An improved recursive graph bipartitioning algorithm for well balanced domain decomposition

Astrid Casadei
University of Bordeaux
CNRS (LaBRI UMR 5800)
Bordeaux, France
Astrid.Casadei@inria.fr

Pierre Ramet
University of Bordeaux
CNRS (LaBRI UMR 5800)
Bordeaux, France
ramet@labri.fr

Jean Roman
Inria
Bordeaux Institute of Technology (IPB)
CNRS (LaBRI UMR 5800)
Bordeaux, France
Jean.Roman@inria.fr

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 critical 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 [1] 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.

Keywords—Graph partitioning, domain decomposition, nested dissection, parallel sparse hybrid solvers.

I. INTRODUCTION AND MOTIVATION

Nested Dissection (ND) has been introduced by A. George in 1973 [2] 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 [3] or more generally of overlap graphs [4]. In d-dimension, such n-node graphs have separators whose size grows as 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® [5] or SCOTCH® [6].

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® [7], PAStiX® [8] and SUPERLU® [9].

However, if we examine precisely the complexity analysis for the estimation of asymptotic bounds for fill-in or operation count when using ND ordering [1], 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® [10], MAPHYS [11], [12], PDSLIN [13] and SHYLU® [14] implement different versions of this hybridization 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 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

1http://www.cs.umn.edu/~metis
2http://gforge.inria.fr/projects/scotch
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.

The construction of a domain decomposition tool leading to a good balancing of both the internal node set size and the halo node size for all the domains is then a critical point for load balancing and efficiency issues in a parallel computation context. To our knowledge, such a tool does not exist and standard partitioning techniques, even by using k-way partitioning approach, which intends to construct directly a domain decomposition of a graph in $k$ sets of independent vertices [15], do not lead in general to good results for the two coupled criteria, and for general irregular graphs coming from real-life scientific applications.

For this purpose, we revisit the original algorithm introduced by Lipton, Rose and Tarjan [1] 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 [16], [17] 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 [18]. 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.

**II. 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.

More specifically, 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. If \( C \) is a set of vertices, we will denote by \(|C|\) the sum of the non-halo weights of vertices in \( C \) and by \(|C|_h\) the sum of its halo weights. 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 order to maintain the balance during the uncoarsening process, the Fiduccia-Mattheyses has also been modified. FM method is an algorithm implemented in SCOTCH to refine an existing separator. It 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. 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 (found in the last pass). Moreover, even moves have a slight preference for moving vertices in part 0, while odd passes favor part 1 instead.

\( FM \) has three objectives when it moves a vertex: getting a reasonable imbalance \( \Delta = |P_0| - |P_1| \), minimizing the separator \( S \) and moving a vertex to the preferred \( pref \) part \(^b\). More specifically, \( FM \) will ensure that once the move is done, the new imbalance will not exceed \( \max(|\Delta|, \Delta_{th}) \), where \( \Delta \) is the current imbalance and \( \Delta_{th} \) is a fixed imbalance given by the user. This means that if current imbalance is outside the scope of \( \Delta_{th} \), it will not be degraded, and if it is within, it will remain within. If there is no possible move respecting this rule, the pass will end. Otherwise, \( FM \) 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, \( FM \) eventually selects a move to the preferred part of the pass.

In order to prevent the local search to make the same choices several times, a Tabu search is implemented. A set of tabu

\(^b\)Note that \( FM \) has some exceptions for isolated vertices, i.e. separator vertices which are adjacent to only one of the two parts. In practice, we have special treatments that induces several difficulties that will not be described in this paper.

Our modified \( FM \) algorithm is presented in Algorithm 1. 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 \( |\Delta| \) below the threshold \( \Delta_{th} \) (chosen
by user), then the function \textit{getHalo} is called line 12. This function tries to fix the halo imbalance. It uses the sign of \( \Delta = |P_0|_h - |P_1|_h \) to know the part where a vertex must be moved: a move to part 0 increases \( \Delta \), a move to part 1 decreases it. Then, it picks a vertex whose move to part \( i \) will minimize the new halo imbalance. If there is no such move that improves \( \Delta \) strictly, then the function fails. Here, or if the partition had already a reasonable halo imbalance, the function \textit{getSep} is called instead (line 14), which works as described in the unmodified version.

We conclude this section by giving our strategy to choose the better partition as indicated at line 23 of Algorithm 1. 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 \(|\Delta| \leq \Delta_{th}\), then we keep a partition with \(|\Delta| \leq \Delta_{th}\) and whose \(|\Delta|\) is the smallest.
- if we already found a partition satisfying both \(|\Delta| \leq \Delta_{th}\) and \(|\Delta| \leq \Delta_{th}\), 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 \(|\Delta|\), and if there is still a tie, the one with the smallest \(|\Delta|\).

III. 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 I presents all the testing matrices, giving their size and their number of non-zero entries. The \textit{id} number will be used to identify matrices in the following. For each matrix \( A \), the symmetric graph of \( A + A^t \) is used. Matrices 1-20 come from the University of Florida Sparse Matrix Collection, and the last set of ten matrices comes from our industrial collaborations or partners.

Table I presents all the testing matrices, giving their size and their number of non-zero entries. The \textit{id} number will be used to identify matrices in the following. For each matrix \( A \), the symmetric graph of \( A + A^t \) is used. Matrices 1-20 come from the University of Florida Sparse Matrix Collection, and the last set of ten matrices comes from our 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 minimum 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 \textsc{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 \textit{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 (\( \textit{bal} = 10\% \)) accumulates through levels of the recursion: at level \( n \), imbalance between minimum and maximum subgraph sizes may reach roughly \( \textit{bal} \times n \) percents. This is too loose for our purpose. We cannot decrease \textit{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

\begin{table}[h]
\centering
\caption{Set of test matrices. 1-20 come from the University of Florida Sparse Matrix Collection. 21-30 come from industrial collaborations or partners.}
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{id} & \textbf{Matrix} & \textbf{n} & \textbf{nnz} \\
\hline
1 & Dubcov3 & 146689 & 2489960 \\
2 & wave & 156317 & 2118662 \\
3 & dj_pretok & 182730 & 1512512 \\
4 & turing_m & 189924 & 1557062 \\
5 & stomach & 211360 & 2255876 \\
6 & BenAltec1 & 235874 & 12904622 \\
7 & toros3 & 259156 & 4372658 \\
8 & marion002 & 389874 & 1867114 \\
9 & helm2d03 & 392257 & 2349678 \\
10 & kim2 & 456976 & 10905268 \\
11 & mc2depi & 525825 & 3148809 \\
12 & timl_unsym & 917825 & 3669976 \\
13 & tczem & 921632 & 3673536 \\
14 & ldooor & 952203 & 45570272 \\
15 & bone010 & 986703 & 70679622 \\
16 & ecology1 & 1000000 & 39996000 \\
17 & diecfFilterY & 1102824 & 88203196 \\
18 & thermal2 & 1228045 & 7352268 \\
19 & StocF-1465 & 1465137 & 19540252 \\
20 & Hook-1498 & 1498023 & 59419422 \\
21 & NICE-25 & 1406562 & 35439944 \\
22 & MHD & 485597 & 23747544 \\
23 & Inline & 503712 & 36312630 \\
24 & ultrasound & 531441 & 32544720 \\
25 & Andikw_1 & 943695 & 7608152 \\
26 & Halterc & 1288825 & 18375900 \\
27 & NICE-5 & 2233031 & 175971592 \\
28 & Almond & 6994683 & 102965400 \\
29 & NICE-7 & 8159755 & 661012794 \\
30 & 10millions & 10423737 & 157298268 \\
\hline
\end{tabular}
\end{table}
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 $GG^*$ and the other columns use the level-dependent constraint described above with $bal = 10\%$ and a threshold of $1\%$. $GG^*$ thus refers to a modified SCOTCH strategy with default graph partitioning algorithms but using the level-dependent $bal$ constraint.

A. 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 2. 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 $\choicenextv$ 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 2. 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 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.

B. 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

```latex
\begin{algorithm}
\caption{GreedyGraphGrowing}
\label{alg:greedygraph}
\begin{algorithmic}[1]
\Require graph $G = (V,E)$, number of passes \textit{passnbr}
\Ensure partition $(P_0, S, P_1)$ of $V$ such as $S$ is a (small) separator and $|P_0| \approx |P_1|$
\State $(P_0^*, S^*, P_1^*) \leftarrow (\emptyset, V, \emptyset)$;
\For{$p = 1$ \textbf{to} \textit{passnbr}}
\State $w \leftarrow \text{RandomSeed}(V)$;
\State $P_0 \leftarrow V \setminus \{w\}$;
\State $P_1 \leftarrow \emptyset$;
\State $S \leftarrow \{w\}$;
\While{$|P_0|$ and $|P_1|$ are not balanced}
\State $v \leftarrow \text{getVertex}(S)$;
\State $N \leftarrow \{j \mid (v,j) \in E \text{ and } j \in P_0\}$;
\State $S \leftarrow S \setminus \{v\} \cup N$;
\State $P_0 \leftarrow P_0 \setminus N$;
\State $P_1 \leftarrow P_1 \cup \{v\}$;
\If{$(P_0^*, S^*, P_1^*)$ \textbf{is better than} $(P_0^*, S^*, P_1^*)$}
\State $(P_0^*, S^*, P_1^*) \leftarrow (P_0, S, P_1)$;
\EndIf
\EndWhile
\EndFor
\State \Return $(P_0^*, S^*, P_1^*)$;
\end{algorithmic}
\end{algorithm}
```

Figure 2. Illustration of a corner case with the modified greedy-graph-growing algorithm. The blue partition is eventually split in three disconnected parts.
much as possible, the moment when parts meet each other.

Algorithm 3 describes the new method. Line 3 picks the two seeds \( w_1 \) and \( w_{-\phi} \) 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 \( {-i} \), 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 \( w_{-\phi} \) (i.e. the vertex \( v \) with the smallest value \( \text{dist}(v, w_i) - \text{dist}(v, w_{-\phi}) \)) 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 \( \min_j \{ \text{dist}(v, \text{ctrlpts}_i[j]) \} - \min_j \{ \text{dist}(v, \text{ctrlpts}_{-i}[j]) \} \). In other words, part \( i \) will be attracted by its own control points, and repulsed by the control points of \( {-i} \). (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 3 and 4 show that depending on the graph, both solutions can be better than the other. On the top case in Figure 3, 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 4, 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 II. The columns \( GG \) give the interface and interior imbalance of unmodified SCOTCH with balanced. 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 \( GG^\ast \) 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.

We can see that interface balancing of \( DG \) is much better than unmodified \( GG \) and \( GG^* \) 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
**C. 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 \((V_h, E \cap (V_h \times V_h))\), 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 it. Though, ignoring the position of halo vertices to build a separator could not be enough to reconnect the worst case, it can be totally disconnected, and considering the graph may not be connected, even if the whole graph is. In particular interest for us), gains are very good on both criteria.

\[GG^* \text{ on one third of the test cases, which is rather honorable} \]

**Algorithm 3: DoubleGreedyGraphGrowing**

**Input:** graph \(G = (V, E)\) with halo \(V_h \subset V\), number of passes \(\text{passnbr}\)

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

```plaintext
1 \((P_0^*, S^*, P_1^*) \leftarrow (\emptyset, V, \emptyset)\);
2 \text{for} p = 1 \text{ to } \text{passnbr} \text{ do}
3 \quad (w_0, w_1) \leftarrow \text{RandomSeeds}(V);
4 \quad \text{ctrlpts}_0 \leftarrow \{w_0\}, \text{ctrlpts}_1 \leftarrow \{w_1\};
5 \quad \text{success} \leftarrow \text{false};
6 \quad \text{for} q = 1 \text{ to } \text{triesnbr} \text{ do}
7 \quad \quad P_0, P_1 \leftarrow \emptyset;
8 \quad \quad S_0 \leftarrow \{w_0\}, S_1 \leftarrow \{w_1\};
9 \quad \quad \text{ctrldist} \leftarrow \text{computeDistances}(G, \text{ctrlpts}_0, \text{ctrlpts}_1);
10 \quad \quad \text{while} V \setminus (P_0 \cup P_1) \neq \emptyset \text{ do}
11 \quad \quad \quad \quad \text{if} |P_0| < |P_1| \text{ then}
12 \quad \quad \quad \quad \quad i \leftarrow 0;
13 \quad \quad \quad \quad \text{else}
14 \quad \quad \quad \quad \quad i \leftarrow 1;
15 \quad \quad \quad \quad \text{if} S_i = \emptyset \text{ then}
16 \quad \quad \quad \quad \quad \quad \text{break};
17 \quad \quad \quad \quad \quad v \leftarrow \text{getVertex}(S_i, \text{ctrldist});
18 \quad \quad \quad \quad \quad S_i \leftarrow S_i \setminus \{v\} \cup \{j\} \mid (v, j) \in E \text{ and } j \in V \setminus (P_0 \cup P_1);
19 \quad \quad \quad \quad \quad P_i \leftarrow P_i \cup \{v\};
20 \quad \quad \quad \text{if} |V \setminus (P_0 \cup P_1)| \leq 0, 1/|V| \text{ then}
21 \quad \quad \quad \quad \quad \quad P_{\sim i} \leftarrow V \setminus P_i;
22 \quad \quad \quad \quad \text{success} \leftarrow \text{true};
23 \quad \text{else}
24 \quad \quad \quad \quad \text{ctrlpts}_i \leftarrow \text{ctrlpts}_i \cup \text{findNewControlPoint}(G, P_i, P_{\sim i});
25 \quad \text{if} \text{success} \text{ then}
26 \quad \quad S \leftarrow \text{MinVertexCover}(E \cap (P_0 \times P_1));
27 \quad \quad P_0 \leftarrow P_0 \setminus S, P_1 \leftarrow P_1 \setminus S;
28 \quad \text{if} (P_0, S, P_1) \text{ is better than } (P_0^*, S^*, P_1^*) \text{ then}
29 \quad \quad \quad \text{\((P_0^*, S^*, P_1^*) \leftarrow (P_0, S, P_1)\);}
30 \text{return } (P_0^*, S^*, P_1^*)
```

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

\[\text{Figure 5. In this example, the halo (vertices on the right) is totally disconnected, and require to explore the far-away neighbourhood to reconnect it. Though, ignoring the position of halo vertices to build a separator could lead to a bad separator, if for instance we take the black halo vertices in one part and the gray ones in the other.}\]
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 4 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 $(V'_{h0}, S'_{h1}, V'_{h1})$. 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.

We have applied the same testing protocol than in the previous subsection III-B. The results are shown is Table III where the column $HF$ refers to the halo-first greedy graph growing algorithm. A sign has been added on the right of each column $HF$: a '+' means that this gain is better (i.e. smaller) that the corresponding $DG$ gain and a '-' that $DG$ is better.

On the interface criterion, $HF$ approach is better than unmodified $GG$ and $GG^*$ 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
what are the characteristics of HF over DG on this criteria, we have 16 ‘+’ 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.

IV. 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 mesh called darcy003, without the multilevel framework. Figure 6 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 IV, V and VI. 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*, 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.

V. 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 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 developed in our research team. Secondly, our algorithms will be adapted in the parallel framework PT-SCOTCH in order to address larger problems.

REFERENCES