THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Sébastien FOURESTIER**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Redistribution dynamique parallèle efficace de la charge pour les problèmes numériques de très grande taille

---

Soutenue le : 20 juin 2013

Après avis des rapporteurs :

Patrick Siarry	Professeur, LiSSi	
Eddy Caron	Maître de conférences HDR, ENS Lyon	

Devant la commission d'examen composée de :

Patrick Siarry	Professeur, LiSSi	Président
Eddy Caron	Maître de conférences HDR, ENS Lyon	Rapporteur
François Pellegrini ..	Professeur, Univ. Bdx 1	Directeur de Thèse
Aurélien Esnard	Maître de Conférences, Univ. Bdx 1 ...	Examinateur
Henning Meyerhenke	Karlsruhe Institute of Technology	Examinateur
Laxmikant V. Kale ...	Professeur, University of Illinois	Examinateur

– 2013 –

Redistribution dynamique parallèle efficace de la charge pour les problèmes numériques de très grande taille

Résumé :

Cette thèse traite du problème de la redistribution dynamique parallèle efficace de la charge pour les problèmes numériques de très grande taille.

Nous présentons tout d'abord un état de l'art des algorithmes permettant de résoudre les problèmes du partitionnement, du repartitionnement, du placement statique et du re-placement.

Notre première contribution vise à étudier, dans un cadre séquentiel, les caractéristiques algorithmiques souhaitables pour les méthodes parallèles de repartitionnement. Nous y présentons notre contribution à la conception d'un schéma multi-niveaux k-aire pour le calcul sequentiel de repartitionnements.

La partie la plus exigeante de cette adaptation concerne la phase d'expansion. L'une de nos contributions majeures a été de nous inspirer des méthodes d'influence afin d'adapter un algorithme de raffinement par diffusion au problème du repartitionnement.

Notre deuxième contribution porte sur la mise en œuvre de ces méthodes sur machines parallèles. L'adaptation du schéma multi-niveaux parallèle a nécessité une évolution des algorithmes et des structures de données mises en œuvre pour le partitionnement.

Ce travail est accompagné d'une analyse expérimentale, qui est rendue possible grâce à la mise en œuvre des algorithmes considérés au sein de la bibliothèque Scotch.

Mots clés :

Parallélisme, repartitionnement, redistribution dynamique, graphe, multi-niveaux, heuristiques, petascale

Discipline :

Informatique

LaBRI (UMR CNRS 5800) et Équipe-Projet INRIA Bacchus [1],
Université Bordeaux 1,
351, cours de la libération
33405 Talence Cedex, FRANCE

1. http://http://bacchus.bordeaux.inria.fr

Efficient parallel dynamic load balancing for very large numerical problems

Abstract :

This thesis concerns efficient parallel dynamic load balancing for large scale numerical problems.

First, we present a state of the art of the algorithms used to solve the partitioning, repartitioning, mapping and remapping problems.

Our first contribution, in the context of sequential processing, is to define the desirable features that parallel repartitioning tools need to possess. We present our contribution to the conception of a k-way multilevel framework for sequential repartitioning.

The most challenging part of this work regards the uncoarsening phase. One of our main contributions is the adaptation of influence methods to a global diffusion-based heuristic for the repartitioning problem.

Our second contribution is the parallelization of these methods. The adaptation of the aforementioned algorithms required some modification of the algorithms and data structure used by existing parallel partitioning routines.

This work is backed by a thorough experimental analysis, which is made possible thanks to the implementation of our algorithms into the Scotch library.

Keywords :

Parallelism, repartitioning, dynamic load balancing, graph, multilevel, heuristics, petascale

Discipline :

Informatics

LaBRI (UMR CNRS 5800) et Équipe-Projet INRIA Bacchus [2],
Université Bordeaux 1,
351, cours de la libération
33405 Talence Cedex, FRANCE

2. http://http://bacchus.bordeaux.inria.fr

$$\text{« } i\overline{h}\frac{\partial\psi}{\partial t} + \frac{\overline{h}^2}{2m}\left(\frac{\partial^2\psi}{\partial x^2} + \frac{\partial^2\psi}{\partial y^2} + \frac{\partial^2\psi}{\partial z^2}\right) - V(x,y,z,t)\psi = 0$$

Il a fallu longtemps pour que je me l'avoue, mais je suis maintenant convaincu que, lorsqu'un physicien théoricien écrit ce genre de formule au tableau, il n'est pas en train de décrire le monde, de l'expliciter : en vérité, il le (re)crée, le refabrique »
Jean-Marc Lévy-Leblond, Cahiers art et science numéro 7.

Remerciements

Je remercie en premier lieu François Pellegrini pour son encadrement de qualité, ses conseils avisés et la confiance qu'il m'a témoigné tout au long de ses années de thèse.

Je souhaite ensuite remercier Patrick Siarry et Eddy Caron pour avoir accepté d'être les rapporteurs de mon manuscrit et m'avoir fait part de leurs remarques judicieuses. Je remercie également Aurélien Esnard, Henning Meyerhenke et Laxmikant V. Kale pour avoir accepté d'être membre de mon jury de thèse.

Je remercie les membres de l'équipe du projet Charm++ de l'université d'Illinois avec qui une collaboration de qualité a pu avoir lieu, notamment : Harshitha Menon, Abhinav S Bhatele et Laxmikant V. Kale. Merci également, à Franck Capello et aux autres membres laboratoire commun INRIA-UIUC pour avoir permis la mise en place de ces échanges.

Je dédie un remerciement spécial à Nicolas, à Anne-Laure, à Josy ainsi qu'au personnel de support du Labri de l'INRIA pour avoir grandement facilité mon travail, à la fois par leur contribution professionnelle et leur bonne humeur.

Un grand merci au personnel de l'ENSEIRB-MATMECA, pour leur accompagnement lors de mes premiers enseignements. Je remercie plus spécifiquement Mohamed Mosbah qui m'a épaulé pour mes enseignements de SGBD ainsi que Denis Lapoire pour les projets pédagogiques menés au sein de la filière informatique.

Je tiens également à remercier Emmanuel Jeannot et Louis-Claude Canon pour m'avoir initié à l'utilisation de R pour la génération de graphiques, utilisation qui est rapidement devenu une habitude...

Merci à tous les membres de l'équipe Scalapplix puis ceux de l'équipe Bacchus pour leur présence au quotidien. Une pensée va à Jun-Ho Her dont j'ai partagé le bureau lorsque j'ai commencé à travailler, une autre va à Cédric Lachat pour son soutien lors des derniers mois de ma thèse, une troisième va à tous les moments partagés au cours du cheminement.

Merci aussi à tous les amis avec qui j'ai pu partager de merveilleux moments à Bordeaux ou ailleurs, ces moments qui m'ont permis de me ressourcer tout au long de ces années et ont ainsi rendu possible la réalisation d'un travail de cet ampleur.

Merci enfin à ma famille pour son soutien. Un grand merci à ma grand-mère pour les nombreuses semaines passées en sa compagnie, tantôt pour me ressourcer, tantôt pour rédiger. Un clin d'œil à Jeannot pour avoir prêté sa voiture et permis aux représentants de ma famille d'arriver à temps pour voir une partie de la soutenance quand bien même leur voiture était tombée en panne.

Pour finir, je souhaite remercier toutes les personnes que je n'ai pas citées bien qu'elles auraient mérité de l'être.

Table des matières

Introduction générale

De plus en plus de domaines nécessitent l'utilisation de simulations numériques pour résoudre leurs problèmes. Nous pouvons citer à titre d'exemple les secteurs des prévisions météorologiques, de l'énergie, de l'automobile, de l'industrie lourde, ainsi que de nombreux secteurs de la recherche tels que la physique, la biologie et l'économie. Cette liste non exhaustive a le mérite de montrer la variété des personnes ayant l'usage du calcul scientifique.

Du fait de la masse de calculs et de la quantité de données qu'elles mettent en jeu, les simulations scientifiques de grande taille nécessitent l'usage du parallélisme pour pouvoir être réalisées. Les données et les calculs qui leurs sont associés sont alors répartis sur plusieurs unités de traitement interconnectées, qui travaillent en parallèle à la résolution du problème. Les ordinateurs parallèles actuellement utilisés pour les simulations de grande taille possèdent plusieurs dizaines de milliers d'unités de traitement (appelées par la suite « processeurs », de façon générique). Il leur est donc impossible d'accéder simultanément à la même mémoire, comme c'est le cas des ordinateurs à mémoire partagée, car cela constituerait un goulot d'étranglement. Pour cette raison, ces ordinateurs ont été conçus avec une mémoire distribuée, où chaque processeur dispose de sa propre mémoire et peut échanger des informations avec ses pairs au moyen d'un réseau d'interconnexion dédié. L'utilisation efficace de tels ordinateurs parallèles suppose de répartir équitablement la charge de calcul entre les différents processeurs et de minimiser le surcoût de communication induit par la nécessité d'échanger des informations entre processeurs au cours de l'exécution du programme.

Lorsque les processus réalisant les calculs coexistent pendant toute la durée d'exécution du programme, le problème de répartition de la charge peut être modélisé sous la forme d'un problème de partitionnement d'un graphe non orienté dont les sommets représentent les calculs à effectuer et les arêtes les interdépendances entre calculs. La répartition des calculs sur p processeurs s'exprime alors comme le partitionnement du graphe de calcul (modélisant l'ensemble des calculs à réaliser) en p parties de même poids (pris comme la somme du poids des sommets qu'elles contiennent) de telle sorte que la coupe de ce graphe, c'est-à-dire la somme des poids des arêtes dont les extrémités appartiennent à deux parties différentes, soit minimale. À chaque partie est alors associé un processus exécutant les calculs attribués à celle-ci et hébergeant pour cela l'ensemble des données associées. Dans le cas d'architectures hétérogènes ou hybrides mêlant mémoires distribuée et partagée, le problème à résoudre se complexifie, car les puissances de calcul des processeurs et les coûts de communication entre ces derniers ne sont pas uniformes ; le problème de répartition de la charge est alors désigné par le terme de « placement statique ».

Dans de nombreux cas, la quantité de calcul associée à chaque sommet du graphe peut varier au cours du temps. La structure du graphe peut également changer (ajout ou suppression d'arcs ou d'arêtes, par exemple dans le cas des raffinements de maillages). L'équilibrage initial de la charge peut alors devenir inadapté et un nouveau partitionnement doit alors être calculé. Ce nouveau partitionnement devra, en plus de respecter l'équilibrage de la charge tout en minimisant la taille de la coupe, également minimiser la quantité de données devant être redistribuée

d'un processeur à un autre. Ce problème est désigné par le terme de « repartitionnement ». Étant donné que le coût de la redistribution des données est indépendant de celui des communications nécessaires à la réalisation des calculs, un coût spécifique est considéré, appelé « coût de migration ». Afin d'être applicable à des simulations (et donc des graphes) de grandes tailles, le repartitionnement de graphes doit être lui-même parallèle. Il sera alors réalisé au cours de la simulation numérique, à intervalles réguliers ou lorsque le déséquilibre constaté sera jugé trop important.

Les outils actuels de repartitionnement parallèle de graphes présentent des limitations, notamment en termes de *scalabilité*. Ainsi, l'objectif de cette thèse est, dans un premier temps, de concevoir des méthodes de repartitionnement de graphes intrinsèquement parallèles. Afin de faciliter leur développement, celui-ci a été effectué dans un cadre séquentiel, plus facile à maîtriser. Dans un second temps, il s'agira de les valider en les transposant sous forme parallèle et en les intégrant au sein d'outils utilisables par les codes numériques parallèles nécessaires à la résolution des problèmes numériques actuels.

Le réalisation d'un repartitionneur parallèle de graphes fournissant des résultats d'une qualité comparable à celle des meilleurs outils disponibles nécessite de mener des recherches de plusieurs natures :

- *étude algorithmique* : nous allons étudier les faiblesses des approches existantes et essayer d'y remédier en concevant de nouvelles méthodes ;
- *parallélisation d'applications et distribution de données* : nous allons concevoir des algorithmes parallèles devant être à la fois en mesure de s'exécuter sur un grand nombre de processeurs et de manipuler des graphes de très grandes tailles ;
- *implémentation haute performance* : les algorithmes parallèles conçus devront être *scalables*.

Notre exposé débutera par le rappel des notations et des notions usuelles, de la définition du problème de repartitionnement de graphes ainsi que d'un état de l'art.

Au chapitre 2, nous présenterons la mise en place du repartitionnement séquentiel que nous avons effectuée, notamment en adaptant les algorithmes de partitionnement existants. Nos algorithmes seront validés et leurs particularités mises en lumière par une analyse expérimentale.

La parallélisation de ces algorithmes de repartitionnement fera l'objet du chapitre 3.

Nous conclurons en décrivant les résultats obtenus, ainsi que les pistes dont l'exploration nous semble prometteuse.

Chapitre 1

Présentation du problème, état de l'art

Sommaire

1.1 Introduction

Nous présenterons au cours de ce chapitre les connaissances et procédés actuellement employés pour le partitionnement de graphes, ainsi que leur utilisation pour résoudre les problèmes de repartitionnement, de placement statique et de « re-placement » de graphes.

1.2 Définitions et notations générales

Cette section rappelle brièvement les notions de graphes et de partitionnement de graphes. Nous commencerons par quelques notions générales.

1.2.1 Vocabulaire et notations

Nous utiliserons les notations ensemblistes classiques, ainsi que le vocabulaire français qui leur est habituellement associé.

- Pour tout nombre réel x, la notation $\lceil x \rceil$ (respectivement $\lfloor x \rfloor$) désignera la partie entière supérieure (respectivement inférieure de x).
- Pour un ensemble $X = \{x_i\}_{i \in [\![1;N]\!]}$ et une fonction $f : X \to \mathbb{R}$, la notation $\bar{f}$ désigne la moyenne arithmétique de f sur X :

$$\bar{f} = \frac{1}{N} \sum_{i=1}^{N} f(x_i) = \underset{x \in X}{\text{moy}}\, f(x) \ . \tag{1.1}$$

- Une application $f : X \to Y$ est dite surjective si et seulement si :

$$\forall y \in Y, \exists x \in X, f(x) = y \ .$$

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

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

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

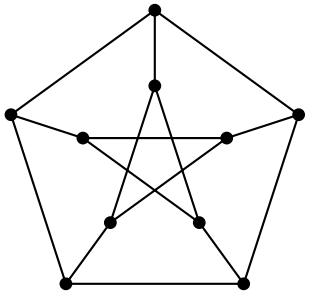
$$f = \Omega(g) \quad \Leftrightarrow \quad \exists k \in \mathbb{R}_+^*, \exists X \in \mathbb{R}, \forall x \in \mathbb{R}, (x > X \Rightarrow |f(x)| \geq k|g(x)|) \ . \tag{1.3}$$

$$f = \Theta(g) \quad \Leftrightarrow \quad f = \mathcal{O}(g) \text{ et } f = \Omega(g) \ . \tag{1.4}$$

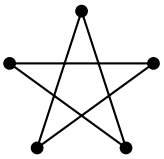
Les matrices et les notions associées seront aussi utilisées dans ce document. Par la suite, nous entendrons par matrice, s'il n'y a pas d'autres précisions, une matrice d'éléments de $\mathbb{R}$. Une matrice de taille n correspondra à la matrice carrée de taille $n \times n$.

Concernant le vocabulaire informatique, nous utiliserons les conventions usuelles concernant les tailles des objets, c'est-à-dire, par exemple, 1 kio = 1024 octets.

Nous allons maintenant définir les objets sur lesquels notre travail se focalise : les graphes.



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



(b) Un sous-graphe du graphe de Petersen (voir définition 12 de la page ci-contre).

FIGURE 1.1 – Exemples de graphes.

1.2.2 Définitions sur les graphes

Définition 2 (Graphe non orienté)
Un graphe non orienté $G = (V, E)$ est une structure composée d'un ensemble V d'éléments, appelés sommets, et d'une collection E de paires de sommets, appelées arêtes. $V(G)$ et $E(G)$ désigneront respectivement l'ensemble des sommets et la collection des arêtes de G.

Par la suite, nous noterons $n = |V|$ le nombre de sommets et $m = |E|$ le nombre d'arêtes de G.

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

Définition 3 (Graphe simple)
Un graphe simple est un graphe sans boucles ni arêtes multiples.

Par la suite, nous ne considérerons, sauf mention contraire, que des graphes simples non vides, c'est-à-dire comportant au moins un sommet, et nous parlerons donc d'ensemble d'arêtes pour $E(G)$. Un exemple de graphe est illustré en figure 1.1(a).

Définition 4 (Degré d'un sommet)
Soit u un sommet du graphe G ; le degré de u, noté $\delta(u)$, est le nombre d'arêtes de $E(G)$ incidentes à u.

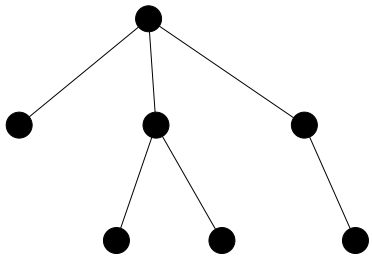
Le degré minimal (respectivement maximal) de G, noté $\delta(G)$ (respectivement $\Delta(G)$) est le minimum (respectivement maximum) des degrés de tous les sommets de G. Le degré moyen de G, noté $\bar{\delta}(G)$, est la moyenne arithmétique des degrés de tous les sommets appartenant à $V(G)$.

$$\delta(G) \stackrel{\text{def}}{=} \min_{u \in V(G)} \delta(u) \ . \tag{1.5}$$

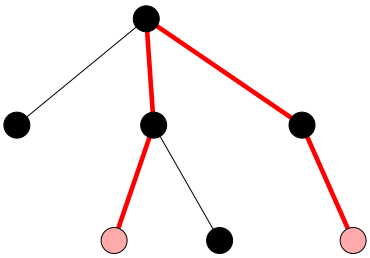
$$\Delta(G) \stackrel{\text{def}}{=} \max_{u \in V(G)} \delta(u) \ . \tag{1.6}$$

$$\bar{\delta}(G) \stackrel{\text{def}}{=} \operatorname{moy}_{u \in V(G)} \delta(u) \ . \tag{1.7}$$

Définition 5 (Chemin)
Un chemin entre deux sommets u et v est une suite $\{\{w_1, w_2\}, \{w_2, w_3\}, \ldots, \{w_{k-1}, w_k\}\}$ d'arêtes de $E(G)$ telle que $u = w_1$ et $v = w_k$.



(a) Un exemple d'arbre.



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

FIGURE 1.2 – Un exemple d'arbre de diamètre 4.

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

Définition 6 (Longueur d'un chemin)
Soit un chemin entre deux sommets u et v, défini par la suite $\{\{w_1, w_2\}, \{w_2, w_3\}, \ldots, \{w_{k-1}, w_k\}\}$. Si les arêtes sont non valuées, la longueur du chemin est définie comme le nombre d'arêtes de la suite. Si les arêtes sont valuées, la longueur du chemin est définie comme la somme des poids des arêtes de la suite.

Définition 7 (Connexité)
Un graphe G est dit connexe lorsqu'il existe un chemin entre tout couple de sommets.

Définition 8 (Composante connexe)
Un ensemble C de sommets tel que chaque couple de sommets soit relié par au moins un chemin est appelé composante connexe du graphe G.

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

Un arbre à n sommets est donc le plus petit, au sens du nombre d'arêtes, graphe connexe à n sommets. Un exemple d'arbre est représenté en figure 1.2(a).

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

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

$$d(G) \stackrel{\text{def}}{=} \max_{(u,v)\in V^2} d(u, v) \ . \tag{1.8}$$

L'arbre présenté précédemment a donc un diamètre égal à 4, comme illustré en figure 1.2(b).

Définition 12 (Sous-graphe)
Un sous-graphe $H(U, F)$ de $G(V, E)$ est un graphe tel que :

– *U est un sous-ensemble de V : $U \subseteq V$;*
– *F est la restriction de E aux couples d'éléments de $U \times U$: $F = E \cap (U \times U)$.*

H est appelé le sous-graphe de G induit par U et il est noté $G|U$. Un exemple de sous-graphe est illustré en figure 1.1(b) de la page 6.

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

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

Définition 14 (Relation d'équivalence)
Une relation d'équivalence $\mathcal{R}$ dans un ensemble E est une relation binaire réflexive, symétrique et transitive.

Définition 15 (Graphe quotient)
Soient $G(V, E)$ un graphe et $\mathcal{R}$ une relation d'équivalence pour les sommets de $V(G)$. On appelle graphe quotient le graphe $G_{|\mathcal{R}}$ défini de la façon suivante :

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

$$v' \in V(G_{|\mathcal{R}}) \iff \exists v \in V, s(v) = v' \quad ; \tag{1.9}$$

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

$$\forall \{u, v\} \in E(G), \{s(u), s(v)\} \in E(G_{|\mathcal{R}}) \Leftrightarrow \neg u \mathcal{R} v \quad . \tag{1.10}$$

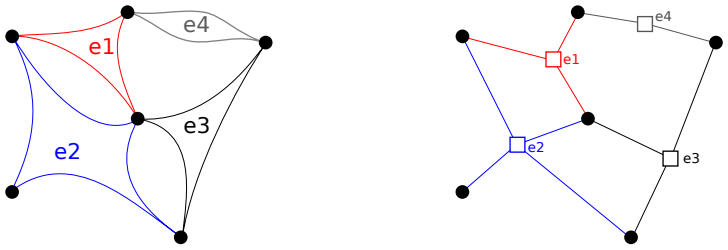
Comme nous évoquerons l'extension du problème de partitionnement de graphes aux hypergraphes, il est nécessaire de préciser leur nature.

Définition 16 (Hypergraphe)
Un hypergraphe $H = (V, \mathcal{E})$ est une structure composée d'un ensemble V d'éléments appelés sommets, et d'un ensemble $\mathcal{E}$ de sous-ensembles de V appelés hyper-arêtes.
$V(H)$ et $\mathcal{E}(H)$ désignent respectivement l'ensemble des sommets et l'ensemble des hyper-arêtes de H.

Un graphe est donc un hypergraphe dont chaque hyper-arête ne comporte que deux sommets. Un exemple d'hypergraphe est illustré en figure 1.3 de la page ci-contre.

1.3 Partitionnement de graphes

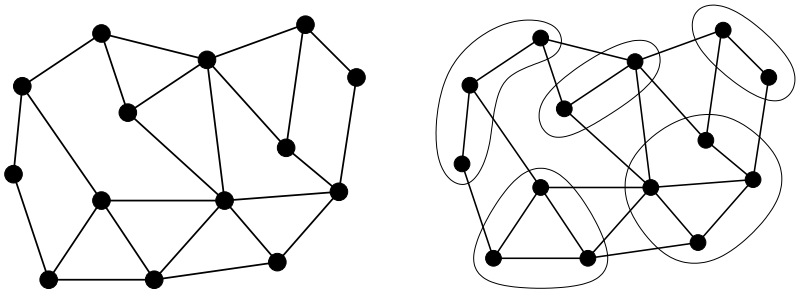
Le problème du partitionnement de graphes a fait l'objet d'une littérature conséquente dont il convient de rappeler les grandes lignes, avant d'aborder le problème plus complexe du repartitionnement de graphes.



(a) Un exemple d'hypergraphe.

(b) Une autre représentation du même hypergraphe.

FIGURE 1.3 – Deux représentations d'un hypergraphe.



(a) Un graphe G.

(b) Une partition Π de G.

FIGURE 1.4 – Partition d'un graphe.

1.3.1 Définitions

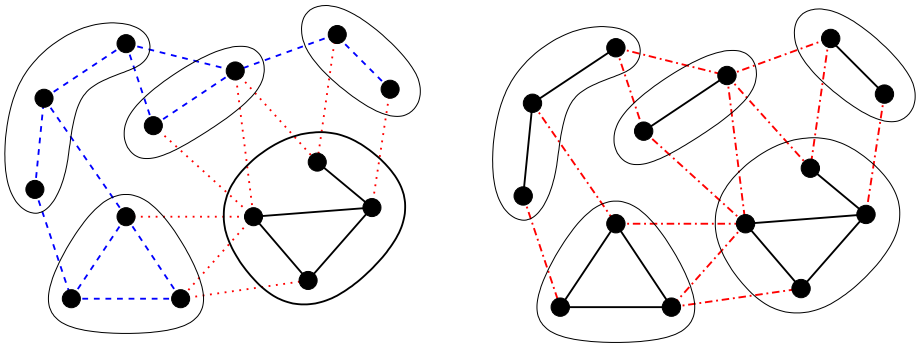
Définition 17 (Partitions de graphes)
Une partition Π *de* V *est une famille* $(V_i)_{i\in[\![1;k]\!]}$ *de* k *sous-ensembles non vides disjoints de* V *telle que l'union de toutes ces parties soit* V*.*

Nous noterons, pour tout sommet v de G, $\pi(v)$, la partie de Π contenant v et $\mathcal{P}(G)$, l'ensemble de toutes les partitions de $V(G)$. Un exemple de partition d'un graphe G est fourni en figure 1.4. Nous pouvons remarquer que le nombre $|\mathcal{P}(G)|$ de k-partitions du graphe G est asymptotiquement minoré par $\Omega(k^n)$. Trouver la meilleure partition satisfaisant un certain critère en parcourant l'ensemble des partitions de G est donc matériellement impossible pour la plupart des graphes. En outre, il a été démontré que ce problème est NP-difficile dans le cas général [GJ79].

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

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

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



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

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

FIGURE 1.5 – Les trois différents types d'arêtes induits par une partition.

3. $E_E(V_i)$ l'ensemble des arêtes externes associées à V_i, c'est-à-dire l'ensemble des arêtes n'ayant aucune extrémité dans V_i.

Pour toute partie V_i, la famille $(E_I(V_i), E_F(V_i), E_E(V_i))$ est une partition de l'ensemble des arêtes $E(G)$. Un exemple illustratif de ces ensembles est visible en figure 1.5(a).

Par extension, on définit :

$$E_I(\Pi) \stackrel{\text{def}}{=} \bigcup_{\pi\in\Pi} E_I(\pi) \ ; \tag{1.11}$$

$$E_E(\Pi) \stackrel{\text{def}}{=} E(G) - E_I(\Pi) = \bigcup_{\pi\in\Pi} E_F(\pi) \ , \tag{1.12}$$

les ensembles qui représentent respectivement les arêtes internes et externes à l'ensemble des parties de la partition. L'ensemble des arêtes externes $E_E(\Pi)$ est très souvent noté $S(\Pi)$ et est appelé séparateur du graphe G pour la partition Π. Ces deux ensembles sont illustrés en figure 1.5(b). Dans la pratique, un tel partitionnement est appelé *partitionnement arête*, car les interfaces entre les différents ensembles de sommets correspondent à des ensembles d'arêtes.

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

- le poids de chaque partie est identique, dans la mesure du possible ;
- le poids de l'interface entre les parties est le plus petit possible.

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

Définition 18 (Problème du k-partitionnement de graphes)
Le problème de k-partitionnement d'un graphe non orienté $G = (V, E)$ à arêtes et sommets valués dans $\mathbb{R}$, s'énonce de la manière suivante :

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

1. *le poids de chaque partie V_i soit égal, à la tolérance τ près, au poids moyen d'une partie w_{moy} :*

$$\forall i \in [\![1; k]\!], |w_v(\pi_i) - w_{moy}| < \tau \ ;$$

2. la coupe $\sum_{\forall i \neq j, x=(v,v') \in V_i \times V_j} w_E(x)$ *soit minimale.*

Définition 19 (Bipartitionnement)
Un 2*-partitionnement est appelé bipartitionnement.*

Le problème de partitionnement est un problème de recherche d'un optimum parmi l'ensemble des partitionnements du graphe G, qui a été démontré comme étant NP-Complet [GJS76, Pap76]. Trouver la meilleure solution dans le cas d'un graphe G quelconque a ainsi une complexité proportionnelle au nombre de partitions de G, c'est-à-dire en $\Omega(k^n)$. Notons par ailleurs qu'il est NP-difficile de vérifier qu'un partitionnement est le meilleur.

Alors que l'utilisation d'un algorithme exact risque d'être coûteuse, les heuristiques calculent un partitionnement acceptable en un temps polynomial. La qualité du résultat obtenu par l'utilisation de ces dernières n'est généralement pas mesurable puisque l'optimum n'est pas connu. Il existe plusieurs familles d'heuristiques, dont une présentation plus complète et étendue aux hypergraphes est disponible dans la thèse de François Pellegrini [Pel95].

Nous allons maintenant présenter les approches couramment utilisées pour résoudre le problème du partitionnement de graphes.

1.3.2 Les algorithmes exacts

Le problème de partitionnement étant NP-complet, la recherche d'une solution optimale dans le cas général nécessite l'exploration systématique de l'espace des solutions. La manière habituelle de réaliser cette exploration s'effectue à l'aide d'un arbre de recherche dont les nœuds correspondent à des sous-ensembles de solutions.

Les méthodes de type « séparation et évaluation » (« *branch and bound* ») permettent d'éviter de parcourir toutes les branches, en « élaguant » les sous-arbres menant à des solutions toutes moins bonnes qu'une solution donnée. Plusieurs variantes existent, selon que l'exploration de l'arbre est réalisée en profondeur, par niveaux ou par ordre croissant des valeurs des fils non encore traités (le parcours de l'arbre est alors analogue à celui effectué par un algorithme de type A* [ST85, Sin87]). Bien que la complexité de cet algorithme soit dans le pire des cas identique à celle d'une énumération explicite, il est en moyenne plus efficace.

Une autre approche est celle de la programmation dynamique. Il s'agit de représenter le problème comme la minimisation avec contraintes d'une fonction de coût quadratique en variables $\{0, 1\}$ [HG86, RH89].

Dans certains cas, le problème de partitionnement peut être réduit à un problème polynomial [Lo84, Sto77].

1.3.3 L'approche spectrale

Définition 20 (Matrice d'adjacence)
La matrice d'adjacence d'un graphe G à n sommets est la matrice $A = (a_{ij})_{(i,j) \in [\![1;n]\!]^2}$ définie par :

$$\forall (i,j) \in [\![1;n]\!]^2, a_{ij} = \begin{cases} 0 & si\ \{i,j\} \notin E(G)\ \ ; \\ 1 & si\ \{i,j\} \in E(G)\ \ . \end{cases} \tag{1.13}$$

Définition 21 (Matrice des degrés)
La matrice D des degrés associée au graphe G est la matrice de taille $|V| = n$ suivante :

$$\forall (i,j) \in [\![1;n]\!]^2, a_{ij} = \begin{cases} \delta(v_i) & si\ i = j \quad ; \\ 0 & sinon \quad . \end{cases} \tag{1.14}$$

La matrice D est donc la matrice diagonale ayant pour termes diagonaux les degrés des sommets.

Définition 22 (Matrice de Laplace)
La matrice de Laplace Q du graphe G est définie par $Q = D - A$, où A est la matrice d'adjacence de G et D la matrice des degrés.

L'approche spectrale du partitionnement de graphe consiste à rechercher les valeurs propres de la matrice de Laplace Q associée au graphe G. La matrice Q étant semi-définie positive, les valeurs propres de Q peuvent donc être ordonnées de la façon suivante : $\lambda_1 = 0 \leq \lambda_2 \leq \ldots \leq \lambda_n$. Il est démontré que la multiplicité de la première valeur propre, égale à 0, correspond au nombre de composantes connexes du graphe G. Si G est connexe, la deuxième plus petite valeur propre λ_2 est strictement positive. Le vecteur propre X_2 associé à la valeur propre λ_2, souvent appelé *vecteur de Fiedler*, a été intensivement étudié par Fiedler [Fie73, Fie75]. Une de ses applications est la possiblité d'ordonner les sommets de G sur la droite des réels en associant à chaque sommet v_i une position x_i correspondant à sa $i^{\text{è}}$ composante du vecteur de Fiedler X_2. Le résultat de Hall [Hal70], qui montre que la solution optimale du « placement » des sommets sur une droite est donné par cette deuxième plus petite valeur propre λ_2, implique que si deux sommets v_i et v_j sont connectés par une arête de $E(G)$, la distance $|x_i - x_j|$ est petite. Les sommets fortement connectés sont donc proches dans l'ordonnancement des sommets. Nous pouvons ainsi déduire une partition $\Pi = (P_0, P_1)$ du graphe G en choisissant un réel ρ et en posant $P_0 = \{v_i | x_i \leq \rho\}$ et $P_1 = \{v_i | x_i > \rho\}$. Dans [PSL90], Pothen *et al.* utilisent la valeur médiane x_m comme valeur de ρ pour effectuer le bipartitionnement récursif d'un graphe.

Bien que cette méthode permette d'obtenir une partition de bonne qualité, elle est très coûteuse en termes de calculs [BS94].

1.3.4 L'approche combinatoire

La spécificité de cette approche consiste à travailler directement sur la structure du graphe. L'idée générale consiste à explorer une partie de l'ensemble $\mathcal{P}(G)$ afin d'y trouver le meilleur candidat qui résolve notre problème.

1.3.4.1 Les algorithmes itératifs d'optimisation

Les algorithmes itératifs d'optimisation fonctionnent en partant d'une partition $\Pi_0 \in \mathcal{P}(G)$ de G valide, et se déplacent dans l'espace des solutions en sélectionnant le voisin le plus à même de réduire la coupe de la partition. L'algorithme s'arrête lorsqu'aucun des voisins n'est satisfaisant. Ces algorithmes convergent donc vers l'optimum local accessible depuis la partition initiale Π_0.

Bien que la convergence ne soit que locale et dépende du point de départ dans $\mathcal{P}(G)$, ces algorithmes sont très populaires, les résultats produits pouvant être améliorés en effectuant plusieurs exécutions à partir de partitions initiales différentes ou en utilisant le schéma multi-niveaux qui sera présenté plus loin. Parmi les algorithmes de cette catégorie, nous pouvons citer l'algorithme de Kernighan-Lin, ainsi que l'algorithme de Fiduccia-Mattheyses.

Dans leur version initiale, ces algorithmes ne s'appliquaient qu'au bipartitionnement de graphes non valués. Bien que ces derniers aient été étendus, par la suite, au problème de k-partitionnement, nous les présenterons dans le contexte du bipartitionnement afin d'en faciliter la compréhension. Ces algorithmes ne permettant d'optimiser qu'un seul critère à la fois, ils sont utilisés dans le cadre du problème de partitionnement afin d'optimiser la coupe à une tolérance d'équilibrage de la charge fixée.

L'algorithme de Kernighan-Lin Cet algorithme [KL70] s'appuie sur des échanges de paires de sommets entre les différentes parties pour parcourir un sous-ensemble des solutions. Ainsi, au voisinage de $\mathcal{P}(G)$, n'importe quelle paire de sommets $(u, v) \in V^2$, telle que $\Pi(u) \neq \Pi(v)$, peut être échangée.

L'algorithme de Kernighan-Lin fonctionne par *passes*. Il effectue donc plusieurs itérations d'une boucle externe, comme nous pouvons le voir à la ligne 3 de l'algorithme 1 de la page suivante. Le *gain* associé à un échange correspond à la différence entre la valeur de la coupe avant l'échange et celle obtenue après celui-ci. Il représente la variation de la qualité du partitionnement suite à l'échange de sommets, une valeur de gain positive correspondant à une amélioration du partitionnement.

La nature itérative de cette méthode est due à la présence d'une boucle externe. Le cœur de l'algorithme est défini par la séquence suivante : la réalisation du choix du meilleur échange possible parmi ceux qui sont disponibles, puis le marquage des deux sommets déplacés comme dorénavant non échangeables. La coupe de la solution est ensuite mise à jour et le mouvement qui vient d'être effectué est mémorisé. Lorsqu'il n'y a plus de mouvements possibles, l'historique des valeurs de coupe est parcouru à la recherche de la meilleure valeur, c'est-à-dire de la valeur maximale de la somme des gains. L'état du graphe est ensuite rechargé à partir de la configuration donnant la valeur de coupe optimale et le marquage des sommets est réinitialisé. Ce processus est itéré jusqu'à ce qu'il ne soit plus possible d'améliorer la qualité de la partition au cours d'une passe.

Nous remarquons que, lorsque tous les mouvements possibles sont effectués dans l'ordre donné par les gains, il est possible de sortir des extrema locaux, car même les mouvements dégradant temporairement la qualité de la partition sont pris en compte si ce sont les seuls disponibles (à la ligne 7 de l'algorithme, le gain peut être négatif). Néanmoins, le choix du mouvement à effectuer lorsque plusieurs mouvements de même gain sont disponibles peut avoir d'importantes conséquences sur la solution finale obtenue. C'est pourquoi plusieurs exécutions de l'algorithme de Kernighan-Lin sont en général effectuées, puis la meilleure solution obtenue est conservée.

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

En pratique, l'algorithme de Kernighan-Lin est utilisé pour raffiner une solution existante ; son exécution est alors bien moins coûteuse que la complexité théorique présentée ci-dessus.

Le fait que l'algorithme de Kernighan-Lin procède par échanges de sommets permet de conserver à chaque échange l'équilibrage dans le cas de sommets non valués. Pour garantir l'équilibrage de la charge lorsque les sommets sont valués, il est nécessaire de prendre en compte des critères supplémentaires ; c'est ce que permet notamment l'algorithme de Fiduccia-Mattheyses.

Algorithme 1: Algorithme de Kernighan-Lin.

Entrée : G : graphe
Sorties : Π : partition de G

```
1  i : rang de l'itération interne
2  S_i : somme des gains jusqu'au rang i
3  répéter                                              ▷ Passe sur le graphe G
4      i ← 0                                            ▷ Initialisation des variables pour une passe
5      S_i ← 0
6      tant que il existe un échange de deux sommets non marqués faire   ▷ Boucle interne
       de l'algorithme Kernighan-Lin
7          Faire le meilleur échange possible
8          Marquer les deux sommets déplacés
9          Enregistrer le gain g_i de l'échange
10         S_{i+1} ← S_i + g_i
11         i ← i + 1
12     Trouver x telle que la somme partielle S_x des gains soit maximale
13     si S_x < 0 alors
14         Annuler tous les échanges effectués
15     sinon
16         Annuler les mouvements de x à i
17 jusqu'à S_x < 0;
```

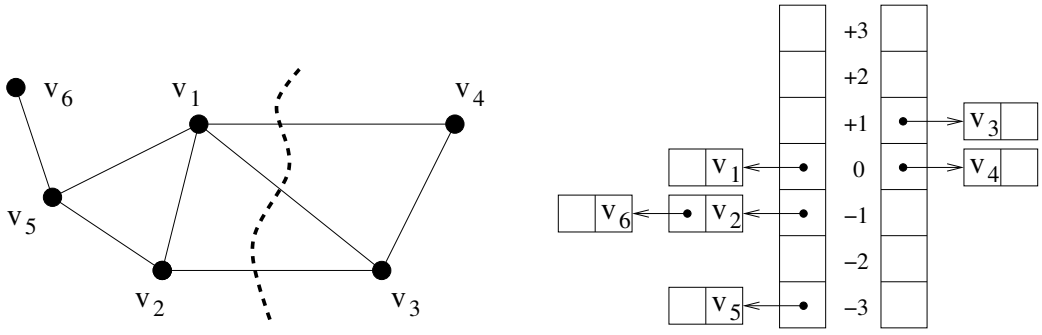
L'algorithme de Fiduccia-Mattheyses Cet algorithme [FM82] correspond à une amélioration en temps quasi-linéaire de l'algorithme de Kernighan-Lin présenté précédemment.

Contrairement à l'algorithme de Kernighan-Lin, l'algorithme de Fiduccia-Mattheyses réalise des mouvements de sommets d'une partie vers une autre, plutôt que des échanges. Il peut donc déséquilibrer la partition et c'est pour éviter cela que seuls les mouvements qui permettent de rester dans une tolérance prédéfinie pour l'équilibre sont réalisables.

L'algorithme maintient pour chaque sommet une valeur de gain, qui représente la variation de la valeur de la coupe si le sommet est migré vers l'autre partie. Les sommets sont classés selon leur gain et un tableau de listes chaînées de sommets, indexé par les gains, est utilisé ; chaque liste contenant les sommets de gain correspondant à l'indice de la case du tableau qui la contient [3]. Ce tableau est borné par une valeur maximale et une valeur minimale du gain (souvent l'opposée de la valeur maximale). En pratique, un tableau de gain est utilisé pour chaque partie, comme cela est illustré en figure 1.6 de la page suivante.

L'algorithme de Fiduccia-Mattheyses est présenté dans l'algorithme 2 de la page 16. Il consiste à sélectionner un sommet v associé au meilleur gain possible et respectant la contrainte d'équilibrage de la charge, à effectuer le mouvement, à marquer v comme étant déplacé et à mettre à jour les gains de ses voisins qui n'ont pas été déjà déplacés. La position de ces voisins dans la table des gains est ensuite mise à jour et le procédé réitéré. Nous constatons que la principale différence avec l'algorithme de Kernighan-Lin tient à la capacité de l'algorithme

3. L'extension de cet algorithme au partitionnement k-aire s'effectue en prenant en compte dans la table, plutôt que le coût de migration d'un sommet dans l'autre partie, toutes les migrations possibles de chaque sommet vers des parties voisines (c'est-à-dire des parties dans lesquelles le sommet a des voisins). Pour un sommet donné, sont alors ajoutées dans la table autant d'entrées que ce dernier aura de parties voisines vers lesquelles migrer.



(a) Bipartition courante d'un graphe non pondéré.

(b) Listes et tableaux de gains correspondant à cette bipartition.

FIGURE 1.6 – Structures de données utilisées par l'heuristique de Fiduccia-Mattheyses.

de Fiduccia-Mattheyses à trouver le sommet de plus grand gain en temps quasi-constant. Une étude plus complète de cette caractéristique est disponible dans la thèse de François PELLEGRINI [Pel95, page 91]. Ce temps quasi-constant est rendu possible grâce à la structure d'ordonnancement des gains, qui associe, à chaque valeur autorisée pour les gains, la liste des sommets correspondant à ce gain. Le temps nécessaire pour sélectionner un sommet de meilleur gain est donc proportionnel au nombre de valeurs autorisées, qui est une constante lors du déroulement de l'algorithme. La mise à jour de cette structure étant effectuée uniquement pour les sommets non marqués, elle devient de moins en moins coûteuse lorsque le nombre de boucles effectuées augmente. La complexité en espace de l'algorithme est en $\Theta(n+m)$ et celle en temps est aussi en $\Theta(n+m)$ [Pel95, pages 92–93].

Comme pour l'algorithme de Kernighan-Lin, nous noterons qu'en général plusieurs sommets correspondent au gain maximal et que le choix d'un sommet peut grandement influencer la solution finale obtenue par l'algorithme [HHK97, Kri84]. Il est difficile de sélectionner le *bon* sommet et c'est pour cela que, comme pour l'algorithme de Kernighan-Lin, de nombreuses implantations de l'algorithme de Fiduccia-Mattheyses effectuent plusieurs exécutions et conservent le meilleur résultat obtenu.

Le principal défaut de ces approches itératives concerne leur vision strictement locale du problème, ce qui implique que le résultat ne peut être au mieux qu'un optimum local, fortement dépendant de la solution initiale. D'autres approches d'algorithmes d'optimisation ayant une vision plus globale peuvent être utilisées.

1.3.4.2 Les algorithmes globaux d'optimisation

Ces approches consistent à considérer le partitionnement dans sa globalité. C'est notamment le cas des algorithmes de grossissement de bulles, des algorithmes évolutionnaires, ou encore des méthodes de type recuit simulé. L'inconvénient de ces approches est que la taille de l'espace des solutions augmente exponentiellement par rapport à la taille du problème, ce qui rend son exploration très coûteuse.

Les algorithmes de grossissement de bulles Les algorithmes de grossissement de bulles [MMS06] font partie de la classe des algorithmes de diffusion. Contrairement aux algorithmes d'optimisation locale présentés précédemment, comme ceux de Kernighan-Lin ou de Fiduccia-Mattheyses, ces algorithmes ne nécessitent pas la connaissance d'une partition de départ.

Algorithme 2: Algorithme de Fiduccia-Mattheyses.

Entrées : G : graphe
Sorties : Π : bipartition de G

répéter ▷ Passe sur le graphe G
  $i \leftarrow 0$ ▷ Initialisation des variables pour une passe
  $S_i \leftarrow 0$
  Calculer les gains des sommets
  Ordonner les gains des sommets
  tant que *Il existe un sommet à déplacer* **faire** ▷ Boucle interne
    Sélectionner le sommet v correspondant au meilleur mouvement possible et respectant la contrainte d'équilibrage de la charge
    $\overline{\pi(v)}$ *est le complémentaire de* $\pi(v)$ *dans* Π
    (Pour la version k-aire, $\overline{\pi(v)}$ *correspondrait à la meilleure partie voisine)*
    Déplacer v de $\pi(v)$ vers $\overline{\pi(v)}$
    Marquer v
    pour *tous les voisins non marqués* u *de* v *dans* G **faire**
      Calculer le gain de u
    Ordonner les gains des sommets non marqués
    $g_i \leftarrow$ le gain du déplacement
    $S_{i+1} \leftarrow S_i + g_i$
    $i \leftarrow i + 1$
  Calculer la meilleure somme partielle S_x des gains
  si $S_x < 0$ **alors**
    Annuler tous les déplacements effectués
  sinon
    Annuler les mouvements de x à i
jusqu'à $S_x < 0$;

Dans les versions les plus simples de cette approche, comme l'algorithme de Farhat [Fah88] ou encore la méthode du *greedy graph growing* [KK95b], les graines sont traitées une par une et les parties sont construites séquentiellement selon un parcours en largeur. Étant donné que ce type d'approche ne tend pas à produire des parties de formes optimales, surtout pour les dernières parties considérées, ces méthodes sont appelées plusieurs fois sur des graines choisies aléatoirement et la meilleure partition est conservée.

Dans les versions plus évoluées de ce type d'algorithme, tels que les algorithmes de grossissement à bulles, les parties sont prises en compte de manière simultanée [DPSW98, MS05]. Leur fonctionnement est présenté dans l'algorithme 3. L'étape principale se situe à la ligne 4 et consiste à faire grossir les bulles depuis leur centre, c'est-à-dire les points d'où la diffusion est initiée — appelés « graines », jusqu'à ce qu'elles se stabilisent, leur union recouvrant alors tous les sommets du graphe. La nouvelle partition est définie en prenant les bulles comme parties et les centres de ces bulles deviennent les nouvelles « graines ». Ce procédé est réalisé jusqu'à ce que le mouvement des centres devienne suffisamment faible. Le grossissement des bulles se fera alors avec un minimum de contact avec les autres bulles, minimisant ainsi la distance maximale des sommets d'une partie au centre de leur bulle. En aboutissant à des formes de parties aussi convexes que possible, cet algorithme optimisera la coupe.

Algorithme 3: Algorithmes de grossissement de bulles.

Entrée : G : graphe
Sorties : Π : partition de G

1 $\mathcal{C}$: ensemble des centres des bulles
2 $\mathcal{C} \leftarrow k$ sommets choisis aléatoirement
3 **répéter**
4 Faire grossir les bulles depuis leur centre $\mathcal{C}$ en réalisant un parcours en largeur : obtention de Π
5 Mettre à jour les centres $\mathcal{C}$ des parties de Π
6 **jusqu'à** *(la distance sur laquelle les centres sont déplacés soit inférieure à un certain seuil)*;

Les résultats obtenus par cette approche sont de bonne qualité [MS06], mais l'algorithme est lent.

Diffusion Cet algorithme [Pel07] s'inspire du mythe du *tonneau des Danaïdes*. Métaphoriquement, le graphe modélise un ensemble de tonneaux troués (les sommets) reliés par des tuyaux par lesquels le liquide peut s'écouler (les arêtes). Chaque partie du graphe reçoit, par unité de temps, un volume donné de liquide (la quantité de calcul à réaliser pour chaque partie) de couleur différente. Ces liquides s'écoulent librement à travers les tuyaux. Lorsque des liquides de couleur différente se rencontrent, seul le liquide dont la quantité est la plus importante reste, les autres disparaissent. Plus le poids d'une arête est élevé, plus la taille du tuyau correspondant sera importante, et donc plus la quantité de liquide qui le traversera sera élevée. Par ailleurs, à chaque unité de temps, les sommets vont perdre du liquide proportionnellement à leur poids. D'itération en itération, la frontière va se stabiliser entre les liquides de différentes couleurs et produire une frontière lissée.

Cet algorithme possède des caractéristiques proches des algorithmes de grossissement de bulles en termes de qualité et de vitesse d'exécution tout en permettant d'améliorer à la fois la

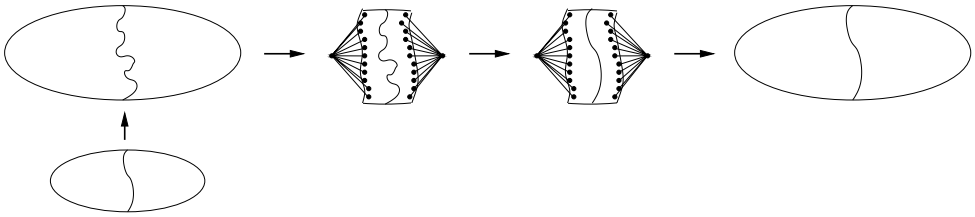


FIGURE 1.7 – Méthode de raffinement sur graphe bande dans le cas d'un bipartitionnement. À chaque étape de raffinement (lors de la phase d'expansion), la frontière du graphe contracté est prolongée sur le graphe plus fin. Un graphe bande de petite largeur est créé autour de cette frontière prolongée. Pour chaque partie, les sommets qui n'appartiennent pas à la bande sont contractés sous la forme d'un sommet ancre.
Une fois que les algorithmes de raffinement ont été appelés sur ce graphe bande, la frontière raffinée est prolongée sur le graphe plus fin.

coupe et l'équilibrage de la charge.

Graphes bandes De par leur nature locale, les algorithmes de raffinement (l'algorithme de diffusion ou un algorithme de type Fiduccia-Mattheyses) se contentent de modifier légèrement un séparateur dont la qualité est bonne sur le graphe contracté de plus gros grain. De fait, il n'est pas nécessaire d'avoir toute l'information du graphe en mémoire pour les exécuter. Ce constat amène naturellement à l'utilisation de graphes bandes définis comme la restriction du graphe plus fin à l'ensemble des sommets situés à une distance au plus d de la frontière qui a été prolongée à partir du graphe contracté. Pour chacune des parties du graphe bande, les sommets qui n'appartiennent pas à la bande sont contractés sous la forme d'un sommet ancre d'un poids égal à la somme des poids de ces derniers. Ensuite, ces sommets ancre sont reliés aux sommets de dernier niveau (ceux situés à une distance d de la frontière), comme illustré en figure 1.7.

La grande majorité des sommets du graphe qui seront pris en compte lors du raffinement se situent à, au plus, une distance 3 de la frontière. Par ailleurs, l'utilisation de l'heuristique de Fiduccia-Mattheyses sur un graphe bande de taille 3 est plus stable que sur le graphe global et produit des résultats de qualité équivalente [CP06a].

Les algorithmes évolutionnaires Les algorithmes évolutionnaires, parmi lesquels figurent les algorithmes génétiques, ont fait l'objet de nombreuses études [CP06a, BM96, CA92, Hol75, SWM00] dont certaines quant à leur usage pour résoudre le problème du partitionnement de graphes [KR03, KT98, RMR09, RMRARD02, TB91]. Le concept de base de ces algorithmes est de tenter d'imiter la nature et sa faculté d'adaptation à un problème donné, grâce à la sélection naturelle décrite par la théorie de l'évolution.

Dans le cas des algorithmes génétiques, le principe consiste en l'exploration de l'espace des solutions à l'aide d'une population d'individus. Ceux-ci vont échanger entre eux certaines de leurs caractéristiques lors d'une étape de reproduction, dans le but d'obtenir de nouveaux individus combinant les qualités de leurs parents, tout en essayant d'en diminuer les défauts. Tous les individus ne se reproduisent pas et il existe plusieurs stratégies pour sélectionner la partie reproductrice de la population. Les principales difficultés d'application de ces algorithmes tiennent, d'une part, à la multitude de paramètres à régler en fonction de la nature du problème et, d'autre part, à leurs coûts en mémoire et en temps [AZ04, BM96].

Bien qu'ils soient eux aussi intrinsèquement parallèle, notre préférence va pour les algorithmes de diffusion qui, bien que nécessitant un temps d'exécution notable, sont, en général, plus rapides.

Le recuit simulé Le recuit simulé est une heuristique introduite comme une approche générale aux problèmes d'optimisation [KGV83, Cer85]. Cette heuristique s'inspire du recuit utilisé en métallurgie, une technique selon laquelle l'alternance entre un réchauffage et un refroidissement maîtrisé permet de minimiser l'énergie du matériau. Elle s'appuie sur l'algorithme de *Metropolis* [MRR$^+$53] de la mécanique statistique. Elle consiste à optimiser une partition initiale en réalisant, à chaque itération, une modification élémentaire de la solution [BM88]. Le schéma de fonctionnement du recuit simulé est présenté par l'algorithme 4 de la page suivante. À la ligne 11, un voisin est choisi. Cette variation de la solution entraîne une variation de l'énergie du système, c'est-à-dire du coût de la solution correspondante, calculé à partir du critère que l'on cherche à optimiser. Si la variation d'énergie est négative, la nouvelle solution optimise mieux le critère choisi et est donc conservée (ligne 14) ; elle devient le nouveau point de départ. Si la variation d'énergie est positive, la nouvelle solution est de moins bonne qualité et, selon la règle de Metropolis, elle n'est prise comme nouveau point de départ qu'avec une probabilité exponentielle (ligne 17).

La probabilité de conservation d'une moins bonne solution comme point de départ est fonction de la température, elle-même fonction du nombre d'itérations effectuées et du nombre maximal d'itérations pouvant être réalisées. Deux approches sont possibles pour le calcul de la température d'une itération donnée : soit la température diminue par paliers, soit elle diminue de manière continue. À haute température, le système est libre de se déplacer dans l'espace des solutions (la probabilité de la ligne 17 est proche de 1) ; à basse température, l'amélioration de la qualité de la solution est privilégiée.

Il a été démontré par Hajek [Haj88] que l'algorithme du recuit simulé converge vers l'optimum global si et seulement si le déroulement du recuit permet à la température T de tendre vers 0 suffisamment lentement. Ceci est, en partie, responsable de la lenteur de cette approche — par ailleurs intrinsèquement séquentielle — qui n'a donc pas été utilisée avec succès dans le cas du partitionnement de graphes, contrairement à de nombreux autres domaines.

Autres approches De nombreuses autres heuristiques ont été proposées afin de trouver des solutions sous-optimales en un temps raisonnable. C'est le cas de la recherche tabou [BB99], des algorithmes de gradients aléatoires adaptatifs (GRASP) [AZ04], des méthodes de diffusion stochastique [TZ93, WRSL05], des méthodes de colonies de fourmis [KvR04, LG99] et d'autres approches [Bok81, ERS90, MLT82, MT91, Nic94, NORL86, SS12, DGRW12] issues de différents domaines, tels la théorie des graphes (recherche d'homomorphismes faibles, partitions de coupe minimale, groupement de processus), l'utilisation de l'information géométrique des graphes (méthodes inertielles et rectilinéaires), etc.

La recherche tabou, introduite par Glover [Glo89], consiste à se déplacer dans le voisinage de la solution courante, tout en gardant en mémoire les dernières solutions précédemment sélectionnées, afin d'éviter de parcourir des cycles dans l'exploration. Elle est donc assez proche du recuit simulé en termes d'exploration.

Les méthodes de colonies de fourmis consistent à imiter le comportement de nombreux insectes, comme les fourmis ou les abeilles, qui exploitent une méthode de résolution collective. En effet, les fourmis utilisent des phéromones pour marquer les différents chemins qu'elles empruntent et la concentration de ces marqueurs chimiques est d'autant plus importante que la fréquence d'utilisation est importante. Pour le k-partitionnement, l'idée est donc d'utiliser k co-

Algorithme 4: Algorithme du recuit simulé.

Entrée : G : graphe
Π : partition de G
i_{max} : nombre maximal d'itérations
C_{max} : énergie maximale permise
Sorties : Π : partition de G

Π' : partition courante de G
Π_m : meilleure partition
C : énergie (coût) de la partition Π
C' : énergie de la partition Π'
C_m : Coût de la meilleure partition
i : nombre d'itérations

$i \leftarrow 0$ ▷ Initialisation
$C \leftarrow$ `coût(`Π`)`
$(\Pi_m, C_m) \leftarrow (\Pi, C)$ ▷ Initialisation de la meilleure solution connue
tant que $i < i_{max}$ *et* $C > C_{max}$ **faire**
  $\Pi' \leftarrow$ `voisin(`Π`)` ▷ Modification élémentaire de la solution
  $C' \leftarrow$ `coût(`Π'`)`
  si $C' < C_m$ **alors** ▷ La solution est de meilleure qualité
    $(\Pi_m, C_m) \leftarrow (\Pi', C')$ ▷ Mise à jour de la meilleure solution connue
    $(\Pi, C) \leftarrow (\Pi', C')$ ▷ Mise à jour de l'état de départ
  sinon
    si `aléatoire()` $< e^{\left(\frac{C-C'}{\texttt{temp}(i, i_{max})}\right)}$ **alors**
      $(\Pi, C) \leftarrow (\Pi', C')$ ▷ Mise à jour probabiliste de l'état de départ
    $i \leftarrow i + 1$
$\Pi \leftarrow \Pi_m$ ▷ Retour de la meilleure solution obtenue

lonies de fourmis dont le but est d'accumuler la nourriture qui est distribuée sur les sommets du graphe. Ces méthodes sont différentes des méthodes de diffusion ou de bulles présentées précédemment, car l'exploration ne se fait pas de manière uniforme, mais est guidée par la répartition des colonies, de la nourriture et par les décisions des fourmis.

Toutes ces méthodes semblent pouvoir conduire à des partitions de bonne qualité, mais leur utilisation reste marginale, à cause de leur temps d'exécution et de la difficulté d'étalonner leurs paramètres correctement.

1.3.5 L'approche « diviser pour résoudre » par bipartionnement récursif

Afin de placer des graphes de plus en plus gros sur des machines parallèles de taille croissante, de nombreux auteurs se sont tournés vers l'approche « diviser pour résoudre ».

Cette approche est conceptuellement plus simple que de k-partitionner les graphes en une seule fois pour obtenir immédiatement le nombre de parties voulu. C'est donc une approximation du k-partitionnement. Elle consiste à partitionner le graphe en un nombre de parties inférieur à celui souhaité (habituellement deux, plus rarement quatre ou huit [HL93]). Ce processus est appliqué récursivement, jusqu'à ce que le nombre de parties souhaitées soit obtenu. Lorsque le graphe est partitionné récursivement en deux, le terme de *bipartitionnement récursif* est employé.

Remarquons que c'est une approche gloutonne, puisque les choix effectués ne pourront être remis en cause. C'est cependant une « bonne » approche gloutonne, parce que les choix initiaux ne sont pas très informatifs, et leurs effets peuvent éventuellement être atténués par des choix ultérieurs. Par exemple — en anticipant sur le problème du placement statique que nous abordons dans la section suivante — deux processus séparés sur deux sous-architectures différentes au début de la récursion peuvent cependant être maintenus à petite distance l'un de l'autre, si les séparations ultérieures les affectent à des sous-architectures toujours proches les unes des autres.

Les très bons résultats obtenus avec l'approche « diviser pour résoudre » ont été justifiés théoriquement par Simon et Teng [ST93] sur les graphes les plus couramment utilisés. Le résultat majeur de leur article est que, lorsqu'on place un graphe de maillage d-dimensionnel [4] de type éléments finis (à degré borné) S sur une architecture cible T par bipartitionnements récursifs, le rapport entre le nombre d'arêtes liant des processus placés sur des processeurs différents (c'est-à-dire le cardinal de l'union des coupes successives) et $|E(S)|$ appartient à $O\left(\left(\frac{|V(T)|}{|V(S)|}\right)^{\frac{1}{d}}\right)$. Ils montrent ainsi que les algorithmes parallèles associant un traitement élémentaire à chaque sommet et dont le schéma de communication suit la topologie de ces graphes sont adaptés au parallélisme massif.

En effet, pour un graphe donné, l'augmentation linéaire de la taille p de l'ordinateur cible n'induit qu'une augmentation en $\Theta(p^{\frac{1}{d}})$ de la taille des coupes (c'est-à-dire du volume de communication à échanger entre processeurs). Les programmes parallèles peuvent donc utiliser des ordinateurs plus gros en limitant la surcharge du réseau de communication. L'efficacité du programme vis-à-vis du nombre de processeurs ne dépend que de la taille des parties, qui détermine le rapport entre temps de calcul et temps de communication.

4. La preuve de Simon et Teng est fondée sur l'existence de théorèmes de séparation pour les familles de graphes étudiés. Leur résultat s'applique donc également aux graphes planaires, à genre borné, à mineur borné, et en général à tous les graphes possédant de « bons » théorèmes de séparation.

1.4 Placement statique de graphes

Comme les problèmes de partitionnement, le problème du placement statique est NP-complet dans le cas général [GJ79].

1.4.1 Définition

Définition 23 (Problème du placement statique de graphes)
Soit $S = (V_S, E_S)$ et $T = (V_T, E_T)$ deux graphes non orientés (définissant respectivement le graphe source — représentant le programme — et le graphe cible — représentant la topologie) à arêtes et sommets valués dans $\mathbb{R}$. Le problème du placement du graphe S sur le graphe T s'énonce de la manière suivante :

Trouver un couple ($\phi_{S,T}$,$\rho_{S,T}$) où $\phi_{S,T}$ est une application de V_S dans V_T, et $\rho_{S,T}$ est une application associant à toute arête $\{u, v\} \in E_S$ un chemin reliant $\phi_{S,T}(u)$ à $\phi_{S,T}(v)$ tel que :

1. *la charge $c(v_T) = \dfrac{\sum_{\substack{v_S \in V_S \\ \phi_{S,T}(v_S)=v_T}} w_S(v_S)}{w_T(v_T)}$ de chaque sommet de T soit, à la tolérance τ_1 près, la même :*
$$\forall (t_i, t_j) \in {V_T}^2, |c(t_i) - c(t_j)| < \tau_1 \ ;$$
2. *la fonction de coût de communication soit inférieure à la tolérance τ_2 :*
$$f_C(\phi_{S,T}, \rho_{S,T}) = \sum_{e_S \in E_S} w_S(e_S)|\rho_{S,T}(e_S)| < \tau_2 \ ,$$
où $|\rho_{S,T}(e_S)|$ est la longueur maximale du chemin reliant les extrémités de e_S dans T, appelée dilatation de l'arête e_S.

Le problème du k-partitionnement est donc un cas particulier du problème de placement statique où le graphe cible est un graphe complet non valué. D'un autre côté, le problème du placement statique peut aussi être considéré comme un problème de partitionnement d'un graphe, dans lequel des contraintes supplémentaires ont été incluses ; le terme de partitionnement biaisé est alors utilisé [HLVD97].

1.4.2 L'approche « diviser pour résoudre » par bipartitionnement récursif conjoint

L'approche « diviser pour résoudre », présentée en section 1.3.5 de la page précédente pour le problème du partitionnement, peut être adaptée au problème de placement. Les graphes sources et cibles (habituellement associés au programme et à la topologie) sont récursivement et conjointement bipartitionnés selon l'algorithme 5.

Les lignes 1 à 7 correspondent au cas de base de l'algorithme récursif : si le graphe source est vide ou si le graphe cible est réduit à un seul sommet, alors le résultat est trivial et est directement retourné. Aux lignes suivantes, le cas d'induction est défini. À la ligne 8, le graphe cible est bipartitionné en parties de poids proches. À la ligne 9, le graphe source est bipartitionné en prenant en compte les caractéristiques des deux parties du graphe cible, et notamment leurs poids. La récursivité est mise en œuvre aux lignes 10 et 11.

En réalité, l'algorithme 5 est une version simplifiée de l'algorithme récursif de placement par bipartitionnement conjoint. Les sommets voisins de ceux situés aux extrémités du sous-graphe source en train d'être traité sont à une distance qui est fonction du résultat du bipartitionnement. Ainsi, le prise en compte des gains positifs ou négatifs induits par ces sommets externes au

Algorithme 5: Algorithme récursif `placement`(S, T) de placement par bipartitionnement conjoint.

Entrée : S un graphe source
T un graphe cible
Sorties : La fonction de correspondance $\rho_{S,T}$

```
1  si taille(S) = 0 alors
2  |  retour
3  fin
4  si taille(T) = 1 alors
5  |  resultat(T, S)
6  |  retour
7  fin
8  (T0, T1) ← bipartitionnement_graphe_cible(T)
9  (S0, S1) ← bipartitionnement_graphe_source(S, T0, T1)
10 placement (S0, T0)                                  ▷ Appels récursifs
11 placement (S1, T1)
```

bipartitionnement courant permet d'affiner le résultat du placement obtenu. Le prise en considération de ces « gains externes », qui modifie la valeur de la coupe globale du bipartitionnement en cours, correspond à la réalisation d'un *bipartitionnement biaisé*.

Le calcul des bipartitionnements successifs du graphe cible étant indépendant du graphe source, il peut être réalisé une seule fois, puis stocké sous forme d'arbre. Les auteurs de Scotch (voir section 1.10 de la page 34) ont introduit le terme d'*architecture* pour qualifier le regroupement des informations suivantes :

- les informations du graphe cible (poids de chaque sommet, etc.) ;
- l'arbre obtenu par bipartitionnement récursif.

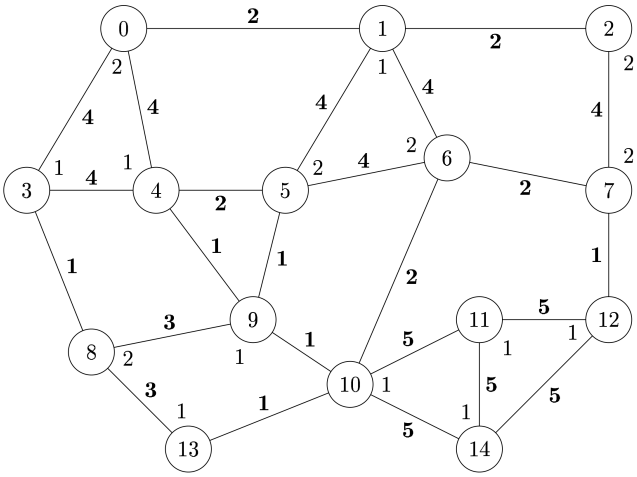
Nous utiliserons dorénavant cette définition du terme *architecture*. Une représentation des informations d'un exemple d'architecture est présentée en figure 1.9 de la page 25.

1.4.3 Exemple de placement statique par bipartitionnement conjoint

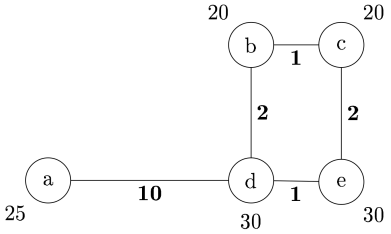
Nous considérerons les graphes de programme et de topologie de la figure 1.8. Dans un premier temps, nous étudierons comment l'architecture est calculée à partir du graphe cible. Ensuite, nous présenterons, étape par étape, comment le placement statique du graphe source est effectué à partir de cette architecture.

À la figure 1.9 est représentée, sous forme d'un arbre et d'une matrice de communication, l'architecture calculée à partir du graphe cible. Les feuilles de l'arbre correspondent aux sommets du graphe cible. Les autres sommets de la figure 1.8(b) de la page suivante (a'', a', b'' et b') représentent les parties du graphe. L'arbre est calculé grâce à un bipartitionnement récursif du graphe cible, de la manière suivante :

- Le sommet a'' est pondéré par la somme des poids de tous les sommets, il représente tout le graphe.
- Les sommets a' et b'' correspondent au premier bipartitionnement du graphe cible. La partie associée à a' contient les sommets a et d, celle associée à b'' contient les sommets b, c et e. On remarque que les poids des deux parties (**55** et **70**) sont les plus équilibrés possibles et que le poids de la frontière (**3** d'après la figure 1.8(b)) est minimal.



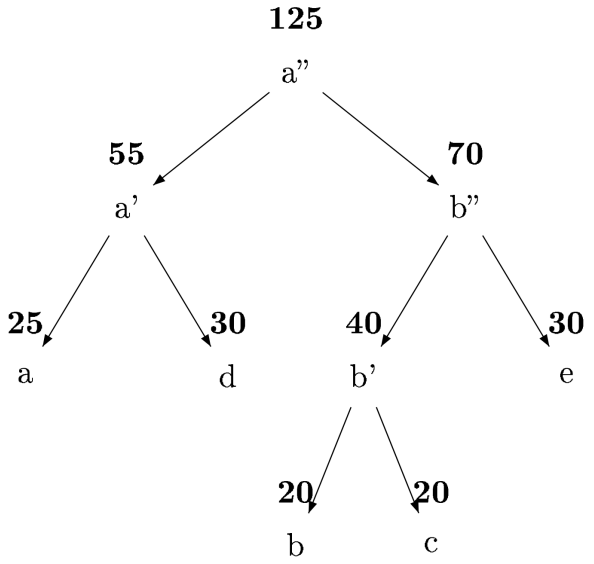
(a) Le graphe source.



(b) Le graphe cible.

FIGURE 1.8 – Exemples de graphes considérés pour un placement statique.

– Ensuite, on effectue à nouveau un bipartitionnement des parties associées aux sommets non terminaux, a' et b''. Dans le premier cas, on obtient les deux feuilles a et d, dans le second cas, on obtient le sommet b' et la feuille e.
– Pour finir, b' est bipartitionné en deux feuilles : b et c.



(a) Arbre contenant le bipartitionnement récursif du graphe cible et les poids des sommets.

	a''	a'	b''	a	d	b'	e	b
a''								
a'	6							
b''	5	7						
a	9	5	12					
d	4	5	2	10				
b'	5	8	2	13	3			
e	4	6	2	11	1	3		
b	5	7	2	12	2	1	3	
c	5	8	2	13	3	1	2	1

(b) Matrice représentant les coûts de communication entre les sommets de l’arbre.

FIGURE 1.9 – Représentation de l’architecture associée au graphe cible de la figure 1.8(b).

Alors que l’arbre permet de stocker de manière structurée les poids des sommets, les poids des arêtes sont stockés dans la matrice de communication. Cette matrice est calculée en partant des feuilles de l’arbre :

– Tous les poids connus (les coûts de communication inter-feuilles) sont ajoutés dans la matrice.
– En remontant dans l’arbre, les poids des autres sommets sont calculés en effectuant la moyenne des poids des fils. Par exemple, le poids correspondant au pseudo-coût de communication entre a' et e est égal à la moyenne des poids sur les arêtes $\{a, e\}$ (**11**) et $\{d, e\}$ (**1**), c’est-à-dire à **6**.

Une fois que l’architecture, calculée à partir du graphe cible, est disponible, un placement statique peut être calculé. La figure 1.10 présente les différentes étapes de récursivité de l’algorithme de placement statique.

1.5 Repartitionnement de graphes

1.5.1 Définition

Définition 24 (Problème du k-repartitionnement de graphes)
Le problème de k-repartitionnement d’un graphe non orienté $G = (V, E)$ à arêtes et sommets valués dans $\mathbb{R}$ où $\psi : V \to [\![1; k]\!]$, $c_m : V \to \mathbb{R}$ désignent les applications qui à chaque sommet associent un ancien numéro de partie et un coût de migration, s’énonce de la manière suivante :

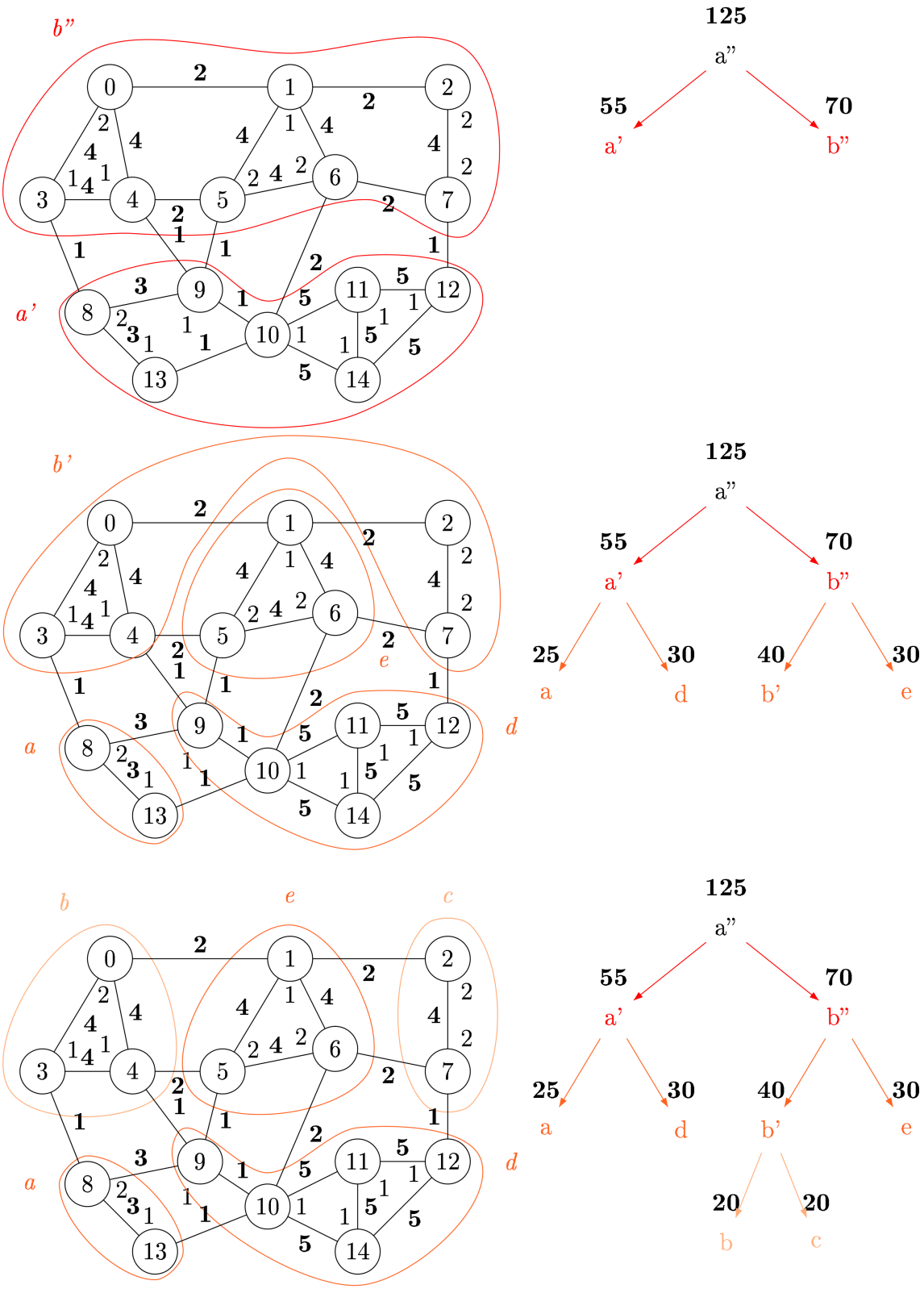


FIGURE 1.10 – Placement statique du graphe source présenté en figure 1.8(a) sur le graphe cible de la figure 1.8(b).

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

1. *le poids de chaque partie* V_i *soit égal, à la tolérance* τ *près, au poids moyen d'une partie* w_{moy} *:*
$$\forall i \in [\![1; k]\!], |w_v(\pi_i) - w_{moy}| < \tau \quad ;$$
2. *la coupe* $\sum_{\forall i\neq j, x=(v,v')\in V_i\times V_j} w_E(x)$ *soit minimale ;*
3. *le coût de migration de chaque partie* $\forall i \in [\![1; k]\!]$, $\sum_{\substack{v\in V_i \\ \psi(v)\neq i}} c_m(v)$ *soit minimal.*

Le problème du k-partitionnement est donc un cas particulier du problème de k-repartitionnement dans lequel la fonction c_m est nulle, c'est-à-dire qu'on ne tient aucunement compte d'un partitionnement précédent.

Parmi les algorithmes proposés (voir [cBB+07] et ses références), on peut distinguer deux classes principales.

1.5.2 L'approche *scratch-remap*

Les méthodes de type *scratch-remap* [OB98, SKK01] découplent le problème du rééquilibrage de la charge du problème du choix des sommets à migrer. Elles calculent un partitionnement du nouveau graphe, puis associent les nouvelles parties ainsi formées aux processus, en s'assurant que le nombre de sommets migrés soit minimal.

Ces méthodes privilégient l'optimisation des indicateurs pris en compte lors des partitionnements, notamment l'équilibrage de la charge et la taille de la coupe, au détriment de la minimisation du nombre de sommets à migrer, qui peut être très important.

1.5.3 L'approche par diffusion

Les méthodes de diffusion [Cyb89, HBE98, SKK97, WC00, WC02, WCE97] modifient itérativement le partitionnement existant en migrant les sommets situés en bordure des parties les plus chargées vers leurs parties voisines, de proche en proche, jusqu'à ce que le déséquilibre ait suffisamment décru.

Ces méthodes, du fait de leur caractère intrinsèquement local, donnent de bons résultats lorsque le nouveau graphe est proche de l'ancien, mais peuvent aboutir à une solution globalement non optimale, en termes de coupe, lorsque les modifications topologiques et de poids du graphe deviennent importantes. En outre, ces méthodes sont coûteuses en temps et difficilement parallélisables, que la migration soit réalisée de façon itérative [SKK01], ou bien calculée au moyen d'un solveur d'optimisation linéaire [MMS09].

1.5.4 L'approche par partitionnement biaisé

Un compromis entre ces deux types de méthodes consiste à utiliser des algorithmes de partitionnement sur une version modifiée du graphe qui intègre, sous forme de sommets et d'arêtes supplémentaires, des informations relatives aux coûts induits par la migration des sommets [cBD+08, cBD+07, Wal10]. En pratique, sont ajoutés au graphe d'origine, pour chaque partie, un sommet ancre représentant cette partie, relié par des arêtes fictives à tous les sommets subsistant du partitionnement initial et appartenant à la partie considérée. Le poids de

ces sommets ancres est considéré comme nul, afin qu'il n'ait pas d'impact sur le repartitionnement. Sur ce nouveau graphe, le fait de migrer un sommet vers une autre partie va induire une augmentation de la taille de la coupe égale au poids de l'arête fictive, et permet ainsi d'intégrer l'optimisation du nombre de sommets migrés à l'objectif de minimisation de la taille de la coupe. Le choix du poids des arêtes fictives par rapport à celui des arêtes d'origine définit le rapport entre le coût de migration d'un sommet et le coût de communication classique représenté par les poids des arêtes.

Les techniques de partitionnement biaisé, déjà utilisées dans le contexte du placement statique (voir section 1.4 de la page 22) permettent, avec très peu de modifications par rapport aux algorithmes de placement statique, d'optimiser à la fois l'équilibrage de la charge entre parties, la taille de la coupe et la quantité de données devant être redistribuées entre les processeurs. C'est cette approche que nous avons choisi de suivre dans le cadre de notre travail.

1.6 Replacement de graphes

Avant de présenter l'approche que nous avons considérée pour aborder le problème du replacement de graphes, il convient de la définir.

1.6.1 Définition

Définition 25 (Problème du replacement de graphes)
Soient $S = (V_S, E_S)$ et $T = (V_T, E_T)$ deux graphes non orientés (définissant respectivement le graphe source et le graphe cible) à arêtes et sommets valués dans $\mathbb{R}$ où $\psi : V \to V_T$, $c_m : V_S \to \mathbb{R}$ désignent les applications qui, à chaque sommet du graphe source, associent un ancien sommet du graphe cible et un coût de migration. Le problème du replacement du graphe S sur le graphe T s'énonce de la manière suivante :

Trouver un couple ($\phi_{S,T}$,$\rho_{S,T}$) où $\phi_{S,T}$ est une application de V_S dans V_T, et $\rho S, T$ est une application associant, à toute arête $\{u, v\} \in E_S$, un chemin reliant $\phi_{S,T}(u)$ à $\phi_{S,T}(v)$ tel que :

1. *la charge $c(v_T) = \dfrac{\sum_{\substack{v_S \in V_S \\ \phi_{S,T}(v_S)=v_T}} w_S(v_S)}{w_T(v_T)}$ de chaque sommet de T soit, à la tolérance τ_1 près, la même :*
$$\forall (t_i, t_j) \in {V_T}^2, |c(t_i) - c(t_j)| < \tau_1 \ ;$$
2. *la fonction de coût de communication soit inférieure à la tolérance τ_2 :*
$$f_C(\phi_{S,T}, \rho_{S,T} = \sum_{e_S \in E_S} w_S(e_S)|\rho_{S,T}(e_S)| < \tau_2 \ .$$
3. *le coût de migration total $\sum_{\substack{v \in V_S \\ \psi(v) \neq \phi_(v)}} c_m(v)$ soit inférieur à la tolérance τ_3.*

1.6.2 Modèle et approche considérés

Le modèle et l'approche considérés pour le repartitionnement de graphes (voir section 1.5.4) s'appuyant sur le partitionnement biaisé déjà utilisé pour le placement, il n'y a pas de contre-indication à l'usage conjoint du placement et du repartitionnement. Toutefois, afin de résoudre le problème du replacement, il est nécessaire d'intégrer la notion de distance du placement statique lorsque sont choisis les poids des arêtes fictives modélisant le coût de migration. Ces

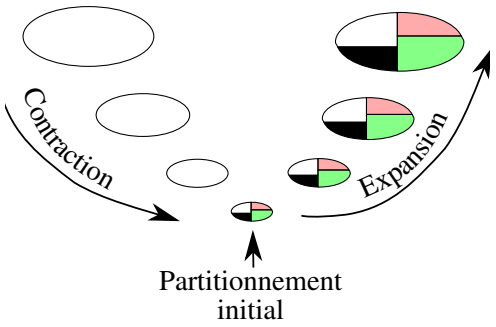


FIGURE 1.11 – Représentation schématique d'un 4-partitionnement à l'aide du schéma multi-niveaux.

modifications sont en général mineures par rapport à celles nécessaires au repartitionnement ; nous ne les évoquerons donc que peu dans la suite de ce document.

1.7 L'approche multi-niveaux

L'efficacité des méthodes combinatoires est étroitement liée à la taille de l'espace de recherche, c'est-à-dire à la taille de $\mathcal{P}(G)$. Les optimisations qu'elles effectuent ne sont généralement que locales, sans prise en compte de la topologie globale du graphe G. Au contraire, les méthodes spectrales exploitent une vision globale du graphe G, mais deviennent inefficaces lorsque la taille du graphe devient importante.

La technique multi-niveaux, introduite par Barnard et Simon [BS94, HL95, vDR94], permet de réduire la taille du graphe, ainsi que celle de l'espace de recherche, tout en offrant l'accès à une vision globale pouvant être exploitée par les algorithmes combinatoires. L'idée directrice est de travailler sur un graphe réduit ayant les mêmes propriétés topologiques que le graphe de départ.

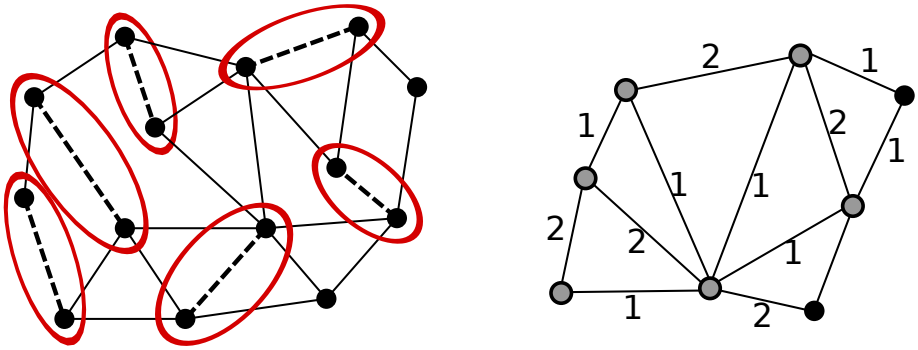
Le schéma multi-niveaux comporte trois étapes :

1. l'étape de contraction, durant laquelle une fonction de contraction est appliquée récursivement afin de diminuer la taille du graphe ;
2. l'étape de partitionnement initial, au cours de laquelle un partitionnement est calculé sur le plus petit graphe ;
3. l'étape d'expansion, durant laquelle la partition initiale est prolongée et raffinée sur les graphes de taille de plus en plus importante, jusqu'au graphe de départ.

Cette procédure est illustrée par la figure 1.11. Nous allons maintenant détailler ces différentes phases.

1.7.1 La phase de contraction

Le but de cette phase est d'obtenir une suite (G_λ) de graphes issus de G tels que leurs topologies soient proches de celle de G et que leurs nombres de sommets soient strictement décroissants.



(a) Le graphe de départ avec poids unitaires. Les arêtes en tirets sont les arêtes qui vont être contractées.

(b) Le graphe contracté. Les nombres sur les arêtes correspondent au poids de ces dernières.

FIGURE 1.12 – Exemple de contraction de graphe par appariement d'arêtes.

Le quotient de $|V_\lambda|$ et de $|V_{\lambda+1}|$,

$$r = \frac{|V_\lambda|}{|V_{\lambda+1}|} \ , \tag{1.15}$$

est appelé rapport de réduction ou de contraction r.

Généralement, les sommets de G_λ sont appariés, c'est-à-dire regroupés par deux, afin de former les sommets de $G_{\lambda+1}$. Le rapport de réduction r vaut, dans ce cas, au maximum deux. Pour apparier les sommets, différentes stratégies sont utilisées. Dans les premières implantations du schéma multi-niveaux, le choix était réalisé aléatoirement [BHJL89, HL95]. Par la suite, c'est la méthode *Heavy Edge Matching* [KK95b] qui sera la plus couramment utilisée, bien que d'autres approches aient été proposées [MPD00].

La technique *Heavy Edge Matching* est la plus populaire dans les outils actuels [kar, pel, wal], car elle est facile à implanter et produit des contractions de bonne qualité en moyenne [KK95a]. Elle procède en considérant les sommets du graphe suivant un ordre aléatoire. Chaque sommet non traité est contracté avec le voisin non encore apparié qui lui relié par l'arête de poids maximal. Ainsi, cet algorithme favorise naturellement l'optimisation de la taille de la coupe. Un exemple de contraction est proposé en figure 1.12.

Il existe, par ailleurs, d'autres techniques de contraction prometteuses qui, plutôt que de former des appariements, regroupent les sommets par ensembles. Les taux de contraction obtenus avec ces techniques peuvent être supérieurs à 2, ce qui permet de diminuer le nombre d'étapes dans le schéma multi-niveaux [CS09a, CS09b].

1.7.2 Le partitionnement initial

La phase de partitionnement initial a pour objectif de calculer, sur un graphe où le nombre de sommets contractés G_λ est suffisamment petit, une partition qui sera par la suite prolongée sur les graphes plus fins. Comme la qualité de cette partition initiale va influer de manière importante sur la qualité de la partition finale, des algorithmes coûteux pouvant être utilisés, mais leur temps d'exécution restera raisonnable, du fait de la petite taille du graphe considéré.

Dans la version actuelle de SCOTCH, le seuil de contraction est fixé, dans le cas du bipartitionnement récursif, à une centaine de sommets. Nous notons G_{λ_m} le graphe le plus contracté dans le schéma multi-niveaux.

Les heuristiques développées précédemment peuvent être appliquées à ce graphe G_{λ_m}. C'est le cas de l'approche spectrale pour [BS94, HL95], des algorithmes itératifs gloutons pour [KK98, Pel95] ou encore des algorithmes génétiques pour [AZ04].

La taille du graphe G_{λ_m} étant petite, le schéma multi-niveaux tend à ce que la partition trouvée corresponde à un optimum aussi global que possible et c'est cette globalité que la dernière phase, la phase d'expansion, va chercher à conserver.

1.7.3 La phase d'expansion

Le but de cette phase est de prolonger, jusqu'au graphe G initial, l'optimum global calculé sur le plus petit graphe G_{λ_m}.

L'idée consiste à procéder par étapes, en prolongeant la partition du niveau $\lambda + 1$ sur le niveau λ et en raffinant la partition obtenue grâce à une heuristique d'optimisation locale telle que celles de Kernighan-Lin ou de Fiduccia-Mattheyses. Le fait que les graphes de deux niveaux consécutifs λ et $\lambda+1$ soient proches topologiquement permet d'espérer que l'optimum local vers lequel va converger l'algorithme de raffinement sur G_λ ne soit pas trop éloigné de la prolongation de celui de $G_{\lambda+1}$. Par induction, si le partitionnement obtenu au niveau le plus contracté G_{λ_m} correspond à l'optimum global, l'optimum de chaque niveau G_λ correspond aussi à l'optimum global sur G_λ.

En conséquence, la contrainte de la proximité topologique entre deux niveaux est la clé de la réussite du schéma ; la qualité de la phase d'appariement des sommets est donc critique. Pour améliorer la qualité de ce schéma, Karypis et Kumar [Kar02] ont proposé d'augmenter le nombre de niveaux et de réduire le taux de contraction r à des valeurs appartenant à l'intervalle $[1,5; 1,8]$ au lieu de le faire tendre vers 2 pour accélérer les calculs.

1.7.4 Schéma multi-niveaux k-aire

Il y a deux manières d'utiliser le schéma multi-niveaux pour réaliser des k-partitionnements :

1. par bipartitionnement récursif. Dans ce cas, un cycle complet du schéma multi-niveaux (contraction, partitionnement initial, expansion) est réalisé à chaque bipartitionnement ;
2. par partitionnement k-aire. Dans ce cas, c'est une k-partition qui est calculée lors de la phase de partitionnement initiale et qui est prolongée lors de la phase d'expansion.

La première approche est plus simple à mettre en œuvre, mais l'appel multiple au schéma multi-niveaux est coûteux en termes de temps d'exécution. La seconde approche est plus rapide mais nécessite, lors de la phase d'expansion, l'utilisation d'algorithmes de raffinement k-aire (de k-partitions) qui sont plus complexes.

1.7.5 Parallélisation

Le schéma multi-niveaux se parallélise facilement. La phase de contraction peut être parallélisée au moyen d'un algorithme de coloration [Che07]. Le partitionnement initial est réalisé au moyen des heuristiques de partitionnement séquentiel. La phase d'expansion parallèle est la plus délicate à mettre en œuvre, car les algorithmes de raffinement habituels (Fiduccia-Mattheyses ou Kernighan-Lin) sont intrinsèquement séquentiels. Nous reviendrons sur ce dernier point au chapitre 3.

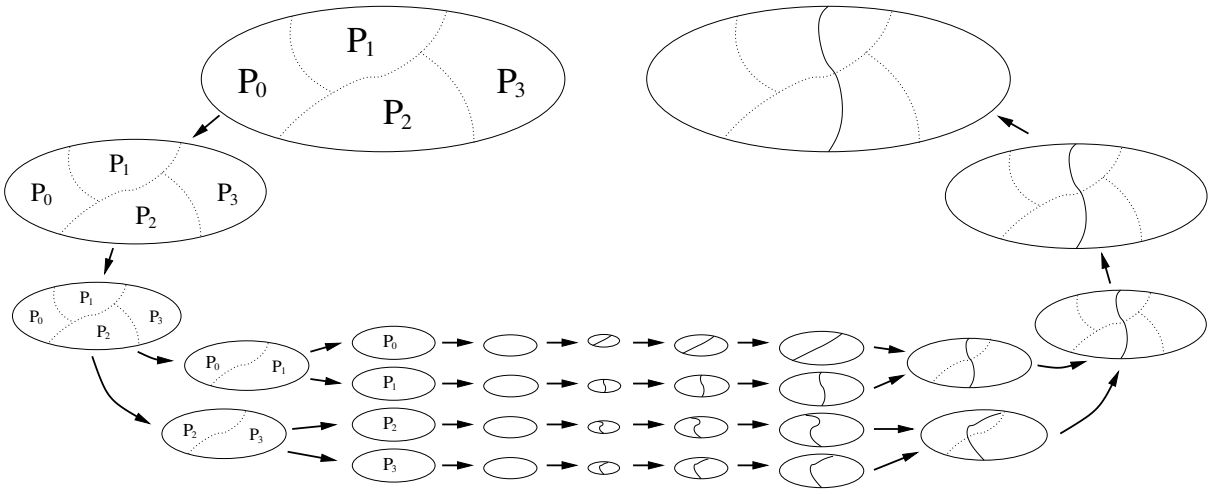


FIGURE 1.13 – Schéma multi-niveaux parallèle avec repliement et duplication sur quatre processeurs. Le calcul du partitionnement séquentiel est réalisé en parallèle, puis, lors de l'expansion, les meilleures partitions sont conservées lors d'un bipartitionnement.

1.7.6 Le repliement avec duplication dans un contexte k-aire

La phase de contraction se déroule en calculant à chaque niveau un graphe avec environ deux fois moins de sommets que le graphe du niveau immédiatement supérieur. Le graphe étant réparti sur un nombre constant de processeurs, ces derniers possèdent, à chaque niveau, un nombre de sommets divisé par deux, le nombre d'arêtes diminue quant à lui de manière moindre [KK95c]. Plus le nombre de sommets par processeur devient faible, plus le temps de communication est dominé par le temps d'initialisation des communications et peut être responsable d'une dégradation des performances en temps.

Devant ce constat, un système de repliement avec duplication de graphe a été ajouté au sein des algorithmes de contraction de SCOTCH. Cet algorithme est illustré par le schéma de la figure 1.13. Nous parlons de repliement lorsque les données correspondant au graphe contracté sont redistribuées sur la moitié des processeurs qui hébergent les données du graphe de plus haut niveau. En outre, le repliement a pour objectif de diminuer la taille de l'interface, et donc le coût et le nombre des communications inter-processeurs. Le repliement a pour effet de laisser la moitié des processeurs inactifs. Aussi, pour tirer parti de cette puissance de calcul, en même temps qu'il est replié, le graphe est dupliqué sur ce second ensemble de processeurs – induisant une charge mémoire constante, plutôt que divisée par deux. Ce processus est alors appelé « repliement avec duplication ». Après la duplication, nous effectuons des permutations des sommets, de telle sorte que le calcul de la partition initiale aboutisse à des solutions différentes et ensuite, lors de la phase d'expansion, nous sélectionnons la meilleure partition.

L'ajout à SCOTCH du repliement avec duplication, ainsi que son analyse, ont été réalisés par Cédric CHEVALIER [CP08]. Comme cette technique n'impacte pas nos travaux, nous ne la détaillerons pas plus avant.

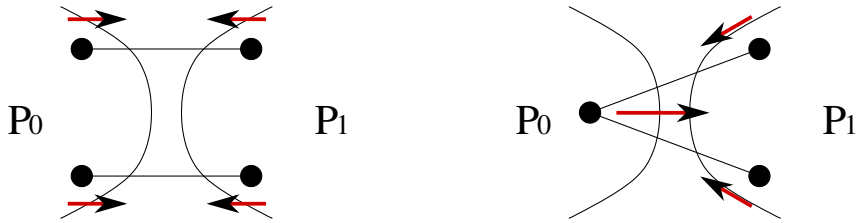


FIGURE 1.14 – Imprécision du modèle de partitionnement de graphe pour représenter un échange de données. Pour les deux bipartitions présentées, la coupe est de 2, mais 4 données associées aux sommets doivent être échangées dans le cas de gauche, contre 3 pour celui de droite.

1.8 Prise en compte des sommets fixes

L'utilisateur des fonctionnalités de (re)partitionnement et de (re)placement peut souhaiter, en plus de fournir une ancienne partition, fixer certains sommets dans des parties données. Cette contrainte supplémentaire de prise en compte d'un ensemble $F \subset V$ de sommets fixes s'exprime de la manière suivante pour le replacement (voir la définition 25 de la page 28 pour les notations) :

$$\forall v \in F, \psi(v) = \rho(v) \ .$$

L'usage de ces sommets fixes peut être multiple. Dans le cas du placement, il peut servir à respecter une contrainte logicielle (une partie des données ne doit pas être migrée) ou matérielle (sur une architecture hétérogène, un processus spécifique doit être placé sur le processeur réalisant les opérations d'entrée-sortie).

Son usage est orthogonal au replacement et offre une flexibilité supplémentaire pour résoudre des problèmes de répartition de la charge complexes. Évoquons, à titre d'exemple, le placement multi-phases, qui consiste à réaliser un placement multi-objectifs en plusieurs phases.

1.9 Partitionnement de graphes et partitionnement d'hypergraphes

La modélisation du comportement d'une application sous forme de graphe peut représenter les échanges de données de façon imprécise. Par exemple, dans le cas des solveurs de systèmes linéaires hybrides [5] ou itératifs, l'opération la plus importante est un produit matrice-vecteur parallèle. Pour cette opération, les données correspondant aux sommets à l'interface doivent être envoyées aux processeurs voisins. Bien que l'envoi de l'information ne soit effectué qu'une fois par processeur voisin, la coupe associée peut comprendre plusieurs arêtes qui ne correspondent pas exactement à la quantité donnée échangée ; ceci est illustré en figure 1.14. C'est pour cette raison que de nombreux auteurs préfèrent utiliser le modèle de communication basé sur les hypergraphes et des outils de partitionnement d'hypergraphes [cA99, cA01, Hen98]. En effet, le paradigme des hypergraphes, grâce à la notion d'hyper-arête — permettant de relier entre eux plus de deux sommets, est capable de modéliser plus précisément les communications.

Bien que le partitionnement d'hypergraphe permette de représenter les échanges de données de manière exacte, réduisant ainsi le coût de communication par rapport au partitionnement de graphe jusqu'à 60 % dans certains cas [cA99], il est plus coûteux en termes de temps d'exécution.

5. La méthode de résolution hybride est basée sur le complément de Schur et consiste à réaliser une résolution directe sur les domaines répartis sur les différents processeurs, puis à effectuer une résolution itérative pour les séparateurs [GH08].

En particulier, pour des graphes 2D ou 3D avec des degrés limités, qui impliquent l'existence de séparateurs lisses et continus, le gain en qualité de coupe induit par l'usage d'un partitionneur d'hypergraphes plutôt qu'un partitionneur de graphes est réduite.

1.10 Plateforme d'expérimentation considérée : Scotch

Afin d'évaluer expérimentalement nos algorithmes de repartitionnement, nous nous appuierons sur Scotch [pel], que nous allons donc présenter dans cette section.

Scotch est un ensemble de « logiciels et bibliothèques séquentiels et parallèles pour le partitionnement de graphes, le placement statique, et la renumérotation par blocs de matrices creuses, et le partitionnement séquentiel de maillages et d'hypergraphes ». C'est un projet développé au sein de l'équipe Satanas du Laboratoire Bordelais de Recherche en Informatique (LaBRI). Il fait partie du projet Bacchus de l'INRIA Bordeaux - Sud-Ouest et est publié sous licence CeCILL-C [CeC].

1.10.1 Objectif

L'objectif de Scotch est énoncé comme suit :

> « Son but est d'appliquer la théorie des graphes, avec une méthodologie de type diviser pour résoudre, à des problèmes d'informatique scientifique tels que le partitionnement de graphes et de maillages, le placement statique, et la renumérotation de matrices creuses, dans le cadre de domaines d'application allant de la mécanique des structures aux systèmes d'exploitation ou à la biochimie.
>
> La distribution Scotch est un ensemble de programmes séquentiels et parallèles et de bibliothèques qui implémentent les algorithmes de placement statique et de renumérotation de matrices creuses conçus au sein du projet Scotch. »

1.10.2 Usages

La partie séquentielle de Scotch permet de :
- partitionner des graphes et des maillages ;
- réaliser des placements statiques ;
- renuméroter des graphes et des maillages ;
- calculer des regroupements.

La partie parallèle propose les fonctionnalités suivantes :
- partitionner des graphes ;
- réaliser des placements statiques ;
- renuméroter des graphes.

1.10.3 Stratégies

Scotch permet d'utiliser et de combiner entre elles nombre des heuristiques présentées dans ce chapitre. Afin d'offrir une grande flexibilité quant à la combinaison de ces dernières, une grammaire permet de configurer la combinaison des heuristiques à l'aide de « chaînes de stratégies ». Celles-ci sont détaillées dans le manuel d'utilisation, aussi nous bornerons-nous à en expliciter un exemple.

L'exemple de chaîne de stratégie présenté en figure 1.15 de la page suivante peut être interprété de la manière suivante :

```
m{
  vert=10000,
  [...]
  low=
    r{
      job=t,
      bal=0.01,
      map=t,
      poli=S,
      sep=
        (m{[...]})
    },
  asc=
    f{
      move=80,
      pass=-1,
      bal=0.01
    }
}
```

FIGURE 1.15 – Un exemple de chaîne de stratégie représenté hiérarchiquement.

Le k-partitionnement est réalisé au moyen d'un schéma multi-niveaux k-aire (`m`) qui comprend des options nécessaires notamment à la configuration de la phase de contraction. Ainsi, `vert=10000` indique que le graphe va être contracté jusqu'à ce qu'il comprenne moins de 10 000 sommets, `low` est la stratégie de k-partitionnement initial et `asc` est la stratégie de raffinement lors de la phase d'expansion. Le partitionnement initial est calculé par bipartitionnement récursif (`r`) et la stratégie de bipartitionnement utilisée (`sep`) se base sur un schéma multi-niveaux (`m`). D'autres options sont fournies, telles que le déséquilibre maximal (`bal=0.01`). La stratégie de raffinement s'appuie sur une version k-aire de l'heuristique de Fiduccia-Mattheyses (`f`).

Pour faciliter la configuration de ces stratégies, des options de configuration ont été mis à disposition des utilisateurs :

- `SCOTCH_STRATQUALITY` : permet de privilégier la coupe ;
- `SCOTCH_STRATBALANCE` : assure le respect de la contrainte d'équilibrage de la charge ;
- `SCOTCH_STRATSPEED` : réduit la complexité de la stratégie afin de gagner en termes de temps d'exécution.

1.11 Mise en œuvre parallèle

La taille des graphes à partitionner étant de plus en plus importante, la mémoire des ordinateurs séquentiels actuels se révèle insuffisante pour permettre l'exécution des algorithmes séquentiels de repartitionnement. D'autre part, pour de nombreuses applications, le repartitionnement de graphes est appelé à intervalles réguliers pour équilibrer dynamiquement la charge au sein de codes exécutés en parallèle. Le temps de repartitionnement séquentiel devient alors trop important par rapport à celui de la tâche principale.

Pour ces deux raisons, certains outils de partitionnement parallèle de graphes ont été enrichis de fonctionnalités de repartitionnement parallèle. C'est le cas de PARMETIS [kar] pour ce qui concerne le repartitionnement de graphes et de ZOLTAN [BDF+99] pour ce qui concerne le

repartitionnement d'hypergraphes.

Tous ces outils, qui s'appuient sur un schéma multi-niveaux parallèle, utilisent des algorithmes qui présentent des faiblesses. Les expliciter nous permettra de motiver le choix de la réalisation du présent travail et de préciser la direction que nous avons choisi de suivre afin de tenter d'y remédier. Auparavant, il nous semble utile de rappeler quelques notions de parallélisme.

1.11.1 Vocabulaire du parallélisme

Nous considérerons par la suite que nous disposons de p processeurs.

Définition 26 (Temps effectif parallèle, accélération)
Le temps effectif parallèle est la durée que l'utilisateur attend pour que le programme parallèle termine. Nous le noterons t_p dans la suite de ce document.

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

*L'accélération (*speed-up *en anglais)* a_p *correspond au rapport* t_p *sur* t_s *:*

$$a_p = \frac{t_p}{t_s} \ . \tag{1.16}$$

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

Définition 27 (Efficacité, rendement)
L'efficacité (ou rendement) η_p *est définie comme le quotient entre* a_p *et* p*. Elle est donc majorée par* 1.

$$0 \leq \eta_p = \frac{a_p}{p} \leq 1 \ . \tag{1.17}$$

Nous dirons qu'un programme est *scalable* en temps lorsque son efficacité tend vers 1 lorsque le nombre de processeurs p tend vers $+\infty$.

1.11.2 Formulation parallèle

Maintenant que nous avons défini des notions d'évaluation de la qualité d'un algorithme parallèle, nous pouvons définir plus précisément ce que signifie l'expression *partitionnement parallèle de graphes*.

Le problème du partitionnement reste celui posé à la définition 18 de la page 10, auquel s'ajoute une contrainte concernant l'utilisation des p processeurs disponibles : le partitionneur doit être *scalable* en mémoire et en temps.

Les problèmes de k-repartitionnement (définition 24 de la page 25), de placement statique (définition 23 de la page 22) et de replacement (définition 25 de la page 28) sont étendus de manière analogue aux problèmes de repartitionnement, de placement statique et de replacement parallèle de graphes.

Les surcoûts en mémoire et en temps dus aux communications entre les processeurs rendent difficile l'obtention d'un programme *scalable* tel que défini plus haut, mais le but visé par toutes les méthodes proposées dans la littérature est de les minimiser autant que possible, tout en conservant un objectif qualitatif pour le résultat. Il est par ailleurs souhaitable que la qualité du

repartitionnement produit par l'outil parallèle soit invariante (varie augmente) lorsque le nombre de processeurs utilisés augmente et soit identique à celle obtenue en séquentiel (ou meilleure).

Le besoin de *scalabilité* mémoire impose de travailler avec des graphes qui seront répartis sur l'ensemble des processeurs. Le fait que les ordinateurs que nous considérons ne disposent pas d'une mémoire partagée nous place, de fait, dans le cadre d'un algorithme parallèle distribué. Nous utiliserons, par ailleurs, des architectures employant des systèmes d'échanges de messages tels que MPI [mpi97].

En pratique, notre objectif sera de proposer des algorithmes adaptés aux architectures *petascale* et *exascale*. Parmi ces architectures, nous pouvons citer le projet *Blue Waters* [blu] et le laboratoire commun associé, l'*INRIA-Illinois Petascale Computing Joint Lab* [joi], au sein duquel nous avons réalisé des collaborations (voir à ce sujet la section 2.7.4 de la page 61).

1.11.3 Résultats actuels en parallèle et positionnement

L'outil de référence pour le repartitionnement parallèle de graphes est ParMeTiS [kar]. En termes de partitionnement, la qualité du partitionnement se dégrade lorsque le nombre de processus augmente [CP06b]. Par ailleurs, l'algorithme de raffinement qu'il utilise nécessite l'allocation d'une matrice de taille $p \times p$ qui, sur un très grand nombre de processeurs, pourrait constituer une limite en termes de *scalabilité*.

La perte de qualité observée avec l'augmentation du nombre de processeurs est due notamment à la difficulté d'obtenir une bonne parallélisation du raffinement réalisé lors de la phase d'expansion du schéma multi-niveaux. Outre cela, ce schéma conserve les qualités explicitées plus haut, notamment des résultats de qualité similaire pour un coût plus faible que ceux qui auraient été obtenus sans ce dernier. En conséquence, son utilisation reste de mise et notre contribution sera, pour l'un de ses axes, d'étudier de nouveaux algorithmes de raffinement ou des apports au schéma multi-niveaux parallèle pour que ce dernier gagne en *scalabilité*.

1.12 Conclusion

Après avoir défini les notions usuelles, nous avons présenté les problèmes que nous aborderons dans le cadre de cette thèse ainsi que plusieurs approches pour les résoudre. Concernant le problème du repartitionnement, il en ressort que, d'une part, l'approche multi-niveaux permet d'améliorer la qualité des résultats obtenus pour un coût moindre et, d'autre part, que l'approche par partitionnement biaisé nous semble préférable dans notre contexte.

Nous allons maintenant présenter notre contribution concernant les algorithmes séquentiels qui peuvent être utilisés pour résoudre le problème du repartitionnement séquentiel de graphes. Ces algorithmes seront, par la suite, utilisés lors de la phase de repartitionnement initial du schéma multi-niveaux parallèle.

Chapitre 2

Contribution au repartitionnement séquentiel

Sommaire

2.1 Introduction

À plusieurs titres, la mise en place du repartitionnement séquentiel est un premier pas vers la réalisation d'un outil de repartitionnement parallèle. C'est un passage nécessaire, car le repartitionneur parallèle intégrera ces fonctionnalités, notamment pour le calcul de la partition initiale au sein du schéma multi-niveaux. C'est aussi l'occasion de réfléchir dès maintenant à la *scalabilité* de nos méthodes et à la préservation de leur qualité dans le contexte de leur formulation parallèle. Enfin, ce travail nous permettra d'évaluer plus aisément la qualité des algorithmes utilisés.

2.2 Principe de fonctionnement

Au chapitre précédent, nous avons vanté les mérites du schéma multi-niveaux pour le calcul de partitionnements et de placements statiques. Nous allons maintenant décrire de quelle manière nous avons adapté ce schéma, ainsi que les algorithmes adaptés au repartitionnement. Afin d'être en mesure d'évaluer le comportement de ces algorithmes, il a fallu les implémenter. Nous avons choisi pour ce faire de nous appuyer sur les algorithmes de partitionnement pré-existant au sein de la bibliothèque Scotch [pel].

2.2.1 Utilisation d'un schéma multi-niveaux k-aire

Jusqu'à maintenant, le calcul de k-partitions avec Scotch était effectué par un algorithme de bissection récursive qui nécessitait la réalisation complète d'un schéma multi-niveaux pour le calcul de chaque bissection.

Les algorithmes de raffinement de bipartitions sont plus faciles à implémenter et plus rapides que leurs équivalents k-aires. En effet, l'information relative à la partie dans laquelle les sommets sont affectés peut être contenue dans un booléen et en conséquence les structures de données manipulées par ces algorithmes peuvent être simplifiées. Toutefois, le coût des appels multiples au schéma multi-niveaux devient rapidement prédominant lorsque le nombre de parties augmente.

Étant donné que nous souhaitons utiliser nos algorithmes de repartitionnement sur de très grands graphes et pour de très grands nombres de parties, nous avons choisi d'utiliser un partitionnement k-aire, dans lequel le schéma multi-niveaux n'est appelé qu'une seule fois. Il n'existait pas de schéma multi-niveaux k-aire au sein de Scotch. Ainsi, la conception et la mise en œuvre d'algorithmes pour le partitionnement k-aire au sein de ce logiciel est la première de nos contributions.

2.2.2 Prise en compte des coûts de migration

Avant d'aborder l'adaptation des trois phases du schéma multi-niveaux au problème de repartitionnement, il convient d'expliciter la manière dont nous avons pris en compte les coûts de migration.

D'un point de vue théorique, nous avons choisi d'adapter les algorithmes de partitionnement implémentés dans Scotch en ajoutant des arêtes fictives aux graphes à repartitionner [cBD$^+$08, cBD$^+$07, Wal10]. Cette approche présente l'avantage de réutiliser les algorithmes de graphes, déjà éprouvés, pour ce problème plus complexe. Notons par ailleurs que le choix d'utiliser des arêtes pour représenter les coûts de migration conduit à les considérer comme des coûts de communication, ce qui est tout à fait naturel, car ils le sont en pratique. Le schéma multi-niveaux k-aire modifié pour prendre en compte ces arêtes fictives est présenté en figure 2.1. Les

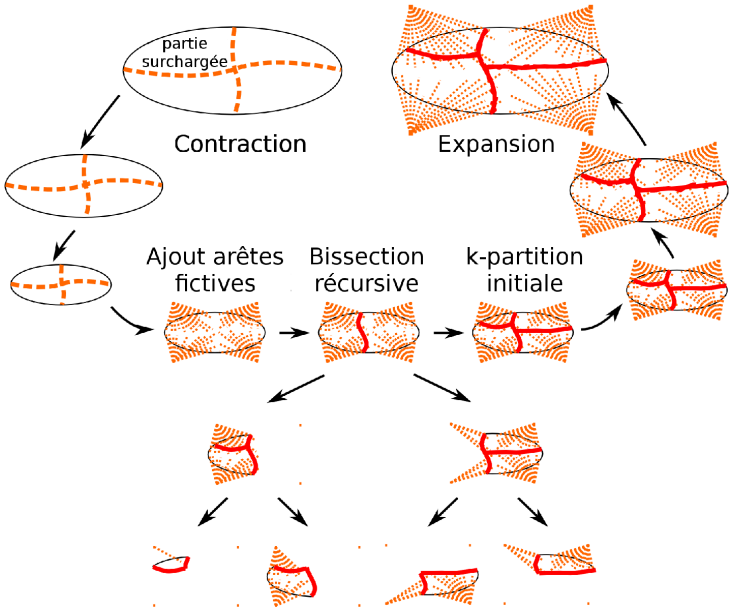


FIGURE 2.1 – Adaptation du schéma multi-niveaux au problème de repartitionnement.

arêtes fictives induisent une augmentation de la taille de la coupe, à chaque fois qu'un sommet est migré dans une partie différente de son ancienne partie.

L'ajout des arêtes fictives est une manière élégante de résoudre le problème du repartitionnement de graphes. Toutefois, dans le cas des graphes de très grande taille, les ajouter réellement dans nos structures de données serait coûteux en termes d'espace mémoire et de temps d'exécution. L'ajout et la suppression de ces arêtes induirait d'importantes manipulations mémoires [6]. Qui plus est, ces arêtes devraient être parcourues en même temps que les arêtes normales, lors de l'exécution des algorithmes de calcul. Pour cette raison, et comme nous allons le voir par la suite, nous avons fait en sorte, dans notre implémentation, de simuler leur existence, plutôt que de les ajouter réellement au sein de nos structures de données.

Afin d'effectuer un repartitionnement, l'utilisateur doit fournir en entrée l'ancienne partition, ainsi que, pour chaque sommet, un coût de migration. Dans SCOTCH, ainsi que dans la majorité des outils de partitionnement de graphes, les poids des sommets et des arêtes sont stockés sous la forme d'entiers. Deux raisons motivent ce choix : d'une part, le traitement de l'arithmétique des entiers est en général plus rapide que celui des réels flottants et permet l'usage de la superscalarité, lorsque les processeurs disposent de plusieurs unités arithmétiques entières ; d'autre part, il permet d'éviter que des arrondis aient lieu lors des calculs. Ce second avantage est critique dans le cas des algorithmes itératifs où, d'une part, plusieurs chemins différents doivent conduire au même résultat et, d'autre part, revenir à un état antérieur doit redonner le même résultat qu'avant. Si ce n'était pas le cas, le déroulement de ces algorithmes pourrait conduire à des boucles infinies, si une suite d'erreurs d'arrondis conduisait à revenir dans une situation antérieure avec des valeurs d'état différentes.

En dépit de toutes ces raisons, l'utilisateur peut souhaiter utiliser des coûts de migration qui ne soient pas des multiples des poids des arêtes du graphe, ou qui soient plus petits que le poids unitaire d'une arête. Afin de répondre à ces besoins, nous avons conçu une interface où les

6. Par exemple, cela nécessiterait pour les structures du graphe au sein de SCOTCH, de l'ajout d'un espace mémoire d'*integer* de taille deux fois égal au nombre d'arêtes fictives.

coûts de migration sont fournis par l'utilisateur sous la forme de deux paramètres : un coefficient réel flottant et un tableau qui associe à chaque sommet du graphe un coût entier. Le coût de migration d'un sommet est donc égal :

- au coût fourni dans le tableau multiplié par le coefficient, si les deux paramètres sont fournis ;
- au coût fourni dans le tableau, si seul ce dernier est fourni ;
- au coefficient, si seul celui-ci est fourni (dans ce cas, le coefficient jouera le rôle de coût de migration global).

Le coefficient réel flottant est approché en interne sous la forme d'une fraction entière. Son dénominateur est multiplié par le poids des arêtes du graphe, ainsi que par le coefficient réel flottant (qui devient alors égal au numérateur), afin d'obtenir un nouveau coefficient entier. Ce dernier est alors utilisé pour calculer, tel que nous l'avons défini ci-dessus, les coûts de migration. Les poids des sommets du graphe, ainsi que les coûts de migration résultant de ces calculs, étant entiers, ils permettent de profiter des avantages de l'arithmétique entière dans nos algorithmes.

Comme nous l'avons dit précédemment, nous avons défini l'interface de nos routines de repartitionnement de telle sorte que l'utilisateur doive fournir l'ancienne partition, ainsi qu'un tableau contenant les coûts de migration associés à chaque arête fictive, et donc à chaque sommet. Nous autorisons ainsi l'utilisateur à fournir des coûts de migration différents des poids des sommets. Par ailleurs, si l'utilisateur ne souhaite pas donner de coûts de migration différents pour chaque sommet, nous lui donnons la possibilité de fournir un seul coût de migration global pour tous les sommets.

Nous allons maintenant nous intéresser à l'adaptation de la première phase du schéma multi-niveaux.

2.3 Adaptation de la phase de contraction

L'adaptation de la phase de contraction au problème de repartitionnement nécessite la prise en compte d'une contrainte supplémentaire. En effet, les informations de l'ancienne partition doivent être propagées jusqu'au graphe le plus petit, pour pouvoir calculer le partitionnement initial lors de la seconde phase du schéma multi-niveaux. Pour ce faire, deux approches existent.

L'approche naïve consiste à ajouter des arêtes fictives représentant l'ancienne partition sur le graphe d'origine avant la phase d'appariement. La taille du graphe augmentera d'autant de « sommets ancres » (voir section 1.5.4 de la page 27) qu'il y avait de parties dans l'ancienne partition, et d'autant d'arêtes qu'il y a de sommets dans le nouveau graphe qui appartiennent à l'ancien graphe. La prise en compte de ces données supplémentaires aurait alors pour effet de ralentir la phase d'appariement et nécessiterait l'ajout de code spécifique pour prendre en compte les sommets ancres, de telle sorte qu'ils ne soient pas appariés à des sommets normaux.

L'autre solution consiste à créer les arêtes fictives seulement après la phase de contraction, au moment où l'on calcule la partition initiale. Pour que cette solution soit possible, il faut que l'information de l'ancienne partition soit disponible pour le plus petit graphe. Elle doit donc être prolongée à chaque contraction de graphe. Afin de ne pas perdre d'information lors de ces contractions, il est alors nécessaire de n'autoriser l'appariement que lorsque les deux sommets appartenaient à la même ancienne partition.

Nous avons donc choisi d'implémenter la seconde solution en ajoutant, dans les algorithmes d'appariement existants, un test, pour s'assurer que les appariements ont bien lieu entre des sommets qui n'appartenaient pas à des anciennes parties différentes.

2.4 Adaptation du placement initial

Une k-partition initiale du graphe le plus contracté (nous avons fait le choix d'arrêter la contraction à 10 000 sommets [7]) est classiquement calculée par bipartitionnement récursif. Pour être en mesure de calculer un k-repartitionnement initial, nous avons donc adapté les routines de bipartitionnement récursif au repartitionnement. Comme pour la contraction, deux approches peuvent être envisagées.

La première consiste à ajouter telles quelles les arêtes fictives au graphe le plus contracté. Contrairement à la phase de contraction, et du fait de la petite taille du graphe à repartitionner, les coûts en termes de mémoire et de calcul sont raisonnables. Il est toutefois nécessaire d'ajouter du code, afin de gérer de manière spécifique les sommets ancres à l'intérieur des algorithmes de bipartitionnement récursif.

La seconde approche s'appuie sur une spécificité de Scotch. En effet, afin d'être en mesure de calculer des placements statiques, du code spécifique permettant de réaliser des bipartitionnement biaisés a déjà été implémenté. Lorsque nous devons décider si un sommet doit être conservé dans la partie courante ou migré vers l'autre partie, les algorithmes de bipartitionnement de Scotch (voir section 1.4.2 de la page 22) prennent en compte la taille de la coupe, mais aussi le fait que changer le sommet de partie peut le rapprocher (ou l'éloigner) des sommets qui ont déjà été placés sur des processeurs distants. Pour pouvoir prendre cela en compte, un *gain externe* est associé à chaque sommet, il représente l'influence de l'environnement extérieur. En ajoutant, pour chaque sommet, la valeur de son coût de migration à ce gain externe, le repartitionnement a pu être intégré à nos algorithmes de bipartitionnement avec un coût de développement très raisonnable. Un exemple d'utilisation du tableau de gain externe, afin de prendre en compte le coût de migration, est illustré par la figure 2.2.

2.5 Adaptation de la phase d'expansion

Lors de la phase d'expansion, la k-partition qui a été calculée sur le graphe le plus contracté va être prolongée de proche en proche sur le graphe le plus fin. À chaque étape, la partie de chaque sommet contracté est associée aux sommets appariés correspondants dans le graphe plus fin. Une fois que la partition du graphe contracté a été prolongée telle quelle sur le graphe plus fin, des algorithmes de raffinement sont utilisés, afin d'adapter la partition à la topologie du nouveau graphe.

Nous allons maintenant décrire les méthodes de raffinement que nous avons implémentées. Il s'agit d'une version k-aire de l'heuristique de Fiduccia-Mattheyses et d'une méthode de diffusion sur graphes bandes. L'extension au repartitionnement k-aire de ces routines de raffinement, déjà présentes dans Scotch pour le calcul des bipartitionnements, constitue une partie importante de notre contribution à la mise en place d'outils de repartitionnement séquentiels.

2.5.1 Algorithme de Fiduccia-Mattheyses k-aire

Cette méthode [San89] est utilisée dans la plupart des partitionneurs de graphes, notamment MeTiS. Elle consiste à déplacer itérativement les sommets du séparateur entre les k parties, dans le but de diminuer la taille de la coupe, tout en s'assurant que l'équilibrage de la charge respecte une valeur de tolérance fournie par l'utilisateur.

7. Cela permet d'avoir à la fois une occupation raisonnable en termes d'espace mémoire et un graphe suffisamment petit pour exécuter des algorithmes coûteux et suffisamment grand pour conserver une bonne précision de l'information.

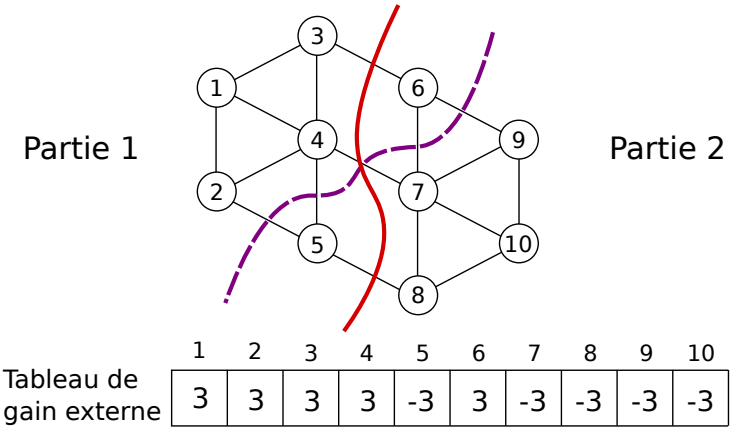


FIGURE 2.2 – Exemple d'utilisation du tableau externe de gain afin de prendre en compte un coût de migration de trois lors du partitionnement d'un graphe dont le poids des arêtes est égal à un. Si l'on ne prenait pas en compte les coûts de migration, la partition optimale serait la partition en traits pleins (rouge), avec une coupe égale à trois. Leur prise en compte induit une coupe de neuf pour la partition en traits pleins (trois, plus le coût de migration des sommets 5 et 6), alors que la partition en traits pointillés (violette), qui devient alors la partition optimale, a une coupe de cinq.

Cette méthode peut être appliquée telle quelle sur un graphe auquel des arêtes fictives auront préalablement été ajoutées. La seule modification à réaliser serait alors d'ajouter des tests pour vérifier que les sommets ancres ne sont jamais déplacés. L'ajout de ces tests se ferait par ailleurs à coût de développement très faible, car ces derniers sont déjà présents pour la gestion des sommets fixes — les sommets ancres seraient alors considérés comme des sommets fixes particuliers.

Bien que la méthode décrite précédemment soit aisée à mettre en place, nous avons préféré, pour des raisons d'optimisation de notre implémentation, gérer les coûts de migration en adaptant l'algorithme. Ainsi, nous modélisons ces derniers par l'ajout d'un coût de migration supplémentaire aux gains associés aux déplacements de sommets vers des parties différentes de leur ancienne partie. Cette approche nous permet d'éviter la perte de mémoire et de temps due à la création d'un graphe avec arêtes fictives. En outre, elle nous permet aussi de gérer un problème qui apparaît, comme nous allons le voir, lorsque l'ancienne partition devient trop déséquilibrée.

Comme nous l'avons vu en section 1.3.4 de la page 12, les algorithmes de type Fiduccia-Mattheyses utilisent une structure de données dédiée, afin de trier les déplacements possibles des sommets selon leur gain. Pour trouver le prochain déplacement potentiel, il suffit d'extraire le sommet suivant de la structure de données, de vérifier qu'il respecte bien les critères d'équilibrage de la charge et, si ce n'est pas le cas, de le retirer de la structure et d'essayer à nouveau. Lorsqu'un sommet respectant les critères d'équilibrage de la charge a été trouvé ou qu'aucun ne convient, les sommets qui en ont été retirés sont remis dans la structure de données.

Bien que cette implémentation fonctionne bien pour le partitionnement de graphes, il peut survenir d'importants ralentissements (temps d'exécution en moyenne deux fois plus long) lorsqu'elle est utilisée pour le repartitionnement. C'est notamment le cas lorsque les coûts de migration sont très élevés devant les poids des arêtes et que les poids des sommets changent beaucoup entre l'ancien graphe (correspondant à l'ancien partitionnement) et le graphe courant. Dans ce cas, comme les coûts de migration sont élevés, l'algorithme va essayer de déplacer le moins

possible de sommets. Cependant, comme les poids des sommets ont beaucoup évolué, il doit quand même en déplacer suffisamment pour équilibrer les poids des parties. En conséquence, la plupart des déplacements de sommets de gains élevés ne pourront être effectués, car même si le déplacement apportait un gain important en termes de coupe, l'équilibrage de la charge obtenu après le déplacement serait encore plus éloigné de la tolérance autorisée qu'auparavant. Pour trouver un déplacement de sommet permettant d'améliorer l'équilibrage de la charge, il est donc nécessaire de parcourir nombre de ces déplacements de sommets de gain élevé, mais impossibles à réaliser car incompatibles avec le respect de la contrainte d'équilibrage de la charge. Étant donné qu'après avoir trouvé un sommet valide, tous les sommets qui ont été parcourus sont réinsérés dans la structure de données et triés selon leur coût, un surcoût important est payé lors de chaque recherche de déplacement de sommet.

Nous avons résolu ce problème de la manière suivante : lorsqu'un déplacement est recherché dans la structure de données, tous les déplacements qui ont été parcourus, mais qui ne respectaient pas les contraintes d'équilibrage de la charge, seront réinsérés dans la structure de données avec un gain modifié, comportant uniquement les informations en termes de coupe : le coût de migration qui avait été ajouté pour prendre en compte les spécificités du repartionnement est retiré du gain utilisé pour trier la structure de données. Ainsi, ces déplacements de sommets aux coûts de migration élevés seront placés plus loin dans la structure de données. Ils ne seront donc plus parcourus à chaque recherche de déplacement et, en conséquence, n'induiront plus de surcoût. Par la suite, si ces déplacements de sommets sont de nouveau extraits de la structure, alors leur coût de migration sera de nouveau pris en compte pour le calcul du gain. Par ailleurs, à chaque nouvelle passe de notre algorithme (c'est-à-dire à chaque nouvelle itération de la boucle la plus globale), tous les gains sont recalculés et tous les coûts de migration sont donc de nouveau pris en compte lors de l'ajout des déplacements de sommets dans la structure de données. Comme nous le verrons en section 2.7.1.2 de la page 52, cette optimisation permet d'économiser en temps d'exécution, tout en conservant des partitions d'une qualité très proche de celles obtenues avec la version standard de l'algorithme.

Les algorithmes de type Fiduccia-Mattheyses calculent de bonnes partitions tout en préservant l'équilibrage de la charge. Pourtant, ils ont deux faiblesses. Premièrement, comme ils ne réalisent que des optimisations locales, ils peuvent calculer des partitions dont les frontières, qui seraient composées seulement d'une juxtaposition de segments localement optimaux, ne seraient pas globalement optimales. Deuxièmement, ces algorithmes ne se parallélisent pas efficacement. Ces algorithmes étant par nature itératifs, les déplacements de sommets ne peuvent être réalisés en parallèle, car chaque déplacement de sommet nécessite de recalculer les gains des sommets voisins.

C'est dans l'optique de dépasser ces deux défauts qu'une seconde méthode de raffinement a été ajoutée à Scotch : une méthode globale de raffinement par diffusion sur graphes bandes.

2.5.2 Graphes bandes

Dans notre implémentation k-aire du graphe bande (voir section 1.3.4.2 de la page 15), la création de ce dernier n'est possible que si, pour chaque partie, il existe au moins un sommet à distance d de la frontière. Avec cette implémentation, il n'est pas possible de créer de graphe bande sur les graphes possédant un degré moyen élevé et découpés en de nombreuses parties. Bien que nous n'ayons pas eu le temps de le faire, rendre possible cette création en reliant, dans chaque partie, les ancres aux sommets qui sont à la distance maximale de la frontière — pouvant être inférieure à d — est une piste de travail intéressante, pour améliorer cette implémentation. Nous avons fait le choix de considérer, par défaut, des graphes bandes de taille 3.

Alors que l'algorithme de diffusion, que nous allons étudier en section 2.5.3, est en général utilisé sur un graphe bande (exception faite du cas où l'utilisateur fournit lui-même un graphe avec des ancres), nous avons conçu notre implémentation k-aire de l'algorithme Fiduccia-Mattheyses de sorte qu'elle puisse être exécutée, soit sur le graphe complet, afin d'explorer un large ensemble de déplacements, soit sur un graphe bande, pour obtenir de bons résultats, en un temps plus court. Cette dernière option est le choix par défaut que nous avons retenu dans la stratégie de repartitionnement séquentielle de Scotch.

2.5.3 Diffusion

Étant donné que notre objectif est d'employer cet algorithme (voir section 1.3.4.2 de la page 15) pour raffiner une frontière préexistante proche de la frontière optimisée, l'utiliser sur les graphes bandes nous permet d'obtenir un résultat équivalent, tout en consommant moins de ressources. Lorsque cet algorithme est utilisé sur un graphe bande, la quantité de liquide (la quantité de calcul à réaliser) ajoutée à chaque pas de temps à chaque partie apparaît dans les sommets ancre, en quantité proportionnelle au poids optimal des parties.

Le schéma de la figure 2.3(a) de la page 48 illustre la manière dont la version k-aire de cet algorithme fonctionne dans le cas du partitionnement. Comme le sommet 8 a un poids plus important, il perd plus de liquide par pas de temps que les autres. Ainsi, même s'il obtient beaucoup de liquide du sommet 10, il va n'en transmettre qu'une petite fraction au sommet 5.

Nous allons maintenant décrire comment nous avons adapté cet algorithme au problème du repartitionnement.

L'exécution de cet algorithme sur un graphe bande auquel des arêtes fictives ont été ajoutées n'apporte pas le résultat escompté lorsque les coûts de migration sont élevés. En effet, le liquide ne va cesser de faire des aller-retours entre les sommets ancres et les sommets du graphe. Seule une faible quantité de liquide traversera la frontière et, sa couleur étant minoritaire, disparaîtra. En conséquence, l'algorithme ne va converger que faiblement et la partition ne sera que très peu raffinée.

Afin de prendre en compte les coûts de migration de manière correcte et d'être en mesure d'adapter cet algorithme au problème du repartitionnement, nous nous sommes inspirés du *modèle d'influence*. Dans ce modèle de diffusion, exploré par Wan *et al.* [WRSL05], les sommets influencent leurs voisins en diffusant des informations à propos de leur état courant.

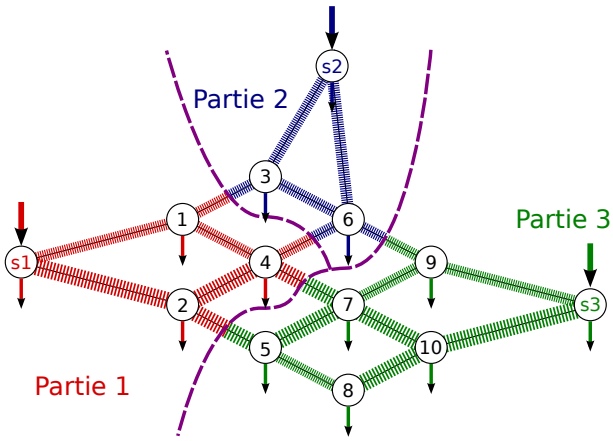
Le fonctionnement de la diffusion k-aire ainsi étendue est présenté sur l'algorithme 6 de la page ci-contre. Deux exemples de fonctionnement de cet algorithme étendu sont présentés sur les figures 2.3(b) et 2.3(c) de la page 48. Dans le premier exemple, la partie courante de tous les sommets est la même que leur ancienne partie ; dans le second, le sommet 5 est dans une partie différente de son ancienne partie. Dans les deux cas, si un sommet possède des voisins appartenant à son ancienne partie ou une autre partie que sa partie courante, il peut rediriger une partie du liquide qui le traverse, afin de contribuer à la diminution de la migration globale. En pratique, une partie du *liquide standard* (le liquide utilisé dans la version partitionnement de l'algorithme) qui traverse ces sommets, proportionnellement à leur coût de migration, est convertie en *liquide de migration*. Ce liquide de migration sera diffusé vers les voisins qui favoriseront le retour du sommet vers son ancienne partie. Si le sommet va diffuser du liquide de son ancienne partie (la nouvelle partie courante du sommet est la même que son ancienne partie), le liquide de migration va être diffusé vers les sommets de son ancienne partie ; sinon, le liquide ira vers les sommets appartenant à des parties différentes de son ancienne partie. Sur le schéma de la figure 2.3(b), tous les sommets sont dans leur ancienne partie. En conséquence, le liquide de migration est conservé dans chaque partie, plutôt que d'être perdu à la frontière. Sur celui

Algorithme 6: Algorithme de diffusion k-aire étendu au problème du repartitionnement.

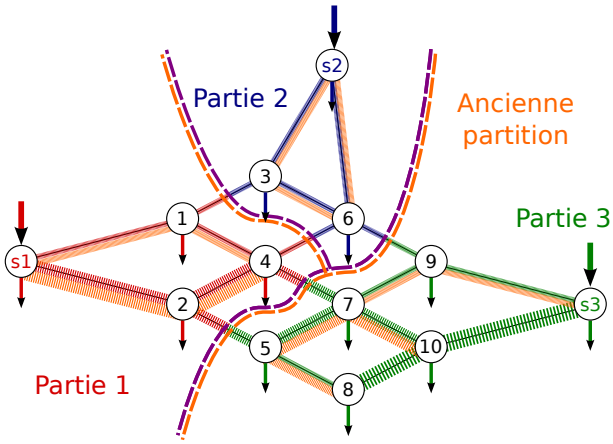
Entrée : G : graphe
o : ancienne partition de G
Π : partition courante de G
p_m : nombre de passes à réaliser
Sorties : Π : partition raffinée de G

$d_t(v)$, $d_{t+1}(v)$: valeur de diffusion courante et future du sommet v
$\Pi_t(v)$, $\Pi_{t+1}(v)$: partie courante et future du sommet v
$m_{o_t}(v)$, $m_{o_{t+1}}(v)$: valeur de diffusion du liquide migration courante et future vers les sommets actuellement dans l'ancienne partie du sommet v
$m_{d_t}(v)$, $m_{d_{t+1}}(v)$: valeur de diffusion du liquide migration courante et future vers les sommets actuellement dans des parties différentes de l'ancienne partie du sommet v
s_π : sommet source de la partie π
$l(\pi)$: tableau temporaire contenant la contribution en termes de liquide des voisins actuellement dans la partie π

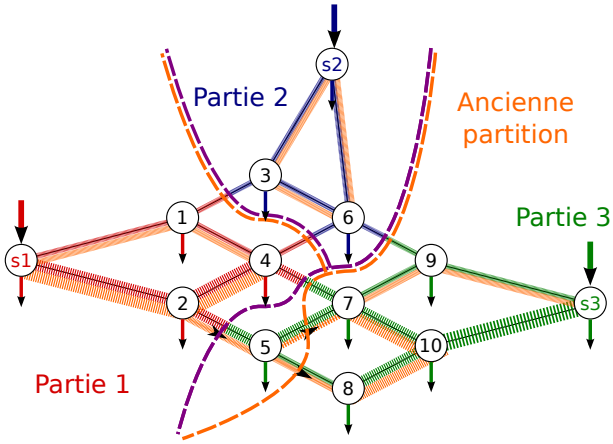
Initialiser d_t, d_{t+1}, m_{o_t}, $m_{o_{t+1}}$, m_{d_t} et $m_{d_{t+1}}$ à 0 ▷ Initialisation
Initialiser Π_t à Π
tant que *nombre de passes réalisées* $< p_m$ **faire**
  pour *les k-parties* π **faire** ▷ Rechargement des sommets source
    $\sigma_{s_\pi} \leftarrow \sum_{e=(s_\pi,v)} w_e$ ▷ Somme des poids de toutes les arêtes partant des sommets source
    $d_t(s_\pi) \leftarrow d_t(s_\pi) + \frac{|V|}{k \times \sigma_{s_\pi}}$ ▷ Mise à jour diffusion des sommets source
  pour *les sommets* v *du graphe* **faire**
    Réinitialiser l à 0
    pour *les arêtes* $e = (v, v')$ *du graphe* **faire** ▷ Calcul de la contribution des voisins
      $d \leftarrow d_t(v')$ ▷ Récupérer la contribution standard du voisin
      si $\Pi_t(v') = o(v)$ **alors** ▷ Ajouter sa contribution en termes de migration
        $d \leftarrow d + m_{o_t}(v')$
      sinon
        $d \leftarrow d + m_{d_t}(v')$
      $l(\Pi_t(v')) \leftarrow l(\Pi_t(v')) + d$ ▷ Ajout de sa contribution
    $\pi_{\max} \leftarrow \max(l)$ ▷ Récupérer la partie avec le plus de liquide
    $\Pi_{t+1}(v) \leftarrow \pi_{\max}$ ▷ Mise à jour de la partie du sommet courant
    $d \leftarrow l(\pi_{\max})$ ▷ Récupérer la quantité de liquide la plus abondante
    $d \leftarrow d - w_v$ ▷ Perte de liquide du sommet courant
    $\sigma \leftarrow \sum_{e=(v,v')} w_e$ ▷ Somme des poids des arêtes adjacentes à v
    $\sigma_o \leftarrow \sum_{e=(v,v') \text{ et } \pi(v')=o(v)} w_e$ ▷ Somme des poids des arêtes allant vers $o(v)$
    $m \leftarrow 0$
    si *(*$\sigma_o \neq 0$*) et (*$\sigma_o \neq \sigma$*)* **alors** ▷ Si la redirection de liquide peut diminuer la migration
      $m \leftarrow m_c(v)$ ▷ Récupérer le coût de migration du sommet
      $m \leftarrow \frac{d}{\sigma+m} m$ ▷ Le normaliser
      $d \leftarrow d - m$ ▷ Le retirer de la quantité de liquide standard
      si $o(v) = \Pi_{t+1}(v)$ **alors** ▷ S'il va diffuser du liquide de son ancienne partie
        $m_{o_{t+1}}(v) \leftarrow \frac{m}{\sigma_o}$ ▷ Le liquide de migration va aller vers les sommets dans l'ancienne partie
        $m_{d_{t+1}}(v) \leftarrow 0$
      sinon ▷ S'il va diffuser du liquide différent de celui de son ancienne partie
        $m_{o_{t+1}}(v) \leftarrow 0$
        $m_{d_{t+1}}(v) \leftarrow \frac{m}{\sigma - \sigma_o}$ ▷ Le liquide de migration va aller vers les autres parties
    sinon ▷ Le sommet courant ne diffusera pas de liquide de migration
      $m_{o_{t+1}}(v) \leftarrow 0$
      $m_{d_{t+1}}(v) \leftarrow 0$
    $d_{t+1}(v) \leftarrow \frac{d}{\sigma}$ ▷ Mise à jour de la quantité de liquide standard
  $d_t \leftarrow d_{t+1}$ ▷ Mise à jour des valeurs courantes
  $\Pi_t \leftarrow \Pi_{t+1}$
  $m_{o_t} \leftarrow m_{o_{t+1}}$
  $m_{d_t} \leftarrow m_{d_{t+1}}$
$\Pi \leftarrow \Pi_t$ ▷ Retour de la partition raffinée



(a) Partitionnement.



(b) Repartitionnement avec tous les sommets dans leur partie d'origine.



(c) Repartitionnement dans le cas général. Le sommet 5, dont la partie d'origine est 1, se trouve dans la partie 3.

FIGURE 2.3 – Extension de l'algorithme de diffusion au problème de repartitionnement. Trois *liquides standard* (rouge, bleu et vert) apparaissent (cette apparition de liquide est représentée par les grosses flèches) à partir des sommets source `s1`, `s2` et `s3` ; leur quantité est proportionnelle à la taille optimale de leur partie. Tous les sommets qui ne sont pas des sommets source perdent à chaque étape du liquide en quantité proportionnelle à leur poids (cette perte de liquide est représentée par les petites flèches). Les lignes en pointillés représentent les flux de liquide allant d'un sommet à un autre. L'épaisseur des lignes est proportionnelle à la quantité de liquide traversant l'arête. Le liquide orange correspond au *liquide de migration*, c'est-à-dire à la fraction du *liquide standard* qui a été redirigée de sorte à favoriser une diminution globale du coût de migration.

de la figure 2.3(c), le sommet 2 donne une partie de son liquide rouge au sommet 5 pour l'aider à revenir dans sa partie d'origine. Le sommet 5, quant à lui, ne transmet pas de liquide vert au sommet 2, de façon à augmenter sa probabilité de retour vers son ancienne partie. Comme cela n'aidera pas le sommet 5 à retourner dans son ancienne partie, les sommets 7 et 8 ne lui donnent pas de leur liquide de migration.

L'avantage du raffinement par diffusion est qu'il fournit une bonne coupe, tout en prenant en compte les contraintes d'équilibrage de la charge (modélisées par la perte de liquide de chaque sommet à chaque pas de temps). En outre, il est global, facilement parallélisable et *scalable*.

Il a toutefois trois défauts : il est plus coûteux que les heuristiques de type Fiduccia-Mattheyses, la tolérance de l'équilibrage de la charge ne peut pas être fixée à une valeur arbitraire (comme nous le verrons par la suite, elle est proche de 5 % en moyenne) et il ne peut être utilisé que si un graphe bande k-aire a pu être créé.

2.6 Adaptation des algorithmes au replacement

La plupart des algorithmes de repartitionnement s'adaptent aisément au problème du replacement. Il n'y a rien à faire pour les algorithmes de bipartitionnement, car les spécificités dues au placement et au repartitionnement sont toutes les deux prises en compte au moyen des gains externes (voir les sections 1.4.2 de la page 22 et 2.4 de la page 43). En ce qui concerne l'heuristique de Fiduccia-Mattheyses k-aire, il suffit de prendre en compte la notion de distance lors du calcul du gain apporté par le déplacement d'un sommet. L'adaptation de l'algorithme de diffusion se réalise elle aussi en prenant en compte la notion de distance, lors du calcul des contributions en termes de liquides des sommets voisins (dans la boucle de la ligne 15 de l'algorithme 6 de la page 47). Bien qu'il n'y ait qu'une quantité réduite de code à modifier, par manque de temps, nous avons choisi de ne pas finaliser l'adaptation de l'algorithme de diffusion au problème du replacement.

2.7 Résultats expérimentaux pour le repartitionnement séquentiel

Nous présenterons dans un premier temps l'analyse quantitative des algorithmes implémentés, puis nous nous intéresserons aux résultats obtenus dans le cadre d'une collaboration avec l'équipe du projet Charm++ [KK93] de l'université d'Urbana-Champaign.

2.7.1 Analyse quantitative

Les résultats qui vont suivre ont été calculés sur la plate-forme de test PLAFRIM mise en place dans le cadre de l'*action de développement INRIA PLAFRIM* avec le soutien du LaBRI, de l'IMB et des entités suivantes : le Conseil Régional d'Aquitaine, le FeDER, l'Université de Bordeaux et le CNRS (voir `https://plafrim.bordeaux.inria.fr/`). Chacun des nœuds de la plate-forme PLAFRIM comprend deux processeurs Intel® Nehalem Xeon® X5550 quadri-cœurs cadencés à 2,66 GHz et 24 Gio de mémoire centrale. Ces résultats ont fait l'objet d'une publication [FP11] et d'un rapport de recherche [FP13].

TABLE 2.1 – Description des graphes de test. Les cardinalités des sommets et des arêtes, $|V|$ et $|E|$, sont données en milliers.

Graphe	Description	Taille ($\times 10^3$)		Degré moyen
		$\|V\|$	$\|E\|$	
`10millions`	Électromagnétisme 3D, CEA	10423	78649	15,09
`af_shell10`	Mécanique des structures	1508	25582	33,93
`audikw_1`	Mécanique des structures	943	38354	81,28
`cage15`	Électrophorèse d'ADN	5154	47022	18,24
`conesphere1m`	Électromagnétisme 3D, CEA	1055	8023	15,21
`coupole8000`	Mécanique des structures 3D, CEA	1768	41656	47,12
`dielFilterV3real`	Électromagnétisme	1102	44101	79,98
`ecology1`	Problème 2D/3D	1000	1998	4,00
`ldoor`	Mécanique des structures	952	22785	47,86
`thermal2`	Thermodynamique	1228	3676	5,99

2.7.1.1 Protocole

Les graphes que nous avons utilisés dans les expériences décrites ci-après proviennent de plusieurs domaines applicatifs. Les poids de leurs sommets et de leurs arêtes sont égaux à 1. Leurs caractéristiques sont détaillées au tableau 2.1.

Nous avons considéré les quatre métriques suivantes :

- la *coupe* : soit $G = (V, E)$ un graphe, w_e le poids d'une arête $e \in E$ et S l'ensemble des arêtes appartenant au séparateur. La coupe sont calculée de la manière suivante : $\frac{\sum_{s \in S} w_s}{\sum_{e \in E} w_e}$. La coupe considérée dans cette section est donc égale à la coupe définie au chapitre 1 divisée par la somme des poids des arêtes du graphe ;
- le *déséquilibre* ;
- le taux de *migration* : nous considérons une ancienne partition et un repartitionnement, soit V_m l'ensemble des sommets qui appartiennent dans le repartitionnement à une partie différente de leur ancienne partie. Le taux de migration (ou fraction de sommets migrés) est alors défini de la manière suivante : $\frac{|V_m|}{|V|}$. Nous l'exprimerons sous la forme d'un pourcentage ;
- le *temps* d'exécution : cette durée est exprimée en secondes. SCOTCH étant un logiciel séquentiel, il est exécuté sur un processeur. Étant donné que METIS ne propose pas de fonctionnalités de repartitionnement et que nous n'avons pas réussi à faire fonctionner PARMETIS sur un seul processeur, nous utilisons PARMETIS sur deux processeurs. Pour des raisons de clarté, nous donnerons les temps tels quels, même si PARMETIS tire parti de l'accélération parallèle sur ces deux processeurs.

Nous appellons *coût de migration* le coût de migration global fourni pour tous les sommets du graphe tel que défini à la section 2.2.2 de la page 40.

Nous avons évalué les stratégies de repartitionnement séquentielles au moyen du protocole suivant :

1. Nous calculons une *partition initiale* de 128 parties avec la stratégie par défaut de SCOTCH (`sq`) et une contrainte d'équilibrage de la charge de 0,05 (c'est-à-dire 5 %) [8].

8. Cette contrainte d'équilibrage de la charge correspond pour PARMETIS au paramètre `ubvec` dont la valeur

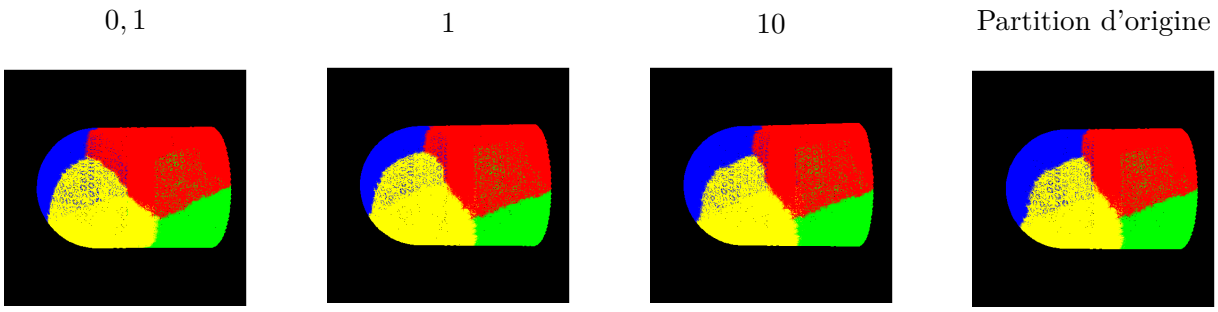


FIGURE 2.4 – Plusieurs images du graphe `altr4` repartitionné en quatre parties avec SCOTCH en prenant les valeurs 0,1, 1 et 10 comme coût de migration et un augmentant le poids d'un sommet de la partie bleue d'un quart du poids total du graphe.

2. Nous calculons un *graphe modifié* en augmentant de un les poids des sommets qui appartiennent aux 32 premières parties de la *partition initiale*. Ceci conduit à une augmentation du déséquilibre qui atteint, en moyenne, 0,16.
3. Nous utilisons plusieurs stratégies pour calculer un repartitionnement, toujours sur 128 parties, de la *partition initiale* sur le *graphe modifié*, tout en spécifiant une contrainte d'équilibrage de la charge de 0,05. Pour tous les graphes et toutes les stratégies considérés, cette étape est réalisée 100 fois avec, à chaque fois, un coût de migration différent variant entre 0,5 et 50. Nous incrémentons le coût de migration avec un pas de 0,01 lorsqu'il est inférieur à 1 puis avec un pas de 1 lorsqu'il est supérieur à 1. Ce coût de migration m correspond pour PARMETIS au paramètre `itr`, dont la valeur est calculée de la manière suivante : `itr` $= \frac{1}{m}$.

Un exemple visuel de l'évolution du repartitionnement en fonction du coût de migration est illustré par la figure 2.4.

Nous avons considéré les stratégies suivantes :

- `rb` : le calcul de la partition initiale du schéma multi-niveaux k-aire est réalisé par *bipartitionnement récursif*, la phase d'expansion est réalisée sans raffinement. Cette stratégie sera utilisée comme point de comparaison afin d'évaluer l'apport des différentes stratégies de raffinement ;
- `rbf` : le calcul de la partition initiale du schéma multi-niveaux k-aire est réalisé par *bipartitionnement récursif*. Lors de la phase d'expansion, la version *basique* de notre implémentation k-aire de l'heuristique de *Fiduccia-Mattheyses* est utilisée (voir section 2.5.1 de la page 43). L'usage de cette stratégie devrait apporter un surcoût en temps important ;
- `rf` : le calcul de la partition initiale du schéma multi-niveaux k-aire est réalisé par *bipartitionnement récursif*. Lors de la phase d'expansion, la version optimisée de notre mise en œuvre k-aire de l'heuristique de *Fiduccia-Mattheyses* est utilisée. Cette stratégie nous permettra d'évaluer l'apport de l'optimisation réalisée en section 2.5.1 de la page 43. Elle sera aussi un point de comparaison avec le raffinement par diffusion seulement ;
- `rd` : le calcul de la partition initiale du schéma multi-niveaux k-aire est réalisé par *bipartitionnement récursif*. Lors de la phase d'expansion, l'algorithme de raffinement par *Diffusion* est utilisé (cf. section 2.5.3 de la page 46). L'algorithme de diffusion étant par nature *scalable*, l'analyse de cette stratégie nous permettra d'avoir une première validation de son usage futur pour le repartitionnement parallèle ;

a été fixée à 1,05.

– `sq` : c'est la stratégie par défaut de Scotch lorsqu'on utilise la routine `SCOTCH_graph Remap`[9]. Cette dernière privilégie la coupe (*Quality*) plutôt qu'un équilibrage de la charge strict[10]. La partition initiale du schéma multi-niveaux k-aire est calculée par bipartitionnement récursif. Lors de la phase d'expansion, un raffinement sur graphe bande est réalisé en appelant l'algorithme de diffusion, puis la version optimisée de l'heuristique de Fiduccia-Mattheyses k-aire. Cette stratégie, qui combine deux algorithmes de raffinement, nous permettra de quantifier dans quelle mesure l'absence de l'utilisation de l'heuristique de Fiduccia-Mattheyses a un impact sur le raffinement ;
– `pm` : c'est la stratégie par défaut de *ParMetis* 4.0.2 [kar], en utilisant la routine `ParMETIS_V3_AdaptiveRepart`.

2.7.1.2 Comportement de l'heuristique de Fiduccia-Mattheyses

Sur la figure 2.5 de la page suivante, nous pouvons observer le comportement des stratégies de type Fiduccia-Mattheyses sur le graphe `10millions`. La partition calculée avec la stratégie `pm` a une coupe de qualité légèrement supérieure et une valeur de déséquilibre un peu au dessus de la tolérance pour des coûts de migration supérieurs à 1. `rbf` et `rf` sont plus sensibles aux variations du coût de migration que `pm`. En termes de temps d'exécution, `rf` est un peu plus coûteux que `pm` et, comme expliqué en section 2.5.1 de la page 43, `rbf` est la stratégie la plus coûteuse en temps. Comme `rbf` explore, à chaque itération, un ensemble de déplacements de sommets plus grand, il est capable d'atteindre un meilleur équilibrage de la charge. La probabilité pour `rbf` de trouver, parmi les déplacements de gains élevés déjà parcourus, un candidat respectant la contrainte d'équilibrage de la charge est faible. Elle est par ailleurs fonction de la manière dont l'espace des solutions va être exploré et donc du coût de migration. Le choix d'effectuer ces déplacements de gains élevés va en outre modifier de manière importante les choix de déplacements futurs. En conséquence, même si leur acceptabilité est peu fréquente, la prise en compte des déplacements aux coûts élevés déjà parcourus peut changer de manière importante le sous-ensemble de l'espace des solutions exploré. Cela explique la forte dispersion des valeurs obtenues avec la stratégie `rbf` en termes de déséquilibre et de temps pour des coûts de migration élevés.

Sur la figure 2.6 de la page 54, nous pouvons observer la répartition par stratégie de tous nos résultats. Ce graphique confirme nos premières observations. Les deux stratégies `rbf` et `rf` produisent des partitions possédant une coupe et un nombre de sommets migrés proches. `rbf` peut être très coûteuse, mais fournit un meilleur équilibrage de la charge. `pm` est la stratégie la plus rapide ; elle produit des partitions avec une meilleure coupe, mais un moins bon équilibrage de charge. Le pourcentage de sommets migrés lorsqu'on utilise la stratégie `pm` varie entre 40 % et 45 % pour les coûts de migration inférieurs à 1, puis reste proche de 40 %[11].

Étant donné que `rf` respecte l'équilibrage de la charge et donne une coupe proche de `rbf`, tout en étant plus rapide, nous allons considérer, par la suite, uniquement cette implémentation.

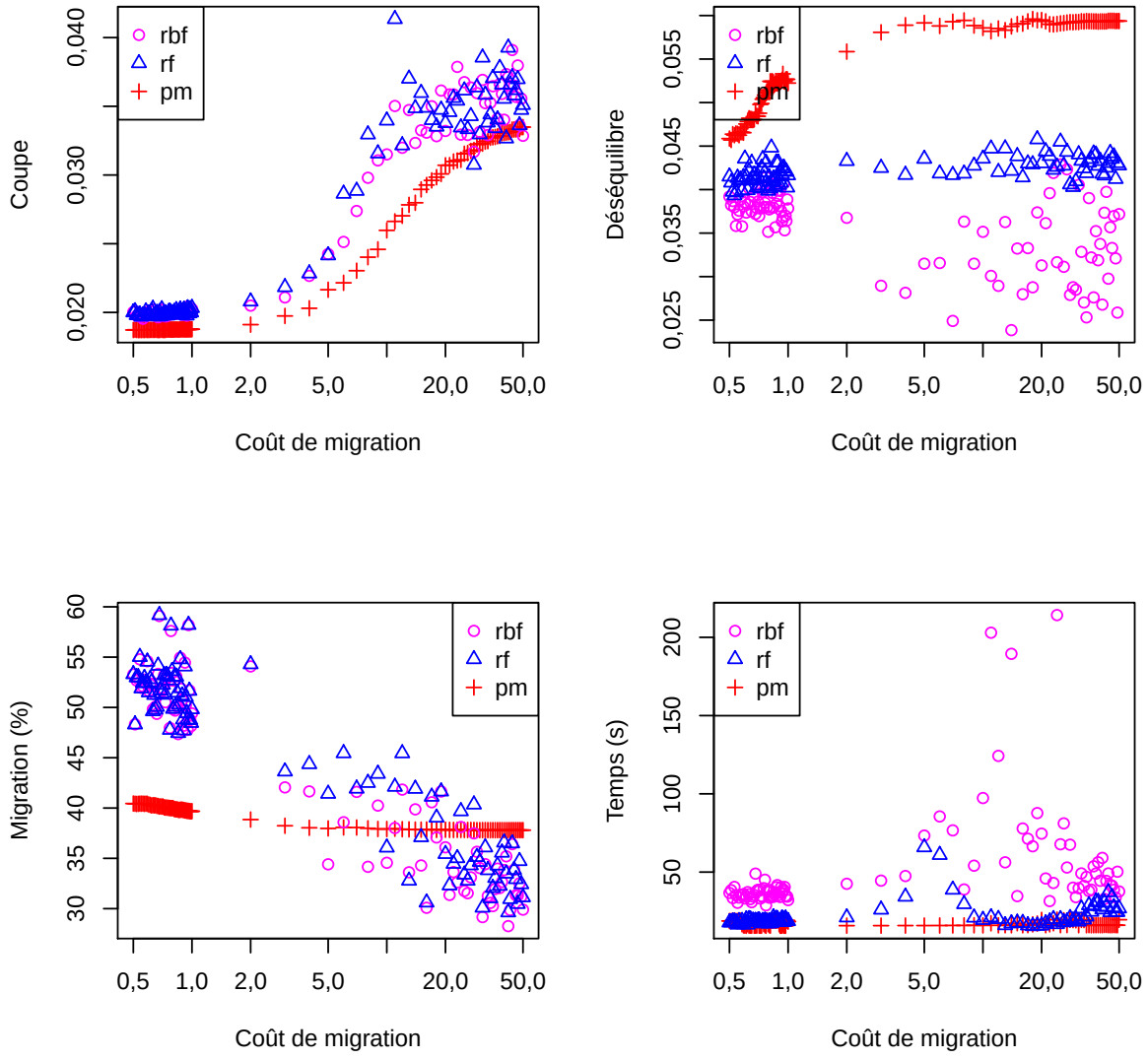


FIGURE 2.5 – Comportement, sur le graphe `10millions` et pour les quatre indicateurs considérés, des stratégies de type Fiduccia-Mattheyses k-aire de base (`rbf`) et optimisée (`rf`) de SCOTCH, ainsi que de la stratégie par défaut de PARMETIS (`pm`).

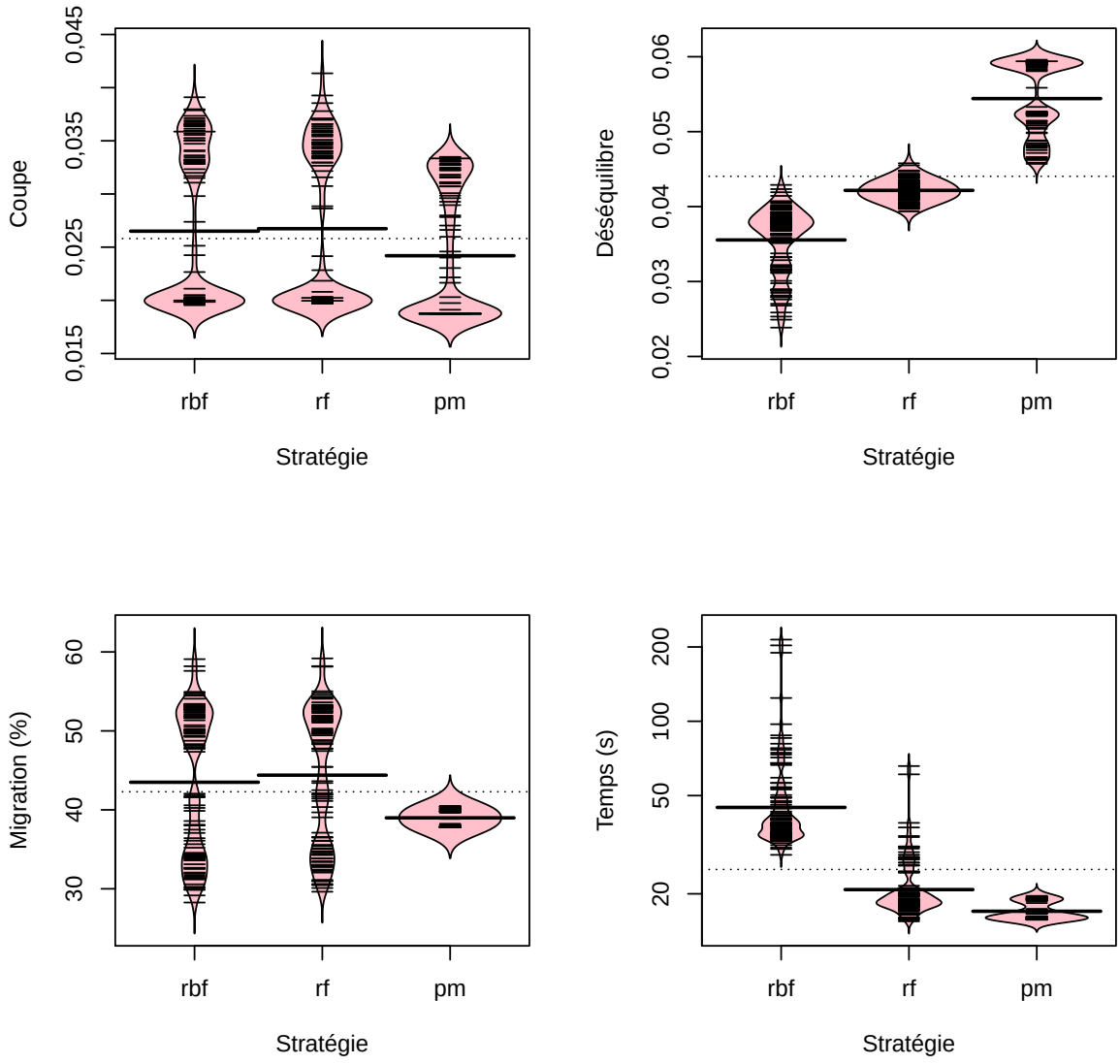


FIGURE 2.6 – Ces graphiques en haricot [Kam08] donnent, pour tous nos résultats avec les stratégies `rbf`, `rf` et `pm`, une estimation de la densité locale pour les quatre indicateurs considérés. La courbe symétrique, qui fait le contour de la forme rose, correspond au tracé d'une estimation non paramétrique de la densité. La ligne horizontale en pointillés représente la valeur moyenne globale pour toutes les stratégies. Les lignes horizontales épaisses représentent les moyennes de indicateurs pour chaque stratégie. Les petites lignes horizontales permettent de connaître les valeurs obtenues par individu. Lorsque l'échelle de l'axe des y est logarithmique, les moyennes présentées sont géométriques.

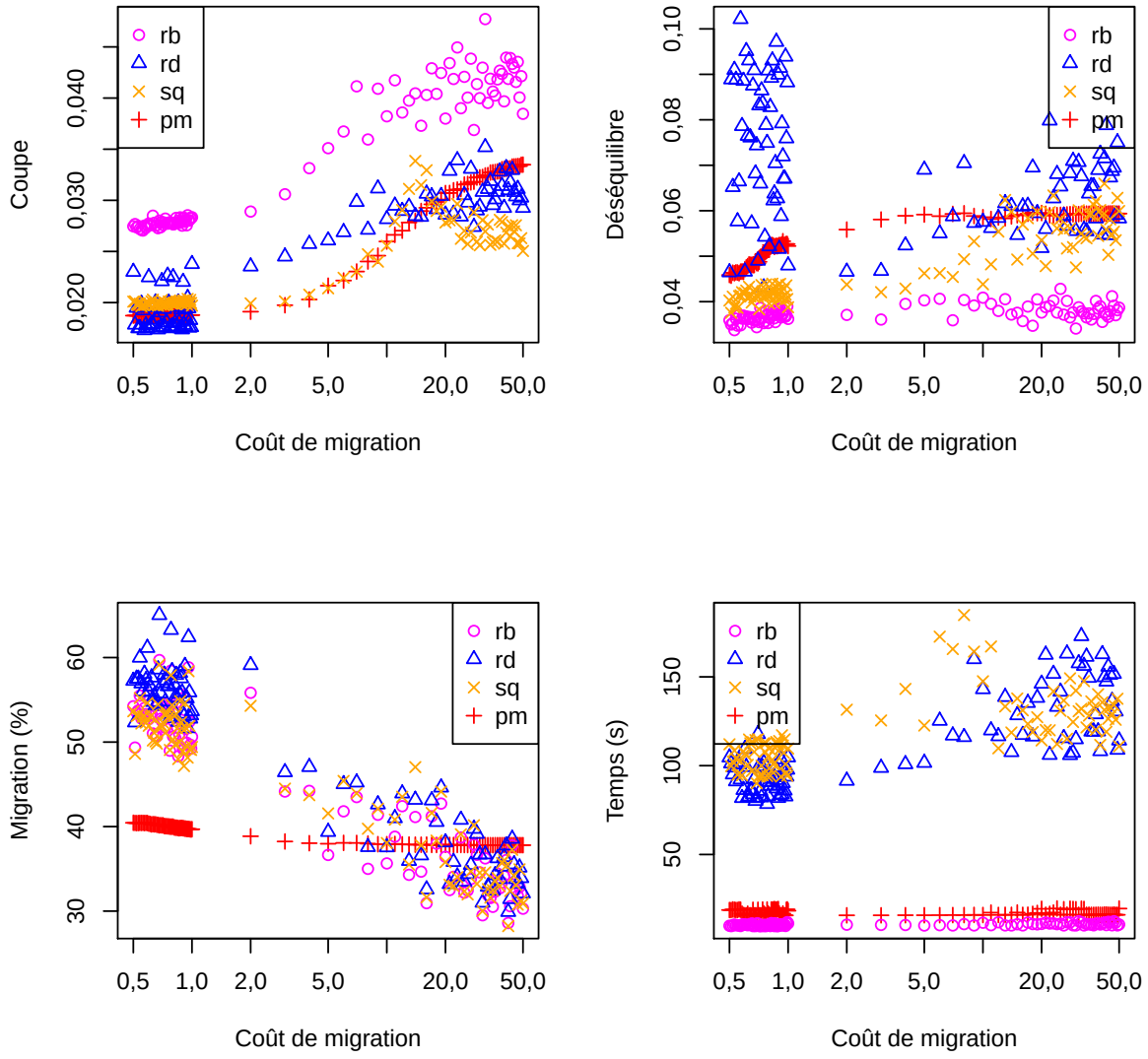


FIGURE 2.7 – Comportement, sur le graphe `10millions` et pour les quatre indicateurs considérés, de la stratégie sans raffinement (`rb`), de la stratégie avec raffinement par diffusion (`rd`) et des stratégies par défaut de SCOTCH (`sq`) et de PARMETIS (`pm`).

2.7.1.3 Comportement du raffinement par diffusion k-aire

Sur la figure 2.7 de la page précédente, nous pouvons observer, sur le graphe `10millions`, le comportement de la stratégie de raffinement par diffusion (`rd`) en comparaison avec les stratégies `sq`, `pm` et la stratégie de base `rb`. La stratégie `rd` produit des partitions avec une coupe qui est moins sensible à la variation du coût de migration. Elle ne respecte pas toujours la contrainte d'équilibrage de la charge et est plus coûteuse, en termes de temps d'exécution, que les stratégies de type Fiduccia-Mattheyses. À l'instar des stratégies `rf` et `sq`, elle est plus sensible au coût de migration que la stratégie `pm`. `sq`, qui combine les raffinements des stratégies `rd` et `rf`, produit une coupe proche de `pm`, qui est de meilleure qualité, pour des coûts de migration élevés et un meilleur équilibrage de la charge, mais est plus coûteuse, en termes de temps d'exécution.

Comme l'algorithme de diffusion prend en compte la contrainte d'équilibrage de la charge au moyen de la perte d'une partie du liquide qui traverse les sommets, la valeur du déséquilibre final ne peut être contrainte par une valeur arbitraire. Nous avons mesuré expérimentalement une valeur moyenne d'équilibrage de la charge proche de $0,05$.

`rd` a le même comportement sur les autres graphes, excepté sur `audikw_1`, `cage15` et `dielFilterV3real`. Sur ces trois graphes à grands degrés, notre algorithme de création de graphes bandes k-aires ne parvient pas à produire un graphe utilisablei, car il ne peut trouver, pour chaque partie, au moins un sommet à distance 3 de la frontière (cf. section 2.5.2). Comme le graphe bande ne peut être créé, `rd` se comporte exactement comme la stratégie avec expansion sans raffinement `rb`.

Pour la clarté de notre analyse, nous allons par la suite nous intéresser uniquement aux autres graphes, que nous allons appeler : graphes de type « maillage ».

2.7.1.4 Analyse des stratégies

Migration Sur la figure 2.8 de la page suivante, nous pouvons observer que toutes les stratégies de Scotch sont plus sensibles au coût de migration que la stratégie de ParMeTiS. La stratégie `rd` migre toujours plus que les autres. En comparaison de `rb` et `rf`, `sq` migre plus pour les petits coûts de migration et moins pour des coûts de migration élevés. Pour les coûts de migration élevés, nous remarquons que toutes les stratégies migrent un pourcentage des sommets proche de 40 %.

Coupe et équilibrage de la charge Sur les figures 2.9 de la page 58 et 2.10 de la page 59, nous pouvons voir un résumé des résultats obtenus en termes de coupe et d'équilibrage de la charge pour toutes les stratégies considérées. Comme attendu, la moins bonne coupe ($0,0361$) est obtenue avec la stratégie sans raffinement : `rb`. Comme la partition initiale est équilibrée avant la phase d'expansion sans raffinement, `rb` donne un bon équilibrage ($0,0366$). `pm` donne la meilleure coupe ($0,0289$) et `rf` donne le meilleur équilibrage ($0,0340$). `sq` donne une coupe proche de celle obtenue avec `pm` ($0,0297$, avec un ratio — défini dans la légende de la figure 2.9 — de $1,028$), tout en donnant un meilleur équilibrage de la charge ($0,0435$, avec un ratio de $0,852$). En moyenne, `pm` dépasse légèrement (de $0,001$) la contrainte d'équilibrage de la charge de $0,05$;

9. Cette stratégie peut être activée en utilisant l'option `SCOTCH_STRATQUALITY`.
10. Cette stratégie peut être activée en utilisant l'option `SCOTCH_STRATBALANCE`.
11. Nous avons expliqué en section 2.7.1.1 la manière dont nous avons converti le coût de migration afin qu'il soit pris en compte par ParMeTiS via le paramètre `itr`. Nous avons utilisé, pour obtenir nos résultats, des valeurs du paramètre `itr` comprises dans l'intervalle $[0,02; 2,0]$. Dans la documentation de ParMeTiS, il est indiqué que l'on peut utiliser des valeurs d'`itr` comprises dans l'intervalle $[0,000001; 1000000]$. Nous avons essayé ces valeurs et nous n'avons pas observé de changement notable du nombre de migrations.

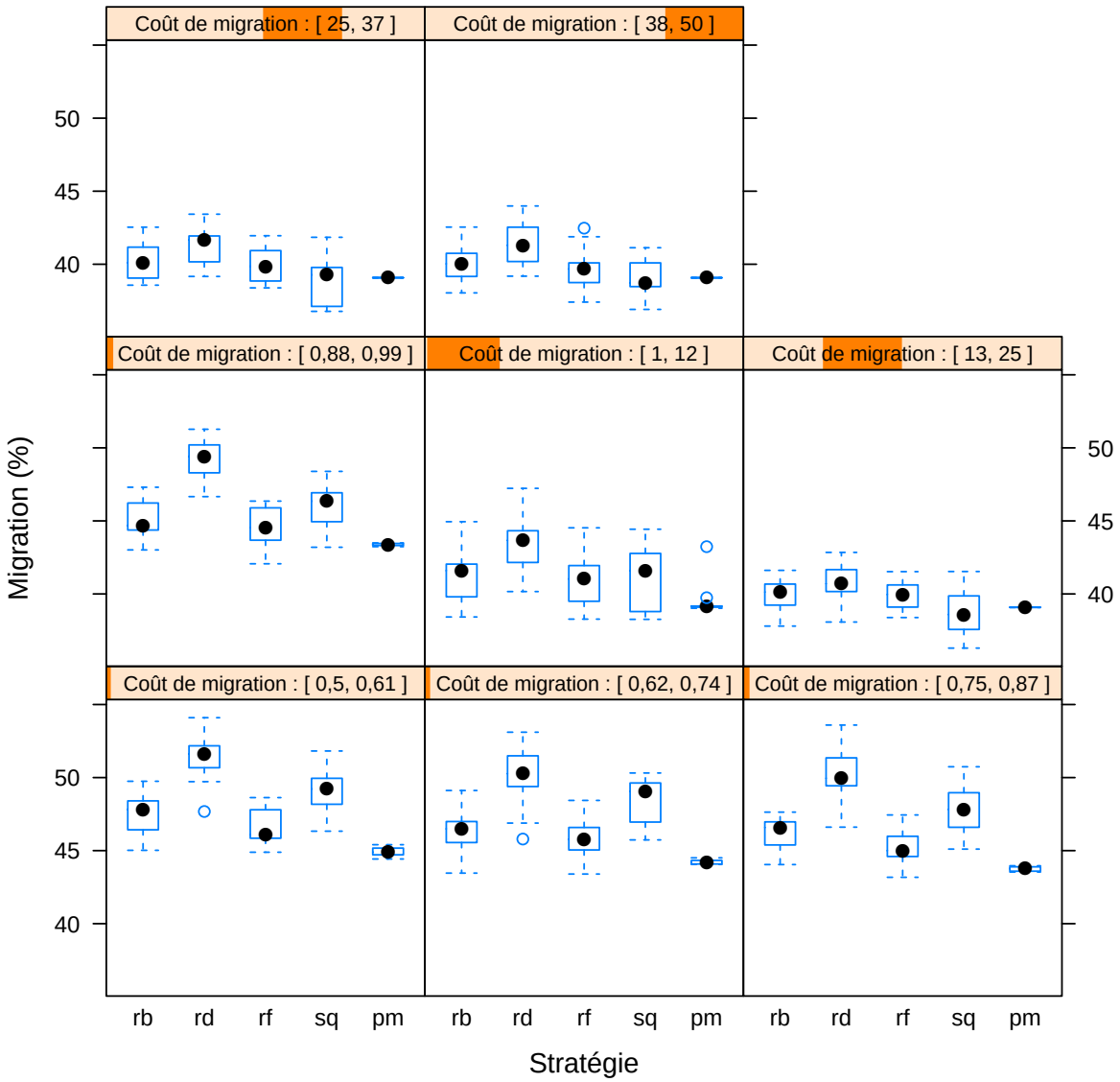


FIGURE 2.8 – Graphique de « boîte à moustaches » en treillis [Sar08] (c'est-à-dire une grille de plusieurs graphiques de « boîtes à moustaches ») montrant, pour les graphes de type « maillage », le comportement en termes de migration de toutes les stratégies. Chacun des 8 graphiques correspond à un intervalle de coût de migration dont les bornes sont précisées au-dessus de celui-ci. Cet intervalle est par ailleurs représenté en orange foncé.

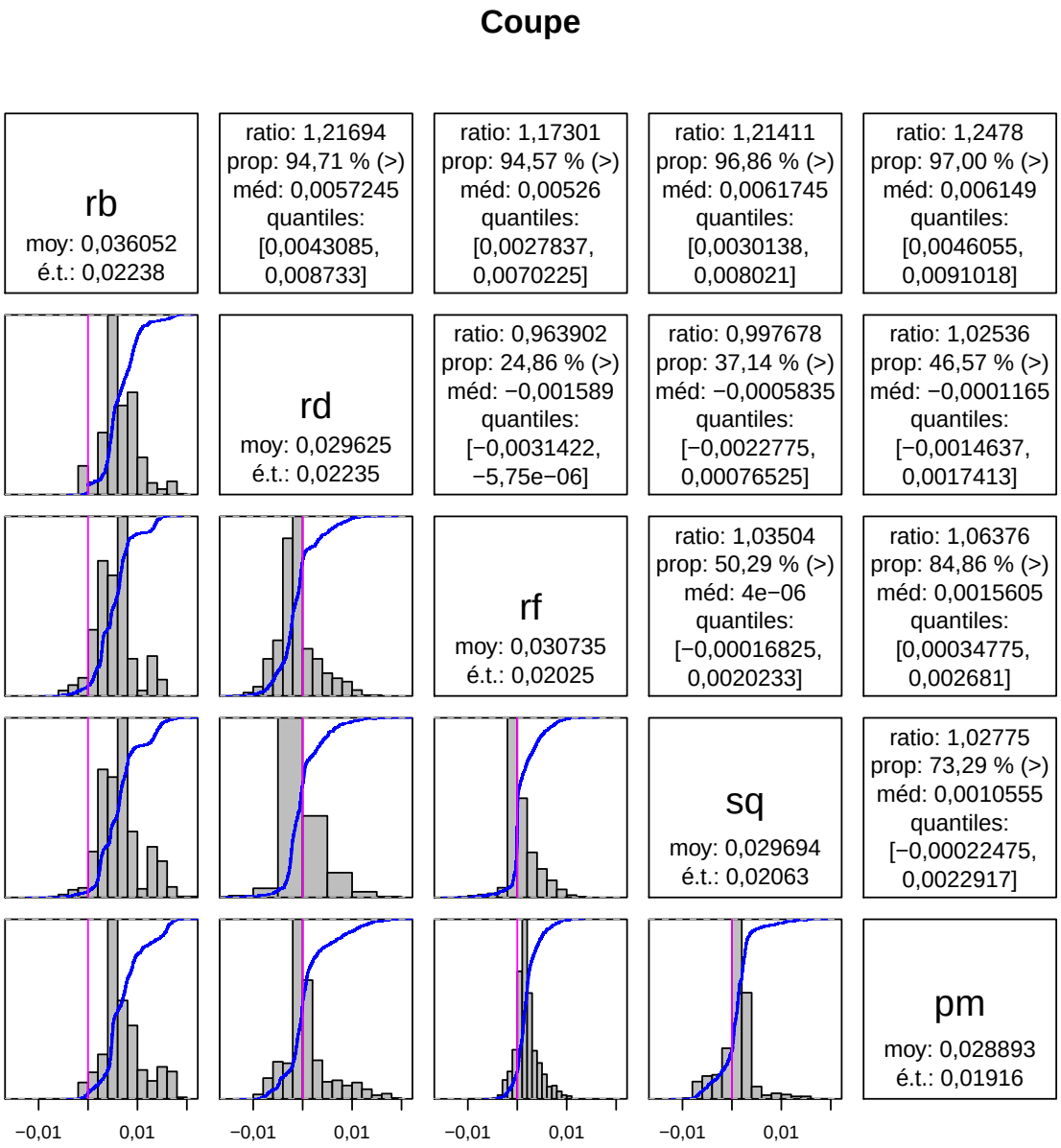


FIGURE 2.9 – Résultats en termes de coupe du repartitionnement séquentiel sur des graphes de type « maillage ». Le nom de la stratégie, sa moyenne et son écart-type apparaissent sur la diagonale. La répartition des différences entre la stratégie à la verticale et celle à droite est fournie sur la partie inférieure de la matrice. La fonction de répartition empirique est tracée en bleu. Sur la partie supérieure plusieurs métriques sont affichées afin de faciliter la comparaison entre la stratégie de gauche (g) et celle d'en bas (b), elles sont définies de la manière suivante : `ratio` est égal à $\frac{\text{moy}(g)}{\text{moy}(b)}$; `prop` correspond à la proportion des exécutions où la valeur obtenue avec la stratégie g est supérieure à celle obtenue avec la stratégie b ; `méd` est la médiane de $g - b$. Les 25è et 75è quantiles sont donnés entre crochets.

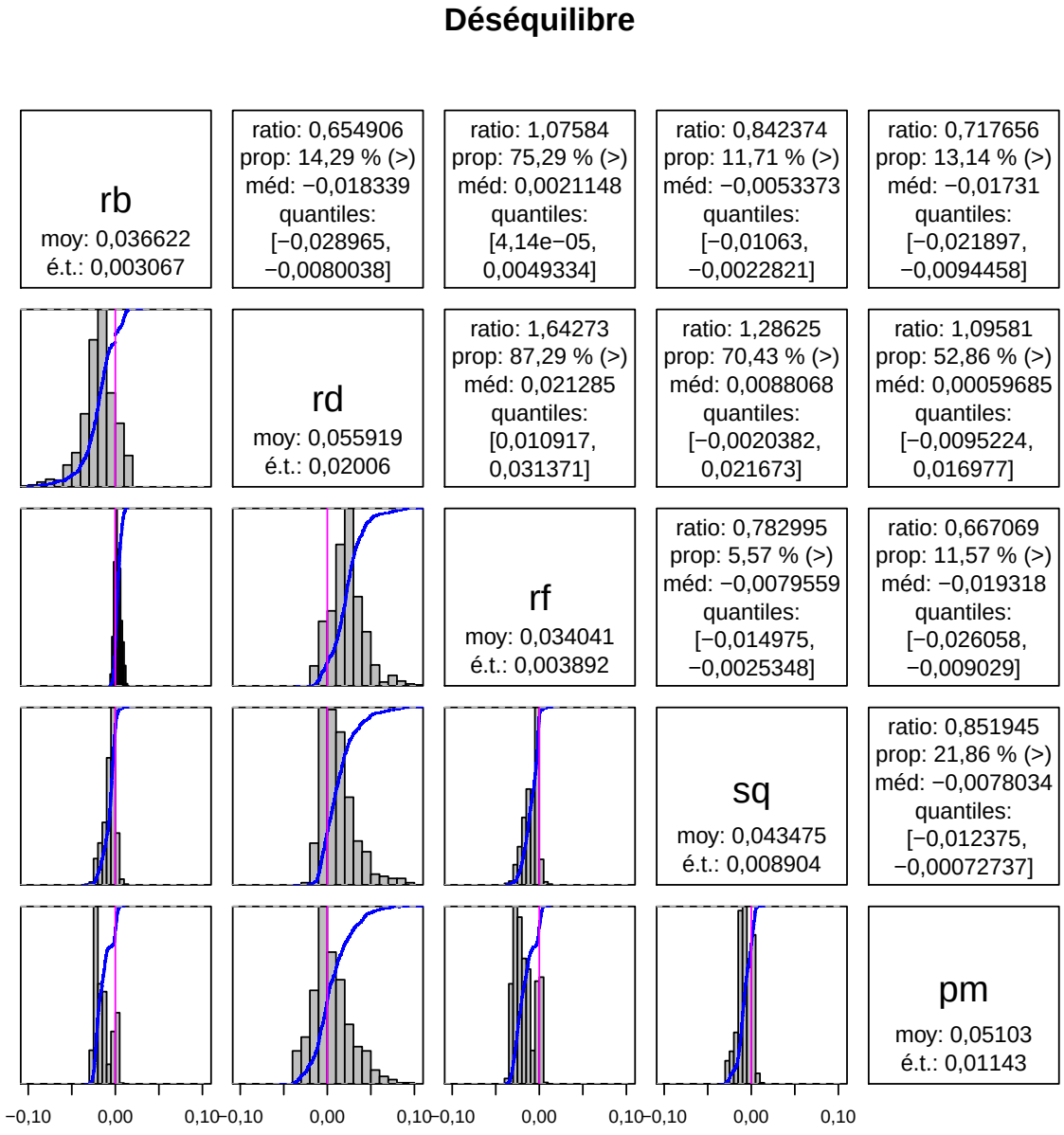


FIGURE 2.10 – Résultats en termes d'équilibrage de la charge du repartitionnement séquentiel.

TABLE 2.2 – Temps d'exécution moyen pour tous les résultats obtenus avec des graphes de type « maillage ». PARMETIS est exécuté sur deux processeurs, alors que ses méthodes sont exécutées séquentiellement.

Graphes	`rb`	`rd`	`rf`	`sq`	`pm`
`10millions`	10,603	111,150	45,645	118,542	17,056
`af_shell10`	1,548	13,149	3,453	14,564	2,040
`conesphere1m`	1,551	11,464	4,991	14,363	1,580
`coupole8000`	2,490	11,014	3,459	11,768	5,231
`ecology1`	0,533	2,095	7,290	2,650	0,700
`ldoor`	1,302	13,828	2,274	15,397	2,033
`thermal2`	0,778	3,289	1,906	4,126	1,108
Temps moyen global	2,68655	23,713	9,860	37,370	4,250

`rd` fait de même avec un écart un peu plus important $(0,006)$. En moyenne, `rd` donne la seconde meilleure coupe, qui est très proche de celle de `pm` ($0,0296$ avec un ratio de 1.025 et 53 % de ses valeurs inférieures à celles de `pm`). `rf` obtient une coupe honorable de $0,0307$.

Temps d'exécution Dans le tableau 2.2, nous pouvons voir que le temps d'exécution de `rb` est largement inférieur aux autres. Nous en déduisons que la majorité du temps d'exécution est consommé lors de la phase de raffinement. Les stratégies de type Fiduccia-Mattheyses sont plus rapides que celles qui utilisent la diffusion lors de la phase de raffinement. `pm` est un peu plus rapide que `rf` (la somme des temps d'exécution sur les deux processeurs est un peu inférieur à celui de `rf`). En moyenne, l'utilisation de `rd` sur un processeur est $5,6$ fois plus coûteux que d'utiliser `pm` sur deux processeurs et l'utilisation de `sq` est $8,8$ fois plus coûteuse.

2.7.1.5 Une stratégie *scalable* pour le repartitionnement parallèle

Pour résumer, `pm` et `sq` sont les meilleures stratégies en temps de coupe et d'équilibrage de la charge. Les stratégies basées sur SCOTCH sont plus sensibles aux variations du coût de migration. Les stratégies de type Fiduccia-Mattheyses sont rapides. Elles donnent une bonne coupe, tout en respectant une contrainte d'équilibrage de la charge paramétrable. `rd` donne une bonne coupe, un équilibrage de la charge proche de $0,05$ et est plus coûteux.

Comme `rd` est algorithmiquement plus *scalable* que les stratégies de type Fiduccia-Mattheyses, nous pouvons déduire, à partir de ces résultats, la stratégie pour le repartitionnement parallèle suivante :

1. Réaliser une phase de contraction parallèle afin d'obtenir un graphe suffisamment petit pour pouvoir être traité sur un seul processeur.
2. Sur chaque processeur, réaliser un repartitionnement séquentiel sur ce graphe contracté en utilisant la stratégie `sq`.
3. Lors de la phase d'expansion parallèle, tant que le graphe bande peut être stocké sur un seul processeur, calculer un graphe bande k-aire et réaliser, sur chaque processeur, un raffinement par diffusion, puis par un algorithme de type Fiduccia-Mattheyses.
4. Lorsque le graphe bande est trop grand pour pouvoir être stocké sur un seul processeur, calculer un graphe bande k-aire parallèle et réaliser un raffinement par diffusion.

2.7.2 Apport de l'implémentation du raffinement par diffusion avec fils d'exécution

La proposition de stratégie *scalable* pour le repartitionnement parallèle du paragraphe précédent s'appuie sur l'expérience que l'algorithme de diffusion se parallélise bien [Pel07] et sur la supposition qu'il en est de même pour l'adaptation de ce dernier au repartitionnement.

Afin de confirmer expérimentalement la *scalabilité* de l'adaptation au repartitionnement de l'algorithme de diffusion et avant d'étudier une solution entièrement parallèle, nous avons choisi d'étudier les performances d'une version avec fils d'exécution (*threads*).

La figure 2.11 de la page suivante présente les résultats obtenus pour un partitionnement en 128 parties réalisé avec 8 fils d'exécution. Nous observons un gain moyen en termes de temps d'exécution de $37,78$ % sur l'ensemble du programme.

2.7.3 Apport du raffinement par diffusion adapté au replacement

Comme nous l'avons dit en section 1.6.2 de la page 28, une fois que les algorithmes de repartitionnement sont validés, leur adaptation au replacement est, mis à part pour l'algorithme de diffusion k-aire, aisée à mettre en place. Étant donné que nous avons privilégié l'étude du repartitionnement parallèle, nous n'avons pas eu le temps de finaliser l'adaptation de l'ensemble des routines de repartitionnement au replacement. Il reste à ajouter la prise en compte de la notion de distance dans l'algorithme de diffusion k-aire pour le liquide de migration. Bien que ce manque induise une prise en compte partielle du coût de migration, nous présentons quelques résultats préliminaires en figure 2.12 de la page 63. Nous avons utilisé le même protocole que celui présenté en section 2.7.1.1 de la page 50, en modifiant les points suivants. Nous ne considérons que le graphe `10millions` et les coûts de migration $0,1$, 1 et 10. Plutôt que de réaliser un partitionnement en 128 parties, nous réalisons un placement sur un tore 3D comprenant 8 processeurs ($2 \times 2 \times 2$).

Nous observons, en figure 2.12(a), que le replacement apporte par rapport au repartitionnement un gain moyen en termes de coupe (qui prend en compte la dilatation) de 6 %. Les résultats en termes de nombre de sommets migrés, présentés en figure 2.12(b), montrent que le replacement a une sensibilité moindre au coût de migration. Ceci s'explique par la prise en compte partielle du coût de migration dans l'algorithme de diffusion, qui implique que le replacement migre en moyenne moins que le repartitionnement et, par corollaire, que la contrainte imposée par l'ancienne partition est plus forte pour le premier. Nous pouvons ainsi espérer obtenir un gain supérieur en termes de coupe, lorsque l'implantation sera finalisée.

2.7.4 Repartitionnement séquentiel de graphes dans Charm++

Nous présenterons dans cette section des résultats obtenus dans le cadre d'une collaboration avec Abhinav Bhatele, Harshitha Menon et Laxmikant V. Kale du *Parallel Programming Laboratory* de l'*University of Illinois at Urbana-Champaign* (États-Unis). Des résultats plus complets, mais plus anciens, ont fait l'objet d'un rapport de recherche [BFM+12]. L'objet de notre étude a été l'évaluation de la pertinence de l'usage des outils de repartitionnement de graphes afin d'équilibrer dynamiquement la charge d'un programme parallèle. Pour réaliser cette dernière, nous nous sommes appuyés sur le modèle de programmation de Charm++ [KK93].

Les expérimentations ont été mises en œuvre à l'aide de l'ordinateur parallèle Steele. Steele est un ordinateur parallèle Dell de l'université de Purdue et est administré par le *Rosen Center for Advanced Computing*. Chaque nœud comprend deux processeurs *quad-core*, de type Intel

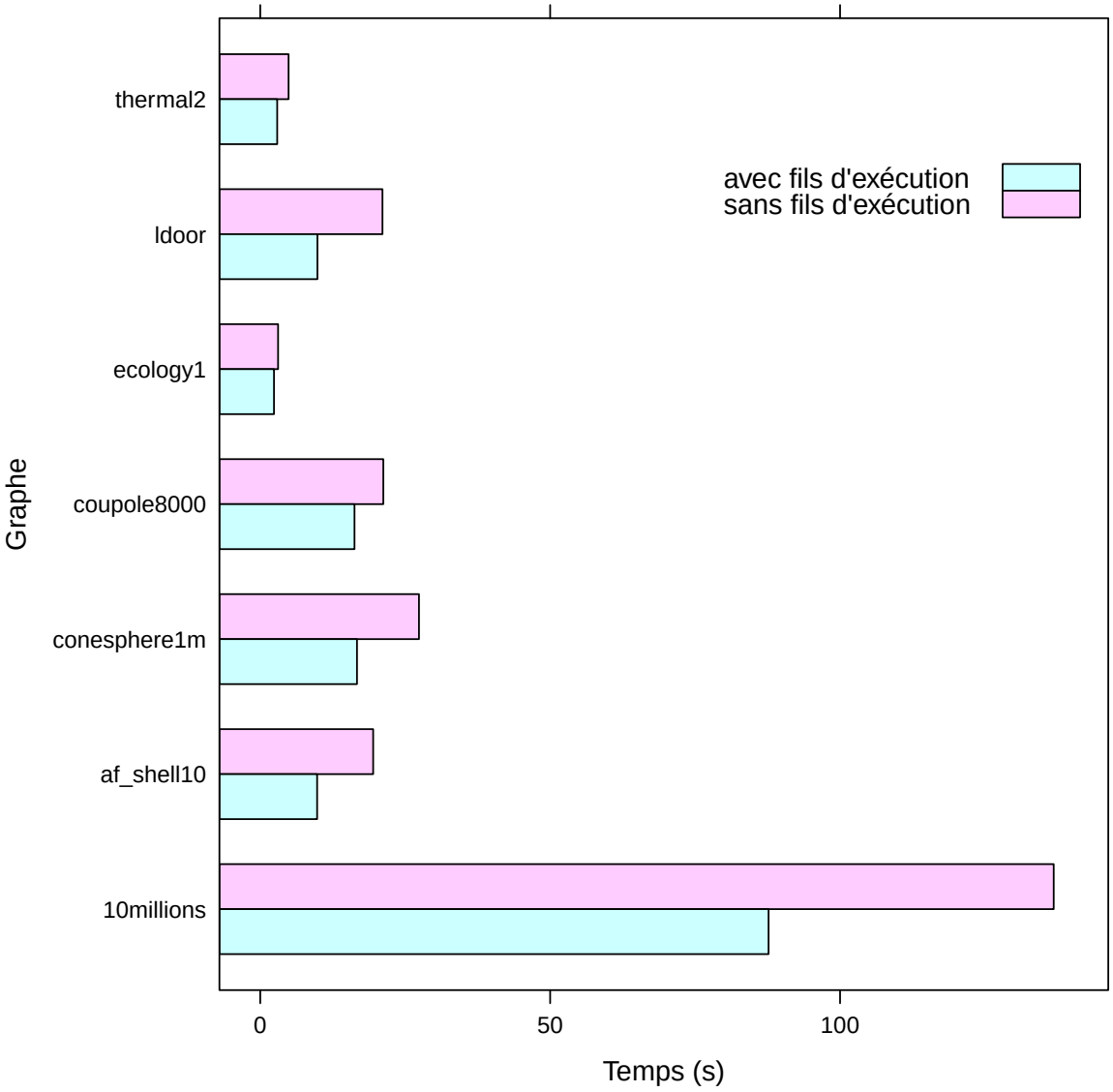
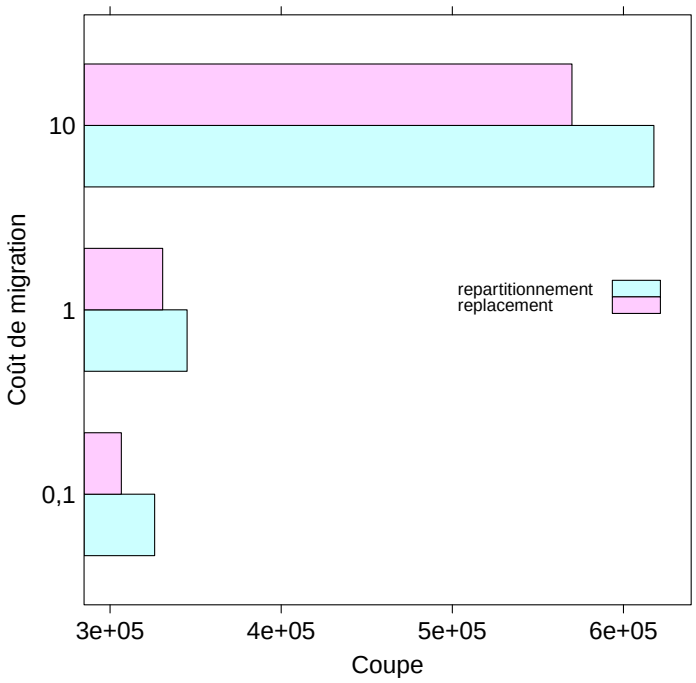
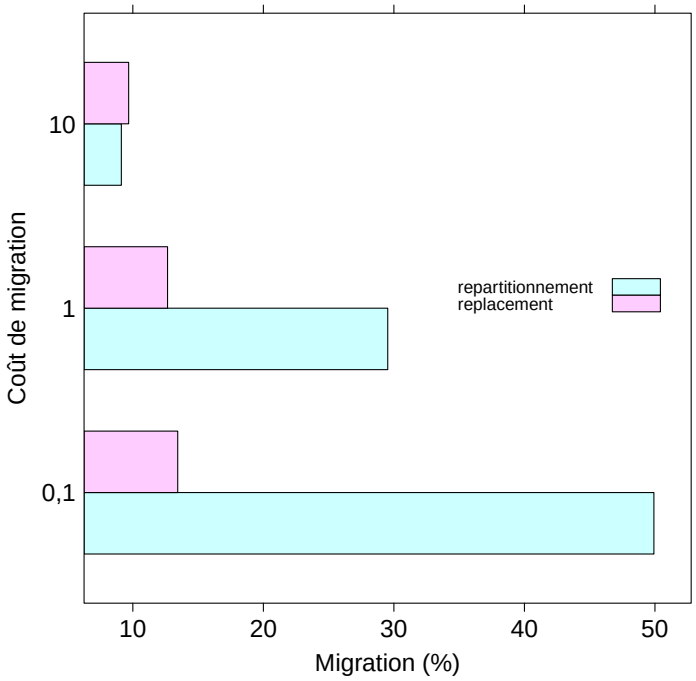


FIGURE 2.11 – Temps d'exécution obtenus avec la version avec fils d'exécution de l'algorithme de diffusion. Les 8 fils d'exécution sont exécutés sur 4 processeurs Intel(R) Core(TM) i5-2520M cadencés à 2.50GHz.



(a) Variation de la coupe en fonction du coût de migration.



(b) Variation du taux de migration en fonction du coût de migration.

FIGURE 2.12 – Comparaison des coupes (a) et des pourcentages de somments migrés (b) obtenus pour le replacement du graphe `10millions`.

E5410 cadencés à 2, 33 GHz ou de type Intel E5450 cadencés à 3, 00 GHz. Certains nœuds sont reliés par *InfiniBand*, d'autres par *Gigabit Ethernet*.

2.7.4.1 Équilibrage dynamique de la charge au sein de CHARM++

CHARM++ est une plate-forme de programmation parallèle portable. Les programmes écrits à partir de cette plate-forme peuvent être exécutés tels quels sur des ordinateurs MIMD avec ou sans mémoire partagée. Par ailleurs, CHARM++ implémente des mécanismes de haut niveau et des stratégies qui facilitent le développement d'applications parallèles complexes.

Dans les applications écrites avec CHARM++, les calculs à réaliser sont décomposés en processeurs virtuels ou objets, appelés *chares*. Au cours de l'exécution de l'application, ces derniers sont placés sur les processeurs physiques de l'ordinateur parallèle par la plate-forme d'équilibrage de la charge de CHARM++. Si le déséquilibre de la charge devient trop important, le placement courant va être modifié en migrant des objets vers d'autres processeurs. Ce réquilibrage se fait de manière transparente pour l'utilisateur. En effet, CHARM++ instrumente le code de l'application afin de mesurer les poids des tâches, ainsi que le graphe de communication des objets et utilise ces informations pour choisir les objets à migrer [BK00]. Les poids mesurés lors des itérations précédentes sont utilisés pour calculer une prévision de l'état futur. Pour que cette prévision soit valide, il faut que les informations mesurées varient lentement ; elles peuvent varier brusquement, mais seulement si cela se produit de manière éparse. Pour pouvoir gérer les cas où les poids varient brusquement et à une fréquence élevée, CHARM++ donne la possibilité à l'utilisateur de fournir ses propres estimations de l'évolution des poids au cours de l'exécution de l'application.

Plusieurs stratégies de rééquilibrage de la charge sont intégrées directement dans CHARM++. Nous utiliserons deux d'entre elles à des fins de comparaisons pour nos expérimentations : *GreedyLB* et *RefineLB*.

2.7.4.2 Protocole

Les mesures de performances qui vont suivre ont été réalisées en exécutant BT_MZ, une application multi-zone de la *NAS Parallel Benchmark suite (NPB)* [dWJ03]. C'est une mise en œuvre parallèle d'un code de résolution d'un système d'équations aux dérivées partielles non linéaires utilisant des matrices tridiagonales par blocs. Au sein de chaque classe de problèmes, des tailles de zones irrégulières sont considérées, faisant de cette application un bon candidat pour évaluer l'efficacité des stratégies d'équilibrage de la charge. Nous avons considéré pour nos expériences les problèmes de classes C et D de la version 3.3 des *NPBs*. L'application crée 256 zones pour la classe C et 1024 zones pour la classe D. Pour ces classes, la taille de la grille est respectivement de $480 \times 320 \times 28$ et de $1632 \times 1216 \times 34$. Les valeurs qui sont à la frontière entre les zones sont échangées à chaque itération. Le nombre d'objets créés par processeur par CHARM++ varie lui aussi selon la classe du problème. Par exemple, lors de l'exécution de la classe D sur 256 processeurs, il y a en moyenne quatre objets par processeur. Pour cette application, la quantité de données à transférer pour migrer un objet est importante. Pour nos expérimentations, nous avons utilisé la classe C pour les exécutions sur 32 et 64 processeurs et la classe D pour les exécutions sur 128 et 256 processeurs.

Nous avons considéré les cinq métriques suivantes :

– *Speedup in execution time* : le temps d'exécution d'une itération de l'application (représenté par l'accélération par rapport à la stratégie sans équilibrage dynamique de la charge, appelée `NoLB`). C'est le meilleur indicateur de l'efficacité d'une stratégie d'équilibrage de

la charge ;
- *Strategy time* : le temps nécessaire pour le calcul de l'équilibrage de la charge lors d'une itération ;
- *Migration time* : le temps nécessaire pour la migration des objets lors d'une itération.
Ces deux dernières métriques permettent, tout en considérant la fréquence de l'équilibrage de la charge, de déterminer si l'équilibrage est bénéfique ;
- *Number of migrations* : le nombre d'objets migrés permet de connaître la quantité de données qui devra être déplacée et les coûts de communication associés ;
- *Speedup in total application time* : le temps d'exécution total de l'application (représenté par l'accélération par rapport à la stratégie sans équilibrage dynamique de la charge `NoLB`), qui comprend l'exécution d'une itération, l'équilibrage de la charge et la migration des objets.

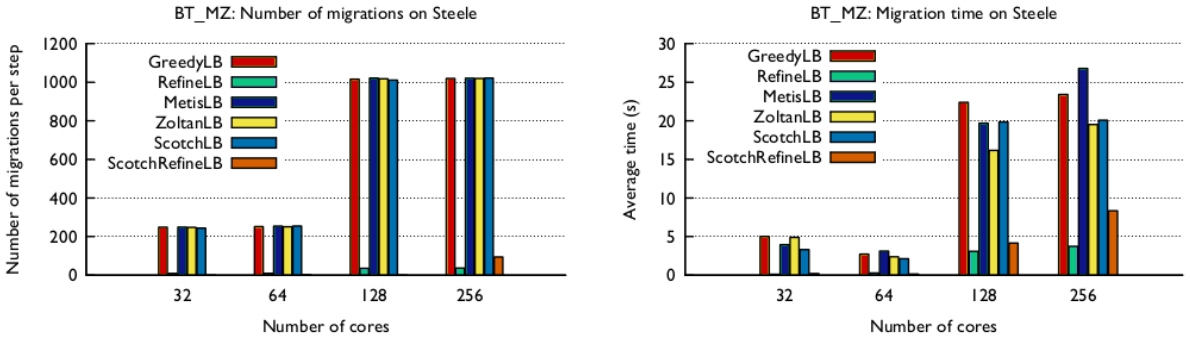
Nous avons considéré les stratégies suivantes :
- `NoLB` : cette stratégie qui ne rééquilibre pas la charge est utilisée comme référence dans le cas de l'accélération ;
- `GreedyLB` est une stratégie d'équilibrage de la charge globale basée sur une heuristique gloutonne qui place itérativement les objets les plus lourds sur les processeurs les moins chargés ;
- `RefineLB` est une stratégie de rééquilibrage de la charge qui migre des objets depuis les processeurs qui ont plus de tâches que la moyenne (en commençant par le processeur le plus chargé) vers ceux qui en ont moins que la moyenne. L'objectif de cette stratégie est de réduire le nombre d'objets migrés ;
- `ScotchLB` : cette stratégie utilise les fonctionnalités de partitionnement de Scotch pour calculer, un nouvel équilibrage de la charge *from scratch* ;
- `ScotchRefineLB` : cette stratégie commence par calculer un partitionnement, puis utilise les fonctionnalités de repartitionnement de Scotch pour calculer un nouvel équilibrage de la charge, tout en minimisant les migrations. C'est cette stratégie qui utilise les fonctionnalités de repartitionnement que nous avons implémentées dans le cadre de cette thèse ;
- `MetisLB` : les deux stratégies de partitionnement de MeTiS sont considérées : par bipartitionnement récursif et par partitionnement k-aire ;
- `ZoltanLB` : cette stratégie utilise les fonctionnalités de partitionnement d'hypergraphe de Zoltan.

Pour les stratégies basées sur Scotch, les options `STRAT_QUALITY` et `STRAT_BALANCE` ont tous les deux été considérés. Pour chaque stratégie, lorsque plusieurs configurations sont possibles, chacune d'entre elles a été exécutée et nous avons conservé à chaque fois le meilleur résultat.

2.7.4.3 Résultats

Le nombre de migrations réalisées et le temps nécessaire à la réalisation des migrations sont présentés en figure 2.13 de la page suivante. Les stratégies de rééquilibrage de la charge `RefineLB` et `ScotchRefineLB`, qui prennent en compte le coût de migration, migrent comme attendu moins d'objets. En conséquence, le temps de migration mesuré pour ces dernières est beaucoup plus faible.

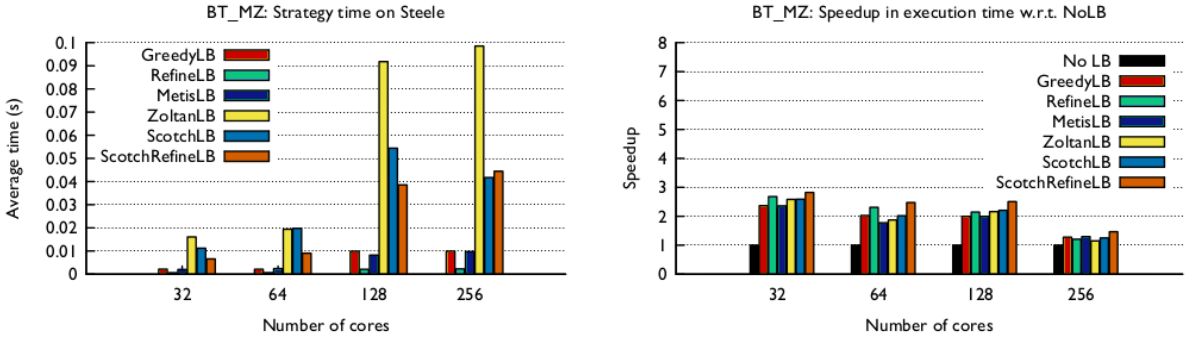
Le temps nécessaire à chaque stratégie pour calculer le nouvel équilibrage de la charge est donné en figure 2.14(a) de la page suivante. Nous observons que ce temps augmente avec la taille du problème, tout en restant négligeable devant le temps d'exécution total d'une itération. L'accélération pour une itération est présentée en figure 2.14(b) de la page suivante. `ScotchRefineLB` obtient les meilleurs résultats, avec une accélération allant de $2,5$ à 3 par rapport à `NoLB`. En



(a) Nombre de migrations.

(b) Temps de migration.

FIGURE 2.13 – Nombre de migrations et temps de migration pour une itération de BT_MZ.



(a) Temps d'équilibrage de la charge.

(b) Accélération.

FIGURE 2.14 – Temps d'équilibrage de la charge (*Strategy time*) et accélération, pour une itération de BT_MZ.

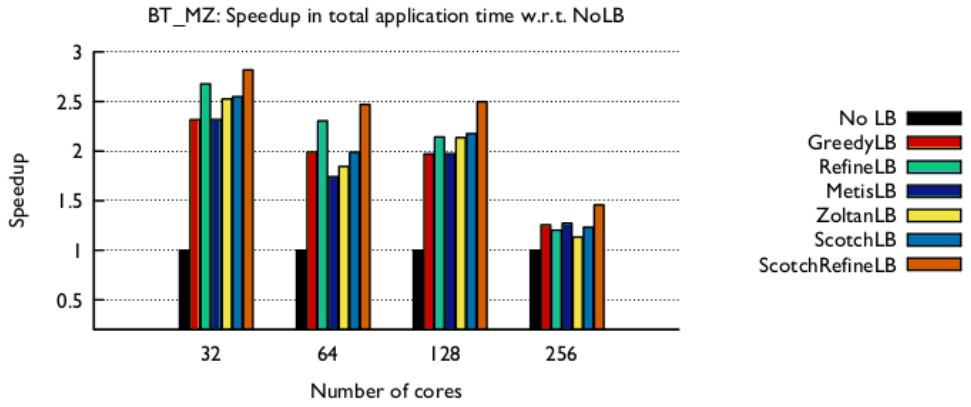


FIGURE 2.15 – Accélération pour une exécution complète de BT_MZ.

comparaison avec les autres stratégies, `ScotchRefineLB` procure une accélération en moyenne 11 % plus élevée.

Les résultats obtenus pour une exécution complète de BT_MZ avec un nouvel équilibrage de la charge calculé toutes les 1000 itérations sont présentés en figure 2.15 de la page ci-contre. Comme le temps de calcul du nouvel équilibrage de la charge est négligeable, le temps d'exécution de l'application correspond surtout à la somme du temps d'une itération et du temps de migration. La tendance observée est similaire à celle d'une itération : `ScotchRefineLB` donne une accélération en moyenne 12 % plus élevée que les autres stratégies. En comparaison avec `NoLB`, la stratégie `ScotchRefineLB` donne une accélération allant de $1,5$ à $2,8$ et la stratégie `ScotchLB` une accélération allant de $1,2$ à $2,5$.

2.7.4.4 Synthèse

Les stratégies d'équilibrage de la charge basées sur Scotch améliorent les performances de l'application multi-zone BT des NAS de 12 % par rapport aux stratégies préexistantes dans Charm++, ainsi que celles basées sur MeTiS ou Zoltan (sans repartitionnement). Elle réduisent par ailleurs le nombre de migrations, et donc le coût de migration. `ScotchRefineLB` migre 11 fois moins d'objets que `MetisLB` et `ZoltanLB`. En comparaison avec `NoLB`, `ScotchRefineLB` donne une accélération allant de $1,5$ à $2,5$ pour l'exécution de BT_MZ lorsque nous avons considéré des problèmes de classe D.

Ces résultats montrent l'intérêt d'utiliser des outils de repartitionnement de graphes pour équilibrer dynamiquement la charge au sein d'une application parallèle, plutôt que d'autres heuristiques telles que les approches gloutonnes ou des fonctionnalités de partitionnement simple.

2.8 Conclusion

Dans ce chapitre, nous avons présenté la mise en place d'algorithmes de repartitionnement séquentiel basés sur un schéma multi-niveaux. Parmi les contributions nécessaires à cette mise en place, nous nous sommes inspirés des méthodes d'influence afin d'adapater un algorithme de raffinement par diffusion au repartitionnement. Nous avons, par ailleurs, étudié expérimentalement les caractéristiques des différentes stratégies de repartitionnement que nous avons proposées. Cette étude nous a permis d'esquisser une stratégie de repartitionnement parallèle. Enfin, des résultats réalisés en collaboration avec l'équipe de Charm++ de l'Université d'Urbana-Champaign soulignent l'efficacité de notre approche pour l'équilibrage dynamique de la charge.

Chapitre 3

Contribution au repartitionnement parallèle

Sommaire

3.1 Introduction

Au chapitre précédent, nous avons décrit les stratégies de repartitionnement séquentielles que nous avons ajoutées à Scotch puis, à partir des résultats expérimentaux rassemblés, nous avons brièvement proposé une stratégie de repartitionnement parallèle *scalable*. En nous appuyant sur ces travaux, nous allons maintenant aborder la conception et la mise en œuvre d'algorithmes pour le repartitionnement parallèle au sein du logiciel Scotch. Nous étudierons ensuite leur comportement expérimental.

3.2 Principe de fonctionnement

Aisément parallélisable, le schéma multi-niveaux reste un outil de choix pour le calcul de partitionnements parallèles. Nous allons maintenant aborder les contributions que nous avons réalisées afin d'adapter les algorithmes de partitionnement parallèle de PT-Scotch au problème du repartitionnement.

3.2.1 Utilisation d'un schéma multi-niveaux k-aire parallèle

Comme pour le partitionnement séquentiel, jusqu'à maintenant, PT-Scotch utilisait un algorithme de bissection récursive qui nécessitait la réalisation complète d'un schéma multi-niveaux parallèle pour le calcul de chaque bissection. Pour les raisons déjà mises en avant au chapitre 2, nous avons choisi d'utiliser un schéma multi-niveaux k-aire comme base pour l'ajout des fonctionnalités de repartitionnement parallèle.

Suite au travail de post-doctorat de Jun-Ho Her, une première version fonctionnelle du schéma multi-niveaux k-aire parallèle était présente dans PT-Scotch ; ceci a facilité notre travail. Avant d'adapter les fonctionnalités de partitionnement parallèle k-aire au repartitionnement, nous avons toutefois dû réaliser des modifications au niveau des structures de données, des algorithmes existants et des interfaces, afin de rendre le code cohérent avec les choix réalisés dans les dernières versions de la bibliothèque PT-Scotch. Ainsi, il a fallu modifier la manière dont la partition était stockée en mémoire, ajouter les informations relatives aux sommets fixes et au repartitionnement, remplacer certaines informations copiées par des pointeurs sur l'information d'origine, etc.

3.2.2 Repartitionnement parallèle par bipartitionnement récursif

Bien que nous ayons fait le choix de privilégier l'utilisation du schéma multi-niveaux k-aire, il pré-existait dans Scotch une stratégie de partitionnement parallèle basée sur un bipartitionnement récursif parallèle. Dans le cas général — et pour les résultats que nous présenterons par la suite — nous n'utilisons pas cette stratégie, car le repartitionnement initial est calculé au moyen des algorithmes de bipartitionnement récursif séquentiels.

L'usage du bipartitionnement récursif parallèle reste toutefois nécessaire pour calculer un (re)partitionnement lorsque le nombre de parties est plus important que le nombre de sommets pouvant être stockés sur un nœud. Comme la phase de contraction nécessite, pour le repartitionnement, qu'il y ait au sein du graphe contracté au moins un sommet pour chaque ancienne partie, le repartitionnement initial devra être réalisé en parallèle. Il est donc nécessaire pour cela de disposer d'un bipartitionnement récursif parallèle. Comme nous l'avons présenté dans le contexte séquentiel en section 2.4 de la page 43, nous l'avons adapté en nous appuyant sur les fonctionnalités de partitionnement biaisé de Scotch. Étant donné que ce travail n'est pas

notre priorité, le code que nous avons produit est encore en version bêta et nécessite une phase de test approfondie, avant de pouvoir être pleinement utilisé.

3.3 Adaptation de la phase de contraction

L'adaptation de la phase de contraction du schéma multi-niveaux parallèle k-aire a nécessité, comme pour le contexte séquentiel, l'ajout de contraintes sur l'appariement des sommets. Il a aussi été nécessaire d'enrichir le repliement avec duplication, afin de transmettre aux graphes contractés les informations relatives à l'ancienne partition.

3.3.1 Disponibilité de l'information relative à l'ancien partitionnement

Comme pour le schéma multi-niveaux séquentiel, nous avons choisi d'utiliser les arêtes fictives seulement après la phase d'appariement. Pour rendre l'information de l'ancienne partition disponible lors du calcul du partitionnement séquentiel, nous l'avons prolongée à chaque contraction. Comme pour la contraction séquentielle, nous avons choisi, afin de ne pas perdre d'information lors de ces contractions, de n'autoriser l'appariement entre deux sommets que s'ils appartiennent à la même ancienne partition.

3.3.2 Utilisation du repliement avec duplication

Afin de permettre l'utilisation de ces fonctionnalités pour le repartitionnement, nous avons dû ajouter le transfert des informations de l'ancienne partition lors du repliement et de la duplication.

Par ailleurs, étant donné que l'usage du repliement et de la duplication induit une charge mémoire constante, plutôt que de la diviser par deux, nous avons choisi, pour les stratégies de repartitionnement parallèle, de ne l'utiliser que lorsque le graphe possède un nombre de sommets inférieur à 100 000.

3.4 Adaptation du partitionnement séquentiel

Les algorithmes pré-existants pour le partitionnement parallèle par bipartitionnement récursif effectuaient une multi-centralisation du graphe, afin que chaque processeur puisse calculer un partitionnement séquentiel différent, puis que le meilleur résultat puisse être conservé. Nous avons choisi de conserver cette approche pour le repartitionnement et le partitionnement k-aire. Par ailleurs, l'adaptation du partitionnement séquentiel au repartitionnement a nécessité la modification des méthodes de centralisation du graphe parallèle, afin de centraliser également l'ancienne partition.

Le choix de la meilleure partition était effectué en considérant la taille de la coupe et le déséquilibre de la charge de la manière suivante :

1. s'il y a eu des erreurs dans le calcul multi-séquentiel des partitions intiales, ne conserver que les partitions valides ;
2. parmi les partitions valides, choisir celles qui ont la meilleure coupe ;
3. parmi ces dernières, conserver celle qui a le meilleur équilibrage de la charge.

Afin de prendre en compte le coût de migration et de mieux respecter la contrainte d'équilibrage de la charge, nous avons adapté la méthode de sélection suivante :

1. s'il y a eu des erreurs, ne conserver que les partitions valides ;
2. parmi les partitions valides, si certaines respectent la contrainte d'équilibrage de la charge et que d'autres ne la respectent pas, ne conserver que les premières. Si aucune d'elles ne respecte la contrainte, conserver celle qui a le meilleur équilibrage ;
3. choisir, parmi les partitions conservées, celles qui ont la meilleure coupe ;
4. parmi ces dernières, conserver celle qui possède le meilleur équilibrage de la charge.

Tout comme pour la phase de contraction séquentielle, nous avons choisi d'effectuer le calcul (séquentiel) de la partition initiale du schéma multi-niveaux parallèle lorsque le graphe atteint une taille de moins de 10 000 sommets. Il peut arriver, toutefois, lorsque le nombre de processeurs est petit et que le processus de repliement avec duplication — qui commence lorsque le nombre de sommets est inférieur à 100 000 sommets — replie le graphe sur un seul processeur, avant que ce dernier n'ait atteint 10 000 sommets, que le calcul de la partition initiale soit réalisé sur un graphe de taille plus élevée.

3.5 Adaptation de la phase d'expansion

Nous l'avons évoqué, la faiblesse des stratégies de type Fiduccia-Mattheyses est leur manque de *scalabilité* sur un très grand nombre de processeurs. De fait, nous avons opté pour une phase d'expansion privilégiant le raffinement par diffusion. Après un court retour sur le repliement avec duplication et son impact sur la phase d'expansion, nous allons présenter la manière dont nous avons parallélisé l'algorithme de raffinement par diffusion, puis nous aborderons la multi-centralisation des graphes bandes, qui permet l'utilisation des algorithmes de raffinement séquentiels sur les premiers niveaux de l'expansion parallèle.

3.5.1 Repliement avec duplication

Lors de la phase d'expansion, les sous-ensembles de processeurs qui partageaient des graphes dupliqués vont se synchroniser, afin de choisir laquelle des deux partitions calculées est la meilleure. Ce choix était réalisé de manière analogue à celui de la meilleure partition lors de la multi-centralisation (voir section 3.4 de la page précédente).

3.5.2 Diffusion parallèle et graphe bande décentralisé

Afin de faciliter la compréhension de la parallélisation de l'algorithme de diffusion, il convient de la décomposer en deux parties. D'une part, les itérations relatives aux sommets normaux du graphe bande décentralisé sont locales à chaque processus ou restreintes à leur processeurs voisins. D'autre part, les itérations relatives aux ancres effectuent des modifications qui doivent être mises en commun entre tous les processus ayant des sommets d'une même partie. En termes de liquide, nous pouvons reformuler la phrase précédente de la manière suivante : le liquide des sommets normaux se transmet de proche en proche et ne nécessite, d'un point de vue parallèle, qu'une communication de type *halo* (où chaque processeur reçoit uniquement les informations des sommets voisins à ses sommets locaux), alors que la quantité de liquide transmise par les ancres est liée aux informations calculées par tous les processeurs possédant des sommets dans les parties des ancres, et nécessite donc une communication globale.

Alors qu'une communication de type *halo* ne posera pas de problème en termes de *scalabilité*, la communication globale nécessaire afin de conserver un fonctionnement des ancres identique au contexte séquentiel peut constituer un goulot d'étranglement. C'est en expérimentant plusieurs

approches pour effectuer cette communication globale que nous avons successivement exploré trois possibilités d'adaptation de l'algorithme de raffinement par diffusion.

Lors de notre première tentative, nous avons essayé de construire un graphe bande décentralisé qui permette d'utiliser telle quelle la parallélisation de la diffusion, en utilisant une communication de type *halo* sur tous les sommets du graphe bande. Pour ce faire, nous avons construit le graphe bande en ajoutant, pour chaque processeur, une ancre locale pour chaque partie. Nous avons aussi ajouté, entre chaque ancre locale d'une même partie, des arêtes de poids égal à un — formant ainsi une clique — de telle sorte que ces ancres locales se transmettent le liquide qu'elles possèdent. Pour rester cohérent, chaque ancre locale reçoit une fraction de la quantité de liquide globale affectée à sa partie, proportionnelle au nombre de voisins locaux qu'elle possédait. La faiblesse de cette approche réside dans le fait que l'augmentation de la taille de la clique entre les ancres d'une même partie est corrélée avec l'augmentation du nombre de processeurs. Ainsi, plus le nombre de processeurs augmente, plus la circulation de liquide entre les ancres est importante, ralentissant par là même la diffusion du liquide vers les autres sommets, et donc la convergence de l'algorithme.

Dans la seconde version, afin de résoudre ce problème, nous avons essayé d'adapter l'algorithme, dans l'objectif de le rendre plus local. Nous avons ainsi retiré la clique d'arêtes entre les ancres et nous avons considéré chaque ancre comme purement locale. Cette approximation, bien qu'elle résolve les problèmes de *scalabilité*, a un impact non négligeable sur la qualité des résultats obtenus. Elle a donc été abandonnée.

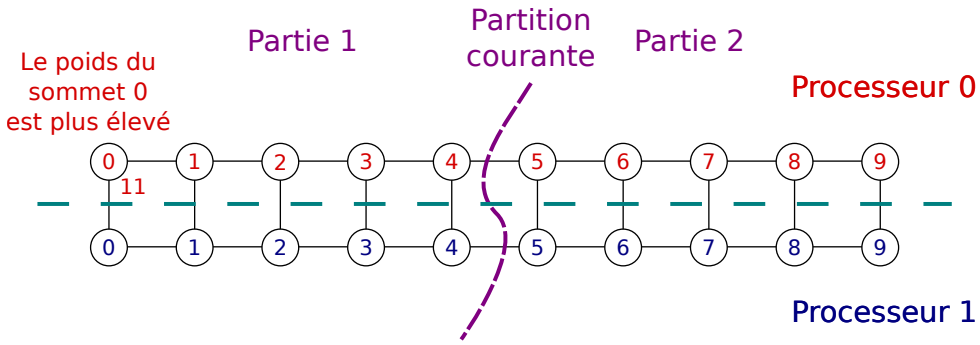
L'approche que nous avons finalement adoptée consiste à garder les ancres locales sans cliques et à calculer à chaque itération les quantités de liquide locales conservées par chacune de ces dernières. À la fin de chaque itération, après la communication de type *halo* entre les sommets normaux du graphe bande, la quantité de liquide locale à chaque ancre est mise en commun au moyen d'une communication de type *allReduce*. Puis, chaque processeur calcule localement, à partir de ces valeurs globales, les qualités de liquide qu'il va conserver, qui seront proportionnelles aux poids des arêtes locales partant de l'ancre considérée.

3.5.3 Limite de la convergence de l'algorithme de diffusion

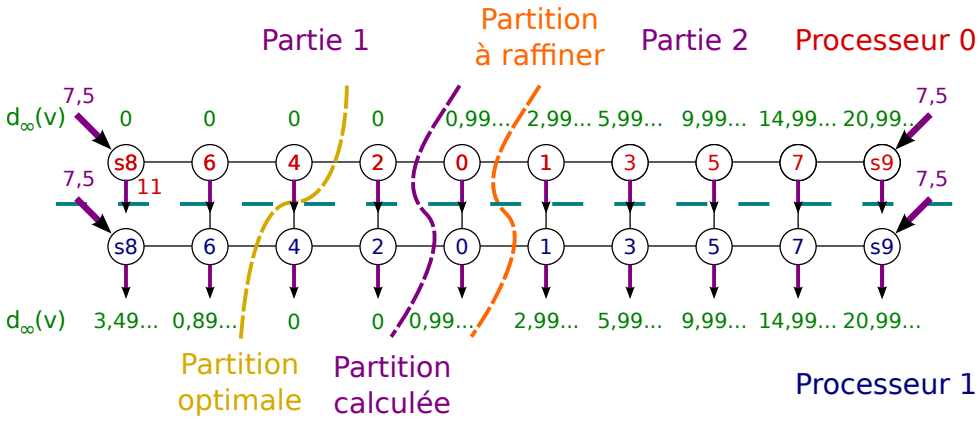
À chaque itération, les sommets du graphe diffusent la quantité de liquide qu'ils possèdent vers leurs voisins et perdent une quantité de liquide égale à leur poids. En conséquence, un voisin ne possédant pas encore de liquide sera, à l'itération suivante, capable d'en diffuser seulement si la somme des fractions de liquide fournies par ses voisins sera supérieure à son poids. Si son poids et le degré de ses voisins sont élevés, plusieurs itérations seront nécessaires afin qu'il puisse diffuser à son tour du liquide. Il est aussi possible que la quantité de liquide reçue par le sommet reste toujours inférieure à son poids, limitant ainsi l'avancée du liquide. La figure 3.1 de la page suivante illustre un cas extrême, où le front d'onde ne peut se déplacer que d'un niveau après la frontière à raffiner, alors que la convergence optimale nécessiterait que le front d'onde puisse aller à plus de deux niveaux de la frontière.

Cette limite de convergence est, au moins en partie, corrigée par le schéma multi-niveaux qui, avec une croissance assez lente de la taille du graphe à raffiner lors de la phase d'expansion, rend la faible convergence de l'algorithme de diffusion suffisante.

Afin d'améliorer cette convergence, nous avons imaginé une variation de l'algorithme où, à la place de supprimer, à chaque itération, une quantité liquide égale au poids du sommet, le liquide n'est plus supprimé, mais conservé par les sommets, jusqu'à ce qu'une quantité égale au poids de ces derniers soit stockée ; les sommets diffusent ensuite le liquide normalement. Après une très courte étude expérimentale, il est apparu que le stockage d'une partie du liquide diminuait



(a) Graphe à raffiner.



(b) Raffinement du graphe par diffusion parallèle.

FIGURE 3.1 – Limite de la convergence de l'algorithme de diffusion. La quantité de liquide $d_\infty(v)$, reçue par les sommets 0 après un très grand nombre d'itérations, est trop faible pour pouvoir être diffusée aux sommets 2. En conséquence, la partition calculée est différente de la partition optimale.

TABLE 3.1 – Description des graphes de test parallèles. Les cardinalités des sommets et des arêtes, $|V|$ et $|E|$, sont données en milliers.

Graphe	Description	Taille ($\times 10^3$)		Degré moyen
		$\|V\|$	$\|E\|$	
`10millions`	Électromagnétisme 3D, CEA	10423	78649	15.09
`23millions`	Électromagnétisme 3D, CEA	23114	175686	15.20
`45millions`	Électromagnétisme 3D, CEA	45241	335749	14.84
`af_shell10`	Mécanique des structures	1508	25582	33.93
`conesphere1m`	Électromagnétisme 3D, CEA	1055	8023	15.21
`ecology1`	Problème 2D/3D	1000	1998	4.00
`ldoor`	Mécanique des structures	952	22785	47.86
`thermal2`	Thermodynamique	1228	3676	5.99

la quantité de liquide à diffuser, et donc la vitesse de convergence de l'algorithme.

En conclusion, il nous semble intéressant de poursuivre l'étude d'autres algorithmes de raffinement proches de l'algorithme de diffusion, que ce soient ses variantes, des algorithmes à jetons ou des algorithmes inspirés par le comportement des fourmis ou des abeilles.

3.5.4 Graphe bande multi-centralisé

Nous l'avons vu au chapitre précédent, la faiblesse de l'algorithme de diffusion est le fait de ne pas pouvoir spécifier une borne donnée d'équilibrage de la charge. Afin de permettre d'avoir un meilleur équilibrage de la charge, mais aussi pour profiter de l'exploration d'un espace de solutions plus large, nous avons enrichi le mécanisme de multi-centralisation de graphes afin qu'il puisse être utilisé avec les graphes bandes. La disponibilité de ce mécanisme nous permet, sur la quantité de données plus petite qu'est le graphe bande, d'utiliser les algorithmes de raffinement séquentiels et notamment l'heuristique de Fiduccia-Mattheyses.

3.6 Résultats quantitatifs pour le repartitionnement parallèle

Nous allons maintenant présenter l'analyse quantitative des algorithmes que nous avons adaptés au problème de repartitionnement parallèle. Nous étudierons, dans un premier temps, les caractéristiques des stratégies de raffinement parallèles que nous proposons. Ces dernières seront ensuite comparées aux stratégies séquentielles présentées au chapitre précédent. Enfin, nous terminerons ce paragraphe par une étude de *scalabilité*.

3.6.1 Protocole

Les graphes que nous avons choisi d'utiliser pour l'analyse des stratégies de repartitionnement parallèle sont les graphes de type « maillage » utilisés pour l'analyse des stratégies séquentielles (voir section 2.7.1.1 de la page 50) auquels nous avons retiré `coupole8000` — qui a un comportement proche des graphes de type « non-maillage » — et ajouté des graphes de plus grande taille (`23millions` et `45millions`). Leurs caractéristiques sont détaillées au tableau 3.1.

Nous avons considéré les mêmes métriques que celles utilisées pour l'analyse quantitative du repartitionnement séquentiel, à savoir : la coupe, le déséquilibre, le taux de migration exprimé en pourcentage et le temps d'exécution.

Le protocole utilisé pour évaluer les stratégies de repartitionnement parallèles est le même que celui utilisé pour analyser les stratégies de repartitionnement séquentielles, mis à part le fait que les repartitionnements sont calculés en parallèle sur un nombre donné de processeurs. Voici, à titre de rappel, un résumé des étapes de notre protocole :

1. calcul d'une partition initiale de 128 parties ;
2. calcul d'un graphe modifié en augmentant de 1 les poids des sommets qui appartiennent aux 32 premières parties (soit un quart du nombre total de parties) de la partition initiale ;
3. exécution de plusieurs stratégies de repartitionnement séquentielles et parallèles de la partition initiale sur le graphe modifié.

Nous avons considéré les stratégies séquentielles suivantes :

- `rb` : Le calcul de la partition initiale du schéma multi-niveaux k-aire s'effectue par *bipartitionnement récursif*, la phase d'expansion est réalisée sans raffinement. Cette stratégie sera utilisée comme référence afin d'évaluer l'apport des différentes stratégies de raffinement ;
- `rd` : Le calcul de la partition initiale du schéma multi-niveaux k-aire s'effectue par *bipartitionnement récursif*. Lors de la phase d'expansion, l'algorithme de raffinement par *diffusion* est utilisé (cf. section 2.5.3 de la page 46). Cette stratégie sera utilisée comme référence vis-à-vis des stratégies parallèles de Scotch qui utilisent principalement la diffusion pour le raffinement ;
- `sq` : C'est la stratégie par défaut de Scotch lorsqu'on utilise la routine `SCOTCH_graphRemap`. La partition initiale du schéma multi-niveaux k-aire est calculée par bipartitionnement récursif. Lors de la phase d'expansion, un raffinement sur graphe bande est réalisé en appelant successivement l'algorithme de diffusion et la version optimisée de l'heuristique de Fiduccia-Mattheyses k-aire,

ainsi que les stratégies parallèles suivantes :

- `psr` : le partitionnement séquentiel du graphe contracté (qui s'effectue lorsque le graphe est de taille inférieure à 10 000 sommets) est calculé au moyen de la stratégie `sq`, la phase d'expansion parallèle est réalisée sans raffinement ;
- `pd` : le partitionnement séquentiel du graphe contracté est calculé au moyen de la stratégie `sq`, la phase d'expansion parallèle est réalisée par diffusion parallèle sur des graphes bandes de largeur trois, si ces derniers ont pu être créés ;
- `pdb2` : cette stratégie est une variante de la précédente. Le partitionnement séquentiel du graphe contracté est calculé au moyen de la stratégie `sq`, la phase d'expansion parallèle est réalisée par diffusion parallèle sur des graphes bandes de largeur trois, si ces derniers ont pu être créés, sinon une tentative de raffinement par diffusion parallèle sur graphes bandes de largeur deux est effectuée ;
- `pdc` : cette stratégie est identique à la stratégie `pd`, avec un comportement particulier lorsque le graphe bande est de petite taille. Lorsque le nombre de sommets du graphe bande est inférieur à 100 000, un raffinement multi-centralisé est réalisé, en appelant l'algorithme de diffusion séquentiel, puis la version optimisée de l'heuristique de Fiduccia-Mattheyses k-aire ;
- `pdcb2` : cette stratégie est une variante de la précédente. Si les graphes bandes de largeur trois n'ont pas pu être créés, une tentative de raffinement sur graphes bandes de largeur deux est effectuée ;
- `pm` : c'est la stratégie par défaut de *ParMetis* 4.0.2 en utilisant la routine `ParMETIS_V3_AdaptiveRepart`.

Nous avons utilisé les ordinateurs parallèles suivants, afin d'effectuer les calculs des repartitionnements :

– PLAFRIM : chacun de ses nœuds comprend deux processeurs Intel® Nehalem Xeon® X5550 quadri-cœurs cadencés à 2,66 GHz et 24 Gio de mémoire centrale ;
– Titane, mis à disposition par le CCRT : chacun des nœuds que nous avons utilisés comprend deux processeurs Intel® Nehalem quadri-cœurs cadencés à 2,93 Ghz, ainsi que 24 Gio de mémoire.

3.6.2 Études des stratégies de raffinement par diffusion sur graphes bandes

Afin de comparer le comportement des stratégies de raffinement par diffusion sur graphes bandes, nous nous appuyons sur des résultats obtenus sur la plate-forme PLAFRIM en considérant les graphes `10millions`, `af_shell10`, `conesphere1m`, `ecology1`, `ldoor` et `thermal2`.

Nous présentons tout d'abord les spécificités des stratégies sans raffinement que nous utilisons comme références. Nous étudions ensuite l'apport du raffinement sur graphes bandes centralisés lors des premiers niveaux de l'expansion parallèle, puis l'impact de l'usage de graphes bandes de taille deux lorsque la création de graphes bandes de taille trois échoue.

3.6.2.1 Stratégies sans raffinement

Le comportement des stratégies sans raffinement est présenté en figure 3.2 de la page suivante. Alors que ces stratégies ont un profil de coupe similaire, le déséquilibre et la migration présentent des variations plus marquées, notamment pour la stratégie parallèle (`psr`) exécutée sur 32 et 256 processeurs. Nous remarquons ainsi qu'au prix d'une fraction de sommets migrés plus élevée, la stratégie `psr` obtient — à coupe équivalente — un meilleur équilibre. Ces variations s'expliquent, d'une part, par une phase de contraction s'effectuant de manière différente, suivant le nombre de processeurs considérés, et d'autre part, parce que le choix de la « meilleure » partition initiale (après la phase de multi-centralisation) est réalisé parmi un ensemble de solutions dont la taille augmente avec le nombre de processeurs. Ces stratégies au comportement moyen proche peuvent dans certains cas présenter un écart plus marqué. Aussi, nous les ajouterons à nos résultats lorsque nous considérons que cela permet de mieux les interpréter.

3.6.2.2 Comportement des stratégies `pd` et `pdc`

Sur les figures 3.3 à 3.6, pages 79 et suivantes, nous pouvons observer le comportement des stratégies `pd` et `pdc` sur les graphes `10millions` et `af_shell10`.

Comme nous pouvons le voir en figures 3.3 et 3.5, la limite forte de 100 000 sommets, au dessus de laquelle le graphe bande ne sera pas centralisé avec la stratégie `pdc`, ne permet d'utiliser les stratégies de raffinement séquentielles que sur un petit nombre de niveaux (c'est le cas du graphe `af_shell10`) ou ne rend pas possible leur utilisation (ce que nous observons pour le graphe `10millions`). En conséquence, les résultats obtenus par les stratégies `pd` et `pdc` peuvent être identiques, comme nous pouvons l'observer en figure 3.4. En ce qui concerne le graphe `af_shell10`, pour un nombre moyen de raffinements réalisés lors de la phase d'expansion de 5, seuls 1 ou 2 de ces derniers sont effectués de manière centralisée. Les courbes présentées en figure 3.6 dépeignent, pour les différents stratégies, un comportement assez proche en termes de coupe, de déséquilibre et de migration. En termes de temps d'exécution, la stratégie avec raffinement centralisée apporte un surcoût moyen de 0,41s.

Le diagramme en haricot en figure 3.7 de la page 83 nous permet d'effectuer une analyse plus précise de ces deux stratégies pour tous les graphes que nous considérons. En moyenne, l'utilisation de la stratégie `pdc` permet d'obtenir une coupe de 0,034563 (écart-type : 0,02244) sur 8 processeurs, de 0,035699 (é.t. : 0,02445) sur 32 processeurs et de 0,034605 (é.t. : 0,02468)

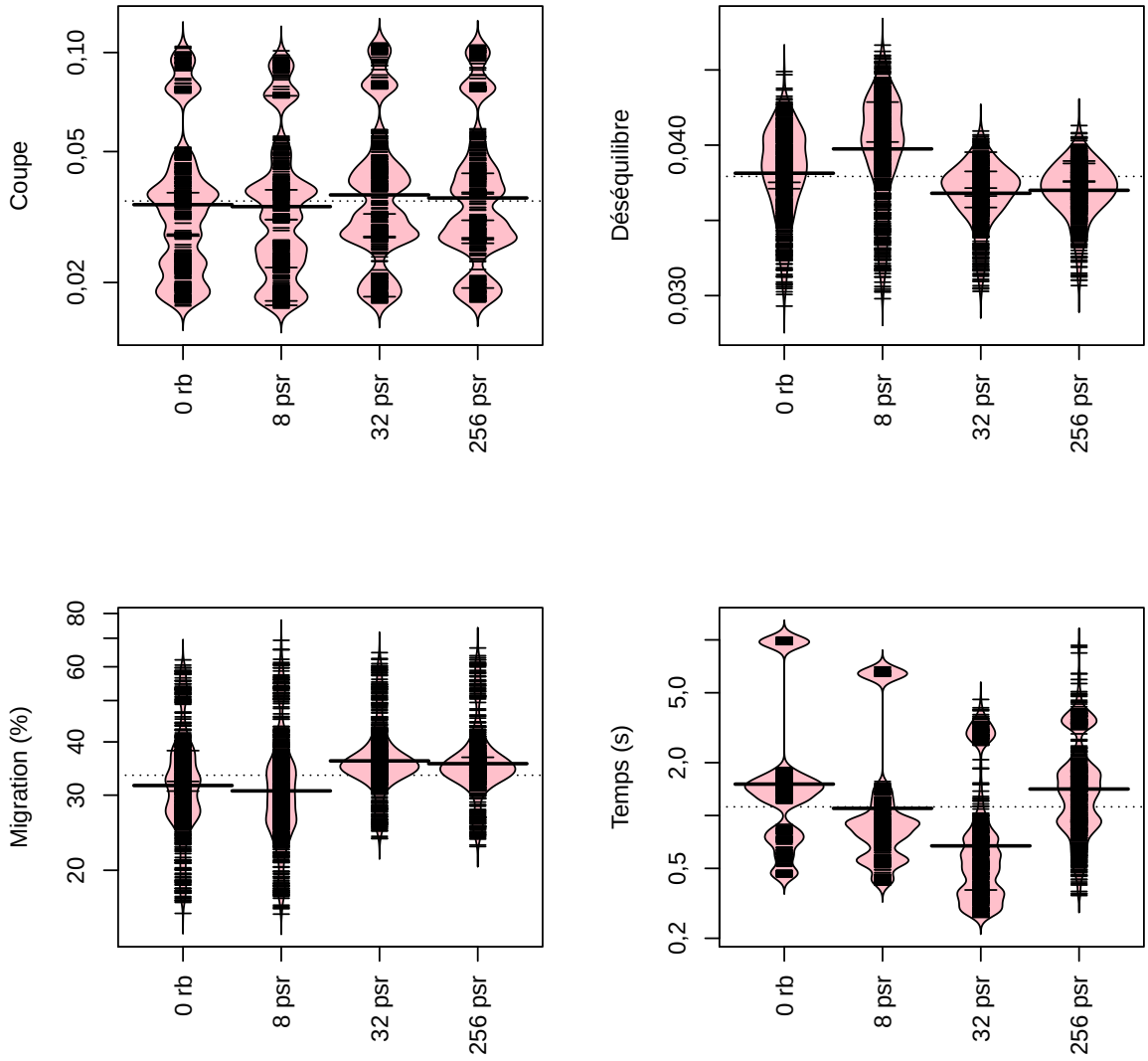


FIGURE 3.2 – Ces graphiques en haricot donnent, pour tous nos résultats avec les stratégies `rb` et `psr` exécutées sur 1 (représenté par « 0 » sur les graphiques afin de souligner le fait que nous utilisons une stratégie « séquentielle »), 8, 32 et 256 processeurs, une estimation de la densité locale pour les quatre indicateurs considérés. La courbe symétrique, qui fait le contour de la forme rose correspond au tracé d'une estimation non paramétrique de la densité. La ligne horizontale en pointillés représente la valeur moyenne globale pour toutes les stratégies. Les lignes horizontales épaisses représentent les moyennes pour chaque stratégie. Les petites lignes horizontales permettent de connaître les valeurs obtenues par individu. Lorsque l'échelle de l'axe des y est logarithmique, les moyennes présentées sont géométriques.

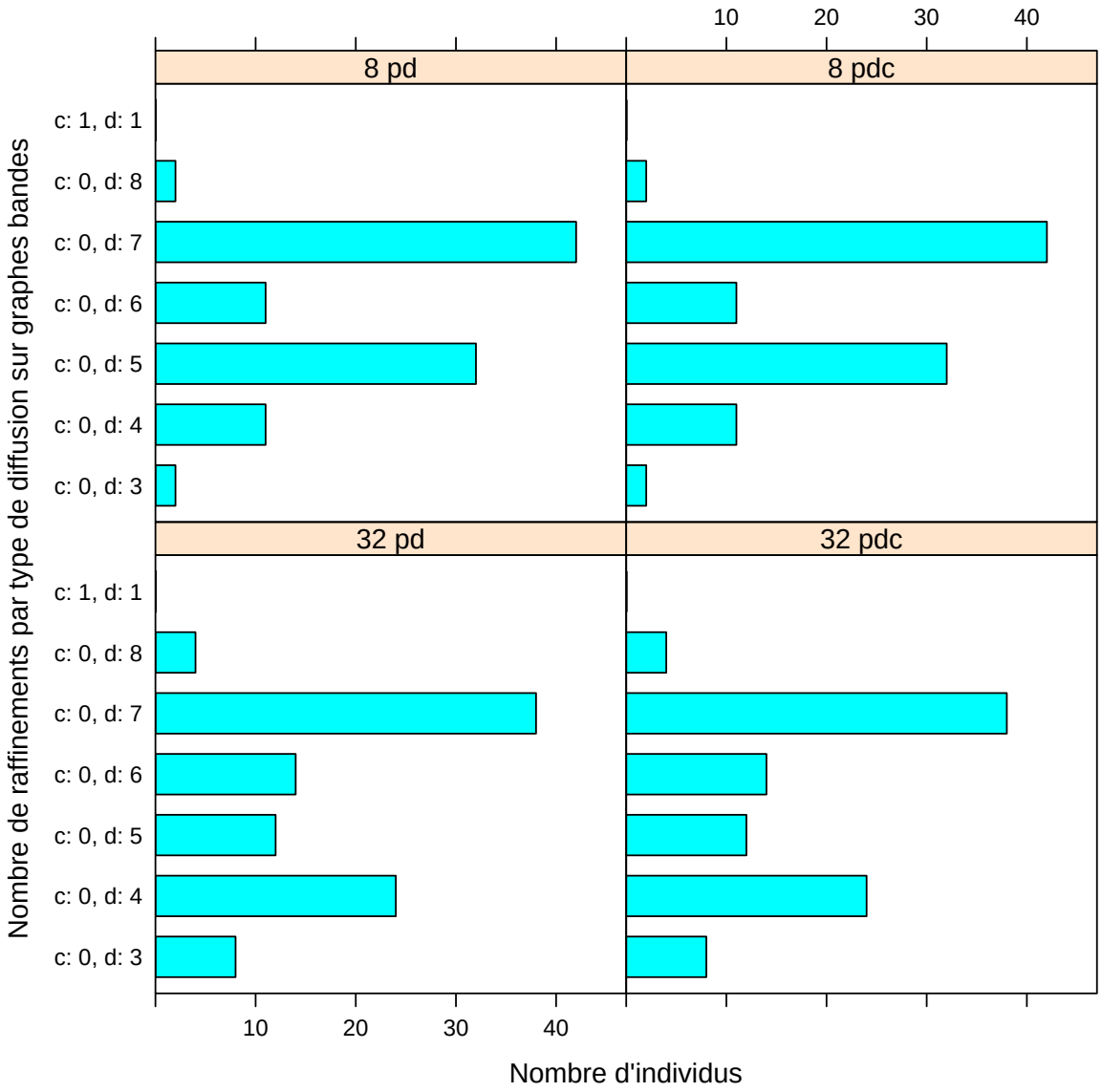


FIGURE 3.3 – Nombre de raffinements par diffusion centralisée (`c`) ou parallèle (`d`) sur graphes bandes, lors de la phase d'expansion, pour les stratégies `pd` et `pdc` sur le graphe `10millions`. La phase de contraction se déroulant différemment suivant le nombre de processeurs, les résultats présentés pour 8 processeurs correspondent à une phase d'expansion en 10 niveaux, alors que ceux présentés pour 32 processeurs correspondent à 12 niveaux de raffinement.

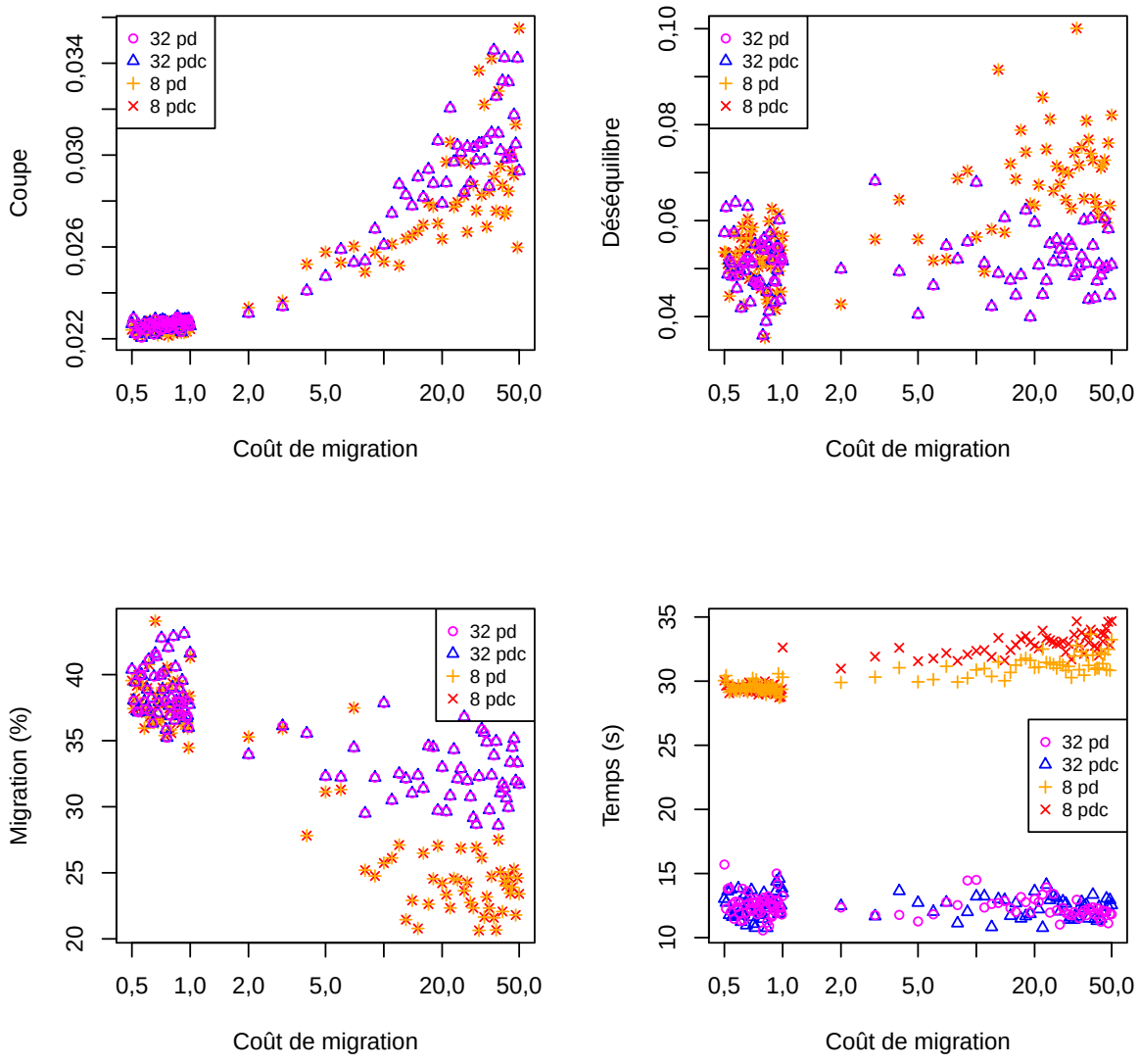


FIGURE 3.4 – Comportement des stratégies `pd` et `pdc` sur le graphe `10millions`.

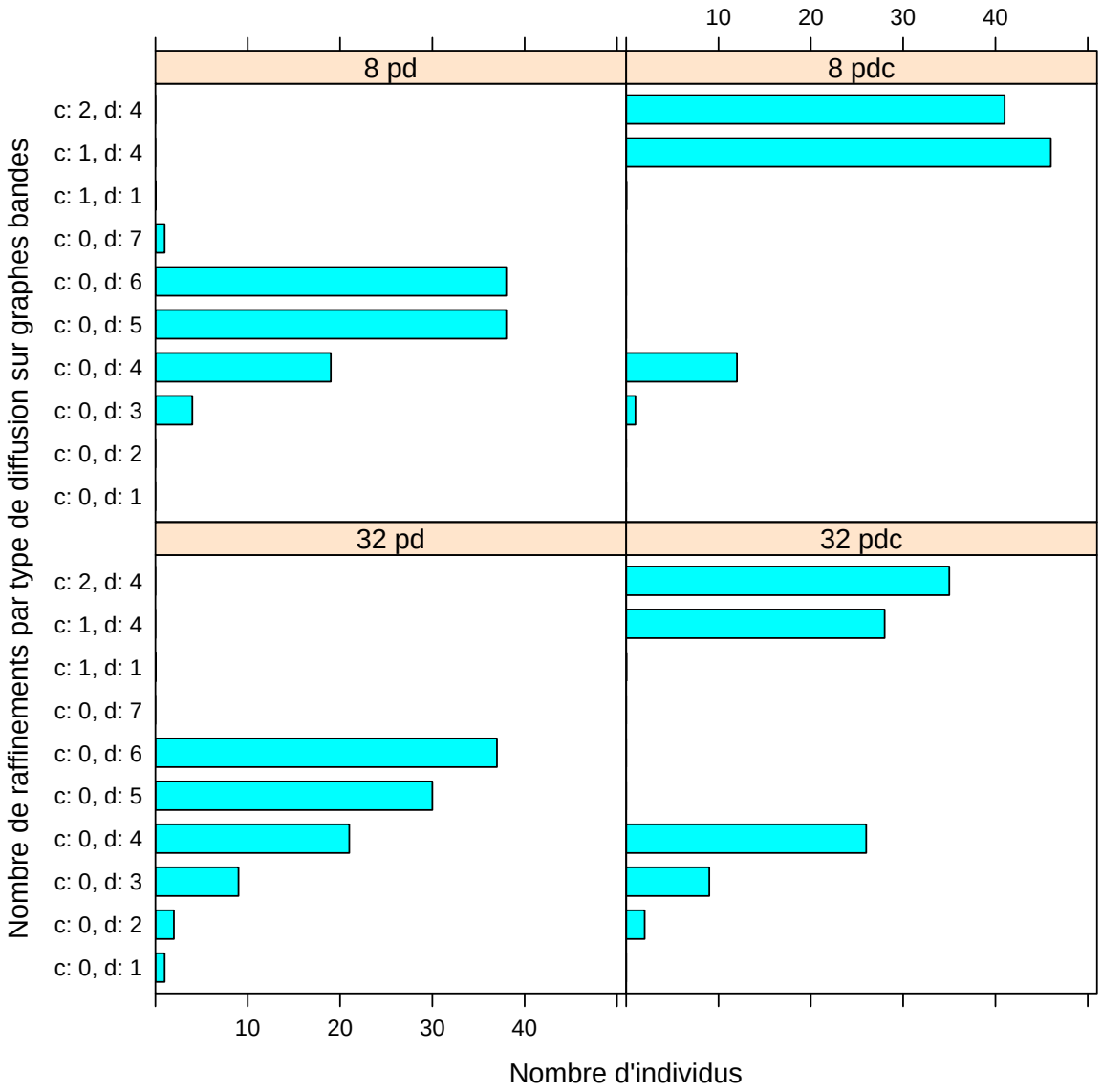


FIGURE 3.5 – Nombre de raffinements par diffusion centralisée ou parallèle sur graphes bandes, lors de la phase d'expansion, pour les stratégies `pd` et `pdc` sur le graphe `af_shell10`. La phase de contraction se déroulant différemment suivant le nombre de processeurs, les résultats présentés pour 8 processeurs correspondent à une phase d'expansion en 7 niveaux alors que ceux présentés pour 32 processeurs correspondent à 8 niveaux de raffinement.

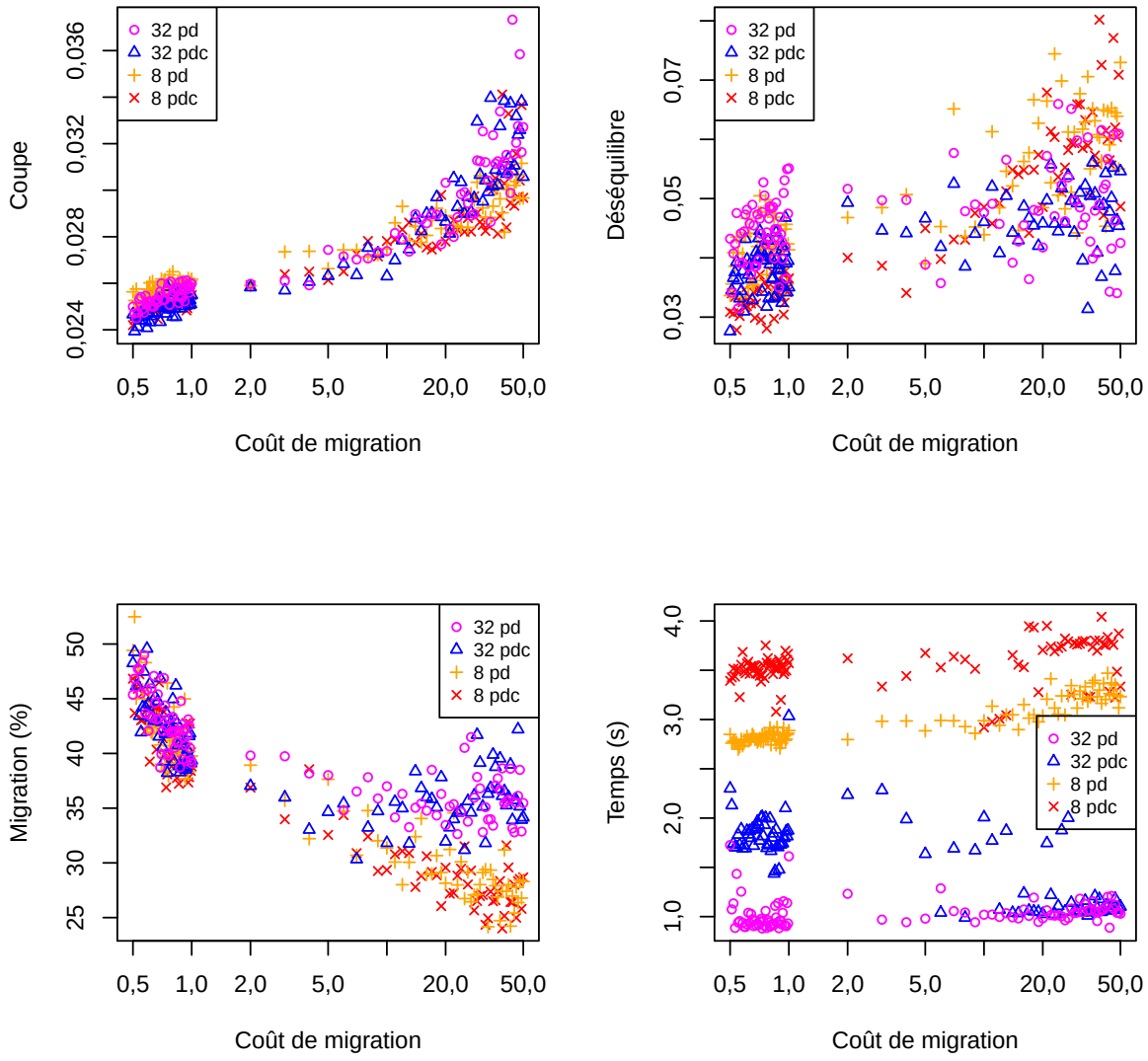


FIGURE 3.6 – Comportement des stratégies `pd` et `pdc` sur le graphe `af_shell10`.

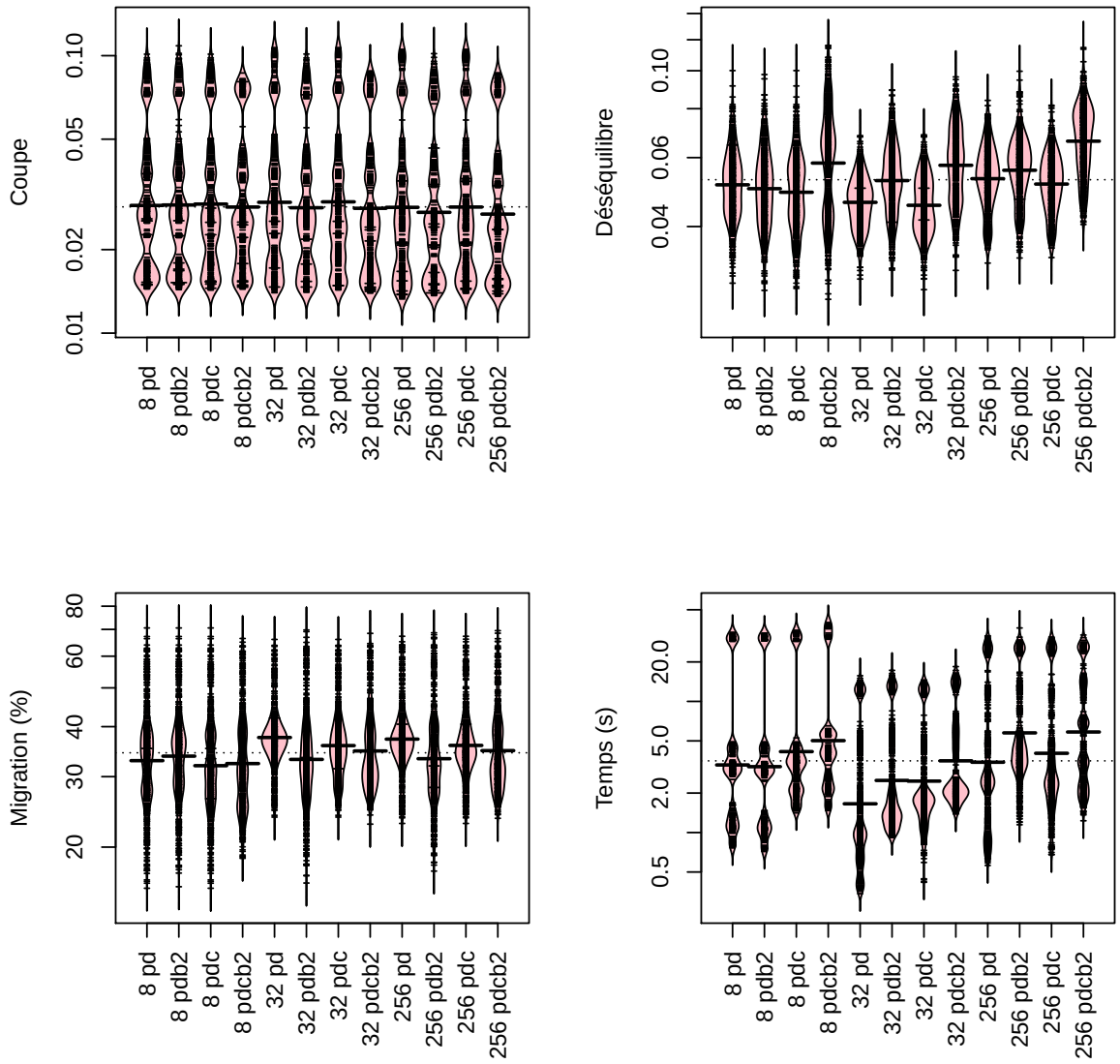


FIGURE 3.7 – Ces graphiques en haricot donnent, pour tous nos résultats, avec les stratégies parallèles `pd`, `pdb2`, `pdc` et `pdcb2` exécutées sur 8, 32 et 256 processeurs, une estimation de la densité locale pour les quatre indicateurs considérés.

sur 256 processeurs ce qui est très proche de celles obtenues par la stratégie `pd`, respectivement de $0,034372$ (é.t. : $0,02259$), $0,035657$ (é.t. : $0,02451$) et $0,034582$ (é.t. : $0,02471$). En termes de déséquilibre, les valeurs obtenues avec la stratégie `pdc` : $0,050188$ (é.t. : $0,0114$), $0,045967$ (é.t. : $0,007764$) et $0,052096$ (é.t. : $0,008794$) sont meilleures que celles obtenues avec la stratégie `pd` : $0,052144$ (é.t. : $0,01051$), $0,046629$ (é.t. : $0,006909$) et $0,053894$ (é.t. : $0,009684$). Le pourcentage de migrations moyen est proche de 40 % avec les valeurs $33,2$, $36,6$ et $36,7$ pour `pdc` ; $34,1$, $38,2$ et $38,0$ pour `pd`. La stratégie `pdc` a un temps d'exécution moyen de $7,64$s, $3,80$s et $7,32$s alors que la stratégie `pd` s'exécute en $7,12$s, $3,37$s et $7,05$s.

En résumé, l'ajout du raffinement sur graphe bande centralisé apporte une amélioration moyenne de l'équilibrage de l'ordre de 1 %, tout en nécessitant un temps de calcul supplémentaire d'en moyenne $0,41$s. La coupe obtenue par chaque stratégie est équivalente et le nombre de sommets migrés, en moyenne proche de 40 %, évolue peu. Étant donné que le respect de l'équilibrage de la charge est l'une des difficultés apportées par l'usage du raffinement par diffusion et que le surcoût induit par la centralisation est limité aux premiers niveaux de l'expansion, nous avons choisi de privilégier, dans les études qui vont suivre, l'usage de la stratégie `pdc`.

3.6.2.3 Impact du raffinement sur graphes bandes de taille deux

Sur les figures 3.8 de la page suivante et 3.9 de la page 86, nous pouvons observer le comportement des stratégies `pd` et `pdb2` sur le graphe `conesphere1m`. Le graphe `conesphere1m` est un graphe sur lequel il est, dans les premiers niveaux d'expansion, souvent impossible de créer un graphe bande de taille trois. Il est donc le candidat idéal pour réaliser une première évaluation de l'apport de la stratégie `pdb2`.

Les résultats présentés en figure 3.8 montrent que, lorsque la création d'un graphe bande de taille trois échoue, la tentative d'effectuer un raffinement sur graphe bande de taille deux (la particularité de la stratégie `pdb2`) est parfois couronnée de succès. Sur 8 processeurs, la stratégie `pdb2` permet d'accroître le nombre d'exécutions où un raffinement peut être réalisé lors de la phase d'expansion. Sur 32 processeurs, avec la stratégie `pdb2`, nous observons une diminution du nombre d'exécutions où aucun raffinement ne peut avoir lieu, au profit d'une augmentation des nombres d'exécution où un et trois raffinements sont effectués.

Comme nous pouvons le voir en figure 3.9, la possibilité d'effectuer des raffinements supplémentaires sur les premiers niveaux de la phase d'expansion, grâce à la considération de graphes bandes de taille deux, a naturellement un impact sur les résultats obtenus. Ainsi, sur 32 processeurs, la stratégie `pdb2` apporte une meilleure coupe, au détriment du déséquilibre. Les résultats obtenus en termes de migration sont proches, bien que `pdb2` ait tendance à migrer moins pour des coûts de migration élevés. Du fait du temps nécessaire à la réalisation de raffinements supplémentaires, la stratégie `pdb2` est plus coûteuse en termes de temps d'exécution.

Le diagramme en haricot de la figure 3.10 de la page 87 nous permet de comparer les stratégies `pd`, `pdb2` et `pdc`, `pdcb2` en considérant les résultats obtenus avec le graphe `conesphere1m`. Nous remarquons tout d'abord que la stratégie `psr`, bien qu'apportant un petit déséquilibre de $0,04$, permet d'obtenir une coupe dont la qualité est en moyenne moins bonne sur 32 processeurs $0,088681$ (é.t. : $0,01078$) que sur 8 processeurs $0,082203$ (é.t. : $0,008925$). En conséquence, pour obtenir un raffinement de qualité équivalente à celui qui serait obtenu sur 8 processeurs, les algorithmes de raffinement devront, sur 32 processeurs, améliorer la coupe dans une plus grande mesure. Alors que les stratégies `pd` et `pdb2` ont un comportement très proche sur 8 processeurs, sur 32 processeurs, `pdb2` apporte une meilleure coupe ($0,078864$, é.t. : $0,008223$) que `pd` ($0,085418$, é.t. : $0,0122$) au prix d'un moins bon équilibre ($0,053466$, é.t. : $0,007906$ pour `pdb2` et $0,046347$, é.t. : $0,00828$ pour `pd`). Cette tendance à ce que l'utilisation des graphes bandes

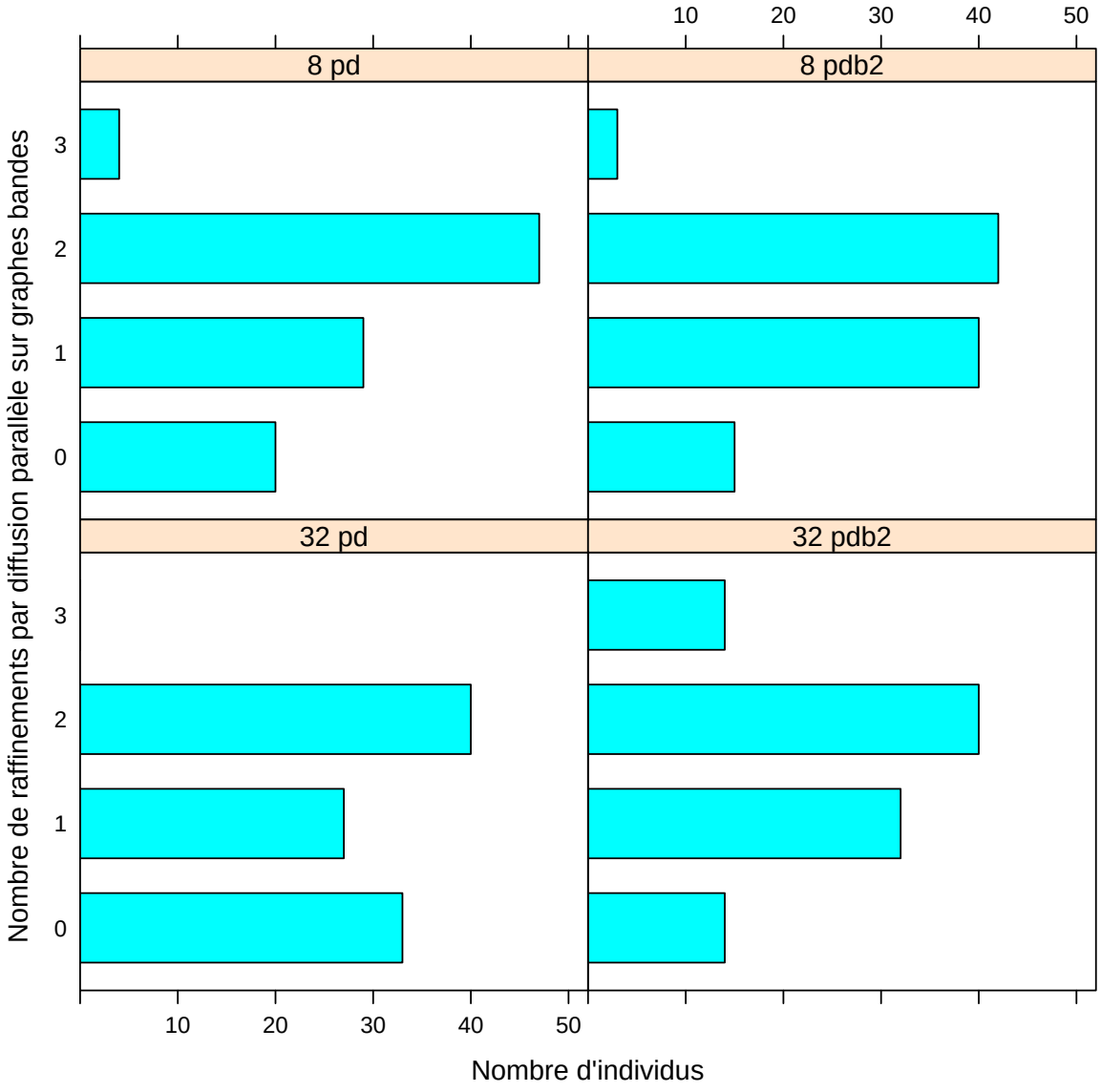


FIGURE 3.8 – Nombre de raffinements par diffusion parallèle sur graphes bandes lors de la phase d'expansion pour les stratégies `pd` et `pdb2`, sur le graphe `conesphere1m`.

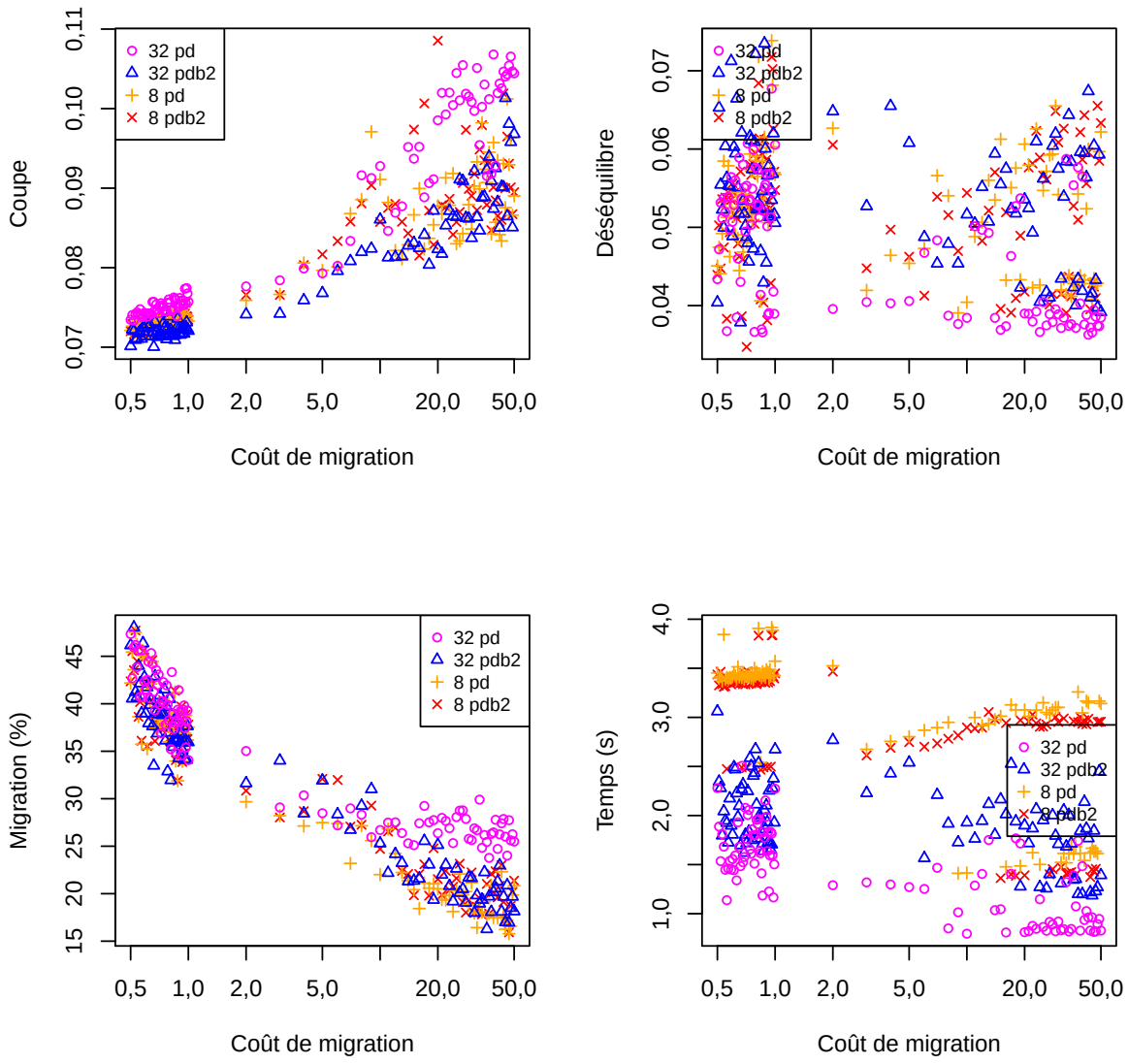


FIGURE 3.9 – Comportement des stratégies `pd` et `pdb2` sur le graphe `conesphere1m`.

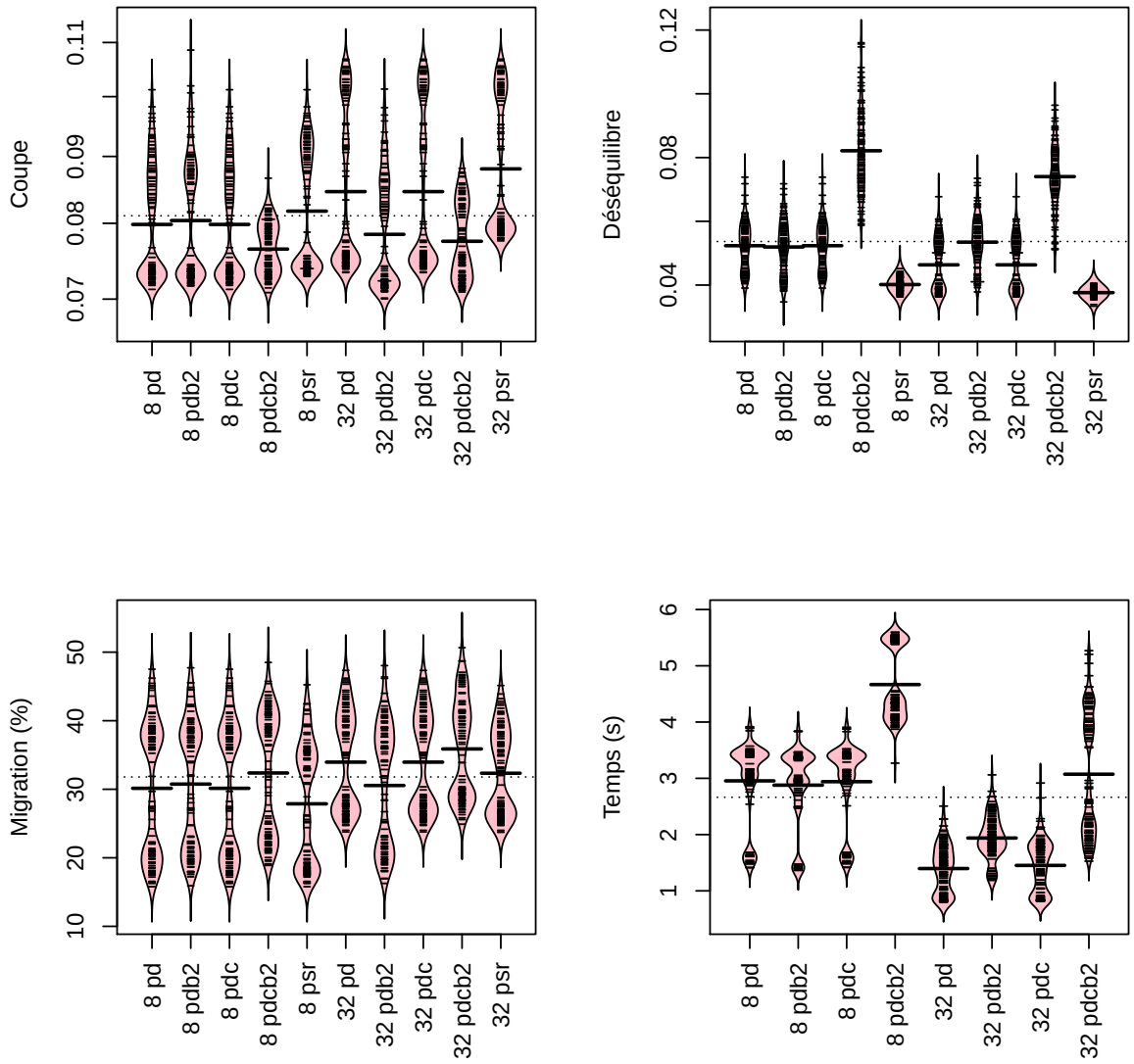


FIGURE 3.10 – Ces graphiques en haricot donnent, pour nos résultats sur le graphe `conesphere1m` avec les stratégies parallèles `pd`, `pdb2`, `pdc` et `pdcb2` exécutées sur 8 et 32 processeurs, une estimation de la densité locale pour les quatre indicateurs considérés.

de taille deux induise une meilleure coupe et un moins bon déséquilibre s'amplifie lorsque nous mettons en corrélation les stratégies `pdc` et `pdcb2`. Sur 8 et 32 processeurs, la stratégie `pdcb2` apporte, en comparaison avec la stratégie `pdc`, un gain de coupe non négligeable, mais aussi une forte augmentation du déséquilibre moyen, de l'ordre de $0,03$. De ces résultats, nous pouvons déduire que l'information contenue dans les graphes bandes de taille deux n'est pas suffisante pour réaliser un raffinement permettant par la suite d'optimiser au mieux l'équilibrage de la charge. Ce constat est d'autant plus visible lorsque nous comparons les stratégies avec graphes bandes multi-centralisés. Ces stratégies où l'usage supplémentaire de l'heuristique de Fiduccia-Mattheyses sur les graphes bandes des premiers niveaux devrait — en accord avec les résultats obtenus au précédent chapitre — améliorer l'équilibrage, contribuent ici à l'augmenter.

Afin d'élargir les conclusions réalisées en nous intéressant au graphe `conesphere1m`, nous allons maintenant étudier les résultats obtenus sur tous les graphes considérés, qui sont présentés en figure 3.7 de la page 83. Ces résultats globaux corroborent les tendances déjà observées. Bien qu'en moyenne la coupe soit peu fonction de la stratégie utilisée et reste proche de $0,034$, nous observons de meilleurs résultats pour les stratégies avec raffinement sur graphes bandes de taille deux. En termes de déséquilibre, la variation est plus marquée et nous obtenons ainsi — en considérant toutes les exécutions sur 8, 32 et 256 processeurs — une variation du déséquilibre moyen entre les stratégies `pdc` ($0,049417$) et `pdcb2` ($0,062421$) de $0,013$. Cette variation est moins marquée entre les stratégies `pd` et `pdb2`, puisqu'elle est égale à $0,0026$.

Notre objectif étant d'obtenir la meilleure coupe, tout en respectant la contrainte d'équilibrage de la charge, nous pourrions conclure au premier abord que l'ajout de raffinement sur les premiers niveaux de la phase d'expansion au moyen des graphes bandes de largeur deux, en permettant d'obtenir une meilleure coupe, nous aide à atteindre cet objectif. Or, ce gain n'est possible qu'au prix d'une perte notable d'équilibrage de charge et a tendance à empêcher les raffinements suivants de préserver un équilibrage de charge en dessous de la borne fixée. Pour cette raison, nous privilégerons les stratégies sans raffinement sur graphes bandes de taille deux et, en conséquence, les stratégies `pdb2` et `pdcb2` ne seront plus considérées par la suite.

3.6.3 Mise en regard des stratégies de raffinement séquentielles et parallèles

Les figures 3.11 de la page suivante et 3.12 de la page 90 nous permettent d'effectuer une première mise en corrélation des stratégies parallèles et séquentielles sur le graphe `10millions`. En ce qui concerne les stratégies séquentielles, nous avons considéré les stratégies `rb`, `rd` et `sq`. Nous appuyant sur les analyses précédentes, nous ne considérons que notre stratégie parallèle `pdc`, ainsi que celle proposée par PARMETIS : `pm`.

En termes de coupe, nous remarquons sur les résultats de la figure 3.11 que la stratégie `sq` permet d'obtenir une coupe en moyenne plus petite que `pdc` de $0,003$ pour les coûts de migration les plus faibles. Pour les coûts de migration supérieurs à 10, la stratégie parallèle `pdc` donne de meilleurs résultats que la stratégie `rd` et est proche de `sq`, tout en étant plus stable. Cette meilleure coupe et cette stabilité sont obtenues grâce à un relâchement de la contrainte d'équilibrage de la charge ; ainsi, la stratégie `pdc` donne des résultats avec un déséquilibre légèrement plus élevé. La stratégie `pdc` migre un nombre de sommets proche des stratégies séquentielles et son temps d'exécution est évidemment plus court.

Cette première analyse est corroborée par les résultats obtenus sur les autres graphes. Ainsi, le diagramme de la figure 3.12 nous permet de mettre en évidence, sur tous nos résultats, un premier comportement global des stratégies étudiées. En termes de coupe, toutes les stratégies de diffusion et la stratégie `pm` obtiennent des valeurs proches de la stratégie `sq`, la stratégie `rd` donnant une coupe de moins bonne qualité. Les valeurs de coupe de la stratégie `pdc` sont, à

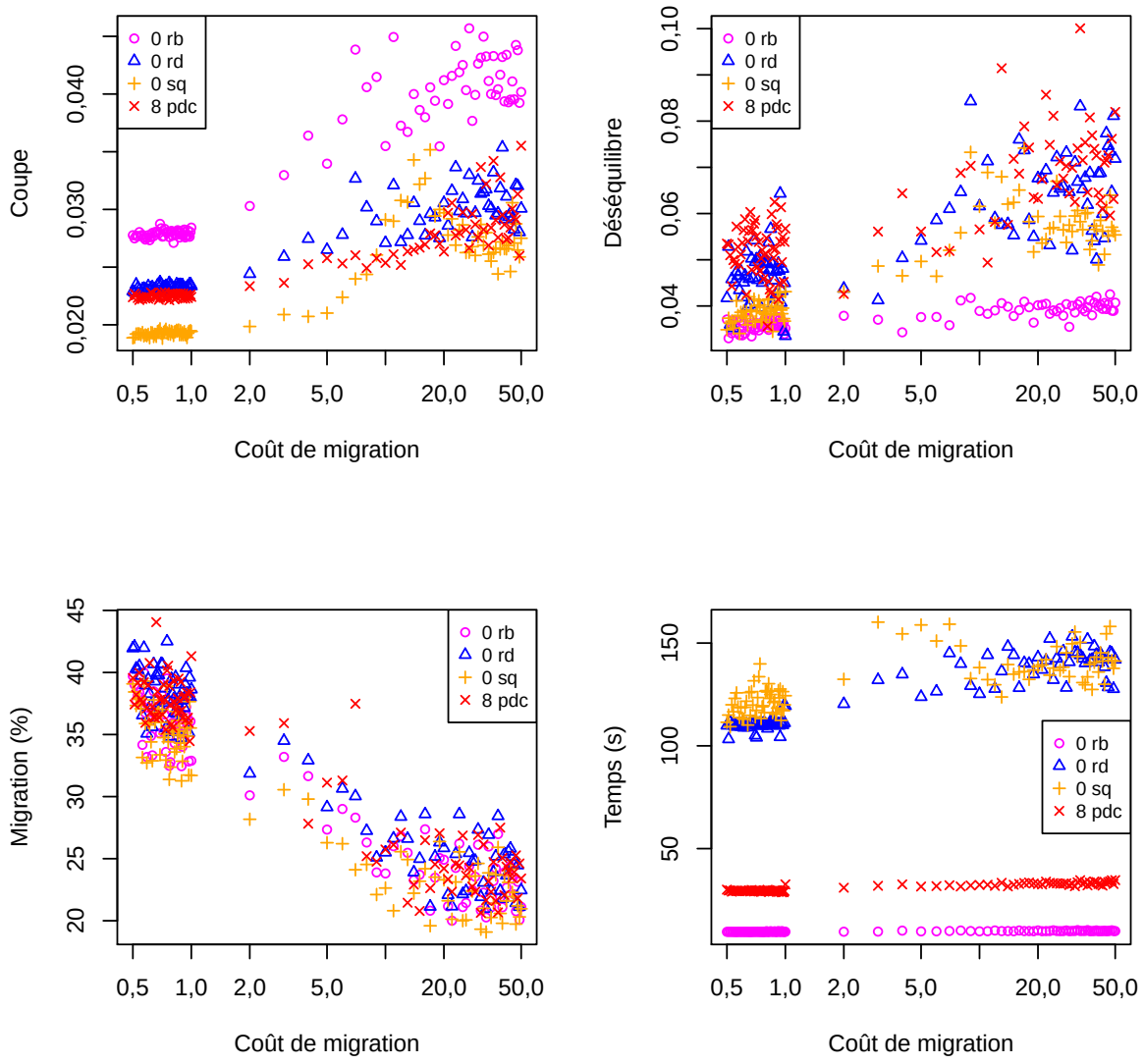


FIGURE 3.11 – Comparaison du comportement de la stratégie parallèle `pdc` sur 8 processeurs avec les stratégies séquentielles `rb`, `rd` et `sq` pour le graphe `10millions`.

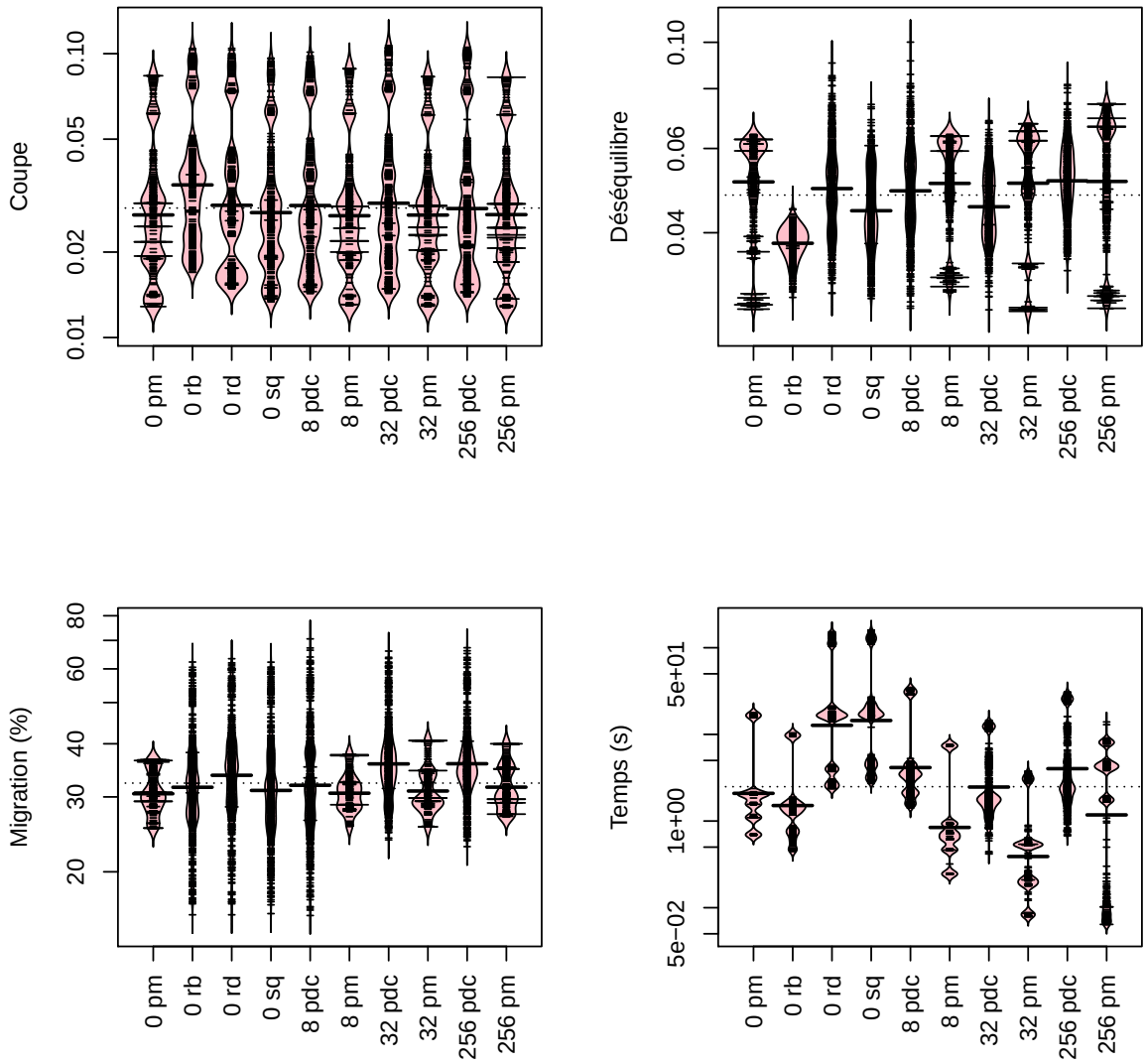


FIGURE 3.12 – Ces graphiques en haricot donnent, pour tous nos résultats avec les stratégies séquentielles `rb`, `rd` et `sq`, ainsi que pour les stratégies parallèles `pdc` et `pm` exécutées sur 8, 32 et 256 processeurs, une estimation de la densité locale pour les quatre indicateurs considérés.

l'image de celles obtenues avec la stratégie `rd`, un peu plus élevées que celles obtenues avec la stratégie `pm`. En termes de déséquilibre, les stratégies `pdc` et `pm` ont, comme les stratégies `sq` et `rd`, une valeur moyenne proche $0,05$. Le déséquilibre obtenu avec la stratégie `pm` a tendance à être plus élevé que celui obtenu avec la stratégie `pdc`. Par ailleurs, comme lors de l'analyse séquentielle, nous notons que la stratégie `pm` est moins sensible au coût de migration. Concernant la migration, nous remarquons que la stratégie `pdc` a tendance à migrer plus de sommets que les autres.

3.6.4 Analyse des stratégies parallèles

Après avoir donné une image globale des tendances des stratégies parallèles, il convient maintenant de revenir sur ces dernières en détail et de proposer une analyse plus fine.

3.6.4.1 Migration

La figure 3.13 de la page suivante, nous permet d'observer — comme nous l'avions déjà fait en séquentiel — que toutes les stratégies de Scotch sont plus sensibles au coût de migration que les stratégies utilisant ParMeTiS. En règle générale, le comportement en termes de migrations de la stratégie `pdc` est proche de celui de la stratégie `rd`. Pour les grands coûts de migration et un nombre de processeurs supérieur à 32, la stratégie `pdc` migre plus que les stratégies `rd` et `pm`. Pour les petits coûts de migration, la stratégie `pm` migre moins de sommets et pour les coûts de migration élevés, toutes les stratégies migrent un pourcentage de sommets, en moyenne, proche de 30 %.

3.6.4.2 Coupe et équilibrage de la charge

Un résumé des résultats obtenus en termes de coupe et d'équilibrage de la charge est présenté en figures 3.15 de la page 94 et 3.16 de la page 95.

La coupe moyenne obtenue avec la stratégie `pdc` sur 32 ($0,035699$, é.t. : $0,02445$) et 256 ($0,034605$, é.t. : $0,02468$) processeurs est, comme nous pouvions nous y attendre, proche de celle de la stratégie séquentielle `rd` ($0,034788$, é.t. : $0,02308$). La stratégie `pm` recueille sur 32 ($0,031349$, é.t. : $0,01909$) et 256 ($0,031496$, é.t. : $0,01909$) processeurs, une coupe de meilleure qualité, proche de celle obtenue par la stratégie `sq` ($0,032127$, é.t. : $0,0202$).

Alors que, sur 32 processeurs, la stratégie `pdc` obtient un déséquilibre de $0,045967$ (é.t. : $0,007764$) — proche de celui obtenu en séquentiel par la stratégie `sq` ($0,045315$, é.t. : $0,008851$) — sur 256 processeurs, la valeur de $0,052096$ (é.t. : $0,008794$) dépasse légèrement la contrainte d'équilibrage de la charge et est alors proche des valeurs moyennes obtenues avec la stratégie `pm` ($0,052423$, é.t. : $0,01212$, sur 32 processeurs et $0,053519$, é.t. : $0,0148$ sur 256 processeurs).

3.6.4.3 Temps d'exécution

En termes de temps d'exécution, nous observons sur les résultats présentés en figure 3.17 que les stratégies parallèles peuvent être jusqu'à dix fois plus rapides que les stratégies séquentielles et que la stratégie `pdc` est en général plus lente que la stratégie `pm`. Alors que la stratégie `pm` est $4,8$ fois plus rapide sur 32 processeurs, elle n'est plus que $2,3$ fois plus rapide sur 256 processeurs.

Rappelons par ailleurs que pour ces temps d'exécution, que ParMeTiS utilise une matrice de taille $p \times p$ (voir section 1.11.3 de la page 37) et que Scotch offre des fonctionnalités supplémentaires, telles que la gestion des sommets fixes.

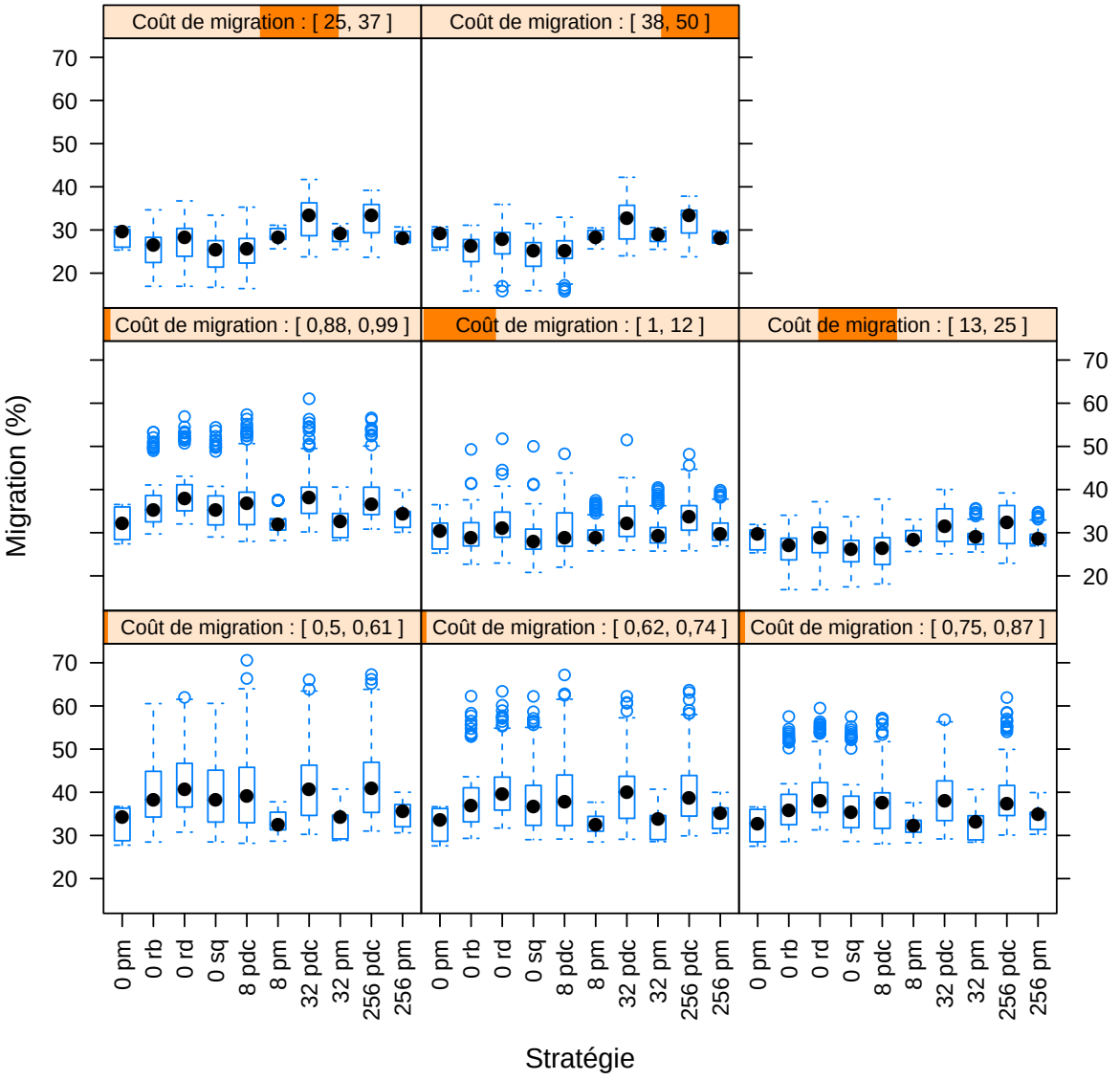


FIGURE 3.13 – Graphique de boîte à moustaches en treillis montrant le comportement des différentes stratégies en termes de migration.

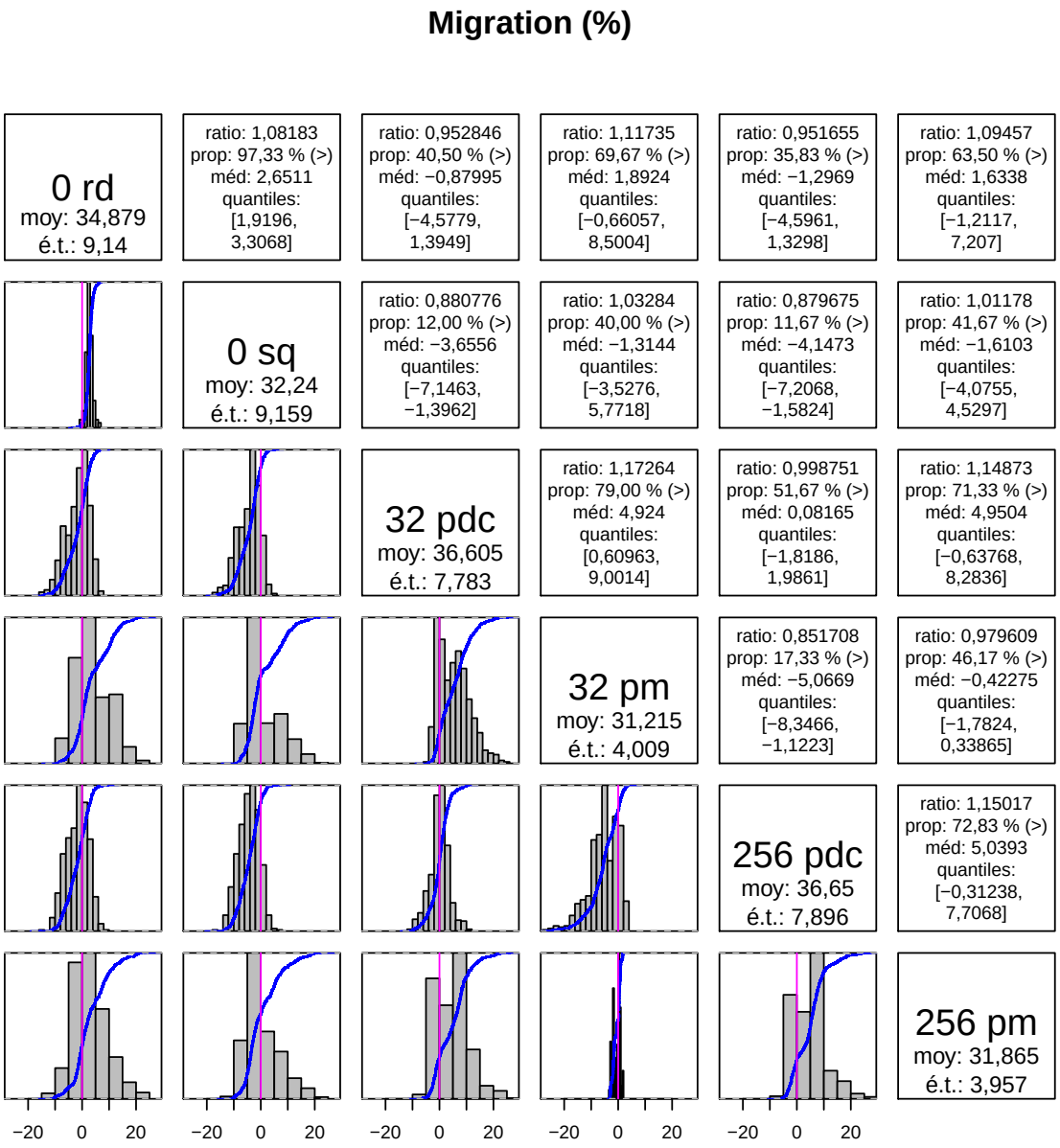


FIGURE 3.14 – Résultats en termes de migration du repartitionnement parallèle. Le nom de la stratégie, sa moyenne et son écart-type apparaissent sur la diagonale. La répartition des différences entre la stratégie à la verticale et celle à droite est fournie sur la partie inférieure de la matrice. La fonction de répartition empirique est tracée en bleu. Sur la partie supérieure, plusieurs métriques sont affichées, afin de faciliter la comparaison entre la stratégie de gauche (g) et celle d'en bas (b), elles sont définies de la manière suivante : `ratio` est égal à $\frac{\text{moy}(g)}{\text{moy}(b)}$; `prop` correspond à la proportion des exécutions où la valeur obtenue avec la stratégie g est supérieure à celle obtenue avec la stratégie b ; `méd` est la médiane de $g - b$. Les 25è et 75è quantiles sont donnés entre crochets.

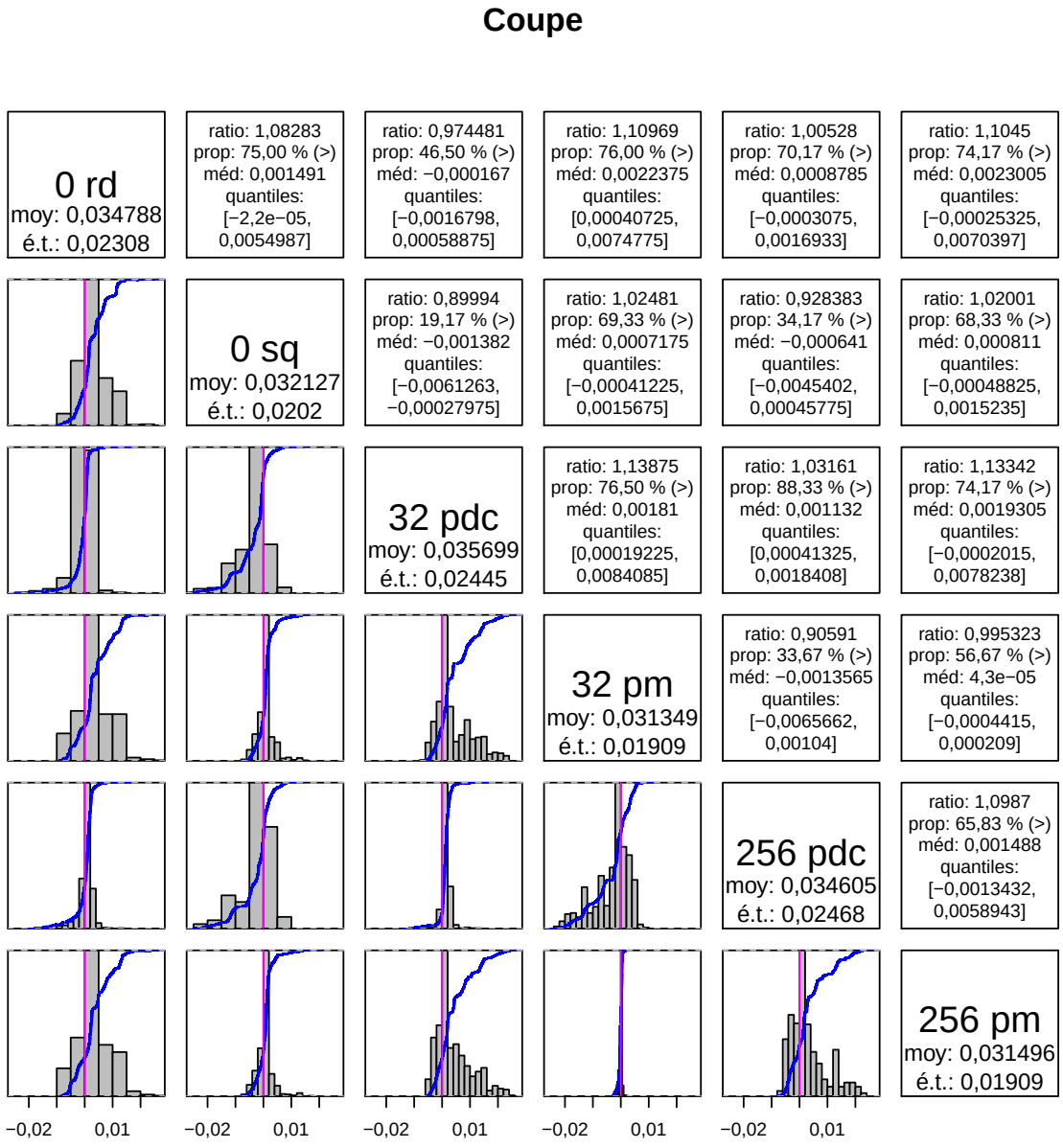


FIGURE 3.15 – Résultats en termes de coupe du repartitionnement parallèle.

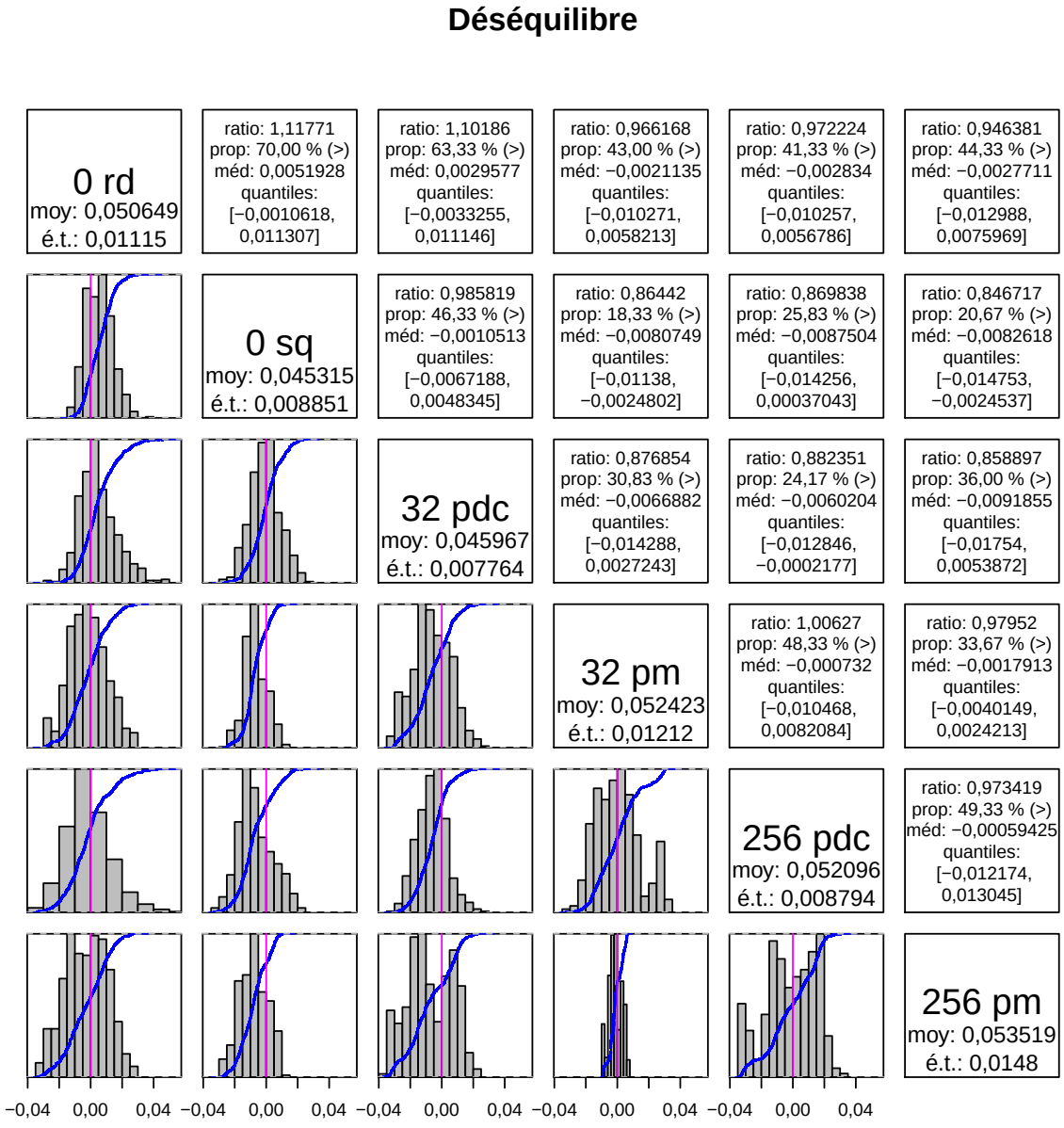


FIGURE 3.16 – Résultats en termes d'équilibrage de la charge du repartitionnement parallèle.

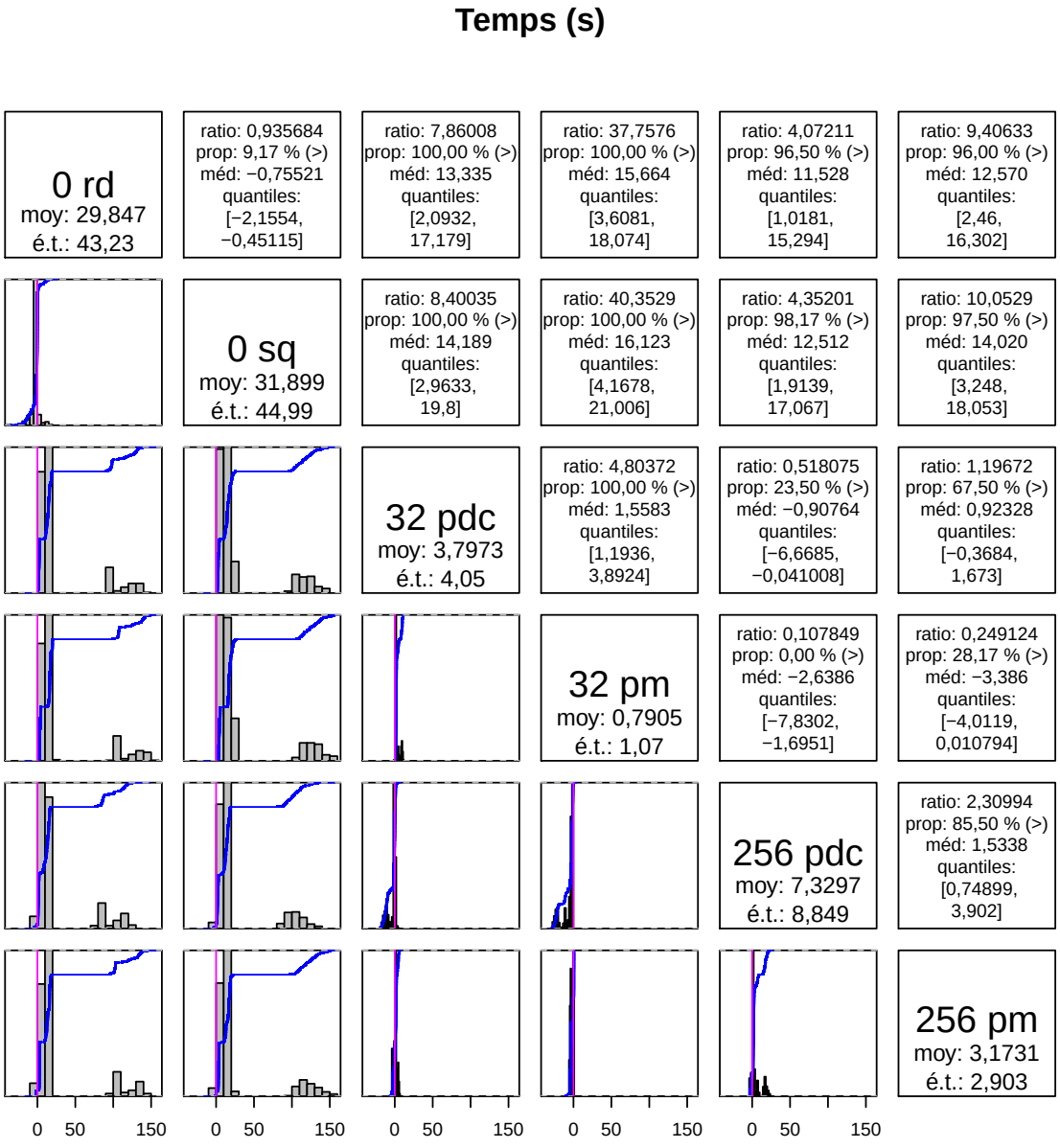


FIGURE 3.17 – Résultats en termes de temps d'exécution du repartitionnement parallèle.

3.6.5 Étude de la *scalabilité* des stratégies

Afin de compléter l'analyse réalisée à la section précédente, il convient maintenant d'étudier le comportement des stratégies de repartitionnement parallèle lorsqu'un nombre plus important de parties et de processeurs est considéré.

3.6.5.1 Protocole

Nous avons considéré, pour l'étude de la *scalabilité*, seulement les graphes de plus grande taille, à savoir : `10millions`, `23millions` et `45millions`. Nous avons suivi un protocole équivalent à celui présenté en section 3.6.1 de la page 75, en considérant toutefois des partitions comprenant 1024 parties. En accord avec les justifications évoquées plus haut, nous nous bornerons à étudier les stratégies parallèles `pdc` et `pm`, ainsi que la stratégie `psr`, qui nous permettra de disposer d'éléments supplémentaires pour nourrir notre analyse. Les résultats que nous présenterons dans cette section ont été obtenus en utilisant jusqu'à 3072 processeurs de l'ordinateur parallèle Titane.

3.6.5.2 Raffinement sur graphes bandes

Le nombre de raffinements par diffusion parallèle sur graphes bandes est illustré pour la stratégie `pdc` en figure 3.18 de la page suivante. Alors que la phase d'expansion se déroule pour les graphes `23millions` et `45millions` avec un nombre de raffinements d'une moyenne trois, les résultats obtenus avec le graphe `10millions` présentent une phase d'expansion avec un seul, voire aucun raffinement. Cette très faible proportion de raffinements effectués lors des 13 niveaux de la phase d'expansion pour le graphe `10millions` s'explique pour le fait qu'avec l'augmentation du nombre de parties, la densité du nombre de sommets à la frontière augmente elle aussi. La distance à la frontière des sommets de certaines parties devient presque toujours inférieure à trois, faisant ainsi échouer la création du graphe bande (voir la section 2.5.2 de la page 45).

3.6.5.3 *Scalabilité* du schéma multi-niveaux

La figure 3.19 de la page 99 présente, pour les graphes `10millions`, `23millions` et `45millions`, les temps d'exécution des trois phases du schéma multi-niveaux parallèle lorsque la stratégie `pdc` est exécutée sur 256, 512, 1024 et 2048 processeurs. Nous remarquons tout d'abord que la phase la moins coûteuse est celle du repartitionnement initial. La phase de contraction est la seconde phase la plus coûteuse. Pour les graphes considérés, son coût augmente proportionnellement au nombre de processeurs, mettant ainsi en évidence le fait que le temps de calcul est négligeable devant celui des communications. Étudier le temps nécessaire à la réalisation de cette phase sur des graphes de plus grande taille nous permettrait de mieux évaluer dans quelle mesure le recouvrement calcul-communication peut avoir lieu.

La phase d'expansion est la phase la plus coûteuse. Plus le graphe est de grande taille — et a donc une densité de sommets à la frontière plus faible — plus les résultats obtenus sur les plus petits nombres de processeurs ont des temps d'exécution proches, signe d'un recouvrement calcul-communication efficace.

3.6.5.4 Analyse par graphes de la *scalabilité* des stratégies

Migration Comme illustré en figure 3.20 de la page 100, le pourcentage de sommets migrés reste stable avec l'augmentation du nombre de processeurs. En effet, la moyenne des écarts-types

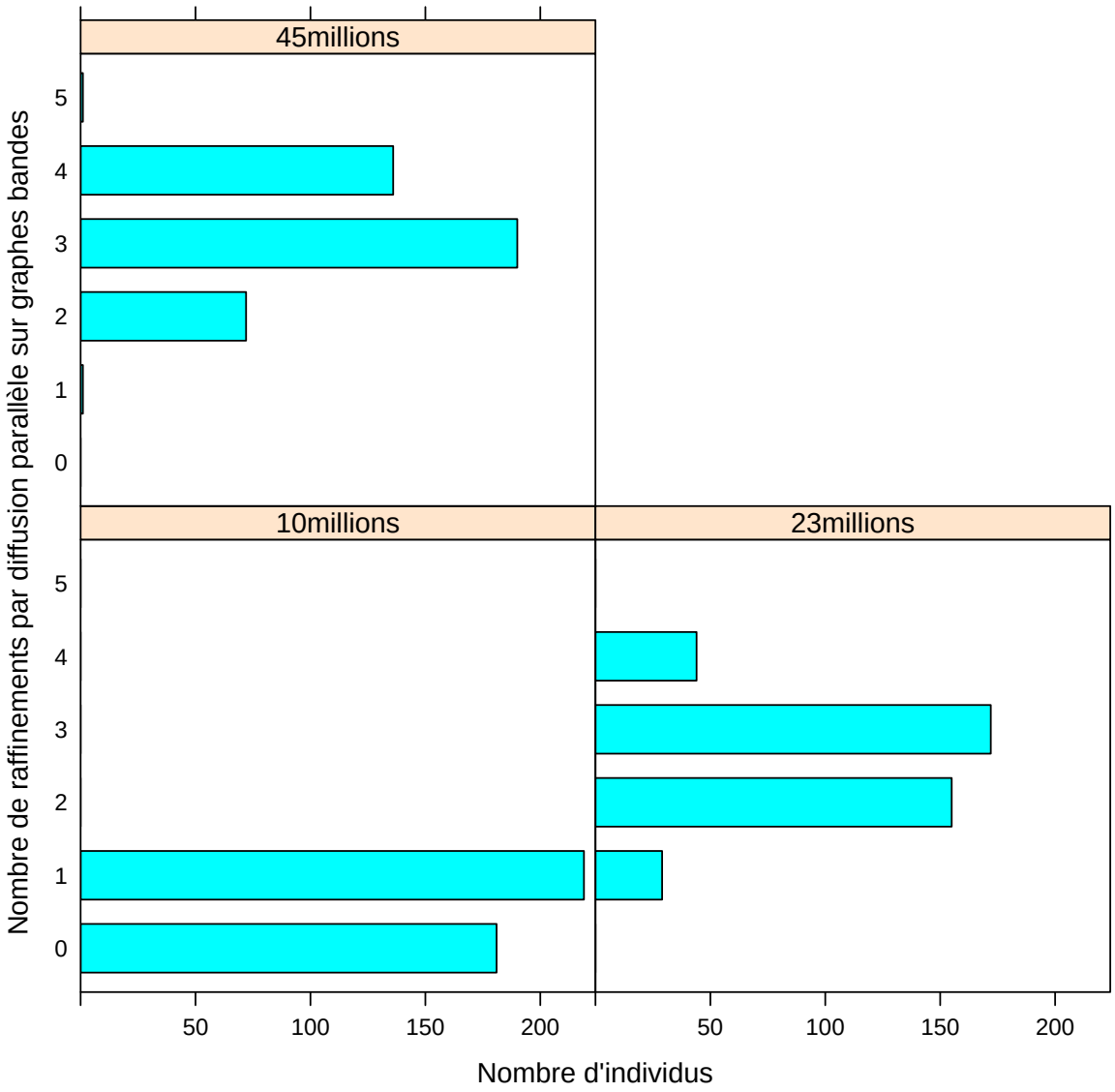


FIGURE 3.18 – Nombre de raffinements par diffusion parallèle sur graphes bandes lors de la phase d'expansion pour les graphes `10millions`, `23millions` et `45millions` et la stratégie `pdc` exécutée sur 256, 512, 1024 et 2048 processeurs. La phase d'expansion s'effectue en 13 niveaux pour le graphe `10millions`, en 14 ou 15 niveaux pour le graphe `23millions` et en 16 niveaux pour le graphe `45millions`.

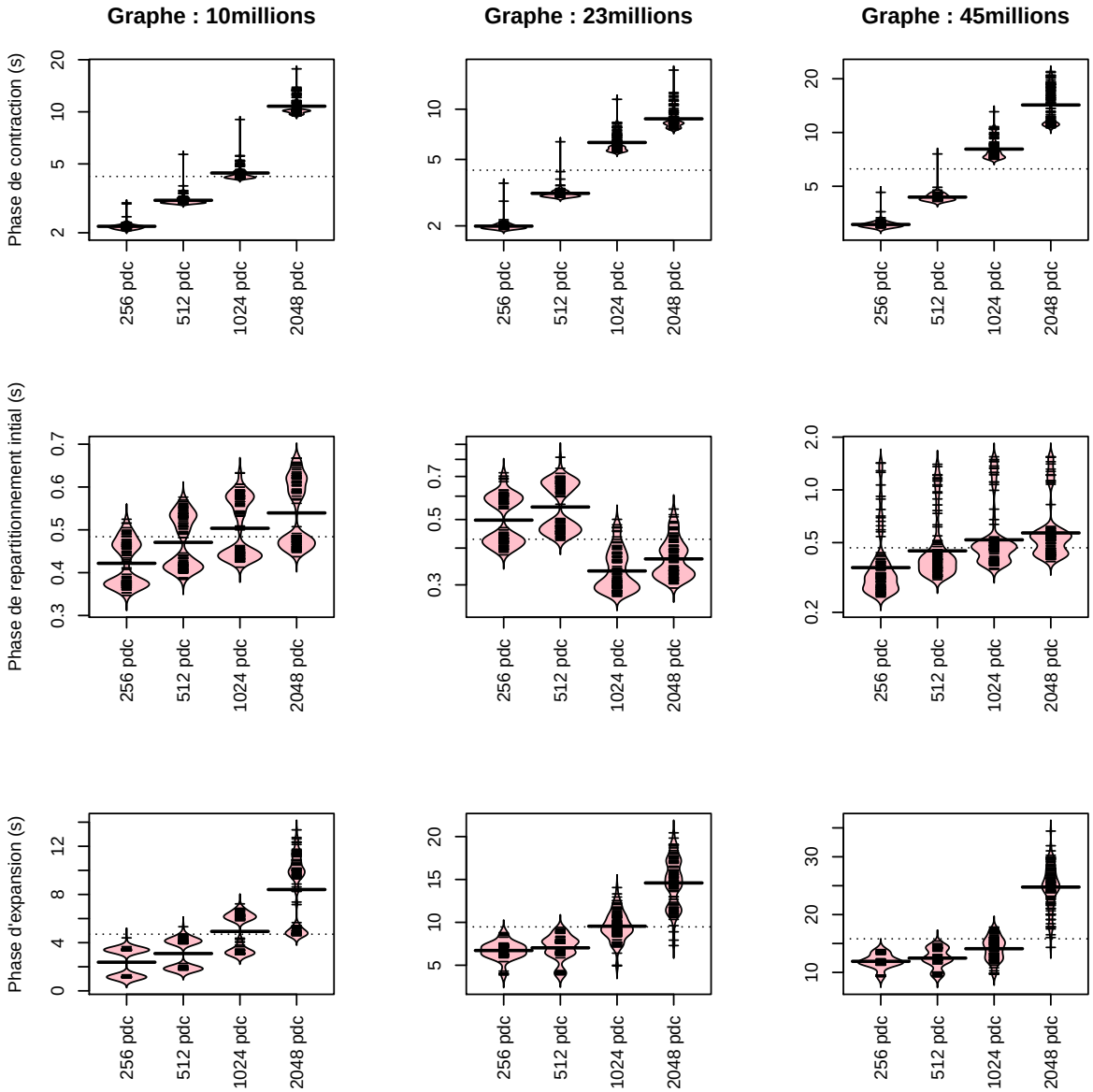


FIGURE 3.19 – Ces graphiques en haricot donnent, pour les graphes `10millions`, `23millions` et `45millions` et la stratégie `pdc` exécutée sur 256, 512, 1024 et 2048 processeurs, une estimation de la densité locale pour les temps d'exécution des trois phases du schéma multi-niveaux parallèle.

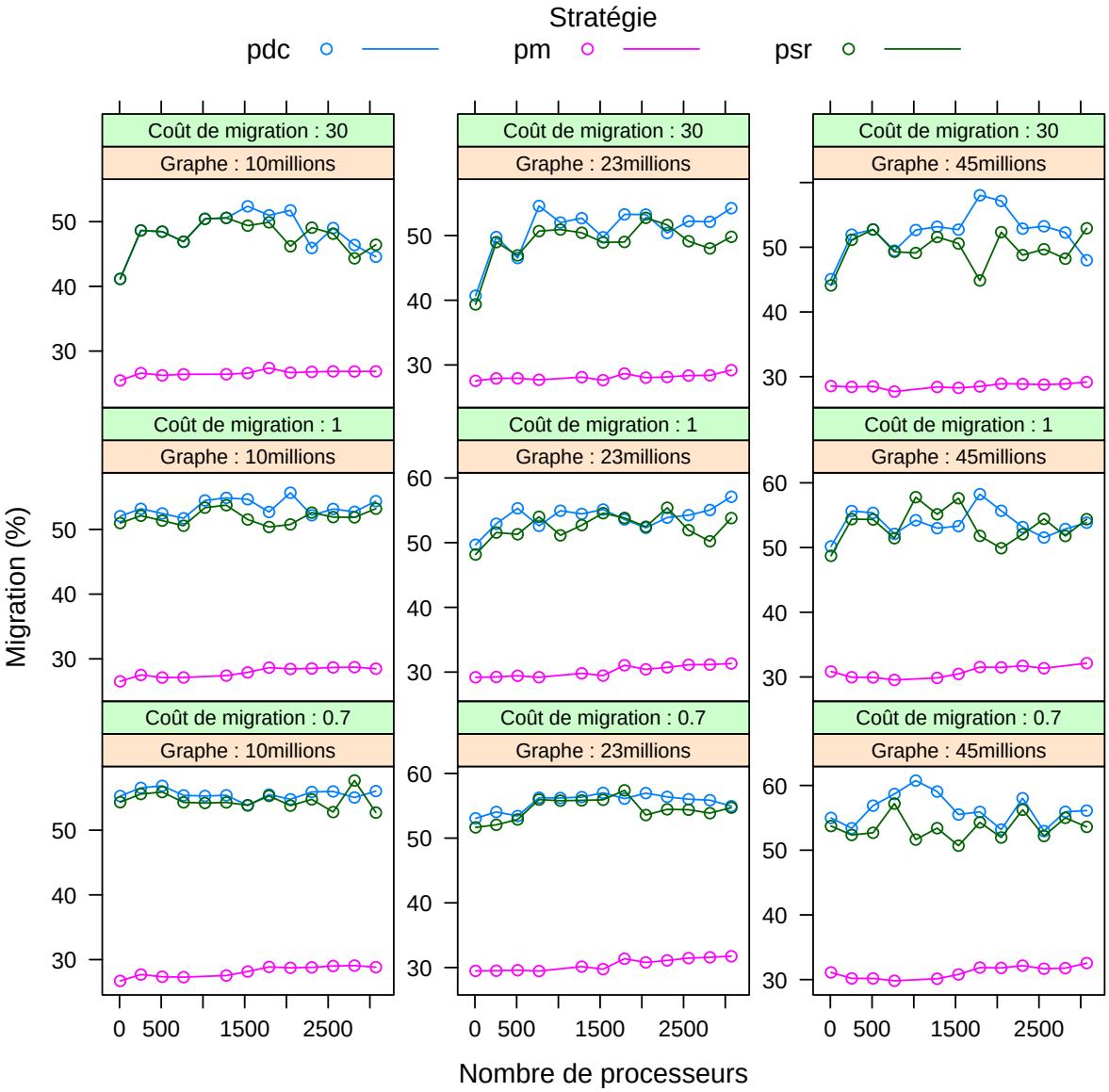


FIGURE 3.20 – Graphique en treillis montrant l'évolution du pourcentage de sommets migrés en fonction du nombre de processeurs.

par stratégie et par graphique est égale à $1,70$. Les stratégies de Scotch, avec des médianes de $55,92$, $53,59$ et $51,75$ pour les coûts de migration 0.7, 1 et 30 pour la stratégie `pdc`, migrent plus que ParMeTiS, dont les médianes sont : $29,97$, $29,54$ et $27,99$.

Coupe La figure 3.21 de la page suivante illustre l'évolution de la coupe en fonction du nombre de processeurs. Nous remarquons tout d'abord que la stratégie `pdc` n'apporte pas d'amélioration notable par rapport à `psr` pour le graphe `10millions` et un coût de migration de 30. Ceci s'explique par le fait que, comme nous l'avons dit plus haut, un unique raffinement est réalisé. Pour la même raison, la stratégie `pdc` a une coupe sensiblement plus élevé que la stratégie `pm` pour le graphe `10millions` ; les différences des médianes pour les coûts de migration $0,7$ et 1 sont égales à $0,0148$ et $0,0153$.

Pour les coûts de migration $0,7$ et 1, le comportement des stratégies est proche. Alors que l'écart moyen des médianes entre les stratégies `pdc` et `pm` est de $0,0083$ pour le graphe `23millions`, il descend à $0,0057$ pour le graphe `45millions`. Pour le coût de migration 30, la coupe obtenue par `pdc` est très proche de `pm` ; nous observons des médianes de $0,0476$, $0,0461$ et $0,0351$, $0,0352$ pour les stratégies `pdc` et `pm` pour les graphes `23millions` et `45millions`.

Enfin, la douce pente décroissante de la coupe obtenue par la stratégie `pdc` avec l'augmentation du nombre de processeurs permet de quantifier l'apport du repartitionnement multi-centralisé.

Équilibrage de la charge La figure 3.22 de la page 103 illustre l'évolution de l'équilibrage de la charge en fonction du nombre de processeurs. Notons tout d'abord que les valeurs d'équilibrage de la charge de la stratégie `pdc` sont très fortement corrélées à celles de la stratégies `psr`. Sur certains graphiques, notamment ceux du graphe `10millions`, l'équilibrage de la charge augmente avec le nombre de processeurs, tout en restant inférieur à la contrainte de $0,05$; ces variations s'expliquent par l'utilisation de la multi-centralisation, qui permet par ailleurs, comme nous l'avons dit plus haut, d'améliorer la coupe.

Pour les expérimentations considérées, les valeurs de déséquilibre sont plus sensibles au coût de migration qu'aux graphes. Les écarts moyens des médianes entre les stratégies `pdc` et `pm` pour les trois graphes confondus sont égaux à 0.0110, $0,0130$ et $0,0260$ pour les coûts de migration $0,7$, 1 et 10.

Temps d'exécution Les résultats en termes de temps d'exécution présentés en figure 3.23 de la page 104 sont similaires pour les deux stratégies `pdc` et `pm`. Les résultats plus succincts présentés en figure 3.17 de la page 96 soulignaient sur 256 processeurs la plus grande rapidité de la stratégie `pm`. Du fait du grand nombre de processeurs utilisés ici, cette plus grande vitesse d'exécution est masquée par le coût des communications, dont les résultats montrent qu'il est équivalent pour les graphes considérés et un nombre de parties égal à 1024.

Bien que l'algorithme de raffinement utilisé par ParMeTiS utilise une matrice de taille $p \times p$, ce coût notable en termes de ressources mémoires n'a pas empêché la bonne exécution de la stratégie `pm` pour le nombre de processeurs considérés.

Une étude plus approfondie du comportement des deux stratégies pourrait considérer des graphes de taille encore plus grande, un nombre de processeurs encore plus grand et un nombre de parties supérieur à 1024.

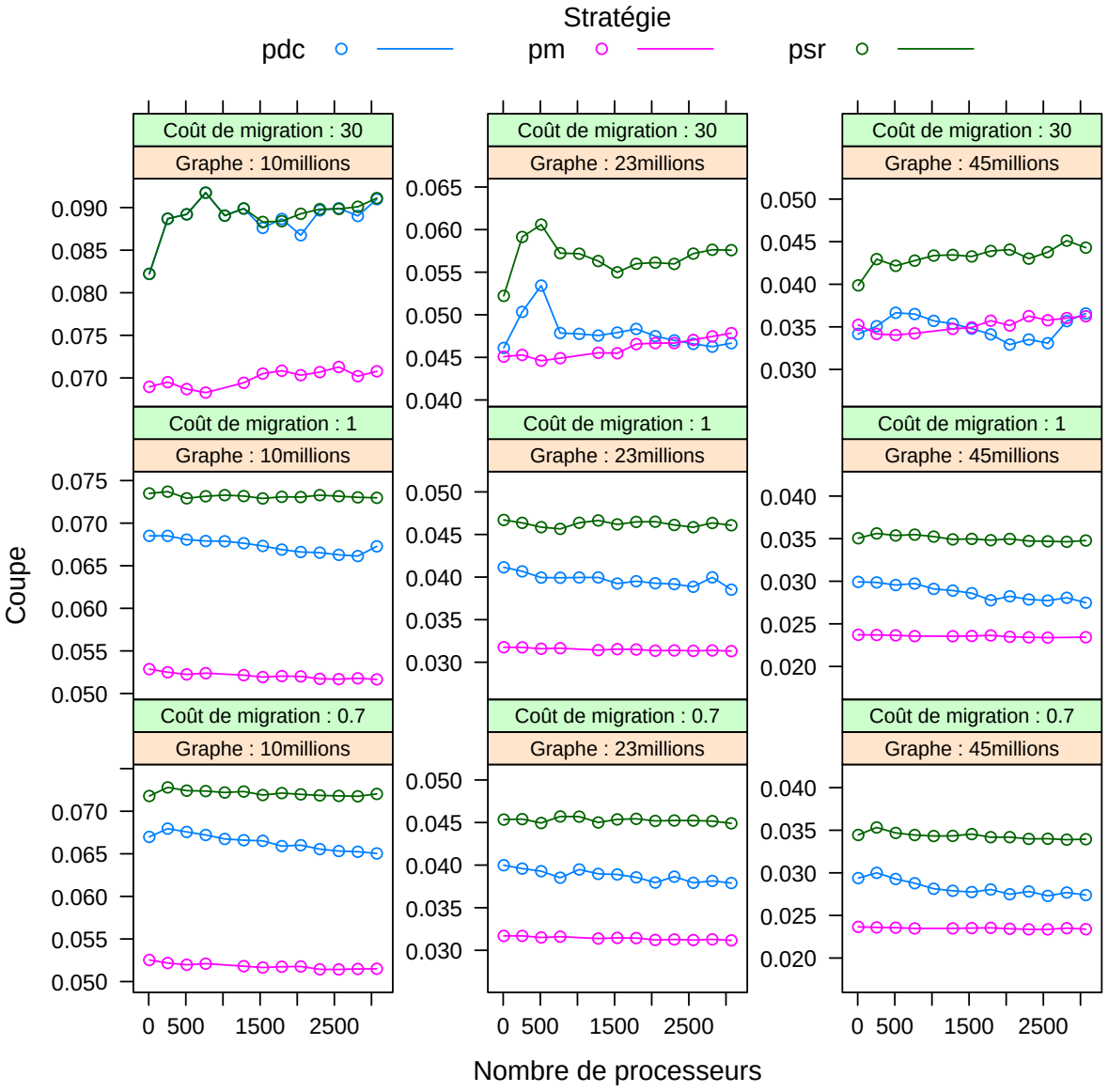


FIGURE 3.21 – Graphique en treillis montrant l'évolution de la coupe en fonction du nombre de processeurs.

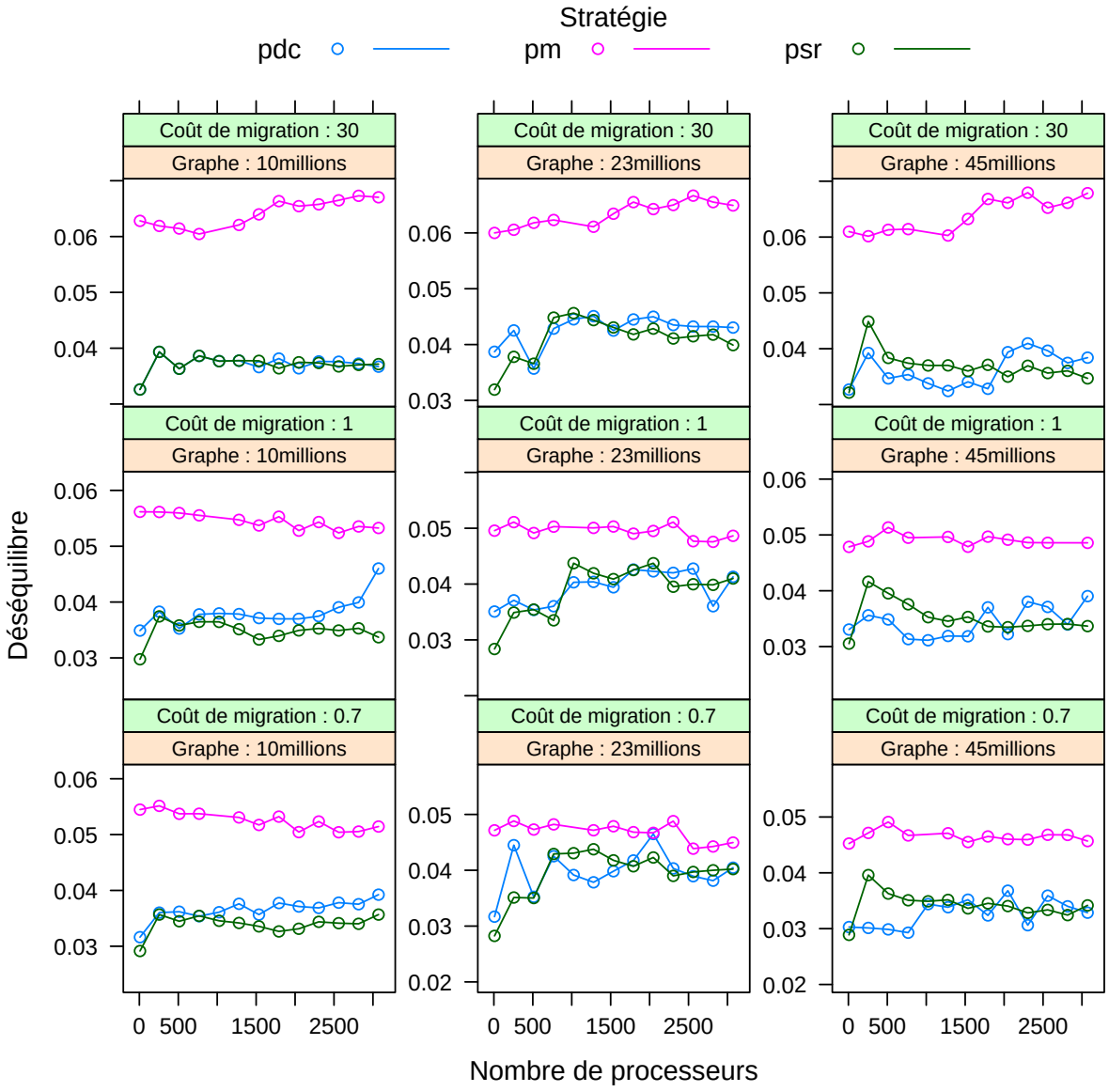


FIGURE 3.22 – Graphique en treillis montrant l'évolution du déséquilibre en fonction du nombre de processeurs.

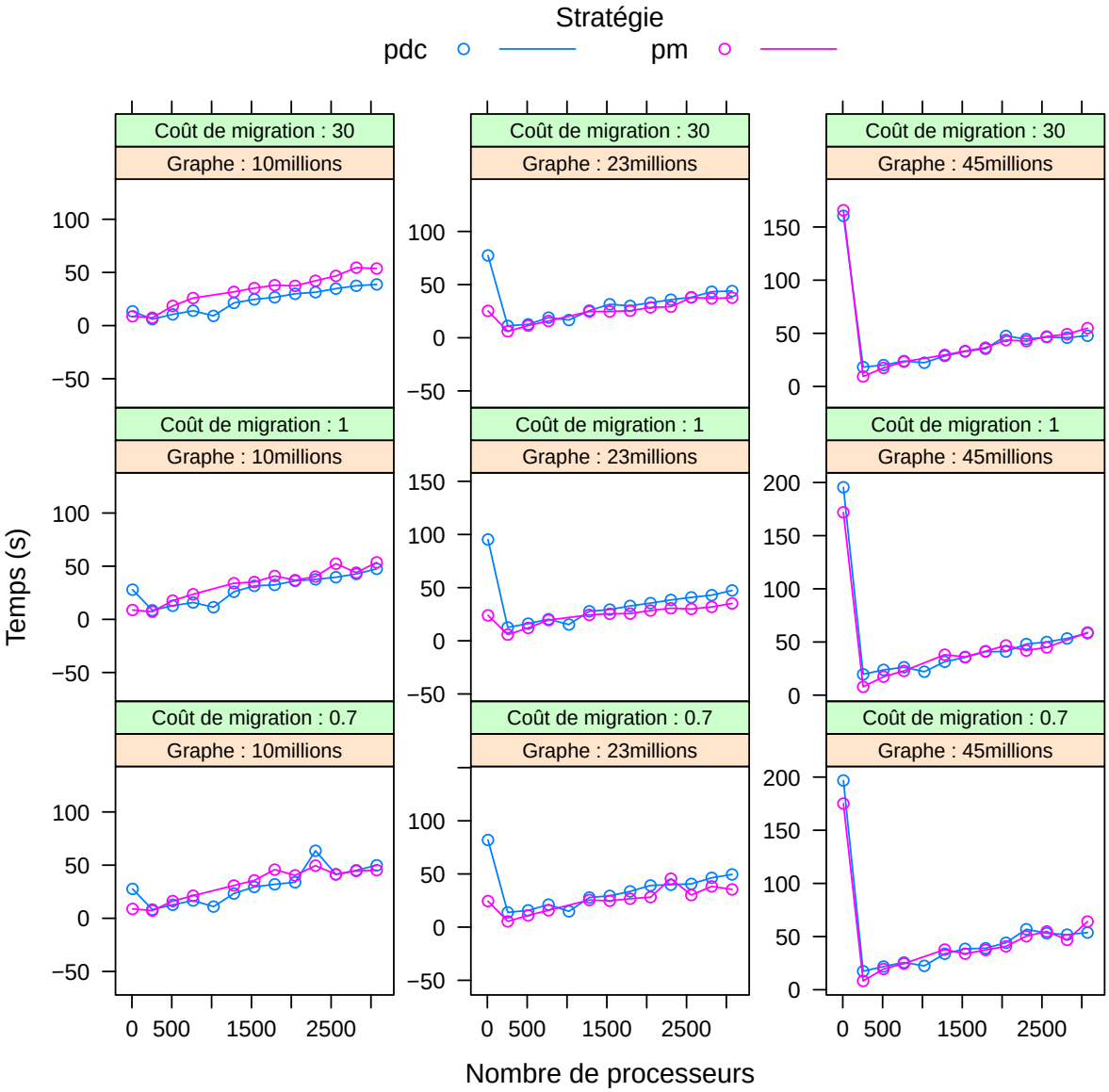


FIGURE 3.23 – Graphique en treillis montrant l'évolution du temps d'exécution en fonction du nombre de processeurs.

3.7 Conclusion

Après avoir présenté la conception d'un ensemble d'algorithmes pour le repartitionnement parallèle de graphes, nous avons étudié expérimentalement les stratégies proposées.

Alors que la multi-centralisation permet d'obtenir un meilleur équilibrage de la charge pour une coupe équivalente, l'utilisation de graphes bandes de taille deux induit une perte d'équilibrage de la charge importante. En conséquence, la stratégie que nous avons proposée et qui répond le mieux à nos attentes est la stratégie `pdc`. Cette stratégie a un comportement proche de la stratégie séquentielle `rd` bien que l'équilibrage de la charge obtenu puisse être de moins bonne qualité sur un nombre de processeurs élevés.

Sur un grand nombre de processeurs, la stratégie que nous proposons obtient un meilleur équilibrage de la charge, mais une coupe de moins bonne qualité et migre plus de sommets que PARMETIS. Notre stratégie est plus coûteuse en termes de temps d'execution, bien que l'écart se réduise fortement lorsque le nombre de processeurs est supérieur à 1000.

Conclusion et perspectives

Conclusion

Ce mémoire présente le travail que nous avons effectué sur la redistribution dynamique parallèle de la charge pour les problèmes numériques de très grande taille.

Nous avons tout d'abord présenté un état de l'art de quelques algorithmes permettant de résoudre les problèmes variés du partitionnement, du repartitionnement, du placement statique et du replacement. En nous appuyant sur cette étude, nous avons défini un ensemble de caractéristiques algorithmiques nécessaires à la réalisation d'un outil de repartitionnement parallèle.

Après avoir proposé une approche bâtie sur les algorithmes utilisés pour résoudre le problème du partitionnement, nous avons dû adapter ces algorithmes au problème du repartitionnement. Outre les nombreux comportements imprévus auquels nous avons fait face et qui nous ont amenés à enrichir nos algorithmes, une de nos contributions majeures, d'un point de vue théorique, a été notre inspiration des méthodes d'influence, afin d'adapter l'algorithme de raffinement par diffusion au problème du repartitionnement et du replacement.

Ce travail théorique a, tout au long de notre thèse, été accompagné par une analyse expérimentale des algorithmes considérés, qui a été rendue possible par l'implémentation de ces derniers au sein de la bibliothèque Scotch.

Dans la première étape de notre travail, nous nous sommes intéressés au contexte d'exécution séquentiel, afin d'établir une première étude de la qualité des algorithmes mis en jeu. Nous avons présenté notre contribution à la conception d'un schéma multi-niveaux k-aire au sein de la bibliothèque Scotch, ainsi que notre adaptation de ce schéma multi-niveaux au calcul du repartitionnement. La partie de cette adaptation la plus exigeante a été la phase d'expansion. En outre, nous avons profité de l'ajout des fonctionnalités de repartitionnement pour mettre en place, au sein de la bibliothèque Scotch, la possibilité d'utiliser des sommets fixes, dont l'usage — souvent conjoint au repartitionnement — est multiple. Afin d'obtenir des résultats de qualité, tout en proposant une stratégie de repartitionnement utilisant des algorithmes se parallélisant facilement, nous avons réalisé une étude approfondie des divers algorithmes et combinaisons de ces derniers pouvant être utilisés pour le raffinement effectué pendant le phase d'expansion. Nous avons, par ailleurs, dans le cadre d'une collaboration avec l'équipe travaillant sur la plate-forme Charm++, effectué une analyse de l'usage de Scotch en tant qu'équilibreur dynamique de la charge au sein de Charm++. Cette étude est venue corroborer nos premiers résultats.

Dans un second temps, nous nous sommes consacré à la parallélisation de ces algorithmes de repartitionnement. L'adaptation du schéma multi-niveaux a nécessité un travail d'adaptation des structures de données mises en œuvre, ainsi que du repliement et de la duplication. Comme pour le repartitionnement séquentiel, la partie la plus ardue a été l'adaptation de la phase d'expansion. Après l'analyse de plusieurs possibilités, nous avons parallélisé avec succès le raffinement par diffusion. Afin d'améliorer les résultats obtenus, nous avons ajouté la possibilité de centraliser un graphe bande et ainsi d'utiliser les algorithmes de raffinement séquentiels sur les premiers

niveaux de la phase d'expansion. Nous avons ensuite réalisé une étude expérimentale des outils de repartitionnement parallèle que nous avons mis en place. Grâce à cette dernière, nous avons pu mettre en exergue l'apport du raffinement centralisé, ainsi que la nécessité d'utiliser des graphes bandes de taille trois. La stratégie parallèle que nous proposons a un comportement proche de la stratégie séquentielle avec raffinement par diffusion. Alors que notre stratégie a tendance à migrer plus que PARMETIS et peut aboutir à une coupe de moins bonne qualité — notamment lorsque les graphes bandes ne peuvent être créés, elle respecte mieux la contrainte d'équilibrage de la charge. En termes de temps d'exécution, PARMETIS est plus rapide, sur un faible nombre de processeurs et les temps d'exécution ont tendance à être proches, sur un nombre de processeurs plus élevé.

Les outils séquentiels de repartitionnement que nous avons mis en place sont disponibles dans la version 6.0 de SCOTCH. Leur implémentation parallèle est actuellement présente dans la version en développement de SCOTCH et il est prévu qu'elle soit mise à disposition des utilisateurs au sein de la prochaine version officielle.

Perspectives

De nombreuses perspectives sont apparues au cours de notre travail. Nous présenterons tout d'abord une liste des perspectives à court terme, qui s'inscrivent dans la poursuite immédiate de nos travaux de thèse.

Concernant le contexte séquentiel, les pistes suivantes mériteraient d'être suivies :

- améliorer la robustesse de l'algorithme de création de graphe bande afin qu'il puisse contenir des parties de largeur variable et ainsi augmenter le nombre de configurations dans lesquelles il pourra être utilisé ;
- analyser les résultats obtenus lorsqu'un raffinement par diffusion est réalisé sur un graphe bande avec des parties de largeur variable ;
- tester les fonctionnalités de replacement séquentiel que nous avons ajoutées à SCOTCH ;
- tester sur un large ensemble de cas l'usage des sommets fixes et les coûts de migration variables.

Comme nous l'avons abordé précédemment, plusieurs pistes nous semblent intéressantes à explorer concernant le contexte parallèle :

- ajouter dans l'algorithme de diffusion de critères de convergence, de sorte que le nombre de passes à effectuer soit choisi dynamiquement lors de l'exécution ;
- étudier plus finement l'impact de la largeur du graphe bande sur le raffinement par diffusion. Explorer la possibilité de modifier cette largeur en fonction du niveau courant de l'expansion ;
- implémenter une version parallèle biaisée de l'heuristique de Fiduccia-Mattheyses, afin d'améliorer l'équilibrage de la charge en l'utilisant sur les niveaux les plus bas de l'expansion parallèle, lorsque la centralisation du graphe bande devient trop coûteuse, mais que le surcoût parallèle de cet algorithme est encore léger ;
- valider notre implémentation des algorithmes de replacement parallèle.

D'autres sujets mériteraient d'être approfondis dans le cadre de nos collaborations :

- étendre les algorithmes de replacement, afin de proposer des fonctionnalités de re-regroupement (*reclustering*) ;
- continuer les collaborations en cours au sein du laboratoire commun INRIA-UIUC sur les sujets connexes suivants :
  - le placement multi-phases,

- l'analyse du placement statique (puis du replacement) sur des cas réels et leur intégration dans une bibliothèque, pour la rendre disponible via une interface MPI,
- l'analyse du replacement et du repartitionnement parallèle afin de continuer le travail en cours avec l'équipe de Charm++.

À plus long terme, nous pouvons donner plusieurs directions méritant d'être explorées afin d'obtenir les performances attendues sur les architectures *petascale* et *exascale*.

Il convient de continuer à améliorer la *scalabilité* des stratégies proposées. La robustesse de l'algorithme de diffusion doit être améliorée. D'autres algorithmes similaires peuvent également être considérés, tels que des algorithmes de raffinement à jetons pouvant s'inspirer du comportement des colonies de fourmis ou d'abeilles.

La prise en compte de l'hétérogénéité de l'architecture cible lors de la répartition de la charge est un autre élément essentiel pour le passage à l'échelle. L'amélioration de la performance des algorithmes de replacement que nous avons proposés nous semble être le premier pas à effectuer dans cette direction.

Enfin, une limite qui apparaît sur les architectures *exascale* est l'important coût des synchronisations lors des échanges de données de type *halo*. Afin de pouvoir profiter pleinement de ces architectures, il convient donc de conceptualiser des algorithmes de repartitionnement faisant la plus large place possible aux calculs et aux communications asynchrones.

Références bibliographiques

[AZ04] S. Areibi et Y. Zeng : Effective memetic algorithms for VLSI design automation = genetic algorithms + local search + multi-level clustering. *Evolutionary Computation*, 12(3):327–353, 2004.

[BB99] R. Battiti et A. A. Bertossi : Greedy, prohibition, and reactive heuristics for graph partitioning. *IEEE Transactions on Computers*, 48(4):361–385, 1999.

[BDF+99] E.G. Boman, K.D. Devine, L.A. Fisk, R. Heaphy, B. Hendrickson, V. Leung, C. Vaughan, U.V. Çatalyurek, D. Bozdag et W. Mitchell : Zoltan home page. `http://www.cs.sandia.gov/Zoltan`, 1999.

[BFM+12] A. Bhatele, S. Fourestier, H. Menon, L. Kale et F. Pellegrini : Applying graph partitioning methods in measurement-based dynamic load balancing. Rapport technique PPL 2012, Department of Computer Science, University of Illinois at Urbana-Champaign, March 2012.

[BHJL89] T. Bui, C. Heigham, C. Jones et T. Leighton : Improving the performance of the kernighan-lin and simulated annealing graph bisection algorithms. *In DAC '89 : Proceedings of the 26th ACM/IEEE conference on Design automation*, pages 775–778, New York, NY, USA, 1989. ACM Press.

[BK00] R. K. Brunner et L. V. Kalé : Handling application-induced load imbalance using parallel objects. *In Parallel and Distributed Computing for Symbolic and Irregular Applications*, pages 167–181. World Scientific Publishing, 2000.

[blu] About the blue waters project. `http://www.ncsa.illinois.edu/BlueWaters/`.

[BM88] S. W. Bollinger et S. F. Midkiff : Processor and link assignment in multicomputers using simulated annealing. *In Proceedings of the 11th Int. Conf. on Parallel Processing*, pages 1–7. The Penn. State Univ. Press, août 1988.

[BM96] T. N. Bui et B. R. Moon : Genetic algorithm and graph partitioning. *IEEE Trans. Comput.*, 45(7):841–855, 1996.

[Bok81] S. H. Bokhari : On the mapping problem. *IEEE Transactions on Computing*, C-30(3):207–214, 1981.

[BS94] S. T. Barnard et H. D. Simon : A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency : Practice and Experience*, 6(2):101–117, 1994.

[CA92] T. Chockalingam et S. Arunkumar : A randomized heuristics for the mapping problem : The genetic approach. *Parallel Computing*, 18:1157–1165, 1992.

[cA99] U.V. Çatalyurek et C. Aykanat : Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. on Par. and Dist. Syst.*, 10(7):673–693, juillet 1999.

[cA01] U.V. Çatalyurek et C. Aykanat : A hypergraph-partitioning approach for coarse-grain decomposition. *In Proc. ACM/IEEE conference on Supercomputing (CDROM)*, pages 28–es, New York, NY, USA, 2001.

[cBB+07] U.V. Çatalyurek, D. Bozdag, E.G. Boman, K.D. Devine, R.T. Heaphy et L.A. Riesen : Hypergraph-based dynamic partitioning and load balancing. Tech. Report SAND2007-0043P, Sandia National Laboratories, Albuquerque, NM, 2007.

[cBD+07] U.V. Çatalyurek, E.G. Boman, K.D. Devine, D. Bozdag, R.T. Heaphy et L.A. Riesen : Hypergraph-based dynamic load balancing for adaptive scientific computations. *In Proc. of 21st International Parallel and Distributed Processing Symposium (IPDPS'07)*. IEEE, 2007.

[cBD+08] U.V. Çatalyurek, E.G. Boman, K.D. Devine, D. Bozdag, R.T. Heaphy et L.A. Riesen : A repartitioning hypergraph model for dynamic load balancing. Tech. Report SAND2008-2304J, Sandia National Laboratories, Albuquerque, NM, 2008.

[CeC] CeCILL : "CEA-CNRS-INRIA Logiciel Libre" free/libre software license. Available from `http://www.cecill.info/licenses.en.html`.

[Cer85] V. Cerny : A thermodynamical approach to the travelling salesman problem : an efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51, 1985.

[Che07] C. Chevalier : *Conception et mise en oeuvre d'outils efficaces pour le partitionnement et la distribution parallèles de problèmes numériques de très grande taille.* Thèse de Doctorat, LaBRI, Université Bordeaux I, 351 cours de la Libération, 33405 Talence, France, septembre 2007.

[CP06a] C. Chevalier et F. Pellegrini : Improvement of the efficiency of genetic algorithms for scalable parallel graph partitioning in a multi-level framework. *In Proc. Euro-Par'06, LNCS 4128*, pages 243–252, septembre 2006.

[CP06b] Cédric Chevalier et François Pellegrini : PT-Scotch : Un outil pour la renumérotation parallèle efficace de grands graphes dans un contexte multi-niveaux. *In Actes de RenPar'17 / SympA'2006 / CFSE'5 / JC'2006*, page 8 pages, Canet en Roussillon, France, octobre 2006. 8 pages.

[CP08] C. Chevalier et F. Pellegrini : PT-Scotch : A tool for efficient parallel graph ordering. *Parallel Computing*, 34:318–331, 2008.

[CS09a] C. Chevalier et I. Safro : Comparison of coarsening schemes for multilevel graph partitioning. *In* Thomas Stützle, éditeur : *Learning and Intelligent Optimization*, pages 191–205. Springer-Verlag, Berlin, Heidelberg, 2009.

[CS09b] C. Chevalier et I. Safro : Weighted aggregation for multi-level graph partitioning. *In Combinatorial Scientific Computing*, numéro 09061 de Dagstuhl Seminar Proceedings. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany, 2009.

[Cyb89] G. Cybenko : Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.*, 7:279–301, octobre 1989.

[DGRW12] D. Delling, A. V. Goldberg, I. Razenshteyn et R. F. Werneck : Exact combinatorial branch-and-bound for graph bisection. *In Graph Partitioning and Graph Clustering*, 10th DIMACS Implementation Challenge Workshop, Georgia

Institute of Technology, Atlanta, GA, février 2012. Contemporary Mathematics 588. American Mathematical Society and Center for Discrete Mathematics and Theoretical Computer Science, 2013.

[DPSW98] R. Diekmann, R. Preis, F. Schlimbach et C. Walshaw : Aspect Ratio for Mesh Partitioning. *In* D. Pritchard et J. Reeve, éditeurs : *Euro-Par'98 Parallel Processing*, volume 1470 de *LNCS*, pages 347–351. Springer, Berlin, 1998.

[dWJ03] Rob F. Van der Wijngaart et Haoqiang Jin : NAS Parallel Benchmarks, Multi-Zone Versions. Rapport technique NAS Technical Report NAS-03-010, NASA Advanced Supercomputing (NAS) Division, July 2003.

[ERS90] F. Ercal, J. Ramanujam et P. Sadayappan : Task allocation onto a hypercube by recursive mincut bipartitionning. *Journal of Parallel and Distributed Computing*, 10:35–44, 1990.

[Fah88] C. Fahrat : A simple and efficient automatic FEM domain decomposer. *Computers and Structures*, 28(5):579–602, 1988.

[Fie73] M. Fiedler : Algebraic connectivity of graphs. *Czechoslovak Math. J.*, 23:298–305, 1973.

[Fie75] M. Fiedler : A property of eigenvectors of non-negative symmetric matrices and its application to graph theory. *Czechoslovak Math. J.*, 25:619–633, 1975.

[FM82] C. M. Fiduccia et R. M. Mattheyses : A linear-time heuristic for improving network partitions. *In Proceedings of the 19th Design Automation Conference*, DAC '82, pages 175–181, Piscataway, NJ, USA, 1982. IEEE Press.

[FP11] S. Fourestier et F. Pellegrini : Adaptation au repartitionnement de graphes d'une méthode d'optimisation globale par diffusion. *In Actes des 20e Rencontres francophones du parallélisme (RenPar'11)*, Saint-Malo, France, mai 2011.

[FP13] S. Fourestier et F. Pellegrini : Toward a scalable refinement strategy for multilevel graph repartitioning. Rapport de recherche RR-8246, INRIA, février 2013.

[GH08] J. Gaidamour et P. Hénon : A parallel direct/iterative solver based on a Schur complement approach. *In Proc. 11^{th} Int. Conf. on Comp. Sci. and Eng.*, pages 98–105, Sao Paulo, 2008.

[GJ79] M. R. Garey et D. S. Johnson : *Computers and Intractablility : A Guide to the Theory of NP-completeness*. W. H. Freeman, San Francisco, 1979.

[GJS76] M. R. Garey, D. S. Johnson et L. Stockmeyer : Some simplified NP-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[Glo89] F. Glover : Tabu search – part 1. *ORSA Journal of Computing*, 1(3):190–206, 1989.

[Haj88] B. Hajek : Cooling schedules for optimal annealing. *Mathematics of Operations Research*, 13(2):311–329, 1988.

[Hal70] K. M. Hall. : An r-dimensional quadratic placement algorithm. *Management Science*, 17(3):219–229, 1970.

[HBE98] Y. F. Hu, R. J. Blake et D. R. Emerson : An optimal migration algorithm for dynamic load balancing. concurrency : Practice and experience, 1998.

[Hen98] B. Hendrickson : Graph partitioning and parallel solvers : Has the emperor no clothes ? *In IRREGULAR'98 : solving irregularly structured problems in parallel*, numéro 1457 de LNCS, pages 218–225, août 1998.

[HG86] P. Hansen et W. C. Giauque : Task allocation in distributed processing systems. *Operations Research Letters*, 5(3):137–143, août 1986.

[HHK97] L. Hagen, D. Huang et A. Kahng : On implementation choices for iterative improvement partitioning algorithms, 1997.

[HL93] B. Hendrickson et R. Leland : Multidimensional spectral load balancing. Rapport technique SAND93–0074, Sandia National Laboratories, janvier 1993.

[HL95] B. Hendrickson et R. Leland : A multilevel algorithm for partitioning graphs. *In Proceedings of Supercomputing*, 1995.

[HLVD97] B. Hendrickson, R. Leland et R. Van Driessche : Skewed graph partitioning. *In Proceedings of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, Albuquerque, NM, mars 1997. IEEE.

[Hol75] J. Holland : *Adaptation in Natural and Artificial Systems.* University of Michigan Press, Ann Harbor, 1975.

[HSS10] M. Holtgrewe, P. Sanders et C. Schulz : Engineering a scalable high quality graph partitioner. *In Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, 2010.

[joi] Joint laboratory for petascale computing. `http://jointlab.ncsa.illinois.edu/publications.html`.

[Kam08] P. Kampstra : Beanplot : A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software, Code Snippets*, 28(1):1–9, 2008.

[kar] MeTiS : Family of multilevel partitioning algorithms. `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[Kar02] G. Karypis : Multilevel hypergraph partitioning. Rapport technique 02-025, University of Minnesota, 2002.

[KGV83] S. Kirkpatrick, C. D. Gelatt et M. P. Vecchi : Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[KK93] L.V. Kale et S. Krishnan : Charm++ : a portable concurrent object oriented system based on c++. *SIGPLAN Not.*, 28(10):91–108, octobre 1993.

[KK95a] G. Karypis et V. Kumar : Analysis of multilevel graph partitioning. *In Proc. ACM/IEEE conference on Supercomputing (CDROM)*, pages 19–es, décembre 1995.

[KK95b] G. Karypis et V. Kumar : Multilevel graph partitioning schemes. *In Proc. 24th Intern. Conf. Par. Proc., III*, pages 113–122. CRC Press, 1995.

[KK95c] George Karypis et Vipin Kumar : Analysis of multilevel graph partitioning. *In Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '95, New York, NY, USA, 1995. ACM.

[KK98] G. Karypis et V. Kumar : Metis *A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices.* University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis, MN 55455, U.S.A., septembre 1998.

[KL70] B. W. Kernighan et S. Lin : An efficient heuristic procedure for partitioning graphs. *BELL System Technical Journal*, pages 291–307, février 1970.

[KR03] A. Kaveh et H. A. B. Rahimi : A hybrid graph-genetic method for domain decomposition. *Finite. Elem. Anal. Des.*, 29:1237–1247, 2003.

[Kri84] B. Krishnamurthy : An improved min-cut algorithm for partitioning vlsi networks. *IEEE Trans. Computers*, 33(5):438–446, 1984.

[KT98] A. I. Khan et B. H. V. Topping : Subdomain generation for parallel finite element analysis. *Comput. Syst. Eng.*, 4:96–129, 1998.

[KvR04] P. Korošec, J. Šilc et B. Robič : Solving the mesh-partitioning problem with an ant-colony algorithm. *Parallel Computing*, 30(5-6):785–801, 2004.

[LG99] A. E. Langham et P. W. Grant : Using competing ant colonies to solve k-way partitioning problems with foraging and raiding strategies. *In ECAL '99 : Proceedings of the 5th European Conference on Advances in Artificial Life*, pages 621–625, London, UK, 1999. Springer-Verlag.

[Lo84] V. M. Lo : Heuristic algorithms for task assignment in distributed systems. *In International Conference on Distributed Computer Systems*, pages 30–39. IEEE, 1984.

[Mey12] H. Meyerhenke : Shape optimizing load balancing for parallel adaptive numerical simulations using mpi. *In Graph Partitioning and Graph Clustering*, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, février 2012. Contemporary Mathematics 588. American Mathematical Society and Center for Discrete Mathematics and Theoretical Computer Science, 2013.

[MLT82] P. R. Ma, E. Y. S. Lee et M. Tsuchiya : A task allocation model for distributed computing systems. *IEEE Transactions on Computers*, C-31(1):41–47, janvier 1982.

[MMS06] H. Meyerhenke, B. Monien et S. Schamberger : Accelerating shape optimizing load balancing for parallel fem simulations by algebraic multigrid. *In Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 57–57, Washington, DC, USA, 2006. IEEE Computer Society.

[MMS09] H. Meyerhenke, B. Monien et T. Sauerwald : A new diffusion-based multilevel algorithm for computing graph partitions. *JPDC*, 69:750–761, septembre 2009.

[MPD00] B. Monien, R. Preis et R. Diekmann : Quality matching and local improvement for multilevel graph-partitioning. *Parallel Computing*, 26(12):1609–1634, 2000.

[mpi97] Mpi-2.0 standards. `http://www.mpi-forum.org/docs/docs.html`, 1997.

[MRR+53] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller et E. Teller : Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, juin 1953.

[MS05] H. Meyerhenke et S. Schamberger : Balancing parallel adaptive fem computations by solving systems of linear equations. *In Proceedings of the 11th international Euro-Par conference on Parallel Processing*, Euro-Par'05, pages 209–219, Berlin, Heidelberg, 2005. Springer-Verlag.

[MS06] H. Meyerhenke et S. Schamberger : A parallel shape optimizing load balancer. *In Proc. Europar, Dresden, LNCS 4128*, pages 232–242, septembre 2006.

[MT91] T. Muntean et E.-G. Talbi : A parallel genetic algorithm for process-processors mapping. *High performance computing*, 2:71–82, 1991.

[Nic94] D. M. Nicol : Rectilinear partitioning of irregular data parallel computations. *Journal of Parallel and Distributed Computing*, 23:119–134, 1994.

[NORL86] B. Nour-Omid, A. Raefsky et G. Lyzenga : Solving finite element equations on concurrent computers. *In* A. K. Noor, éditeur : *Parallel Computations and Their Impact on Mechanics*, pages 209–227. ASME Press, 1986.

[OB98] L. Oliker et R. Biswas : Plum : Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 52:150–177, 1998.

[Pap76] C. H. Papadimitriou : The NP-completeness of the bandwidth minimization problem. *Computing*, 16:263–270, 1976.

[pel] Scotch : Static mapping, graph partitioning, and sparse matrix block ordering package. `http://www.labri.fr/~pelegrin/scotch/`.

[Pel95] F. Pellegrini : *Application de méthodes de partition à la résolution de problèmes de graphes issus du parallélisme*. Thèse de Doctorat, LaBRI, Université Bordeaux I, 351 cours de la Libération, 33405 Talence, France, janvier 1995.

[Pel07] 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.

[PSL90] A. Pothen, H. D. Simon et K.-P. Liou : Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis*, 11(3):430–452, juillet 1990.

[RH89] C. Roucairol et P. Hansen : Cut cost minimization in graph partitioning. *Numerical and Applied Mathematics*, pages 585–587, 1989.

[RMR09] A. Rama Mohan Rao : Distributed evolutionary multi-objective mesh-partitioning algorithm for parallel finite element computations. *Comput. Struct.*, 87(23-24):1461–1473, décembre 2009.

[RMRARD02] A. Rama Mohan Rao, T. V. S. R. Appa Rao et B. Dattaguru : Automatic decomposition of unstructured meshes employing genetic algorithms for parallel FEM computations. *Int. J. Struct. Eng. Mech.*, 14:625–647, 2002.

[San89] L.A. Sanchis : Multiple-way network partitioning. *Computers, IEEE Transactions on*, 38(1):62 –81, janvier 1989.

[Sar08] D. Sarkar : *Lattice : Multivariate Data Visualization with R*. Springer, New York, 2008.

[Sin87] J. B. Sinclair : Efficient computation of optimal assignments for distributed tasks. *Journal of Parallel and Distributed Computing*, 4:342–362, 1987.

[SKK97] K. Schloegel, G. Karypis et V. Kumar : Parallel multilevel diffusion algorithms for repartitioning of adaptive meshes. Rapport technique 97-014, University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, 1997.

[SKK01] K. Schloegel, G. Karypis et V. Kumar : Wavefront diffusion and lmsr : Algorithms for dynamic repartitioning of adaptive meshes. *Trans. Parallel Distrib. Syst.*, 12:451–466, mai 2001.

[SS12] P. Sanders et C. Schulz : High quality graph partitioning. *In Graph Partitioning and Graph Clustering*, 10th DIMACS Implementation Challenge Workshop, Georgia Institute of Technology, Atlanta, GA, février 2012. Contemporary Mathematics 588. American Mathematical Society and Center for Discrete Mathematics and Theoretical Computer Science, 2013.

[ST85] C.-C. Shen et W.-H. Tsai : A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Transactions on Computers*, C-34(3):197–203, mars 1985.

[ST93] H. D. Simon et S.-H. Teng : How good is recursive bipartition. Rapport de recherche, NASA Ames Research Center, juin 1993.

[Sto77] H. S. Stone : Multiprocessor scheduling with the aid of network flow algorithms. *IEEE Transactions on Software Engineering*, SE 3(2):85–93, janvier 1977.

[SWM00] J. Soper, C. Walshaw et Cross M. : A combined evolutionary search and multilevel optimisation approach to graph partitioning. Rapport technique 00/IM/58, University of Greenwich, London, UK, avril 2000.

[TB91] E.-G. Talbi et P. Bessière : A parallel genetic algorithm for the graph partitioning problem. *In Proceedings of the 5th international conference on Supercomputing*, ICS '91, pages 312–320, New York, NY, USA, 1991. ACM.

[TZ93] L. Tao et Y. C. Zhao : Multi-way graph partition by stochastic probe. *International Journal of Computers & Operations Research*, 20(3):321–347, 1993.

[vDR94] R. van Driessche et D. Roose : A graph contraction algorithm for the calculation of eigenvectors of the laplacian matrix of a graph with a multilevel method. Rapport technique TW 209, Katholieke Universiteit Leuven, mai 1994.

[wal] Jostle : Graph partitioning software. `http://staffweb.cms.gre.ac.uk/~c.walshaw/jostle/`.

[Wal10] C. Walshaw : Variable partition inertia : graph repartitioning and load-balancing for adaptive meshes. *In* S. Chandra M. Parashar et X. Li, éditeurs : *Advanced Computational Infrastructures for Parallel and Distributed Adaptive Applications*. Wiley, New York, 2010.

[WC00] C. Walshaw et M. Cross : Parallel optimisation algorithms for multilevel mesh partitioning. *Parallel Computing*, 26(12):1635–1660, 2000.

[WC02] C. Walshaw et M. Cross : Dynamic Mesh Partitioning and Load-Balancing for Parallel Computational Mechanics Codes. *In* B. H. V. Topping, éditeur : *Computational Mechanics Using High Performance Computing*, pages 79–94. Saxe-Coburg Publications, Stirling, 2002.

[WCE97] C. Walshaw, M. Cross et M. G. Everett : Parallel Dynamic Graph Partitioning for Adaptive Unstructured Meshes. *J. Parallel Distrib. Comput.*, 47(2):102–108, 1997.

[WRSL05] Y. Wan, S. Roy, A. Saberi et B. Lesieutre : A stochastic automaton-based algorithm for flexible and distributed network partitioning. *In Swarm Intelligence Symposium, 2005. SIS 2005. Proceedings 2005 IEEE*, pages 273–280, juin 2005.