An improved recursive graph bipartitioning algorithm for well balanced domain decomposition

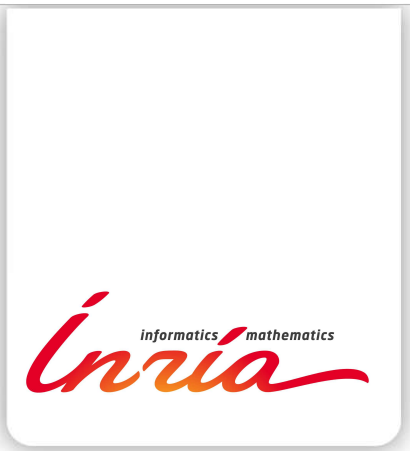
Astrid Casadei, Pierre Ramet, Jean Roman

An improved recursive graph bipartitioning algorithm for well balanced domain decomposition

Astrid Casadei, Pierre Ramet, Jean Roman

RESEARCH REPORT
N° 8582
August 2014
Project-Teams Hiepacs

ISSN 0249-6399 ISRN INRIA/RR--8582--FR+ENG



An improved recursive graph bipartitioning algorithm for well balanced domain decomposition

Astrid Casadei*†‡, Pierre Ramet*†‡, Jean Roman*†§

Project-Teams Hiepacs

Research Report n° 8582 — August 2014 — 26 pages

Abstract: In the context of hybrid sparse linear solvers based on domain decomposition and Schur complement approaches, getting a domain decomposition tool leading to a good balancing of both the internal node set size and the interface node set size for all the domains is a critical point for load balancing and efficiency issues in a parallel computation context. For this purpose, we revisit the original algorithm introduced by Lipton, Rose and Tarjan [13] in 1979 which performed the recursion for nested dissection in a particular manner. From this specific recursive strategy, we propose in this paper several variations of the existing algorithms in the multilevel Scotch partitioner that take into account these multiple criteria and we illustrate the improved results on a collection of graphs corresponding to finite element meshes used in numerical scientific applications.

Key-words: Graph partitioning, domain decomposition, nested dissection, parallel sparse hybrid solvers.

* Inria Bordeaux - Sud-Ouest, Talence, France
† Laboratoire Bordelais de Recherche en Informatique, Talence, France
‡ Bordeaux University, Talence, France
§ Institut Polytechnique de Bordeaux, Talence, France

RESEARCH CENTRE BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Un algorithme de bipartionnement récursif amélioré pour une décomposition de domaine équilibrée

Résumé : Dans le cadre des solveurs linéaires creux hybrides basés sur l'approche de décomposition de domaines avec complément de Schur, la mise au point d'un outil de décomposition de domaines permettant d'obtenir à la fois un bon équilibrage des intérieurs et des interfaces pour tous les domaines est un point critique pour l'équilibrage de la charge et l'efficacité dans un cadre de calculs parallèles. Dans cet objectif, nous réexaminons l'algorithme orignal introduit par Lipton, Rose et Tarjan [13] en 1979 et qui effectue la récursivité de la dissection emboîtée d'une manière particulière. En partant de cette stratégie récursive spécifique, nous proposons dans ce papier plusieurs variantes des algorithmes existants dans le partitioneur multiniveau Scotch qui prennent en considération ces multiples critères et nous illustrons l'amélioration de nos résultats sur une collection de graphes correspondants à des maillages d'élements finis utilisés dans des applications numériques scientifiques.

Mots-clés : partionnement de graphes, décomposition de domaines, dissection emboîtée, solveurs parallèles creux hybrides

1 Introduction and Motivation

Nested Dissection (ND) has been introduced by A. George in 1973 [4] and is a well-known and very popular heuristic for sparse matrix ordering to reduce both fill-in and operation count during Cholesky factorization. This method is based on graph partitioning and the basic idea is to build a "good separator" that is to say a "small size separator" S of the graph associated with the original matrix in order to split the remaining vertices in two parts P_0 and P_1 of "almost equal sizes". The vertices of the separator S are ordered with the largest indices, and then, the same method is applied recursively on the two subgraphs induced by P_0 and P_1. Good separators can be built for classes of graphs occurring in finite element problems based on meshes which are special cases of bounded density graphs [15] or more generally of overlap graphs [14]. In d-dimension, such n-node graphs have separators whose size grows as $\mathcal{O}(n^{(d-1)/d})$. In this paper, we focus on the cases $d = 2$ and $d = 3$ which correspond to the most interesting practical cases for numerical scientific applications. ND has been implemented by graph partitioners such as MeTiS[a] [11] or Scotch[b] [16].

Moreover, ND is based on a divide and conquer approach and is also very well suited to maximize the number of independent computation tasks for parallel implementations of direct solvers. Then, by using the block data structure induced by the partition of separators in the original graph, very efficient parallel block solvers have been designed and implemented according to supernodal or multifrontal approaches. To name a few, one can cite MUMPS[c] [2], PaStiX[d] [9] and SuperLU[e] [6].

Additionally, ND has the advantage of building a hierarchical domain decomposition, which is a valuable feature for hierarchical parallel sparse linear solvers using h-matrices. On the contrary, k-way partitioning, the most popular alternative to ND, does not allow to do that. This latter method intends to construct directly a domain decomposition of a graph in k sets of independent vertices [12].

However, if we examine precisely the complexity analysis for the estimation of asymptotic bounds for fill-in or operation count when using ND ordering [13], we can notice that the size of the halo of the separated subgraphs (set of external vertices adjacent to the subgraphs and previously ordered) play a crucial role in the asymptotic behavior achieved. The minimization of the halo is in fact never considered in the context of standard graph partitioning and therefore in sparse direct factorization studies.

In this paper, we focus on hybrid solvers combining direct and iterative methods and based on domain decomposition and Schur complement approaches. The goal is to provide robustness similar to sparse direct solvers, but memory usage more similar to preconditioned iterative solvers. Several sparse solvers like HIPS[f] [3], MaPHyS [1,5], PDSLIN [18] and ShyLU[g] [17] implement different versions of this hybridification principle.

In this context, the computational cost associated to each subdomain for which a sparse direct elimination based on ND ordering is carried out, as well as the computational cost of the

[a] `http://www.cs.umn.edu/~metis`
[b] `http://gforge.inria.fr/projects/scotch`
[c] `http://graal.ens-lyon.fr/MUMPS/`
[d] `http://pastix.gforge.inria.fr/`
[e] `http://crd-legacy.lbl.gov/~xiaoye/SuperLU/`
[f] `http://hips.gforge.inria.fr/`
[g] `http://trilinos.sandia.gov/packages/shylu/`

iterative part of the hybrid solver, critically depend on the halo size of the subdomains. Moreover, for generic hybrid solvers, a good tradeoff must be found between the number of subdomains, which globally influences the quality of the convergence in terms of number of iterations, and the size of the subdomains. However, if we must consider medium or large subdomain sizes for numerical issues, one can use a parallel sparse direct solver on each subdomain leading to the use of a second level of parallelism.

For this purpose, we revisit the original algorithm introduced by Lipton, Rose and Tarjan [13] in 1979 which performed the recursion for nested dissection in a different manner: at each level, we apply recursively the method to the subgraphs induced by $P_0 \cup S$ on one hand, and $P_1 \cup S$ on the other hand (see Figure 1 on the right). In these subgraphs, vertices already ordered (and belonging to previous separators) are the halo vertices. The partition of these subgraphs will be performed with three objectives: balancing of the two new parts P'_0 and P'_1, balancing of the halo vertices in these parts P'_0 and P'_1, and minimizing the size of the separator S'.

We implement this strategy in the Scotch partitioner. Scotch strategy is based on the multilevel method [7,10] which consists in three main steps: the (sub)graph is coarsened multiple times until it becomes small enough, then an algorithm called greedy graph growing is applied on the coarsest graph to find a good separator, and finally the graph is uncoarsened, projecting at each level the coarse separator on a finer graph and refining it using the Fiduccia-Mattheyses algorithm [8]. This paper studies variations of these three algorithmic steps in order to take into account the balancing criteria for both the internal and halo nodes, the goal being to achieve in the end a well balanced domain decomposition well suited for parallel hybrid solvers. However, as we consider a bi-partitioning method, the number of subdomains generated will be 2^k if we stop the recursion at some level k.

The paper is organized as follows. In Section II, we focus on the coarsening and the uncoarsening steps, and we present the modifications compared to standard multilevel partitioning strategy. Section III presents greedy graph growing approaches for the coarsest graph and several adaptations developed to get balanced halo and interior node sizes. Some experimental results illustrate the different studied strategies. In Section IV, we validate our work by comparing the achieved results with the ones of the native Scotch partitioner on a collection of large 3D problems coming from numerical simulations. Finally, we conclude and give some perspectives for future works in the last section.

2 Multilevel Framework

The Scotch default strategy consists in a multilevel method, which is one of the best ways to find good separators. This takes two sub-methods as parameters: an effective partitioning strategy, which is greedy graph growing algorithm (GG) by default in Scotch; and a method allowing to enhance an existing separator, here the Fiduccia-Mattheyses algorithm (FM). The idea is to coarsen the graph multiple times to simplify it, then to apply the effective partitioning strategy GG on the coarsest graph, and finally to project the separator back on finest graphs. At each level, the projection is refined with the FM algorithm. The first part of this section will describe our modification of the multiflevel framework, then we will detail FM algorithms and its changes.

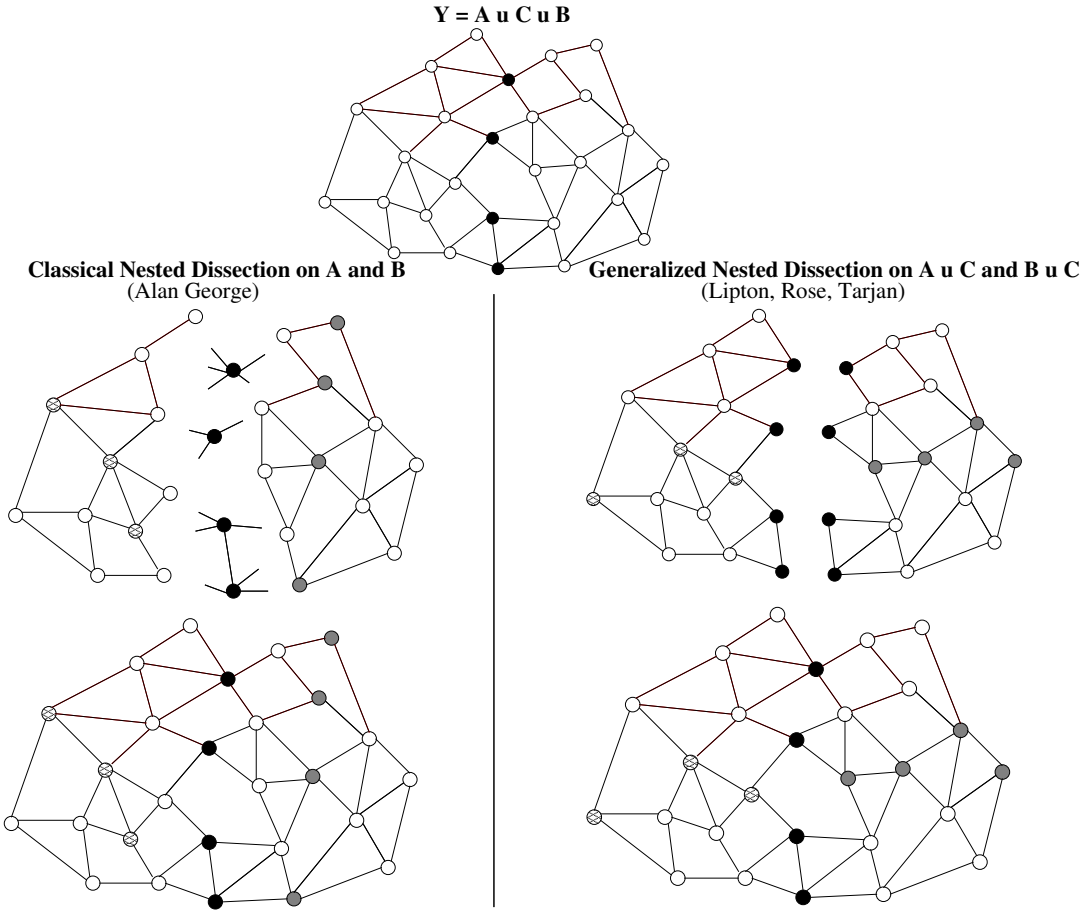


Figure 1: *On the left:* classical recursion which is performed on P_0 and P_1. Objectives are to balance the sizes of the subgraphs and to minimize the separator size. However, halo sizes, represented by the nodes in black and grey, can be unbalanced: they are respectively 4, 5, 6 and 8. *On the right:* recursion is performed here on $P_0 \cup S$ and $P_1 \cup S$ and the halo vertices are balanced among the parts, leading to interface sizes equal to 5, 5, 6 and 6.

2.1 Modified multilevel framework

The multilevel framework works as follow. At each step of the coarsening stage, a matching of the vertices is performed, and the matched vertices are merged, summing their weight, to form the weight of the new vertices. This process is repeated until the graph obtained is small enough. Then, GG is applied on the (weighted) coarsest graph, making a first guess of the final separator. At each stage of the uncoarsening, two vertices that were matched at a finer level are assigned to the same part than their coarse equivalent. This way, if the global balance was achieved in the coarser graph, it is still in the finer; yet, the uncoarsening may lead to a thick locally non-optimal separator, requiring to use a refinement algorithm. To reduce the research domain of the algorithm, Scotch builds a band graph of width 3 around the uncoarsened separator and runs FM on it. Note that to have a good separator, this refinement is applied at each step of the uncoarsening, not only on the finest graph.

Our aim to balance the halo vertices requires to modify slightly the multilevel framework. Indeed, some halo vertices may be matched with non-halo vertices, and the sum of their weights would not mean anything. Thus, vertices have now two weights: a non-halo and a halo weight. When two vertices are matched, the two non-halo weights are added together, and the same is done for their halo weights. If the initial graph is unweighted, the non-halo weight of a vertex is one if it is out of the halo, zero otherwise; the halo weight of a vertex is one if the vertex is in the halo, zero otherwise. In the context of these two different kinds of weight, we redefine a halo vertex as *a vertex which has a non-zero halo weight*. Since the matching procedure treats halo and non-halo vertices the same way, we expect that the ratio of halo vertices is almost the same in the finest and the coarsest graphs.

In the following, if C is a set of vertices, we denote by $\overline{C}$ its subset of halo vertices. $|C|$ is the sum of the non-halo weights of vertices in C and by $|\overline{C}|$ the sum of halo weights.

2.2 Original Fiduccia-Mattheyses algorithm

FM method is an algorithm implemented in Scotch to refine an existing separator. Note that two mains versions exists; in the following, we talk about the one which optimize a **vertex** separator. FM is described on Algo. 1.

FM is based on a local search around the initial separator. A *move* of the search consists in picking a vertex from the current separator and putting it in one of the two parts. To keep a correct separator, the neighbours of the vertex in the other part also need to enter it (see Figure 2). FM algorithm makes several passes (set of consecutive moves) and keeps going on while the maximum number of passes is not reached and the last pass brings improvement; the next pass begins from the best separator ever found (and found in the last pass). Moreover, even passes have a slight preference for moving vertices in part 0, while odd passes favor part 1 instead.

Function *getSep* at line 11 of Algo. 1 chooses the best possible couple (s, i) to move (where $s \in S$ and $i \in \{0, 1\}$ is some part). It has three objectives: getting a reasonable imbalance $\Delta = |P_0| - |P_1|$, minimizing the separator S and moving a vertex to the preferred $pref$ part [h]. More specifically, *getSep* ensure that once the move is done, the new imbalance do not exceed $\max(|\Delta|, \Delta_{th})$, where Δ is the current signed imbalance and Δ_{th} a fixed absolute imbalance given

[h] Note that FM has some exceptions for isolated vertices, *i.e.* separator vertices which are adjacent to only one of the two parts. In pratice, we have special treatments that induces several difficulties that will not be described in this paper.

Algorithm 1: Original FM

Input: graph: $G = (V, E)$, number of passes: $passnbr$, number of hill-climbing moves by pass: $movenbr$, maximum acceptable imbalance: Δ_{th}, initial partition: (P_0, S, P_1) with $S \neq \emptyset$

Output: partition (P_0, S, P_1) of V such as S is a (small) separator and $|P_0| \approx |P_1|$

$(P_0^*, S^*, P_1^*) \leftarrow (P_0, S, P_1)$;
$passnum \leftarrow 0$;
repeat
  $(P_0, S, P_1) \leftarrow (P_0^*, S^*, P_1^*)$;
  $\Delta \leftarrow |P_0| - |P_1|$;
  $tabu \leftarrow \emptyset$;
  $movenum \leftarrow 0, enhanced \leftarrow false$;
  $pref \leftarrow mod(passnum, 2)$;
  while $movenum < movenbr$ **do**
    $(f, v, i) \leftarrow getSep(S \setminus tabu, \max(\Delta_{th}, |\Delta|), pref)$;
    if $\neg f$ **then** `/* No movable vertex */`
      break;
    `/* Move` v `from separator to part` i `*/`;
    $R \leftarrow \{w | (v, w) \in E \text{ and } w \in P_{\neg i}\}$;
    $S \leftarrow S \setminus \{v\} \cup R$;
    $P_i \leftarrow P_i \cup \{v\}$, $P_{\neg i} \leftarrow P_{\neg i} \setminus R$;
    $\Delta \leftarrow |P_0| - |P_1|$;
    $movenum$++;
    $tabu \leftarrow tabu \cup \{v\}$;
    if (P_0, S, P_1) *is better than* (P_0^*, S^*, P_1^*) **then**
      $(P_0^*, S^*, P_1^*) \leftarrow (P_0, S, P_1)$;
      $movenum \leftarrow 0, enhanced \leftarrow true$;
  $passnum$++;
until $\neg enhanced$ *or* $(passnum = passnbr)$;
return (P_0^*, S^*, P_1^*);

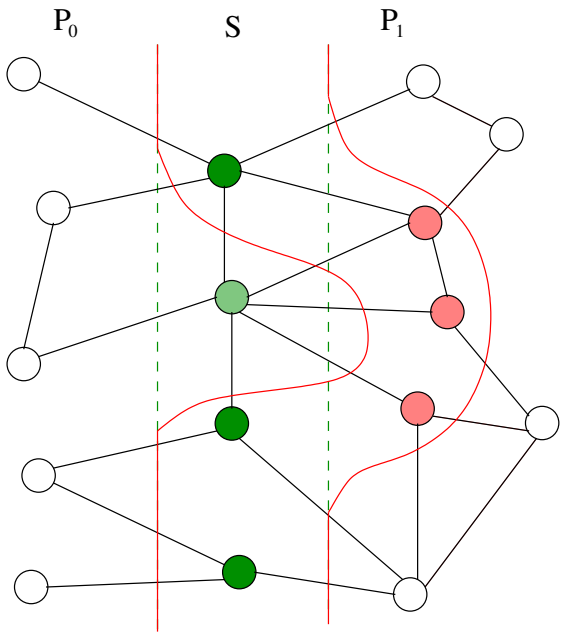


Figure 2: A move of FM algorithm. At first, the separator S contains all green vertices. The light green vertex is chosen to be put in part P_0. The neighbours in P_1 of the light green vertex (set R in Algo. 1, vertices in pink here) are added to S to keep a valid separator.

by user. This means that if current imbalance is outside the scope of Δ_{th}, it is not be degraded, and if it is within, it remains within. If there is no possible move respecting this rule, *getSep* returns $f = false$ and the pass ends. Otherwise, it takes, among valid choices, one that leads to the smallest possible new separator (which can be larger than the current one). If there are still several possibilities, *getSep* eventually selects a move to the preferred part of the pass.

After each move, Algo. 1 checks whether the current separator is the best separator ever found. The choice is made as follow (line 22):

- if a partition in which $|\Delta| \leq \Delta_{th}$ has never been found, the partition whose $|\Delta|$ is the smallest is kept.
- if a partition satisfying $|\Delta| \leq \Delta_{th}$ has already been found, only partitions satisfying this condition are kept and the one which has the smallest separator is kept. In case of equality, the one with the smallest $|\Delta|$ is selected.

In order to prevent the local search to make the same choices several times, a *Tabu* search is implemented. A set of *tabu* vertices is maintained and reset at each new pass. Whenever a vertex is chosen, it is put in the *tabu* set and is not be allowed to move again until next pass. This way, during a pass, a vertex may enter the separator, be chosen to leave it, and re-enter by the move of some of its neighbours, but then it remains in the separator.

A pass stops when either there is no possible move remaining, or the last *movenbr* moves did not bring any improvement. Giving the possibility of continuing a pass for *movenbr* moves after the last improvement may allow to get out of a local minimum.

To finish, a particular case is to be noticed: sometimes, separator S may contain vertices which are linked to only one (or zero) of the two parts. These vertices are called *isolated vertices* and have no reason to remain in the separator. They can simply be retrieved from S and be put in the part to which they are adjacent (or in any part if they are not adjacent to any part). Removing these useless vertices is a priority to SCOTCH: while *getSep* iterates on choices, if an isolated vertex is found, it is picked immediatly, even if it is *tabu* or if the move makes the imbalance get out from the scope of Δ_{th}.

2.3 Modifications to Fiduccia-Mattheyses algorithm

Our modified FM algorithm is presented in Algorithm 2. The choice of a vertex v to move from the separator to a part i is revisited. If the current partition does not have an absolute halo imbalance $|\overline{\Delta}|$ below the threshold $\overline{\Delta}_{th} \geq 0$ (chosen by user), then the function *getHalo* is called line 12. This function tries to fix the halo imbalance. It uses the sign of $\overline{\Delta} = |\overline{P_0}| - |\overline{P_1}|$ to know the part where a vertex must be moved: a move to part 0 increases $\overline{\Delta}$, a move to part 1 decreases it. Then, it picks a vertex whose move to part i minimizes the new halo imbalance. If there is no such move that improves $\overline{\Delta}$ strictly, then the function fails. Here, or if the partition had already a reasonable halo imbalance, the function *getSep* is called instead (line 14), which works as described in the unmodified version. Note that as *getSep*, *getHalo* has a special case to take isolated vertices in priority in all cases.

We conclude this section by giving our strategy to choose the *better* partition as indicated at line 24 of Algorithm 2. We proceed as follows:

- if we never found a partition in which $|\Delta| \leq \Delta_{th}$, we keep the partition whose $|\Delta|$ is the smallest.
- if we already found a partition satisfying $|\Delta| \leq \Delta_{th}$, but never a partition satisfying both $|\Delta| \leq \Delta_{th}$ and $|\overline{\Delta}| \leq \overline{\Delta}_{th}$, then we keep a partition with $|\Delta| \leq \Delta_{th}$ and whose $|\overline{\Delta}|$ is the smallest.
- if we already found a partition satisfying both $|\Delta| \leq \Delta_{th}$ and $|\overline{\Delta}| \leq \overline{\Delta}_{th}$, we keep only partitions satisfying these two conditions; among them, we choose the one which has the smallest separator. In case of equality, we pick the one with the smallest $|\overline{\Delta}|$, and if there is still a tie, the one with the smallest $|\Delta|$.

3 Graph Partitioning Algorithms

We also need to adapt the greedy graph growing (GG) algorithm in order to compute a partitioning which takes halo weight into account. The next subsection will present the GG algorithm and a straightforward adaptation. Some unsatisfactory results lead us to consider two other approaches that are presented in the last two subsections and named double GG and halo-first GG respectively. In the next section of the paper, another set of results will be presented to highlight the differences between these two new approaches when the number of domains increases.

Table 1 presents all the testing matrices, giving their size and their number of non-zero entries. The *id* number will be used to identify matrices in the following. For each matrix A,

Algorithm 2: modified FM

Input: graph: $G = (V, E)$, number of passes: $passnbr$, number of hill-climbing moves by pass: $movenbr$, maximum acceptable imbalance: Δ_{th} and $\overline{\Delta}_{th}$, initial partition: (P_0, S, P_1) with $S \neq \emptyset$

Output: partition (P_0, S, P_1) of V such as S is a (small) separator and $|P_0| \approx |P_1|$

$(P_0^*, S^*, P_1^*) \leftarrow (P_0, S, P_1)$;
$passnum \leftarrow 0$;
repeat
  $(P_0, S, P_1) \leftarrow (P_0^*, S^*, P_1^*)$;
  $\Delta \leftarrow |P_0| - |P_1|$, $\overline{\Delta} \leftarrow |\overline{P_0}| - |\overline{P_1}|$;
  $tabu \leftarrow \emptyset$;
  $movenum \leftarrow 0, enhanced \leftarrow false$;
  $pref \leftarrow mod(passnum, 2)$;
  while $movenum < movenbr$ **do**
    $f \leftarrow false$;
    if $|\overline{\Delta}| > \overline{\Delta}_{th}$ **then**
      $(f, v, i) \leftarrow getHalo(S \setminus tabu, \overline{\Delta})$;
    if $\neg f$ **then**
      $(f, v, i) \leftarrow getSep(S \setminus tabu, \max(\Delta_{th}, |\Delta|), pref)$;
    if $\neg f$ **then** `/* No movable vertex */`
      break;
    `/* Move` v `from separator to part` i `*/`;
    $R \leftarrow \{w | (v, w) \in E \text{ and } w \in P_{\neg i}\}$;
    $S \leftarrow S \setminus \{v\} \cup R$;
    $P_i \leftarrow P_i \cup \{v\}$, $P_{\neg i} \leftarrow P_{\neg i} \setminus R$;
    $\Delta \leftarrow |P_0| - |P_1|$, $\overline{\Delta} \leftarrow |\overline{P_0}| - |\overline{P_1}|$;
    $movenum$++;
    $tabu \leftarrow tabu \cup \{v\}$;
    if (P_0, S, P_1) *is better than* (P_0^*, S^*, P_1^*) **then**
      $(P_0^*, S^*, P_1^*) \leftarrow (P_0, S, P_1)$;
      $movenum \leftarrow 0, enhanced \leftarrow true$;
  $passnum$++;
until $\neg enhanced$ *or* $(passnum = passnbr)$;
return (P_0^*, S^*, P_1^*);

the symetric graph of $A + A^t$ is used. Matrices 1-20 come from the University of Florida Sparse Matrix Collection [i], and the last set of ten matrices comes from our industrial collaborations or partners.

id	Matrix	n	nnz
1	Dubcova3	146689	3489960
2	wave	156317	2118662
3	dj_pretok	182730	1512512
4	turon_m	189924	1557062
5	stomach	213360	3236576
6	BenElechi1	245874	12904622
7	torso3	259156	4372658
8	mario002	389874	1867114
9	helm2d03	392257	2349678
10	kim2	456976	10905268
11	mc2depi	525825	3148800
12	tmt_unsym	917825	3666976
13	t2em	921632	3673536
14	ldoor	952203	45570272
15	bone010	986703	70679622
16	ecology1	1000000	39996000
17	dielFilterV3real	1102824	88203196
18	thermal2	1228045	7352268
19	StocF-1465	1465137	19540252
20	Hook_1498	1498023	59419422
21	**NICE-25**	140662	5547944
22	**MHD**	485597	23747544
23	**Inline**	503712	36312630
24	**ultrasound**	531441	32544720
25	**Audikw_1**	943695	76708152
26	**Haltere**	1288825	18375900
27	**NICE-5**	2233031	175971592
28	**Almond**	6994683	102965400
29	**NICE-7**	8159758	661012794
30	**10millions**	10423737	157298268

Table 1: Set of test matrices. 1-20 come from the University of Florida Sparse Matrix Collection. 21-30 come from industrial collaborations or partners.

Algorithms will be judged upon two criteria. For each domain, the sizes of the interior and of the halo are measured. Then, we compute the difference between the maximum and the minium for both, providing two metrics: *interface imbalance*, and *interior imbalance*. For example, on the bottom-right graph of Figure 1, interior sizes are all equal to 4, thus the interior imbalance is null, and halo sizes are 5, 5, 6, and 6 respectively, giving a halo imbalance equal to 1.

Note that the graph partitioning technique used for ND is designed for reordering purpose. In this context, the main objective of Scotch software is to minimize the size of the separator while keeping a local imbalance for interior sizes that does not exceed a fixed percentage, named

[i] `www.cise.ufl.edu/research/sparse/matrices`

bal. The recursion is performed until a fixed number of vertices is reached. Thus, the branches of the decomposition tree may have different heights. On the contrary, we focus in this paper on decomposition domain : so we want to choose the number of domains and thus the number of levels in the recursion. Furthermore, a default value for local constraint of interior imbalance ($bal = 10\%$) accumulates through levels of the recursion: at level i, imbalance between minimum and maximum subgraph sizes may reach roughly $bal \times i$ percents. This is too loose for our purpose. We cannot decrease *bal* too much because the constraint would be too tight for having a chance to minimize the separator. Thus, to achieve a good balancing, we use a constraint that depends on the level: on the higher levels, subgraphs are big, so we can use a tighter constraint while giving the possibility to optimize the separator size; on bottom level, subgraphs are small and we use a looser constraint. More precisely, if p levels are requested, level i will try to get a local imbalance of at most $max(\frac{bal}{2^{p-i+1}}, minbal)$, where *minbal* is a threshold ensuring that the constraint does not become too small.

In the following, all tests are done with a fixed number of levels of recursion. The column *GG* (standing for Greedy Graph Growing) in the results refers to the unmodified Scotch strategy, with $bal = 10\%$. The column $GG^\star$ and the other columns use the level-dependent constraint described above with $bal = 10\%$ and a threshold of 1%. $GG^\star$ thus refers to a modified Scotch strategy with default graph partitioning algorithms but using the level-dependent *bal* constraint.

3.1 Greedy Graph Growing

The algorithm implemented by the Scotch software to find a good separator in a graph $G = (V, E)$ at the bottom of the multi-level technique is the greedy graph growing method. The idea is to pick a random seed vertex in the graph, and to make a part grow from this seed, until it reaches the half of the graph size. It is described in Algorithm 3. At line 3, the seed w is chosen. Singleton $\{w\}$ is the initial separator S between the parts P_1, empty, and P_0, containing all other vertices. Then, at each step, a vertex v from current separator S is chosen (l. 8), and passed from the separator S to the growing part P_1 (l. 10 and 12). The choice is oriented by the minimization of the current separator. Additionally, the set N of all neighbours of v in P_0 are retrieved from P_0 (l. 11) and added to S (l. 10), so that S remain a separator for the parts. The process is repeated until both parts have almost the same size.

The result of this algorithm is very dependent on the random seed chosen at the beginning. Thus, Scotch tries several passes with different seeds (l. 2), and it eventually selects the best partition (P_0, S, P_1) found in all passes (l. 13).

In a first attempt to adapt this algorithm to our purpose of balancing the halo, we made the following changes. First, the choice of the vertex to move from S to P_1 was now oriented by the halo balance. More specifically, if P_1 had not as much halo vertices as P_0 (relatively to the respective size of the parts), then $choiceOfVertexInSeparator$ preferably chooses a vertex v inside the halo. Second, note that it is needed to have both the halo and the non-halo vertices in separator for this strategy to work well. We thus chose the random seed inside halo, since halo vertices are often close to each other.

We tested this adapted algorithm. Unfortunately, if often fail to improve the default partitioning strategy. A typical situation that occurs is represented on Figure 3. On a connected graph, the greedy graph growing algorithm ensures the connectedness of the growing P_1, but not of P_0. On the original algorithm, it has no consequences in most cases. But, our modified halo-cared algorithm is almost always able to trade a better halo balance against a new connected component in P_0. For instance here, P_0 is in blue and growing P_1 in purple. The halo vertices

Algorithm 3: GreedyGraphGrowing

Input: graph $G = (V, E)$, number of passes *passnbr*
Output: partition (P_0, S, P_1) of V such as S is a (small) separator and $|P_0| \approx |P_1|$

$(P_0^*, S^*, P_1^*) \leftarrow (\emptyset, V, \emptyset)$;
for $p = 1$ **to** *passnbr* **do**
  $w \leftarrow RandomSeed(V)$;
  $P_0 \leftarrow V \setminus \{w\}$;
  $P_1 \leftarrow \emptyset$;
  $S \leftarrow \{w\}$;
  while $|P_0|$ *and* $|P_1|$ *are not balanced* **do**
    $v \leftarrow getVertex(S)$;
    $N \leftarrow \{j | (v, j) \in E \text{ and } j \in P_0\}$; `/* neighbours of` v `in` P_0 `*/`
    $S \leftarrow S \setminus \{v\} \cup N$;
    $P_0 \leftarrow P_0 \setminus N$;
    $P_1 \leftarrow P_1 \cup \{v\}$;
  if (P_0, S, P_1) *is better than* (P_0^*, S^*, P_1^*) **then**
    $(P_0^*, S^*, P_1^*) \leftarrow (P_0, S, P_1)$;
return (P_0^*, S^*, P_1^*);

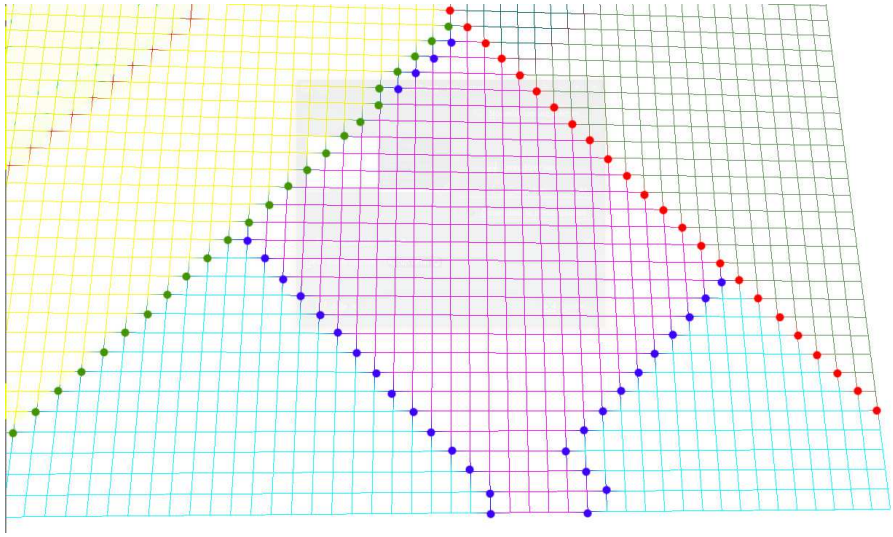


Figure 3: Illustration of a corner case with the modified greedy-graph-growing algorithm. The blue partition is eventually split in three disconnected parts.

are in green and red. The new separator is in dark blue. One can notice that the four green halo vertices at the top left belongs to P_0 (which has thus three connected components). As it was growing, P_1 reached the border of the graph while it had enough halo vertices; this explain why these four vertices are not merged with P_1. This kind of situation can happen very often.

3.2 Double Greedy Graph Growing with Halo Care

The previous algorithm managed a good halo balance in general, but often at the price of a disconnected part P_0. We thus decided to use two initial seeds, one for each part, and to make both parts grow simultaneously. However, this new strategy can lead to blocking. Indeed, when growing, one part may block the progression of the other: this happens when $V \setminus (P_0 \cup P_1)$ is not empty but has no vertex reachable from some part P_i. To avoid this problem, we need to delay, as much as possible, the moment when parts meet each other.

Algorithm 4 describes the new method. Line 3 picks the two seeds w_i and $w_{\neg i}$ in the halo. They are also chosen as far as possible from each other, so that parts meet as late as possible. Both parts, initially empty, are grown from their respective seed vertex. At each step, we choose the smallest part i (l. 11-14), and a vertex v from its boundary S_i (l. 17). v is chosen according to the halo balance situation; if part i has less halo vertices than part $\neg i$, a halo vertex is taken if possible, and if it has more halo vertices, a non-halo vertex is picked preferably. If several choices of vertices remain, then the vertex which is the nearest from w_i and the farthest from $w_{\neg i}$ (*i.e.* the vertex v with the smallest value $dist(v, w_i) - dist(v, w_{\neg i})$) is taken. This is still in the purpose of making the parts meet as late as possible. Like the single-seed greedy graph growing presented before, v is then added to P_i (l. 19), retrieved from S_i, and S_i is updated to remain the boundary of P_i (l. 18). The process is carried on until all vertices are in one of the two parts, meaning $V \setminus (P_0 \cup P_1) = \emptyset$ (end of while loop l. 10), or we are blocked, namely $S_i = \emptyset$ (l. 15).

In the latter case, a solution would be to put all remaining vertices of $V \setminus (P_0 \cup P_1)$ in the part to which they are adjacent. If few vertices remain, this is actually the solution we take lines 21-23. Otherwise, we empty P_0 and P_1 (l. 21) and retry to make grow the parts from w_0 and w_1, using some additional information to avoid to get blocked again. More specifically, we define a set of *control points* of each part i, containing only their respective seed at the beginning. When blocked, we add a new control point to the part i which could not grow. This control point is defined by the vertex of P_i which is the nearest from the untaken vertices. Then, parts are made grown again. When we choose a vertex to add to part i (l. 17), the first criterion is still the halo situation, and the second criterion is now the vertex v with the smallest value $\min_j\{dist(v, ctrlpts_i[j])\} - \min_j\{dist(v, ctrlpts_{\neg i}[j])\}$. In other words, part i will be *attracted* by its own control points, and *repulsed* by the control points of $\neg i$. (Note that this rule is in fact a generalization of the previous one).

The strategy described before is repeated until we either succeed to construct a partition (P_0, P_1) of V from (w_0, w_1), or we reach *triesnb* tries, meaning we failed. If we succeed, we can construct a separator S by applying a minimum vertex cover algorithm on the edges on the frontier of P_0 and P_1 (l. 27).

Like in the previous algorithm, several passes are made with different couples of seed vertices. We eventually select the best partition found among the successful passes on line 29.

To conclude on this algorithm, let us develop a little more on the seeds choices. We mentioned we took vertices *in the halo* and *as far as possible*, so there are two meanings. First, we can take two vertices of the halo that are as far as possible from each other *in the whole graph*. And second, we can *extract* the graph of the halo (the following section will explain how to do that) and pick two vertices that are the farthest from each other in this halo graph. Figure 4 and 5

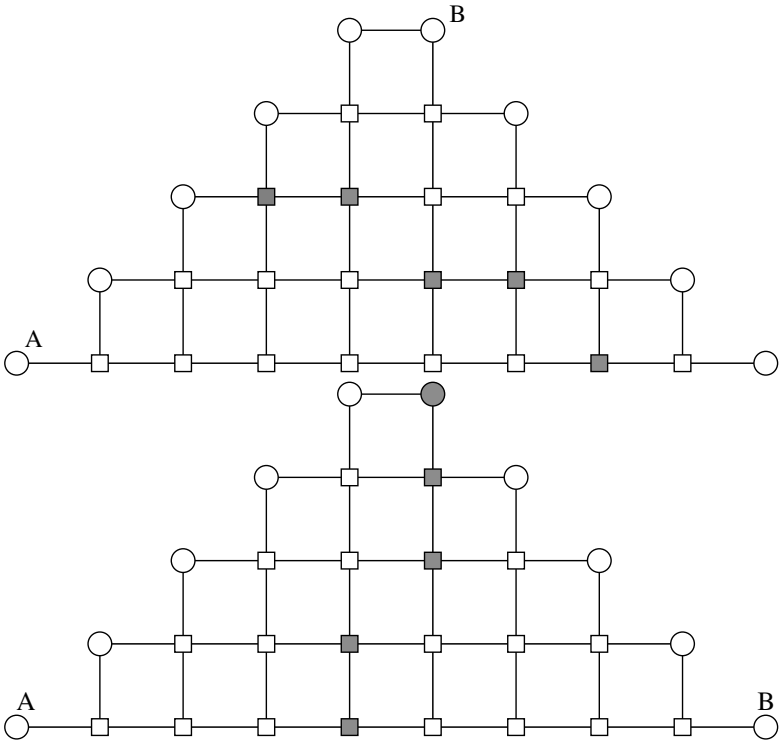


Figure 4: An example showing taking seed vertices A and B "as far as possible along halo" (below) can be better than taking seed vertices in halo as far as possible in the whole graph (above). Halo vertices are denoted by circles.

show that depending on the graph, both solutions can be better than the other. On the top case in Figure 4, vertices A and B are at distance 9 in the whole graph, which is actually the maximum. The rightmost vertex is also at distance 9 of A and has no reason to be preferred. Yet, the separator found with these seeds does not achieve a good halo balance (3 against 7). On the bottom one, the graph of halo has been built and the leftmost and rightmost vertices have been found to be the farthest ones. This time, the separator balances well halo vertices. On the second example in Figure 5, both cases achieve a good halo balance, but the sizes of the separators are significantly different: the *as far along halo* strategy gives a separator of size 9, which is far more than the other, of size only 5. Thus, both strategies have benefits, and when we try several passes, we try at least each of them.

We tested double greedy graph growing on 4 levels of recursion (i.e. 16 domains). The results are presented in Table 2. The columns *GG* give the interface and interior imbalance of unmodified Scotch with $bal = 10\%$. Other columns only give a percentage relatively to the corresponding *GG* column. For example, for the matrix `ecology1` (16), double greedy graph growing (denoted by the *DG* column) achieves a halo imbalance $62,9\%$ better than unmodified *GG*, that is a halo imbalance of $(1 - 0.629) \times 564 \simeq 209$. The column $GG^\star$ refers to a modified Scotch with a balance depending on the level (see introduction of this section). For each criterion, we highlighted in bold the best result.

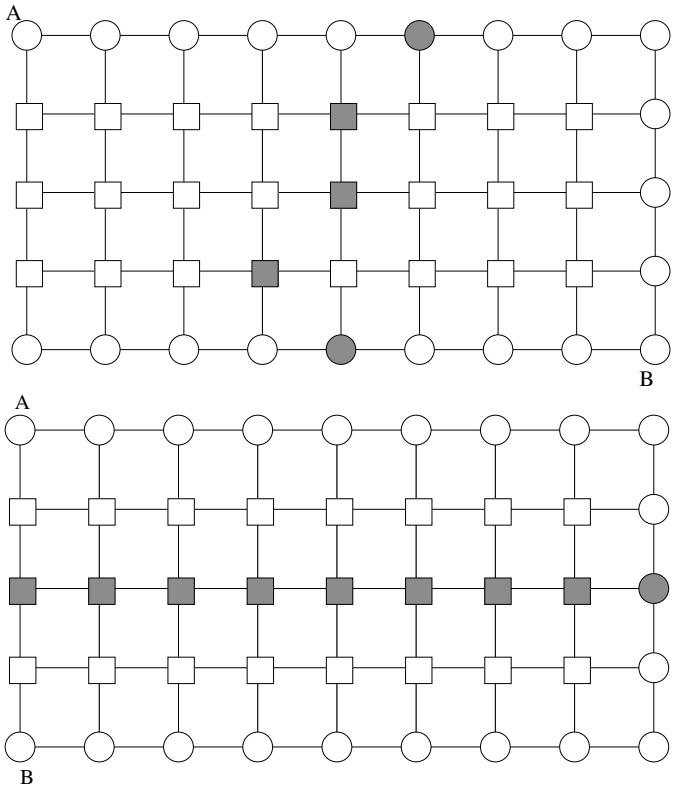


Figure 5: An example showing taking seed vertices A and B in halo as far as possible int the whole graph (above) can be better than taking seed vertices as far as possible along the halo (below). Halo vertices are denoted by circles.

Algorithm 4: DoubleGreedyGraphGrowing

Input: graph $G = (V, E)$, number of passes $passnbr$
Output: partition (P_0, S, P_1) of V such as S is a (small) separator, $|P_0| \approx |P_1|$ and $|\overline{P_0}| \approx |\overline{P_1}|$

```
1  (P_0^*, S^*, P_1^*) ← (∅, V, ∅);
2  for p = 1 to passnbr do
3      (w_0, w_1) ← RandomSeeds(V) ;
4      ctrlpts_0 ← {w_0}, ctrlpts_1 ← {w_1};
5      success ← false;
6      for q = 1 to triesnb do
7          P_0, P_1 ← ∅;
8          S_0 ← {w_0}, S_1 ← {w_1};
9          ctrldist ← computeDistances(G, ctrlpts_0, ctrlpts_1);
10         while V \ (P_0 ∪ P_1) ≠ ∅ do
11             if |P_0| < |P_1| then
12                 i ← 0;
13             else
14                 i ← 1;
15             if S_i = ∅ then
16                 break;
17             v ← getVertex(S_i, ctrldist) ;
18             S_i ← S_i \ {v} ∪ {j|(v, j) ∈ E and j ∈ V \ (P_0 ∪ P_1)} ;
19             P_i ← P_i ∪ {v} ;
20         if |V \ (P_0 ∪ P_1)| ≤ 0, 1|V| then
21             P_¬i ← V ∪ P_i;
22             success ← true;
23             break;
24         else
25             ctrlpts_i ← ctrlpts_i ∪ findNewControlPoint(G, P_i, P_¬i);
26     if success then
27         S ← MinVertexCover(E ∩ (P_0 × P_1));
28         P_0 ← P_0 \ S, P_1 ← P_1 \ S;
29         if (P_0, S, P_1) is better than (P_0^*, S^*, P_1^*) then
30             (P_0^*, S^*, P_1^*) ← (P_0, S, P_1);
31 return (P_0^*, S^*, P_1^*);
```

	interface imbalance			interior imbalance		
id	*GG*	% *GG**	% *DG*	*GG*	% *GG**	% *DG*
1	297	7,4	**-19,2**	1311	**-69,1**	-23,3
2	1112	1,6	**-40,0**	4678	**-66,5**	-78,9
3	522	-11,3	**-39,7**	1635	**-13,9**	-13,0
4	244	-9,0	**-11,5**	1568	-23,7	**-31,1**
5	475	**-17,9**	-3,8	4605	**-85,6**	-62,3
6	869	-21,3	**-41,3**	4107	**-78,5**	-49,6
7	**905**	49,4	26,4	4942	**-69,1**	-63,7
8	261	0,0	**-46,7**	420	**0,0**	79,8
9	365	-3,8	**-44,9**	8128	**-82,8**	-65,2
10	1002	14,0	**-32,1**	4852	**-73,2**	-44,6
11	509	-3,7	**-44,4**	2831	**-84,1**	27,5
12	569	-25,0	**-59,1**	22416	**-79,5**	-67,5
13	532	-11,8	**-58,5**	12049	**-69,0**	-49,5
14	756	-23,1	**-46,3**	17458	-61,9	**-66,6**
15	6678	6,6	**-29,8**	35466	**-73,9**	-68,4
16	564	-11,9	**-62,9**	15092	**-65,5**	-58,0
17	2130	18,6	**-50,4**	32202	**-72,9**	-67,7
18	335	17,0	**-19,7**	28057	**-70,1**	-64,9
19	2604	-16,4	**-42,0**	25853	**-66,5**	-59,5
20	9990	-14,2	**-25,8**	73635	**-80,7**	-16,5
21	927	0,6	**-40,9**	2377	-58,6	**-59,5**
22	3468	5,0	**-78,5**	**3336**	4,3	18,2
23	1869	7,1	**-3,2**	14424	**-73,3**	-64,4
24	2460	-9,1	**-73,4**	8940	-44,1	**-60,6**
25	6837	-11,6	**-69,7**	26877	-63,0	**-68,9**
26	780	-15,9	**-53,8**	24987	-58,3	**-65,2**
27	6168	-27,5	**-68,4**	67721	-68,7	**-76,2**
28	4344	-15,6	**-41,4**	240729	-76,9	**-77,1**
29	11539	8,6	**-51,3**	244959	-73,9	**-77,2**
30	9936	-6,9	**-52,0**	286992	**-72,6**	-8,5

Table 2: Results with double greedy graph growing compared to Scotch greedy graph growing

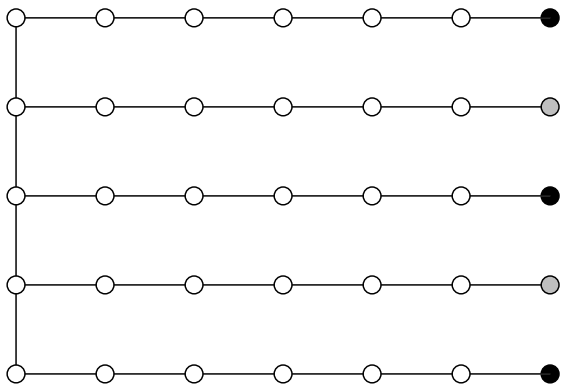


Figure 6: In this example, the halo (vertices on the right) is totally disconnected, and require to explore the far-away neighbourhood to reconnect it. Though, ignoring the position of halo vertices to build a separator could lead to a bad separator, if for instance we take the black halo vertices in one part and the gray ones in the other.

We can see that interface balancing of DG is much better than unmodified GG and $GG^\star$ in all but two matrices. Gain can be up to $78,5\%$ on matrix `MHD` (22), with an average of 40% on all matrices. DG also achieves a better interior balancing in general compared to GG, on all but two matrices; the average gain for this column is 45%. It is also better than $GG^\star$ on one third of the test cases, which is rather honorable since it has one more criterion to optimize. Moreover, we can see that on all but one industrial matrices (which are of particular interest for us), gains are very good on both criteria.

3.3 Halo-first Greedy Graph Growing

In the previous section, we have studied an algorithm that constructs a separator for the parts and for the halo at once. This gives the priority on minimizing the separator, while trying to balance the halo when possible. In this section, we review another approach, which consists in finding a halo separator first. Once this is done, we construct a separator for the whole graph, making the parts grow from the parts induced by this halo separator.

Before splitting the graph of the halo, we first have to build the graph. We could take the graph $(V_h = \overline{V}, E_h = E \cap (V_h \times V_h))$, defined by the restriction of the whole graph to the halo vertices and the edges connecting them. Nevertheless, this graph may not be connected, even if the whole graph is. In the worst case, it can be totally disconnected, and considering the far-away neighbourhood may not be enough to reconnect it. Though, this is important to take graph connections into account, because choosing which of the halo vertices will be in each part at random would often lead to a very poor separator of the whole graph (See Figure 6).

To deal with this issue, we use the following algorithm to build a graph containing all relevant informations about halo. A partition of the halo vertices is maintained. At the beginning, each halo vertex is in a different set of the partition, and a set V_h' is initialized with all halo vertices. Then, we make simultaneous breadth-first searches from all the sets of the partition. When two search bubbles corresponding to different sets meet, this means a shortest-path between any two sets of the partition has been found. All vertices of this path are added to V_h'. The sets which have met are merged, and the breadth-first-search is carried on. The process stops when either

all sets of the partition have merged - meaning the graph is connected -, or all breadth-first searches have finished. Finally, the graph of the halo is defined by $(V'_h, E'_h = E \cap (V'_h \times V'_h))$. Ignoring the time for partition managing operations (which is almost constant), the complexity to build the halo graph is equivalent to a single global breadth-first search, that is $\Theta(|V|+|E|)$.

Now, let *buildConnectedHalo* be a function building such a graph (V'_h, E'_h). Algorithm 5 gives the main steps to find a separator. Line 4, a first greedy graph growing algorithm is performed to compute a cut of the halo (V'_{h0}, S'_h, V'_{h1}). Next, a kind of double greedy graph growing is done line 5, beginning with the set of seed V'_{h0} for part 0, and V'_{h1} for part 1 (note that this version of DG has not to care about halo since the halo is given to it as the initial set of seeds to use). Finally, we get a partition (P_0, S, P_1). As in the other algorithms, these steps can be repeated, doing several passes and keeping the best one.

Algorithm 5: HaloFirstGreedyGraphGrowing

Input: graph $G = (V, E)$, number of passes *passnbr*
Output: partition (P_0, S, P_1) of V such as S is a (small) separator, $|P_0| \approx |P_1|$ and $|\overline{P_0}| \approx |\overline{P_1}|$

1 $(P_0^*, S^*, P_1^*) \leftarrow (\emptyset, V, \emptyset)$;
2 $(V'_h, E'_h) \leftarrow buildConnectedHalo(V, E, V_h)$;
3 **for** $p = 1$ **to** *passnbr* **do**
4 $\quad (V'_{h0}, S'_h, V'_{h1}) \leftarrow greedyGraphGrowing(V'_h, E'_h)$;
5 $\quad (P_0, S, P_1) \leftarrow doubleGreedyGraphGrowing(V, E, V'_{h0}, V'_{h1})$;
6 $\quad$ **if** (P_0, S, P_1) *is better than* (P_0^*, S^*, P_1^*) **then**
7 $\quad\quad (P_0^*, S^*, P_1^*) \leftarrow (P_0, S, P_1)$;
8 **return** (P_0^*, S^*, P_1^*);

We have applied the same testing protocol than in the previous subsection 3.2. The results are shown is Table 3 where the column HF refers to the halo-first greedy graph growing algorithm. We also reported results of both DG and HF in table 4 to compare them together; the best is in bold.

On the interface criterion, HF approach is better than unmodified GG and $GG^\star$ in all but four matrices. The worst case is the `turon_m` (4), but this apparent failure is due to the fact that GG performs very well on this matrix: the interface imbalance is only 244 for a number of vertices of 189924. Globally, the average gain of HF over unmodified GG on interface imbalance is 38%, with a maximum of 75, 4%; this is almost as good as DG. If we compare gains of HF over DG on this criteria, HF beats DG is 16 cases out of 30, which confirms this tendency.

Moreover, HF achieves gains on the interior imbalance in all but one matrix. On average, interior imbalance gain of HF is 56%. This is better than DG of about 10%, and if gains are compared one by one, HF beats DG on two third of the matrices. DG is not obsolete however: for instance, on matrix `ultrasound` (24), DG performs better than HF on both criteria.

4 Experimental Results

In this section, we present additional results. In order to see what are the characteristics of GG, DG and HF, we drew the partitioning performed by these algorithms on 16 domains on a small

	interface imbalance			interior imbalance		
id	GG	% $GG^\star$	% HF	GG	% $GG^\star$	% HF
1	297	7,4	**-29,3**	1311	-69,1	**-73,5**
2	1112	1,6	**-34,9**	4678	-66,5	**-74,2**
3	522	-11,3	**-63,4**	1635	**-13,9**	-7,2
4	244	**-9,0**	103,3	1568	-23,7	**-24,5**
5	475	-17,9	**-26,7**	4605	**-85,6**	-74,3
6	869	-21,3	**-48,2**	4107	**-78,5**	-65,6
7	905	49,4	**-24,2**	4942	-69,1	**-72,2**
8	261	0,0	**-69,0**	420	0,0	**-11,4**
9	365	-3,8	**-41,4**	8128	**-82,8**	-69,2
10	1002	14,0	**-66,9**	4852	**-73,2**	-47,5
11	509	-3,7	**-50,3**	2831	**-84,1**	-19,1
12	569	-25,0	**-70,1**	22416	**-79,5**	-72,9
13	532	-11,8	**-37,0**	12049	**-69,0**	-52,3
14	756	**-23,1**	-20,4	17458	-61,9	**-67,2**
15	6678	6,6	**-22,4**	35466	**-73,9**	-68,2
16	564	-11,9	**-54,4**	15092	**-65,5**	-59,2
17	2130	18,6	**-44,8**	32202	-72,9	**-73,0**
18	**335**	17,0	14,6	28057	**-70,1**	-68,3
19	2604	**-16,4**	-15,8	25853	**-66,5**	-49,7
20	9990	-14,2	**-55,4**	73635	**-80,7**	-79,6
21	927	0,6	**-49,4**	2377	-58,6	**-64,5**
22	3468	5,0	**-75,4**	**3336**	4,3	16,1
23	1869	7,1	**-24,2**	14424	**-73,3**	-72,0
24	2460	-9,1	**-55,9**	8940	-44,1	**-51,5**
25	6837	-11,6	**-65,5**	26877	-63,0	**-80,3**
26	780	-15,9	**-33,6**	24987	-58,3	**-66,0**
27	6168	-27,5	**-73,6**	67721	-68,7	**-74,5**
28	4344	-15,6	**-47,9**	240729	**-76,9**	-73,2
29	11539	8,6	**-57,0**	244959	**-73,9**	-70,9
30	9936	-6,9	**-55,9**	286992	**-72,6**	-71,4

Table 3: Results with halo first greedy graph growing compared to Scotch greedy graph growing

	interface imbalance			interior imbalance		
id	*GG*	% *DG*	% *HF*	*GG*	% *DG*	% *HF*
1	297	-19,2	**-29,3**	1311	-23,3	**-73,5**
2	1112	**-40**	-34,9	4678	**-78,9**	-74,2
3	522	-39,7	**-63,4**	1635	**-13**	-7,2
4	244	**-11,5**	103,3	1568	**-31,1**	-24,5
5	475	-3,8	**-26,7**	4605	-62,3	**-74,3**
6	869	-41,3	**-48,2**	4107	-49,6	**-65,6**
7	905	26,4	**-24,2**	4942	-63,7	**-72,2**
8	261	-46,7	**-69**	420	79,8	**-11,4**
9	365	**-44,9**	-41,4	8128	-65,2	**-69,2**
10	1002	-32,1	**-66,9**	4852	-44,6	**-47,5**
11	509	-44,4	**-50,3**	2831	27,5	**-19,1**
12	569	-59,1	**-70,1**	22416	-67,5	**-72,9**
13	532	**-58,5**	-37	12049	-49,5	**-52,3**
14	756	**-46,3**	-20,4	17458	-66,6	**-67,2**
15	6678	**-29,8**	-22,4	35466	**-68,4**	-68,2
16	564	**-62,9**	-54,4	15092	-58	**-59,2**
17	2130	**-50,4**	-44,8	32202	-67,7	**-73**
18	335	**-19,7**	14,6	28057	-64,9	**-68,3**
19	2604	**-42**	-15,8	25853	**-59,5**	-49,7
20	9990	-25,8	**-55,4**	73635	-16,5	**-79,6**
21	927	-40,9	**-49,4**	2377	-59,5	**-64,5**
22	3468	**-78,5**	-75,4	3336	18,2	**16,1**
23	1869	-3,2	**-24,2**	14424	-64,4	**-72**
24	2460	**-73,4**	-55,9	8940	**-60,6**	-51,5
25	6837	**-69,7**	-65,5	26877	-68,9	**-80,3**
26	780	**-53,8**	-33,6	24987	-65,2	**-66**
27	6168	-68,4	**-73,6**	67721	**-76,2**	-74,5
28	4344	-41,4	**-47,9**	240729	**-77,1**	-73,2
29	11539	-51,3	**-57,0**	244959	**-77,2**	-70,9
30	9936	-52,0	**-55,9**	286992	-8,5	**-71,4**

Table 4: Comparison between double greedy graph growing and halo first greedy graph growing

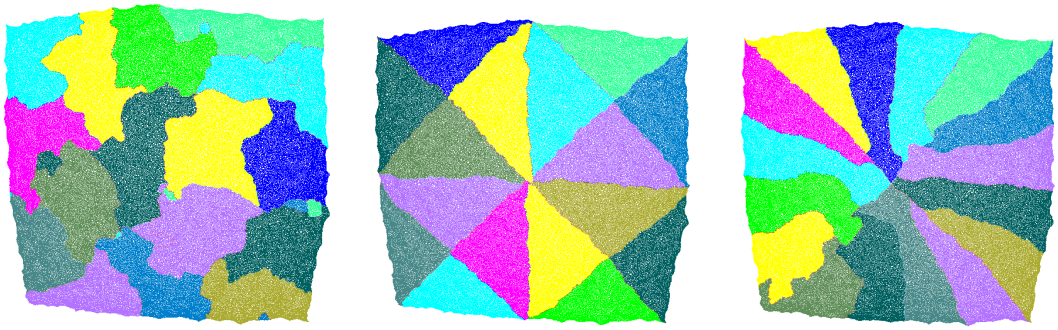


Figure 7: Partitioning of the graph of matrix `darcy003` in 16 domains with SCOTCH. From left to right, the method applied was greedy graph growing, double greedy graph growing and halo-first greedy graph growing.

mesh called `darcy003`, without the multilevel framework. Figure 7 gives the results obtained. It can be seen that *GG* makes domains with irregular shapes, leading to an interface imbalance of 224. On the contrary, *DG* performs better on this example, getting 16 triangular-shaped domains. The halo imbalance of 151 comes from the fact that the eight triangles in the center have their three edges touching other domains, whereas the eight on the corners have one edge on the border, touching no other domain. Finally, *HF* is the best with a halo imbalance of 145. To achieve that, it builds some kind of long-shaped domains around the "center" of the mesh.

To conclude this section, we present a complementary study with a variable number of domains. As previously said in the introduction, we are interested in domain decomposition for a hybrid solver where each domain will be factorized in parallel of the others. Each single factorization will be performed with a direct solver which can be parallel itself. So, we can exploit *two* levels of parallelism and thus we can afford to use larger domains. This is interesting when solving ill-conditioned linear systems for which too much domains often leads to bad convergence issues in terms of number of iterations. For these reasons, we target a number of domains which is not too high, typically between 64 and 512.

Results are reported in Tables 5, 6 and 7. For this study, we focus on the three largest matrices of our pool: `Almond` (28), `NICE-7` (29) and `10millions` (30). The column *dom* gives the number of domains in which the graph was splitted. First, we can see that, in almost all configurations, one of our strategies outperforms $GG^\star$, and if not, at least one of them is very close. We remark that on more than 16 domains, double *DG* sometimes does not always work well: on the matrix `10millions` (30), it worsen both interface and interior imbalance compared to original *GG*. *HF* provides better results, with significant gains on both criteria on most cases. In particular, it is the best (or very close to) for 512 domains on all three matrices on both criteria. Thus, we think that for a large number of domains, *HF* should be favoured. However, in the context of parallel partitioning, one can consider trying both approaches and taking the best of *DG* and *HF*.

5 Conclusion

In this paper, our objective was to build a good domain decomposition of a graph to be used as an entry by a hybrid solver. To get a good load balancing, we needed to get both balanced interior

	interface imbalance				interior imbalance			
dom	*GG*	% *GG**	% *DG*	% *HF*	*GG*	% *GG**	% *DG*	% *HF*
16	4344	-15,6	-41,4	**-47,9**	240729	-76,9	**-77,1**	-73,2
32	3179	**-34,8**	-31,5	-0,5	133886	-73,8	**-80,1**	-76,9
64	2258	-5,8	**-47,6**	-17,8	83819	**-80,5**	-79,2	-79,1
128	1822	-29,5	**-42,9**	-32,6	48087	-78,4	-77,6	**-80,9**
256	1071	-2,2	**-44,4**	2,5	27695	-81,6	-77,9	**-83,0**
512	910	0,8	-17,1	**-22,3**	16243	-83,8	-80,0	**-84,7**

Table 5: Results of *DG* and *HF* with different numbers of domains, compared to Scotch greedy graph growing, for matrix `Almond`

	interface imbalance				interior imbalance			
dom	*GG*	% *GG**	% *DG*	% *HF*	*GG*	% *GG**	% *DG*	% *HF*
16	11539	8,6	-51,3	**-57,0**	244959	-73,9	**-77,2**	-70,9
32	10991	-26,3	**-45,7**	-38,5	188834	-80,3	-81,0	**-81,6**
64	8997	-27,5	40,7	**-30,8**	101838	-75,9	-6,6	**-80,2**
128	5694	-14,4	11,7	**-26,9**	66792	-83,9	-1,9	**-84,3**
256	4554	-3,2	**-33,1**	-27,6	34314	-77,4	7,3	**-82,4**
512	3762	-16,7	-30,8	**-33,1**	19734	-79,1	-6,3	**-84,2**

Table 6: Results of *DG* and *HF* with different numbers of domains, compared to Scotch greedy graph growing, for matrix `NICE-7`

	interface imbalance				interior imbalance			
dom	*GG*	% *GG**	% *DG*	% *HF*	*GG*	% *GG**	% *DG*	% *HF*
16	9936	-6,9	-52,0	**-55,9**	286992	**-72,6**	-8,5	-71,4
32	6666	-0,5	43,7	**-56,6**	188900	-69,5	13,0	**-77,7**
64	7036	-13,3	**-47,4**	-31,1	125444	-76,9	-18,8	**-78,4**
128	4564	-11,2	4,0	**-48,7**	79754	**-82,9**	-52,9	-78,8
256	3114	-6,2	163,5	**-32,5**	42931	-79,5	-30,2	**-80,8**
512	2336	-19,5	22,2	**-54,2**	25800	**-83,5**	49,3	-83,4

Table 7: Results of *DG* and *HF* with different numbers of domains, compared to Scotch greedy graph growing, for matrix `10millions`

node and interface node set sizes. We decided to revisit the recursive algorithm introduced by Lipton and al. in the context of generalized nested dissection. This led us to keep track of the halo vertices during the recursion of the algorithm. In the software context of the SCOTCH partitioner, we modified the multilevel framework and adapted the Fiduccia-Mattheyses algorithm to refine separators during the uncoarsening steps. We also proposed two effective alternatives to the greedy graph growing algorithm for partitioning the coarsest graph: double *GG* and halo-first greedy *GG*. All those changes do not impact the computational complexity of the SCOTCH partitioner. We obtained very good balance gains on both criteria on most matrices and in particular on the biggest industrial test cases which have several millions of vertices. Our new algorithms keep behaving well even when we increase the number of domains, in particular HF.

In the short term, we will first study the impact of our work on the quality of the parallel performances on the MAPHYS hybrid solver which is developped in our research team. Secondly, our algorithms will be adapted in the parallel framework PT-SCOTCH in order to address larger problems.

References

[1] E. Agullo, L. Giraud, A. Guermouche, A. Haidar, and J. Roman. Parallel algebraic domain decomposition solver for the solution of augmented systems. *Advances in Engineering Software*, 60–61(0):23 – 30, 2013. CIVIL-COMP: Parallel, Distributed, Grid and Cloud Computing.

[2] P. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal parallel distributed symmetric and unsymmetric solvers. *Comput. Methods Appl. Mech. Eng*, 184:501–520, 1998.

[3] J. Gaidamour and P. Hénon. A parallel direct/iterative solver based on a Schur complement approach. In *IEEE 11th International Conference on Computational Science and Engineering*, pages 98–105, Sao Paulo, Brésil, July 2008.

[4] A. George. Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[5] L. Giraud, A. Haidar, and Y. Saad. Sparse approximations of the Schur complement for parallel algebraic hybrid linear solvers in 3D. Rapport de recherche RR-7237, INRIA, March 2010.

[6] L. Grigori, J. A. Demmel, and X. S. Li. Parallel Symbolic Factorization for Sparse LU with Static Pivoting. *SIAM J. Sci. Comput.*, 29(3):1289–1314, 2007.

[7] B. Hendrickson and R. Leland. A multi-level algorithm for partitioning graphs. In *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, pages 28–28, 1995.

[8] B. Hendrickson and E. Rothberg. Improving the run time and quality of nested dissection ordering. *SIAM Journal on Scientific Computing*, 20(2):468–489, 1998.

[9] P. Hénon, P. Ramet, and J. Roman. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2):301–321, January 2002.

[10] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.

[11] G. Karypis and V. Kumar. *A Software Package For Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices*, 1998.

[12] G. Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96–129, 1998.

[13] R. J. Lipton, Donald J. Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16(2), 1979.

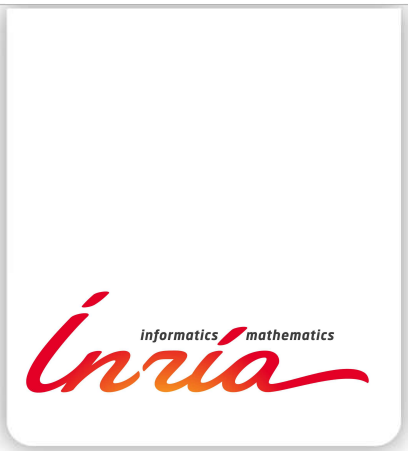
[14] G. L. Miller, S.-H. Teng, and S. A. Vavasis. A unified geometric approach to graph separators. In *Proc. 31st Annual Symposium on Foundations of Computer Science*, pages 538–547, 1991.

[15] G. L. Miller and S. A. Vavasis. Density graphs and separators. In *Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 331–336, 1991.

[16] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, volume 1067 of *Lecture Notes in Computer Science*, pages 493–498. Springer, 1996.

[17] S. Rajamanickam, E.G. Boman, and M.A. Heroux. ShyLU: A hybrid-hybrid solver for multicore platforms. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 631–643, May 2012.

[18] Ichitaro Yamazaki and Xiaoye S. Li. On techniques to improve robustness and scalability of a parallel hybrid linear solver. In *High Performance Computing for Computational Science – VECPAR 2010*, volume 6449 of *Lecture Notes in Computer Science*, pages 421–434. Springer, 2011.



RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399