Équilibrage dynamique avec nombre variable de processeurs par une méthode de repartitionnement de graphe

Clément Vuchener, Aurélien Esnard

Équilibrage dynamique avec nombre variable de processeurs par une méthode de repartitionnement de graphe

Clément Vuchener, Aurélien Esnard

INRIA Bordeaux-Sud-Ouest et LaBRI UMR CNRS 5800,
220, Avenue de la Vieille Tour,
33405 Talence - France
{vuchener,esnard}@inria.fr

RÉSUMÉ. *L'équilibrage dynamique de charge est une étape cruciale qui conditionne la performance des codes adaptatifs dont l'évolution de la charge est difficilement prévisible. Néanmoins, l'ensemble des travaux dans ce domaine se limitent — à notre connaissance — au cas où le nombre de processeurs est fixé initialement et n'est pas remis en cause lors de l'équilibrage. Cela peut s'avérer particulièrement inefficace, notamment du point de vue de la consommation des ressources. Nous proposons dans cet article deux nouveaux algorithmes de repartitionnement de graphe permettant de faire varier le nombre de processeurs, en supposant que la charge du graphe est déjà équilibrée. Ces algorithmes optimisent conjointement la coupe et la migration des données en s'appuyant sur un modèle de partitionnement de graphe à sommets fixes. Des résultats expérimentaux valident nos travaux en les comparant à d'autres approches.*

ABSTRACT. *Dynamic load balancing is an important step affecting the performance of adaptive codes whose load evolution is difficult to predict. Nevertheless, as far as we know, other studies are limited to an initially fixed number of processors which is not modified during the balancing phase. It can be very inefficient, more particularly in terms of resource consumption. In this paper, we present two new graph repartitioning algorithms which permit a variable number of processors, assuming the load is already balanced. These algorithms optimize both edge cut and data migration using graph partitioning with fixed vertices. Experimental results validate our work comparing it with other approaches.*

MOTS-CLÉS : *calcul haute-performance, équilibrage dynamique de charge, partitionnement de graphe.*

KEYWORDS: *high-performance computing, dynamic load balancing, graph partitioning.*

DOI:10.3166/TSI.31.1-21 © 2012 Lavoisier

Technique et science informatiques – n° 1/2012, 1-21

1. Introduction

Dans le domaine du calcul scientifique, l'équilibrage de la charge est un problème crucial, qui conditionne la performance des programmes parallèles. On applique généralement un algorithme d'équilibrage statique, qui répartit la charge de calcul entre les différents processeurs, préalablement à l'exécution du programme parallèle. Pour certaines applications scientifiques, comme les codes AMR (*Adaptative Mesh Refinement*), il est difficile voire impossible de prédire l'évolution de la charge avant même l'exécution de la simulation. Lors de l'exécution, il est nécessaire de calculer régulièrement un nouvel équilibrage de la charge. On parle alors d'équilibrage dynamique.

Une approche très répandue pour résoudre le problème d'équilibrage (statique ou dynamique) s'appuie sur un modèle de graphe. Chaque sommet du graphe représente une tâche élémentaire de calcul (le plus souvent associée à une donnée du problème) et chaque arête représente une dépendance dans le calcul entre deux tâches. Pour équilibrer la charge entre M processeurs, on effectue alors un *partitionnement* du graphe en M parties, chacune étant affectée à un processeur. Au besoin, si la charge évolue, on peut procéder à un *repartitionnement*. Outre les critères classiques du partitionnement, on considère dans le problème du repartitionnement qu'il faut optimiser l'ensemble des critères suivants (Schloegel *et al.*, 2000) :

1. minimiser le temps de calcul (T_{comp}), ce qui revient à équilibrer le poids des parties (à un facteur de déséquilibre près) ;
2. minimiser le temps de communication (T_{comm}), ce qui revient à minimiser la coupe du graphe induite par la nouvelle partition ;
3. minimiser le temps de migration (T_{mig}) des données de l'ancienne partition vers la nouvelle ;
4. minimiser le temps de repartitionnement (T_{repart}).

Par ailleurs, il faut noter que le repartitionnement et la migration qui en découle ne s'effectue pas à chaque itération, mais périodiquement, disons toutes les α itérations. Il en résulte que le temps total d'une période du code s'écrit : $T_{total} = \alpha.(T_{comp} + T_{comm}) + T_{mig} + T_{repart}$. Si l'on suppose T_{repart} négligeable devant les autres termes, et si l'on considère que T_{comp} est implicitement minimisé en équilibrant les parties, il en résulte que pour minimiser T_{total}, il faut minimiser $\alpha.T_{comm} + T_{mig}$. Au final, cela met en évidence qu'il y a un compromis à faire entre l'optimisation du temps de communication (T_{comm}) et l'optimisation du temps de migration (T_{mig}). Ce compromis est contrôlé par le paramètre α, qui dépend de l'application visée.

Comme nous allons le voir à la section suivante, il existe de nombreux travaux autour de l'équilibrage dynamique et du repartitionnement (Hendrickson, Devine, 2000 ; Teresco *et al.*, 2006). Cependant, tous ces travaux se limitent — à notre connaissance — au cas où le nombre de processeurs est fixé initialement et n'est pas remis en cause lors de l'équilibrage. Cela peut s'avérer particulièrement inefficace, notam-

ment du point de vue de la consommation des ressources (Iqbal, Carey, 2005). Nous proposons dans cet article deux nouveaux algorithmes de repartitionnement de graphe permettant de faire varier le nombre de processeurs, en supposant que la partition initiale est déjà équilibrée. Nous appelons ce problème : le *repartitionnement* $M \times N$ de graphe. Nos algorithmes optimisent conjointement la coupe et la migration en s'appuyant sur un modèle de partitionnement de graphe à sommets fixes (Catalyurek *et al.*, 2009).

Même si le problème que nous étudions ici peut être vu comme une étape intermédiaire au cas d'étude plus général où le nombre de processeurs varient en même temps que la charge, nous pensons néanmoins que nos travaux ont déjà un intérêt dans plusieurs cas. Tout d'abord, il faut remarquer qu'il est souvent difficile de prévoir le nombre de ressources optimal pour une application. Il peut donc être utile de l'adapter dynamiquement au cours des calculs. De plus, nos algorithmes peuvent être utilisés lorsque la charge évolue uniformément. Dans ce cas, il peut s'avérer utile d'ajuster le nombre de processeurs utilisés de manière plus fine qu'en doublant trivialement le nombre de processeurs ($N = 2.M$). Par ailleurs, beaucoup d'applications adaptatives disposent d'une stratégie d'équilibrage *ad hoc*, le plus souvent incrémentale. Dans ce cas, nos algorithmes peuvent s'appliquer périodiquement pour ajuster le nombre de processeurs en supposant que la charge a préalablement été équilibrée.

Dans la section suivante, nous allons examiner les travaux existants autour de l'équilibrage dynamique. Puis, nous présentons à la section 3 un premier algorithme de repartitionnement $M \times N$ de graphe, s'appuyant sur un résultat théorique de repartitionnement d'une chaîne (Sec. 3.1). Nous continuons à la section 4 en proposant une généralisation de notre algorithme de repartitionnement plus favorable encore à la migration. Ces deux algorithmes s'accompagnent de résultats expérimentaux détaillés à la section 5. Nous terminons en présentant les conclusions et perspectives de nos travaux.

2. Travaux existants

Il existe de nombreux travaux de recherche dans le domaine de l'équilibrage dynamique (Hendrickson, Devine, 2000 ; Teresco *et al.*, 2006). Nous présentons brièvement les trois méthodes de repartitionnement les plus populaires : le *Scratch-Remap*, les méthodes diffusives, le repartitionnement à base de sommets fixes.

L'approche la plus simple est certainement le *Scratch-Remap* (Oliker, Biswas, 1998), qui consiste à calculer une nouvelle partition *from scratch*, c'est-à-dire sans prendre en compte l'ancienne. Cette technique minimise naturellement la coupe, mais ne contrôle pas du tout la migration. Pour réduire ce dernier coût, on effectue alors une étape supplémentaire de *remapping*, en cherchant à renuméroter les nouvelles parties pour maximiser les données restant en place. Les méthodes diffusives, comme celle implantée dans le logiciel ParMetis (Schloegel *et al.*, 1997), s'inspirent du problème

physique de la diffusion de la chaleur pour rééquilibrer la charge. C'est une méthode itérative où à chaque itération, deux processeurs échangent une quantité de données proportionnelle à leur différence de charge. Ainsi, après plusieurs itérations, la charge converge vers un nouvel équilibre, définissant ainsi une nouvelle partition. Une autre approche proposée consiste à repartitionner en minimisant conjointement la coupe et le volume de données à déplacer (Aykanat *et al.*, 2007 ; Catalyurek *et al.*, 2009). Pour chaque partie, on ajoute un sommet fixe de poids nul. Ces sommets particuliers sont reliés par de nouvelles arêtes — dites de migration — à tous les sommets dans la partie à laquelle ils sont associés. Pour repartitionner, on effectue alors un partitionnement du graphe enrichi, en conservant fixés les nouveaux sommets dans leurs parties respectives. Ainsi, si un sommet normal change de partie, cela implique de couper une arête de migration et donc de compter un coût supplémentaire de migration associé à cette arête. Un partitionneur cherchant à minimiser la coupe minimisera également le volume de migration. Cette approche a notamment été mise en œuvre dans le logiciel Zoltan en utilisant par ailleurs un modèle à base d'hypergraphe plutôt que de graphe (Catalyurek *et al.*, 2009). *Scotch* a récemment ajouté une méthode similaire de repartitionnement de graphe basée sur des sommets fixes, combinée à une méthode de raffinement local diffusive (Fourestier, Pellegrini, 2011).

Il existe encore de nombreux travaux sur l'équilibrage dynamique incluant des méthodes géométriques de type *Recursive Coordinate Bisection* (RCB) (Hendrickson, Devine, 2000) ou à base de *Space-Filling Curve* (SFC) (Pilkington, Baden, 1996), les méthodes spectrales récursives (Van Driessche, Roose, 1995), ou encore d'autres approches plus exotiques comme le partitionnement basé sur un modèle de *skewed graph* (Hendrickson *et al.*, 1997).

Tous ces travaux sont très intéressants, mais sont limités au cas où le nombre de processeurs est fixé initialement et n'est pas remis en cause lors de l'équilibrage. À notre connaissance, il n'y a pas de travaux qui étudient la problématique du partitionnement de graphe/hypergraphe en faisant varier le nombre de processeurs. Néanmoins, quelques travaux récents ont montré l'intérêt d'un tel équilibrage, en ajustant dynamiquement le nombre de processeurs dans un code adaptatif (AMR) afin d'économiser les ressources utilisées (Iqbal, Carey, 2005).

3. Repartitionnement $M \times N$ d'un graphe basé sur une chaîne

Avant d'étudier le problème de repartitionnement d'un graphe G de M vers N processeurs, nous allons nous intéresser au cas particulier d'un graphe de type chaîne, représentant simplement un tableau de données à une dimension. Rappelons que nous nous intéressons uniquement au problème de variation du nombre de processeurs, en supposant que le poids du graphe n'est pas modifié. Par ailleurs, nous supposons *dans un premier temps* que les ensembles des processeurs émetteurs et récepteurs sont disjoints, c'est-à-dire que les données ne restent pas en place lors de la migration. Nous

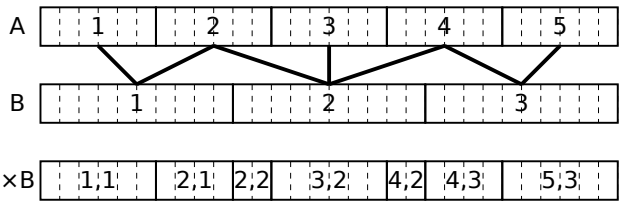


Figure 1. Repartitionnement de $M = 5$ vers $N = 3$ d'une chaîne de longueur $L = 30$ éléments

reviendrons sur ce point à la section 3.4. Des résultats expérimentaux comparatifs avec d'autres algorithmes de repartitionnement sont présentés à la section 5.

3.1. Repartitionnement $M \times N$ d'une chaîne

Considérons une chaîne de longueur L. Une partition en M de cette chaîne peut être obtenue simplement en la découpant en M intervalles égaux et contigus. On peut alors repartitionner cette chaîne en N parties en utilisant un même découpage. Il en résulte les deux découpages A et B représentés sur la figure 1. Notons que pour simplifier notre propos, nous allons supposer que L est multiple de M et N.

On peut dénombrer les communications en réalisant simplement l'intersection des motifs A et B, ce que nous notons $A \times B$ sur la figure 1. Dans le cas où M et N sont premiers entre eux, il est facile de voir en examinant le découpage $A \times B$ que le nombre de messages est exactement de $M + N - 1$. Lorsque M et N ont un multiple commun, il est alors possible de se ramener au cas précédent en posant $M' = M/PGCD(M, N)$ et $N' = N/PGCD(M, N)$. En effet, il faut remarquer que le même motif de communication se répète $PGCD(M, N)$ fois, avec $M' + N' - 1$ messages à chaque fois. Au final, on établit que le nombre de communications est $M + N - PGCD(M, N)$ dans le cas général.

Cette méthode de repartitionnement produit un schéma de communication qui possède de bonnes propriétés. En effet, comme nous allons le prouver dans la section suivante le nombre de communications est minimal. Par conséquent, cela a tendance à réduire le nombre de latences et à maximiser le débit atteint pour les communications point-à-point. Par ailleurs, il faut constater que ce schéma de communication est bien équilibré, puisque chaque émetteur a un nombre de récepteurs qui diffère au plus de 1 (et réciproquement). Plus précisément, chaque ancienne partie envoie ses données vers $\lceil \frac{N}{M} \rceil$ ou $\lceil \frac{N}{M} \rceil + 1$ nouvelles parties et chaque nouvelle partie reçoit depuis $\lceil \frac{M}{N} \rceil$ ou $\lceil \frac{M}{N} \rceil + 1$ parties. Ainsi, chaque processeur communiquera avec le moins possible d'autres processeurs. Les communications pourront donc s'effectuer en parallèle avec le moins de contention possible.

3.1.1. Preuve d'optimalité du schéma de communication

Soit $G = (A, B, E)$ le graphe biparti des communications. Les données sont repartitionnées de $M = |A|$ processeurs vers $N = |B|$ processeurs. On suppose, sans perte de généralité, que l'on échange un volume total de données de $M \times N$ unités. Comme les partitions initiale et finale sont parfaitement équilibrées, un processeur de A envoie exactement N unités de données et un processeur de B reçoit exactement M unités.

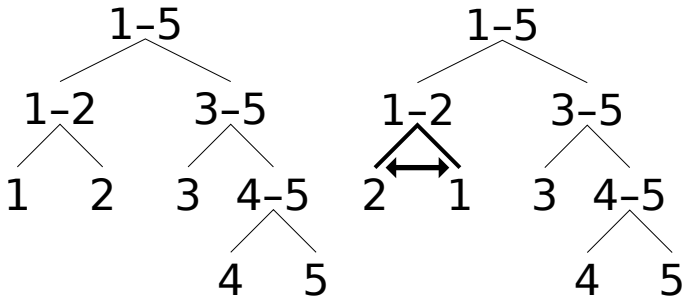
Soit K le nombre de composantes connexes de G, notées $G_i = (A_i, B_i, E_i)$ avec $i \in [1, K]$. Si l'on effectue le bilan des communications au sein de la composante G_i, il apparaît que $M_i = |A_i|$ processeurs envoient un volume $M_i \times N$ vers $N_i = |B_i|$ processeurs qui reçoivent le volume $N_i \times M$. Il en résulte que le volume de communication d'une composante est $V_i = M_i \times N = N_i \times M$, avec M_i et N_i non nuls. Comme V_i est un multiple commun de M et N, on peut affirmer que $\forall i, V_i \geq PPCM(M, N)$. Par conséquent, le volume total de communication $M \times N = \sum_{i \in [1,K]} V_i$ est supérieur à $K \times PPCM(M, N)$. En utilisant la propriété $PGCD(M, N) \times PPCM(M, N) = M \times N$, on en déduit simplement que $K \leq PGCD(M, N)$. Comme G_i est connexe [1], son nombre d'arêtes $|E_i|$ est au moins $M_i + N_i - 1$. Le nombre total d'arêtes $|E| = \sum_{i \in [1,k]} |E_i|$ est donc supérieur à $\sum_{i \in [1,K]} M_i + \sum_{i \in [1,K]} N_i - K = M + N - K$. Il en résulte que le nombre total de communications $|E|$ est supérieur ou égal à $M + N - PGCD(M, N)$, car $K \leq PGCD(M, N)$. Comme nous l'avons vu à la section précédente, cette borne inférieure est atteinte.

3.2. Recherche de chemin

Le repartitionnement $M \times N$ de la chaîne peut s'appliquer aux graphes si l'on dispose d'un ordre sur les sommets. Cela revient à rechercher un chemin dans le graphe. Cette approche s'avère en pratique trop contraignante et nous avons choisi d'utiliser un chemin sur les parties du graphe afin de construire notre algorithme de repartitionnement.

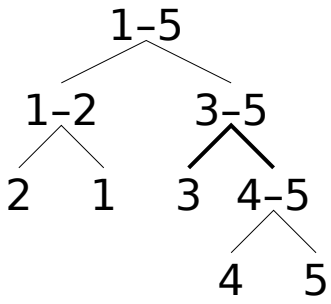
Étant donné un partitionnement initial du graphe en M, nous cherchons un *bon* chemin dans le graphe des parties (ou *graphe quotient*). Un bon chemin est connexe pour éviter la coupure des nouvelles parties et deux parties consécutives doivent être très connectées pour permettre de bien placer une partie entre celles-ci. Rechercher un tel chemin est malheureusement un problème NP-difficile (similaire au problème du voyageur de commerce (Garey, Johnson, 1979)). Pour résoudre ce problème, nous proposons une heuristique qui s'appuie sur l'arbre de bissections récursives (issu du partitionnement initial). Notre algorithme effectue donc un parcours en largeur de l'arbre pour choisir l'ordre des fils de chaque nœud, un nœud de l'arbre représentant

1. Pour tout graphe connexe $G = (V, E)$, $|E| \geq |V| - 1$.

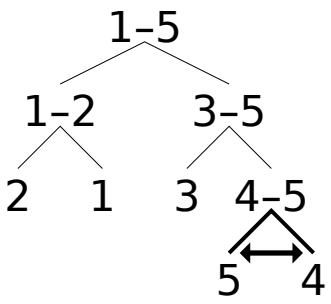


(a) L'arbre de bissections initial

(b) Choix de l'ordre des parties 1 et 2



(c) Choix de l'ordre des parties 3 et 4 - 5



(d) Choix de l'ordre des parties 4 et 5

Figure 2. Exécution de notre algorithme de recherche d'un chemin sur l'exemple de la figure 3a. L'ordre final est 2, 1, 3, 5, 4

un ensemble de parties ou une seule dans le cas d'une feuille. L'ordre est choisi en fonction de la connexion de chaque fils avec les parties précédentes et suivantes. Plus précisément, on examine la connexion des fils avec la partie précédente au même niveau dans l'arbre et la partie suivante au niveau du père (car l'ordre de ses fils n'est pas encore connu). Si nécessaire, on inverse l'ordre des deux fils, et on passe au nœud suivant dans le parcours de l'arbre.

Un exemple d'exécution de cet algorithme est présenté sur la figure 2. Le graphe est partitionné en $M = 5$ comme sur la figure 3a. La figure 2a présente l'arbre de bissection initial[2]. On débute le parcours en largeur. La racine, contenant toutes les parties, n'a aucun précédent ou suivant, on ne peut pas choisir d'ordre. On fait ensuite le choix au niveau du fils gauche sur la figure 2b : il n'y a aucun partie avant, les connexions avec 3 - 5 sont comparées car on ne sait pas encore si la partie suivante

2. Notons que cet arbre binaire n'est pas parfaitement équilibré dans le cas général. Ainsi, pour $M = 5$, la première bissection contient $2/5$ de la charge à gauche et $3/5$ à droite, les parties les plus grosses étant placées à droite.

sera 3, 4 ou 5. La partie 1 est plus connecté avec 3 - 5 donc l'ordre choisi est 2 puis 1. À l'étape suivante (Fig. 2c), on connaît la partie précédente car elle vient d'être choisie mais il n'y a pas de partie suivante. Il faut donc comparer les connexions entre 1 et 3 et entre 1 et 4 - 5. Enfin à la dernière étape (Fig. 2d), on compare les connexions entre 3 et 4 et entre 3 et 5. Au final, on obtient le chemin représenté sur la figure 3b.

3.3. Algorithme de repartitionnement $M \times N$ avec des sommets fixes

En s'inspirant des techniques de repartitionnement à sommets fixes (Aykanat *et al.*, 2007 ; Catalyurek *et al.*, 2009), on propose un nouvel algorithme de repartitionnement $M \times N$ s'appuyant sur le chemin obtenu dans la section 3.2. Notre algorithme se décompose en plusieurs étapes.

1. Le graphe est initialement partitionné en M parties (Fig. 3a).

2. Les anciennes parties sont renumérotées en respectant le chemin trouvé avec l'algorithme de la section précédente (Fig. 3b).

3. N sommets de poids nul sont ajoutés, chacun étant fixé dans une partie différente. On note w_i un tel sommet associé à la partie i ($i \in [1, N]$).

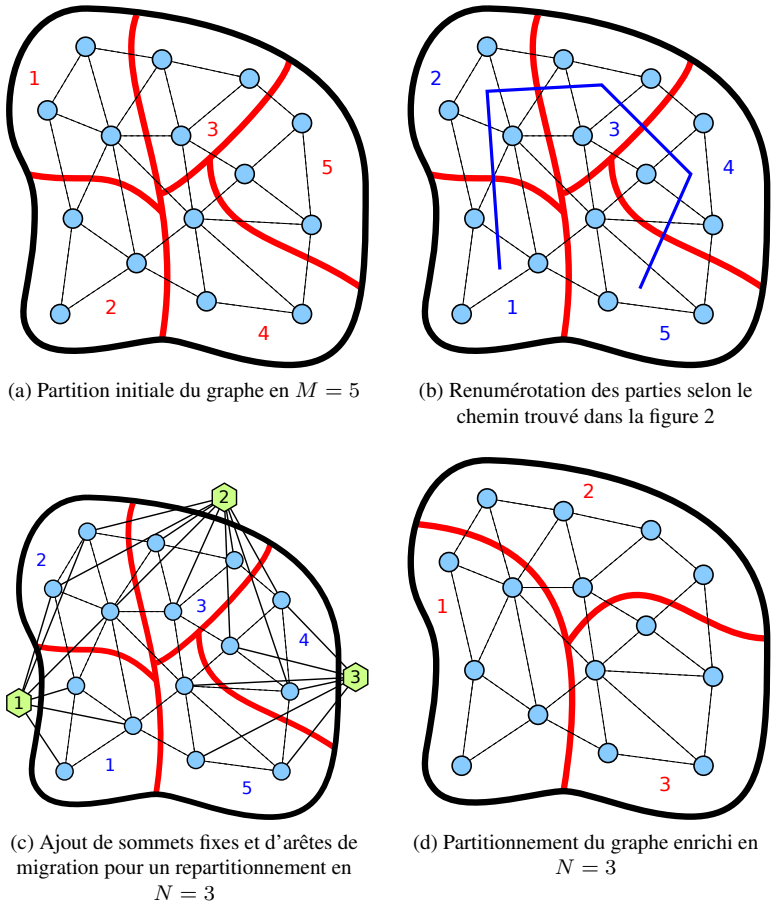
4. On note V_k la partie k du graphe initial ($k \in [1, M]$). Des arêtes sont ajoutées entre le sommet fixe w_i et tous les sommets de V_k si il existe une communication entre les parties k et i au sens du schéma de communication optimal étudié en 3.1. Notons qu'un sommet fixe peut être connecté à plusieurs parties V_k, comme l'illustre la figure 3c.

5. Le graphe ainsi obtenu est partitionné en N en utilisant un partitionneur acceptant des sommets fixes (Fig. 3d).

Nous appelons *arête de migration* les nouvelles arêtes ajoutées, connectées aux sommets fixes. En cherchant à minimiser la coupe des arêtes de migration, le partitionneur préférera placer la nouvelle partie i à l'emplacement des anciennes parties k reliées au sommet fixe w_i. Ainsi, notre algorithme guide le partitionneur pour qu'il respecte le schéma de communication précédent, dont nous avons établi les bonnes propriétés (Sec. 3.1).

3.4. Optimisation de la migration : renumérotation des nouvelles parties

Nous avons supposé jusqu'à présent que les ensembles de processeurs émetteurs et récepteurs étaient disjoints. Dans le cas contraire, il est nécessaire d'optimiser le schéma de communication produit par notre algorithme afin de minimiser la migration en laissant un maximum de données sur place. Pour ce faire, nous utilisons une procédure de *remapping* similaire à celle utilisée dans le *Scratch-Remap* (*cf.* Sec. 2). Afin d'obtenir une renumérotation adéquate des nouvelles parties, nous utilisons l'algorithme glouton proposé dans PLUM (Oliker, Biswas, 1998). On constate empiriquement que cet algorithme appliqué à la matrice de communication produite par notre algorithme de repartitionnement donne de très bons résultats.



(a) Partition initiale du graphe en $M = 5$

(b) Renumérotation des parties selon le chemin trouvé dans la figure 2

(c) Ajout de sommets fixes et d'arêtes de migration pour un repartitionnement en $N = 3$

(d) Partitionnement du graphe enrichi en $N = 3$

Figure 3. Exemple de repartitionnement d'un graphe de $M = 5$ vers $N = 3$

4. Repartitionnement $M \times N$ d'un graphe : généralisation

Afin d'améliorer l'algorithme de repartitionnement $M \times N$ que nous venons de présenter, nous allons généraliser notre approche en se basant sur une structure plus complexe qu'un simple chemin dans le graphe quotient associé au partitionnement initial. Nous appelons cette nouvelle structure, *l'hypergraphe de repartitionnement* (Sec. 4.1) et nous verrons comment il est possible de s'en servir pour construire une généralisation de notre algorithme de repartitionnement aux sections 4.2 et 4.3. Des résultats expérimentaux comparatifs avec d'autres algorithmes de repartitionnement sont présentés à la section 5.

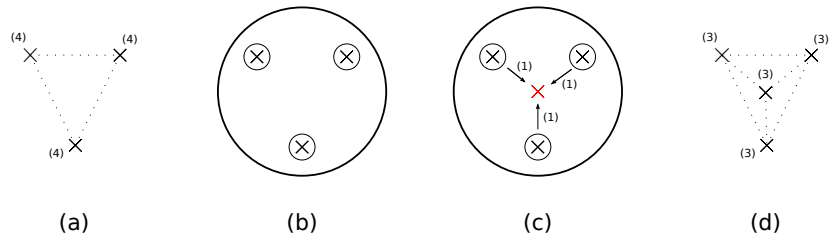


Figure 4. Exemple d'hypergraphe de repartitionnement dans le cas 3×4 *: injection d'une nouvelle partie dans le graphe quotient*

4.1. Hypergraphe de repartitionnement

Étant donné le partitionnement d'un graphe G en M parties, on veut effectuer un repartitionnement de G en N parties en optimisant les critères classiques de coupe et de migration que nous avons déjà énoncés. Afin de modéliser précisément une stratégie de repartitionnement, nous allons introduire la notion d'hypergraphe de repartitionnement.

Soit $C = (C_{i,j})$, la matrice de communication de dimension $M \times N$ associée au repartitionnement de G. L'élément $C_{i,j}$ donne la quantité de données envoyées par le processeur de rang i au processeur de rang j. Le nombre de communications total est donc le nombre de termes non nuls dans C. La matrice C peut être interprétée comme un hypergraphe H, dit *hypergraphe de repartitionnement*. Cet hypergraphe se compose de M sommets représentant les parties initiales et de N hyperarêtes représentant les parties obtenues après le repartitionnement. Ainsi, un sommet i de H appartient à une hyperarête j si des données sont échangées entre l'ancienne partie i et la nouvelle partie j lors de la migration.

Considérons l'exemple de la figure 4 qui illustre le cas d'un repartitionnement 3×4, ce qui revient en d'autres termes à « injecter » une nouvelle partie. Étant donné le graphe quotient dessiné en (a) possédant $M = 3$ parties (chacune de poids 4), il est possible d'injecter une $4^{\text{ème}}$ partie en respectant l'hypergraphe de repartitionnement schématisé en (b). Cet hypergraphe composé de $N = 4$ hyperarêtes modélise précisément comment réaliser un repartitionnement en contrôlant le schéma de communication associé. Les hyperarêtes de taille 1 imposent de conserver les anciennes parties. La grande hyperarête de taille 3 impose que l'on ajoute une nouvelle partie à partir des 3 anciennes, ce qui implique la migration d'une portion des données (de poids 1), comme le suggère la figure (c). Au final, on obtient un nouveau graphe quotient en $N = 4$ parties (chacune de poids 3) représenté en (d). On donne ci-dessous la matrice de communication associée à l'exemple de la figure 4.

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

Il faut noter que l'hypergraphe de repartitionnement permet de modéliser le « schéma de communication » sans détailler — contrairement à la matrice de communication — les volumes de données à transférer. Comme nous le verrons à la section 4.2, cette représentation en hypergraphe est plus facile à manipuler dans l'optique de la mettre en correspondance avec le graphe quotient.

4.1.1. Classes de matrices de communication optimales

Il existe plusieurs classes de matrices de communication optimales en nombre de communications. Nous avons déjà présenté à la section 3.1 un schéma de communication optimal basé sur la chaîne. La figure 5a donne une représentation de cette matrice « en escalier » dans le cas 7×10, ainsi qu'une représentation de l'hypergraphe de repartitionnement associé. Cette classe de matrice de communication n'est pas très favorable à la migration, dans le sens où elle ne minimise pas naturellement le volume de données à faire migrer. C'est pourquoi nous avons introduit à la section 3.4 une étape supplémentaire de *remapping* pour corriger ce problème. Nous allons voir maintenant comment il est possible de construire d'autres matrices de communication optimales à la fois en nombre de communications et en volume de migration.

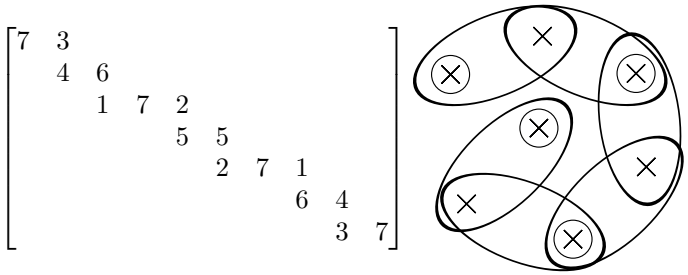
Considérons le cas $M < N$, où le nombre de processeurs augmente. On peut décomposer C en deux blocs (D, R) : un bloc D de dimension $M \times M$ correspondant aux données restant en place et un bloc R de dimension $M \times N - M$ correspondant à la migration des données provenant des M anciens processeurs vers les $N - M$ nouveaux. Optimiser le volume de migration signifie que l'on prend soin de laisser un maximum de données sur place, c'est-à-dire que $\sum_{0 \leq k < M} C_{k,k}$ est maximal. Ainsi, il apparaît évident qu'une matrice de communication C optimisera la migration si elle se compose d'un bloc diagonal $D = M.I_M$ avec I_M la matrice identité d'ordre M. Dans ce cas précis, il est facile de montrer que le nombre de communications pour C est optimal si R est lui-même optimal [3].

On peut alors envisager plusieurs possibilités pour construire le bloc R comme par exemple sur les figures 5b et 5c. Dans la suite de l'article, nous nous intéressons plus particulièrement au cas où ce second bloc est la matrice « en escalier » de dimension $M \times N - M$ correspondant au repartitionnement d'une chaîne, dont nous avons déjà prouvé l'optimalité à la section 3.1 (Fig. 5b).

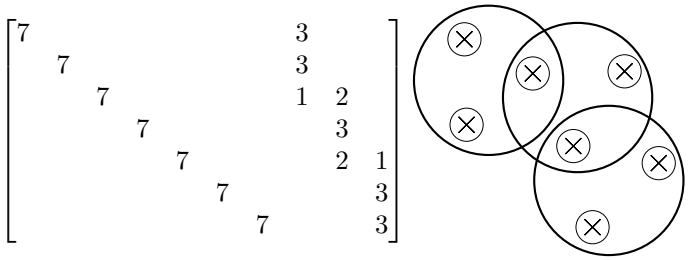
Dans le cas où le nombre de processeurs diminue ($M > N$), la matrice de communication se construit en transposant la matrice de communication optimale obtenue précédemment. L'hypergraphe de repartitionnement associé est alors l'hypergraphe dual.

---

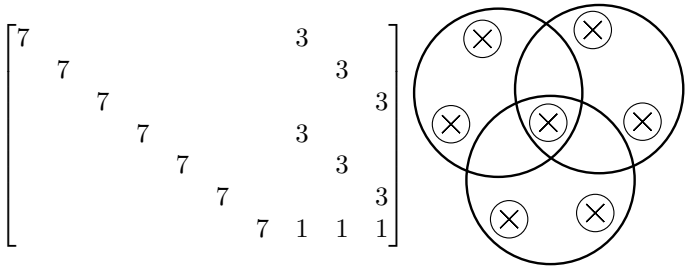
3. Preuve — Soit $Z(M, N) = M + N - PGCD(M, N)$ le nombre de communications minimal pour un problème de dimension $M \times N$. Le nombre de communications associé à la matrice $C = (D, R)$ est $M + Z(M, N - M) = M + N - PGCD(M, N - M)$, car R de dimension $M \times N - M$ est optimal. Or $PGCD(M, N) = PGCD(M, N - M)$. Par conséquent, la matrice de communication C est également optimale. □



(a) Repartitionnement basé sur une chaîne



(b) Repartitionnement avec un bloc diagonal et le reste basé sur une chaîne



(c) Autre exemple de repartitionnement utilisant un bloc diagonal

Figure 5. Matrices de communication optimales et représentation des hypergraphes de repartionnement associés dans le cas 7×10

4.2. Matching entre l'hypergraphe de repartitionnement et le graphe quotient

Afin de réaliser un « bon » repartitionnement, il faut trouver où placer les nouvelles parties par rapport à l'ancienne partition. Cela nécessite de mettre en correspondance les sommets de l'hypergraphe de repartitionnement et ceux du graphe quotient associé à la partition (problème de *matching*). Les sommets de l'hypergraphe se trouvant dans une même hyperarête doivent correspondre à des parties « proches » dans le graphe quotient. En effet, ces parties devront transmettre des données à une même nouvelle partie.

Afin de caractériser de telles parties « proches » correspondant aux hyperarêtes, on calcule un score. Pour mesurer la proximité des parties, on utilise les arêtes du graphe quotient. Pour formuler ce score, l'hypergraphe de repartitionnement et le graphe quotient sont notés sous forme matricielle. La matrice H de l'hypergraphe est de taille $M \times N$ et ses élément $H_{v,e}$ sont non nuls si et seulement si l'hyperarête e contient le sommet v. Le graphe quotient est représenté par sa matrice d'adjacence Q dont l'élément $Q_{i,j}$ est le poids de l'arête (i, j).

Un *matching* est représenté par une matrice de permutation X de taille $M \times M$ dont l'élément $X_{i,j}$ vaut 1 si le sommet i de H est associé au sommet j de Q ; les autres éléments sont nuls.

Dans l'équation 1, $X_{i,i'}$, $X_{j,j'}$, $H_{i,k}$ et $H_{j,k}$ sont des valeurs binaires, leur produit est non nul quand les sommets i' et j' de Q sont respectivement associés aux sommets i et j de H, qui se trouvent dans la même hyperarête k. Le score vaut alors la somme, pour chaque hyperarête, du poids des arêtes $Q_{i',j'}$ dont les extrémités sont associées à des sommets qui appartiennent à cette hyperarête. Il est ainsi favorisé d'associer à chaque hyperarête un sous-graphe complet ou quasi-complet dans Q.

$$score(X) = \sum_{i,j,i',j',k} X_{i,i'} X_{j,j'} H_{i,k} H_{j,k} Q_{i',j'} \tag{1}$$

On peut réorganiser l'équation du score de la façon suivante.

$$score(X) = \sum_{i,i'} X_{i,i'} \sum_{j,j'} X_{j,j'} (\sum_{k} H_{i,k} H_{j,k}) Q_{i',j'} \tag{2}$$

En notant x le vecteur de taille M^2 tel que $x_k = X_{i,i'}$ avec $k = iM + i'$ et $\otimes$ le produit de Kronecker [4], on peut reformuler le score sous la forme :

$$score(x) = x^T A x \quad \text{avec } A = HH^T \otimes Q \text{ de taille } M^2 \times M^2 \tag{3}$$

Formulé ainsi, il apparaît que notre problème d'optimisation est un problème classique d'optimisation quadratique à variables binaires (x_k vaut 0 ou 1), sous les contraintes suivantes :

$$\begin{cases} \forall i, & \sum_{i'} x_{iM+i'} = 1 & \text{(contrainte sur les lignes de } X) \\ \forall i', & \sum_{i} x_{iM+i'} = 1 & \text{(contrainte sur les colonnes de } X) \end{cases} \tag{4}$$

Ce problème d'optimisation est NP-difficile (Garey, Johnson, 1979). Notons que notre fonction score est convexe, car la matrice A est symétrique définie-positive. Il

4. Soient A une matrice de taille $P \times Q$ et B une matrice de taille $R \times S$. Par définition, leur produit de Kronecker est la matrice $A \otimes B$ de taille $PR \times QS$, définie par blocs successifs de taille $R \times S$, le bloc d'indice (i, j) valant $A_{i,j} \cdot B$.

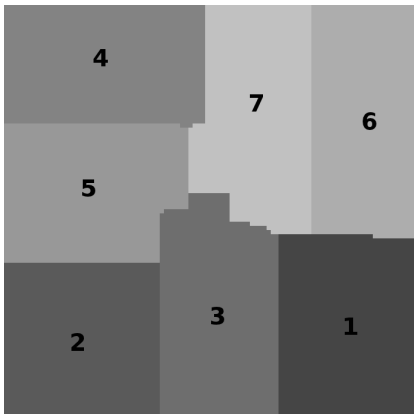


Figure 6. Exemple de partition initiale en $M = 7$ d'une grille 100×100

existe dans la littérature de nombreuses heuristiques pour tenter de résoudre efficacement ce problème : heuristiques probabilistes classiques (recuit-simulé, algorithme génétique), méthodes spectrales pour du matching graphe-graphe (Leordeanu, Hebert, 2005 ; Duchenne *et al.*, 2009), recherche combinatoire de type *branch & bound*, *etc.* Dans cet article, nous utilisons une simple méthode de recuit simulé qui nous donne des résultats satisfaisants.

4.3. Algorithme de repartitionnement $M \times N$ généralisé

Nous allons maintenant présenter notre algorithme de repartitionnement généralisé qui reprend largement les principes de l'algorithme de repartitionnement basé sur la chaîne décrit à la section 3.3. Pour illustrer notre propos, nous considérons l'exemple d'une grille 100×100 partitionnée en $M = 7$ (Fig. 6) et repartitionnée en $N = 10$. Voici les étapes principales de notre algorithme.

1. Étant donné la partition initiale du graphe G en M (Fig. 6), on construit le graphe quotient Q (Fig. 7a).

2. On se donne alors un hypergraphe de repartitionnement H que l'on choisit « optimal » en nombre de communications et en volume de migration, comme par exemple l'hypergraphe de la figure 7b. Comme nous l'avons vu à la section 4.1.1, plusieurs choix sont ici possibles pour H. Ici, nous avons choisi H construit à partir d'une matrice composée d'un bloc diagonal et d'un bloc en "escalier" correspondant à la figure 5b.

3. Afin de faire correspondre les sommets de H et de G dont la numérotation ne correspond pas, nous appliquons un algorithme de *matching*, basé sur une heuristique de type recuit-simulé (Sec. 4.2). Dans cet exemple, la permutation trouvée $(3, 2, 5, 4, 7, 6, 1)$ correspond à la solution optimale, celle qui maximise la fonction *score* (Fig 7c).

4. Grâce à ce *matching*, on peut enrichir le graphe G de sommets fixes. On ajoute alors un sommet fixe dans G par hyperarête de H et celui-ci est relié à tous les sommets des parties qui se trouvent dans cette hyperarête (Fig. 7d).

5. Comme dans la section 3.3, il ne reste plus qu'à partitionner en N le graphe G enrichi de sommets fixes pour obtenir la partition finale en N (Fig. 8b).

Dans cet algorithme, si on remplace H par l'hypergraphe de la figure 5a, on retrouve le repartitionnement basé sur la chaîne. En effet, le matching avec cet hypergraphe « en forme de chaîne » revient à rechercher un chemin dans le graphe quotient.

4.4. Étude de cas

Pour mieux illustrer le comportement de notre algorithme généralisé, nous allons étudier le cas du repartitionnement 7×10 de la grille 100×100 déjà présentée en exemple. Le graphe associé à la grille se compose de $10\,000$ sommets et de $19\,000$ arêtes. Nous avons utilisé pour effectuer notre repartitionnement l'hypergraphe « optimal » correspondant à la figure 7b.

La figure 8a montre la partition initiale de la grille en $M = 7$ obtenue à partir de Scotch. L'équilibrage est quasi-parfait et la coupe vaut 361. En appliquant notre algorithme de repartitionnement généralisé, on obtient la partition finale en 10 représentée sur la figure 8b. Nous avons utilisé un facteur de repartitionnement [5] de $1/100$ pour favoriser une migration optimale par rapport à la coupe. Cette nouvelle partition présente un déséquilibre de 1% et une coupe de 495. A titre comparatif, un partitionnement classique de la grille en 10 donne une coupe de 468 avec Scotch, ce qui permet d'évaluer un surcoût pour notre méthode de l'ordre de 6%.

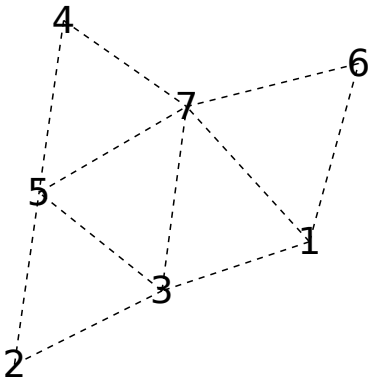
Si l'on s'intéresse maintenant au schéma de communication résultant, il faut remarquer qu'il respecte exactement le modèle imposé par l'hypergraphe de repartitionnement choisi et détaillé par la figure 7. En effet, on injecte bien les 3 nouvelles parties 8, 9 et 10 (couleurs vertes) au « centre » des anciennes parties $(2, 3, 5)$, $(4, 5, 7)$ et $(1, 6, 7)$.

Pour conclure, il faut remarquer que le nombre de communications obtenu est minimal (16 au total) et que le volume de migration obtenu est quasi-optimal (3003 contre 3000). La stratégie de repartitionnement précédente (présentée à la section 3) donne dans ce cas un volume de migration de 3571 au mieux, ce qui représente un gain de 19% pour la nouvelle approche.

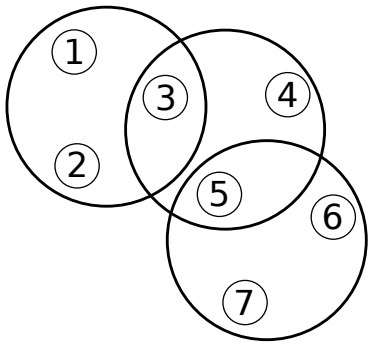
5. Résultats expérimentaux

Notre méthode de repartitionnement $M \times N$ a été comparée avec une méthode

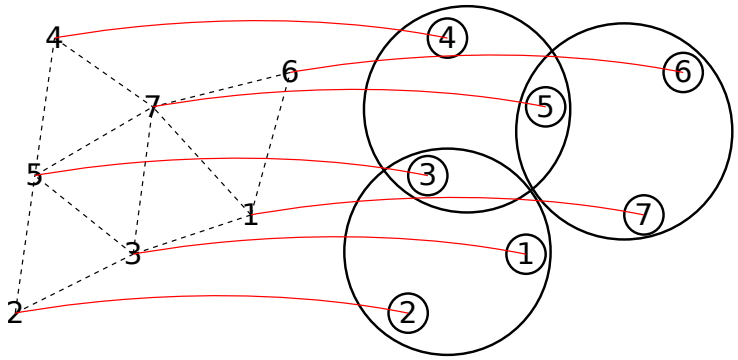
5. Le facteur de repartitionnement est le coût relatif des arêtes normales par rapport aux arêtes de migration. Cela permet d'ajuster l'importance de la coupe par rapport à la migration.



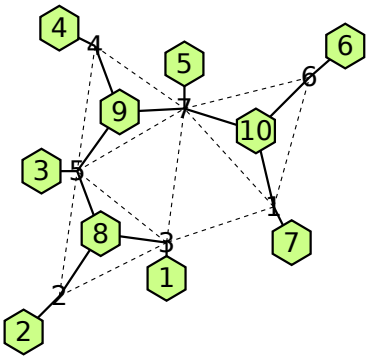
(a) Graphe quotient de la partition initiale de la Fig. 6



(b) Hypergraphe de repartitionnement pour le cas 7×10

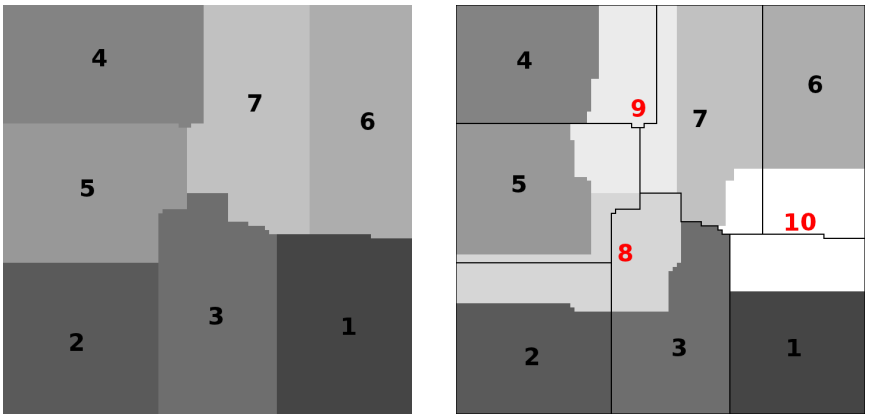


(c) Matching de l'hypergraphe de repartitionnement avec le graphe quotient



(d) Ajout des sommets fixes

Figure 7. Cas 7×10 *: utilisation du matching entre le graphe quotient et un hypergraphe de repartitionnement pour ajouter les sommets fixes*



(a) Partition initiale en 7 (b) Partition finale en 10

Figure 8. Repartitionnement en 7×10 *d'une grille 2D. Chaque couleur représente une partie. Les contours de la partition initiale en 7 sont affichés en noir sur la partition finale. Les 3 nouvelles parties ajoutées sont représentées par les couleurs les plus claires*

Scratch-Remap, une méthode diffusive implantée dans ParMetis 4.0 et la méthode de repartitionnement de Scotch 6.0. Même si ces méthodes (présentées à la section 2) ne sont pas directement conçues pour utiliser un nombre variable de processeurs, il est tout de même possible de les appliquer dans ce contexte [6]. Pour nos expériences, nous utilisons un graphe correspondant à une grille 3D de $32 \times 32 \times 32$ éléments (32 768 sommets et 95 232 arêtes). Ce graphe est initialement partitionné en $M = 8$ parties avec Scotch. Cette partition initiale est ensuite repartitionnée en $N \in [2, 24]$ de différentes manières en utilisant :

– la méthode *Scratch-Remap*, réalisée à l'aide de Scotch (sans l'option repartitionnement) ;

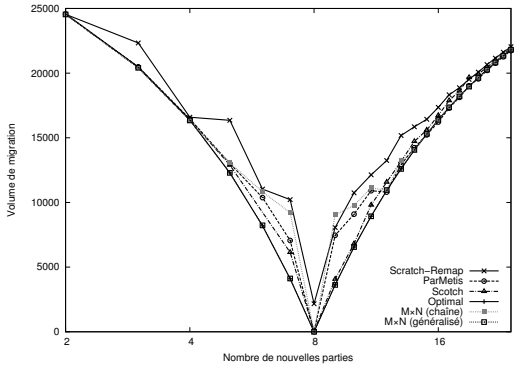
– l'algorithme de repartitionnement de ParMetis avec un facteur de repartitionnement de $1000/1$ qui est la valeur recommandée par la documentation ;

– l'algorithme de repartitionnement de Scotch avec un facteur de repartitionnement de $1/1$;

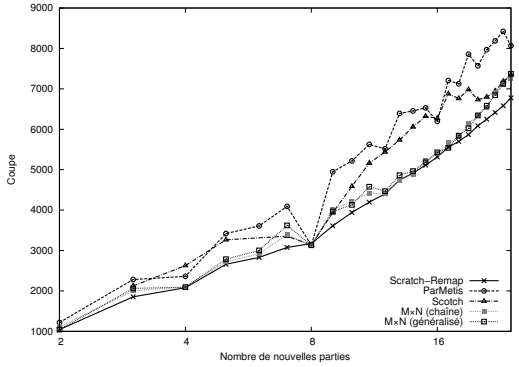
– les algorithmes $M \times N$ (chaîne) et $M \times N$ (généralisé) présentés dans cet article avec un facteur de repartitionnement de $1/100$ pour forcer le partitionneur à respecter le schéma de communication souhaité. Pour appliquer ces méthodes, nous utilisons le partitionneur Scotch [7].

6. Dans le cas $M < N$, cela revient à considérer $N - M$ parties initialement vides.

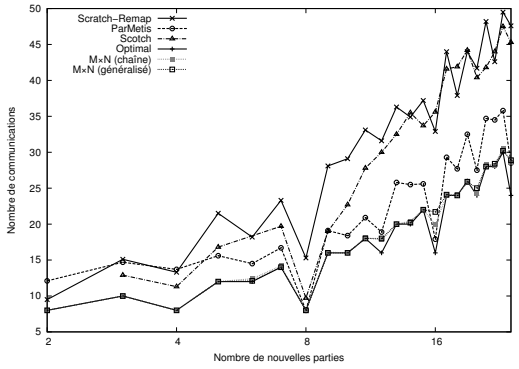
7. Les partitionneurs actuels ont malheureusement des difficultés pour partitionner ce type de graphe enrichis de nombreuses arêtes connectées à des sommets fixes. Le problème vient essentiellement des méthodes



(a) Volume total de migration



(b) Coupe de la nouvelle partition



(c) Nombre de communications

Figure 9. Repartitionnement d'une grille $32 \times 32 \times 32$ *de* $M = 8$ *vers* $N \in [2, 24]$ *processeurs*

Cette procédure est répétée 10 fois pour obtenir des résultats en moyenne. Tous les résultats expérimentaux sont présentés sur la figure 9. La figure 9a donne les volumes de migrations des différentes méthodes ainsi que le volume de migration optimal théorique : $M \times \frac{V}{N}$ si $N > M$, $N \times \frac{V}{M}$ sinon, avec V le volume total. On peut voir que la méthode $M \times N$ basée sur la chaîne, bien que souvent bonne, pose problème dans certains cas défavorables (ajout ou retrait d'un seul processeur). En revanche, la méthode $M \times N$ généralisée donne un volume très proche de l'optimal dans tous les cas, grâce au choix d'un hypergraphe de repartitionnement optimisant la migration. Sur la figure 9b, on voit que cette optimisation du volume de migration vient au prix d'une coupe légèrement plus importante que celle du *Scratch-Remap* qui est la référence pour ce critère. Les autres repartionneurs qui ne sont pas prévus pour le cas du changement de nombre de processeurs ont des difficultés à optimiser conjointement les deux critères. Notons que pour Scotch ou ParMetis, le choix d'un facteur de repartitionnement plus élevé (i.e. un coût de migration plus faible) améliorerait la coupe au prix d'un volume de migration plus important, les rapprochant ainsi de la méthode *Scratch-Remap*. La figure 9c permet de vérifier que le schéma de communication imposé par la méthode $M \times N$ est bien respecté et que le nombre de communications est donc optimal. Contrairement au Scratch-Remap ou à Scotch, ParMetis donne d'assez bons résultats pour ce critère, sans être optimal comme la méthode $M \times N$.

Le temps de communication dans la phase de migration pour un repartitionnement de 8 vers 11 parties a été mesuré expérimentalement avec OpenMPI sur un réseau InfiniBand. La figure 10 présente les temps relatifs par rapport à la méthode *Scratch-Remap* et montre une amélioration de l'ordre de 6% pour la méthode $M \times N$ généralisé par rapport à la méthode Scratch-Remap. Cela est evidemment dû a fait que le volume de migration est plus petit dans notre méthode, mais aussi au fait que le schéma de communication imposé par notre méthode est mieux agencé. En effet, un nombre de communication plus petit tend à augmenter la taille moyenne des messages et donc à améliorer le débit cumulé.

6. Conclusion et perspectives

Nous avons présenté dans cet article deux algorithmes de repartitionnement de graphe permettant de faire varier le nombre de processeurs en optimisant les communications entre ceux-ci. Notre premier algorithme s'appuie sur un résultat théorique de repartitionnement d'une chaîne, optimal en nombre de communications. Le second

---

de bissections récursives, qui sont mal adaptés pour prendre en compte des sommets fixes, contrairement aux méthodes *k-way*. De tels travaux existent comme par exemple kPaToH (Aykanat *et al.*, 2008), mais ce partitionneur n'est actuellement pas disponible. Scotch propose un raffinement multi-niveaux *k-way*, mais la partition du graphe le plus contracté est cependant obtenue par bissections récursives. Dans certains cas critiques, il faut lourdement corriger cette partition pendant la phase de raffinement. La partition finale ainsi obtenue est de bonne qualité mais le temps d'exécution du partitionneur Scotch est considérablement allongé.

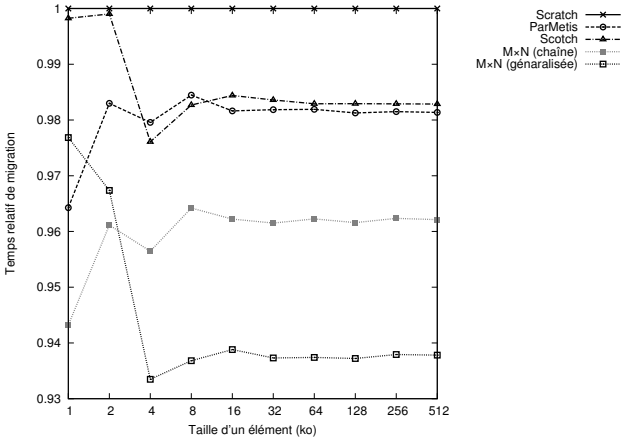


Figure 10. Temps de migration pour une grille $32 \times 32 \times 32$ *de 8 parties vers 11 suivant la taille des éléments (temps relatif par rapport au Scratch-Remap)*

algorithme formule une généralisation qui permet, en plus, d'optimiser le volume de migration.

Les résultats expérimentaux obtenus pour nos algorithmes ont montré qu'il était possible d'optimiser la migration sans dégrader la coupe exagérément. De plus, il est intéressant de voir que sur un cas simple comme une grille 3D, nos stratégies donnent comparativement de meilleurs résultats que les partitionneurs actuels.

Notons par ailleurs que notre méthode de repartitionnement $M \times N$ s'étend naturellement aux hypergraphes, mais il est difficile de la mettre en œuvre dans la mesure où il n'existe pas actuellement de partitionneur d'hypergraphe *k-way* supportant les sommets fixes.

Nous envisageons plusieurs perspectives pour nos travaux. Tout d'abord, nos algorithmes peuvent facilement être parallélisés en utilisant un partitionneur parallèle. Nous nous concentrons actuellement sur le problème plus général de repartitionnement d'un graphe dans le cas où la charge et le nombre de processeurs varient. Les algorithmes que nous présentons ici sont une étape importante dans ce sens et nous espérons que ces nouveaux travaux puissent avoir un réel impact pour l'équilibrage dynamique des simulations numériques adaptatives.

Bibliographie

Aykanat C., Cambazoglu B. B., Findik F., Kurc T. (2007, January). Adaptive decomposition and remapping algorithms for object-space-parallel direct volume rendering of unstructured grids. *J. Parallel Distrib. Comput.*, vol. 67, p. 77–99.

Aykanat C., Cambazoglu B. B., Uçar B. (2008, May). Multi-level direct k-way hypergraph partitioning with multiple constraints and fixed vertices. *J. Parallel Distrib. Comput.*, vol. 68,

p. 609–625.

Catalyurek U. V., Boman E. G., Devine K. D., Bozdağ D., Heaphy R. T., Riesen L. A. (2009). A repartitioning hypergraph model for dynamic load balancing. *J. Parallel Distrib. Comput.*, vol. 69, n° 8, p. 711–724.

Duchenne O., Bach F. R., Kweon I.-S., Ponce J. (2009). A tensor-based algorithm for high-order graph matching. In *Cvpr*, p. 1980-1987. IEEE.

Fourestier S., Pellegrini F. (2011, mai). Adaptation au repartitionnement de graphes d'une méthode d'optimisation globale par diffusion. In *Proc. renpar'20, saint-malo, france.*

Garey M. R., Johnson D. S. (1979). *Computers and intractibility: A guide to the theory of np-completeness*. W. H. Freeman.

Hendrickson B., Devine K. (2000). Dynamic load balancing in computational mechanics. In *Computer methods in applied mechanics and engineering*, vol. 184, p. 485–500.

Hendrickson B., Leland R. W., Driessche R. V. (1997). Skewed graph partitioning. In *Eighth siam conf. parallel processing for scientific computing.*

Iqbal S., Carey G. F. (2005). Performance analysis of dynamic load balancing algorithms with variable number of processors. *Journal of Parallel and Distributed Computing*, vol. 65, n° 8, p. 934 - 948.

Leordeanu M., Hebert M. (2005). A spectral technique for correspondence problems using pairwise constraints. In *Proceedings of the tenth ieee international conference on computer vision - volume 2*, p. 1482–1489. Washington, DC, USA, IEEE Computer Society.

Oliker L., Biswas R. (1998, August). Plum: parallel load balancing for adaptive unstructured meshes. *J. Parallel Distrib. Comput.*, vol. 52, p. 150–177.

Pilkington J., Baden S. (1996, mars). Dynamic partitioning of non-uniform structured workloads with spacefilling curves. *Parallel and Distributed Systems, IEEE Transactions on*, vol. 7, n° 3, p. 288 -300.

Schloegel K., Karypis G., Kumar V. (1997). Multilevel diffusion schemes for repartitioning of adaptive meshes. *Journal of Parallel and Distributed Computing*, vol. 47, n° 2, p. 109 - 124.

Schloegel K., Karypis G., Kumar V. (2000). A unified algorithm for load-balancing adaptive scientific simulations. In *Proceedings of the 2000 acm/ieee conference on supercomputing (cdrom).* Washington, DC, USA, IEEE Computer Society.

Teresco J. D., Devine K. D., Flaherty J. E. (2006). Partitioning and dynamic load balancing for the numerical solution of partial differential equations. In T. J. Barth *et al. (Eds.),* Numerical solution of partial differential equations on parallel computers*, vol. 51, p. 55-88. Springer Berlin Heidelberg.*

Van Driessche R., Roose D. (1995). Dynamic load balancing with a spectral bisection algorithm for the constrained graph partitioning problem. In B. Hertzberger, G. Serazzi (Eds.), High-performance computing and networking*, vol. 919, p. 392-397. Springer Berlin / Heidelberg.*

Reçu le 26 septembre 2011
Accepté le 21 juin 2012

Clément Vuchener. *Clément Vuchener est doctorant en Informatique à l'université Bordeaux I, membre de l'équipe-projet INRIA HiePACS et du Laboratoire Bordelais de Recherche en Informatique (LaBRI). Sa thèse porte sur l'équilibrage de charge pour les couplages de code.*

Aurélien Esnard. *Aurélien Esnard est maître de conférence à l'université Bordeaux I depuis 2006, membre de l'équipe-projet INRIA HiePACS et du Laboratoire Bordelais de Recherche en Informatique (LaBRI). Dans le contexte du calcul haute-performance (HPC), ses travaux portent sur la thématique du couplage de codes et plus récemment sur l'équilibrage dynamique de charge.*