Graph Repartitioning with both Dynamic Load and Dynamic Processor Allocation

Clément Vuchener, Aurélien Esnard

Graph Repartitioning with both Dynamic Load and Dynamic Processor Allocation

Clément VUCHENER [a] and Aurélien ESNARD [a]
[a] *Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France.*
CNRS, LaBRI, UMR 5800, F-33400 Talence, France.
HiePACS Project, INRIA, F-33400 Talence, France.

Abstract. Dynamic load balancing is an important step conditioning the performance of parallel programs, like adaptive mesh refinement codes. If the global workload varies drastically over time (such that memory is exceeded), it can be relevant to adjust the number of processors while maintaining the load balanced. We propose two different solutions, that extend classic graph repartitioning approaches to accept a variable number of processors: one based on biased partitioning method and one based on a diffusive method. We call this problem: the $M \times N$ *graph repartitioning problem*. Finally, an experimental study on real-life graphs validates our algorithms against state-of-the-art methods.

Keywords. high-performance computing, parallelism, dynamic load-balancing, graph partitioning, repartitioning.

1. Introduction

In the field of scientific computing, the load balancing is an important step conditioning the performance of parallel programs. The goal is to distribute the computational load across multiple processors in order to minimize the execution time. For some scientific applications, whose load evolution is unpredictable (e.g. adaptive mesh refinement), it is required to periodically compute a new balancing at runtime, using a dynamic load balancing algorithm.

This is a well-known problem [1,2], that is unfortunately NP-hard. The most common approach to solve it uses graph repartitioning methods. Given a weighted graph G and an *unbalanced* partition P of this graph in M parts, the classic repartitioning problem aims to compute a new partition P' of G (in M parts) that satisfies the following objectives: 1) balancing the *load* across subdomains, 2) minimizing the *edge-cut*, and 3) minimizing the *migration volume*. The first objective aims to minimize the whole computation time, which consists of dividing the graph in equal weight subdomains (up to an imbalance tolerance). The second objective aims to minimize the communication time, which consists of minimizing the weight of edges cut between subdomains. Finally, the third objective aims to minimize the migration time, which consists of minimizing the weight of vertices that are redistributed among subdomains. Therefore, there is a good trade-off to find between these different criteria.

When the global load of an application varies drastically (such that memory is exceeded), it can be profitable to adapt the number of processors used at runtime in or-

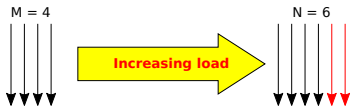


Figure 1. Dynamic processor allocation for a single code whose load is increasing.

der to preserve the parallel code efficiency and/or to improve the resource consumption (Fig. 1). For instance, in the context of adaptive mesh refinement (AMR), Iqbal *et al.* suggest the following approach [3,4]: one usually starts computation on a coarse grid with few processors, then one changes the number of processors working on the computation at runtime[1] as the grid is progressively refined or coarsened. We call this problem of dynamic load balancing: *the $M \times N$ graph repartitioning problem.*

Nowadays, the most common approach for dynamic load balancing of parallel adaptative simulations uses graph/hypergraph repartitioning, with popular methods such as scratch-remap [6], diffusion [7,8] or biased partitioning with fixed vertices [9,10]. Basically, these methods aim to optimize different primary objectives: scratch-remap first minimizes the edge-cut, while diffusion first minimizes the migration volume. The biased partitioning tends to find a trade-off between these two criteria. All these repartitioning methods are currently available in mature and efficient software tools, such as ParMetis [11], Zoltan [12] or Scotch [13]. Nevertheless, all these methods are limited to the case where the number of processors is initially fixed and will not be modified at runtime.

In recent work [14], we focused on this particular problem. We proposed a new $M \times N$ graph repartitioning algorithm, limited to the case where the initial partition in M is well balanced. Using a biased partitioning method with fixed vertices in a similar way to Zoltan [10], our approach enforces an *optimal* communication scheme for migration, while minimizing the *edge-cut* as a trade-off. One proves our communication scheme to be optimal in the way it reaches minimal migration volume and minimal number of messages and gives a method to construct such an *optimal* migration matrix. Unfortunately, this solution doesn't work anymore in the general case where P is initially unbalanced.

In this paper, we first present background definitions and a formal statement of the $M \times N$ graph repartitioning problem, illustrated by some samples (Sec. 2). Then, we propose two different and original solutions to solve this problem: one based on biased partitioning method that extends our previous work (Sec. 3) and another one based on a diffusive method (Sec. 4). Our methods require two main steps: first, one constructs a *good* migration matrix, then one computes a new partition according to this matrix. Finally, we give experimental results comparing our approach against state-of-the art partitioning tools (Sec.5).

[1] Thanks to dynamic processor allocation mechanisms such as those provided by MPI2 (i.e. `MPI_COMM_SPAWN` routine) [5].

2. $M \times N$ Graph Repartitioning

In this section, we first give some background definitions about graph partitioning & repartitioning and describe some metrics for migration matrix. Then, we propose a formal statement of the $M \times N$ graph repartitioning problem, illustrated by some samples.

2.1. Background Definitions

Let consider a graph $G = (V,E)$, where V is the vertex set and E is the edge set. Each vertex v has a weight w_v and a size s_v, that represent the computational load and the redistribution cost, respectively. In this paper, we assume that w_v and s_v are the same. In addition, each edge e has a weight w_e, that represents the communication cost.

A *k-way* partition of V is a set of k disjoint subsets $V_1, V_2, \ldots, V_k$, such as all subsets cover V. The *partition imbalance* is defined as $ub(G,P) = \max_{V_i \in P}(\sum_{v \in V_i} w_v/(W/k))$ with W the total vertex weight of G. A partition P is said to be balanced if $ub(G,P) \leq (1+\varepsilon)$ with ε the imbalance tolerance. A typical imbalance tolerance is $\varepsilon = 0.05$, that means that each part must not exceed 5% of the ideal weight. The cut size (or *edge-cut*) of a given partition P is defined as the weight of edges whose ends belong to two different parts in P: $cut(G,P) = \sum_{e \in F} w_e$ where $F = \{(a,b) \in E, a \in V_i \wedge b \in V_j \wedge i \neq j\}$.

We define the *quotient graph* G/P with respect to the partition P of G, the graph where vertices from a same part are merged into a single vertex. Therefore, this quotient graph has as many vertices as there are parts in P and the weight of an edge is equal to the edge-cut between the two parts.

In the context of graph repartitioning, we consider two partitions of G: the former partition P in M parts and the newer partition P' in N parts. The *migration matrix* C is defined as a matrix $C = (C_{i,j})$ of size $M \times N$, where $C_{i,j}$ is the weight of vertices in both the former part i and the newer part j. It represents the amount of data exchanged between the "former" processor i and the "newer" processor j. In other terms, non-diagonal terms of C represent the amount of data redistributed to a different processor, while diagonal terms $C_{i,i}$ represent the amount of data that remains on the same processor and are not redistributed. As a consequence, the migration matrix satisfies the following constraints: the weight of the row i of C is equal to the weight of the former part V_i of P, while the weight of the column j is equal to the weight of the newer part V'_j of P'.

To evaluate the *redistribution cost* for a given migration matrix C, the common metrics used are $TotalV$ and $MaxV$ [6,8]. The migration volume $TotalV$ is defined as the sum of non-diagonal terms, i.e., $TotalV = \sum_{i \neq j} C_{i,j}$. In the same way, one can define $MaxV$ as the maximum amount of data sent (or received) by a processor. One introduces in this paper two new metrics, called $TotalZ$ and $MaxZ$. First, the total number of migration messages, $TotalZ$, is defined as the number of non-diagonal terms in C that are not null. It reflects the number of messages exchanged between former and newer processors. Then, $MaxZ$ is the maximum number of messages sent (or received) by a processor.

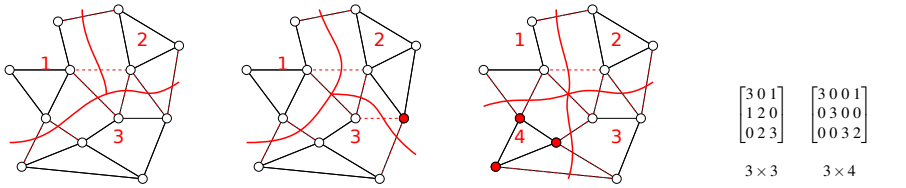
In the case where both P and P' are perfectly balanced and $M \neq N$, we showed in a previous work [14] that $TotalZ$ is minimal for $max(M,N) - GCD(M,N)$ and that $TotalV$ is minimal for $|M-N| \times W/max(M,N)$.

2.2. Problem Statement

Let us consider an imbalance tolerance ε. Given an unbalanced graph partition P of G in M parts, the $M \times N$ repartitioning problem aims to compute a new graph partition P' of G in N parts that satisfies the following objectives:

1. balancing the load, $ub(G,P') \leq 1+\varepsilon$,
2. minimizing the communication cost, $cut(G,P')$,
3. minimizing the redistribution cost, $TotalV$.

Firstly, if $M = N$, this problem is the classic graph repartitioning problem; secondly, if the initial partition P is already balanced and $M \neq N$, this is a particular case we have recently studied in [14]; finally, if P is unbalanced and $M \neq N$, this is the general case we investigate in this paper.



(a) Initial partition (b) Repartitioning 3×3 (c) Repartitioning 3×4 (d) Migration matrices

Figure 2. Example of graph repartitioning in the cases 3×3 and 3×4. The partition is given by the red line. The redistributed vertices are in red and the edges cut are in dashed red.

The figure 2 presents two examples of $M \times N$ graph repartitioning: one in 3 parts and another in 4 parts. The initial partition in $M = 3$ parts given on figure 2a is unbalanced, for the ideal weight of each subdomain should be 4 (assuming each vertex weight is 1). The figure 2b shows a new balanced partition in $N = 3$ parts, where 8 edges are cut and only 1 vertex is redistributed ($TotalV = 1$). The figure 2c presents another partition in $N = 4$ parts where the cut is 11, the migration volume $TotalV$ is 3 ($MaxV = 3$) and the number of messages $TotalZ$ is 2 ($MaxZ = 2$). The figure 2d gives the migration matrices in this two cases.

3. An Extended Biased Partitioning Algorithm for MxN Repartitioning

The first solution we propose is based on a biased partitioning method that extends our previous solution [14] to the case where P is unbalanced.

3.1. Construction of a Good Migration Matrix

In this section, we describe how to create a good migration matrix that minimizes both $TotalV$ and $TotalZ$ while respecting the load balance constraint.

Our algorithm uses a greedy approach to choose which processors communicate together and how much data is moved. It fills the migration matrix element by element, starting with a null matrix. Adding an element can be seen as taking vertices from a former part (associated with the row) and giving them to a newer part (associated with

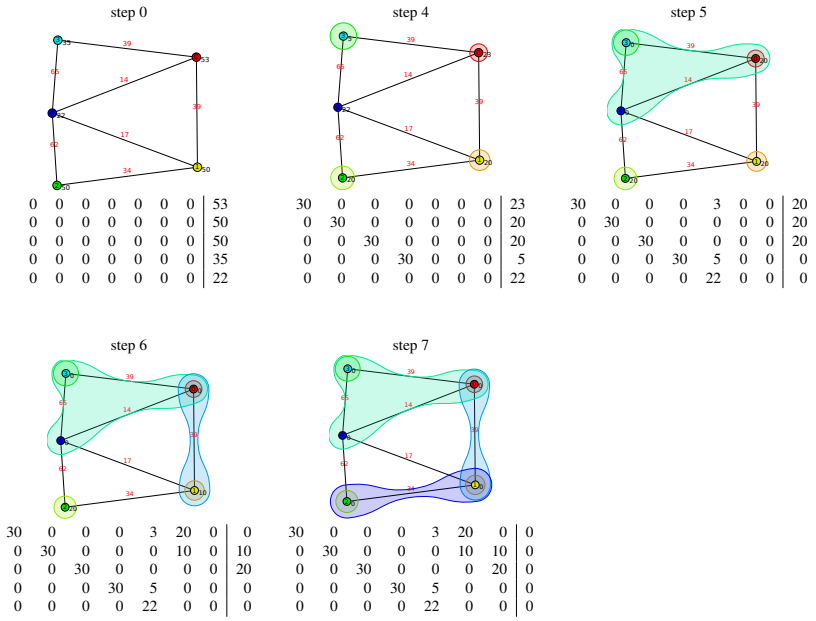


Figure 3. Illustration of the greedy algorithm on a 5×7 sample, that constructs iteratively the 7 new parts.

the column). In order to add as few elements as possible in the matrix, and thus reducing the number of migration messages, each element added is chosen as large as possible.

Firstly, in order to minimize the migration volume, one tries to maximize the values on the matrix diagonal by adding an element in the diagonal for each former part whose weight is higher than the ideal newer part weight. The value of this element is the ideal newer part weight, thus, as the newer part is complete, there will be no other non-zero element in this column. This means that this newer part will be made only from vertices of the same former part.

Then, the remaining columns are filled from several former parts according to the quotient graph G/P. Only former parts whose row is not yet filled (all its vertices are not given yet), are considered. Starting from a peripheral former part, other former parts are added until the newer part has taken enough vertices. Former parts are chosen according to their connection with the already chosen former parts in the quotient graph using a score computed from the set V_k of chosen former parts: $|E_k| \times \sum_{e \in E_k} w_e$ with E_k the edges of G/P restricted to V_k.

The figure 3 shows an example execution of this algorithm. At each step, we present the matrix with, on its right, how many vertices remains for each row (former part). The sets of former parts V_k that each newer part k takes vertices from, are drawn over the quotient graph. Starting with an empty matrix at step 0, elements are added on the diagonal until step 4. The former part 4 is too small to accept a diagonal element. The next newer part at step 5 is built by taking all the vertices from part 3 and then part 4 and completed with a few vertices from part 0. Only former parts 0, 1 and 2 still have vertices to give and will be used to build the two remaining newer parts in a similar way at step 6 and 7.

3.2. Biased Partitioning according to a Migration Matrix

In order to achieve a repartitioning that respects the previously built migration matrix C, we propose an algorithm that extends the approach used in Zoltan [10] based on fixed vertices[2]. A fixed vertex is added for each newer part j and is connected to all the regular vertices of G belonging to a former part i if and only if $C_{i,j}$ is not zero. These new edges added to G are called *migration edges*. In our approach, a fixed vertex can be connected to multiple former parts (depending on the migration matrix), while in the Zoltan method it is strictly connected to a single former part. Finally, this enriched graph is partitioned in N parts: when minimizing the edge-cut, the partitioner will try to cut as few as possible migration edges, enforcing the communication scheme imposed by the migration matrix [14].

4. An Extended Diffusion-Based Algorithm for MxN Repartitioning

The second solution we propose is based on a diffusive method in a similar way to ParMetis [7,8], but extended for the $M \times N$ case. In the original diffusive method, vertices are migrated only between two neighboring parts according to the quotient graph. Hence, it is required to extend this method to make it possible for vertices to migrate to newer *empty* parts when $N > M$. As in the previous solution (Sec. 3), one first looks for a good migration matrix C, then one computes the new partition with respect to C, such that it is decided which vertices of G will move.

We illustrate our algorithm on a simple 5×7 repartitioning case, running on a regular grid G (of size 100×100) shown on figure 4. The initial partition P of G in $M = 5$ parts is shown on figure 4a. This partition is unbalanced by changing vertex weight according to the figure 4b, given a total load increase of $+67.8\%$ with maximum imbalance of 40%[3]. The quotient graph $Q = G/P$ sums up the initial state (Fig. 4c).

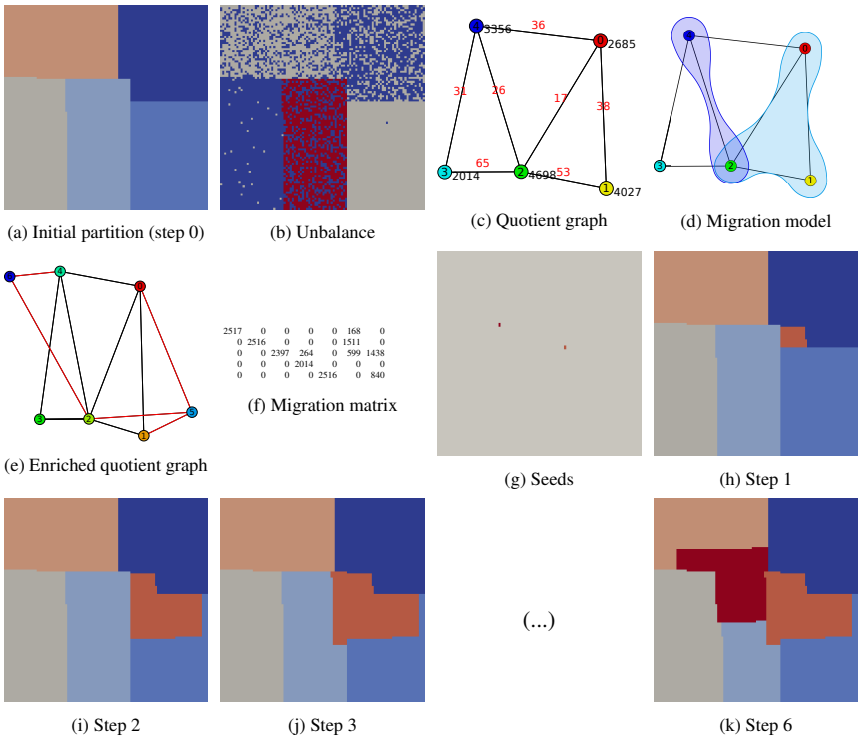
4.1. Construction of a Good Migration Matrix

The first step of our algorithm must decide where to place the newer parts relatively to the former parts. When $N > M$, there are $K = M - N$ newer parts which must be created from zero. Thanks to a greedy heuristic that retains the spirit of the method explained at section 3.1, one computes a migration model $\mathscr{M}$ as shown on figure 4d. Two new parts are added: the part 5 that will take vertices from former parts $\{0,1,2\}$ and the part 6 that will take vertices from former parts $\{2,4\}$. Then, one creates a graph $\tilde{Q}$, that enriches the quotient graph Q by adding K new vertices connected to some vertices of Q according to the migration model $\mathscr{M}$ (Fig. 4e).

The second step consists in computing the vertex weight $C_{i,j}$ to be moved from part i to part j, for all edges (i,j) in $\tilde{Q}$, such that the migration volume is minimized ($TotalV = \sum_{i \neq j} C_{i,j}$). This is simply computed with a linear programming solver (GLPK [15]) in a similar way to [16,17]. The figure 4f shows the optimal migration matrix C obtained for out chosen model $\mathscr{M}$, with a migration volume of 4820 (28.7%).

[2] When computing a partition, fixed vertices are those previously assigned to a given part, while regular vertices are free to be assigned in any parts.

[3] More precisely, one sets a weight of 2 to gray vertices and 3 to red vertices, while all blue vertices remain unchanged with a weight of 1.



(a) Initial partition (step 0) (b) Unbalance (c) Quotient graph (d) Migration model

(e) Enriched quotient graph (f) Migration matrix (g) Seeds (h) Step 1

(i) Step 2 (j) Step 3 (k) Step 6

Figure 4. Illustration of the diffusion-based repartitioning algorithm in the case 5×7 on a regular grid.

4.2. Migration of Vertices according to a Migration Matrix

Once we have obtained a good migration matrix C, a classic diffusion-scheme is performed to choose precisely which vertices will migrate across subdomain frontiers and to compute the new partition P'. When $M < N$, our method requires an additionnal step. Indeed, as the newer parts j (such that $M \leq j < N$) are intially empty, no frontier exists to initiate the migration of vertices. To overcome this issue, some seeds are computed for those parts as shown on figure 4g. Then, one performs $TotalZ$ local migration steps between parts taken two by two. For each step, $C_{i,j}$ vertex weight must be moved from the source part i to the destination part j following the classic Fiduccia-Mattheyses gain formula to improve the edge-cut [18]. The figure 4 shows this 6 steps, that starts to create the new part 5 on steps 1 to 3, and so on, until to obtain the final partition on step 6 with an edge-cut of 403 (Fig. 4k).

5. Preliminary Experimental Results

We present in this section some preliminary results for a variety of real-life graphs, described on table 1. Those graphs are publicly available from *the university of Florida sparse matrix collection* [19] except for *grid3d*, which is a regular $100 \times 100 \times 100$ cubic grid. Our $M \times N$ graph repartitioning methods, based on biased partitioning (MxN BIASED) and based on diffusion (MxN DIFF), are compared against state-of-the-art par-

graph	description	$\|V\|$	$\|E\|$	d
grid3d	3D grid	1,000,000	2,970,000	5.94
cfd2	computational fluid dynamics	123,440	1,482,229	24.02
crankseg_2	structural problem	63,838	7,042,510	220.64
thermal2	thermal problem	1,228,045	3,676,134	5.99
brack2	2D/3D problem	62,631	366,559	11.71
wave	2D/3D problem	156,317	1,059,331	13.55
cage12	DNA electrophoresis	130,228	951,154	14.61

Table 1. Description of the graphs used in benchmark. The value d represents the average degree of the graph, it is computed from $\frac{2\times|E|}{|V|}$.

titioners (Zoltan, Scotch, ParMetis) and against a Scratch-Remap[4] method (SR). We use Zoltan 3.6, ParMetis 4.0.2 and Scotch 6.0 with default parameters and a migration cost fixed to 1. These partitioners are not designed for $M \times N$ repartitioning, but it is still possible to perform a classic repartitioning in N parts, assuming some parts are empty when $M < N$. The MxN BIASED method uses a modified version of Scotch with a *k-way greedy graph growing* heuristic as initial partitioning in the multilevel framework. We prefer such a heuristic compared to the classic recursive bipartitioning, that fails to handle fixed vertices in some cases.

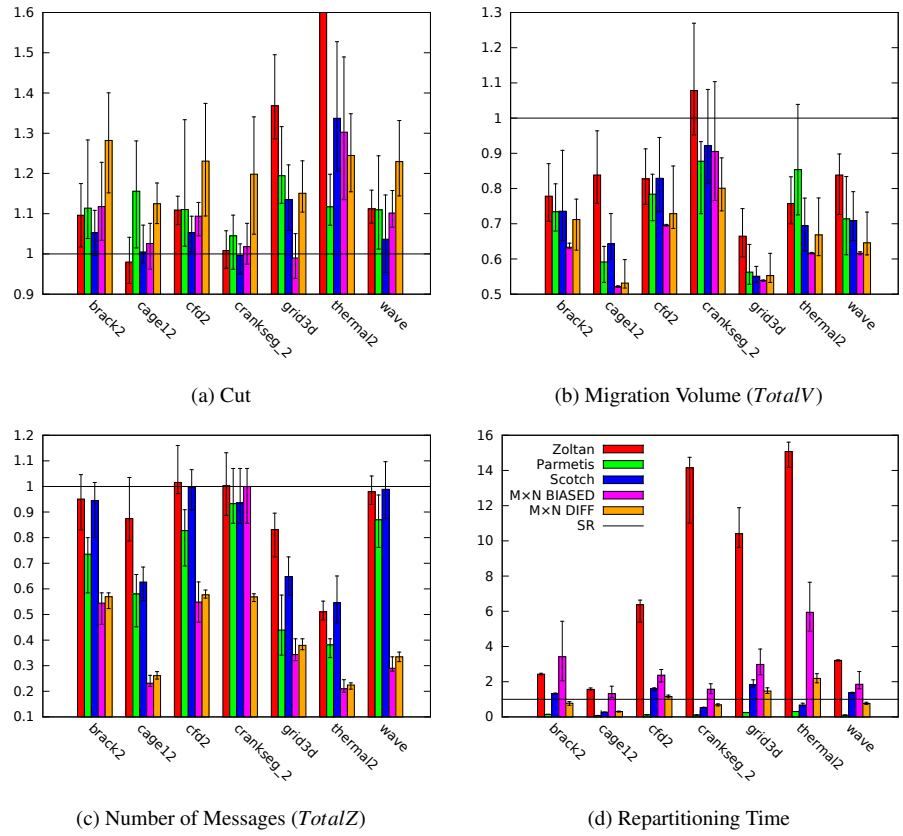
For each graph, one considers an initial balanced partition in $M = 8$ (with respect to an imbalance tolerance $\varepsilon = 1\%$). Vertex weights are heterogeneously increased such that the global graph load is increased of 50% with a maximum imbalance of 33% between parts. Then, we increase the number of processors proportionally to the load, that leads to $N = 12$. One repeats all experiments 10 times with different weight distributions. The figure 5 presents the results obtained for different metrics (cut, migration volume, number of messages, repartitioning time) relatively to the SR method. Error bars show the minimum and maximum values during the repeated experiments.

As expected, SR gives the lowest cut and the highest migration volume, for minimizing the cut is its main objective while the migration volume is only optimized after the partition is built. Zoltan does not work well for $M \times N$ repartitioning and gives high migration volume and sometimes very high cut. ParMetis offers some good migration volume. It has a cut slightly higher than other partitioning tools but it is very fast[5] and may not perform as many partition refinements as others. Scotch gives a good migration volume with a cut slightly higher than SR while going as fast. All these partitioners use as many messages as the SR approach.

The two MxN approaches give the best migration volume and the lowest number of messages. The MxN BIASED method gives a good migration with a cut similar to other partitioners, but is slightly slower than other tools because of the added complexity of the migration edges. On the contrary, the MxN DIFF method gives a higher cut, but is faster as it does not fully recompute a new partition. The preliminary results obtained for our approach show it is possible to optimize the migration phase (by minimizing $TotalV$ and $TotalZ$) while keeping the edge-cut quite low as a trade-off.

[4]The Scratch-Remap method used for experiments is implemented with Scotch.

[5]ParMetis doesn't work in sequential; so one runs our experiments with 2 processors.



(a) Cut

(b) Migration Volume ($TotalV$)

(c) Number of Messages ($TotalZ$)

(d) Repartitioning Time

Figure 5. Results obtained for a 8×12 repartitioning of several unbalanced graphs, relatively to the Scratch-Remap method.

6. Conclusion and Prospects

We have presented in this two algorithms for repartitioning unbalanced graph with a dynamic number of processors. Both algorithms provides very good migration in regard to volume and number of messages, at the cost of a slightly increased edge-cut. The diffusion scheme could be further improved with a global approach thus reducing the edge-cut overhead.

In future work, we plan to validate our approaches on more varied and larger real-life cases. To achieve this, our algorithm should run in parallel. This can be achieved for the biased repartitioning scheme by using a parallel partitioner that handle fixed vertices. The performance of the biased repartitioning method could also be improved by integrating the migration constraints in the partitioner heuristic instead of adding many edges and thus increasing the size of the graph. As concern the linear programing solver used in the diffusion-based repartitioning method, it can be extended to build migration matrices that optimize other metrics than $TotalV$ like $MaxV$, $TotalZ$, $MaxZ$ or a linear combination of them.

References

[1] 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.

[2] 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.

[3] 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.

[4] Saeed Iqbal and GrahamF. Carey. Performance of parallel computations with dynamic processor allocation. *Engineering with Computers*, 24:135–143, 2008.

[5] Message-Passing Interface Forum. *MPI-2.0: Extensions to the Message-Passing Interface*. MPI Forum, june 1997.

[6] Leonid Oliker and Rupak Biswas. Plum: parallel load balancing for adaptive unstructured meshes. *J. Parallel Distrib. Comput.*, 52:150–177, August 1998.

[7] 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.

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

[9] 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.

[10] 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.

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

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

[13] François Pellegrini. Scotch. `http://www.labri.fr/perso/pelegrin/scotch/`.

[14] Clément Vuchener and Aurélien Esnard. Dynamic Load-Balancing with Variable Number of Processors based on Graph Repartitioning. In *HIPC 2012*, Pune, India, 2012. 9 pages.

[15] GLPK (GNU Linear Programming Kit). `http://www.gnu.org/software/glpk/glpk.html`.

[16] Chao-Wei Ou and Sanjay Ranka. Parallel incremental graph partitioning using linear programming. In *Proceedings Supercomputing '94*, pages 458–467, 1994.

[17] Y. F. Hu, R. J. Blake, and D. R. Emerson. An optimal migration algorithm for dynamic load balancing. *Concurrency: Practice and Experience*, 10(6):467–483, 1998.

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

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