte un compromis entre coupe et volume de migration. *ParMetis* prend en compte un rapport de coût entre la coupe et le volume de migration. Comme avec *Zoltan*, ce paramètre est fixé à 1. *ParMetis* est un partitionneur parallèle qui ne peut pas être utilisé avec moins de deux processus. Dans les expériences suivantes, *ParMetis* profite donc de deux processus, alors que les autres partitionneurs sont utilisés en séquentiel.

Scotch est capable de repartitionner à l'aide d'une méthode de partitionnement biaisé. La stratégie de partitionnement de *Scotch* peut être choisie en détail à l'aide d'une chaîne de stratégie. Mais dans le cas du repartitionnement, nous laissons *Scotch* construire la chaîne de stratégie pour obtenir une partition équilibrée (`SCOTCH_STRATBALANCE`), de qualité (`SCOTCH_STRATQUALITY`) et adaptée au repartitionnement (`SCOTCH_STRATREMAP`). *Scotch* permet aussi de choisir l'importance relative de la coupe et de la migration. La valeur choisie pour ce paramètre est encore 1.

Pour évaluer ces méthodes de repartitionnement, nous avons besoin de créer des partitions initiales déséquilibrées. Les anciennes partitions utilisées dans les repartitionnements suivants sont créées à partir de partitions équilibrées dont le poids des sommets, valant initialement 1, est modifié pour simuler un raffinement du maillage. En parcourant les parties dans un ordre aléatoire, la charge de la première partie n'est pas modifiée, la charge de la seconde est augmentée d'une certaine quantité, celle de la troisième de deux fois cette quantité, etc., la charge de la dernière partie augmente de $M - 1$ fois cette quantité. L'augmentation de la charge est choisie

1. Il est utilisé ici pour partitionner des graphes. Un graphe n'est qu'un hypergraphe particulier ne possédant que des hyper-arêtes de taille 2. La coupe connectivité $- 1$ équivalent à la coupe du graphe pour les hyper-arêtes de taille 2.



(a) Partition en 12 parties. (b) Modification du poids des sommets.

FIGURE 5.1 – Exemple de déséquilibrage d'une partie en 12.

de façon à ce que l'augmentation totale de la charge soient proportionnelle à l'augmentation du nombre de ressources. Par exemple, en partant d'une ancienne partition équilibrée en $M = 8$ parties, si la charge totale augmente de 50 %, l'ancienne partition (possédant maintenant un déséquilibre de 33 %) est repartitionnée en $N = 12$ parties.

Pour augmenter la charge d'une partie, des sommets sont choisies aléatoirement pour avoir un poids plus élevé. Pour chaque partie, ce nouveau poids est choisi pour être minimal. Par exemple, si la charge augmente mais sans doubler la charge de la partie, le nouveau poids sera 2 ; si la charge augmente plus que le double, il n'est plus possible d'atteindre la charge désirée, le nouveau poids est donc 3 voire plus. Dans une partie donnée, les poids des sommets sont alors 1 ou le nouveau poids choisi pour cette partie.

Un exemple d'un tel déséquilibrage est présenté sur la figure 5.1. Une grille d'éléments carré est initialement partitionnée en 12 parties équilibrées (figure 5.1a). Les poids des sommets sont ensuite modifiés comme indiqué sur la figure 5.1b : en noir, les sommets qui ont gardé un poids de 1 ; en gris, les sommets qui ont un poids 2 ; et en blanc un poids 3. La densité des poids dans une même partie est à peu près uniforme. Mais la charge de chaque partie augmente d'une quantité différente : de celle ne contenant que des sommets de poids 1 à celle contenant les sommets de poids 3.

Les expériences suivantes sont répétées plusieurs fois : 10 fois pour toutes les expériences sauf celles sur la complexité en temps (le nombre d'itérations varie suivant le cas). À chaque repartitionnement le déséquilibre est différent : l'ordre des parties utilisé lors du déséquilibrage et les sommets dont les poids sont modifiés sont tirés aléatoirement à chaque fois mais la variation de la charge totale est toujours la même et identique à la variation du nombre de processeurs.

5.2 Influence du coût de migration et du facteur de repartitionnement

Comme il a été dit dans le chapitre 4, il est important de bien choisir le poids des arêtes de migration ajoutées dans le graphe original. Comme les partitionneurs utilisent généralement des poids entiers pour les arêtes, il n'est pas possible d'avoir un poids inférieur à 1. Pour pouvoir appliqué un poids moins important aux arêtes de migration, nous réutilisons la méthode utilisée par *Zoltan* : les poids de toutes les arêtes du graphe original sont multipliés par le *facteur de repartitionnement* (*repart multiplier*). Ainsi le poids des arêtes de migration devient moins important relativement aux autres arêtes. Dans cette section, nous étudions le comportement de notre méthode de repartitionnement $M \times N$ biaisée en fonction du rapport du poids des arêtes « normales » par rapport à celui des arêtes de migration $\frac{repartmult}{migcost}$.

Les expériences présentées dans cette section correspondent au repartitionnement d'une grille 3D $100 \times 100 \times 100$ (1 000 000 sommets et 2 970 000 arêtes, ce qui donne un degré moyen de $5,94$). Nous étudions le cas 8×10 où la charge augment de $+25\,\%$ (le déséquilibre est $17\,\%$). Quelque graphiques pour le cas 8×12 où la charge augment de $+50\,\%$ (le déséquilibre est $33\,\%$) sont présentées dans l'annexe C.1. Les valeurs données sont les résultats moyens et les barres d'erreurs donnent les résultats minimums et maximums. Le coût de migration et le facteur de repartitionnement ne concerne que le partitionnement biaisé mais les valeurs moyennes de la stratégie *Scratch-Remap* sont données comme repères.

Comme le repartitionnement biaisé impose le schéma de communication, un moyen simple de vérifier son efficacité et de regarder les valeurs de TotalZ effectivement obtenues après le repartitionnement. Les figures C.1 et 5.2 présentent les valeurs de TotalZ dans deux cas de repartitionnement, respectivement 8×12 et 8×10. Comme il a déjà été décrit au chapitre 3, l'optimisation de la diagonale (*GreedyD*) crée de trop grande hyper-arêtes (une nouvelle partie reçoit des sommets d'un trop grand nombre d'anciennes parties). On vérifie ici que cela rend le partitionnement plus difficile : un coût de migration de 4 ne suffit plus à imposer le schéma de communication pour cette stratégie. On voit sur la figure 5.2 que TotalZ est minimisé à partir d'un coût de migration 4 fois supérieur pour la méthode *BiasedGreedy*. Ce seuil est un peu plus élevé pour *BiasedGreedyD*. Un rapport $\frac{repartmult}{migcost}$ de 1 offre un nombre de message bas, sans être optimale alors qu'avec un facteur de repartitionnement plus important le nombre de messages augmente très rapidement. Dans le cas du repartitionnement 8×12 (figure C.1), la stratégie *GreedyD* crée de moins grande hyper-arêtes et la différence entre *biasedGreedy* et *BiasedGreedyD* s'estompe.

Le coût de migration et le facteur de repartitionnement permettent de paramétrer le compromis entre migration et coupe. En effet, la coupe, donnée sur la figure 5.3, est légèrement plus basse avec un coût de migration faible et un facteur de repartitionnement important. Cette variation est plus marquée avec la méthode *BiasedGreedyD* car le partitionnement est plus difficile. Cette différence de coupe reste assez faible et devient même négligeable dans des cas plus favorable comme le repartitionnement 8×12 (figure C.2).

Sur la figure 5.4, le volume total de migration TotalV évolue dans le sens contraire de la coupe : un coût de migration important minimise le volume de migration, alors qu'un facteur de repartitionnement plus important donne un volume équivalent au *Scratch-Remap*. La variation plus importante de la coupe avec la méthode *BiasedGreedyD* s'explique par son optimisation plus agressive de la migration. Contrairement à la coupe, la variation de TotalV est bien plus importante.

Le coût de migration et le facteur de repartitionnement influencent également le temps de partitionnement. La figure 5.5 montre que le temps de partitionnement augmente avec un coût

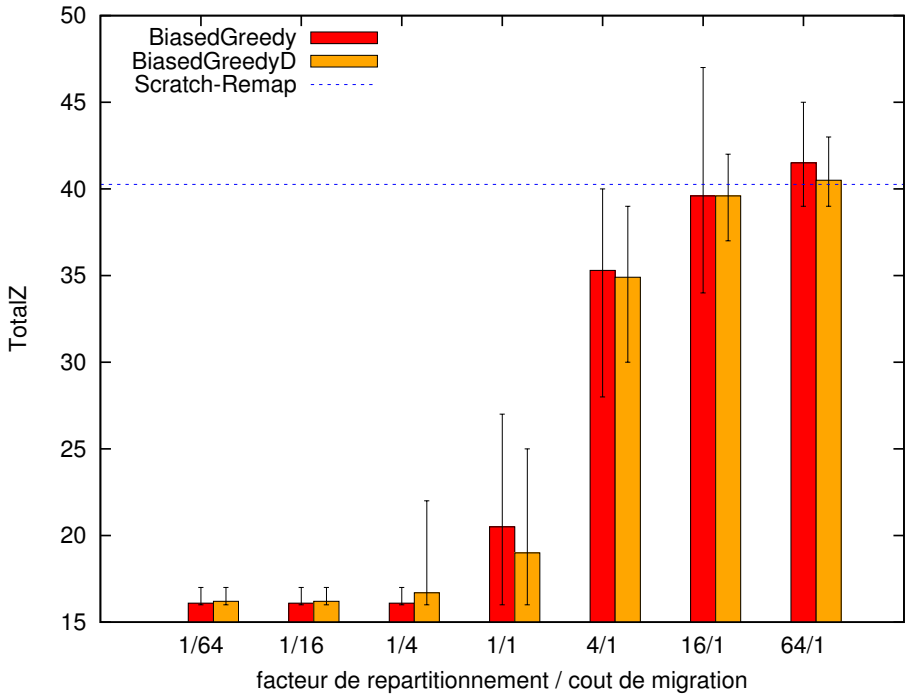


FIGURE 5.2 – TOTALZ 8×10 suivant *repartmult/migcost*.

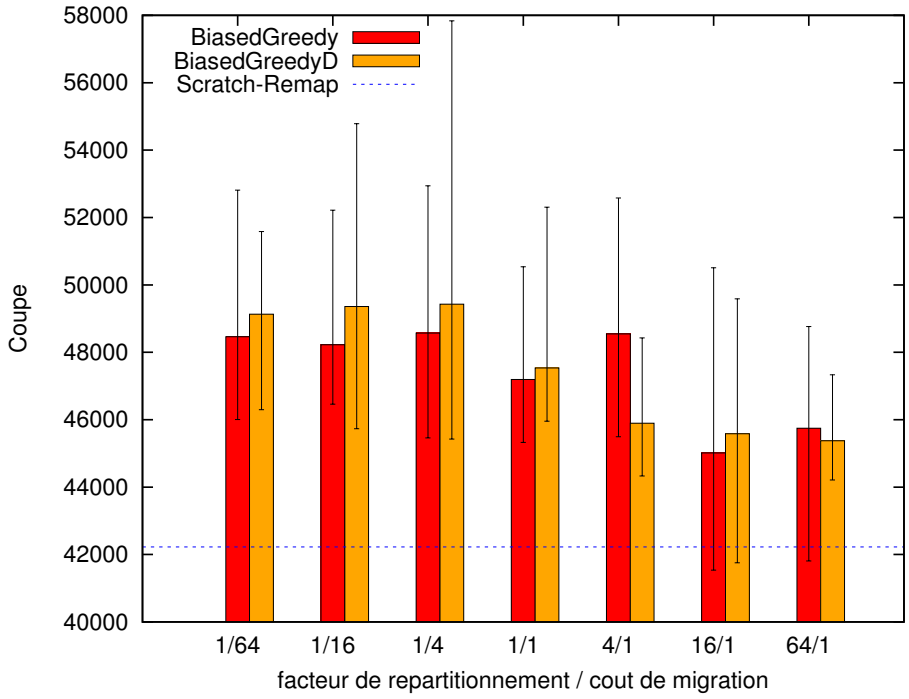


FIGURE 5.3 – Coupe 8×10 suivant *repartmult/migcost*.

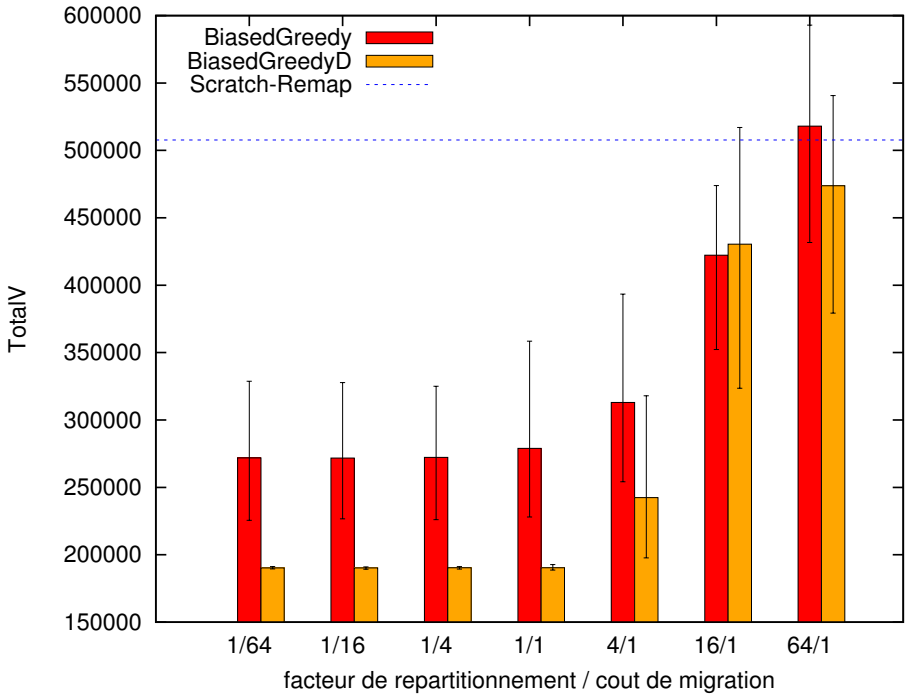


FIGURE 5.4 – TOTALV 8×10 suivant *repartmult/migcost*.

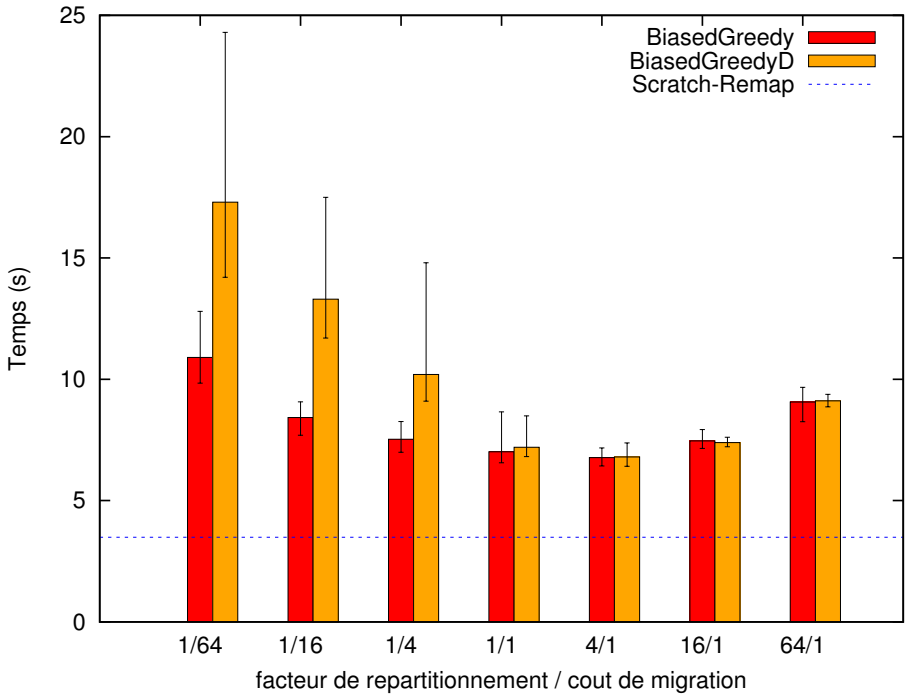


FIGURE 5.5 – Temps 8×10 suivant *repartmult/migcost*.

de migration ou un facteur de repartitionnement plus important. Les poids plus important affectés aux arêtes ralentissent les algorithmes de partitionnement, en particulier, les algorithmes de raffinement de type FM, où il faut sélectionner un déplacement dans un intervalle de gain plus important.

Il est donc nécessaire de choisir un coût de migration suffisamment élevé pour obtenir une bonne migration mais sans le prendre plus haut que nécessaire pour éviter un ralentissement du partitionnement. Utiliser un facteur de repartitionnement important n'a pas d'intérêt : TotalZ et TotalV sont alors équivalents, au *Scratch-Remap* mais la coupe et le temps de repartitionnement sont plus importants. Dans les expériences suivantes, le coût de migration utilisé sera fixé à 10 pour un facteur de repartitionnement de 1. Cela permettra d'obtenir une bonne migration au prix d'une légère augmentation du temps de partitionnement et de la coupe.

5.3 Influence du nouveau nombre de processeurs

La prochaine série d'expérience étudie l'influence du nouveau nombre de processeurs. Une grille 3D $100\times100\times100$ est repartitionnée depuis 8 anciennes parties vers un nombre variable N de nouvelles parties. Dans chaque cas, la charge est augmentée proportionnellement à l'augmentation de ressources, soit $\frac{N}{8}$.

La figure 5.6 présente la coupe obtenue pour les différents repartitionneurs. La méthode *Scratch-Remap* offre la meilleure coupe comme on pouvait s'y attendre : c'est son seul objectif en dehors de l'équilibrage. Nos méthodes de repartitionnement donnent une coupe un peu plus élevée que le *Scratch-Remap* mais globalement meilleure que *ParMetis* et *Zoltan*. En particulier, les méthodes *BiasedGreedy* et *BiasedGreedyD* donnent une bonne coupe comme celle du repartitionneur *Scotch*.

Il est également intéressant de remarquer que, bien que généralement très proches, les méthodes *BisaedGreedy* et *BiasedGreedyD* donnent une coupe différente pour les petites valeurs de N (8×9 et 8×10) : en effet, la méthode *BiasedGreedyD* donne de trop grandes hyper-arêtes pour les petites variations du nombre de processeur ce qui rend le repartitionnement plus difficile et dégrade en conséquence la coupe pour cette méthode.

Les méthodes diffusives (*DiffTV* et *DiffMV*) donnent une coupe plus élevée que nos méthodes biaisés mais meilleures que *Zoltan* et *ParMetis*. La diffusion est réalisée en une seule passe sans profiter d'un partitionnement multi-niveaux complet. La partition obtenue est donc de moins bonne qualité.

Le volume total de migration TotalV est donné sur la figure 5.7. La méthode *Scratch-Remap* donne le volume de migration le plus élevé mais celui-ci varie peu et il est asymptotiquement rejoint par les autres méthodes de repartitionnement quand N devient très grand comparé à $M = 8$. La stratégie *BiasedGreedyD* donne le meilleur volume de migration quelle que soit la valeur de N. Sa variante sans optimisation de la diagonale de la matrice de migration *BiasedGreedy* donne un TotalV plus élevé, surtout quand N et M sont proches et donne même la plus importante migration (hors *Scratch-Remap*) dans le cas 8×9.

La méthode diffusive *DiffTV* optimisant TotalV donne également un volume de migration très bas. Il n'est pas toujours aussi bas que *BiasedGreedyD* à cause du choix du graphe quotient augmenté limitant les possibilités de migration. La variante *DiffMV* optimisant MaxV donne un volume de migration total un peu plus élevé, mais celui reste assez bas.

Les autres partitionneurs, en particulier *ParMetis* et *Scotch* donnent aussi un volume de migration très bas, alors que *Zoltan* se comporte un peu moins bien.

La figure 5.8 donne le volume de migration maximal par partie MaxV. Plusieurs courbes

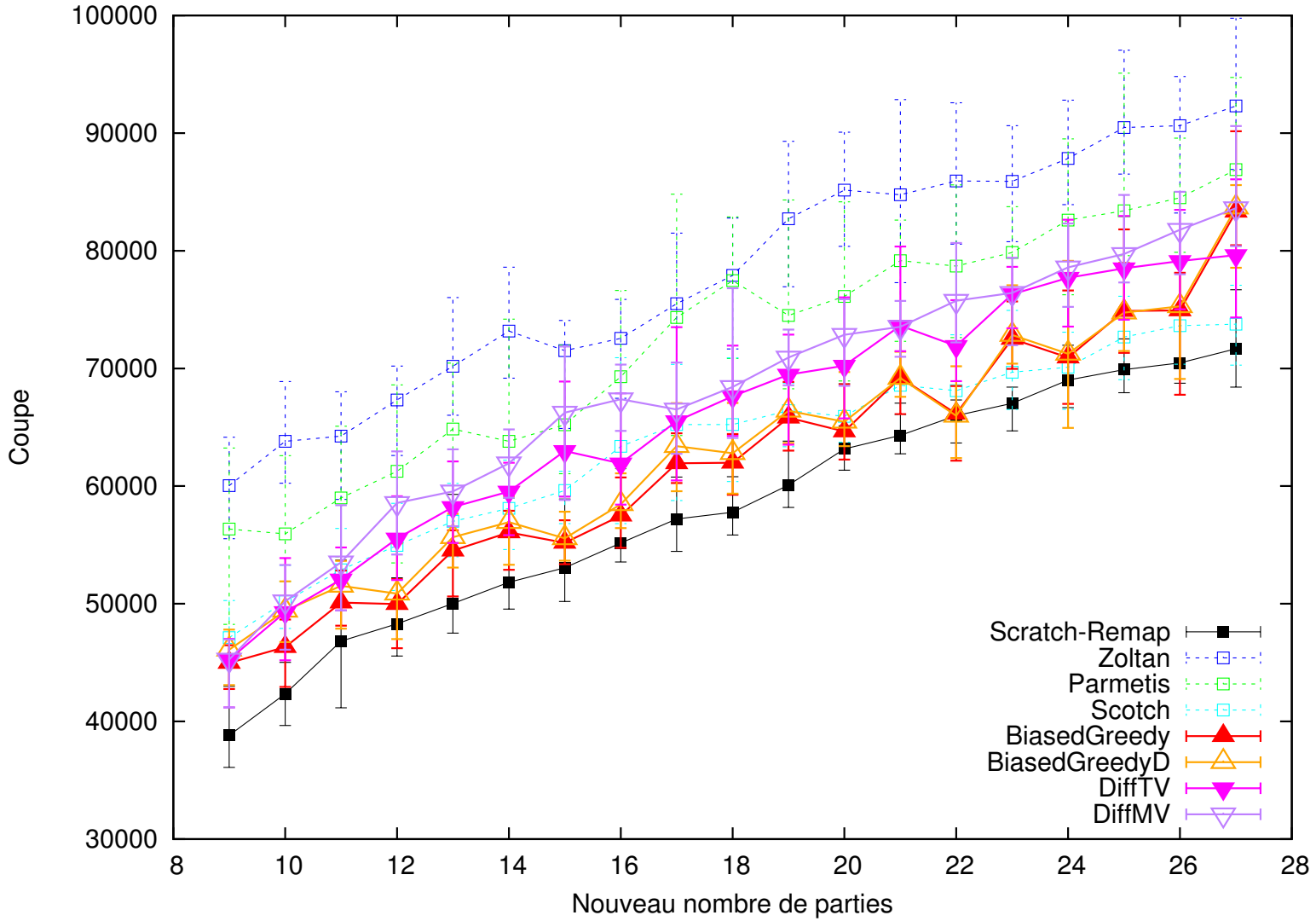


FIGURE 5.6 – Coupe suivant N pour un repartitionnement $8 \times N$.

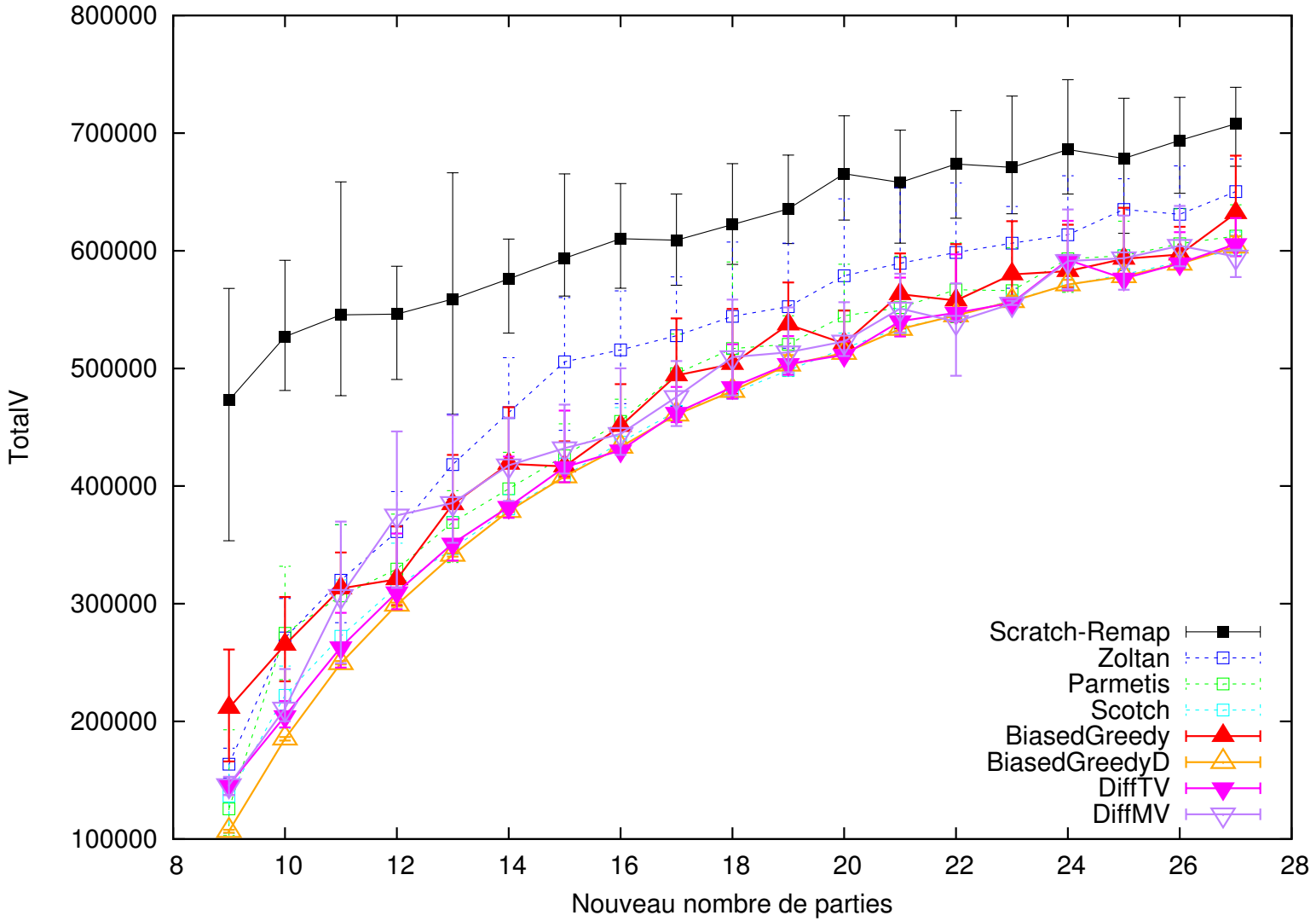


FIGURE 5.7 – TOTALV suivant N pour un repartitionnement $8 \times N$.

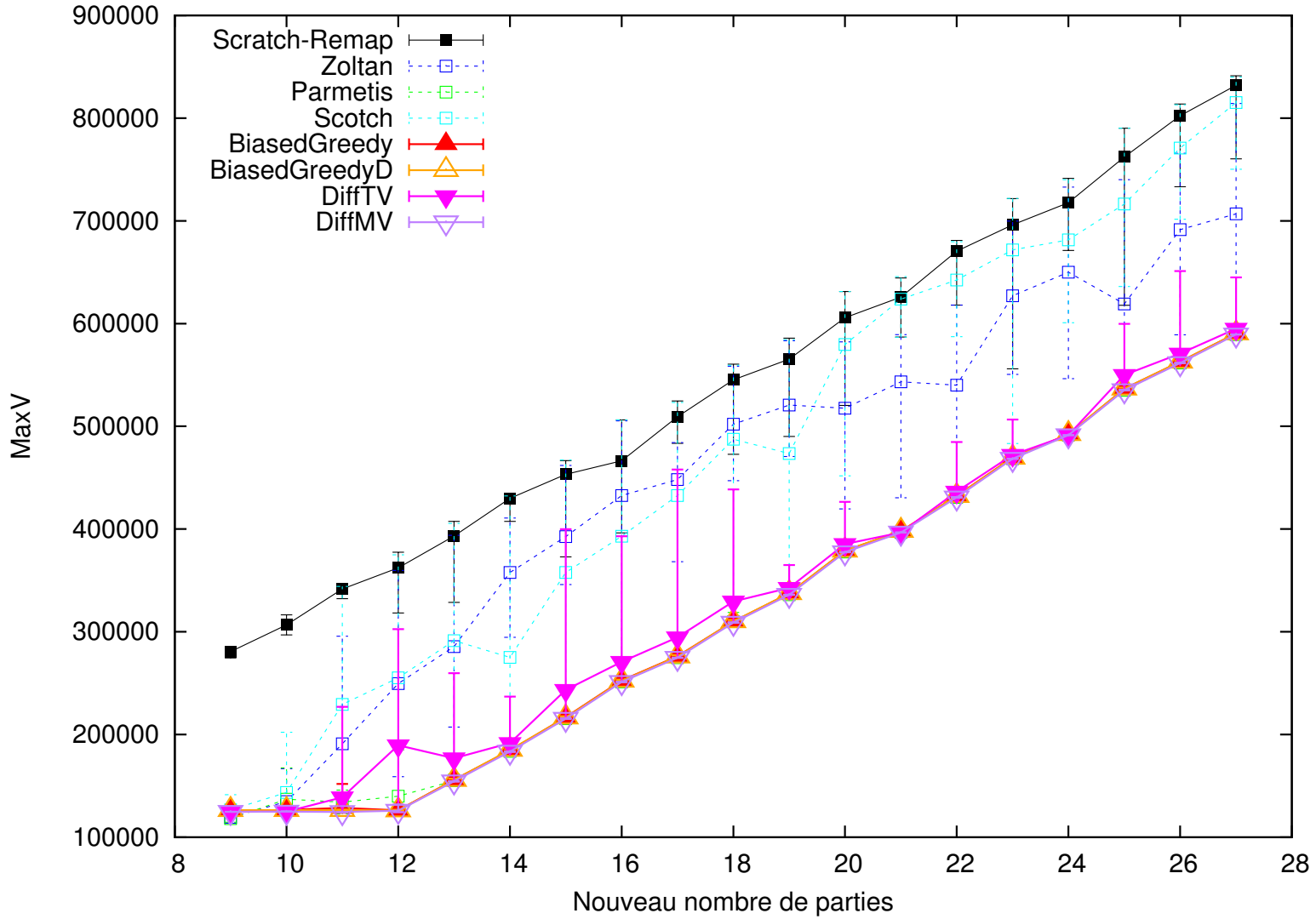


FIGURE 5.8 – MAXV suivant N pour un repartitionnement $8 \times N$.

sont confondues et donnent le MaxV le plus bas : *BiasedGreedy*, *BiasedGreedyD*, *DiffMV* et *ParMetis*. Il apparait ici que MaxV est une métrique minimisée par *ParMetis* mais pas par les autres repartitionneurs *Zoltan* et *Scotch*. La stratégie *DiffTV* donne un MaxV légèrement plus élevé. Il apparait que les métriques TotalV et MaxV ne peuvent pas être minimisées en même temps mais minimiser l'une suffit pour maintenir l'autre très basse.

Quand N est grand comparé à M, les valeurs optimales de MaxV augmentent linéairement à la même vitesse que les pires (la stratégie *Scratch-Remap*). La différence relative devient donc moins importante quand N est très grand.

Les figures 5.9 et 5.10 montrent respectivement le nombre de messages total (TotalZ) et par partie (MaxZ). Le nombre de messages est une métrique qui n'est pas habituellement prise en compte par les repartitionneurs. Toutes nos méthodes donnent un nombre de messages très bas alors que *Zoltan* et le *Scratch-Remap* donnent le nombre de message le plus élevé. *ParMetis* et *Scotch* donnent des résultats intermédiaires.

Les méthodes *BiasedGreedy* et *BiasedGreedyD* donnent le TotalZ le plus bas. En effet ces méthodes essayent de minimiser ce critère par construction à l'aide de la recherche des sous-ensembles (cf. section 3.3.3). L'optimisation de la diagonale n'influe pas sur ce critère. Bien que cette métrique n'est pas minimisée par la méthode *DiffTV*, la minimisation de TotalV semble également minimiser TotalZ dans la plupart des cas. Le nombre de messages pour *DiffMV* est plus élevé mais reste inférieur aux méthodes classiques de repartitionnement.

Le nombre maximal de messages par partie MaxZ fait apparaitre la différence entre *BiasedGreedy* et *BiasedGreedyD* : quand N et M sont proches, *BiasedGreedyD* donne un plus grand nombre de messages par partie. En particulier, dans le cas 8×9, MaxZ devient très élevé pour *BiasedGreedyD* et dépasse les repartitionneurs classiques. À partir du cas 8×15, les deux méthodes deviennent quasiment identiques.

Nous avons vu que nos méthodes permettent une meilleure migration selon les métriques TotalV, MaxV, TotalZ et MaxZ mais avec une coupe légèrement plus élevée. Le gain en terme de volume de migration se réduit quand N devient grand devant M mais la différence de coupe reste la même. Nos méthodes sont donc plus intéressantes quand l'écart entre l'ancien nombre de processeurs et le nouveau n'est pas trop grand. Malgré cela, nos méthodes de repartitionnement garde l'avantage d'un faible nombre de messages dans tous les cas.

La différence entre les deux variantes de notre repartitionnement biaisé *BiasedGreedy* et *BiasedGreedyD* se réduit avec l'augmentation du nombre de nouvelles parties : sauf lorsque la variation du nombre de processeurs est très petite, ces méthodes sont quasiment identiques.

5.4 Comparaison sur des graphes complexes

Pour valider nos approches de repartitionnement, nous présentons dans cette section, des résultats obtenus pour des graphes issus d'applications variées. Ces graphes, présentés dans le tableau 5.1, sont disponibles dans la collection de matrice creuse de l'université de Floride [13], à l'exception de *grid3d100* qui est une grille cubique régulière de $100 \times 100 \times 100$. Les mesures des différentes métriques sont données relativement au *Scratch-Remap*.

On étudie plus particulièrement le cas 8×12, la charge de la partition initiale a été augmentée de 50 %, ce qui donne un déséquilibre de 33 %. Les résultats dans le cas du repartitionnement 8×10 (augmentation de la charge de 25 % et déséquilibre de 17 %)sont disponibles dans l'annexe C.2.

La figure 5.11 donne les coupes obtenues avec les différents repartitionneurs relativement à celle du *Scratch-Remap*. Les repartitionneurs classiques, *Zoltan*, *ParMetis* et *Scotch*, donnent une coupe généralement légèrement supérieure à celle du *Scratch-Remap*, parfois égale. Dans le cas de

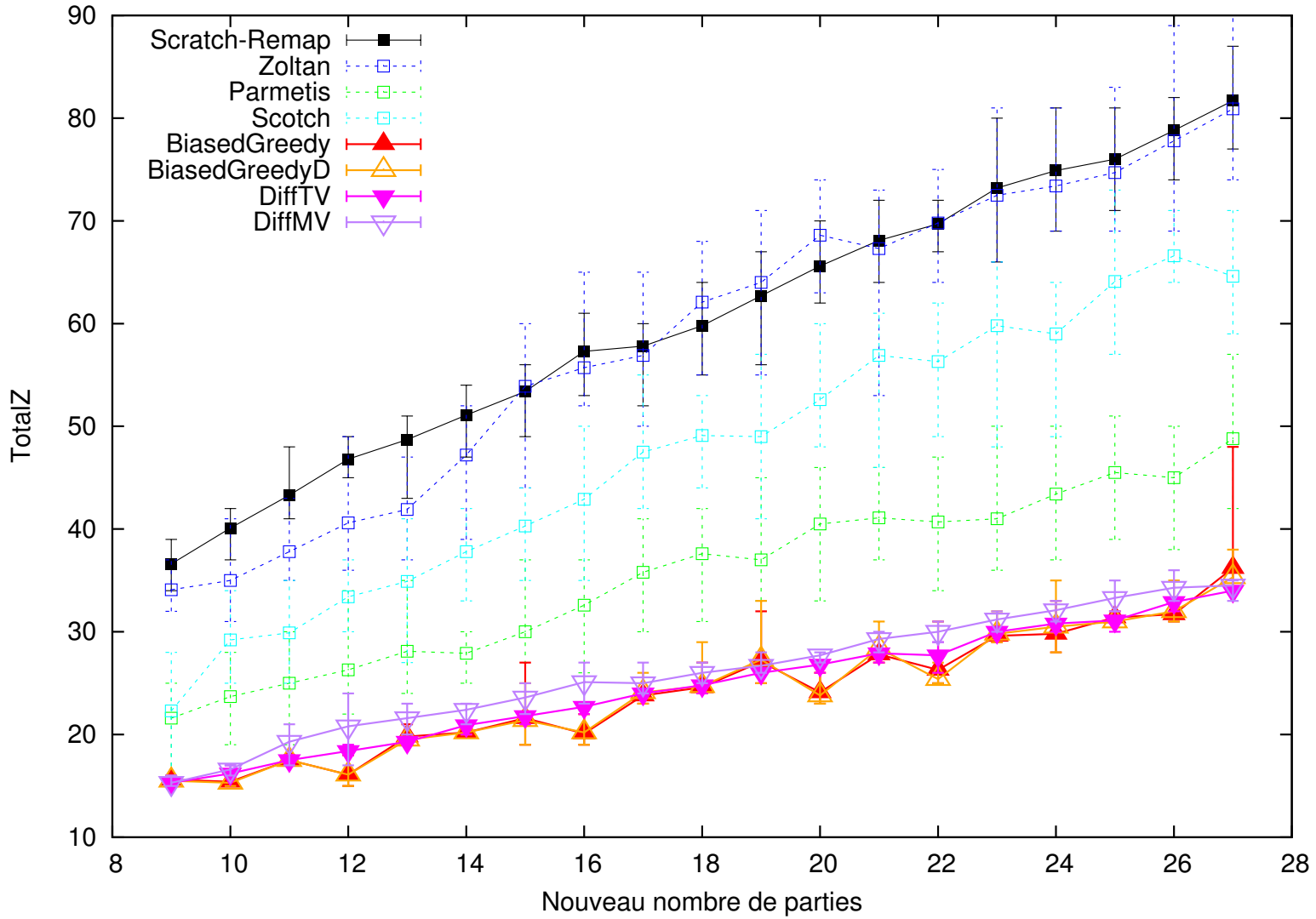


FIGURE 5.9 – TOTALZ suivant N pour un repartitionnement $8 \times N$.

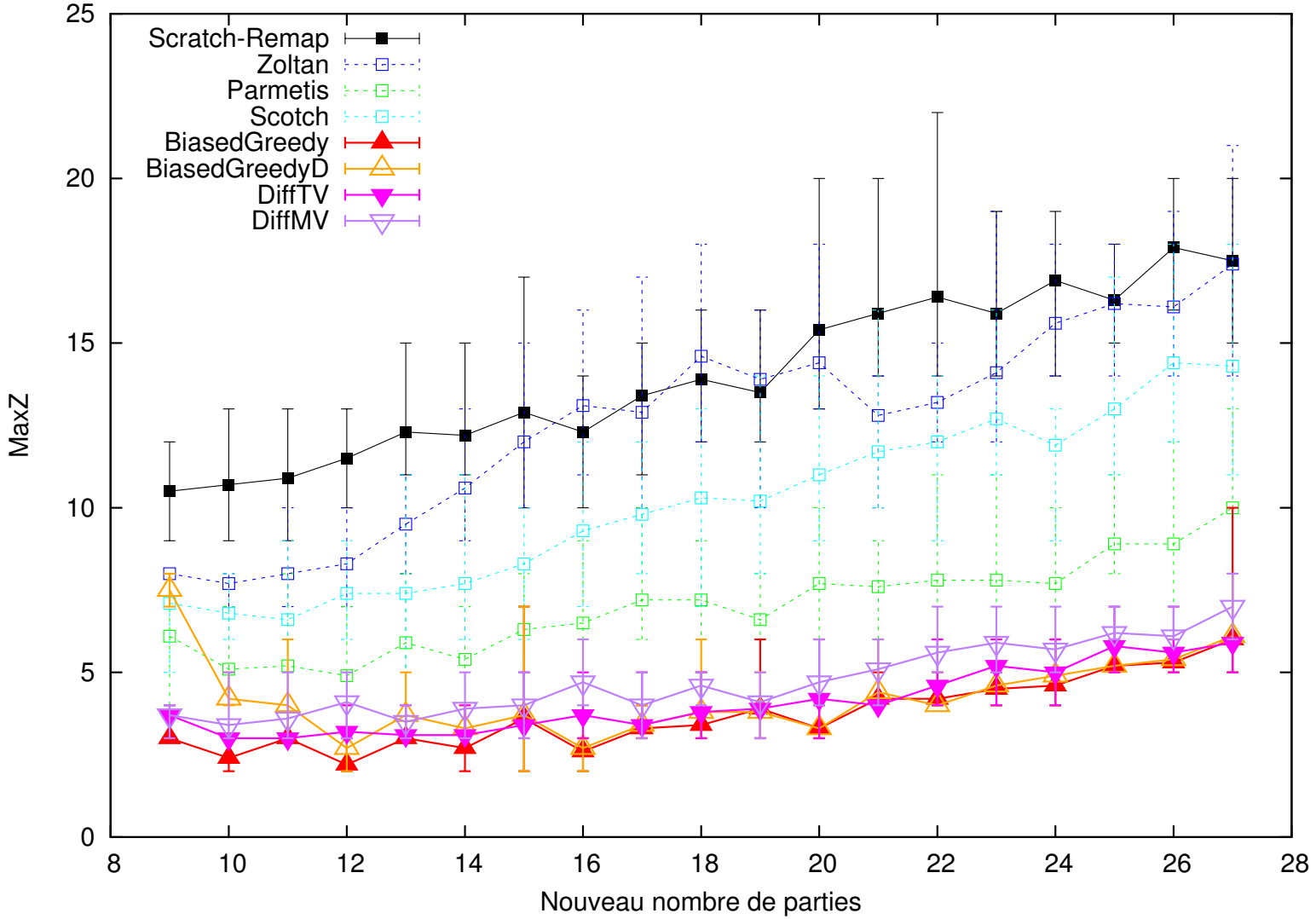


FIGURE 5.10 – MAXZ suivant N pour un repartitionnement $8 \times N$.

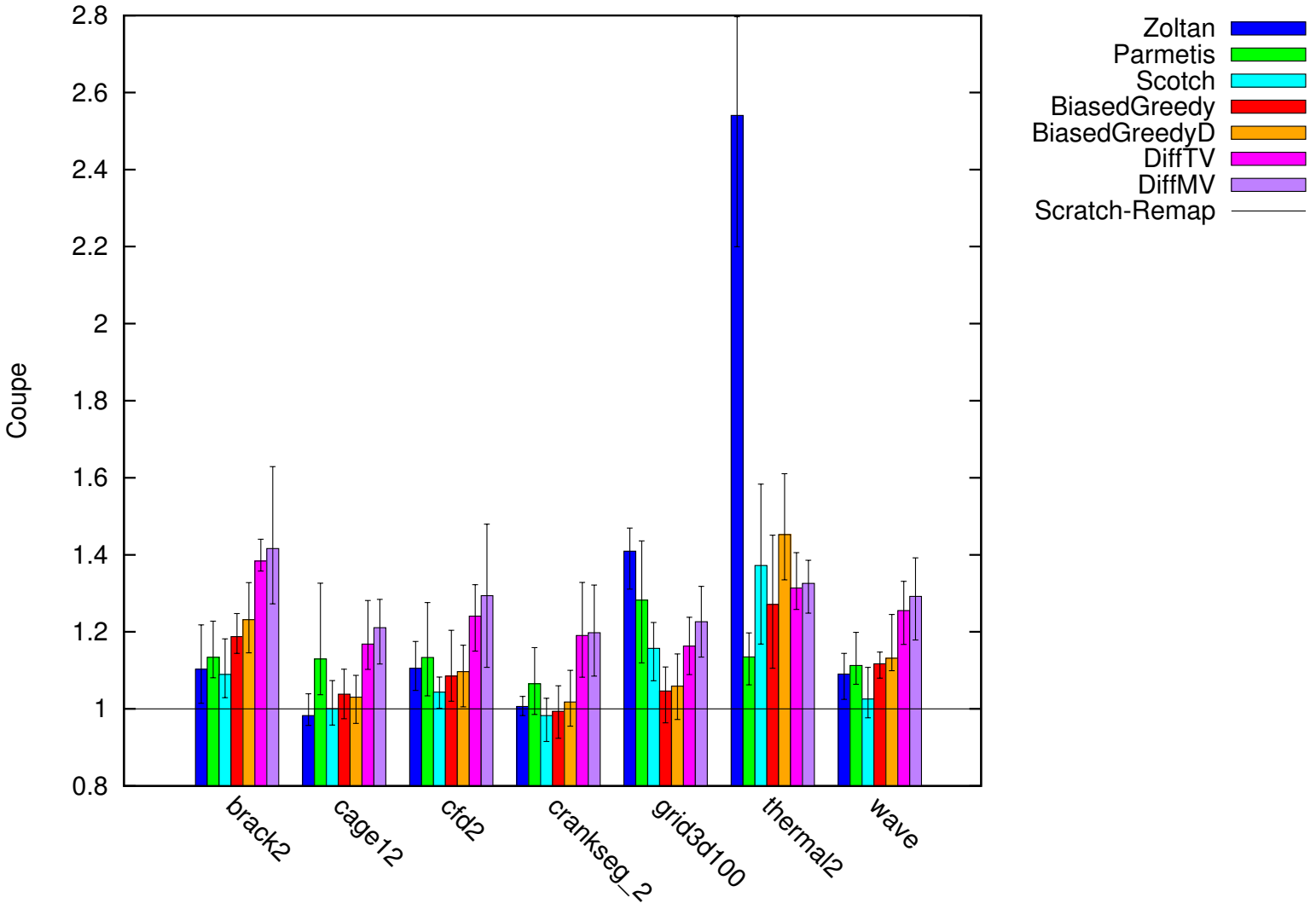


FIGURE 5.11 – Coupe relative au *Scratch-Remap* pour un repartitionnement 8×12.

Graphe	Description	$\|V\|$	$\|E\|$	d
grid3d100	Grille 3D régulière	1 000 000	2 970 000	5,94
cfd2	Mécanique des fluides numériques (*CFD*)	123 440	1 482 229	24,02
crankseg_2	Problème structurel	63 838	7 042 510	220,64
thermal2	Problème thermique	1 228 045	3 676 134	5,99
brack2	Problème 2D/3D	62 631	366 559	11,71
wave	Problème 2D/3D	156 317	1 059 331	13,55
cage12	Électrophorèse d'ADN	130 228	951 154	14,61

TABLE 5.1 – Description des graphes utilisés pour nos expériences. Le degré moyen d est calculé à l'aide de la formule $\frac{2\times|E|}{|V|}$.

thermal2, *Zoltan* ne trouve pas de bonne partition. Les stratégies *BiasedGreedy* et *BiasedGreedyD* donnent une coupe comparable à ces partitionneurs classiques. Les méthodes diffusives (*DiffTV* et *DiffMV*), ne profitant pas d'un partitionneur multi-niveaux complet, donnent une coupe souvent plus élevée.

Comme on peut le voir sur la figure 5.12, toutes nos méthodes donnent un nombre de messages TOTALZ bien inférieur aux autres repartitionneurs. Les deux stratégies *Greedy* donnent un TOTALZ à peu près équivalent et légèrement inférieur aux méthode diffusives. Bien que cette métrique ne soit pas minimisée par les programmes linéaires utilisés, le TOTALZ obtenu est bas. C'est un effet de bord de la minimisation de TOTALV ou MAXV par le programme linéaire : en minimisant le volume de migration, beaucoup de communications possibles, d'après le graphe quotient enrichi choisi $\tilde{Q}$, sont de volume nul. Le petit nombre d'arêtes ajoutées dans $\tilde{Q}$ pour les nouvelles parties favorise également un nombre de messages bas.

Pour le graphe *crankseg_2*, on remarque que le nombre de messages pour les stratégies biaisées est anormalement élevé. Le partitionneur n'a pas trouvé de partition respectant le schéma de communication imposé par les sommets fixes. Malgré l'utilisation d'un algorithme de partitionnement k-aire direct, les heuristiques utilisées ne permettent pas toujours de trouver une bonne solution. Cette échec est sûrement dû au degré élevé de ce graphe qui rend le partitionnement plus difficile. Les méthodes diffusives appliquent strictement la matrice de migration et n'ont pas ce problème. Seule la coupe peut être dégradée avec ces méthodes diffusives, si l'heuristique choisissant les sommets à migrer fonctionne mal.

Les volumes totaux de migrations TOTALV sont présentés sur la figure 5.13. À l'exception de *crankseg_2* pour lequel les méthodes biaisées échouent, *BiasedGreedyD* donne toujours le volume de migration total le plus bas et aussi le plus stable (d'après les barres d'erreur). Comme on l'a déjà vu, *DiffTV* est contraint par le graphe quotient enrichi utilisé et ne peut pas autant réduire le volume de migration que *BiasedGreedyD*.

Concernant MAXV (figure 5.14), la comparaison entre *DiffTV* et *DiffMV* est inversée par rapport à TOTALV. Les minimisations de TOTALV et de MAXV ne sont pas liées. Minimiser l'un ne suffit pas à avoir l'autre minimal. Les deux méthodes *Greedy* donnent également un MAXV très bas. Parmi les autres repartitionneurs, seul *ParMetis* semble minimiser ce critère. *Zoltan*, *Scotch* et le *Scratch-Remap* donnent des volumes de migrations par partie (MAXV) beaucoup plus élevés.

Toutes nos méthodes de repartitionnement donnent un nombre de messages par partie (MAXZ) plus faible que les autres repartitionneurs (figure 5.15). La méthode *BiasedGreedy* donne généralement les meilleurs MAXZ bien que les heuristiques peuvent mal fonctionner dans certains cas (*cage12* et *crankseg_2*). La méthode *BiasedGreedyD* donne un MAXZ légèrement plus élevé que *BiasedGreedy* : dans le cas 8×12, les anciens et nouveaux nombres de parties ne sont pas

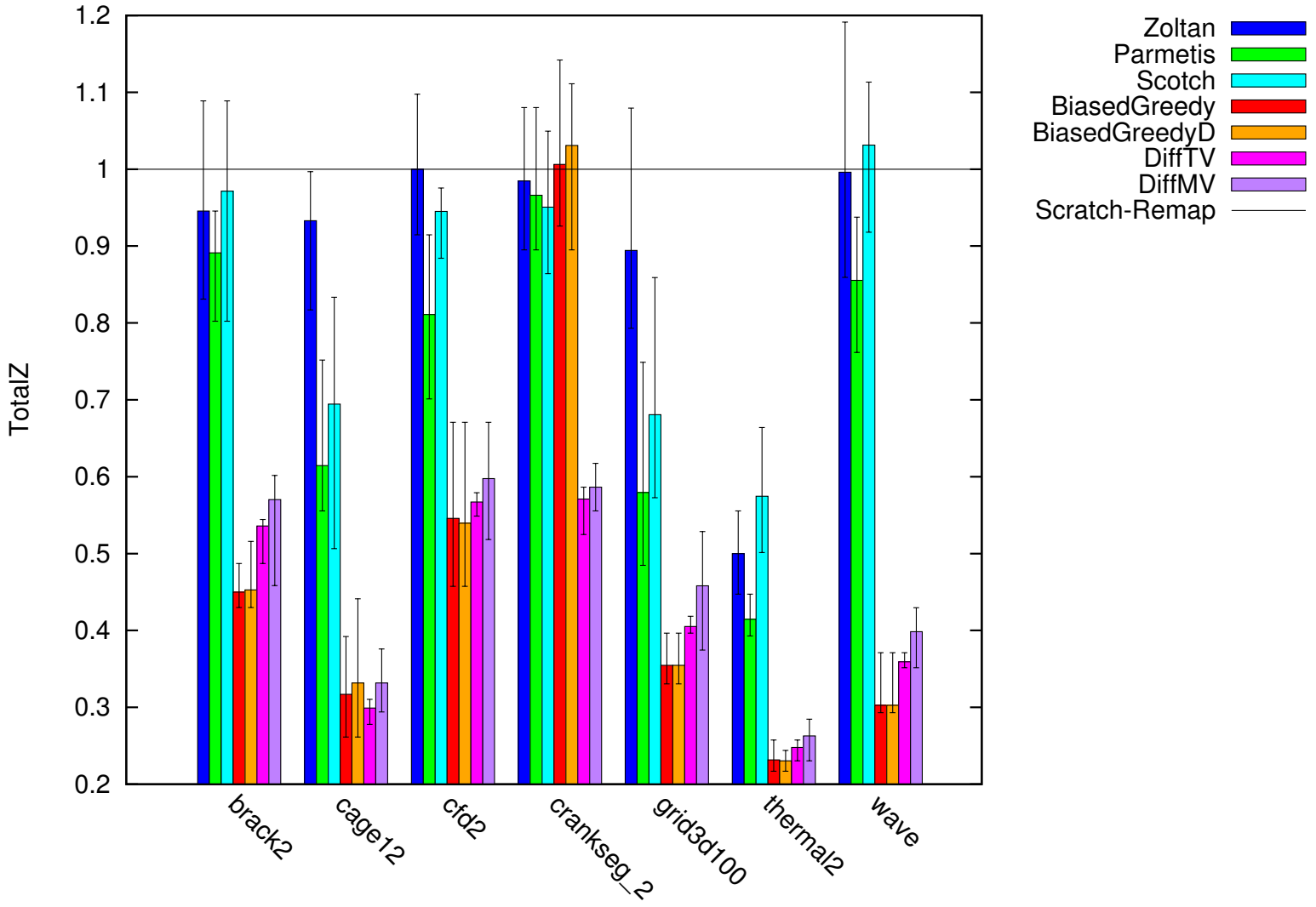


FIGURE 5.12 – TOTALZ relatif au *Scratch-Remap* pour un repartitionnement 8×12.

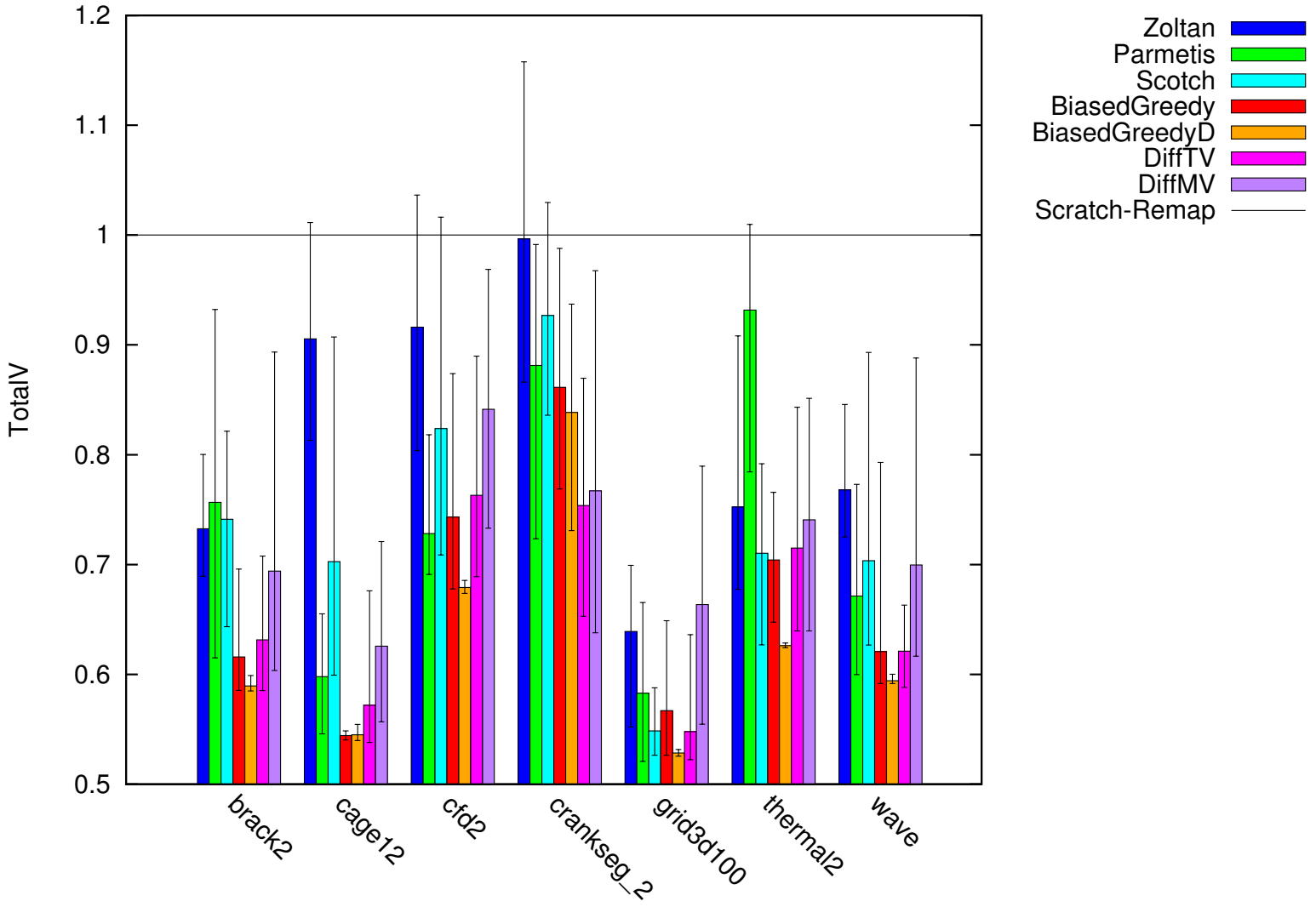


FIGURE 5.13 – TOTALV relatif au *Scratch-Remap* pour un repartitionnement 8×12.

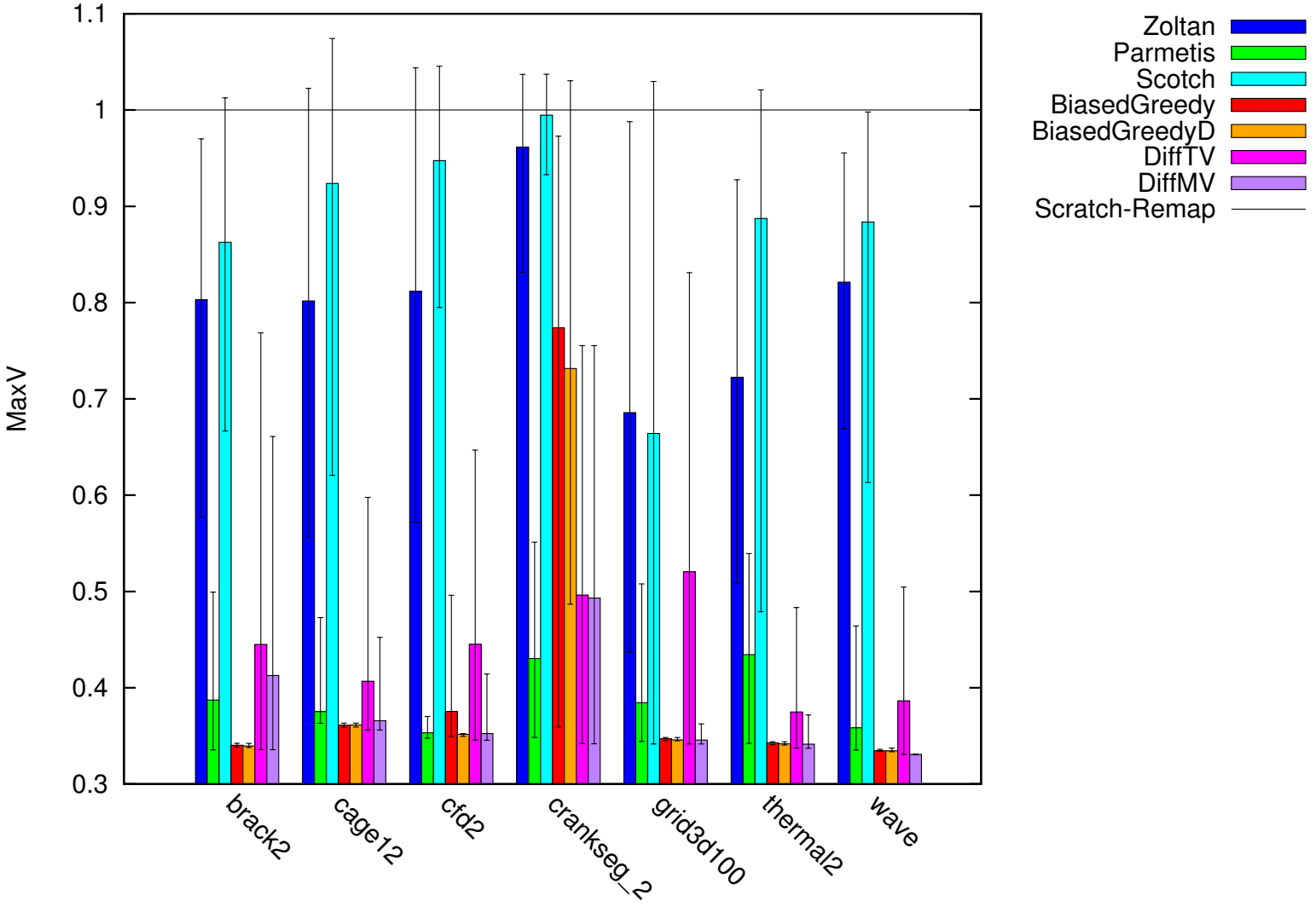


FIGURE 5.14 – MAXV relatif au *Scratch-Remap* pour un repartitionnement 8×12.

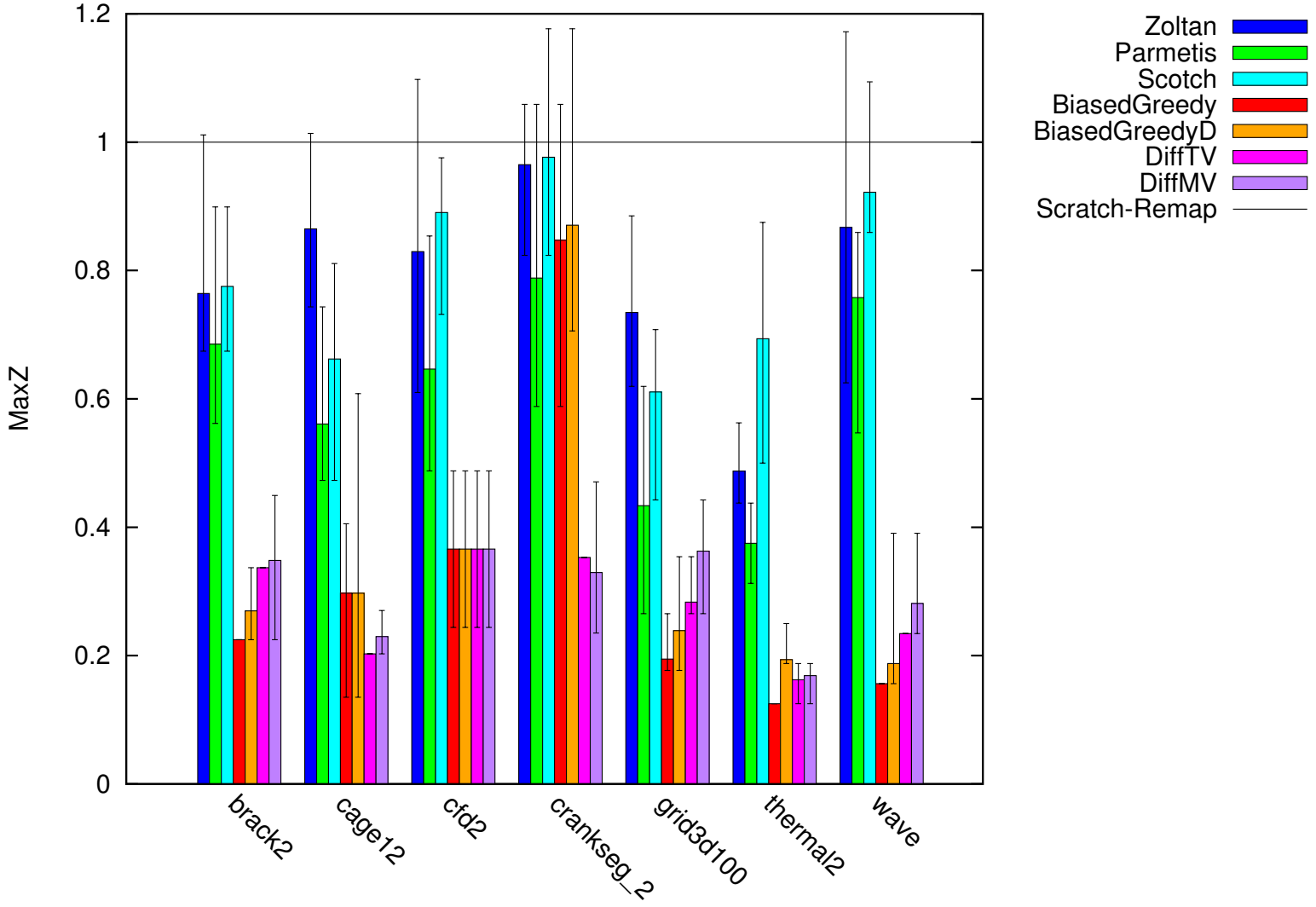


FIGURE 5.15 – MAXZ relatif au *Scratch-Remap* pour un repartitionnement 8×12.

trop proches. La distinction entre *BiasedGreedy* et *BiasedGreedyD* est plus significative quand le nombre de parties change peu, par exemple dans le cas 8×10 (figure C.7). Les méthodes *DiffTV* et DiffMV ont également un MaxZ bas. En effet, MaxZ est limité par le degré du graphe quotient enrichi utilisé par le programme linéaire et celui-ci est construit de façon à garder un degré faible (au plus 3 arêtes pour chaque partie ajoutée).

Le cas 8×10 présenté dans l'annexe C.2, donne des résultats similaires mais la différence entre *BiasedGreedy* et *BiasedGreedyD* est plus marqué par rapport aux critères TotalV et MaxZ.

Les résultats dans le cas 8×10 sont assez similaires mais avec une différence plus marquée entre *BiasedGreedy* et *BiasedGreedyD* sur les critères TotalV et MaxV.

Ces expériences valident nos algorithmes de repartitionnement $M \times N$ sur des graphes complexes pour une augmentation de charge de 50 % (8×12) ou de 25 % (8×10). Nos méthodes sont capables de mieux optimiser les différentes métriques liées à la migration (TotalV, MaxV, TotalZ et MaxZ) que les autre repartitionneurs, en obtenant une coupe légèrement supérieure à une méthode *Scratch-Remap* et comparable aux autres repartitionneurs.

Les méthodes utilisant le partitionnement biaisé donnent les meilleurs résultats mais manquent de fiabilité. Le partitionnement est biaisé pour préférer appliquer une matrice de migration donnée mais les heuristiques de partitionnement peuvent faire des mauvais choix (par exemple dans le cas de *crankseg_2*.

Les méthodes diffusives sont plus fiables : la matrice de migration est strictement appliquée mais les résultats sont de moins bonne qualité. La migration peut être mal optimisée à cause d'un mauvais choix du graphe quotient enrichi. De plus, notre mise en œuvre rudimentaire de repartitionnement diffusif ne profite pas d'une approche multi-niveaux et se fait en une seule passe ; la coupe n'est donc pas très bien minimisée.

5.5 Étude de la complexité en temps

Dans cette section, nous étudions le temps d'exécution du partitionnement biaisé et des repartitionneurs classiques. Les méthodes diffusives ne sont pas évalués car leur mise en œuvre actuelle n'est que préliminaire [2].

Les deux paramètres importants conditionnant la complexité du repartitionnement sont la taille du graphe ($|V|$ et $|E|$) et le nombre de parties finales (N).

Pour étudier l'influence du nombre de parties (figure 5.16), nous repartitionnons la grille 3D $100 \times 100 \times 100$ dans le cas $M \times N$ avec M et N variables mais en gardant le rapport $N/M = 3/2$ fixe.

Alors que la méthode de *Scratch-Remap* et les autres repartitionneurs ont une complexité en temps logarithmique par rapport au nombre de parties grâce à la méthode des bissections récursives, notre partitionnement biaisé est très rapidement ralenti avec l'augmentation du nombre de parties.

Lorsque le nombre de parties est petit, le temps d'exécution de notre partitionnement biaisé est légèrement supérieur au repartitionneur de *Scotch* mais suit la même progression. Notre partitionnement biaisé et le repartitionnement de *Scotch* se basent sur le même partitionneur (ils utilisent les mêmes algorithmes de contraction et de raffinement du graphe) mais notre

2. En effet, l'heuristique de type FM utilisée pour migrer le sommets de chaque message, ne profite pas d'une approche multi-niveaux et chaque message est traitée comme un partitionnement complet.

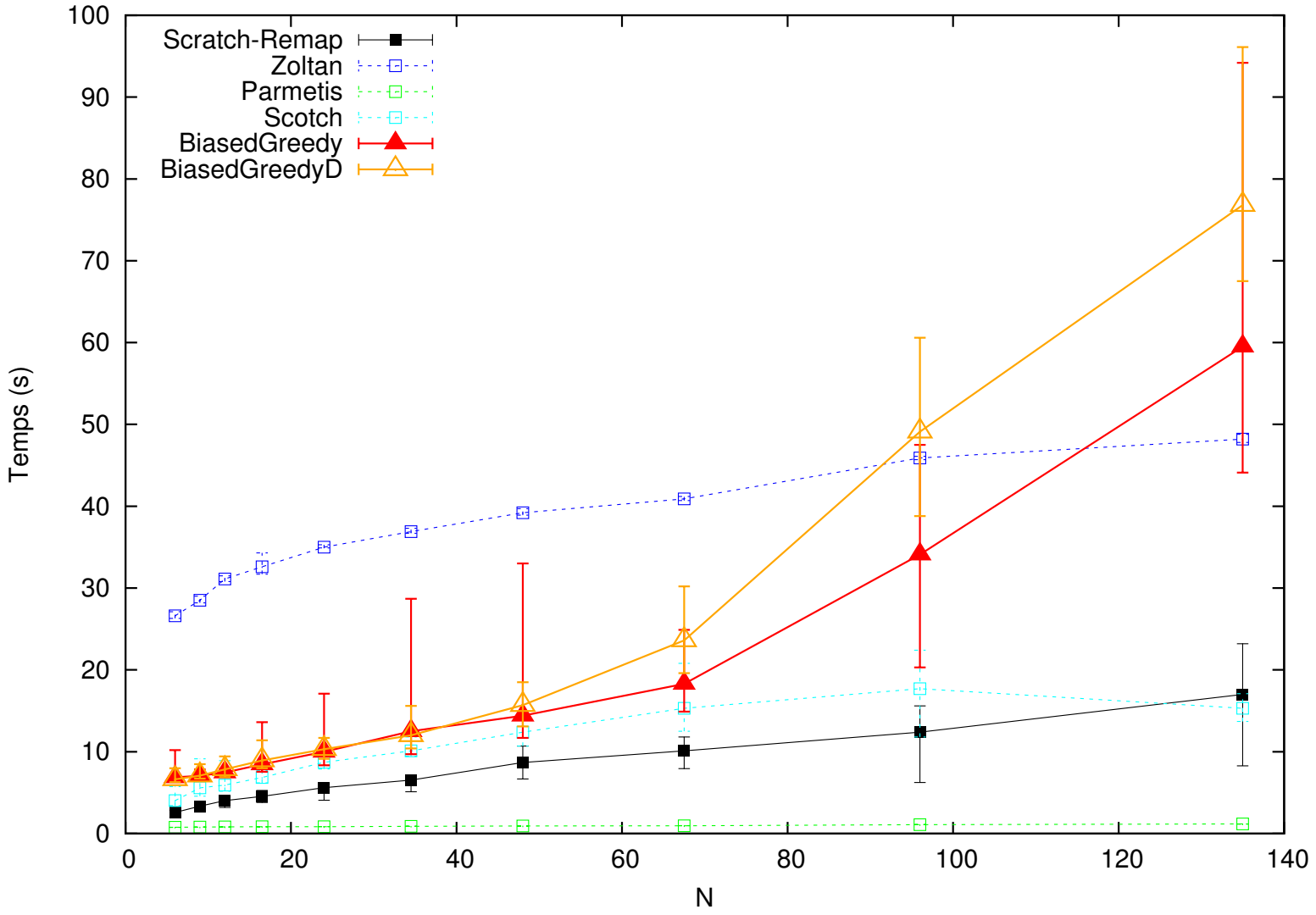


FIGURE 5.16 – Temps d'exécution du repartitionnement $M \times N$ suivant le nouveau nombre de parties N avec $M = \frac{2}{3}N$.

partitionnement biaisé utilise un graphe enrichi qui possède plus d'arêtes (chaque message ajoute autant d'arêtes qu'il y a de sommets dans une ancienne partie).

Quand le nombre de parties est plus important, le partitionnement biaisé ralentit beaucoup plus rapidement. Le KGGGP utilisé en tant que partitionnement initial dans l'algorithme multi-niveaux devient prédominant par rapport aux phases de contraction et d'expansion, à cause de sa complexité linéaire par rapport au nombre de parties (cf. section 4.2.3).

Dans une seconde expérience, nous générons des grilles 3D cubiques de tailles variables qui sont repartitionnées dans le cas 8×12. La figure 5.17 donne les temps d'exécution en fonction du nombre de sommets $|V|$ qui est à peu près proportionnel au nombre d'arêtes $|E|$ (il y a environ 3 arêtes pour chaque sommet). Comme les autres méthodes, notre partitionnement biaisé a une complexité en temps linéaire par rapport à la taille du graphe.

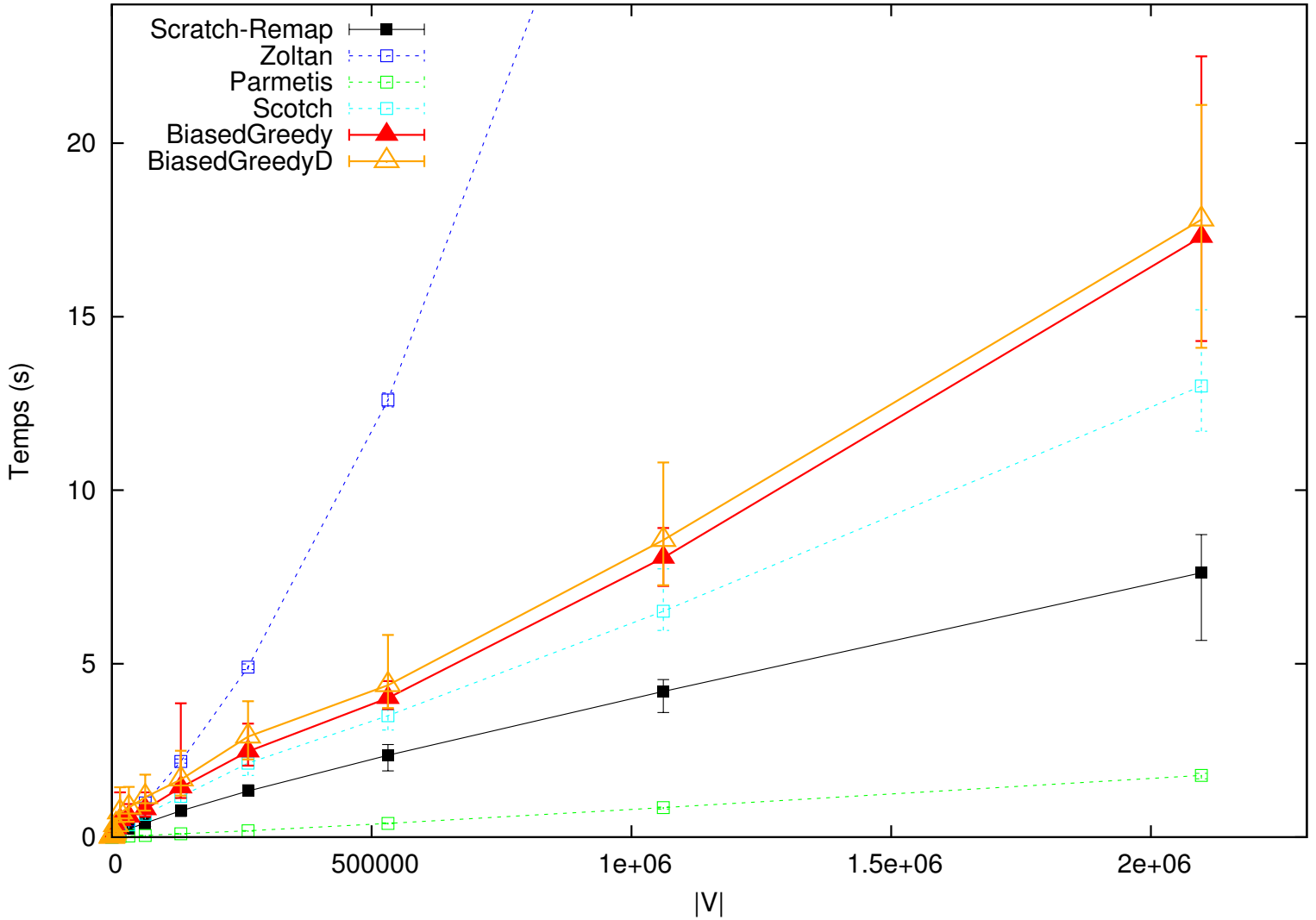


FIGURE 5.17 – Temps d'exécution du repartitionnement 8×12 suivant la taille du graphe ($|V|$).

Chapitre 6

Conclusion et Perspectives

6.1 Conclusion

Nous avons présenté dans cette thèse plusieurs méthodes de rééquilibrage dynamique avec un nombre variable de processeurs, basées sur le repartitionnement de graphe. Les objectifs de ce repartitionnement sont de rééquilibrer la partition et de minimiser la coupe tout en optimisant la migration caractérisée par les métriques TotalV (le volume total de migration), MaxV (le volume maximal de migration par partie), TotalZ (le nombre total de messages) et MaxZ (le nombre maximal de messages par partie).

Notre méthode se décompose en deux grandes étapes : d'abord une matrice de migration est construite pour optimiser les différentes métriques de migration ; puis un algorithme de repartitionnement calcule une nouvelle partition du graphe en prenant en compte la matrice de migration précédemment construite.

Une étude théorique du repartitionnement dans le cas où l'ancienne partition est équilibrée nous a permis de trouver les valeurs optimales pour TotalV et TotalZ. Plusieurs algorithmes ont été présentés pour essayer de construire des matrices de migration optimales. Les méthodes *1D*, *Matching*, *Greedy* et *GreedyD* permettent d'obtenir une nombre de messages TotalZ quasi optimal. De plus, les méthodes *Matching* et *GreedyD* minimisent TotalV mais au coût d'un MaxZ parfois très élevé. Les méthodes *1D* et *Greedy* gardent un MaxZ faible, mais donnent un TotalV légèrement plus élevé. La méthode *Matching* n'est utilisable que dans le cas équilibré ou quasiment équilibré, ce qui limite son intérêt en pratique.

Nous avons également présenté un programme linéaire (*PL*) permettant d'optimiser une des différentes métriques TotalV, MaxV, TotalZ ou MaxZ, ou combinaison linéaire de celles-ci, étant donné un graphe quotient enrichi indiquant les messages autorisés. Un mauvais choix du graphe quotient enrichi limite la qualité des résultats de cette méthode.

Nous avons ensuite proposé deux méthodes de repartitionnement de graphe (*Biased* et *Diff*) se basant sur une matrice de migration construite à l'étape précédente. Le partitionnement biaisé utilisant des sommets fixes permet d'obtenir une nouvelle partition de qualité offrant une bonne coupe mais rend le partitionnement plus difficile et donc plus lent. Cette méthode a aussi l'inconvénient de nécessiter une nombre de message TotalZ minimal. On préférera donc l'utiliser conjointement avec les méthodes de construction de matrice minimisant cette métrique. Le partitionnement diffusif permet d'appliquer strictement une matrice de migration quel que soit son nombre de messages. Mais notre mise œuvre préliminaire n'utilise qu'une passe sans

multi-niveaux et donne par conséquent une coupe plus élevée.

Nous avons aussi montré qu'il est possible de modéliser le nombre de messages TotalZ et le volume de migration total TotalV à l'aide de la coupe d'un hypergraphe enrichi. Il est ainsi possible de repartitionner sans calculer une matrice de migration à l'avance. Mais l'hypergraphe ainsi créé est très complexe et très difficile à partitionner avec les outils actuels.

Dans le cadre de notre méthode de partitionnement biaisé, nous avons mis en évidence des limitations du partitionnement basé sur les bissections récursives. Nous avons donc réalisé une méthode de partitionnement k-aire direct intégrée dans le partitionneur *Scotch*. Cette méthode de KGGGP (*k-way greedy graph growing partitioning*) permet de mieux appliquer notre méthode de partitionnement biaisé mais possède une complexité par rapport au nombre de parties plus importante que les méthodes de bissections récursives.

Les méthodes de repartitionnement biaisé et diffusif ont été évaluées par une suite d'expériences. Nous avons vu que ces approches sont plus intéressantes quand le nombre de processeurs varie peu. Quand le nombre de processeurs varie beaucoup, une méthode *Scratch-Remap* donne un volume de migration équivalent mais une meilleure coupe, tout en étant plus simple à réaliser. Néanmoins, même dans ce cas, nos méthodes permettent d'obtenir un nombre de messages bien plus faible. Nous avons validé ces méthodes sur une série de graphes complexes issus d'applications réelles. Le partitionnement biaisé offre de bons résultats mais il ralentit rapidement avec l'augmentation du nombre de parties. Les résultats obtenus à l'aide du partitionnement diffusif sont encourageants mais laissent place à des améliorations. L'état de la mise en œuvre du partitionnement diffusif ne nous permet pas actuellement de conclure sur sa rapidité d'exécution.

La mise en œuvre de ces algorithmes et les modifications apportées à *Scotch* pour le KGGGP sont intégrées dans la bibliothèque LBC2 [16] dont le code source est disponible à l'adresse : `https://gforge.inria.fr/frs/download.php/33277/lbc2-r1844.tar.gz`.

6.2 Perspectives

En vue d'améliorer les différents algorithmes présentés dans cette thèse, plusieurs pistes sont envisagées.

Nous avons vu que la qualité de la matrice de migration obtenue par le programme linéaire est limitée par le choix du graphe quotient enrichi. Un meilleur algorithme d'enrichissement du graphe quotient devrait être développé pour pouvoir effectivement atteindre des volumes de migration minimaux. De plus, l'algorithme présenté dans cette thèse ne permet pas de traiter le cas où le nombre de processeurs diminue.

Le programme linéaire construisant la matrice de migration permet de minimiser une combinaison linéaire des métriques de migration. Cet objectif mélange des métriques de natures très différentes. La recherche des coefficients permettant d'obtenir un bon compromis est un problème complexe qui mériterait d'être étudié.

Pour améliorer à la fois la qualité de la coupe et le temps d'exécution de notre méthode diffusive, il faudrait utiliser celle-ci avec une approche multi-niveaux. Ce repartitionnement serait alors une méthode hybride utilisant un algorithme de *local matching* [58] pour la phase de contraction (permettant ainsi de garder l'information de l'ancienne partition sur le graphe contracté), le partitionnement diffusif lors du partitionnement initial, puis une méthode de raffinement biaisé lors de l'expansion.

Le principal problème de notre méthode de partitionnement biaisé est son temps d'exécution. Celui-ci est principalement dû à notre partitionnement KGGGP en $O(k|E|)$. Pour réduire sa complexité, il faudrait ne pas considérer pour chaque sommet tous les déplacements vers les k parties, mais par exemple seulement vers les parties voisines ou en s'inspirant de certaines méthodes de raffinement k-aire ayant une bonne complexité en $O(|E|)$ [35].

Une autre cause de ralentissement du partitionnement biaisé est l'ajout d'un grand nombre d'arêtes. Une solution serait l'utilisation d'*arêtes virtuelles* : au lieu d'ajouter des sommets fixes et des arêtes de migration, les heuristiques de partitionnement sont modifiées pour prendre directement en compte le repartitionnement $M \times N$. C'est, par exemple, l'approche déjà utilisée dans *Scotch* [18].

Pour pouvoir appliquer ces méthodes dans le cadre d'applications réelles, il serait nécessaire de rendre nos algorithmes parallèles. Pour paralléliser le repartitionnement, il est possible de réutiliser les partitionneurs parallèles (*PT-Scotch*, *Zoltan*, *ParMetis*, ...). Pour paralléliser la méthode KGGGP, comme celle-ci est utilisée sur un petit graphe contracté grâce au multi-niveaux, une méthode naïve serait de profiter de l'aléatoire pour calculer des partitions différentes sur chaque processeur, puis de diffuser la meilleure partition obtenue [9, 50].

Le repartitionnement $M \times N$ pourrait aussi trouver une application dans l'optimisation des communications entre des codes couplés. Actuellement chaque code est partitionné indépendamment de l'autre alors que les processeurs des différents codes communiquent régulièrement entre eux. Pour optimiser ces communications, il faudrait utiliser une méthode de *co-partitionnement*.

Une approche consisterait à commencer par partitionner l'un des codes, puis de partitionner le second code en prenant en compte la partition de l'autre (comme dans un repartitionnement). Dans ce contexte, il n'est pas nécessaire de minimiser le volume total de migration TotalV, car toutes les données de couplage sont systématiquement échangées lors de cette étape. En revanche, il est toujours possible d'optimiser le nombre de messages (TotalZ ou MaxZ) ainsi que la taille de chacun (MaxV).

Annexe A

Programmes linéaires pour optimiser les différentes métriques

Dans chaque programme, les entrées sont :
— la variation de charge d_i idéale pour chaque processeur i (la différence entre la nouvelle charge équilibrée pour la partition en N parties et l'ancienne charge dans la partition déséquilibrée en M parties) ;
— une tolérance au déséquilibre ub qui correspond à la surcharge maximale autorisée.

On cherche à calculer les valeurs optimales des e_{ij} qui est la quantité de données envoyées du processeur i vers le processeurs j.

A.1 Minimisation de TotalV

$$\text{minimiser} \quad \sum_{i,j} e_{ij}$$

Contraintes linéaires :

$$\forall i \in V, \quad v_i = \sum_j (e_{ji} - e_{ij})$$

Bornes :

$$\begin{aligned} &\forall i \in V, && v_i \leq d_i + ub \\ &\forall (i,j) \in E, && 0 \leq e_{ij} \end{aligned}$$

A.2 Minimisation de MaxV

$$\text{minimiser} \quad v$$

Contraintes linéaires :

$$\forall i \in V, \quad v_i = \sum_j (e_{ji} - e_{ij})$$
$$\forall i \in V, \quad v'_i = \sum_j (e_{ji} + e_{ij}) - v$$

Bornes :

$$\begin{aligned} &\forall i \in V, && v_i \leq d_i + ub \\ &\forall i \in V, && v'_i \leq 0 \\ &\forall (i,j) \in E, && 0 \leq e_{ij} \\ & && 0 \leq v \end{aligned}$$

A.3 Minimisation de TotalZ

x_{ij} et e_{ij} entiers.

$$\text{minimiser} \quad \sum_{ij} x_{ij}$$

Contraintes linéaires :

$$\begin{aligned} &\forall i \in V, && v_i = \sum_j (e_{ji} - e_{ij}) \\ &\forall (i,j) \in E, && a_{ij} = e_{ij} - x_{ij} \\ &\forall (i,j) \in E, && b_{ij} = e_{ij} - W x_{ij} \end{aligned}$$

Bornes :

$$\begin{aligned} &\forall i \in V, && v_i \leq d_i + ub \\ &\forall i \in V, && 0 \leq a_{ij} \\ &\forall i \in V, && b_{ij} \leq 0 \\ &\forall (i,j) \in E, && 0 \leq e_{ij} \\ &\forall (i,j) \in E, && 0 \leq x_{ij} \leq 1 \end{aligned}$$

A.4 Minimisation de MaxZ

x_{ij} et e_{ij} entiers.

$$\text{minimiser} \quad z$$

Contraintes linéaires :

$$\begin{aligned}
&\forall i \in V, && v_i = \sum_j (e_{ji} - e_{ji}) \\
&\forall (i,j) \in E, && a_{ij} = e_{ij} - x_{ij} \\
&\forall (i,j) \in E, && b_{ij} = e_{ij} - W x_{ij} \\
&\forall i \in V, && z_i = \sum_j (x_{ij} + x_{ji}) - z
\end{aligned}$$

Bornes :

$$\begin{aligned}
&\forall i \in V, && v_i \leq d_i + ub \\
&\forall i \in V, && 0 \leq a_{ij} \\
&\forall i \in V, && b_{ij} \leq 0 \\
&\forall i \in V, && z_i \leq 0 \\
&\forall (i,j) \in E, && 0 \leq e_{ij} \\
&\forall (i,j) \in E, && 0 \leq x_{ij} \leq 1 \\
& && 0 \leq z
\end{aligned}$$

Annexe B

Documentation du KGGGP dans *Scotch*

Notre méthode de partitionnement KGGGP a été réalisée en modifiant *Scotch 6.0*. Cette méthode est indiquée dans les chaînes de stratégie par la lettre **g**.

Cette stratégie accepte les paramètres suivants :

- **bal** indique la tolérance au déséquilibre utilisée. La valeur par défaut est 0.05 (5 %).
- **type** indique le critère de sélection utilisé. Les valeurs acceptées sont :
  - **g** pour le gain de coupe (par défaut) ;
  - **d** pour le critère *diff*.
- **seed** indique le type de graines à utiliser :
  - **n** pour n'utiliser aucune graine (par défaut) ;
  - **r** pour utiliser des graines aléatoire ;
  - **o** pour utiliser des graines éloignées.

  Les graines **r** et **o** ne sont pas compatibles avec le partitionnement à sommets fixes.
- **pass** indique le nombre de passes à effectuer. La meilleure partition est retenue.
- **center** indique s'il faut utiliser les centres des parties comme graines à la passe suivante. Les valeurs possibles sont **y** pour les utiliser ou **n** pour ne pas les utiliser (par défaut). Si les centres sont utilisés comme graines, le type de graine donné par **seed** n'est utilisé que pour la première passe.
- **ref** donne la stratégie de raffinement à appliquer après chaque passe. Aucune stratégie de raffinement n'est utilisée par défaut.

Plus de détails sur les chaînes de stratégies en général peuvent être trouvés dans le manuel de *Scotch 6.0* [51].

Annexe C

Résultats supplémentaires

C.1 Influence du coût de migration et du facteur de repartitionnement

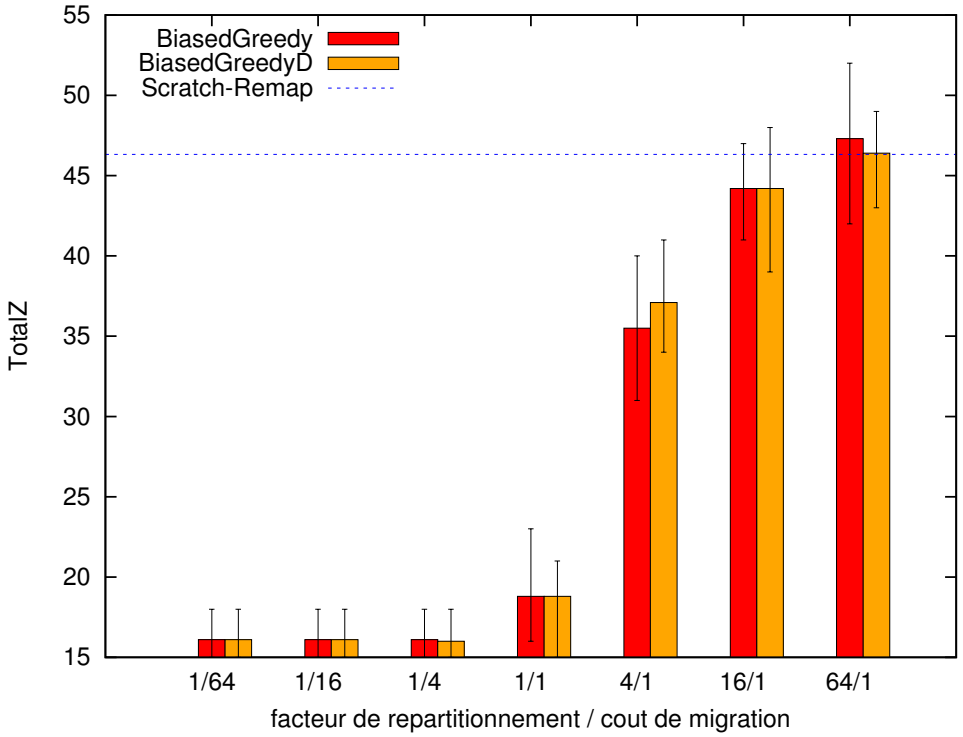


FIGURE C.1 – TOTALZ 8×12 suivant *repartmult/migcost*.

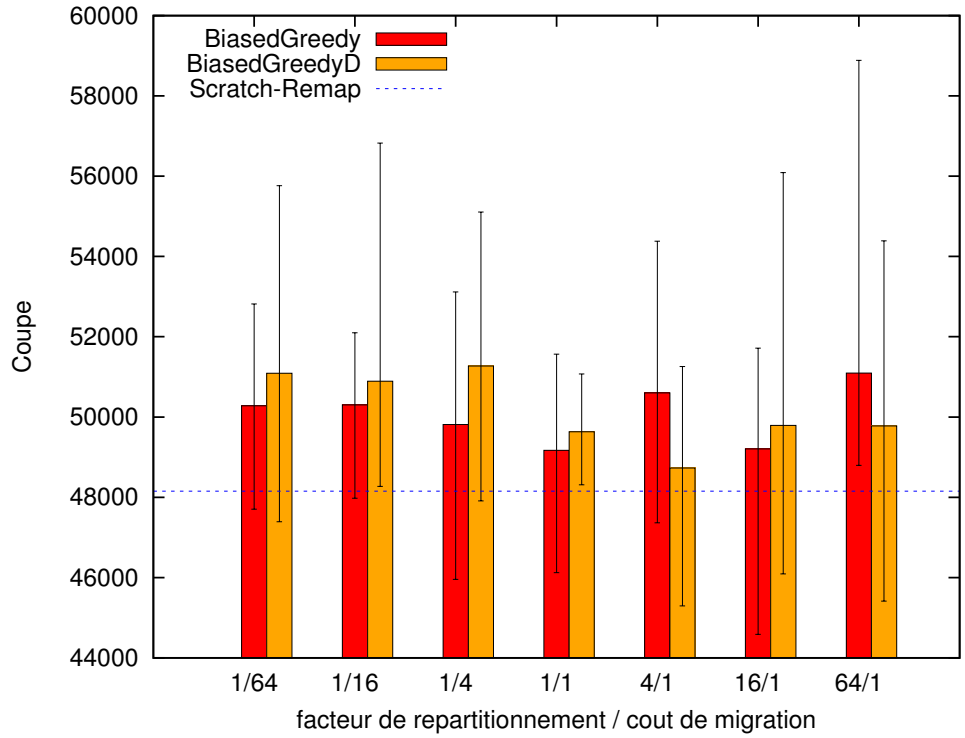


FIGURE C.2 – Coupe 8×12 suivant *repartmult*/*migcost*.

C.2 Comparaison sur des graphes complexes

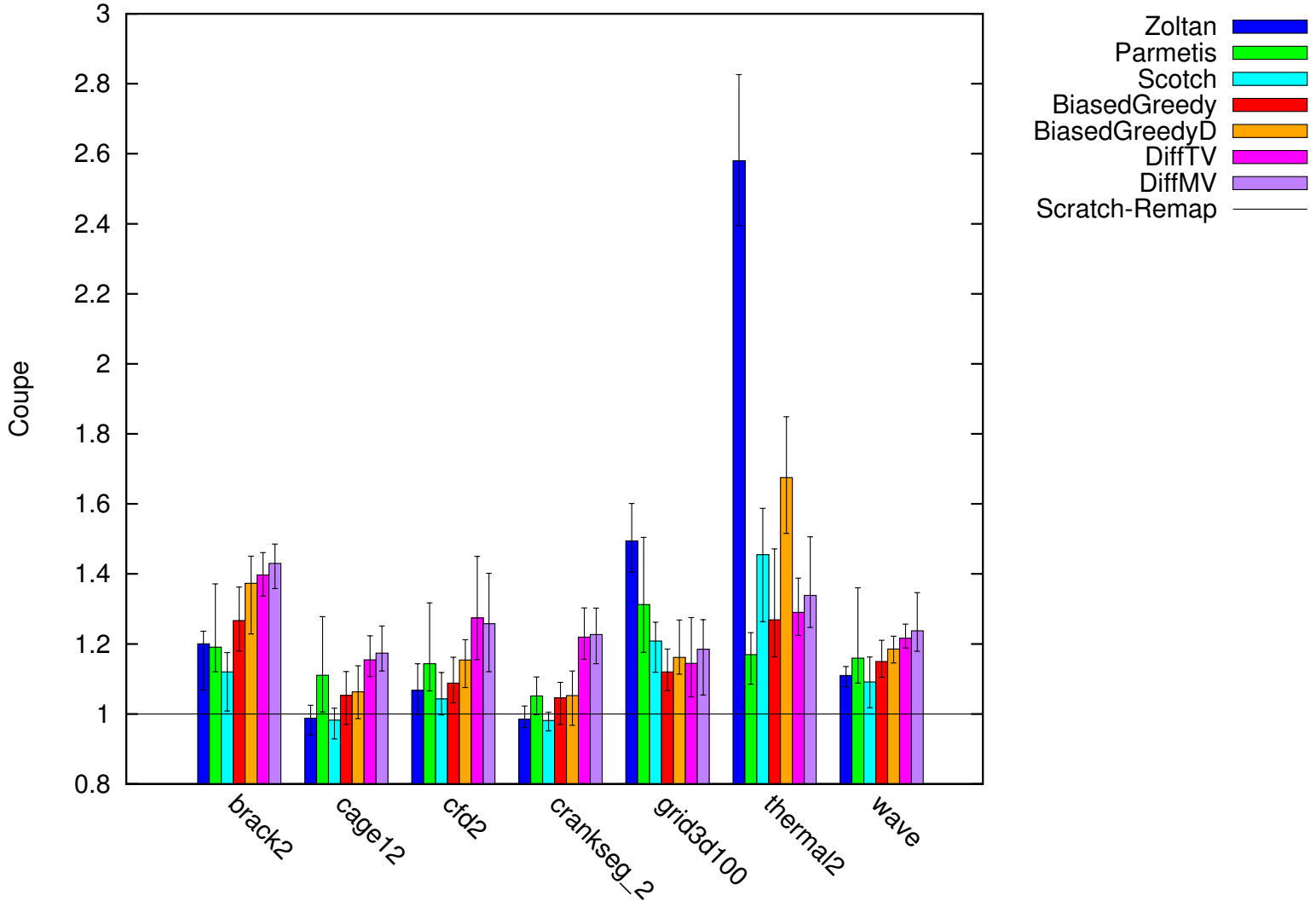


FIGURE C.3 – Coupe relative au *Scratch-Remap* pour un repartitionnement 8×10

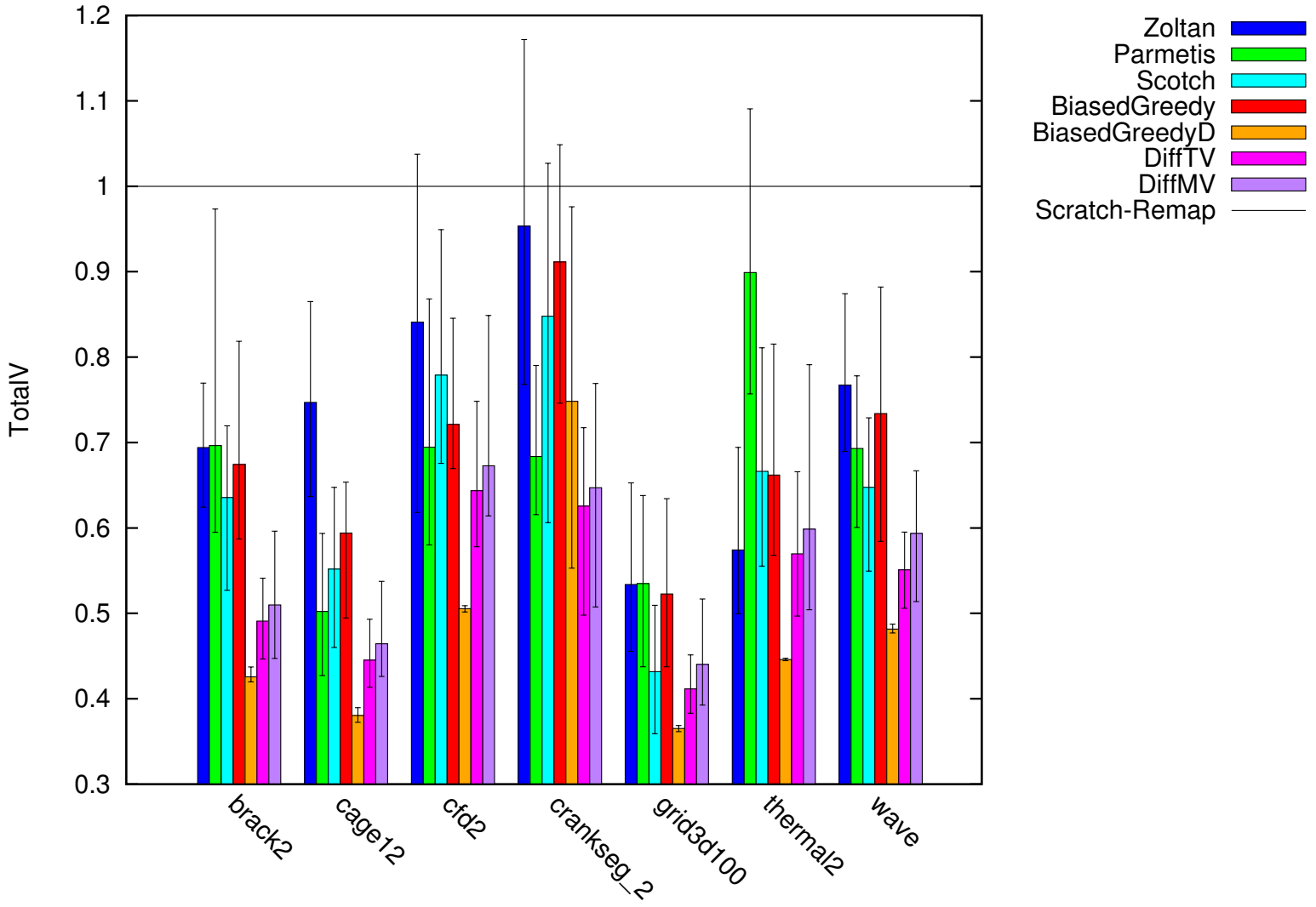


FIGURE C.4 – TOTALV relatif au *Scratch-Remap* pour un repartitionnement 8×10

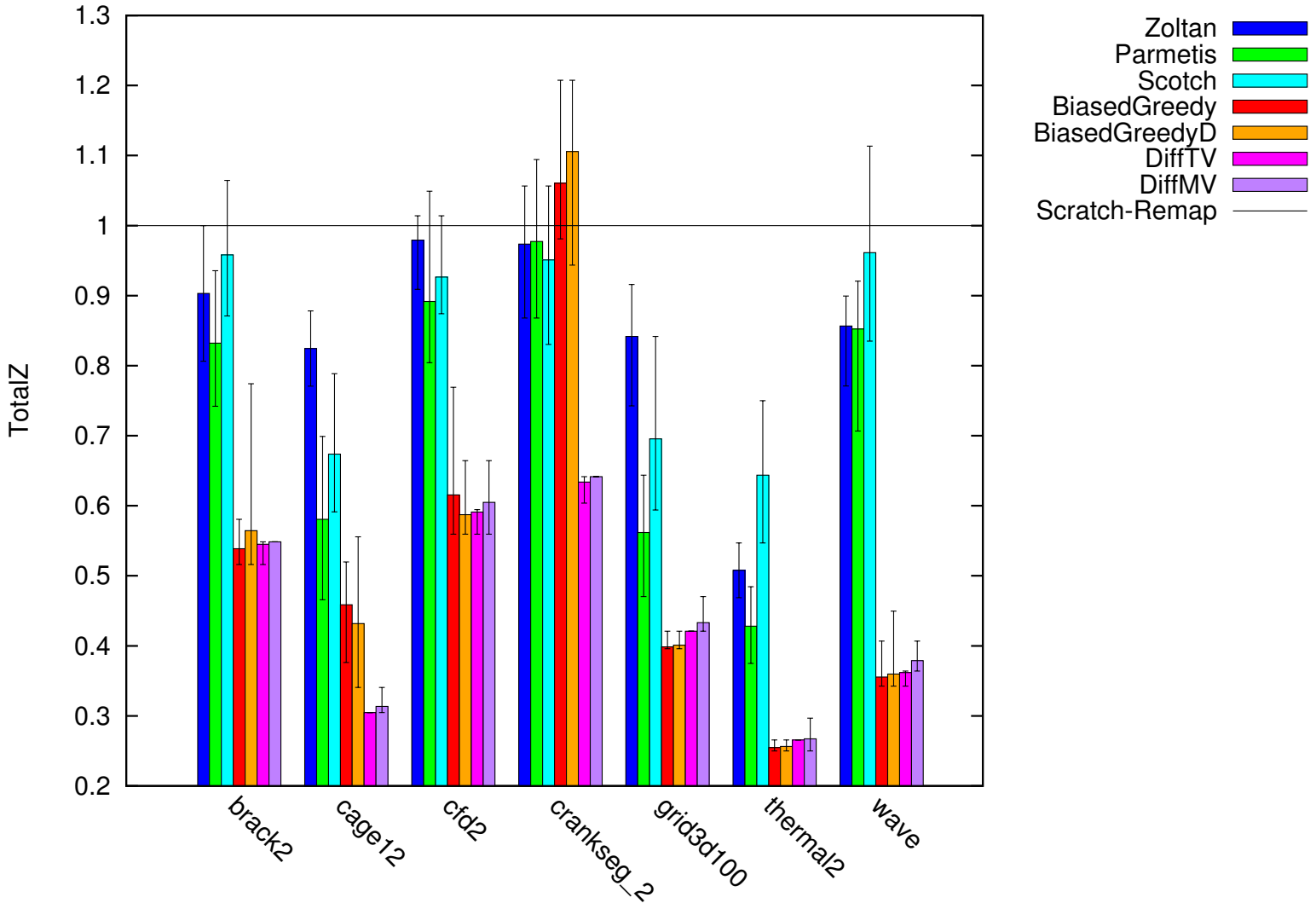


FIGURE C.5 – TOTALZ relatif au *Scratch-Remap* pour un repartitionnement 8×10

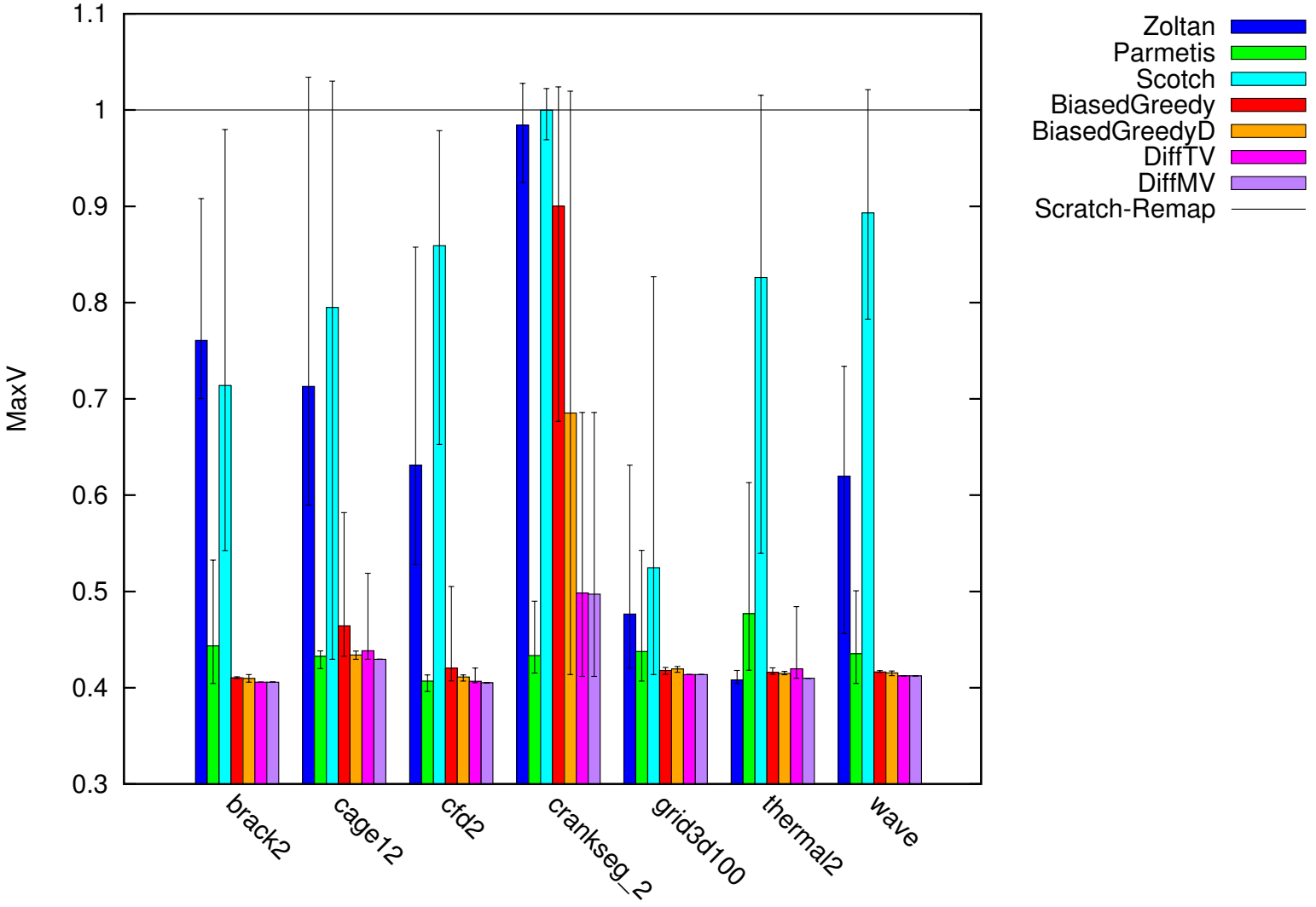


FIGURE C.6 – MAXV relatif au *Scratch-Remap* pour un repartitionnement 8×10

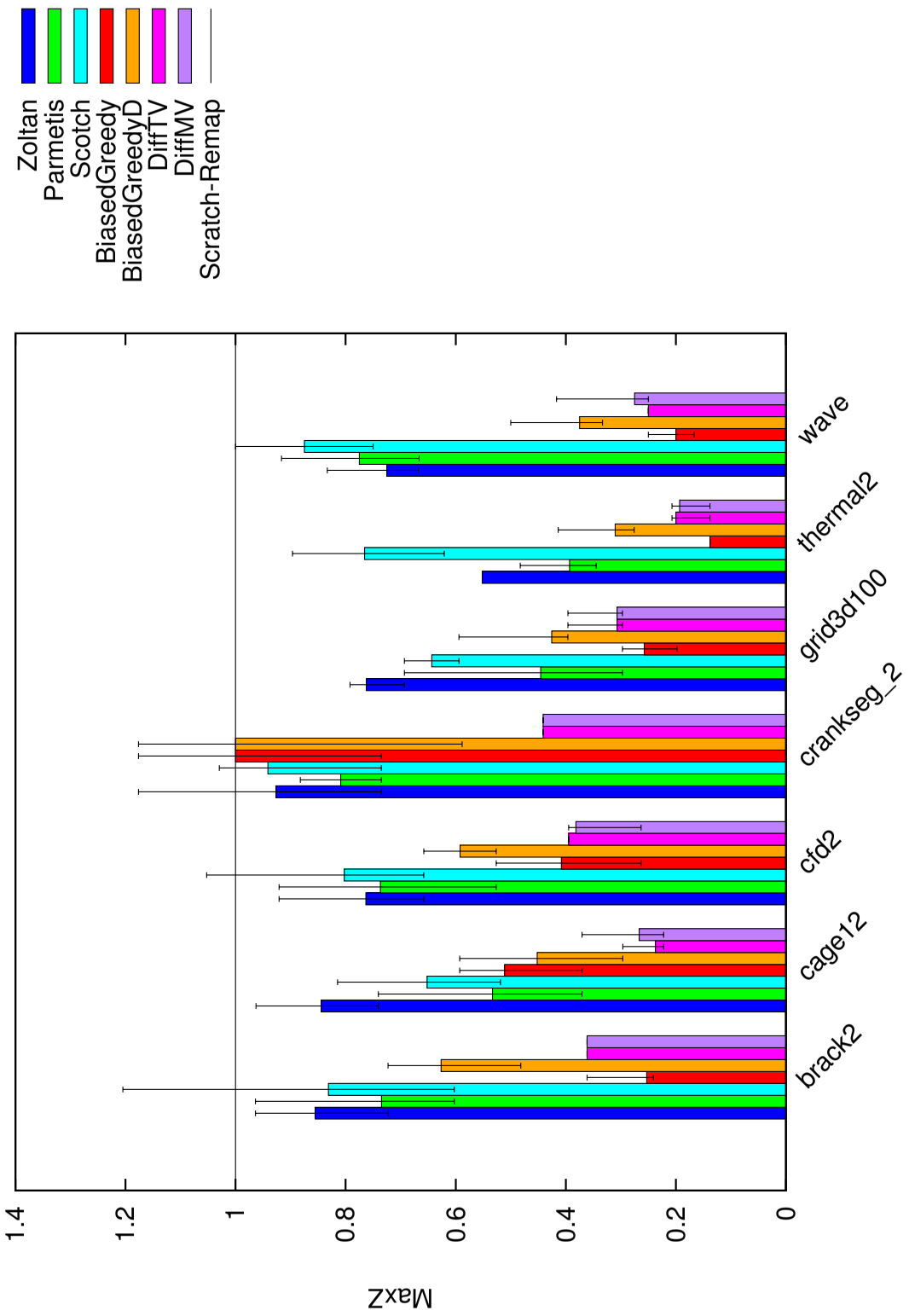


FIGURE C.7 – MAXZ relatif au *Scratch-Remap* pour un repartitionnement 8×10

Bibliographie

[1] Zoltan : Parallel partitioning, load balancing and data-management services. `http://www.cs.sandia.gov/Zoltan/Zoltan.html`.

[2] Cevdet Aykanat, B. Barla Cambazoglu, Ferit Findik et Tahsin Kurc : Adaptive decomposition and remapping algorithms for object-space-parallel direct volume rendering of unstructured grids. *J. Parallel Distrib. Comput.*, 67:77–99, janvier 2007.

[3] Cevdet Aykanat, B. Barla Cambazoglu et Bora Uçar : Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *J. Parallel Distrib. Comput.*, 68:609–625, mai 2008.

[4] Stephen T Barnard et Horst D Simon : Fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency : Practice and Experience*, 6(2):101–117, 1994.

[5] Roberto Battiti et Alan Bertossi : Differential greedy for the 0-1 equicut problem. *In in Proceedings of the DIMACS Workshop on Network Design : Connectivity and Facilities Location*, pages 3–21. American Mathematical Society, 1997.

[6] Roberto Battiti et Alan Bertossi : Greedy, prohibition, and reactive heuristics for graph partitioning. *IEEE Transactions on Computers*, 48:361–385, 1998.

[7] Andrew E Caldwell, Andrew B Kahng et Igor L Markov : Hypergraph partitioning with fixed vertices. *In Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, pages 355–359. ACM, 1999.

[8] U. V. Çatalyürek, E. G. Boman, K. D. Devine, D. Bozdağ, R. Heaphy, et L. A. Fisk : Hypergraph-based dynamic load balancing for adaptive scientific computations. *Proceedings of 21st International Parallel and Distributed Processing Sympsosium (IPDPS)*, 2007.

[9] Umit V. Çatalyürek, Erik G. Boman, Karen D. Devine, Doruk Bozdağ, Robert T. Heaphy et Lee Ann Riesen : A repartitioning hypergraph model for dynamic load balancing. *J. Parallel Distrib. Comput.*, 69(8):711–724, 2009.

[10] Cédric Chevalier et François Pellegrini : Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework. *In Euro-Par 2006 Parallel Processing*, volume 4128, pages 243–252, Dresden, septembre 2006. Springer.

[11] Jr. Ciarlet, P. et F. Lamour : On the validity of a front-oriented approach to partitioning large sparse graphs with a connectivity constraint. *Numerical Algorithms*, 12(1):193–214, 1996.

[12] G. Cybenko : Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7(2):279–301, octobre 1989.

[13] Timothy A. Davis et Yifan Hu : The university of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1 :1–1 :25, décembre 2011.

[14] Ralf Diekmann, Robert Preis, Frank Schlimbach et Chris Walshaw : Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM. *Parallel Computing*, 26(12): 1555–1581, 2000.

[15] Olivier Duchenne, Francis R. Bach, In-So Kweon et Jean Ponce : A tensor-based algorithm for high-order graph matching. *In CVPR*, pages 1980–1987. IEEE, 2009.

[16] Aurélien Esnard et Clément Vuchener : Library for Balancing Code Coupling. `http://gforge.inria.fr/projects/mpicpl/`.

[17] C. M. Fiduccia et R. M. Mattheyses : A linear-time heuristic for improving network partitions. *19th Design Automation Conference*, pages 175–181, 1982.

[18] Sébastien Fourestier : *Redistribution dynamique parallèle efficace de la charge pour les problèmes numériques de très grande taille.* Thèse de doctorat, Université Bordeaux 1, juin 2013.

[19] Sébastien Fourestier et François Pellegrini : Adaptation au repartitionnement de graphes d'une méthode d'optimisation globale par diffusion. *In Proc. RenPar'20, Saint-Malo, France*, mai 2011.

[20] Michael R. Garey et David S. Johnson : *Computers and Intractibility : A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[21] Norman E. Gibbs, William G. Poole, Jr. et Paul K. Stockmeyer : An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM Journal on Numerical Analysis*, 13(2):236–250, 1976.

[22] B. Hendrickson et R. Leland : A multilevel algorithm for partitioning graphs. *In Proceedings of 1995 ACM/IEEE conference on Supercomputing*, 1995.

[23] Bruce Hendrickson, Robert W. Leland et Rafael Van Driessche : Skewed graph partitioning. *In Eighth SIAM Conf. Parallel Processing for Scientific Computing*, 1997.

[24] Bruce Hendrickson et Tobert Leland : Chaco : Software for partitioning graphs. `http://www.sandia.gov/~bahendr/chaco.html`.

[25] Roger A. Horn et Charles R. Johnson : *Matrix Analysis.* Cambridge University Press, 1985.

[26] Y. F. Hu et R. J. Blake : An optimal dynamic load balancing algorithm. Rapport technique, Daresbury Laboratory, 1995.

[27] Saeed Iqbal et Graham F. Carey : Performance analysis of dynamic load balancing algorithms with variable number of processors. *Journal of Parallel and Distributed Computing*, 65(8):934 – 948, 2005.

[28] Saeed Iqbal et Graham F. Carey : Performance of parallel computations with dynamic processor allocation. *Engineering with Computers*, 24:135–143, 2008.

[29] Sachin Jain, Chaitanya Swamy et K. Balaji : Greedy algorithms for k-way graph partitioning. *In the 6th international conference on advanced computing*, 1998.

[30] George Karypis : hMETIS. `http://glaros.dtc.umn.edu/gkhome/metis/hmetis/overview`.

[31] George Karypis : METIS. `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`.

[32] George Karypis : PARMETIS. `http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview`.

[33] George KARYPIS, Rajat AGGARWAL, Vipin KUMAR et Shashi SHEKHAR : Multilevel hypergraph partitioning : Applications in VLSI domain. *IEEE Transactions on VLSI Systems*, 7(1):69–79, 1999.

[34] George KARYPIS et Vipin KUMAR : A fast and high quality multilevel scheme for partitioning irregular graphs. *In International Conference on Parallel Processing*, pages 113–122, 1995.

[35] George KARYPIS et Vipin KUMAR : Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998.

[36] George KARYPIS et Vipin KUMAR : A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1), 1999.

[37] George KARYPIS et Vipin KUMAR : Parallel multilevel k-way partitioning scheme for irregular graphs. *SIAM Review*, 41(2):278–300, 1999.

[38] George KARYPIS et Vipin KUMAR : Multilevel k-way hypergraph partitioning. *VLSI Design*, 11(3):285–300, 2000.

[39] B. W. KERNIGHAN et S. LIN : An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49:291–307, février 1970.

[40] Scott KIRKPATRICK, C. D. GELATT et Mario P. VECCHI : Optimization by simmulated annealing. *science*, 220(4598):671–680, mai 1983.

[41] Marius LEORDEANU et Martial HEBERT : A spectral technique for correspondence problems using pairwise constraints. *In Proceedings of the Tenth IEEE International Conference on Computer Vision - Volume 2*, ICCV '05, pages 1482–1489, Washington, DC, USA, 2005. IEEE Computer Society.

[42] Henning MEYERHENKE, Burkhard MONIEN et Stefan SCHAMBERGER : Graph partitioning and disturbed diffusion. *Parallel Comput.*, 35(10-11):544–569, octobre 2009.

[43] Katta G. MURTY : *Linear programming*. John Wiley & Sons, 1983.

[44] Leonid OLIKER et Rupak BISWAS : Efficient load balancing and data remapping for adaptive grid calculations. *In 9th ACM Symposium on Parallel Algorithms and Architectures*, pages 33–42, 1997.

[45] Leonid OLIKER et Rupak BISWAS : PLUM : parallel load balancing for adaptive unstructured meshes. *J. Parallel Distrib. Comput.*, 52:150–177, août 1998.

[46] Chao-Wei OU et Sanjay RANKA : Parallel incremental graph partitioning using linear programming. *In Proceedings Supercomputing '94*, pages 458–467, 1994.

[47] David A. PAPA et Igor L. MARKOV : Hypergraph partitioning and clustering. *In In Approximation Algorithms and Metaheuristics*, 2007.

[48] François PELLEGRINI : SCOTCH. `http://www.labri.fr/perso/pelegrin/scotch/`.

[49] François PELLEGRINI : A parallelisable multi-level banded diffusion scheme for computing balanced partitions with smooth boundaries. *In* T. Priol A.-M. KERMARREC, L. Bougé, éditeur : *Euro-Par 2007 Parallel Processing*, volume 4641 de *Lecture Notes in Computer Science*, pages 195–204, Rennes, France, août 2007. Springer.

[50] François PELLEGRINI : PT-Scotch and libPTScotch 6.0 user's guide, 2012.

[51] François PELLEGRINI : Scotch and libScotch 6.0 user's guide, 2012.

[52] J.R. PILKINGTON et S.B. BADEN : Dynamic partitioning of non-uniform structured workloads with spacefilling curves. *Parallel and Distributed Systems, IEEE Transactions on*, 7(3):288 –300, mars 1996.

[53] Alex Pothen, Horst D. Simon et Kan-Pu Liou : Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11(3):430–452, mai 1990.

[54] Hans Sagan : *Space-Filling Curves*. Springer-Verlag, 1994.

[55] Kirk Schloegel, George Karypis et Vipin Kumar : Multilevel diffusion schemes for repartitioning of adaptive meshes. *Journal of Parallel and Distributed Computing*, 47(2):109 – 124, 1997.

[56] Kirk Schloegel, George Karypis et Vipin Kumar : Parallel multilevel diffusion algorithms for repartitioning of adaptive meshes. Rapport technique, University of Minnesota, Department of Computer Science and Army HPC Center, 1997.

[57] Kirk Schloegel, George Karypis et Vipin Kumar : A unified algorithm for load-balancing adaptive scientific simulations. *In Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[58] Kirk Schloegel, George Karypis et Vipin Kumar : Wavefront diffusion and LMSR : Algorithms for dynamic repartitioning of adaptive meshes. *IEEE Trans. Parallel Distrib. Syst.*, 12(5):451–466, mai 2001.

[59] Daniel G Schweikert et Brian W Kernighan : A proper model for the partitioning of electrical circuits. *In Proceedings of the 9th Design Automation Workshop*, pages 57–62. ACM, 1972.

[60] Horst D. Simon et Shang-Hua Teng : How good is recursive bisection ? *SIAM J. Sci. Comput*, 18:1436–1445, 1995.

[61] Aleksandar Trifunovic et William J. Knottenbelt : Parkway 2.0 : A parallel multilevel hypergraph partitioning tool. *In in Proc. 19th International Symposium on Computer and Information Sciences*, pages 789–800. Springer, 2004.

[62] Brendan Vastenhouw et Rob H. Bisseling : A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.

[63] Umit V. Çatalyürek et Cevdet Aykanat : Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, juillet 1999.

[64] Ümit V. Çatalyürek et C. Aykanat : PaToH : A Multilevel Hypergraph Partitioning Tool. `http://bmi.osu.edu/~umit/software.html#patoh`, 1999.

Liste des publications

[65] Clément Vuchener et Aurélien Esnard : Repartitionnement d'un graphe de M vers N processeurs : application pour l'équilibrage dynamique de charge. *In 20ème Rencontres francophones du parallélisme (RenPar'20)*, page 8, Saint-Malo, France, 2011.

[66] Clément Vuchener et Aurélien Esnard : Dynamic Load-Balancing with Variable Number of Processors based on Graph Repartitioning. *In Proceedings of High Performance Computing (HiPC 2012)*, pages 1–9, Pune, Inde, 2012. 9 pages.

[67] Clément Vuchener et Aurélien Esnard : Équilibrage dynamique avec nombre variable de processeurs par une méthode de repartitionnement de graphe. *Technique et Science Informatiques (TSI)*, 31(8-9-10/2012):1251–1271, juin 2012.

[68] Clément Vuchener et Aurélien Esnard : Graph Repartitioning with both Dynamic Load and Dynamic Processor Allocation. *In International Conference on Parallel Computing - ParCo2013*, Advances of Parallel Computing, München, Allemagne, 2013.