Repartitionnement d'un graphe de M vers N processeurs : application pour l'équilibrage dynamique de charge

Clément Vuchener, Aurélien Esnard

RenPar'20 / SympA'14 / CFSE 8
Saint-Malo, France, du 10 au 13 mai 2011

Repartitionnement d'un graphe de M vers N processeurs : application pour l'équilibrage dynamique de charge

Clément Vuchener, Aurélien Esnard

INRIA Bordeaux-Sud-Ouest et LaBRI UMR CNRS 5800,
351, cours de la Libération, 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 un nouvel algorithme de repartitionnement de graphe permettant de faire varier le nombre de processeurs, en supposant que la charge du graphe n'est pas modifiée. Cet algorithme optimise conjointement la coupe et la migration 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.

Mots-clés : calcul haute-performance, équilibrage dynamique de charge, partitionnement de graphe

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. Dans ce contexte, il est nécessaire de calculer périodiquement 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 [2] :

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 de la coupe et celle de la migration selon 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 [3, 9]. 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, notamment du point de vue de la consommation des ressources, comme le démontre *Iqbal et al.* [5]. Nous proposons dans cet article un nouvel algorithme de repartitionnement de graphe permettant de faire varier le nombre de processeurs, en supposant que la charge du graphe n'est pas modifiée. Nous appelons ce problème : le *repartionnement* $M \times N$ de graphe. Notre algorithme optimise conjointement la coupe et la migration en s'appuyant sur un modèle de partitionnement de graphe à sommets fixes [2].

Dans la section suivante, nous allons examiner les travaux existants autour de l'équilibrage dynamique. Puis nous présentons à la section 3 des résultats préliminaires utiles pour la compréhension de notre algorithme de repartitionnement détaillé à la section 4. Finalement, nous présentons à la section 5 quelques résultats expérimentaux comparatifs avec d'autres algorithmes avant de conclure.

2. Travaux existants

Il existe de nombreux travaux dans le domaine de l'équilibrage dynamique, comme le montrent Hendrickson *et al.* [3] ou encore Teresco *et al.* [9]. 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* [6], 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 [8], 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 dans [1, 2] consiste à repartitionner en minimisant conjointement le volume de données à déplacer. 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 conservent 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 données à déplacer. 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 [2].

Il existe encore de nombreux travaux sur l'équilibrage dynamique incluant des méthodes géométriques de type *Recursive Coordinate Bisection* (RCB) [3] ou à base de *Space-Filling Curve* (SFC) [7], les méthodes spectrales récursives [10], ou encore d'autres approches plus exotiques comme le partitionnement basé sur un modèle de *skewed graph* [4]. Tous ces travaux sont très intéressants, mais sont néanmoins 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, seul Iqbal *et al.* [5] étudient la problématique de l'équilibrage de charge en faisant varier le nombre de processeurs. Ils montrent avec l'exemple d'un code adaptatif (AMR) comment il est possible d'économiser les ressources utilisées en ajustant dynamiquement le nombre de processeurs, au

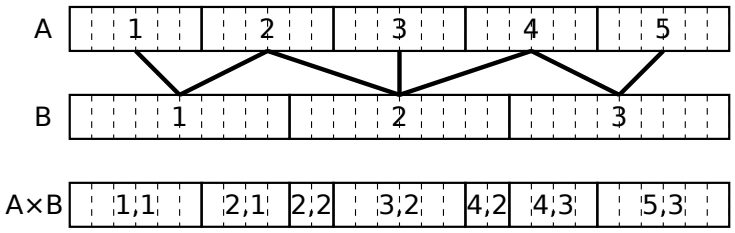


FIGURE 1 – Repartitionnement de $M = 5$ vers $N = 3$ d'une chaîne de longueur $L = 30$ éléments.

lieu d'utiliser systématiquement toutes les ressources.

3. Résultats préliminaires

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, ou plus simplement d'un tableau de données à une dimension. Rappelons que nous nous intéressons uniquement au problème de variation du nombre de processeurs, en supposant que la charge du graphe n'est pas modifiée. Par ailleurs, nous supposons *dans un premier temps* que les ensembles des processeurs émetteurs et récepteurs sont disjoints, c'est-à-dire que aucune donnée ne reste en place lors de la migration. Nous reviendrons sur ce point à la fin de la section suivante (Sec. 4.3).

3.1. Repartitionnement MxN 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 $M' = M/\mathrm{PGCD}(M, N)$ et $N' = N/\mathrm{PGCD}(M, N)$. En effet, il faut remarquer que le même motif de communication se répète $\mathrm{PGCD}(M, N)$ fois, avec $M' + N' - 1$ messages à chaque fois. Au final, on établit que le nombre de communications est $M + N - \mathrm{PGCD}(M, N)$ 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 d'au plus 1 (et réciproquement). 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.

3.2. 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.

4. Algorithme de repartitionnement $M \times N$ d'un graphe

Le repartitionnement $M \times N$ de la chaîne peut s'appliquer dans un cas plus général 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 $M \times N$ de graphe.

4.1. Recherche de chemin

Étant donné un partitonnement initial du graphe en M, nous cherchons un *bon* chemin dans le graphe des parties. 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-complet (similaire au problème du voyageur de commerce). 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 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.

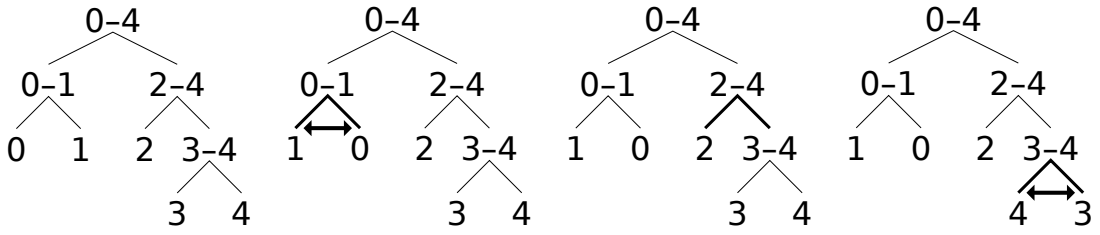
Un exemple d'exécution de cet algorithme est présenté sur la figure 2. Le graphe est partitionné en $M = 5$ comme sur la figure 3a. La figure 2a présente l'arbre de bissection initial [2]. On débute le parcours en largeur. La racine, contenant toutes les parties, n'a aucun précédent ou suivant, on ne 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 2 - 4 sont comparées car on ne sait pas encore si la partie suivante sera 2, 3 ou 4. La partie 0 est plus connecté avec 2 - 4 donc l'ordre choisi est 1 puis 0. À l'étape suivante (Fig. 2c), on connait 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 0 et 2 et entre 0 et 3 - 4. Enfin à la dernière étape (Fig. 2d), on compare les connexions entre 2 et 3 et entre 2 et 4. Au final, on obtient le chemin représenté sur la figure 3b.

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

En s'inspirant des techniques de repartitionnement à sommets fixes [1, 2], on propose un nouvel algorithme de repartitionnement $M \times N$ s'appuyant sur le chemin obtenu dans la section 4.1. 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 ($i \in [1, N]$).

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



(a) L'arbre de bissections initial. (b) Choix de l'ordre des parties 0 et 1. (c) Choix de l'ordre des parties 2 et 3 - 4. (d) Choix de l'ordre des parties 3 et 4.

FIGURE 2 – Exécution de notre algorithme de recherche d'un chemin sur l'exemple de la figure 3a. L'ordre final est 1, 0, 2, 4, 3.

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 préfèrera 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 dans les résultats préliminaires (Sec. 3).

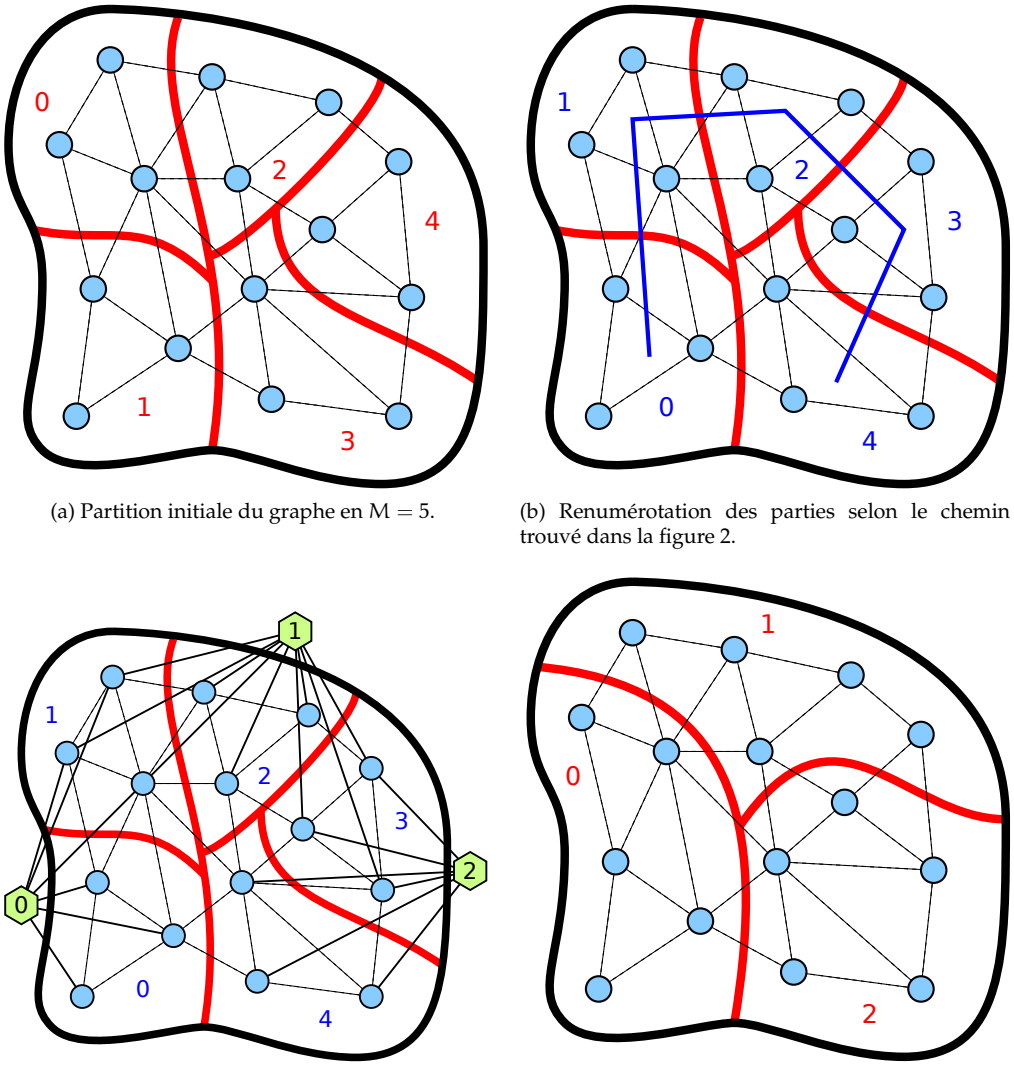
4.3. Optimisation de la migration : renumérotation des nouvelles parties

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 des nouvelles parties adéquate, nous utilisons l'algorithme glouton proposé dans PLUM [6]. On constate que cet algorithme appliqué à la matrice de communication produite par notre algorithme de repartitionnement donne de très bons résultats.

5. Résultats expérimentaux

Notre méthode de repartitionnement $M \times N$ a été comparée avec une méthode *scratch-remap*, une méthode diffusive implantée dans ParMetis et une méthode à base de sommets fixes implantée dans Zoltan. 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. Pour nos expériences, nous utilisons un graphe correspondant à une grille 3D de $32 \times 32 \times 32$ éléments (32768 sommets et 95232 arêtes). Ce graphe est initialement partitionné en $M = 8$ parties avec Zoltan. Cette partition initiale est ensuite repartitionnée en $N \in [2, 32]$ de différentes manières :
– le *scratch-remap* qui est réalisé à l'aide Zoltan (sans l'option repartitionnement) ;
– l'algorithme de repartitionnement de Zoltan avec un *facteur de repartitionnement* [3] par défaut de 100/1 ;
– celui de ParMetis avec un facteur de repartitionnement de 1000/1 qui est la valeur recommandée par la documentation ;
– l'algorithme présenté dans cet article avec différents facteurs de repartitionnement : 1/100 pour une migration minimale en priorité, 1/1 pour un compromis, 100/1 pour une coupe minimale en priorité.

3. 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) Partition initiale du graphe en M = 5.

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

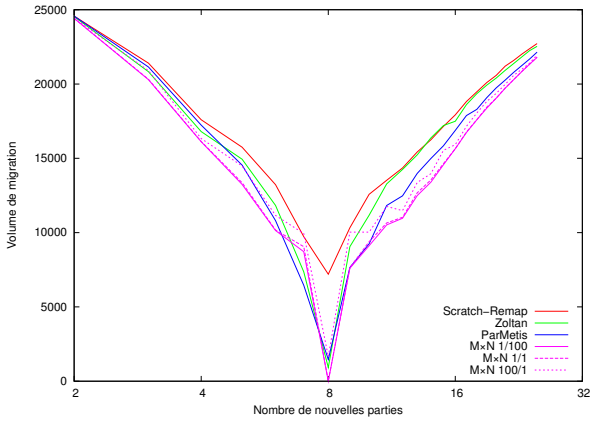
(c) Ajout de sommets fixes et d'arêtes de migration pour un repartitionnement en N = 3.

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

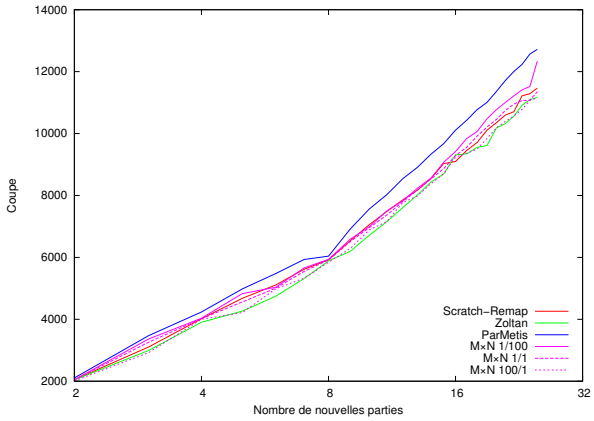
FIGURE 3 – Exemple de repartitionnement d'un graphe de M = 5 vers N = 3.

Cette procédure est répétée 20 fois pour obtenir des résultats en moyenne. Tous les résultats expérimentaux sont présentés sur la figure 4. Sur la figure 4a, on peut voir que le repartitionnement M × N offre la migration la plus basse lorsque lorsque le facteur de repartitionnement la favorise (cas 1/100). La figure 4b nous montre que cela vient au prix d'une légère dégradation de la coupe. On retrouve bien le compromis entre la coupe et la migration. La figure 4c présentant les écarts-types du volume de migration montre que notre approche est globalement plus stable. La figure 4d donne le nombre de communications nécessaire pour la migration. Comme nous l'avons vu à la section 3.1, notre approche tend à minimiser ce critère. Cela est plus ou moins bien respecté selon le coût relatif des arêtes de migration : avec un coût plus faible sur les arêtes de migration, des communications supplémentaires peuvent apparaître. Notons que le nombre de communications est fortement réduit par rapport aux autres stratégies ne prenant pas en compte ce critère.

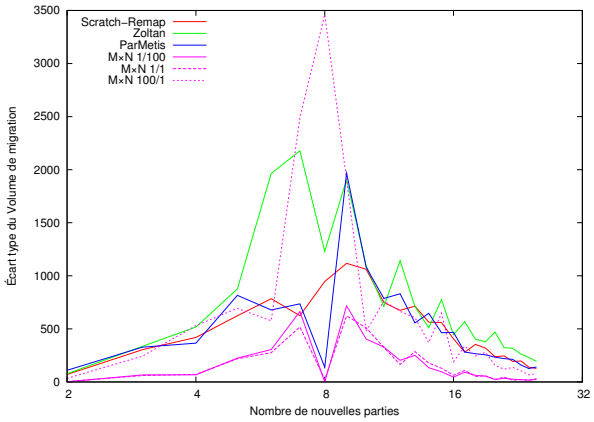
Les communications de migration se faisant en parallèle, le volume total ne donne pas précisément la durée de la phase de migration. Les temps de migration de 8 processeurs vers 12 ont donc été mesurés (pour différentes tailles de données) sur le réseau InfiniBand de PlaFRIM en utilisant OpenMPI. La



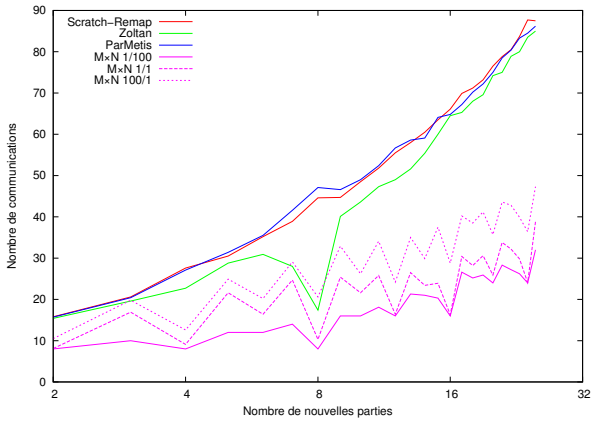
(a) Volume total de migration.



(b) Coupe de la nouvelle partition.



(c) Écart-type du volume total de migration.



(d) Nombre de communications.

FIGURE 4 – Repartitionnement d'une grille $32 \times 32 \times 32$ à partir de 8 processeurs.

figure 5 donne les temps de migration relatifs au temps de la stratégie *scratch-remap* et montrent que nous gagnons jusqu'à 10%.

6. Conclusion

Nous avons présenté dans cet article une méthode d'équilibrage dynamique permettant de faire varier le nombre de processeurs en optimisant les communications entre ceux-ci. Les résultats expérimentaux ont confirmé l'importance du compromis entre coupe et migration. Un coût de migration élevé tend à mieux faire respecter le schéma de communication *optimal* mais cela se fait au détriment de la coupe. Notons par ailleurs que notre approche donne des résultats plus stables comparativement aux autres méthodes.

Nous envisageons plusieurs perspectives pour nos travaux. Tout d'abord, notre algorithme peut être facilement parallélisé en utilisant un partitionneur parallèle et en parallélisant la recherche du chemin. Par ailleurs, on peut envisager améliorer notre algorithme en utilisant directement le graphe des parties plutôt qu'un chemin dans celui-ci. En outre, il serait intéressant d'étudier le problème de repartitionnement $M \times N$ dans le cas plus général où la charge varie.

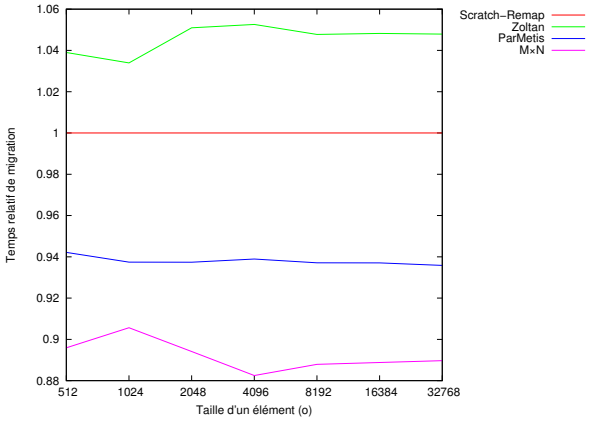


FIGURE 5 – Temps de migration pour une grille $32 \times 32 \times 32$ de 8 parties vers 12 suivant la taille des éléments (temps relatif par rapport au scratch-remap).

Bibliographie

1. Cevdet Aykanat, B. Barla Cambazoglu, Ferit Findik, and Tahsin Kurc. Adaptive decomposition and remapping algorithms for object-space-parallel direct volume rendering of unstructured grids. *J. Parallel Distrib. Comput.*, 67 :77–99, January 2007.
2. Umit V. Catalyurek, Erik G. Boman, Karen D. Devine, Doruk Bozdağ, Robert T. Heaphy, and Lee Ann Riesen. A repartitioning hypergraph model for dynamic load balancing. *J. Parallel Distrib. Comput.*, 69(8) :711–724, 2009.
3. Bruce Hendrickson and Karen Devine. Dynamic load balancing in computational mechanics. In *Computer Methods in Applied Mechanics and Engineering*, volume 184, pages 485–500, 2000.
4. Bruce Hendrickson, Robert W. Leland, and Rafael Van Driessche. Skewed graph partitioning. In *Eighth SIAM Conf. Parallel Processing for Scientific Computing*, 1997.
5. Saeed Iqbal and Graham F. Carey. Performance analysis of dynamic load balancing algorithms with variable number of processors. *Journal of Parallel and Distributed Computing*, 65(8) :934 – 948, 2005.
6. Leonid Oliker and Rupak Biswas. Plum : parallel load balancing for adaptive unstructured meshes. *J. Parallel Distrib. Comput.*, 52 :150–177, August 1998.
7. J.R. Pilkington and S.B. Baden. Dynamic partitioning of non-uniform structured workloads with spacefilling curves. *Parallel and Distributed Systems, IEEE Transactions on*, 7(3) :288 –300, March 1996.
8. Kirk Schloegel, George Karypis, and Vipin Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *Journal of Parallel and Distributed Computing*, 47(2) :109 – 124, 1997.
9. James D. Teresco, Karen D. Devine, and Joseph E. Flaherty. Partitioning and dynamic load balancing for the numerical solution of partial differential equations. In Timothy J. Barth, Michael Griebel, David E. Keyes, Risto M. Nieminen, Dirk Roose, Tamar Schlick, Are Magnus Bruaset, and Aslak Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering*, pages 55–88. Springer Berlin Heidelberg, 2006.
10. Rafael Van Driessche and Dirk Roose. Dynamic load balancing with a spectral bisection algorithm for the constrained graph partitioning problem. In Bob Hertzberger and Giuseppe Serazzi, editors, *High-Performance Computing and Networking*, volume 919 of *Lecture Notes in Computer Science*, pages 392–397. Springer Berlin / Heidelberg, 1995.