Direct solution of larger coupled sparse/dense linear systems using low-rank compression on single-node multi-core machines in an industrial context

1st Emmanuel Agullo
HiePACS team
Inria Bordeaux Sud-Ouest
Bordeaux, France
emmanuel.agullo@inria.fr

2nd Marek Felšöci
HiePACS team
Inria Bordeaux Sud-Ouest
Bordeaux, France
marek.felsoci@inria.fr

3rd Guillaume Sylvand
HiePACS team
Airbus Central R&T
Issy-les-Moulineaux, France
guillaume.sylvand@airbus.com

***Abstract*—While hierarchically low-rank compression methods are now commonly available in both dense and sparse direct solvers, their usage for the direct solution of coupled sparse/dense linear systems has been little investigated. The solution of such systems is though central for the simulation of many important physics problems such as the simulation of the propagation of acoustic waves around aircrafts. Indeed, the heterogeneity of the jet flow created by reactors often requires a Finite Element Method (FEM) discretization, leading to a sparse linear system, while it may be reasonable to assume as homogeneous the rest of the space and hence model it with a Boundary Element Method (BEM) discretization, leading to a dense system. In an industrial context, these simulations are often operated on modern multicore workstations with fully-featured linear solvers. Exploiting their low-rank compression techniques is thus very appealing for solving larger coupled sparse/dense systems (hence ensuring a finer solution) on a given multicore workstation, and – of course – possibly do it fast. The standard method performing an efficient coupling of sparse and dense direct solvers is to rely on the Schur complement functionality of the sparse direct solver. However, to the best of our knowledge, modern fully-featured sparse direct solvers offering this functionality return the Schur complement as a non compressed matrix. In this paper, we study the opportunity to process larger systems in spite of this constraint. For that we propose two classes of algorithms, namely multi-solve and multi-factorization, consisting in composing existing parallel sparse and dense methods on well chosen submatrices. An experimental study conducted on a 24 cores machine equipped with 128 GiB of RAM shows that these algorithms, implemented on top of state-of-the-art sparse and dense direct solvers, together with proper low-rank assembly schemes, can respectively process systems of 9 million and 2.5 million total unknowns instead of 1.3 million unknowns with a standard coupling of compressed sparse and dense solvers.**

***Index Terms*—sparse and dense matrices, large linear systems, direct method, parallel solvers, low-rank compression, Finite Elements Method (FEM), Boundary Elements Method (BEM), FEM/BEM coupling**

I. Introduction

We are interested in the solution of very large linear systems of equations $Ax = b$ with the particularity of having both sparse and dense parts. Such systems appear in an industrial context when we couple two types of finite elements methods, namely the volume Finite Element Method (FEM) [1]–[3] and the Boundary Element Method (BEM) [4], [5]. This coupling is used to simulate the propagation of acoustic waves around aircrafts (see Figure 1, left). In the jet flow created by the reactors, the propagation media (the air) is highly heterogeneous in terms of temperature, density, etc. Hence we need a FEM approach to compute acoustic waves propagation in it. Elsewhere, we approximate the media as homogeneous and use BEM to compute the waves propagation. This leads to a coupled sparse/dense FEM/BEM linear system (1) with two groups of unknowns: x_v related to a FEM volume mesh v of the jet flow and x_s related to a BEM surface mesh s covering the surface of the aircraft as well as the outer surface of the volume mesh (see Figure 1, right). The linear system $Ax = b$ to be solved may be more finely written as:

$$\begin{matrix} R_1 \\ R_2 \end{matrix} \begin{bmatrix} A_{vv} & A_{sv}^T \\ A_{sv} & A_{ss} \end{bmatrix} \times \begin{bmatrix} x_v \\ x_s \end{bmatrix} = \begin{bmatrix} b_v \\ b_s \end{bmatrix}. \quad (1)$$

Fig. 1. An acoustic wave (blue arrow) emitted by the aircraft's engine, reflected on the wing and crossing the jet flow. Real-life case (left) [6] and a numerical model example (right). The red surface mesh represents the aircraft's surface and the surface of the green volume mesh of the jet flow.

In (1), R_1 and R_2 respectively denote the first and the second block rows of the linear system and A is a 2×2 block symmetric coefficient matrix (see Figure 2).

In this work, we seek to solve (1) using a direct method involving a coupling of state-of-the-art sparse direct and dense direct solvers. Due to the size of the objects (full aircrafts) and the acoustic frequencies simulated (up to 20 kHz, for

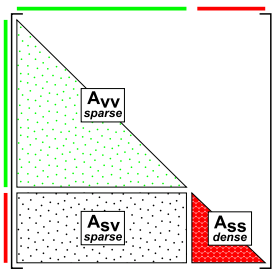

Fig. 2. Internal dimensions and sparsity of A in (1). A_{vv} is a large sparse submatrix representing the action of the volume part on itself, A_{ss} is a smaller dense submatrix representing the action of the exterior surface on itself, and A_{sv} is a sparse submatrix representing the action of the volume part on the exterior surface.

audible frequencies) inducing edge sizes below 1 cm, the number of volume unknowns x_v and surface unknowns x_s can be extremely high and respectively grow like the cube and the square of the simulated acoustic frequency. It is therefore crucial, from an industrial point of view, to be able to reduce the cost of these simulations in terms of memory usage and time, in order to tackle the largest possible spectrum of audiofrequencies on existing workstations.

Many modern fully-featured linear solvers implement low-rank compression techniques in an effort to lower the memory footprint of the computation and potentially reduce the computation time. However, the usage of low-rank compression in the context of coupled sparse/dense systems has not been investigated in the literature to the best of our knowledge. In this work, we explore the possibility to take advantage of these techniques so as to process larger coupled systems on a given multicore workstation.

The most advanced standard coupling of sparse and dense direct solvers is based on the Schur complement functionality of the former. Unfortunately, to the best of our knowledge, the state-of-the-art sparse direct solvers providing this functionality do not allow one to retrieve a compressed version of the Schur complement ($A_{sv}A_{vv}^{-1}A_{sv}^T$ further introduced in Section II) matrix. In the systems we consider, the dense storage of the whole Schur complement may be prohibitively large and lead such a standard coupling to run out of memory. To cope with this constraint, we propose two new classes of algorithms, namely the multi-solve and the multi-factorization methods, implemented on top of state-of-the-art parallel direct solvers and operating on well chosen submatrices.

The rest of the document is organized as follows: in Section II, we formalize the direct solution method for (1) and present the standard sparse/dense solver couplings as well as their limitations. In Section III, we position our work regarding similar approaches found in the literature. In Section IV, we introduce the new multi-solve and multi-factorization algorithms. We present an experimental evaluation and an industrial application of the proposed algorithms in sections V and VI, respectively, and we conclude in Section VII.

II. Direct solution of coupled sparse/dense FEM/BEM systems

A. Formulation

The first step of a direct solution of (1) consists of reducing the problem on the boundaries and simplifying the system to solve. Based on its first block row R_1, we express x_v as:

$$x_v = A_{vv}^{-1}(b_v - A_{sv}^T x_s) \ . \tag{2}$$

Then, substituting x_v in R_2 by (2) yields a reduced system without x_v in R_2. This represents one step of Gaussian elimination, i.e. $R_2 \leftarrow R_2 - A_{sv}A_{vv}^{-1} \times R_1$:

$$\begin{matrix} R_1 \\ R_2 \end{matrix}\begin{bmatrix} A_{vv} & A_{vs} \\ 0 & A_{ss} - A_{sv}A_{vv}^{-1}A_{sv}^T \end{bmatrix} \times \begin{bmatrix} x_v \\ x_s \end{bmatrix} = \begin{bmatrix} b_v \\ b_s - A_{sv}A_{vv}^{-1}b_v \end{bmatrix}. \tag{3}$$

The expression $A_{ss} - A_{sv}A_{vv}^{-1}A_{sv}^T$, which appears in R_2 in (3), is often referred to as the Schur complement [7], noted S, associated with the partitioning v and s of the variables in the system (see Section I). Basically, we have to compute S:

$$S = A_{ss} - A_{sv}A_{vv}^{-1}A_{sv}^T \tag{4}$$

and find x_s by solving the reduced Schur complement system matching R_2 in (3) after elimination of x_v:

$$x_s = S^{-1}(b_s - A_{sv}A_{vv}^{-1}b_v) \ . \tag{5}$$

Once we have computed x_s, we use its value to determine x_v according to (2).

The decomposition of A (see Section I) and the choice to eliminate x_v from R_2 in (1) allows one to take advantage of the sparsity of the submatrix A_{vv} during the solution process. On the contrary, eliminating x_s from R_1 instead of the current choice would result in an important fill-in [8] of A_{vv} where the Schur complement would have been computed.

When solving the problem numerically, rather than actually computing the inverses of A_{vv} and S, we factorize A_{vv} as well as S into products of matrices making the equations easier to solve. For complex (symmetric but not positive definite) matrices, we rely on a LL^T factorization. In case of real matrices, we use a LDL^T factorization instead.

B. Ideal symmetry and sparsity of the factorized coupled system

In this section, we describe the ideal approach for the numerical computation of the solution of (1) which would allow us to fully take advantage of the symmetry of the system by containing the computation in its lower symmetric part as well as to exploit the sparse character of A_{vv} and A_{sv} as much as possible. We then discuss the challenge of implementing this approach in practice.

In theory, we would begin by computing S. To do this, we would factorize A_{vv} into $L_{vv}L_{vv}^T$ and compute $A_{sv}(L_{vv}^T)^{-1}$ in order to express S as:

$$S = A_{ss} - [A_{sv}(L_{vv}^T)^{-1}][A_{sv}(L_{vv}^T)^{-1}]^T \ . \tag{6}$$

Then, we would factorize S into $L_S L_S^T$ and finally compute the solutions x_s and x_v using:

$$\begin{cases} x_s = (L_S L_S^T)^{-1}(b_s - A_{sv}(L_{vv}L_{vv}^T)^{-1}b_v) \\ x_v = (L_{vv}L_{vv}^T)^{-1}(b_v - A_{sv}^T x_s) \ . \end{cases} \quad (7)$$

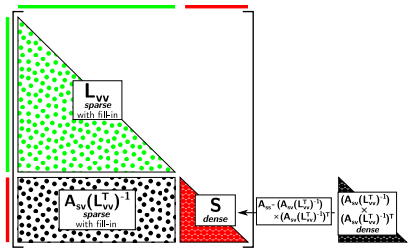

Fig. 3. Ideal approach for computing S according to (6).

In practice, exploiting the sparsity of A_{vv} and A_{sv} during these operations is a hard task. It requires to resort to advanced techniques, including symbolic factorization [9] and management of complex data-structures, e.g. to cope with arising dense submatrices due to fill-in [8]. This is what sparse direct solvers are meant for. They can transparently take care of numerical (e.g. performing the proper factorization of A_{vv} instead of computing its inverse), combinatorial (e.g. reordering the unknowns to limit fill-in) and performance (e.g. relying as much as possible on BLAS-3 operations) issues for us. If we wanted to implement the above approach by ourselves, it would require to (partially) implement a sparse direct solver, which would not only be extremely time-consuming (e.g. MUMPS 5.1.2 has 418,556 lines of code) but might also lead to an under-performing implementation. In this study (and from an industrial perspective), we instead decided to build our coupled solver on top of existing fully-featured sparse and dense well established solvers. This also means that we have to deal with their API. Sections II-C and II-D respectively introduce the available API of sparse and dense direct solvers we can rely on as building blocks for designing a coupled solver.

C. Sparse direct solver building blocks

Most sparse direct solvers do not allow one to express that part of the unknowns is associated with a dense block. In this case, only the A_{vv} block can be handled with the sparse direct solver and other operations must be handled on top of that, leading to a sub-optimal scheme for the reasons just discussed above. We call this first scenario the baseline usage of the sparse direct solver (II-C1). Nonetheless, some fully-featured sparse direct solvers, such as MUMPS [10], PaStiX [11] or PARDISO [12], provide in their API a Schur complement functionality (for MUMPS see option `ICNTL(19)` in [13], for PaStiX see [14] and for PARDISO see Section 1.3 in [15]). It allows one to delegate the computation of S entirely to the sparse direct solver, which can (it is designed for that) fully exploit the symmetry and the sparsity of the system according to the above ideal scenario. We call this scheme the advanced usage of the sparse direct solver (II-C2).

1) Baseline usage: In the first scenario, we use the sparse direct solver only on the A_{vv} block, for performing the *sparse factorization* of the A_{vv} submatrix into $L_{vv}L_{vv}^T$ and for computing $A_{sv}(L_{vv}^T)^{-1}$ using a *sparse solve* step.

2) Advanced usage: In the second scenario, we rely on the above mentioned Schur complement functionality. This feature consists of the factorization of A_{vv} and the computation of the Schur complement $A_{sv}A_{vv}^{-1}A_{sv}^T$ associated with the $\begin{bmatrix} A_{vv} & A_{sv}^T \\ A_{sv} & 0 \end{bmatrix}$ matrix. This functionality represents a building block on its own. In the following, we shall refer to the latter as to *sparse factorization+Schur* step. Note that the resulting Schur complement is returned (due to the API of sparse direct solvers which support this functionality) as a non compressed dense matrix, which will still be a limitation in our context as discussed further.

D. Dense direct solver building blocks

Once the Schur complement is obtained with either the baseline or the advanced usage of sparse direct solvers discussed above, a dense direct solver may be used for some of the operations associated with (7), namely the *dense factorization* of S and the *dense solve* for computing x_s.

E. Baseline sparse/dense solver coupling

A possible way of composing these building blocks is to rely on the above baseline usage of the sparse direct solver (Section II-C1). The first step of the solution process is thus a *sparse factorization* of A_{vv} into $L_{vv}L_{vv}^T$. The factorization is followed by a *sparse solve* step to get $A_{vv}^{-1}A_{sv}^T$, which is, in this baseline usage, non optimally retrieved as a dense matrix. From a combinatorial perspective, the result of $A_{vv}^{-1}A_{sv}^T$ is not dense. However, taking advantage of its sparsity is far from trivial (see Section II-B). It is possible to exploit the sparsity of the operands during the *sparse solve* [16] (for MUMPS see option `ICNTL(20)` in [13], which we always turn on in this study). Nevertheless, because the internal data structures are complex, the user still gets, as in all fully-featured direct solvers we are aware of, the output as dense.

A sparse-dense matrix multiplication (SpMM) then follows to compute $A_{sv}A_{vv}^{-1}A_{sv}^T$, for which it is not evident to exploit the sparsity either. Indeed, $A_{vv}^{-1}A_{sv}^T$ is retrieved as a dense matrix while A_{sv} is a 'raw' sparse matrix yielding a sub-optimal arithmetic intensity in addition to useless computation on the zeros stored in $A_{vv}^{-1}A_{sv}^T$. The subtraction $A_{ss} - A_{sv}A_{vv}^{-1}A_{sv}^T$ finally yields S (see Figure 4).

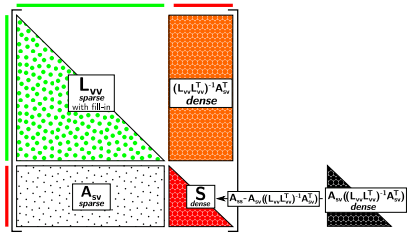

Fig. 4. Computation of S in case of the baseline sparse/dense solver coupling (see Section II-E).

In the next stage, we compute the solutions x_s and x_v following (7). At first, we form the right-hand side $b_s - A_{sv}(L_{vv}L_{vv}^T)^{-1}b_v$ by performing the *sparse solve* $(L_{vv}L_{vv}^T)^{-1}b_v$, the matrix-vector product $A_{sv}((L_{vv}L_{vv}^T)^{-1}b_v)$ and a final vector subtraction. Then, we perform the *dense factorization* of S and a *dense solve* to determine x_s. At the end, we compute the matrix-vector product $A_{sv}^T x_s$ and after a vector subtraction, we compute the *sparse solve* of $(L_{vv}L_{vv}^T)^{-1}(b_v - A_{sv}^T x_s)$ to get x_v.

F. Advanced sparse/dense solver coupling

Alternatively, we can use a *sparse factorization+Schur* step (see Section II-C2) thanks to which the sparse direct solver yields the Schur complement of $A_{sv}A_{vv}^{-1}A_{sv}^T$ associated with the $\begin{bmatrix} A_{vv} & A_{sv}^T \\ A_{sv} & 0 \end{bmatrix}$ matrix. We subtract the resulting matrix from A_{ss} according to (6) to get S (see Figure 3). The advantage of doing so is that we benefit from the fine management of the sparsity and efficient usage of BLAS-3 on non-zero blocks, transparently offered by the sparse direct solver, up to the computation of the Schur. Internally, the solver follows the ideal approach for computing S described in Section II-B, therefore benefiting from all the optimized operations. However, as discussed in Section II-C2 and recaped below, the Schur complement matrix returned by the *sparse factorization+Schur* step is dense. The computation steps following the computation of S then remain identical to the ones of the baseline coupling (see Section II-E).

G. Limitations

1) Baseline coupling: The baseline sparse/dense solver coupling (see Section II-E) presents important limitations in solving larger FEM/BEM systems in terms of both performance and memory consumption. In particular, we loose the ideal symmetry and sparsity condition of the factorized system (see Section II-B) and the result of $A_{vv}^{-1}A_{sv}^T$, stored in an extra matrix, as well as the Schur complement itself are large dense matrices.

2) Advanced coupling: Although the advanced solver coupling (see Section II-F) is an optimal approach in terms of performance, it implies storing the Schur complement matrix in dense format.

Limitations in the API of the sparse direct solver lead, in case of both baseline and advanced sparse/dense solver couplings, to the storage of potentially very large dense matrices. In terms of memory constraints, this quickly becomes a significant limiting factor in solving larger coupled systems. For example, considering the FEM/BEM system (counting 2,090,638 in the sparse part and 168,830 unknowns in the dense part) associated with the simulation in Section VI, the sole storage of the Schur complement matrix would require around 212 GiB of RAM (considering complex matrices). In case of the baseline solver coupling, we would need 2.6 TiB of extra RAM to store the result $A_{vv}^{-1}A_{sv}^T$ of the *sparse solve* in a dense matrix.

We propose to make use of low-rank compression techniques for a more efficient solution of coupled sparse/dense FEM/BEM linear systems. Some state-of-the-art sparse direct solvers provide low-rank compression out-of-the-box [17]–[19], as do some dense direct solvers . However, even if it is possible to directly apply compression on A_{vv}, A_{sv} and A_{ss}, the Schur complement – as well as the result of $A_{vv}^{-1}A_{sv}^T$ in the case of the baseline solver coupling – are still returned in dense format as discussed above. Therefore, in this work, we introduce new classes of algorithms with the aim to cope with the previously exposed limitations of the API of sparse direct solvers preventing us from efficiently exploiting low-rank compression for the solution of larger coupled sparse/dense linear systems.

III. Related work

In the literature, we have encountered similar approaches for solving coupled sparse and dense linear systems based on direct methods [20]–[24]. [20] addresses linear systems with both sparse and dense parts in the context of genomic prediction. According to the authors, such systems may have up to 100,000 unknowns associated with the dense part. However, they evaluate the proposed implementation on a smaller system with 1,279 unknowns in the dense part. [21] is related to soil-structure interaction problems. In this case, the author does not precise the target system size and presents performance evaluation results for systems with BEM-discretized part counting, to the best of our understanding, at most 1,536 boundary elements. In [22], the authors are interested in solution of linear systems arising from a coupled FEM/BEM formulation in the context of acoustic radiation problems. In the performance evaluation, they have processed systems with up to 3,017 unknowns in total. [23] is also set in the acoustic domain. In terms of problem size, the performance of the proposed implementation is evaluated on FEM systems with up to 982,912 degrees of freedom and up to 2,048 for BEM systems. [24] belongs to the domain of soil-structure interaction problems. The largest coupled test cases considered have 52,758 degrees of freedom.

To the best of our understanding, all of these approaches rely on a baseline usage (similar to the one discussed in Section II-E) of the sparse direct solver, i.e., without using the Schur complement functionality. More importantly, the tackled problem size associated with the BEM part (yielding the dense block) is relatively small which makes it possible to handle with one of the above schemes (see sections II-E and II-F). In particular, no compression method is employed. On the contrary, due to their dimension (especially the size of the dense block), the problems we tackle cannot be processed with the above baseline and advanced solver coupling approaches. We therefore introduce new algorithms to bypass these limits.

IV. Multi-solve and multi-factorization algorithms

In this section, we propose two new classes of algorithms, namely the multi-solve and multi-factorization methods, to cope with the limitation in the API of the state-of-the-art sparse direct solvers (see Section II-G). The core idea of both

methods is a blockwise computation of the Schur complement S (see Section II). Multi-solve is a variation of the baseline coupling (see Section II-E) while multi-factorization is a variation of the advanced coupling (see Section II-F).

For each class of algorithms, we propose two variants: a baseline version (see sections IV-A1 and IV-B1) and an extension ensuring compression of the Schur complement matrix S (see sections IV-A2 and IV-B2). The baseline multi-solve and multi-factorization represent the starting point for their compressed Schur counterparts and serve us in the experimental study (see Section V) to assess the intrinsic overhead of the proposed multi-stage algorithms with respect to the single-stage usage of baseline and advanced solver couplings from sections II-E and II-F. Because the activation of the compression within the sparse direct solver does not affect the overall algorithm and is completely transparent from the coupling point of view, we systematically turn it on in practice in this study for both the baseline and compressed Schur variants (see sections V and VI, except, for reference, in the first set of industrial applications). However, as we explain below, in all cases, the Schur blocks are retrieved as non compressed dense matrices. The purpose of the compressed Schur variants (sections IV-A2 and IV-B2) is to build on top of the blocking scheme of their baseline counterparts (from sections IV-A1 and IV-B1, respectively) to compress the dense Schur blocks successively retrieved in order to limit the memory consumption so as to process larger problems.

A. Multi-solve algorithm

In this approach, we build on the baseline solver coupling presented in Section II-E. However, instead of performing the *sparse solve* step $(L_{vv}L_{vv}^T)^{-1}A_{sv}^T$ using the entire submatrix A_{sv}^T, we split the latter in blocks of n_c columns $A_{sv_i}^T$ and perform successive parallel *sparse solve* operations. This leads to a blockwise assembly of the Schur complement matrix S by blocks of columns S_i. Given n_{BEM}, the number of unknowns associated with the formulation of BEM, and n_c, the number of columns of A_{sv}^T in one block $A_{sv_i}^T$, there are n_{BEM}/n_c blocks in total (see Figure 5). We note $A_{sv_i}^T$ and A_{ss_i} the i$^{\text{th}}$ blocks of n_c columns of A_{sv}^T and A_{ss}, respectively. Then, based on the definition of S in (4), S_i is a block of n_c columns of S defined as:

$$S_i = A_{ss_i} - \underbrace{A_{sv}\overbrace{(L_{vv}L_{vv}^T)^{-1}A_{sv_i}^T}^{Y_i}}_{Z_i}. \tag{8}$$

1) Baseline algorithm: The *baseline multi-solve* algorithm (see Algorithm 1) begins by the *sparse factorization* of A_{vv} (line 3) into $L_{vv}L_{vv}^T$. Then, we loop (line 4) over the blocks $A_{sv_i}^T$ of A_{sv}^T and the blocks A_{ss_i} of A_{ss} to compute all the blocks Z_i such as defined in (8). The first step of this computation (line 4) is a *sparse solve* for determining the block $Y_i = (L_{vv}L_{vv}^T)^{-1}A_{sv_i}^T$. Fully-featured sparse direct solvers additionally allow us to benefit from the sparsity of the right-hand side matrix $A_{sv_i}^T$ during the solve operation [16]. However, independently from the sparsity of the input right-hand side, the resulting Y_i is a **dense matrix** [13]. Finally, we have to temporarily store both the $A_{sv_i}^T$ and Y_i blocks explicitly (see Figure 5).

Algorithm 1: *baseline multi-solve* algorithm for computing the Schur complement S based on (8).

1 **Function** `BaselineMultiSolve`(A, b) **:**
2 $\quad A_{vv} \leftarrow$ `SparseFactorization`(A_{vv})
3 $\quad$ **for** $i = 1$ **to** n_{BEM}/n_c **do**
$\quad\quad$ ▷ Using the i^{th} block of columns of A_{sv}^T as right-hand side:
4 $\quad\quad Y_i \leftarrow$ `SparseSolve`$(A_{vv}, A_{sv_i}^T)$
5 $\quad\quad Z_i \leftarrow A_{sv} \times Y_i$ ▷ SpMM
6 $\quad\quad A_{ss_i} \leftarrow A_{ss_i} - Z_i$ ▷ AXPY
7 $\quad b_v \leftarrow$ `SparseSolve`(A_{vv}, b_v)
8 $\quad A_{ss} \leftarrow$ `DenseFactorization`(A_{ss})
9 $\quad x_s \leftarrow$ `DenseSolve`$(A_{ss}, b_s - A_{sv}b_v)$
10 $\quad x_v \leftarrow$ `SparseSolve`$(A_{vv}, b_v - A_{sv}^T x_s)$

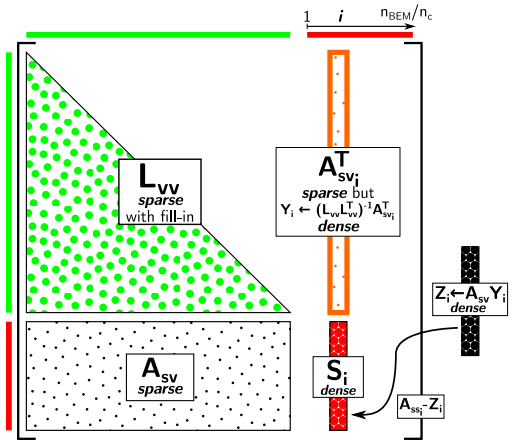

Fig. 5. Computation loop of S in *baseline multi-solve* (see Algorithm 1) $A_{sv_i}^T$ is explicitly stored and Y_i is a **dense matrix**.

2) Compressed Schur variant: In the *compressed Schur multi-solve* variant (see Algorithm 2), we apply low-rank compression techniques to both of the sparse submatrices A_{vv} and A_{sv} as well as to the dense submatrix A_{ss} and the Schur complement matrix S. Because the block Z_i is stored as a non compressed dense submatrix, we have to transform it into a temporary compressed matrix (line 8) before performing the final operation of the computation of the associated Schur complement block S_i (line 9), i.e. $A_{ss_i} - Z_i$ (see Figure 6). Note that A_{ss_i} is initially compressed, however this operation implies a recompression of the block at each iteration of the loop on i.

Moreover, in the *compressed Schur multi-solve* algorithm, we dissociate the parameter n_c, handling the size of blocks $A_{sv_i}^T$ of A_{sv}^T and consequently the size of Y_i and Z_i, from the parameter n_S handling the size of the Schur complement

Algorithm 2: *compressed Schur multi-solve* variant for computing S based on (8).

1 **Function** `CompressedSchurMultiSolve`(A, b) **:**
2 $A_{vv} \leftarrow$ `SparseFactorization`(A_{vv})
3 **for** $i = 1$ **to** n_{BEM}/n_S **do**
4 **for** $j = 1$ **to** n_S/n_c **do**
▷ Using the ij^{th} block of columns of A_{sv}^T as right-hand side:
5 $Y_{ij} \leftarrow$ `SparseSolve`$(A_{vv}, A_{sv_{ij}}^T)$
6 $Z_{ij} \leftarrow A_{sv} \times Y_{ij}$ ▷ SpMM
7 $Z_i \leftarrow$ `Concatenate`(Z_i, Z_{ij})
8 `Compress`(Z_i)
9 $S_i \leftarrow A_{ss_i} - Z_i$ ▷ Compressed AXPY
10 $b_v \leftarrow$ `SparseSolve`(A_{vv}, b_v)
11 $A_{ss} \leftarrow$ `DenseFactorization`(A_{ss})
12 $x_s \leftarrow$ `DenseSolve`$(A_{ss}, b_s - A_{sv}b_v)$
13 $x_v \leftarrow$ `SparseSolve`$(A_{vv}, b_v - A_{sv}^T x_s)$

blocks S_i. The reason for this separation is the overhead associated with the transformation of the blocks Z_i into compressed matrices as well as the computation of $A_{ss_i} - Z_i$ (line 9) which implies a recompression of A_{ss_i}. With the separate parameter n_S, we can use larger blocks Z_i to minimize additional computational cost due to frequent matrix compressions and keep smaller blocks $A_{sv_i}^T$ and Y_i preventing an excessive rise of memory consumption. We discuss this further in Section V-C1.

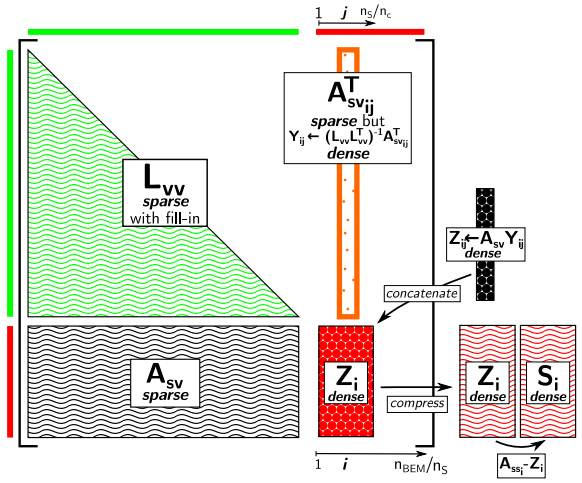

Fig. 6. Computation loop of S in *compressed Schur multi-solve* (see Algorithm 2). Compressed matrices are represented with corrugated background. $A_{sv_i}^T$ is explicitly stored and Y_i is a **dense matrix**.

B. Multi-factorization algorithm

The multi-factorization method is based on the advanced sparse/dense solver coupling presented in Section II-F. Nevertheless, instead of trying to compute the entire Schur complement using a single call to the *sparse factorization+Schur* step, we split A_{sv} and A_{sv}^T submatrices into n_b blocks A_{sv_i} and $A_{sv_j}^T$ respectively. The goal is to call the *sparse factorization+Schur* step on smaller submatrices composed of A_{vv}, A_{sv_i} and $A_{sv_j}^T$ and compute the Schur complement S by square blocks S_{ij} of equal size n_{BEM}/n_b (see Figure 7). We note A_{sv_i} a block of n_{BEM}/n_b rows of A_{sv} and $A_{sv_j}^T$ a block of n_{BEM}/n_b columns of A_{sv}^T. Then, based on the definition of S in (4), S_{ij} is a block of n_{BEM}/n_b rows and columns of S such as:

$$S_{ij} = A_{ss_{ij}} \overbrace{-A_{sv_i} A_{vv}^{-1} A_{sv_j}^T}^{X_{ij}} . \quad (9)$$

1) Baseline algorithm: The computation of the Schur complement S in the *baseline multi-factorization* algorithm (see Algorithm 3) is performed within the main loop on line 2. In this loop, we construct a temporary *non-symmetric* (except when $i = j$) submatrix W from A_{vv}, A_{sv_i} and $A_{sv_j}^T$:

$$W \leftarrow \begin{bmatrix} A_{vv} & A_{sv_j}^T \\ A_{sv_i} & 0 \end{bmatrix} \quad (10)$$

Then, we call the *sparse factorization+Schur* step on W (line 5) relying on the Schur complement feature provided by the sparse direct solver (see Section II-C). This call returns the Schur complement block $X_{ij} = -A_{sv_i}(L_{vv}U_{vv})^{-1}A_{sv_j}^T$ associated with the submatrix W. To determine the block S_{ij} of the Schur complement S, we have to compute $A_{ss_{ij}} + X_{ij}$ (line 6) following (9).

Algorithm 3: *baseline multi-factorization* algorithm for computing the Schur complement S based on (9).

1 **Function** `BaselineMultiFactorization`(A, b) **:**
2 **for** $i = 1$ **to** n_b **do**
3 **for** $j = 1$ **to** n_b **do**
4 $W \leftarrow \begin{bmatrix} A_{vv} & A_{sv_j}^T \\ A_{sv_i} & 0 \end{bmatrix}$
5 $X_{ij} \leftarrow$ `SparseFactorization+Schur`(W)
6 $A_{ss_{ij}} \leftarrow A_{ss_{ij}} + X_{ij}$ ▷ AXPY
7 $A_{vv} \leftarrow$ `SparseFactorization`(A_{vv})
8 $b_v \leftarrow$ `SparseSolve`(A_{vv}, b_v)
9 $A_{ss} \leftarrow$ `DenseFactorization`(A_{ss})
10 $x_s \leftarrow$ `DenseSolve`$(A_{ss}, b_s - A_{sv}b_v)$
11 $x_v \leftarrow$ `SparseSolve`$(A_{vv}, b_v - A_{sv}^T x_s)$

Because W is not symmetric (except when $i = j$), we can not rely on a symmetric mode of the direct solver. We thus have to enter both the lower and upper parts of A_{vv}, leading to a **duplicated storage** (see Figure 7).

Due to a limitation in the API of the sparse direct solver, the *sparse factorization+Schur* step involving W implies a re-factorization of A_{vv} in W at each iteration, although it does

not change during the computation, hence the name of the method - multi-factorization. The more blocks A_{ss} is split into, the more superfluous factorizations of A_{vv} are performed. We discuss this further in Section V-C2.

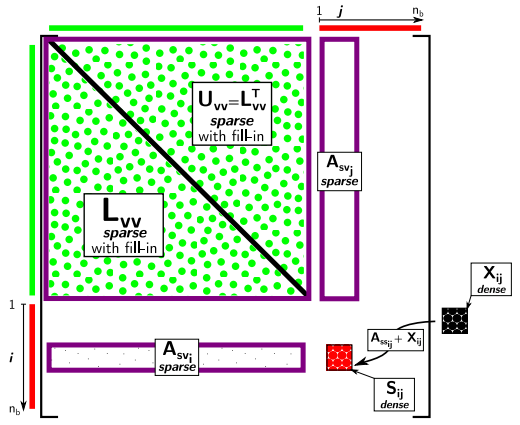

Fig. 7. Computation loop of S in *baseline multi-factorization* (see Algorithm 3). Constructing W requires a temporary **duplicated storage** of A_{vv}, A_{sv_i}, $A^T_{sv_j}$.

As in the multi-solve algorithm, we rely on the ability of the sparse direct solver to process A_{vv} and A_{sv} in a compressed fashion out-of-the-box for us. Nevertheless, the Schur complement itself remains returned as a non compressed dense matrix (see Section II-G).

2) Compressed Schur variant: In the *compressed Schur multi-factorization* variant, in addition to the low-rank compression techniques applied to both of the sparse submatrices A_{vv} and A_{sv}, we compress the X_{ij} Schur block into a temporary compressed matrix as soon as the sparse solver returns it. Hence line 6 in algorithm 3 becomes a compressed AXPY:

$$A_{ss_{ij}} \leftarrow A_{ss_{ij}} + \texttt{Compress}(X_{ij})$$

. The corresponding fully assembled S_{ij} block can then be computed using both the compressed X_{ij} and $A_{ss_{ij}}$ (line 7). Like in the case of *compressed Schur multi-solve* (see Section IV-A2), this operation implies a recompression of the initally compressed $A_{ss_{ij}}$.

V. Experimental results

A. Experimental setup

We have evaluated the previously discussed algorithms allowing for efficient low-rank compression schemes for solving coupled sparse/dense FEM/BEM linear systems such as defined in (1). For the purposes of this evaluation, we used a short pipe test case (see Figure 9) yielding linear systems with real matrices and close enough to those arising from real life models (see Figure 1) while relying on a reproducible example (https://gitlab.inria.fr/solverstack/test_fembem) available for the scientific community. The test case is designed so as we know the expected result in advance. This way, we can determine the relative error of the computed solution.

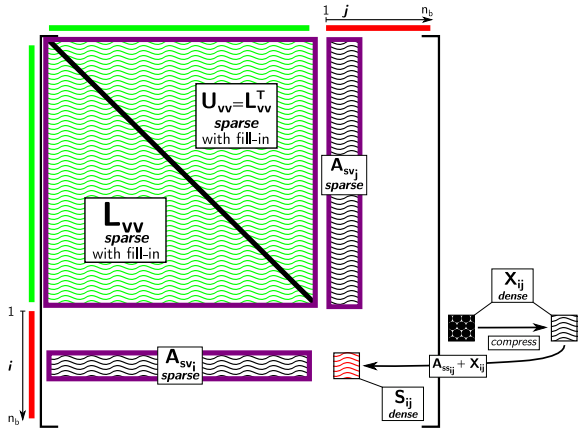

Fig. 8. Computation loop of S in *compressed Schur multi-factorization*. Compressed matrices are represented with corrugated background. Constructing W requires a temporary **duplicated storage** of A_{vv}, A_{sv_i}, $A^T_{sv_j}$.

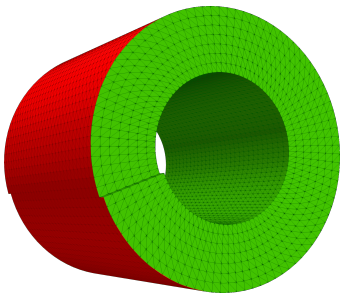

Fig. 9. Short pipe test case (length: 2 m, radius: 4 m) with BEM surface mesh in red and FEM volume mesh in green.

We have implemented the multi-solve and multi-factorization algorithms (see sections IV-A and IV-B) on top of the coupling of the sparse direct solver MUMPS [10] with either the proprietary scalapack-like dense direct solver SPIDO (for the *baseline* variants from sections IV-A1 and IV-B1) or the hierarchical low-rank $\mathcal{H}$-matrix compressed solver HMAT [25], [26] (for the *compressed* variants from sections IV-A2 and IV-B2). In the rest of the document, we thus refer to these *baseline* and *compressed* couplings as to MUMPS/SPIDO and MUMPS/HMAT, respectively. MUMPS and HMAT both provide low-rank compression and expose a precision parameter ϵ set to 10^{-3}. Unless mentionned otherwise, low-rank compression in the sparse solver MUMPS is enabled for all the benchmarks presented in this paper.

We have conducted our experiments on a single `miriel` node on the PlaFRIM platform. A `miriel` node has a total of 24 processor cores running each at 2.5 GHz, and 128 GiB of RAM. The solver test suite is compiled with GNU C Compiler (gcc) 9.3.0, Intel(R) MKL library 2019.1.144, and MUMPS 5.2.1.

B. Solving larger systems

In this section, the goal is to determine what are the largest systems that the multi-solve and multi-factorization algorithms

allow us to process on the target compute node, and the associated computation times. We consider coupled FEM/BEM linear systems with N, the total unknown count, starting at 1,000,000 (Table I, row 1). Then, we increase N until the memory limit is reached. Table I details the proportions of FEM and BEM unknowns for each value of N. In addition, we evaluate multiple configurations for each algorithm. Regarding the *baseline multi-solve* algorithm (see Section IV-A1) relying on the MUMPS/SPIDO coupling, we vary the size n_c of blocks $A_{sv_i}^T$ of columns of the A_{sv}^T submatrix between 32 and 256. For the *compressed Schur multi-solve* (see Section IV-A2), relying on the MUMPS/HMAT coupling, the size of blocks of columns of S and A_{sv}^T is handled by two different parameters, n_S and n_c respectively. In this case, we set n_c to a constant value of 256 columns (motivated by the results of the study in Section V-C) and vary n_S in the range from 512 to 4,096. In the case of the multi-factorization algorithm, both the *baseline multi-factorization* (see Section IV-B1) and the *compressed Schur multi-factorization* variants (see Section IV-B2) expose the n_b parameter handling the count of square blocks S_{ij} per block row and block column of the Schur complement submatrix S. The tested values of n_b are between 1 and 10.

Total unknowns (N)	# BEM unknowns (n_{BEM})	# FEM unknowns
1,000,000	37,169	962,831
2,000,000	58,910	1,941,090
4,000,000	93,593	3,906,407
9,000,000	160,234	8,839,766

TABLE I
COUNTS OF BEM AND FEM UNKNOWNS IN THE TARGET SYSTEMS.

In Figure 10, for each solver coupling, we show the best computation times of both variants of multi-solve and multi-factorization algorithms among all of the evaluated configurations and problem sizes. The algorithm allowing us to process the largest coupled sparse/dense FEM/BEM system is the *compressed Schur multi-solve* variant for N as high as 9,000,000 unknowns in total. In the case of the MUMPS/SPIDO coupling, when S and A_{ss} are not compressed, we could reach 7,000,000 unknowns. In the multi-factorization case, the compression of S and A_{ss} did not allow us to lower the memory footprint enough for processing larger systems than what we could achieve without. Indeed, in both cases we could process systems with up to 2,500,000 unknowns which is a considerably smaller size compared to multi-solve. This is due, in particular, to the duplicated storage induced by the loss of symmetry in the multi-factorization method (see Section IV-B) and to the relatively large ratio of FEM / BEM unknowns of the pipe test case (which will differ in the industrial test case). However, both the multi-solve and the multi-factorization methods make it possible to process significantly larger systems than the baseline coupling (see Section II-E) employed in the state-of-the-art, or than its advanced counterpart we proposed (see Section II-F). According to our experiments, the latter allowed us to process at most 1,300,000 unknowns (in $\approx$ 455 seconds) with compression turned on in the sparse direct solver and 1,000,000 ($\approx$ 917 seconds) without any compression.

One may expect that the multi-solve method should always present better computation time than the multi-factorization method due to the superfluous re-factorizations of the A_{vv} submatrix in the latter. However, in Figure 10, we can see that multi-factorization may outperform multi-solve on smaller systems, here for N as high as 2,000,000. Indeed, unlike multi-solve, which relies on a *baseline* usage of the sparse direct solver (see Section II-E), multi-factorization takes advantage of the efficiency of the Schur complement functionality of the sparse solver. On the other hand, multi-factorization implies duplicated storage leading to increased memory consumption and a lot of re-factorizations of A_{vv} when there is not enough memory with respect to the size of the problem. Here, with a fixed amount of available memory, when the problem is small enough, we can use large blocks S_{ij} of the Schur complement S and need only a few re-factorizations, in which case the multi-factorization performs better than multi-solve. For larger problems, multi-factorization is more and more penalized and the multi-solve algorithm becomes the best performing one. We further study these compromises in Section V-C. We can also observe that in the case of multi-solve, the computation time is better for the *baseline multi-solve* variant. However, this does not mean that the compression of A_{ss} and S has no effect nor a negative impact on the efficiency of the algorithm. The computation time of the factorization of the Schur complement is lower for the MUMPS/HMAT coupling but the time spent by MUMPS to perform the *sparse solve* step $A_{vv}^{-1}A_{sv}^T$ is higher for MUMPS/HMAT than for MUMPS/SPIDO. This is to be optimized in the future.

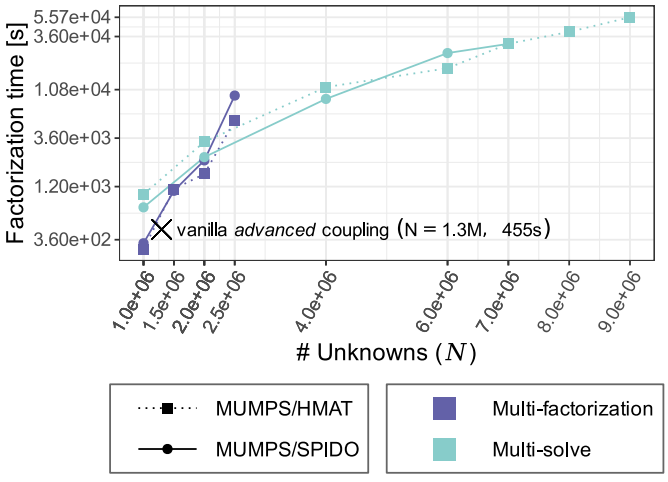

Fig. 10. Best computation times of **multi-solve** and **multi-factorization** for both of the solver couplings MUMPS/HMAT and MUMPS/SPIDO. Parallel runs using 24 threads on single `miriel` node.

Eventually, Figure 11 shows the relative error for the test cases featured in Figure 10. The precision parameter ϵ was set to 10^{-3} for both MUMPS and HMAT solvers providing low-rank compression. Unlike for the fully compressed test cases relying on the MUMPS/HMAT coupling, the relative

error is smaller in the case of MUMPS/SPIDO when the dense part of the linear system is not compressed at all and thus the final result of the computation suffers less from the loss of accuracy due to the compression. It is to note that the low-rank compression in MUMPS was activated for both the MUMPS/SPIDO and the MUMPS/HMAT couplings. In all cases, the relative error is below the selected threshold 10^{-3} which confirms the stability of the approach.

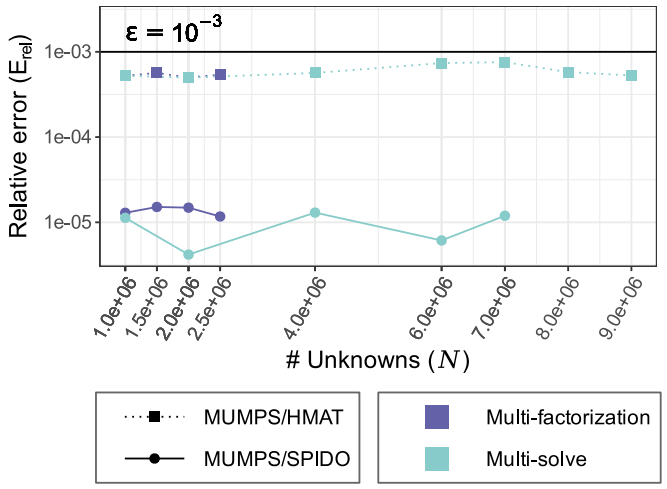

Fig. 11. Relative error E_{rel} for the runs of **multi-solve** and **multi-factorization** having the best execution times and for both of the solver couplings MUMPS/HMAT and MUMPS/SPIDO. Parallel runs using 24 threads on single `miriel` node.

C. Study of the performance-memory trade-off

We now further detail the trade-off between performance and memory consumption of the algorithms.

1) Multi-solve algorithm: We consider a coupled FEM/BEM linear system with N, the total unknown count, fixed to 2,000,000. Regarding the *baseline multi-solve* (see Section IV-A1) relying on the MUMPS/SPIDO coupling, we vary the size n_c of block $A^T_{sv_i}$ of columns of the A^T_{sv} submatrix. For the *compressed Schur multi-solve* (see Section IV-A2), using the MUMPS/HMAT solver coupling, the size of blocks of columns of S and A^T_{sv} is handled by two different parameters, n_S and n_c, respectively. In this case, we first set n_c equal to n_S varying from 32 to 256, then we maintain n_c to a constant value of 256 columns (motivated by the results presented further in this section) and vary n_S between 512 and 4,096. Note that the n_c parameter also handles the number of right-hand sides treated simultaneously by MUMPS during the *sparse solve* step $A_{vv}^{-1}A_{sv}^T$ within the Schur complement computation (see Secion II-A).

In the first place, we focus on the n_c parameter and its impact on the performance of MUMPS within the *baseline multi-solve* algorithm. According to Figure 12, setting n_c to a sufficiently high value, i.e. 256 in this case, can significantly improve the computation time. However, it is not worth to increase this value any further. On the one hand, the performance improvement begins to decrease rapidly. On the other hand, increasing n_c also means a non negligible increase of the memory footprint due to the fact that the result of the *sparse solve* step $A_{vv}^{-1}A_{sv}^T$ is a dense matrix. Based on this result, we choose to set n_c to 256 in case of the *compressed Schur multi-solve* tests. In this compressed variant, if the Schur complement block is too small, it leads to too frequent matrix compressions and increases the computation time, hence the introduction of the separate parameter n_S for the size of Schur complement blocks. We can observe this phenomenon when n_S is too small, i.e. between 32 and 256 in this case. Just like for n_c, there is no need to increase n_S as much as possible. From a sufficiently high value, i.e. 512 in this case, it has only a little impact on the computation time of the *compressed Schur multi-solve* variant. Eventually, when we compare the *baseline multi-solve* to the *compressed Schur multi-solve*, we can observe that compressing the dense submatrices S and A_{ss} allows us to significantly decrease the memory consumption of the multi-solve algorithm.

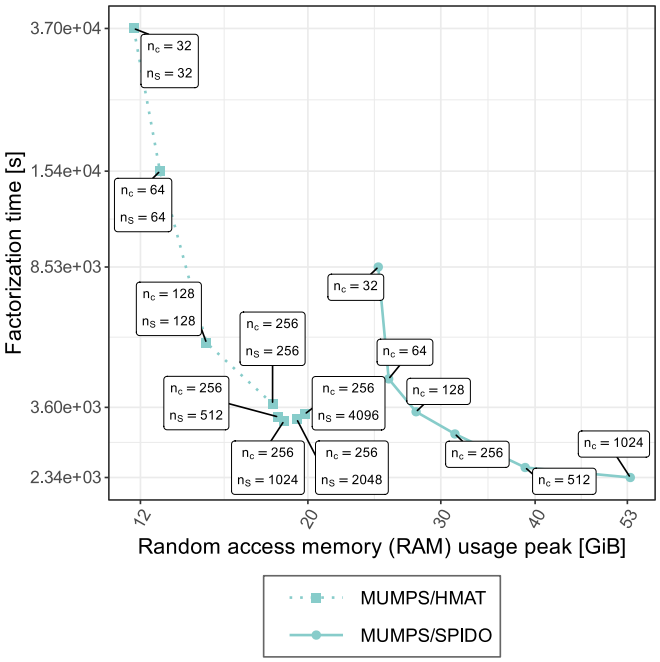

Fig. 12. Comparison between the **multi-solve** implementations for the MUMPS/SPIDO and MUMPS/HMAT couplings on a coupled FEM/BEM system counting 2,000,000 unknowns for varying values of n_c and n_S.

2) Multi-factorization algorithm: We consider a coupled FEM/BEM linear systems with N, the total unknown count fixed to 1,000,000. Both the *baseline multi-factorization* (see Section IV-B1) and the *compressed Schur multi-factorization* variants (see Section IV-B2) expose the n_b parameter handling the count of square blocks S_{ij} per block row and block column of the Schur complement submatrix S. The tested values of n_b are between 1 and 4.

In Figure 13, we can observe the negative impact of the raising number of superfluous re-factorizations of A_{vv} on the performance of the multi-factorization algorithm with the increasing number of Schur complement blocks S_{ij}. On the other hand, smaller Schur complement blocks allow one to reduce the memory footprint of the multi-factorization algorithm. Application of low-rank compression techniques to the dense submatrix A_{ss} and the Schur complement submatrix S, further reduces the memory consumption. However, the gain is not as noticeable as for the multi-solve method.

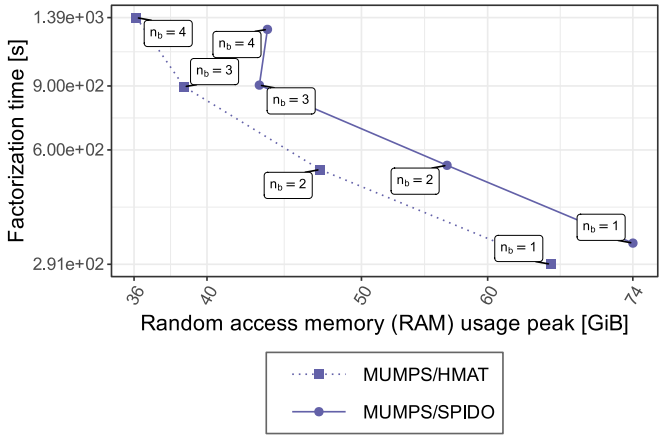

Fig. 13. Comparison between the **multi-factorization** implementations for the MUMPS/SPIDO and MUMPS/HMAT couplings on a coupled FEM/BEM system counting 1,000,000 unknowns for varying values of n_c and n_S.

VI. Industrial application

We present in this section an example of industrial application treated at Airbus Central R&T using the multi-solve and multi-factorization algorithms exposed in this article. The test case used is illustrated in Figure 14. It features 2,090,638 volume unkonwns and 168,830 surface unknowns. The proportion of surface unknowns is higher than in the short pipe test case used earlier, because in the pipe the surface mesh is only the outer surface of the jet flow (i.e. the volume mesh), whereas in this industrial test case it also includes the wing and the fuselage of the aircraft. Hence the relative cost of the (dense) BEM part will be more important and its compression have a bigger impact. Due to the physical model used, the matrix is complex and non-symmetric.

To run these tests, we use Airbus HPC5 computing facility. Each computing node has two Intel(R) Xeon(R) Gold 6142 CPU at 2.60GHz, for a total of 32 cores (hyperthreading is deactivated) and 384 GiB of RAM. The acoustic application `Actipole` is compiled with Intel(R) 2016.4 compilers and libraries, and MUMPS version 5.4.1. Each run presented below uses one node, with one process and 32 threads. For these tests, all the matrices are stored in memory (the out-of-core features of the sparse and dense solvers, when available, were not used). We use simple precision accuracy and, for compressed solvers, the accuracy is set at 10^{-4}, which is considered enough by domain specialist to obtain satisfying final results.

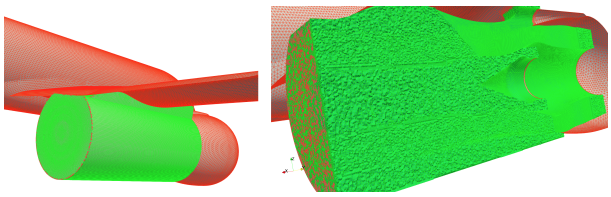

Fig. 14. Industrial test case with BEM surface mesh in red (the right part of the plane, the wing and the engine) and FEM volume mesh in green (the jet flow). On the right, a vertical cut-plane allows to see the inside of the reactor and the flow: the green mesh is made of tetrahedra, while the red mesh is hollow, and made of triangles.

	Algorithm	Dense comp.	Sparse comp.	Schur size	RAM (GiB)	Time (s)
1	state-of-the-art (§II-F)			N/A	OOM	-
2	multi-solve (§IV-A1)			N/A	249	64,969
3	multi-facto. (§IV-B1)			15,000	OOM	-
4	multi-solve (§IV-A1)		x	N/A	224	56,044
5	multi-facto. (§IV-B1)		x	15,000	275	29,089
6	multi-solve (§IV-A2)	x	x	N/A	35	34,192
7	multi-facto. (§IV-B2)	x	x	15,000	82	8,296
8	multi-facto. (§IV-B2)	x	x	30,000	92	4,287
9	multi-facto. (§IV-B2)	x	x	60,000	137	3,090

TABLE II
PERFORMANCE OF VARIOUS ALGORITHMS ON THE INDUSTRIAL TEST CASE WITH COMPRESSION ("COMP.") OPTIONNALY ENABLED. OOM STANDS FOR 'OUT-OF-MEMORY'.

Table II presents the results obtained on this test case using different approaches. For reference, we have performed preliminary experiments with compression turned off both in the sparse (unlike in the rest of the paper) and dense solvers (rows 1 - 3 in the table). In this case, the state-of-the-art advanced sparse/dense solver coupling (see Section II-F) and the multi-factorization algorithm can simply not run on this machine by lack of memory, multi-solve is the only uncompressed solver that can run here. In a first time (rows 4 - 5), adding compression in the sparse solver reduces CPU time and memory consumption for the multi-solve, and allows multi-factorization to complete successfully (using more memory but less time than the multi-solve). In a second time (rows 6 - 7), using compression in the dense solver yields an even larger improvement in CPU time and RAM usage. Finally (rows 8 - 9), multi-factorization can be further accelerated by increasing the Schur block size n_{BEM}/n_b, allowing to reduce the number of factorizations at the cost of an increase in the memory usage. Hence, the benefit of the memory gain coming from our advanced algorithms is twofold: one, it allows us to run cases that were inaccessible otherwise, and second, the memory spared can be used to increase the Schur complement size and reduce even further the CPU-time in the multi-factorization approach. In the view of these results, multi-factorization is the privileged approach in production for this type of test case on this type of machines (but this conclusion strongly depends on the number of unknowns and

the amount of memory available). An example of physical result is presented on Figure 15.

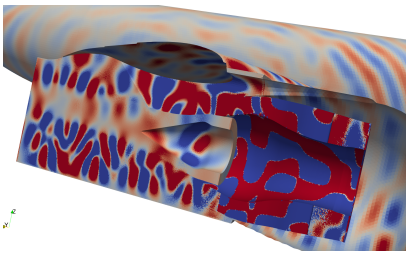

Fig. 15. Industrial test case result: the acoustic pressure is visualized in the flow (at the front) and on the surface of the plane (at the back). The color scale is saturated, so as to see the acoustic pressure on the fuselage, which is much smaller than the pressure in the flow (as one might expect, the noise is much higher *inside* the engine). The blurry pale part of the flow on the left is the hot part of the jet flow, coming out of the combustion chamber (not represented). It underlines the strong heterogeneity of the flow.

VII. Conclusion

We have extended state-of-the-art parallel direct methods for solving coupled sparse/dense systems while maintaining the ability to use fully-featured sparse and dense direct solvers. We have proposed two new classes of algorithms, the multi-solve and multi-factorization methods, which were able to benefit from the most advanced features of the building block solvers (such as the internal management of the Schur complement, compression techniques, and sparse right-hand sides), with which we were able to process academic and industrial aeroacoustic problems significantly larger than standard coupling approaches allow for on a given shared-memory multicore machine. We furthermore showed that the algorithms can take advantage of the whole available memory to increase their performance, in a memory-aware fashion.

We plan to extend this work to the out-of-core and distributed-memory cases. We will also investigate the possibility to produce Schur complement blocks directly in a compressed form (using randomized methods as in [27] or an upgraded sparse solver).

References

[1] S. Brenner and R. Scott, *The mathematical theory of finite element methods*. Springer Science & Business Media, 2007, vol. 15.

[2] A. Ern and J.-L. Guermond, *Theory and practice of finite elements*. Springer Science & Business Media, 2013, vol. 159.

[3] P. Raviart and J. Thomas, "A mixed finite element method for 2-nd order elliptic problems," in *Mathematical Aspects of Finite Element Methods*, ser. Lecture Notes in Mathematics, I. Galligani and E. Magenes, Eds. Springer Berlin Heidelberg, 1977, vol. 606, pp. 292–315. [Online]. Available: http://dx.doi.org/10.1007/BFb0064470

[4] P. K. Banerjee and R. Butterfield, *Boundary element methods in engineering science*. McGraw-Hill London, 1981, vol. 17.

[5] S. A. Sauter and C. Schwab, *Boundary Element Methods*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 183–287. [Online]. Available: https://doi.org/10.1007/978-3-540-68093-2_4

[6] Sebaso, "Jet engine airflow during take-off," https://commons.wikimedia.org/wiki/File:20140308-Jet_engine_airflow_during_take-off.jpg.

[7] F. Zhang, *The Schur complement and its applications*. Springer Science & Business Media, 2006, vol. 4.

[8] A. George, J. W. Liu, and E. Ng, "Computer solution of sparse linear systems," *Academic, Orlando*, 1994.

[9] J.-Y. L'Excellent, "Multifrontal Methods: Parallelism, Memory Usage and Numerical Aspects," habilitation à diriger des recherches, École normale supérieure de Lyon - ENS LYON, Sep. 2012. [Online]. Available: https://tel.archives-ouvertes.fr/tel-00737751

[10] P. R. Amestoy, I. S. Duff, and J.-Y. L'Excellent, "MUMPS multifrontal massively parallel solver version 2.0," 1998.

[11] P. Hénon, P. Ramet, and J. Roman, "PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems," *Parallel Computing*, vol. 28, no. 2, pp. 301–321, Jan. 2002.

[12] O. Schenk and K. Gärtner, "Solving unsymmetric sparse systems of linear equations with PARDISO," *Future Generation Computer Systems*, vol. 20, no. 3, pp. 475–487, 2004.

[13] *MUltifrontal Massively Parallel Solver (MUMPS) User's guide*, CERFACS, ENS Lyon, INPT(ENSEEIHT)-IRIT, Inria, Mumps Technologies, Université de Bordeaux, 2020. [Online]. Available: http://mumps.enseeiht.fr/doc/userguide_5.3.5.pdf

[14] *Parallel Sparse matriX package (PaStiX) Handbook*, Inria. [Online]. Available: https://solverstack.gitlabpages.inria.fr/pastix/index.html

[15] O. Schenk and K. Gärtner, "Parallel Sparse Direct and Multi-recursive iterative linear solvers (PARDISO): User's Guide," https://pardiso-project.org/manual/manual.pdf.

[16] P. Amestoy, J.-Y. L'Excellent, and G. Moreau, "On exploiting sparsity of multiple right-hand sides in sparse direct solvers," *SIAM Journal on Scientific Computing*, vol. 41, no. 1, pp. A269–A291, 2019. [Online]. Available: https://hal.inria.fr/hal-01955659

[17] P. Ghysels, X. Li, F.-H. Rouet, S. Williams, and A. Napov, "An Efficient Multicore Implementation of a Novel HSS-Structured Multifrontal Solver Using Randomized Sampling," *SIAM Journal on Scientific Computing*, vol. 38, 02 2015.

[18] P. R. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, and C. Weisbecker, "Improving multifrontal methods by means of block low-rank representations," *SIAM Journal on Scientific Computing*, vol. 37, no. 3, pp. A1451–A1474, 2015.

[19] G. Pichon, E. Darve, M. Faverge, P. Ramet, and J. Roman, "Sparse supernodal solver using block low-rank compression: Design, performance and analysis," *Journal of Computational Science*, vol. 27, pp. 255–270, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750317314497

[20] A. De Coninck, D. Kourounis, F. Verbosio, O. Schenk, B. De Baets, S. Maenhout, and J. Fostier, "Towards Parallel Large-Scale Genomic Prediction by Coupling Sparse and Dense Matrix Algebra," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2015, pp. 747–750.

[21] M. C. Genes, *Parallel application on high performance computing platforms of 3D BEM/FEM based coupling model for dynamic analysis of SSI problems*. CIMNE, 2013, p. 205–216. [Online]. Available: http://hdl.handle.net/2117/192258

[22] P. Zhang, T. Wu, and R. Finkel, "Parallel computation for acoustic radiation in a subsonic nonuniform flow with a coupled FEM/BEM formulation," *Engineering Analysis with Boundary Elements*, vol. 23, no. 2, pp. 139–153, 1999. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0955799798000733

[23] M. Ganesh and C. Morgenstern, "High-order FEM–BEM computer models for wave propagation in unbounded and heterogeneous media: Application to time-harmonic acoustic horn problem," *Journal of Computational and Applied Mathematics*, vol. 307, pp. 183–203, 2016, 1st Annual Meeting of SIAM Central States Section, April 11–12, 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377042716300711

[24] M. Schauer, J. E. Roman, E. S. Quintana-Ortí, and S. Langer, "Parallel Computation of 3-D Soil-Structure Interaction in Time Domain with a Coupled FEM/SBFEM Approach," *Journal of Scientific Computing*, vol. 52, no. 2, pp. 446–467, Aug 2012. [Online]. Available: https://doi.org/10.1007/s10915-011-9551-x

[25] B. Lizé, "Résolution Directe Rapide pour les Éléments Finis de Frontière en Électromagnétisme et Acoustique : $\mathcal{H}$-Matrices. Parallélisme et Applications Industrielles." Ph.D. dissertation, Université Paris 13, 2014.

[26] "hmat-oss, a hierarchical matrix C/C++ library including a LU solver," https://github.com/jeromerobert/hmat-oss.

[27] P.-G. Martinsson, "Compressing rank-structured matrices via randomized sampling," 2015.