Parallel distributed out-of-core coupled solvers for large sparse/dense FEM/BEM linear systems implementing low-rank compression

Emmanuel Agullo, Marek Felšöci, Guillaume Sylvand

Parallel distributed out-of-core coupled solvers for large sparse/dense FEM/BEM linear systems implementing low-rank compression

Emmanuel Agullo, Marek Felšöci, Guillaume Sylvand

RESEARCH
REPORT
N° 9550
June 2024
Project-Team CONCACE

ISSN 0249-6399 ISRN INRIA/RR--9550--FR+ENG

Parallel distributed out-of-core coupled solvers for large sparse/dense FEM/BEM linear systems implementing low-rank compression

Emmanuel Agullo*, Marek Felšöci†, Guillaume Sylvand‡

Project-Team CONCACE

Research Report n° 9550 — June 2024 — 39 pages

* Centre Inria de l'Université de Bordeaux (emmanuel.agullo@inria.fr)
† Université de Strasbourg, CNRS, Inria, ICube, F-67000 Strasbourg, France (marek.felsoci@inria.fr)
‡ Airbus Central R&T (guillaume.sylvand@airbus.com)

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Abstract: Thanks to out-of-core computation and low-rank compression, sparse and dense direct solvers allow for solving ever larger linear systems. On the one hand, out-of-core techniques help to reduce the memory footprint of the computation by resorting to an extra storage, such as a hard or a solid state drive, in addition to RAM, also referred to as the core memory. The idea is to temporarily move out from the core memory the data that is not being used for the ongoing computation. On the other hand, low-rank matrices represent a way of storing matrices in a compressed form. Compression then helps to reduce not only the memory consumption of a linear system solution but also its computation time. Beyond the scope of a single multi-core shared-memory machine, parallel processing on multiple machines then may allow for solving even larger linear systems by distributing the load of data and computations among the machines. However, application of these techniques for solving coupled sparse/dense linear systems has been little investigated. In the aeronautic industry, such systems arise from acoustic simulations. Existing methods for solving sparse/dense linear systems we have found in the literature resort to distributed-memory parallelism in order to cope with the cost of the solution process in terms of computation time and memory usage but do not implement numerical compression or out-of-core computation. In [5], we introduced two classes of algorithms relying on low-rank compression, namely the multi-solve and the multi-factorization algorithms, for solving this kind of systems. Compared to a reference approach from the state-of-the-art, these algorithms allowed us to process significantly larger coupled systems within a single multi-core machine. However, to the best of our knowledge, none of the existing methods benefit from out-of-core computation, combine it with numerical compression or use these techniques in a distributed-memory environment. In this report, we propose a design of the multi-solve and the multi-factorization algorithms that, in addition to low-rank compression, implements out-of-core computation and distributed-memory parallelism with the aim to process even larger coupled sparse/dense systems. An experimental study conducted on up to 16 computation nodes, each having 48 cores and 180 GB of RAM, shows that the proposed algorithms, implemented on top of state-of-the-art sparse and dense direct solvers can respectively process systems of 42 million and 7 million total unknowns instead of 6 million unknowns with a standard sparse/dense solver coupling.

Key-words: sparse and dense matrices, large linear systems, direct method, parallel distributed solvers, low-rank compression, out-of-core computation, Finite Elements Method (FEM), Boundary Elements Method (BEM), FEM/BEM coupling

Solveurs couplés parallèles distribués et *out-of-core* pour de larges systèmes linéaires FEM/BEM creux/denses implémentant la compression de rang faible

Résumé : Des solveurs creux et denses modernes et bien équipés implémentent des fonctionnalités telles que le calcul *out-of-core* et la compression de rang faible afin de permettre le traitement de systèmes linéaires creux et denses toujours plus grands. D'un côté, grâce aux techniques *out-of-core*, l'on peut réduire l'empreinte mémoire du calcul en ayant recours à un support de stockage supplémentaire en plus de la mémoire vive que l'on appelle parfois aussi *the core memory* en anglais. L'idée est de déplacer temporairement de la mémoire vive (*out of the core memory*) les données qui ne sont pas indispensables au calcul en cours. D'un autre côté, les matrices de rang faible représentent un moyen de stockage de matrice sous une forme compressée. C'est une autre possibilité de réduire non seulement la consommation mémoire d'une résolution de système linéaire mais aussi son temps de calcul. Au-delà d'une seule machine multi-cœurs à mémoire partagée, le traitement en parallèle sur plusieurs machines permettrait de résoudre des systèmes linéaires encore plus grands en distribuant la charge de donnée et de calculs parmi les différentes machines. Cependant, l'application de ces techniques pour résoudre des systèmes linéaires couplés creux/denses n'a été que peu explorée. L'on retrouve ce genre de systèmes dans le contexte des simulations aéroacoustiques dans l'industrie aéronautique. Les approches de résolution existantes que nous avons trouvées dans la littérature reposent sur le parallélisme en mémoire distribuée pour réduire le coût du processus de résolution en termes de temps de calcul et d'utilisation de la mémoire mais elles n'implémentent ni la compression numérique ni le calcul *out-of-core*. Dans [5], nous avons introduit deux classes d'algorithmes faisant appel à des techniques de compression de rang faible, concrètement les algorithmes multi-solve et multi-factorisation, pour résoudre ce type de systèmes linéaires. Comparés à un couplage de solveurs de référence de l'état de l'art, ces algorithmes nous ont permis de traiter des systèmes couplés significativement plus grands sur une seule machine multi-cœurs. Dans ce rapport, nous proposons des algorithmes multi-solve et multi-factorisation qui, en plus de la compression de rang faible, implémentent les techniques *out-of-core* ainsi que le parallélisme en mémoire distribuée dans le but de pouvoir traiter des systèmes couplés encore plus grands. Une étude expérimentale effectuée sur jusqu'à 16 nœuds de calcul, chacun équipé de 48 cœurs et 180 Go de mémoire vive, montre que les algorithmes proposés, implémentés par-dessus des solveurs directs creux et denses de l'état de l'art, sont capables de traiter des systèmes respectivement de 42 millions et 7 millions d'inconnues au total au lieu de 6 millions avec un couplage de solveurs creux et denses standard.

Mots-clés : matrices creuses et denses, grands systèmes linéaires, méthode directe, solveurs parallèles distribués, compression de rang faible, calcul out-of-core, Méthode des éléments finis (FEM), Méthode des éléments finis de frontière (BEM), couplage FEM/BEM

Contents

1 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. In the aeronautical industry, such systems arise from acoustic models. In particular, these allow for simulating the noise produced at ground level by an aircraft during the takeoff and landing phases. The approach we adopt in this study considers the airflow around the aircraft as uniform in almost all the space (zone modeled by Boundary Elements Method – BEM) except for the jet of the engines where the flow is considered as non-uniform (zone modeled by volume Finite Elements Method – FEM). The resulting linear system couples FEM and BEM, and has both sparse parts (from FEM) and dense parts (from BEM). We seek to solve this system using a direct method while taking its singular composition into account. Due to the size of the objects (full aircrafts) and the acoustic frequencies simulated (up to 20 kHz, for audible frequencies) inducing edge sizes below 1 cm, the number of unknowns can be extremely high, which makes the solution of the system a computational challenge.

Existing methods for solving sparse/dense linear systems we have found in the litterature [10, 14, 20, 13, 18] resort to distributed-memory computation in order to cope with the cost of a direct solution in terms of computation time and memory usage. However, one can also take advantage of advanced techniques such as numerical compression and out-of-core computation (storing currently unused data on disk – out of the core memory). In [5], we proposed two classes of algorithms relying on numerical compression, namely the multi-solve and multi-factorization algorithms, allowing us to solve relatively large coupled FEM/BEM systems on a shared-memory workstation. To the best of our knowledge, no approaches benefit from out-of-core computation, combine it with numerical compression or use these techniques in a distributed-memory environment. In this work, we propose a design of the multi-solve and the multi-factorization algorithms that, beyond numerical compression, allows for out-of-core computation and their usage in distributed memory with the aim to process ever larger coupled systems and extend the spectrum of acoustic frequencies we can simulate on existing computers.

The rest of the report is organized as follows. In Section 2, we further introduce the coupled FEM/BEM systems, discuss solution methods and the related work. We then present the multi-solve and the multi-factorization algorithms from [5] we rely on. In Section 3, we detail the design of multi-solve and multi-factorization implementing out-of-core computation and distributed-memory parallelism. We outline an experimental study of the proposed algorithms in Section 4. In Section 5, we analyze the impact of out-of-core in shared memory. In Section 6, we evaluate the algorithms in distributed memory and show how large coupled systems we can solve on a given number of computation nodes. In Section 7, we discuss the perspectives of this work.

2 Background

2.1 Coupled FEM/BEM systems in aeroacoustics

The sparse/dense systems we seek to solve result from the coupling of the volume Finite Element Method (FEM) [12] and the Boundary Element Method (BEM) [19] used to simulate the propagation of acoustic waves around aircrafts (see Fig. 2, left). In the jet flow created by the reactors, the propagation media (the air) is highly heterogeneous in terms of temperature, density, flow, etc. Hence we need a FEM approach to compute acoustic waves propagation in it. Elsewhere, we approximate the media as homogeneous and the flow as uniform, and use BEM to compute the waves propagation. This leads [9] to the 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 and the outer surface of v (see

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

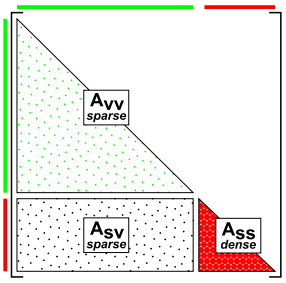


FIG. 1: Structure of A in (1).

Fig. 2, right). Then, A in (1) is a 2×2 block coefficient matrix, where 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, A_{sv} is a sparse submatrix representing the action of the volume part on the exterior surface (see Fig. 1). Within this study, we mostly consider systems where A is symmetric. However, we also deal with a specific industrial situation where A is non-symmetric.

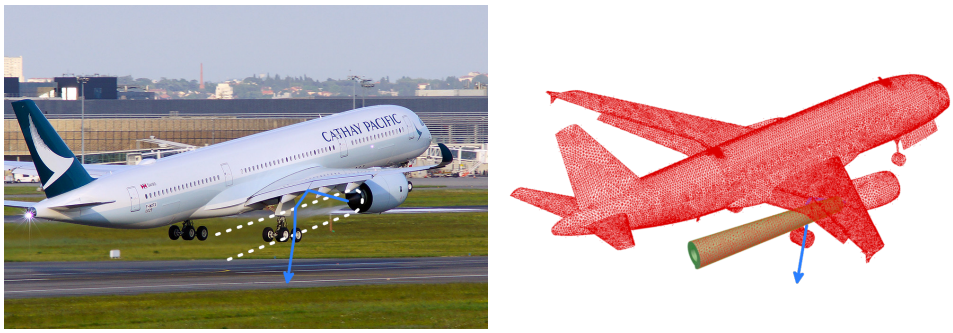


FIG. 2: An acoustic wave (blue arrow) emitted by the aircraft's engine, reflected on the wing and crossing the jet flow. Real-life case (left, original photo by David Barrie) and a numerical model example (right). The red surface mesh covers the aircraft's surface and the surface of the green volume mesh of the jet flow.

2.2 Direct solution of coupled FEM/BEM systems

In this work, we choose to solve (1) with a direct method. In a sparse context, direct methods are known to possibly consume a lot of memory due to a phenomenon referred to as *fill-in* [11] (zeros of the original matrix A become non-zeros in factorized matrix). On the other hand, when they fit in memory, they are extremely robust from a numerical point of view and represent a must-have in an industrial solution where they are commonly employed to solve moderate to relatively large problems.

Most of the state-of-the-art sparse and dense solvers provide building blocks we can directly apply on the three submatrices of A in (1) and compose a coupled sparse/dense solver. We refer to these straightforward solver couplings as to vanilla couplings. However, with growing size of the coupled linear system and especially of the dense part, vanilla solver couplings may quickly run out of memory. To cope with this, we can resort to advanced functionalities implemented in fully-featured direct solvers. For instance, considering the scope of a single shared-memory workstation, numerical compression [15, 6] can be used to reduce the memory footprint as well as the time of computations. Thanks to out-of-core techniques (moving currently unused data to disk), one can further reduce the memory consumption. Parallel computation in distributed memory then allow for solving even larger problems by dividing the load of data and computations

among multiple machines. While individual building blocks of the sparse and the dense direct solvers can benefit from these advanced features, it is not trivial on the articulation between sparse and dense operations. The Application Program Interface (API) of the solvers was simply not designed for this kind of usage. Indeed, no matter the choice of building blocks or advanced features, the intermediate result passed from the sparse to the dense solver is in the form of a non-compressed dense matrix entirely stored in memory. This matrix is referred to as Schur complement and has the size of the dense block A_{ss} in (1), which is still a major drawback for vanilla solver couplings. Due to their dimension (the size of A_{ss} in particular), the problems we are interested in cannot be processed using a vanilla sparse and dense solver coupling.

2.3 Related work

Multiple approaches to solve sparse/dense linear systems based on direct methods have been investigated in the literature [10, 14, 20, 13, 18]. To cope with the cost of the solution in terms of computation time and memory usage, these apporaches resort to distributed-memory computation either only in the dense part of the system or in both sparse and dense parts. However, in the cited papers, the tackled problem size remains relatively small, which makes it possible to handle the system through a vanilla coupling of the state-of-the-art sparse and dense direct solvers as seen above. In [5], we introduced two classes of algorithms, the multi-solve and multi-factorization algorithms, allowing us to solve relatively large coupled FEM/BEM systems on a shared-memory workstation thanks to numerical compression despite the obstacles in composing the APIs of the sparse and the dense direct solvers. Nevertheless, to the best of our knowledge, none of the available approaches benefit from out-of-core computation, combine it with numerical compression or use these techniques in a distributed-memory environment. In this work, we propose multi-solve and multi-factorization algorithms that, in addition to numerical compression, implement out-of-core computation and allow for their usage in distributed memory.

2.4 Multi-solve and multi-factorization algorithms

The common principle of the multi-solve and multi-factorization algorithms consists in composing existing parallel sparse and dense methods on well-chosen submatrices so as to deal with the limitations of the vanilla solver couplings (see Section 2.2). In this section, we present the main algorithmic steps of both multi-solve and multi-factorization. The goal is not to describe them in details (see [5] for that) but to provide the reader with a high-level view of the steps and their nature (such as whether they involve dense, sparse or compressed computation). In sections 3.1 and 3.2, we then propose the design, which makes it possible to perform these steps out-of-core and in distributed memory. The core phase of both multi-solve and multi-factorization is the assembly of the dense Schur complement matrix $S = A_{ss} - A_{sv}A_{vv}^{-1}A_{sv}^T$ associated with the A_{ss} block.

2.4.1 Multi-solve algorithm

Most sparse direct solvers do not provide an API to handle coupled sparse/dense systems and can process exclusively sparse systems. The multi-solve approach accommodates with this constraint by delegating only the A_{vv} block to the sparse direct solver. Using the latter, the A_{vv} block is factorized through a so-called *sparse factorization* into $L_{vv}L_{vv}^T$. The A_{ss} block is handled by the dense direct solver. Because this block may not fully fit in memory, it is split into multiple vertical blocks (see Fig. 3, left) which are assembled one by one, all the processing units tackling the same block i at the same time. To compute such a block S_i of A_{ss}, a block $A_{sv_i}^T$ is first processed through a *sparse solve* step of the sparse direct solver, yielding a dense temporary

block $Y_i = (L_{vv}L_{vv}^T)^{-1}A_{sv_i}^T$. The latter is multiplied by the sparse A_{sv} block resulting in another temporary dense block $Z_i = A_{sv}Y_i$. Finally, we perform the assembly $(A_{ss_i} - Z_i)$ to produce the dense S_i.

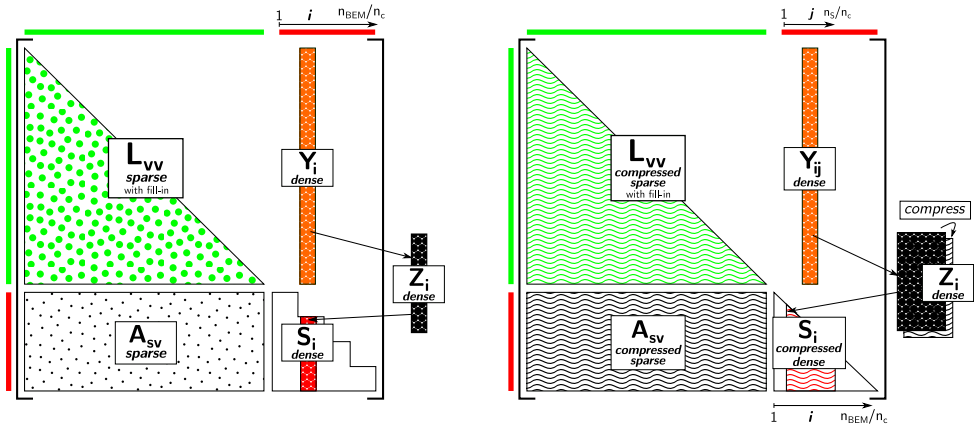


FIG. 3: Block-wise computation of S using the *baseline multi-solve* (left) and the *compressed Schur multi-solve* (right) algorithms.

In the *baseline multi-solve* case, the block S_i is kept in a non-compressed dense form. Conversely, in the *compressed Schur multi-solve* variant, the final assembly step becomes a compressed (through hierarchically low-rank techniques) assembly $A_{ss_i} - \texttt{Compress}(Z_i)$. Note that A_{ss_i} is initially compressed, but this operation implies a recompression of the block at each iteration of the loop on i. In order to reduce the compression cost, this variant allows for computing multiple (typically 2 to 4 in the experiments below) temporary dense blocks $Z_{ij} = A_{sv}Y_{ij}$ eventually forming a larger Z_i block leading to less frequent compressed assemblies of S_i (see Fig. 3, right).

2.4.2 Multi-factorization algorithm

The multi-factorization algorithm is based on a more advanced usage of sparse direct methods consisting in delegating also the management of the dense A_{ss} block to the sparse direct solver. Only supported by a few fully-featured sparse direct solvers, this functionality (referred to as *sparse factorization+Schur*) has the advantage of efficiently handling off-diagonal blocks thanks to the advanced combinatorial (such as management of the fill-in), numerical (such as low-rank compression) and computational (such as level-3 BLAS usage) features of modern sparse direct solvers when processing the off-diagonal A_{sv}^T and A_{sv} sparse-dense coupling parts (see [5] for more details). The Schur complement S in the *baseline multi-factorization* algorithm is computed by square blocks. Computing a block S_{ij} (see Fig. 4, left) amounts to form a temporary submatrix W from A_{vv}, A_{sv_i} and $A_{sv_j}^T$ and call a *sparse factorization+Schur* step on W performed by the sparse direct solver. This call returns the Schur complement block $X_{ij} = -A_{sv_i}(L_{vv}U_{vv})^{-1}A_{sv_j}^T$ associated with W. When $i = j$, W do not have to contain the upper part of A_{vv} nor $A_{sv_j}^T$ which is implicitly known from A_{sv_i}. In this case, we can rely on a symmetric mode of the direct solver and the *sparse factorization+Schur* step returns $X_{ij} = -A_{sv_i}(L_{vv}L_{vv}^T)^{-1}A_{sv_j}^T$. To compute S_{ij}, we perform a final assembly $(A_{ss_{ij}} + X_{ij})$. Current API of modern sparse direct solvers (see extended discussion in [5]) implies a re-factorization of A_{vv} in W within each of the *sparse factorization+Schur* steps, although it does not change during the computation. Furthermore, the API only allows us to retrieve the Schur complement itself as a non-compressed dense matrix, even if compression occurs within the rest of processing.

In the *compressed Schur multi-factorization* variant (see Fig. 4, right), we compress the X_{ij}

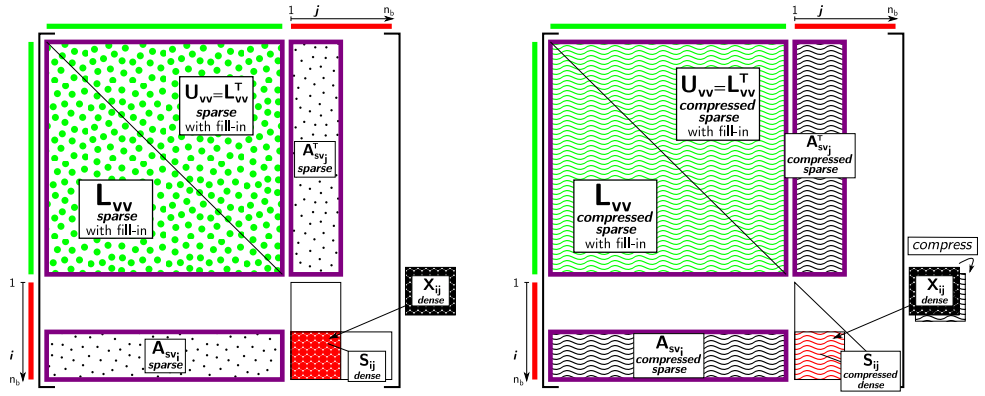


FIG. 4: Block-wise computation of S using the *baseline multi-factorization* (left) and the *compressed Schur multi-factorization* (right) algorithms.

Schur block into a temporary compressed matrix as soon as the sparse solver returns it. Hence, the final assembly step becomes a compressed assembly $A_{ss_{ij}} \leftarrow A_{ss_{ij}} + \texttt{Compress}(X_{ij})$. Like in the case of *compressed Schur multi-solve*, this operation also implies a recompression of the initally compressed $A_{ss_{ij}}$.

3 Out-of-core computation and distributed-memory parallelism

In [5], we showed that thanks to numerical compression, the multi-solve and multi-factorization algorithms (see Section 2.4) allow us to solve relatively large coupled sparse/dense FEM/BEM systems (see Section 2.2) on a shared-memory multi-core workstations. An alternative way of coping with the high memory footprint of the algorithms amounts to using a disk or an SSD as extra storage space in addition to RAM (Random Access Memory), which is also referred to as the *core memory*. This technique is called out-of-core computation. Indeed, the principle of an out-of-core algorithm is to temporarily store the data that is not being used by the ongoing computation out of the core memory, e.g. on disk. On the contrary, we say of the algorithms that operate with data in RAM that they are *in-core* algorithms. Once we have reached the limits of a given workstation, we can distribute data and workload across multiple workstations, e.g. within a high-performance computing cluster.

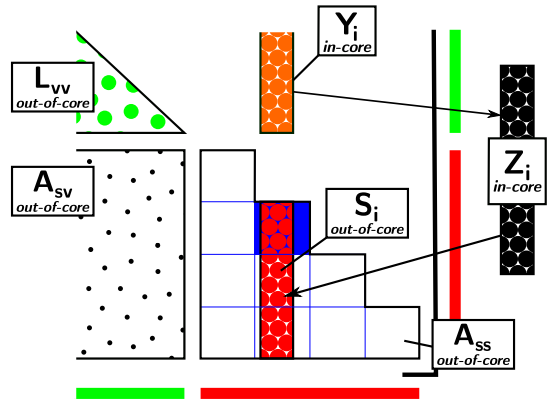
We thus propose a design of the multi-solve and multi-factorization algorithms taking advantage of out-of-core computation in Section 3.1 and distributed-memory parallelism in Section 3.2. Similarly to numerical compression, the direct solvers already implement these features within the individual *sparse factorization*, *sparse solve*, *sparse factorization+Schur*, *dense factorization* and *dense solve* building blocks. However, to efficiently take advantage of them on the articulation between sparse and dense operations, an extra algorithmic effort is required to deal with the current API of the direct solvers as well as the differences between their data structures.

3.1 Design of out-of-core algorithms in shared memory

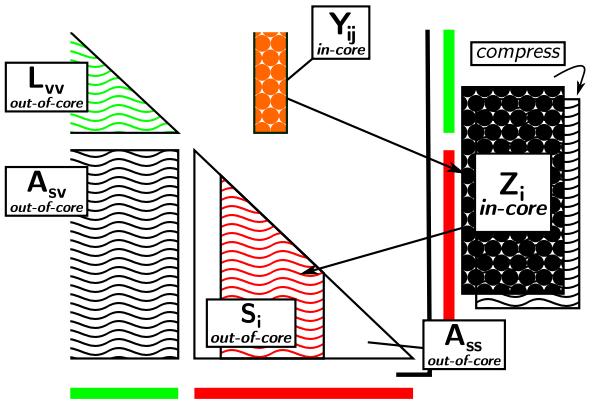
3.1.1 Multi-solve algorithm

In the multi-solve algorithm, the result Y_i of the *sparse solve* step $(L_{vv}L_{vv}^T)^{-1}A_{sv_i}^T$ can only be retrieved in RAM, i.e. in-core. The block Z_i is also stored in-core. In *baseline multi-solve*, A_{ss} is split into square out-of-core blocks (see Fig. 5a). Then, when assembling S_i, we load data into memory from only one out-of-core block at a time. Furthermore, as the dimension of out-of-core

blocks usually does not match the dimension of S_i blocks, we load into memory only the portion of the out-of-core block of A_{ss} which is overlayed by the current S_i block. Regarding *compressed Schur multi-solve* (Fig. 5b), the out-of-core feature of the dense direct solver is dynamically and transparently handled by the runtime on top of which the solver is implemented. The runtime starts to eject data of compressed Z_i and A_{ss} to disk when an arbitrary chosen memory limit is reached during the computation. In the experiments below, the dense solver is instrumented to use not more than 4% of available RAM. The remaining is left for the sparse direct solver and other data.



(a) *baseline multi-solve*, blue lines depict the partitioning of A_{ss} into out-of-core blocks.

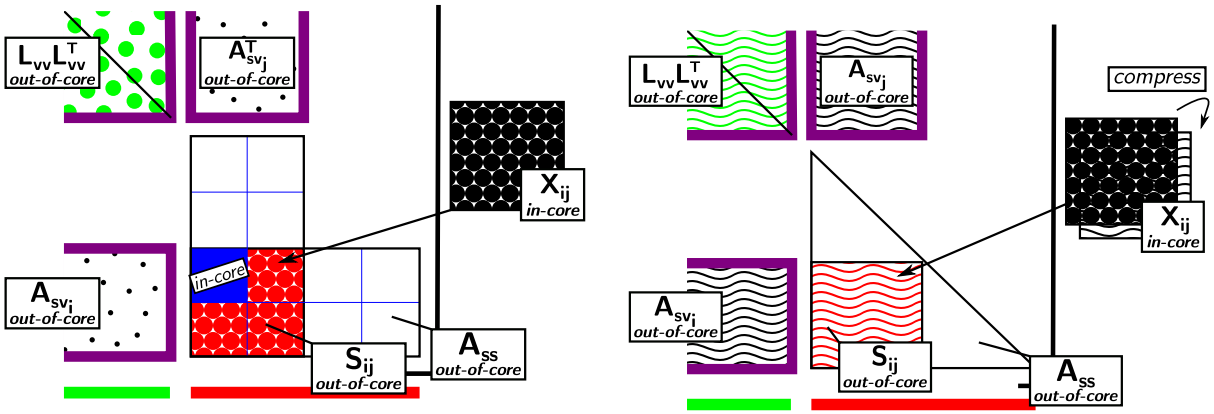


(b) *compressed Schur multi-solve*.

FIG. 5: Out-of-core block-wise computation of S using the multi-solve algorithm (zoom on the Schur complement dense part of the system).

3.1.2 Multi-factorization algorithm

In the multi-factorization algorithm, the Schur complement block X_{ij} (see Fig. 6) associated with the temporary submatrix W is returned entirely assembled in RAM (in-core). Just like for *baseline multi-solve*, in the *baseline multi-factorization* case, A_{ss} is split into square out-of-core blocks (see Fig. 6a). However, here the dimension of the Schur blocks X_{ij} and S_{ij} is a multiple of the out-of-core block dimension. Then, when assembling S_{ij}, we load only one out-of-core block at a time into RAM. Regarding the *compressed Schur multi-factorization* (see Fig. 6b), the out-of-core feature of the dense direct solver, concerning the compressed X_{ij} and A_{ss}, is dynamically and transparently handled by the underlying runtime as it is for *compressed Schur multi-solve* (see Section 3.1.1).



(a) *baseline multi-factorization*, blue lines depict the partitioning of A_{ss} into out-of-core blocks.

(b) *compressed Schur multi-factorization.*

FIG. 6: Out-of-core block-wise computation of S using the multi-factorization algorithm (zoom on the Schur complement dense part of the system).

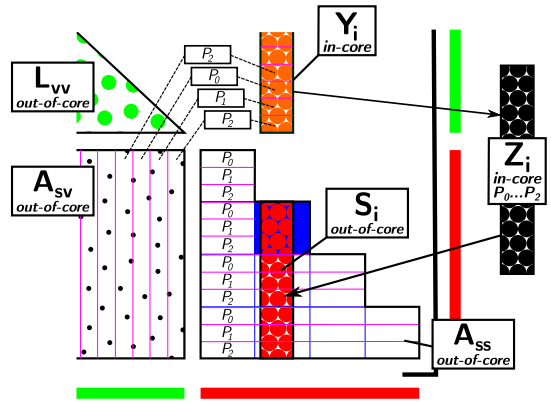
3.2 Extension to distributed memory

3.2.1 Multi-solve algorithm

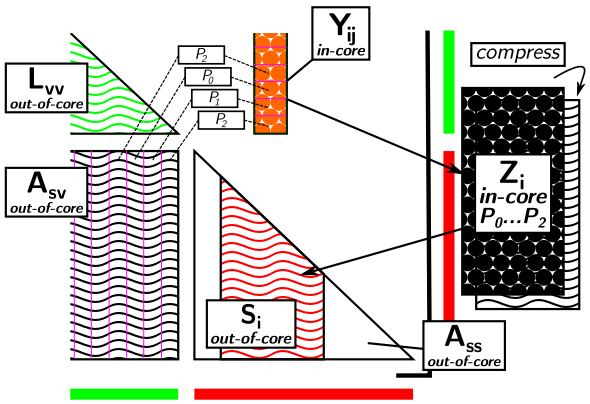
In multi-solve, the *sparse factorization* of A_{vv} as well as the *dense factorization* of the Schur complement S, followed by the solve operations to determine x_s and x_v, can be transparently done in parallel in distributed memory by the sparse and the dense direct solvers. Indeed, the corresponding building blocks take care of distributing data and computations. However, in order to be able to perform the multiplication $A_{sv}Y_i$ leading to Z_i, the columns of A_{sv} must be sorted so as on a given process P_k, the indices of local columns of A_{sv} match those of local rows of Y_i (see Fig. 7). After the multiplication, values from all processes are combined to compute the resulting Z_i which is then sent back to all processes. In the case of *baseline multi-solve*, each out-of-core block of A_{ss} (see Section 3.1.1) is distributed among parallel processes by packs of rows in a cyclic manner. This way, during the final assembly of S_i, we process one parallel out-of-core block at a time. Note that when the out-of-core feature is turned off, all the blocks are kept in memory but the partitioning and the distribution remain the same. As of *compressed Schur multi-solve*, the distribution of the compressed Z_i and A_{ss} is handled transparently by the dense direct solver through the underlying runtime.

3.2.2 Multi-factorization algorithm

In multi-factorization, the *sparse factorization+Schur* steps on the temporary W matrices as well as the *dense factorization* of the Schur complement S, followed by the solve operations to determine x_s and x_v, can be transparently done in parallel in distributed memory by the sparse and the dense direct solvers. Nevertheless, in the *baseline multi-solve* variant, X_{ij} is obtained distributed across parallel processes in a cyclic manner by packs of rows. We choose the size of the latter so as to match the distribution of the out-of-core blocks (see Section 3.1.2) of A_{ss} (see Fig. 8a). As in the case of *baseline multi-solve* (see Section 3.2.1), during the final assembly of S_{ij}, we process one parallel out-of-core block at a time. The algorithm behaves the same way when out-of-core is turned off. Then, all the blocks are simply kept in memory. As of *compressed Schur multi-factorization* (see Fig. 8b), the distribution of A_{ss} is handled transparently by the dense direct solver through the underlying runtime. Moreover, due to the specificities of the hierarchical low-rank compression technique [17], data distribution in the compressed submatrix A_{ss} is more complex and incompatible with data distribution in the dense non-compressed submatrix X_{ij}



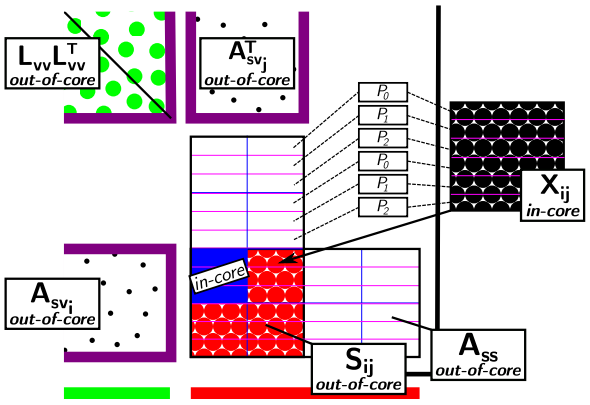
(a) *baseline multi-solve*, blue lines depict the partitioning of A_{ss} into out-of-core blocks and pink lines represent the distribution of A_{sv}, A_{ss} and Y_i among parallel processes.



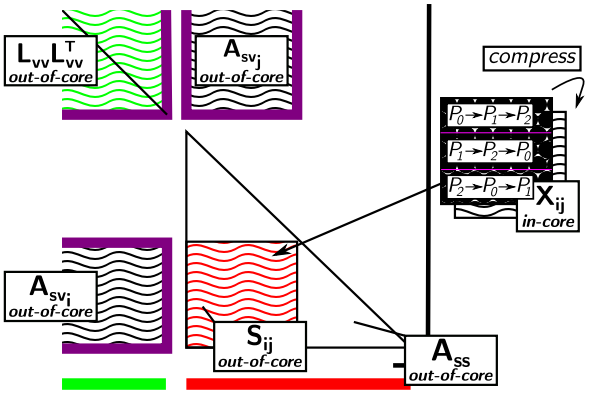
(b) *compressed Schur multi-solve*, pink lines represent the distribution of A_{sv} and Y_{ij} among parallel processes.

FIG. 7: Out-of-core parallel distributed (assuming three parallel processes P_0, P_1 and P_2) block-wise computation of S using the multi-solve algorithm (zoom on the Schur complement dense part of the system). Only parts of L_{vv}, A_{sv}, Y_i and Y_{ij} are represented.

following a standard 2D-block cyclic scheme [8]. Consequently, during the assembly of the block S_{ij}, one process may require a portion of the non-compressed X_{ij} located on another process. To accommodate with this constraint, we propose a communication scheme based on a virtual ring topology for the processes to share their local portions of X_{ij} so as to ensure that each process receives each of the local portions of X_{ij} one at a time without having to store a copy of the entire X_{ij} on each process. Fig. 9 illustrates this communication scheme on an example assuming three parallel processes P_0, P_1 and P_2. This leads to a three-step assembly of a tile S_{ij} based on a non-compressed tile X_{ij} split into three packs of rows $X_{ij\alpha}$, $X_{ij\beta}$ and $X_{ij\gamma}$ distributed among all processes. At first, P_0 assembles the local part of S_{ij} for the locally available pack of rows $X_{ij\alpha}$. Then, it sends the latter to its immediate neighbor P_1. Meanwhile, it receives $X_{ij\gamma}$ from P_2 and continues the assembly of the local part of S_{ij}. In the last step, P_0 receives $X_{ij\beta}$ from P_2 (which received it from P_1) and completes the assembly of its local part of S_{ij}. The remaining parts of S_{ij} are assembled analogically by P_1 and P_2.



(a) *baseline multi-factorization*, blue lines depict the partitioning of A_{ss} into out-of-core blocks and pink lines represent the distribution of A_{sv}, A_{ss} and X_{ij} among parallel processes.



(b) *compressed Schur multi-factorization*, pink lines represent the distribution of the non-compressed X_{ij} among parallel processes.

FIG. 8: Out-of-core parallel distributed (assuming three parallel processes P_0, P_1 and P_2) block-wise computation of S using the multi-factorization algorithm (zoom on the Schur complement dense part of the system). Only parts of $L_{vv}L_{vv}^T$, A_{sv_i} and $A_{sv_j}^T$ are represented.

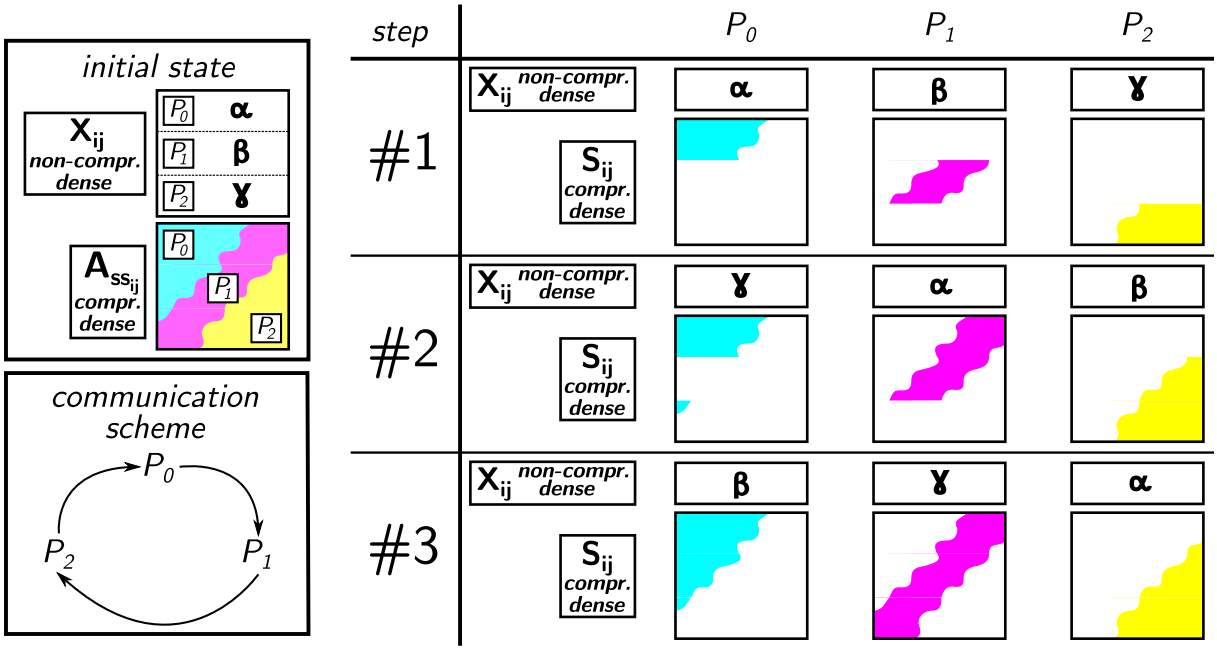


FIG. 9: Parallel distributed assembly of a Schur complement tile S_{ij} by the *compressed Schur multi-factorization* algorithm (assuming three parallel processes P_0, P_1 and P_2).

4 Experimental setup

We conducted an experimental study of the previously discussed multi-solve and multi-factorization algorithms for solving large coupled sparse/dense FEM/BEM linear systems such as defined in (1) in order to evaluate the impact of out-of-core computation and distributed-memory parallelism on their performance. As a reference from the state-of-the-art, we consider a vanilla solver coupling (see Section 2.2) based on the advanced *sparse factorization+Schur* building block of the sparse solver (see Section 2.4.2). From the implementation point of view, we rely on the *baseline multi-factorization* algorithm (see Section 2.4.2) with the MUMPS/SPIDO coupling and n_b set to 1 (see below). In the rest of this document, we refer to this coupling as to advanced vanilla solver coupling.

The multi-solve and multi-factorization algorithms (see Section 2.4) are implemented on top of the coupling of the sparse direct solver MUMPS [7] with either the proprietary scalapack-like dense direct solver SPIDO (for the *baseline* variants) or the hierarchical low-rank $\mathcal{H}$-matrix compressed solver HMAT [17] (for the *compressed* variants). In the following, 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 for controlling the accuracy of the resulting solution approximations.

For the benchmarks, we rely on both industrial and academic test cases. Fig. 10 shows the industrial test case. It features 2,090,638 volume unkonwns and 168,830 surface unknowns.

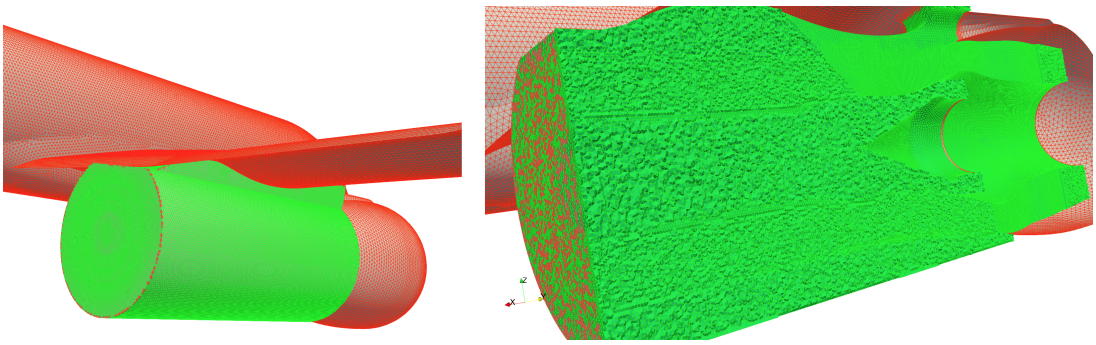


FIG. 10: Industrial test case with BEM surface mesh in red (the right part of the plane, the wing and the engine) and FEM volume mesh in green (the jet flow). On the right, a vertical cut-plane allows to see the inside of the reactor and the flow: the green mesh is made of tetrahedra, while the red mesh is hollow, and made of triangles.

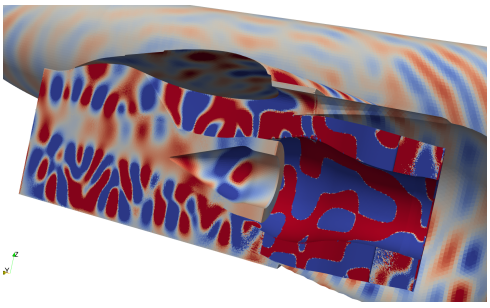


FIG. 11: Industrial test case result.

An example of physical result is presented in Fig. 11 showing the acoustic pressure in the flow (at the front) and on the plane's surface (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. Fig. 12 then illustrates the *wide pipe* academic test case yielding linear systems close enough to those arising from real-life models (see Fig. 10) while relying on a publicly available reproducible example (`https://gitlab.inria.fr/solverstack/test_fembem`).

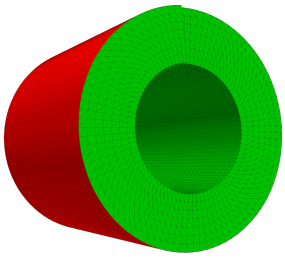


FIG. 12: *wide pipe* (4×2m), FEM volume mesh v, BEM surface mesh s.

We highlight that the proportion of surface unknowns in the industrial test case is higher than in the *wide pipe* where 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 part in the industrial case will be higher and numerical compression as well as out-of-core techniques may have stronger impact on performance.

Note that sections 5 and 6 present the key results of our experimental evaluation. However, further benchmarks results and analysis are provided in the Appendix of this report. We refer to these throughout the rest of the document essentially to support the conclusions we draw.

5 Study of out-of-core in shared memory

In this first part of the study, the goal is to determine whether out-of-core techniques may help us to lower the memory footprint of the two-stage algorithms on a shared-memory workstation so as to allow for processing even larger coupled systems and if so, to which extent. Here, we consider the real-life industrial test case processed at Airbus Central R&T.

The benchmarks in this section were carried on the 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 (Hyper-Threading is turned off) 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. The precision parameter ϵ is set to 10^{-4} (see [5] for more details).

5.1 No numerical compression

In order to evaluate the sole effect of out-of-core computation, we began by performin benchmarks without compression neither in the sparse nor in the dense solver. The corresponding results are provided in Table 1. In the case of multi-solve, when we activate out-of-core in the dense solver (row 2), it allows us to considerably decrease the RAM usage from 255 to 52 GiB for an overhead of only 4% in computation time. When the sparse solver is out-of-core as well (row 3), we can further reduce the memory consumption but the associated slow-down becomes excessively high, i.e. 380%. This is due to the fact that the sparse solver reads the factors of A_{vv} from disk on each call to the *sparse solve* building block during the Schur complement assembly. As the compression is deactivated, this represent large amounts of data. Without out-of-core, multi-factorization can only run with n_b set to 12 blocks (row 4) which leads to 1 day of execution time

compared to 18.1 hours for multi-solve. Activating out-of-core in the dense and then also in the sparse solver allows us to lower n_b to 3 blocks, reduce the number of re-factorizations from 78 (with n_b set to 12) to 6 and process the linear system in 12.8 hours.

	Algorithm	Dense *ooc*	Sparse *ooc*	n_b	RAM (GiB)	Disk (GiB)	Time
1	multi-solve			N/A	255	25	18.1h
2	multi-solve (§3.1.1)	x		N/A	52	229	18.8h
3	multi-solve (§3.1.1)	x	x	N/A	12	265	3.6d
4	multi-facto.			**12**	377	20	**1d**
5	multi-facto. (§3.1.2)	x		12	207	223	1.4d
6	multi-facto. (§3.1.2)	x		6	227	223	20h
7	multi-facto. (§3.1.2)	x	x	12	55	398	1.6d
8	multi-facto. (§3.1.2)	x	x	6	68	408	22.4h
9	multi-facto. (§3.1.2)	x	x	**3**	115	423	**12.8h**

TABLE 1: Performance of the two-stage algorithms on the industrial test case without compression and with out-of-core (*ooc*) optionally enabled.

5.2 Numerical compression in both sparse and dense solvers

Then, we activated the numerical compression in both the sparse and the dense solvers and observe the impact of out-of-core. The results are in Table 2. Regarding multi-solve, the compression in all the parts of the system brings an important gain in time to solution and RAM usage already (row 1). Moreover, activating out-of-core leads to only a limited decrease in memory consumption but to an important slow-down which may go as high as 213% (row 3). Similar observations come out of the analysis of *compressed Schur multi-factorization*. In this case, the runs without out-of-core seem to be the best-performing ones in terms of the computation time. With n_b set to 3 (row 6), we were able to perform the computation in only 50 minutes. For the benchmarks resorting to out-of-core in rows 9 and 12 the memory consumption is even slightly higher than without it (row 6). So far, we have not found the reason for this increase. A deeper investigation is required to explain the phenomenon.

	Algorithm	Dense *ooc*	Sparse *ooc*	n_b	RAM (GiB)	Disk (GiB)	Time
1	multi-solve			N/A	35	4	9.3h
2	multi-solve (§3.1.1)	x		N/A	34	11	10.4h
3	multi-solve (§3.1.1)	x	x	N/A	22	17	29.1h
4	multi-facto.			12	81	5	2.1h
5	multi-facto.			6	122	5	1.1h
6	multi-facto.			3	142	5	50m
7	multi-facto. (§3.1.2)	x		12	89	11	3h
8	multi-facto. (§3.1.2)	x		6	113	11	1.2h
9	multi-facto. (§3.1.2)	x		3	162	11	53m
10	multi-facto. (§3.1.2)	x	x	12	75	17	2.5h
11	multi-facto. (§3.1.2)	x	x	6	103	17	1.2h
12	multi-facto. (§3.1.2)	x	x	3	150	15	52m

TABLE 2: Performance of the two-stage algorithms on the industrial test case with compression in both the sparse and the dense solvers and with out-of-core (*ooc*) optionally enabled.

5.3 Conclusion

When both the sparse and the dense solvers resort to numerical compression, the effect of out-of-core on the computation time and the RAM consumption of the two-stage algorithms is limited thanks to the efficiency of numerical compression. On the other hand, in a framework where numerical compression is not available or not desirable, out-of-core computation allows us to significantly lower the memory footprint of the algorithms, especially when applied to the dense Schur complement part. The counterpart is a longer computation time associated to the usage of a slower type of storage, i.e. a hard disk. In multi-solve, the overhead is mainly associated to the activation of out-of-core in the sparse solver due to the frequent calls to the *sparse solve* building block. However, in multi-factorization, without numerical compression, out-of-core allowed us to perform considerably less calls ($n_b = 3$) to the *sparse factorization+Schur* step compared to in-core computation ($n_b = 12$). In this case, the performance advantage of using out-of-core in terms of computation time was much higher than the associated overhead. Interestingly, in such cases, the out-of-core multi-factorization is faster then both its in-core counterpart and multi-solve.

The Appendix A of this report features an extended experimental study on academic test cases of varying size and proportion of unknowns in the dense part with respect to the sparse part of the underlying coupled linear system. According to these results, the higher the proportion of unknowns in the dense part, the more important the positive effect of out-of-core computation on the memory consumption. This is especially the case for the industrial test case we relied on in this section. The results further show that, on a given shared-memory multi-core machine, we can process problems larger than both the reference advanced vanilla coupling and the multi-solve and multi-factorization algorithms themselves without resorting to out-of-core. Moreover, the additional experiments confirm that the overhead in multi-solve related to the use of out-of-core in the sparse solver can be reduced using higher values of n_c.

6 Study in distributed memory

The second part of the experimental evaluation is dedicated to the study of the proposed algorithms in distributed memory. The distributed-memory parallelism is implemented using the OpenMPI communication library (`https://www.open-mpi.org/`) based on the MPI standard [16].

In Section 6.1, the goal is to know how to run the algorithms efficiently with a given amount of memory. To a certain extent, this method may be related to the so-called memory-aware approaches [1, 2]. We indeed first consider problems that may be processed on a single node to see whether the distributed multi-solve and multi-factorization schemes can take advantage of the available memory and computational resources to increase their performance. In Section 6.2, we then consider both out-of-core computation and distributed-memory parallelism and try to reach the limits of the algorithms on a given number of computational nodes.

Here, the benchmarks rely on the *wide pipe* academic test case and were carried on the `skylake` nodes of the TGCC Joliot-Curie platform (`https://www-hpc.cea.fr/en/Joliot-Curie.html`). A `skylake` node has a total of 48 processor cores running each at 2.7 GHz (Hyper-Threading is not used), 180 GiB of RAM and access to a 300 GB/s Lustre network storage. The solver test suite `test_FEMBEM` for the *wide pipe* case is compiled with GNU C Compiler (gcc) 8.3.0, Open-MPI 4.1.4, StarPU 1.3.8, Intel(R) MKL library 19.0.5.281, and MUMPS 5.5.1. The precision parameter ϵ is set to 10^{-3} (see [5] for more details). Also, we systematically turn on low-rank compression in the sparse solver MUMPS for both MUMPS/SPIDO and MUMPS/HMAT for all the experiments in this section.

6.1 Performance-memory trade-off

6.1.1 Multi-solve

We consider coupled FEM/BEM linear systems with the total unknown count N equal to 1,000,000, 3,000,000 and 7,000,000. We run benchmarks on 1 to 16 nodes, i.e. using 48 to 768 cores.

In the first place, we focus on the *baseline multi-solve* relying on the MUMPS/SPIDO coupling. We vary the size n_c of block $A_{sv_i}^T$ of columns of the A_{sv}^T submatrix in between 128 and 4,096. Fig. 13 shows the computation time of the MUMPS/SPIDO coupling in function of per-node RAM usage peaks. Regarding the n_c parameter, we observe analogous behavior like in the case of the experiments in shared memory [5]. Sufficiently high n_c gives the sparse solver enough right-hand sides to process in parallel during the *sparse solve* step $A_{vv}^{-1}A_{sv}^T$. Additional experimental results in the Appendix of this report (see Fig. 24 therein) show that this leads to an important improvement in computation time and parallel efficiency. However, as the outcome of the operation is a dense non-compressed matrix, too high values for n_c can result in a dramatic increase in RAM consumption. Moreover, ever higher n_c does not translate into a proportional performance improvement. It may even lead to a degradation. For example, with $N = 7{,}000{,}000$ and 8 computational nodes, going from $n_c = 1{,}024$ to $n_c = 4{,}096$ shortens the computation time by 14% but more than doubles the memory footprint. There is no large gap between the parallel efficiency of these two cases either. Then, with $N = 3{,}000{,}000$ and 4 to 8 nodes, increasing n_c from 2,048 to 4,096 even worsens the computation time in addition to a higher RAM consumption. In the end, the optimal value of n_c then depends on the size of the problem and the number of parallel processing units used for the computation.

From a more general point of view, with increasing number of nodes, we can reduce both the memory footprint as well as the computation time of the multi-solve algorithm. This is crucial for our effort to process ever larger coupled FEM/BEM systems. Here, we limited ourselves to at most 7,000,000 of unknowns. For systems of this size, 4 to 8 computational nodes seem to deliver satisfying efficiency both in terms of time and RAM consumption. Pushing up to 16 nodes leads to a relatively small improvement. However, in Section 6.2, we then demonstrate the ability of the algorithm to take advantage of the available memory and computational power of the 16 nodes to process considerably larger coupled systems.

For the *compressed Schur multi-solve*, based on MUMPS/HMAT, the size of blocks of columns of S and A_{sv}^T is handled by two different parameters, n_S and n_c, respectively. For the experiments in shared memory [5], the optimal value of n_c for the MUMPS/HMAT coupling seemed to be 256 for a single problem with 2,000,000 unknowns. Here, we consider systems of varying size and on varying number of nodes. The best n_c setting will therefore not be the same for every case. For the evaluation of multi-solve using the MUMPS/HMAT coupling, we therefore set n_S to multiples (1×, 2× or 4×) of n_c instead of considering a fixed value. For example, according to the results in Fig. 13, the optimal n_c for the problem with 3,000,000 of unknowns running on 8 nodes seems to be 2,048. For the equivalent MUMPS/HMAT benchmark, we thus set n_c to 2,048 and n_S to 2,048 ($1\times n_c$), 4,096 ($2\times n_c$) or 8,192 ($4\times n_c$).

Fig. 14 then compares the computation times and the maximum RAM usage peaks of multi-solve using MUMPS/HMAT with varying n_S to the corresponding best-performing runs relying on MUMPS/SPIDO. At first, we focus on the impact of the varying n_S parameter. The computation times between different values of n_S present only minor differences. Like in shared memory [5], high enough n_S reduces the overhead of compressing the dense Z_i blocks as well as of the corresponding compressed AXPY $S_i = A_{ss_i} - Z_i$ during the Schur complement assembly. The additional parallel efficiency results in the Appendix of this report (see Fig. 25 therein) further corroborate this observation.

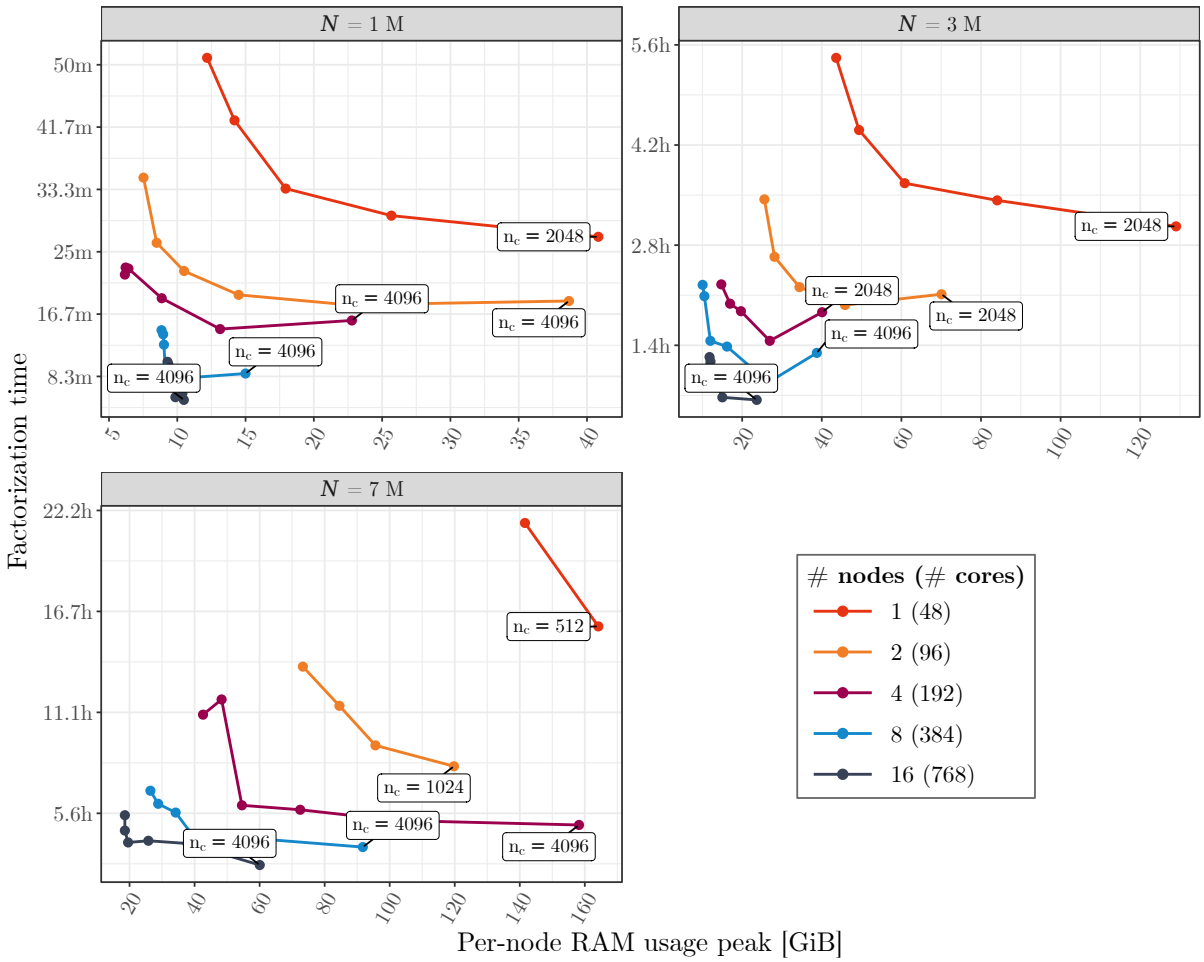


FIG. 13: Computation times of **multi-solve** with MUMPS/SPIDO on FEM/BEM linear systems of varying size N and for varying values of n_c. Labels indicate the highest achieved n_c. Parallel runs on 1 to 16 `skylake` nodes (48 threads/node).

In the second time, we compare the MUMPS/HMAT runs to their MUMPS/SPIDO references. As of the computation time, the compression of S does not bring significant improvements. In some cases MUMPS/HMAT is slightly faster than MUMPS/SPIDO, in some other cases it is the opposite. These differences have two main causes. First, our application presents a non-negligible execution time variability, even in shared memory [4]. Second, the way MUMPS distributes computation load and data varies and impacts the time to solution differently from one execution to another, even if we consider the exact same linear system. As of the RAM usage of MUMPS/HMAT compared to MUMPS/SPIDO, the compression of S brings an important reduction of the application's memory footprint. On smaller problems, i.e. for N up to 3,000,000, this phenomenon is noticeable especially for low numbers of computational nodes, i.e. 1 or 2. However, for larger problems, i.e. starting with N at 7,000,000, the positive impact of compressing S is present even for higher numbers of nodes, i.e. up to 16.

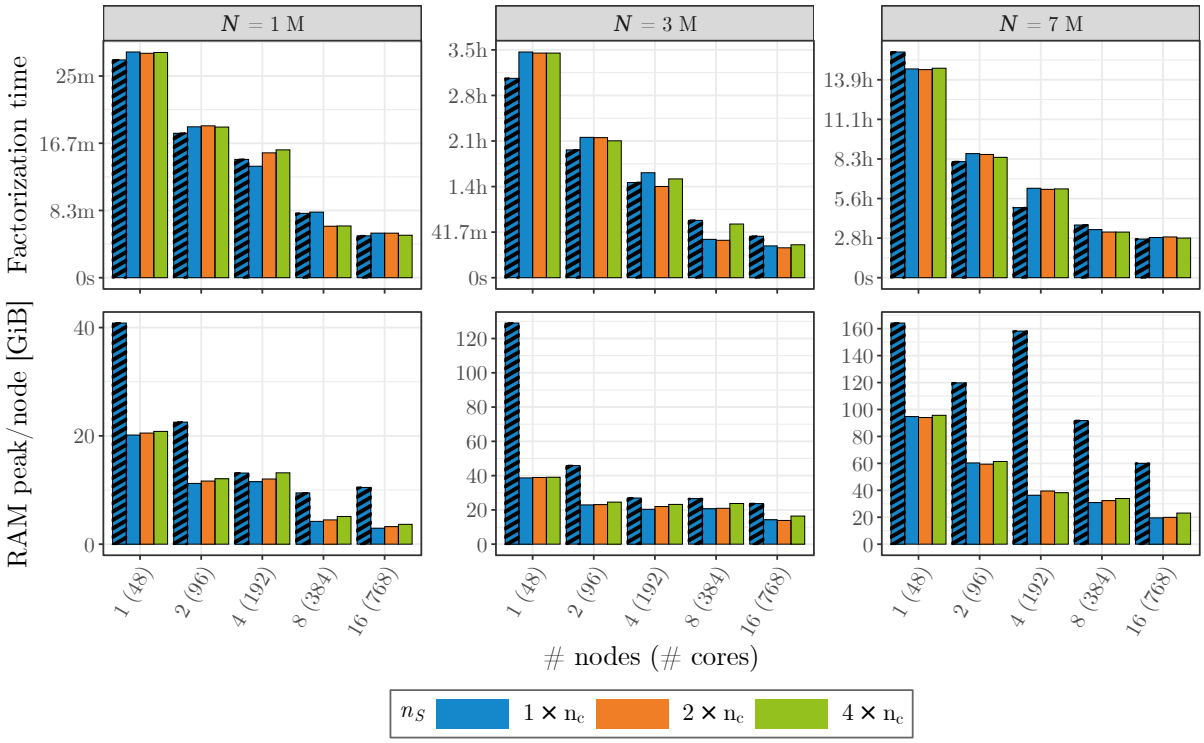


FIG. 14: Computation times and per-node RAM usage peaks of **multi-solve** with MUMPS/HMAT on FEM/BEM linear systems of varying size N and for varying values of n_S. For each parallel configuration (x-axis) and problem size N, the starting value of n_c is based on the best-performing MUMPS/SPIDO execution (striped bars) with the same N and the same parallel configuration. The value of n_S for MUMPS/HMAT (solid-colored bars) represent multiples (1×, 2× and 4×) of this base n_c value. Parallel runs on up to 16 `skylake` nodes.

Eventually, the evaluation of relative error for the test cases from Fig. 13 and 14 is included in the Appendix of this report (see Fig. 26 and 27 therein). It verifies that the relative error is below the selected threshold of 10^{-3} and confirms that the algorithm allows us to reach the expected accuracy.

6.1.2 Multi-factorization

In this section, we involve coupled FEM/BEM linear systems with N, the total unknown count, of 1,000,000, 2,000,000 and 3,000,000. We run benchmarks on 1 to 16 nodes, i.e. using 48 to 768 threads. Both the *baseline multi-factorization* and the *compressed Schur multi-factorization*

variants 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 1, 3, 7 and 11. Note that, from the implementation point of view, running *baseline multi-factorization* (MUMPS/ SPIDO) with n_b set to 1 (blue curves with round points in the figures below) equals to the reference advanced vanilla coupling (see Section 4).

Fig. 15 shows the computation time in function of per-node RAM usage peaks for both the MUMPS/SPIDO and the MUMPS/HMAT couplings. Again, the behavior of the algorithm with respect to the n_b parameter is analogous to the one observed in the case of the experiments in shared memory [5]. The smaller the value of n_b and the lower the number of calls to the *sparse factorization+Schur* building block, causing the superfluous and costly re-factorization of the large sparse A_{vv} submatrix within W. To optimize the performance, the goal is therefore to lower n_b as much as possible before reaching the memory limit. For example, with $N = 2{,}000{,}000$ on one node, the computation time of multi-factorization with $n_b = 3$ (leading to 6 calls to *sparse factorization+Schur*) is more than eight times lower than with $n_b = 11$ (leading to 66 calls to *sparse factorization+Schur*). However, unlike in the case of distributed multi-solve (see Section 6.1.1), computation time of multi-factorization decreases only slightly with increasing number of nodes. This is especially noticeable for small values of n_b. The additional evaluation of scalability and parallel efficiency in the Appendix of this report (see Fig. 28 and 29 therein) confirm this observation. For example, considering 2,000,000 of unknowns, for both MUMPS/SPIDO and MUMPS/HMAT, the computation time remains almost constant for all the parallel configurations, when $n_b < 7$. Given that this happens in particular when n_b is 1, i.e. in the supposedly optimal performance condition, indicates a possible issue in the *sparse factorization+Schur* building block of the sparse solver. We further discuss this in the multi-metric study of the algorithms in [4]. The parallel efficiency of multi-factorization thus remains low (below 20%).

Moreover, the results show that the per-node memory footprint of the distributed multi-factorization algorithm can be reduced when using a higher number of nodes. The larger the problem the more important the decrease. For example, in the case of a coupled system with 3,000,000 unknowns and n_b set to 3 blocks, the maximum RAM peak on 4 nodes is of approximately 125 GiB whereas on 16 nodes it is around 55 GiB. Compression of the dense Schur complement part in the MUMPS/HMAT coupling further favors this effect. However, this becomes less noticeable with increasing number of nodes. Indeed, the reduction of memory footprint brought by the compression is already very important. This way, distributing small amount of data for computation over a too large amount of nodes leads to duplication and, in terms of RAM usage, penalizes MUMPS/HMAT which looses its advantage over MUMPS/SPIDO. At the end, thanks to distributed-memory parallelism, multi-factorization allows us not only to process larger problems but also to use smaller values of n_b compared to a single-node execution. This leads to faster processing of problems that can be treated on one node but only when considering large n_b values leading to poor performance. For example, let us consider the case of the system with 3,000,000 unknowns. Using MUMPS/SPIDO, it could not even be processed on one node. Using MUMPS/HMAT, the lowest possible n_b was 7, leading to 28 calls to the *sparse factorization+Schur* building block. Increasing the number of nodes to 4 allowed us to process this case with $n_b = 3$ and even with $n_b = 1$ which is the optimal situation.

Finally, the evaluation of relative error for the test cases from Fig. 15 is included in the Appendix of this report (see Fig. 30 and 31 therein). It verifies that the relative error is below the selected threshold of 10^{-3} and confirms that the algorithm allows us to reach the expected accuracy.

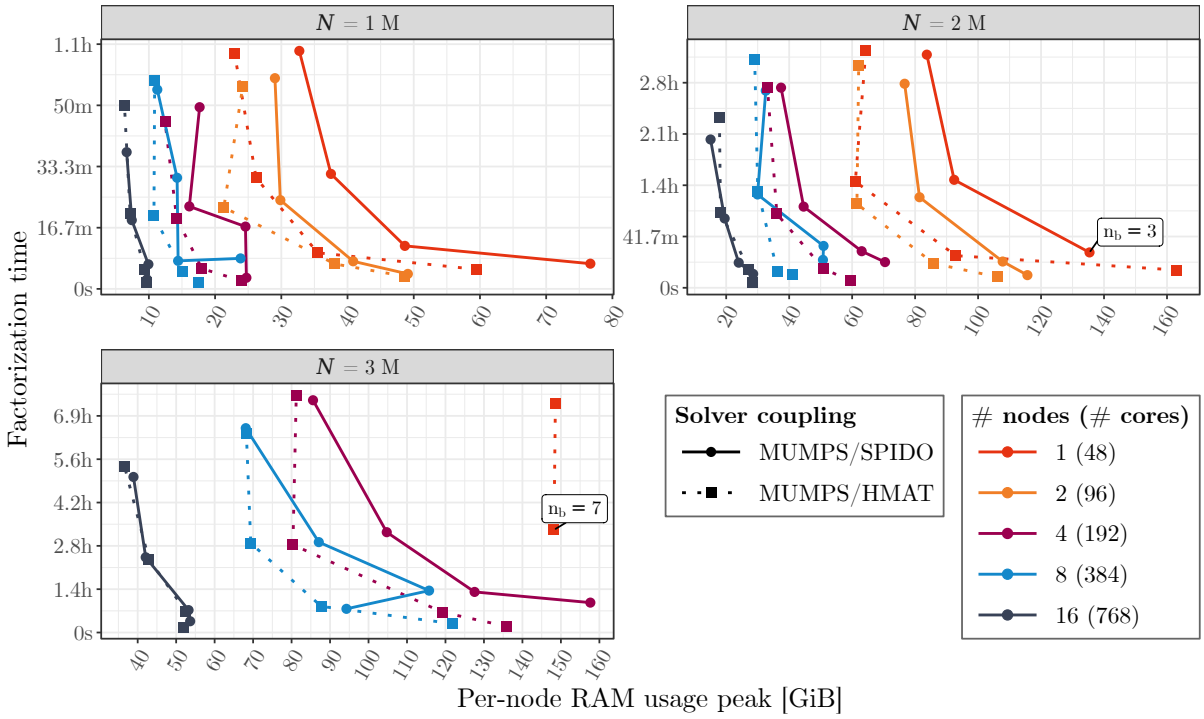


FIG. 15: Computation times of **multi-factorization** with MUMPS/SPIDO and MUMPS/HMAT couplings on FEM/BEM linear systems of varying size N and for n_b from 11 to 1 or until the RAM limit is reached. Labels indicate the lowest n_b we could achieve. Parallel runs on 1 to 16 `skylake` nodes using 48 threads per node.

6.2 Solving larger problems

The goal is now to push the distributed multi-solve and multi-factorization algorithms to their limits on a given number of nodes and considering both numerical compression and out-of-core computation. In this case, we consider 16 `skylake` nodes and increase the total unknown count N of the problem until either the memory or the execution time limit on the platform is reached. As in the rest of this study, we consider both the *baseline* as well as the *compressed Schur* variants of the multi-solve and multi-factorization algorithms relying on the MUMPS/SPIDO and MUMPS/HMAT couplings, respectively. The tested values of the n_c and n_S parameters for multi-solve and of the n_b parameter for multi-factorization were analogous to those from the corresponding sections 6.1.1 and 6.1.2. As a reference from the state-of-the-art, we use the advanced vanilla solver coupling (see Section 4).

In Fig. 16, we show the best computation times for all of the evaluated configurations and problem sizes. Fig. 32 in the Appendix of this report completes the analysis with the corresponding maximum RAM usage peaks. The algorithm allowing us to process the largest coupled sparse/dense FEM/BEM system is multi-solve for N as high as 42,000,000 unknowns in total. In the case of the *compressed Schur multi-solve*, this result was reached without necessity for out-of-core in the Schur complement part of the system. As we showed in Section 5.2, compressing the Schur complement part is very efficient by itself and the usage of out-of-core has thus only a limited effect. On the contrary, in the case of the *baseline multi-solve*, we had to resort to out-of-core in all the parts of the system to prevent running out of memory. However, none of the variants reached the physical memory limit. Therefore, they could possibly process even larger FEM/BEM systems without the execution time restrictions of the platform, i.e. 3 days. With multi-factorization, we could reach N = 7,000,000. Numerical compression, out-of-core or distributed-memory computation did not allow us to push significantly further than what is pos-

sible to achieve with the reference advanced vanilla coupling, i.e. 6,000,000 unknowns. Indeed, in this case, the bottleneck is not the storage of the Schur complement part but the superfluous storage and the loss of symmetry in the multi-factorization algorithm (see Section 2.4.2).

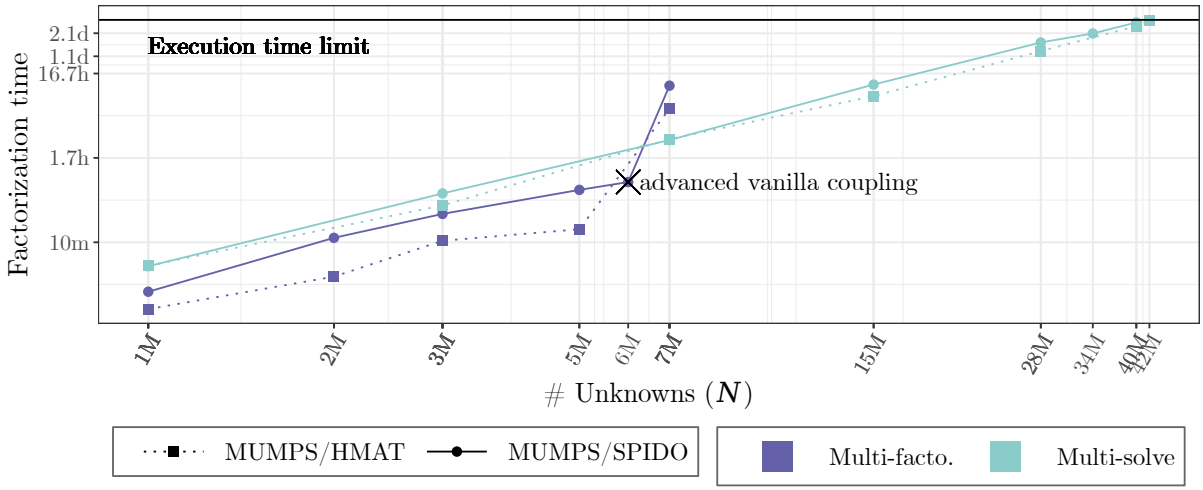


FIG. 16: Best computation times of **multi-solve** and **multi-factorization** for both MUMPS/HMAT and MUMPS/SPIDO with out-of-core optionally enabled. The state-of-the-art is represented by the advanced vanilla coupling. Parallel runs using 768 threads on 16 `skylake` nodes.

6.3 Conclusion

The experimental study of multi-solve and multi-factorization in distributed memory showed that both of the algorithms can make use of the available memory to increase their performance and allow for processing larger coupled FEM/BEM systems compared to their shared-memory counterparts as well as to the state of the art. The experiments further revealed that the multi-solve algorithm is also able to take advantage of an increasing number of parallel computational nodes to accelerate the computation. This was not the case of multi-factorization. The issue might be coming from its core component, i.e. the *sparse factorization+Schur* building block of the sparse direct solver, but a deeper investigation is required to confirm or contradict this hypothesis. Nevertheless, compared to the reference from the state of the art, we could process more than 7× larger (42,000,000 unknowns against 6,000,000) coupled systems thanks to the proposed algorithms. Moreover, without the 3-day time limit on the target platform, we might be able to process even larger coupled systems.

7 Perspectives

We plan to deploy the parallel distributed versions of multi-solve and multi-factorization at Airbus and potentially allow for processing of more demanding industrial simulations. As a long term work, we also plan to investigate alternative solver couplings to cope with the remaining drawbacks of the multi-solve and multi-factorization methods, such as the explicit storage of dense blocks Y_i in multi-solve or the superfluous re-factorizations of A_{vv} in multi-factorization. One idea is to rely on a task-based solver coupling allowing for a block-wise out-of-core computation of the Schur complement without having to perform multiple calls to the sparse solver's API. Another idea is to consider a single solver capable of applying sparse and dense operations based on the partitioning of the system [3].

8 Appendix

A Extended study of out-of-core in shared memory

In Section 5, we presented a study of the multi-solve and multi-factorization algorithms the goal of which was to determine whether out-of-core techniques may help us to lower the memory footprint of the proposed algorithms on a shared-memory workstation so as to allow for processing larger coupled systems and if so, to which extent. So far, we have considered a concrete industrial test case (see Section 4). In this section, we present an extended experimental study involving not only the industrial test case, but also two different academic test cases, namely the *wide pipe* and the *narrow pipe*. We introduced the *wide pipe* test case in Section 4. The *narrow pipe* (see Fig. 17) is a variation of *wide pipe* but with different dimensions which leads in *narrow pipe* to a higher proportion of unknowns in the dense part of the corresponding linear system with respect to its sparse part.

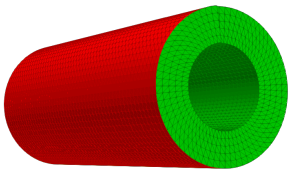


FIG. 17: *narrow pipe* (4 × 0.8 m) with FEM volume mesh v and BEM surface mesh s.

A.1 Academic cases

The experiments on academic test cases were conducted on `bora` nodes on the PlaFRIM platform (`https://plafrim.fr/en/`). A `bora` node has a total of 36 processor cores running each at 2.5 GHz (Hyper-Threading and Turbo-Boost were turned off), 192 GiB of RAM and a local hard disk SATA Seagate ST1000NX0443 turning at 7200 rpm. The solver test suite `test_FEMBEM` for both of the academic test cases is compiled with GNU C Compiler (gcc) 12.1.0, Intel(R) MKL library 2019.1.144, MUMPS 5.5.1 and StarPU 1.3.9. Each run presented below uses one node running one process with 36 threads. Regarding the out-of-core feature, we consider three scenarios. The *All in-core* label is used for benchmarks where out-of-core has been completely disabled in both sparse and dense solvers, i.e. all the data has been stored in the core memory (in-core). Then, the *Out-of-core except Schur complement* label is used for benchmarks where out-of-core has been enabled only in the sparse direct solver, i.e. the dense part (A_{ss} and consequently S) has been stored in-core. Finally, the *All out-of-core* label has been used for benchmarks where out-of-core has been enabled for both the sparse and the dense direct solver, including the Schur complement part S. We rely on the same solver couplings as described in Section 4.

We consider coupled FEM/BEM linear systems with N, the total unknown count, starting at 1,000,000 and increasing until the memory limit is reached. As of the algorithm parameters, we evaluate multiple configurations. For *baseline multi-solve* based 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 128 and 512. For the *compressed Schur multi-solve* variant, relying on the MUMPS/HMAT coupling, the size of blocks of columns of S and A_{sv}^T is handled by the n_S and n_c parameters, respectively. In this case, we set n_c to a constant value of 256 columns (motivated by the results from [5] and the RAM capacity of the `bora` nodes) and vary n_S in the range from 256 to 1,024. In the case of the multi-factorization algorithm, both the *baseline multi-factorization* and the *compressed Schur multi-factorization* variants expose the n_b parameter handling the count of blocks S_{ij} per block row and block column of the Schur complement submatrix S. The tested values of n_b are 1, 3, 7 and 11.

A.1.1 *wide pipe*

At first, we consider the *wide pipe* academic test case. Compared to the *narrow pipe*, it has a larger volume mesh and therefore a higher ratio of FEM-related to BEM-related unknowns.

Multi-solve Figures 18 and 19 show respectively the computation times as well as the peak RAM and hard drive usage of the multi-solve algorithm for both of the solver couplings, i.e. MUMPS/SPIDO and MUMPS/HMAT.

When both the sparse and the dense solver keep all the data in RAM (see the *All in-core* column in the figures), MUMPS/SPIDO is able to process up to 8,000,000 unknowns with n_c set up to 256 columns before reaching the memory limit. When the Schur complement part is compressed, i.e. in the case of MUMPS/HMAT, we can reach up to 12,000,000 unknowns.

Now, using out-of-core exclusively in the sparse solver (see the *Out-of-core except Schur complement* column in the figures) enables MUMPS/SPIDO to solve systems with N up to 9,000,000 before running out of RAM. With MUMPS/HMAT, we can go up to 14,000,000. However, in this case, we were not bound by the RAM, used at 62%, but by the time limit of the PlaFRIM platform for one execution, i.e. 3 days. Putting the RAM consumption aside, running the sparse solver out-of-core slows down the computation in the case of both couplings. In average, we can observe 67% slow-down for MUMPS/SPIDO and 69% for MUMPS/HMAT with respect to the *All in-core* benchmarks. Such an important overhead can be explained by the numerous calls to the *sparse solve* step in MUMPS during the Schur complement assembly. Indeed, in order to perform each of the *sparse solve* steps of MUMPS in multi-solve, the solver reads the factors of the previously factorized A_{vv} from disk. However, thanks to out-of-core, we can rise n_c to 512 which reduces the number of calls to *sparse solve* and thus allow us to counterbalance the slow-down. Understanding the different levels of overhead between MUMPS/SPIDO and MUMPS/HMAT would require a deeper investigation which is beyond the scope of this study.

Running all the solvers out-of-core (see the *All out-of-core* column in the figures) further degrades the time to solution leading to an overhead of 70% for MUMPS/SPIDO and 80% for MUMPS/HMAT. In this case, MUMPS/SPIDO and MUMPS/HMAT could process coupled systems with up to 14,000,000 unknowns while taking respectively 72% and 62% of available RAM. Past this problem size, the execution time exceeded the limit of 3 days on the PlaFRIM platform. This is also the explanation of why MUMPS/SPIDO with $n_c < 512$ could not reach further than 9,000,000 unknowns.

In summary, while it induces an overhead in terms of computation time, out-of-core allowed us to significantly lower the memory footprint of the multi-solve algorithm thanks to its activation in the Schur complement part in particular. As a result, multi-solve could process coupled systems which are up to 1,75× larger than without out-of-core, i.e. 14,000,000 against 8,000,000, and up to 7× larger than the state-of-the-art advanced vanilla coupling, i.e. 14,000,000 against 2,000,000 (see the blue round points in the *All in-core* column of Figure 20 representing an execution of MUMPS/SPIDO using the *baseline multi-factorization* algorithm with the entire Schur complement processed at once, i.e. with n_b set to 1). When out-of-core is used concurrently with numerical compression of S, the impact of out-of-core is only slightly noticeable due to the efficiency of the compression. This trend could be different on larger problems. Nevertheless, such experiments would require a high-performance computing platform with less restrictive execution time limits.

Multi-factorization Figures 20 and 21 respectively show the computation times as well as the peak RAM and hard drive usage of the multi-factorization algorithm for both of the solver couplings, i.e. MUMPS/SPIDO and MUMPS/HMAT. Unlike in the case of multi-solve (see

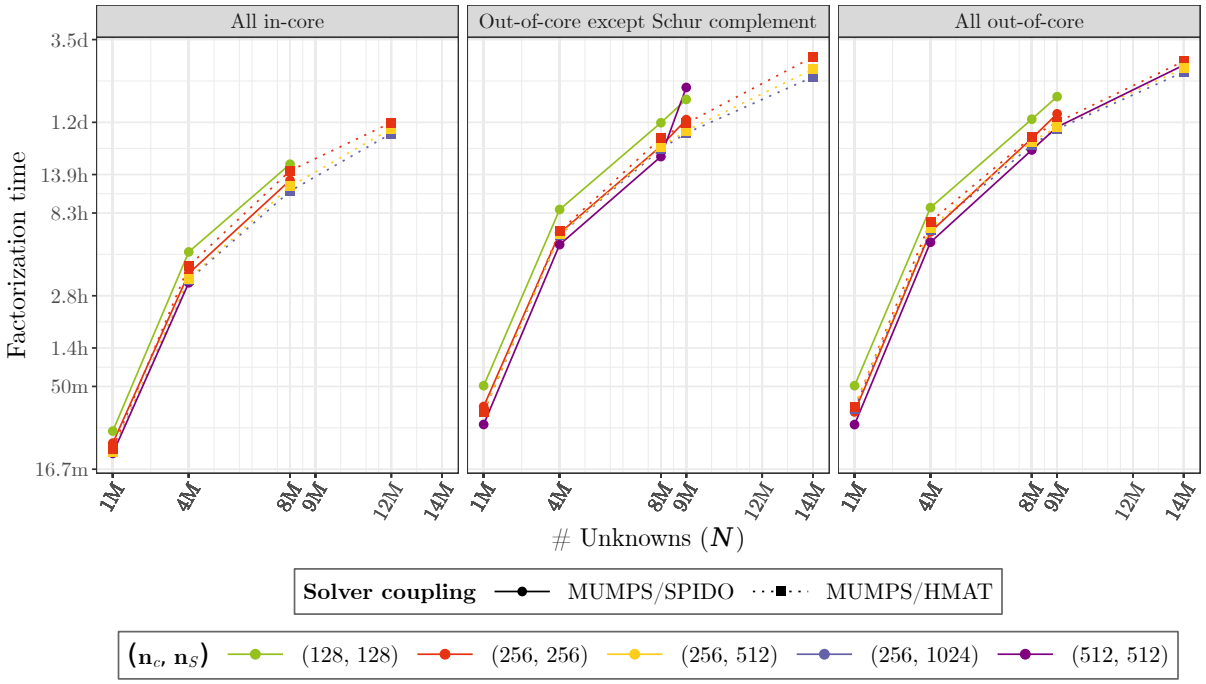


FIG. 18: Computation times of *baseline multi-solve* and *compressed Schur multi-solve* on coupled FEM/BEM linear systems of varying number of unknowns and for varying values of n_c and n_S. We test 3 different configurations of the *out-of-core* feature: completely disabled, enabled **except** for the Schur complement matrix or enabled **including** for the Schur complement matrix. Parallel runs on single `bora` node.

Section A.1.1), we report only a very low computation time overhead associated to the usage of out-of-core. Without out-of-core and with out-of-core activated only in the sparse solver (see the *All in-core* and the *Out-of-core except Schur complement* columns in the figures), the MUMPS/SPIDO coupling can process up to 2,000,000 unknowns before reaching the memory limit. When running both the sparse and the dense solvers out-of-core (see the *All out-of-core* column in the figures), MUMPS/SPIDO allowed us to reach 3,000,000 unknowns with n_b set to 11. Thanks to the compression of the Schur complement part, i.e. with MUMPS/HMAT, we can reach up to 3,000,000 unknowns. Activating out-of-core in the sparse solver then allows for lowering the value of n_b from 11 to 3. This reduces the number of calls to the *sparse factorization+Schur* building block in the core phase of multi-factorization (see Section 2.4.2) and thus significantly accelerates the computation of the 3,000,000-unknown test case.

However, in this situation, the dense Schur complement part is not large enough with respect to the sparse submatrices of A, for it to represent the most important limitation in terms of RAM consumption. Actually, the bottleneck is the amount of RAM required by the sparse solver during the Schur complement assembly. The multi-factorization algorithm implies data duplication related to the loss of symmetry in the temporary submatrix W on which the *sparse factorization+Schur* building block is applied (see Section 2.4.2). The loss of symmetry then amplifies the negative impact of fill-in in W on RAM usage. In addition to that, the out-of-core feature of MUMPS has only a limited influence on the RAM consumption of *sparse factorization+Schur* because the Schur complement is kept in the core memory. This also explains the low overhead of out-of-core in this case. Nevertheless, we expect this trend to vary depending on the ratio of the number of unknowns in the dense part of the system to the number of unknowns in its sparse part. We therefore evaluate out-of-core in the multi-factorization algorithm on a second academic test case with higher proportion of unknowns in the dense part and further on an industrial test case in sections 5 and A.2.

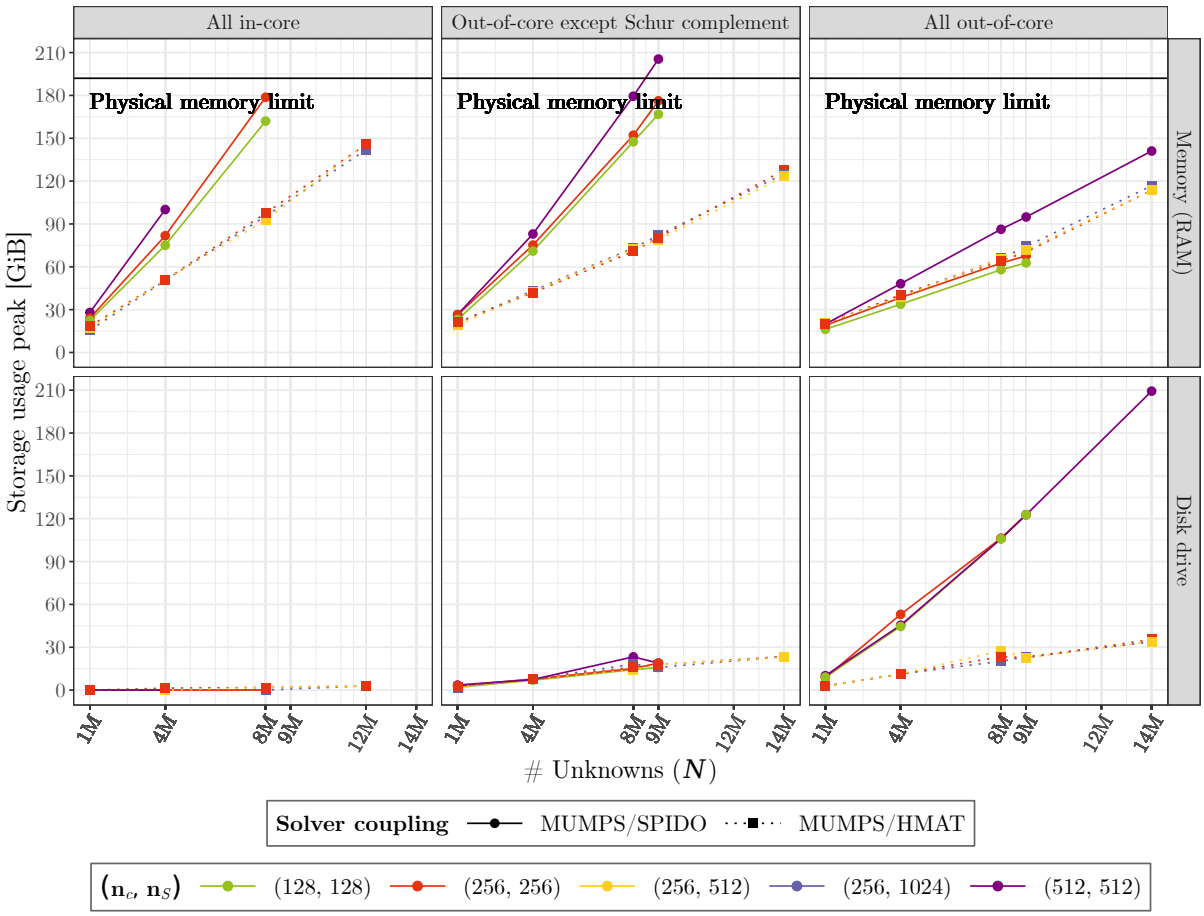


FIG. 19: RAM and hard drive usage peaks of *baseline multi-solve* and *compressed Schur multi-solve* on coupled FEM/BEM linear systems of varying number of unknowns and for varying values of n_c and n_S. We test 3 different configurations of the *out-of-core* feature: completely disabled, enabled **except** for the Schur complement matrix or enabled **including** for the Schur complement matrix. Parallel runs on single `bora` node.

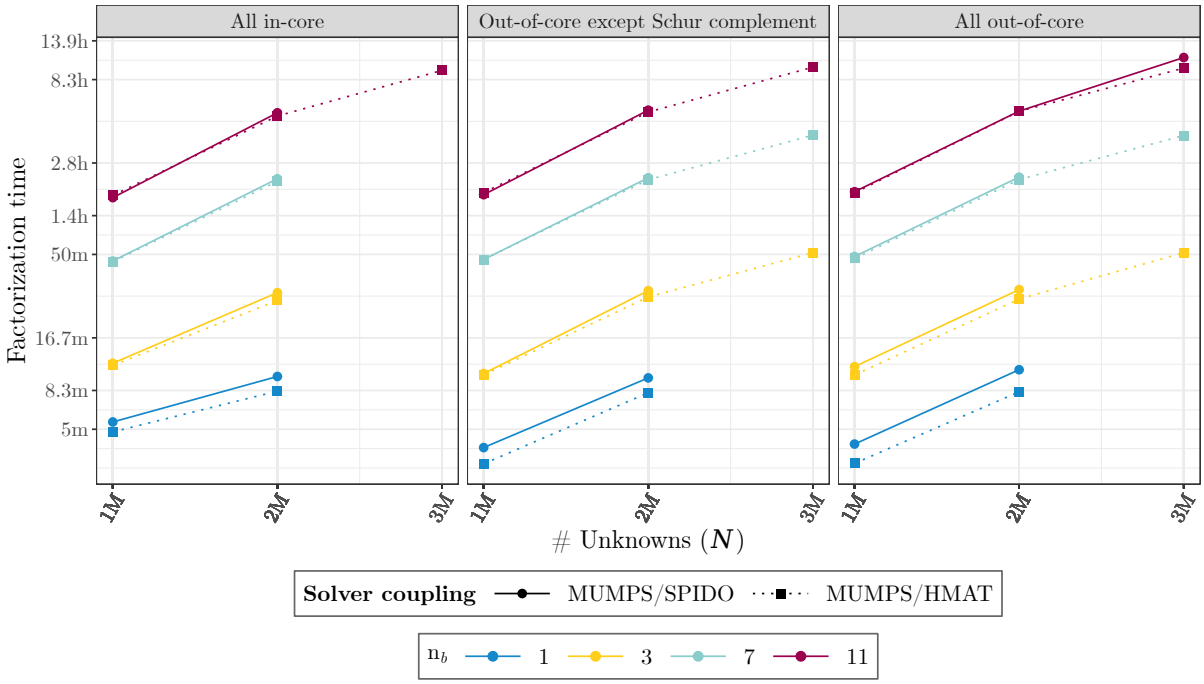


FIG. 20: Computation times of *baseline multi-factorization* and *compressed Schur multi-factorization* on coupled FEM/BEM linear systems of varying number of unknowns and for varying values of n_b. We test 3 different configurations of the *out-of-core* feature: completely disabled, enabled **except** for the Schur complement matrix or enabled **including** for the Schur complement matrix. Parallel runs on single `bora` node.

A.1.2 *narrow pipe*

In the second part of this study, we consider the *narrow pipe* academic test case. Compared to the *wide pipe*, it has a smaller volume mesh and therefore a higher ratio of BEM-related unknowns in the dense part of the resulting system to FEM-related unknowns in the sparse part of the resulting system.

Multi-factorization Considering the *narrow pipe* test case, figures 22 and 23 show respectively the computation times as well as the peak RAM and hard drive usage of the multi-factorization algorithm for both of the solver couplings, i.e. MUMPS/SPIDO and MUMPS/HMAT.

When out-of-core is disabled for both the sparse and the dense solver (see the *All in-core* column of the figure), MUMPS/SPIDO can process up to 3,000,000 unknowns with n_b set to 11 blocks and up to 2,000,000 with n_b in between 1 and 7 blocks before reaching the memory limit. When the Schur complement part is compressed, i.e. in the case of MUMPS/HMAT, we can reach up to 4,000,000 unknowns in total. Moreover, we can process the 3,000,000 problem using only 7 blocks per block row and column of S instead of 11 like in the case of MUMPS/SPIDO.

Activating out-of-core only in the sparse solver (see the *Out-of-core except Schur complement* column in the figure) has again a limited effect. The overhead with respect to the *All in-core* benchmarks was of 3% for MUMPS/SPIDO and 4% for MUMPS/HMAT. However, MUMPS/HMAT could reach 4,000,000 of unknowns with n_b set to 7 instead of 11 and 3,000,000 unknowns with n_b set to 3 instead of 7.

Unlike in the *wide pipe* test case (see Section A.1.1), using out-of-core also on the Schur complement part (see the *All out-of-core* column in the figure) brings more advantages. On the one

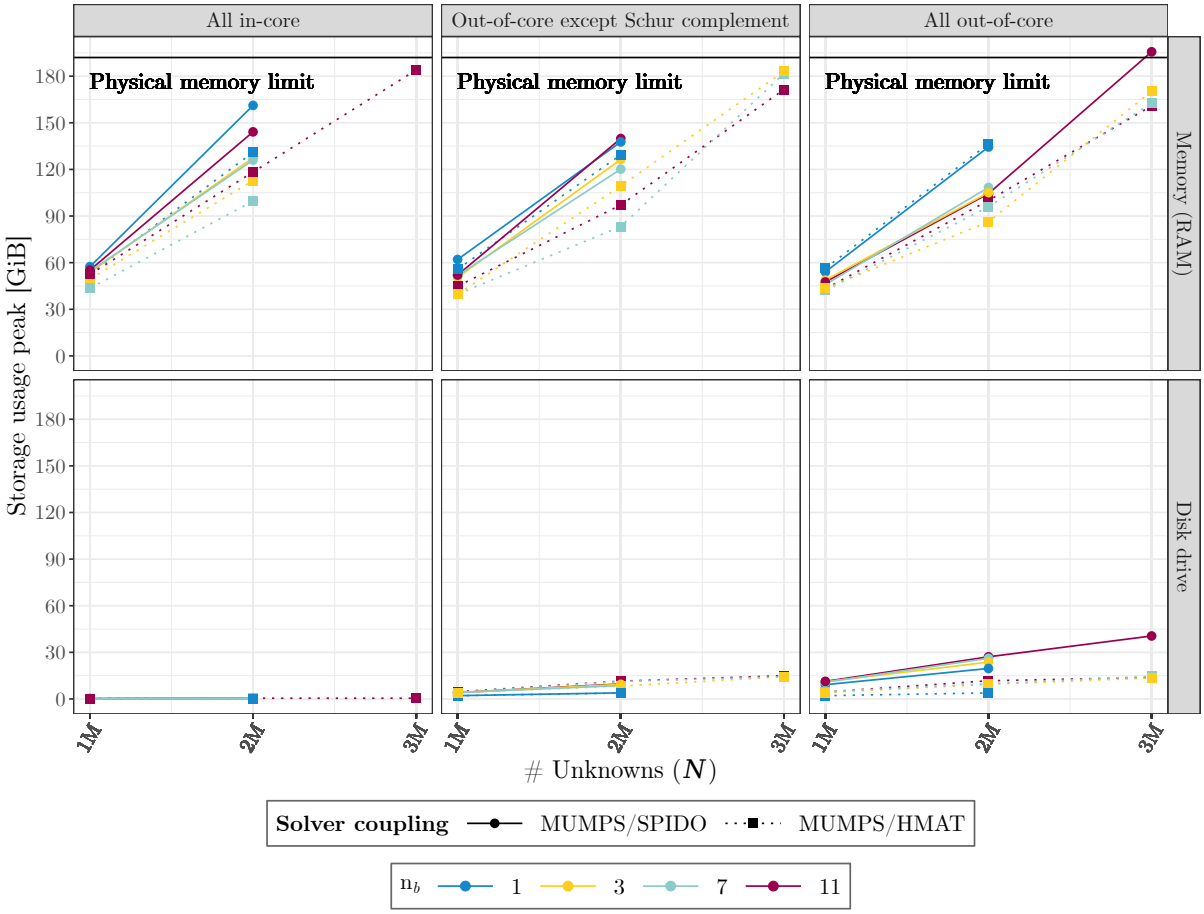


FIG. 21: RAM and hard drive usage peaks of *baseline multi-factorization* and *compressed Schur multi-factorization* on coupled FEM/BEM linear systems of varying number of unknowns and for varying values of n_b. We test 3 different configurations of the *out-of-core* feature: completely disabled, enabled **except** for the Schur complement matrix or enabled **including** for the Schur complement matrix. Parallel runs on single `bora` node.

hand, MUMPS/SPIDO is capable of processing systems with up to 4,000,000 of unknowns using less Schur complement blocks, i.e. with lower n_b. On the other hand, the 3,000,000-unknown case can be processed by both couplings with n_b set to 3 blocks, i.e. resorting to 6 calls to *sparse factorization+Schur* instead of 66 ($n_b = 11$) and 28 ($n_b = 7$) respectively for MUMPS/SPIDO and MUMPS/HMAT without using out-of-core on the Schur complement part. In the *All out-of-core* configuration, the reported slow-down with respect to the *All in-core* benchmarks was of 9% for MUMPS/SPIDO and 3% for MUMPS/HMAT.

Compared to the case of multi-solve (see Section A.1.1), the impact of out-of-core, especially in the dense part of the system, when combined with numerical compression is more present both in terms of better computation time and lower RAM consumption.

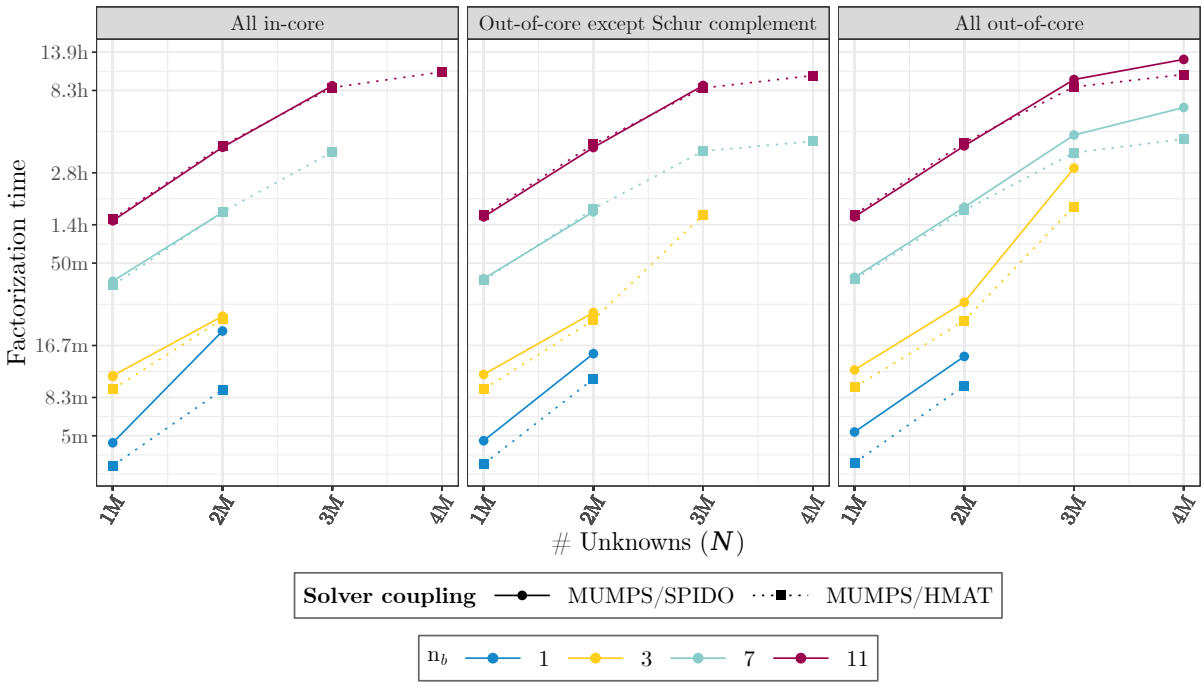


FIG. 22: Computation times of *baseline multi-factorization* and *compressed Schur multi-factorization* on coupled FEM/BEM linear systems of varying number of unknowns and for varying values of n_b. We test 3 different configurations of the *out-of-core* feature: completely disabled, enabled **except** for the Schur complement matrix or enabled **including** for the Schur complement matrix. Parallel runs on single `bora` node.

A.2 Industrial case

This section presents complementary experimental results on the on the real-life industrial case from Airbus Central R&T introduced in Section 4. In Section 5, we have considered the situations with low-rank compression enabled either in none of the solvers or in both the sparse and dense direct solver. Here, we activate the numerical compression in the sparse solver but not in the dense solver. The results are presented in Table 3. In multi-solve, out-of-core in the dense solver (row 2) brings again an important decrease in RAM usage without degrading the computation time. Using out-of-core also in the sparse solver (row 3) leads to a lower overhead compared to the reference benchmarks (see Section 5.1), i.e. 54% instead of 380%, but does not considerably reduce the RAM consumption either which is already low thanks to the out-of-core in the dense solver. Here, multi-factorization performs better in all its configurations. However, thanks to out-of-core in the dense solver (rows 5 and 6) and further in the sparse parts of the system,

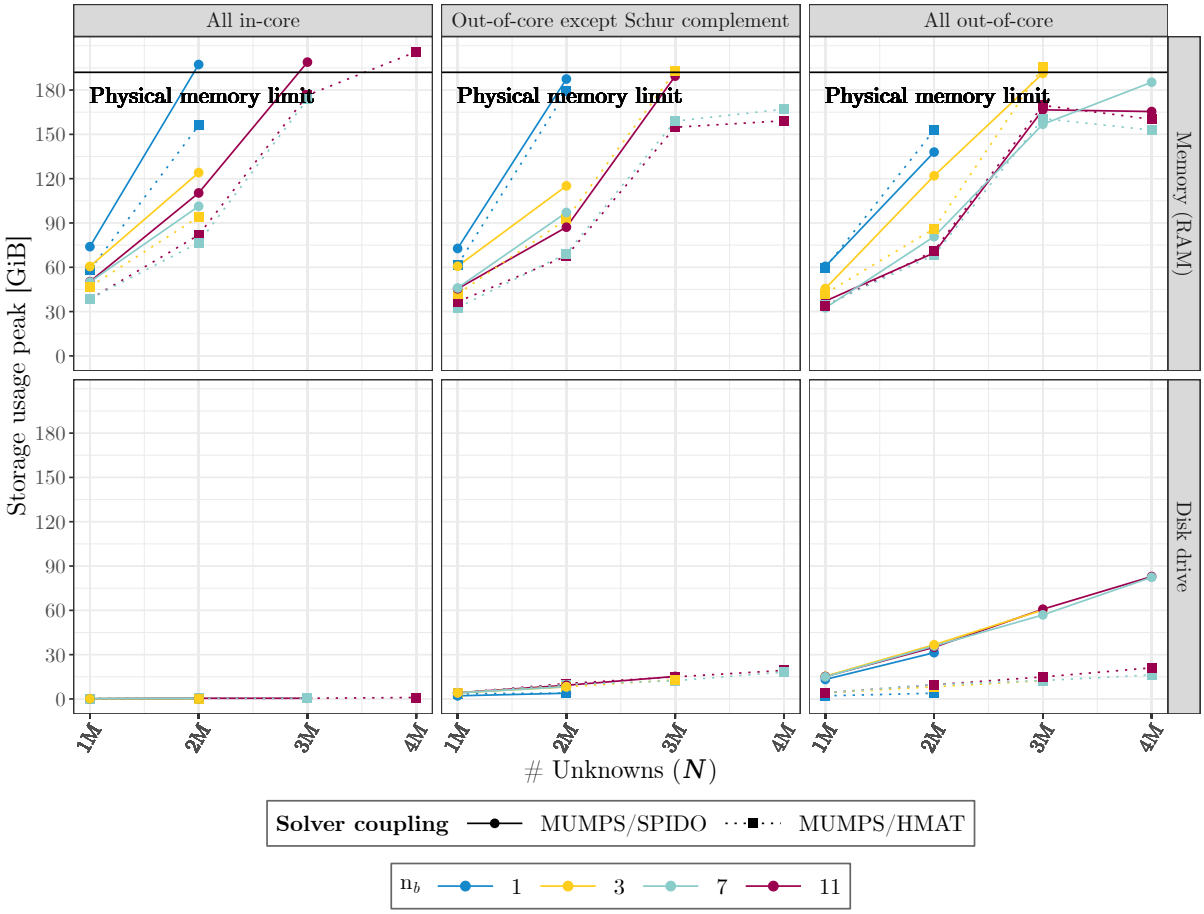


FIG. 23: RAM and hard drive usage peaks of *baseline multi-factorization* and *compressed Schur multi-factorization* on coupled FEM/BEM linear systems of varying number of unknowns and for varying values of n_b. We test 3 different configurations of the *out-of-core* feature: completely disabled, enabled **except** for the Schur complement matrix or enabled **including** for the Schur complement matrix. Parallel runs on single `bora` node.

we can lower n_b to 3 and accelerate the computation 3.25× (row 9) compared to the fastest multi-solve executions (rows 1 and 2).

	Algorithm	Dense *ooc*	Sparse *ooc*	n_b	RAM (GiB)	Disk (GiB)	Time
1	multi-solve			N/A	229	25	15.6h
2	multi-solve (§3.1.1)	x		N/A	24	229	15.6h
3	multi-solve (§3.1.1)	x	x	N/A	14	235	1d
4	multi-facto.			**12**	279	20	**8.3h**
5	multi-facto. (§3.1.2)	x		12	80	223	8.5h
6	multi-facto. (§3.1.2)	x		6	82	223	6.2h
7	multi-facto. (§3.1.2)	x	x	12	57	235	8.9h
8	multi-facto. (§3.1.2)	x	x	6	64	235	6.6h
9	multi-facto. (§3.1.2)	x	x	**3**	120	235	**4.8h**

TABLE 3: Performance of the two-stage algorithms on the industrial test case with compression in the sparse solver, without compression in the dense solver and with out-of-core (*ooc*) optionally enabled.

B Additional experiments in distributed memory

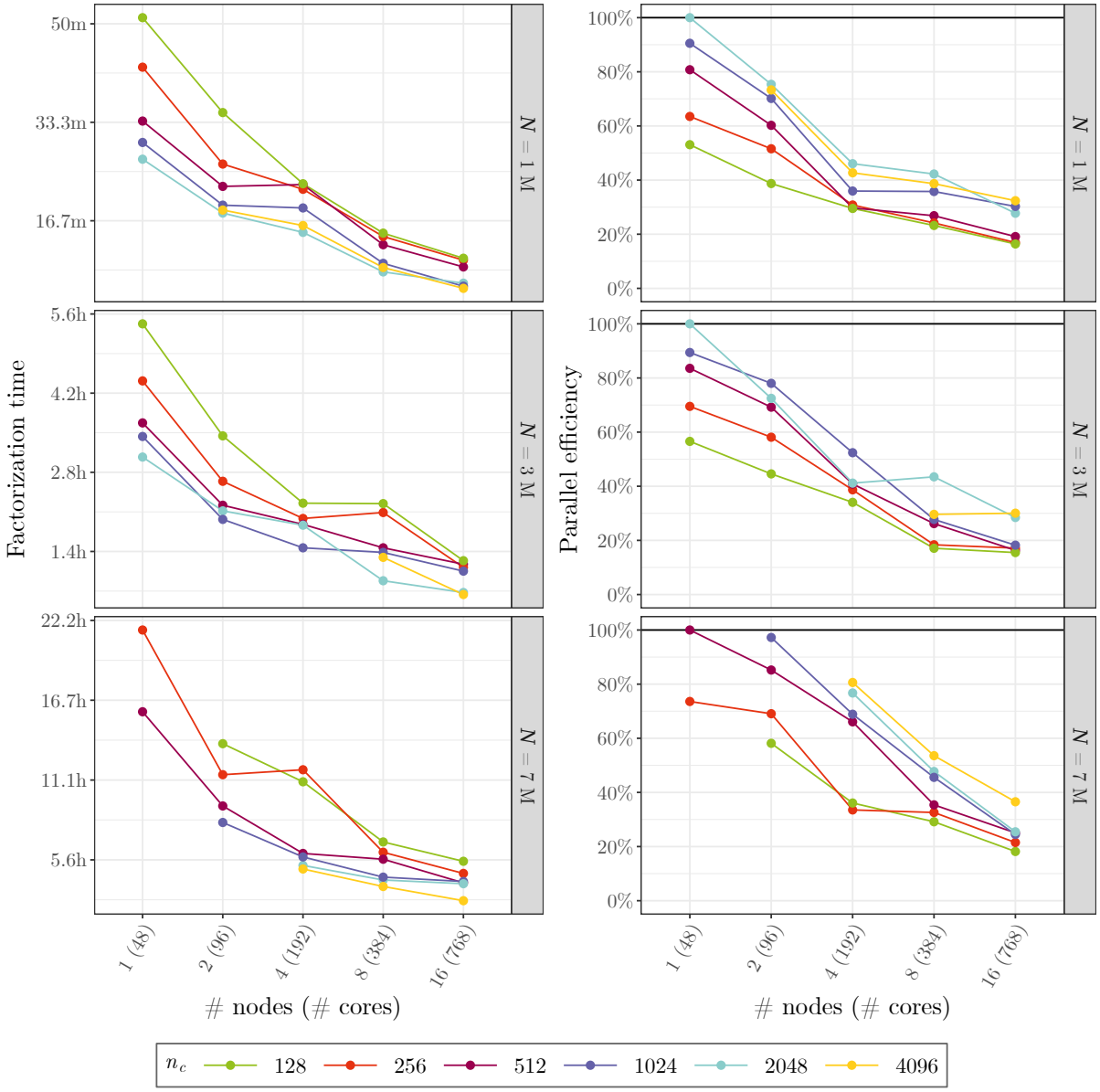


FIG. 24: Scalability and parallel efficiency of *baseline multi-solve* on coupled FEM/BEM linear systems of varying of unknownsand for varying values of n_c and n_S. Parallel runs on multiple `skylake` nodes.

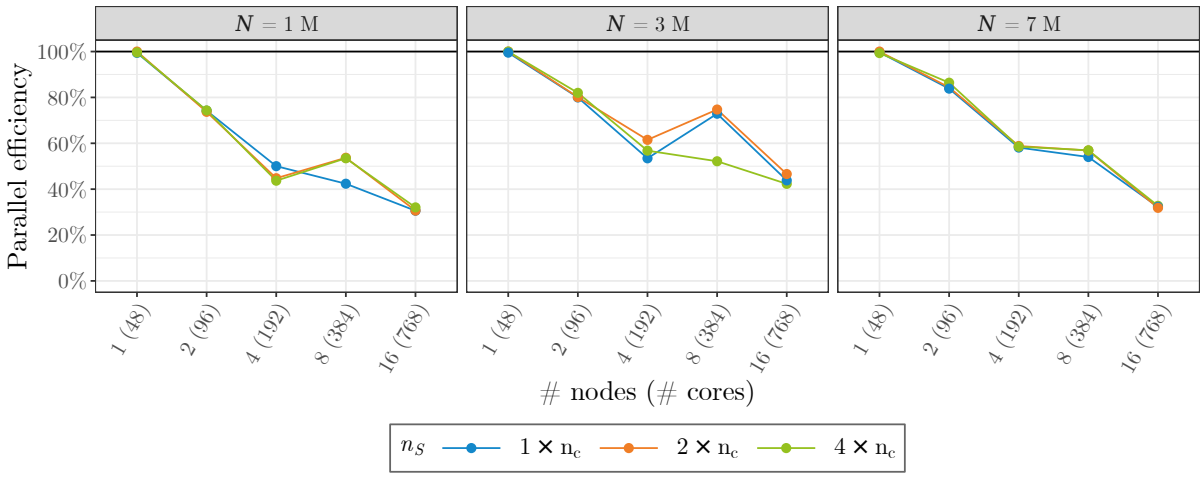


FIG. 25: Parallel efficiency of *compressed Schur multi-solve* on coupled FEM/BEM linear systems of varying number of unknowns and for varying values of n_c and n_S. Parallel runs on multiple `skylake` nodes.

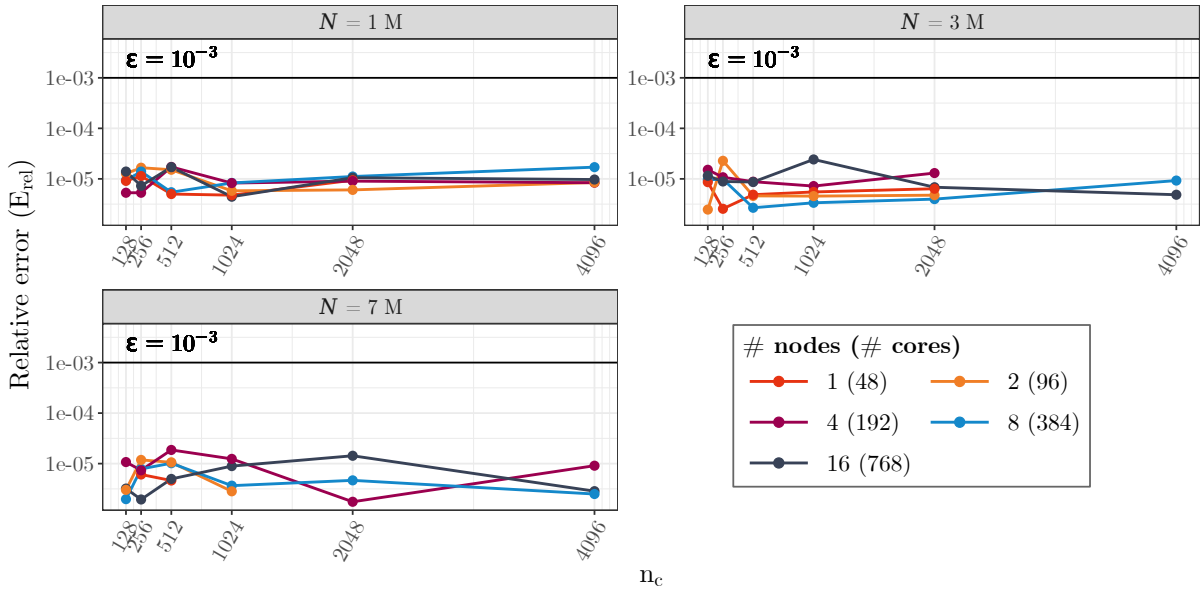


FIG. 26: Relative error (E_{rel}) of *baseline multi-solve* on coupled FEM/BEM linear systems of varying number of unknowns and for varying values of n_c. Parallel runs on multiple `skylake` nodes. For MUMPS, the precision parameter ϵ was set to 10^{-3}. Compared to the fully compressed test cases relying on the MUMPS/HMAT coupling in Fig. 27, the relative error here is smaller as the dense part of the linear system is not compressed at all. The final result thus suffers less from the loss of accuracy due to the compression.

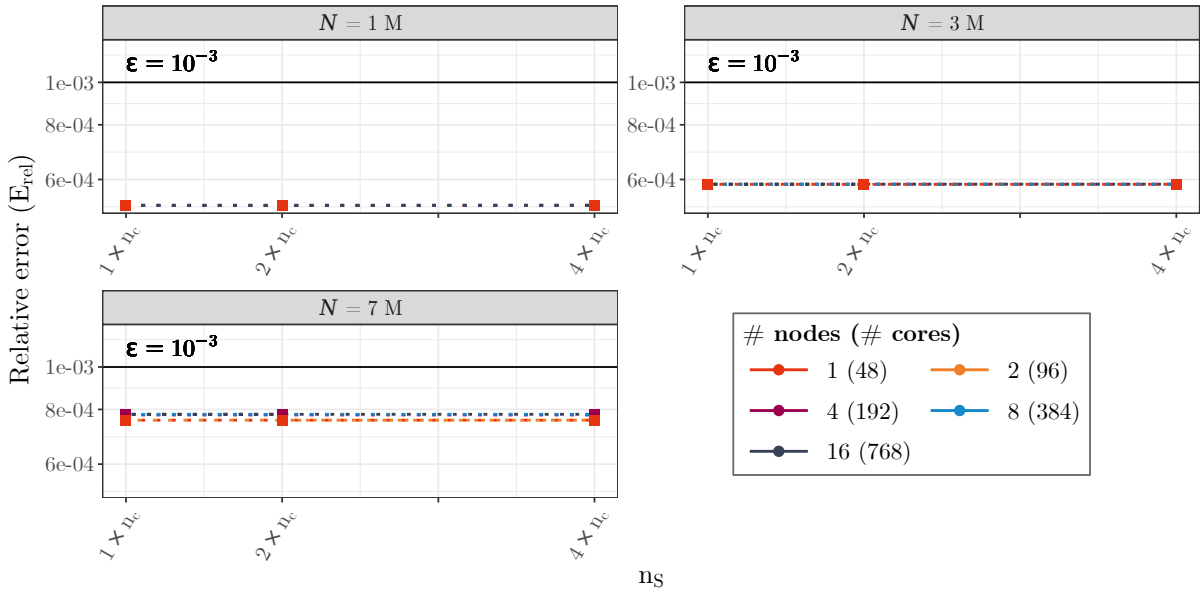


FIG. 27: Relative error (E_{rel}) of *compressed Schur multi-solve* on coupled FEM/BEM linear systems of varying number of unknowns and for varying values of n_S. Parallel runs on multiple `skylake` nodes. For MUMPS and HMAT, the precision parameter ϵ was set to 10^{-3}.

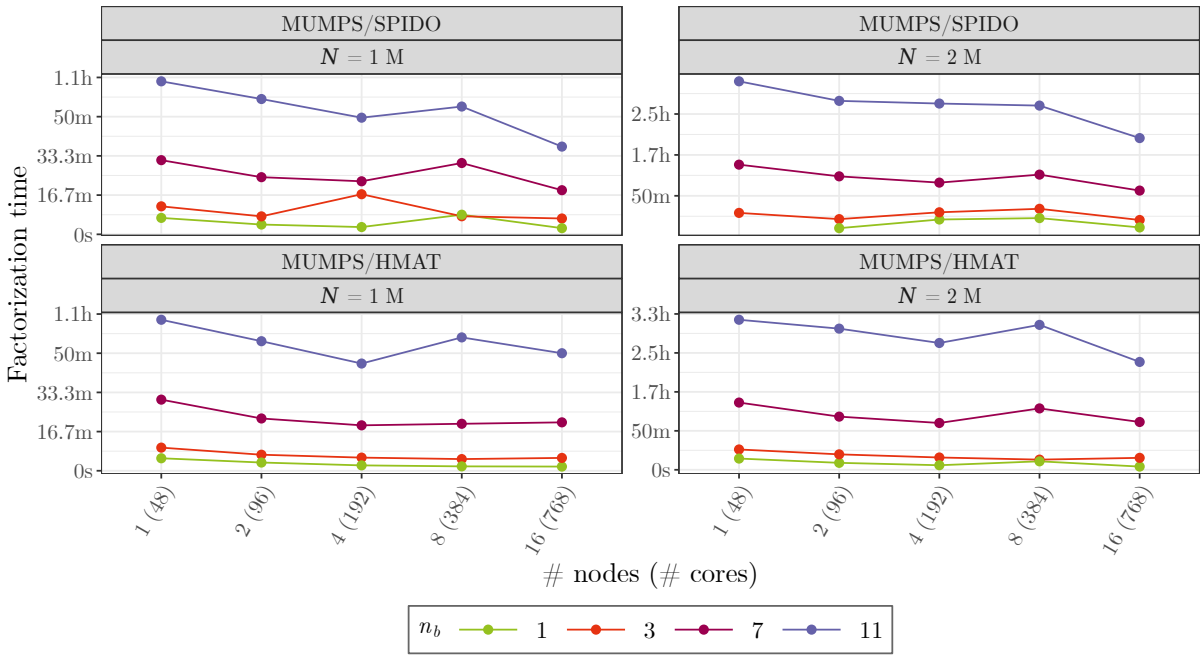


FIG. 28: Scalability of *baseline multi-factorization* and *compressed Schur multi-factorization* on coupled FEM/BEM linear systems of varying number of unknowns and for varying values of n_b. Parallel runs on multiple `skylake` nodes.

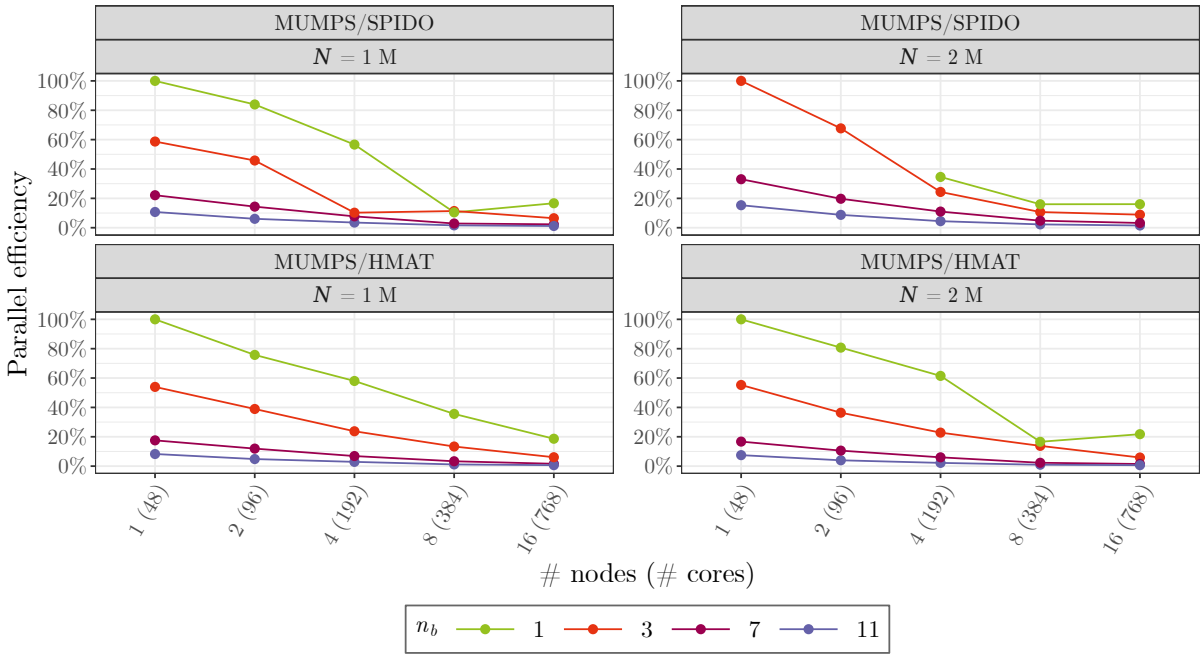


FIG. 29: Parallel efficiency of *baseline multi-factorization* and *compressed Schur multi-factorization* on coupled FEM/BEM linear systems of varying number of unknowns and for varying values of n_b. Parallel runs on multiple `skylake` nodes.

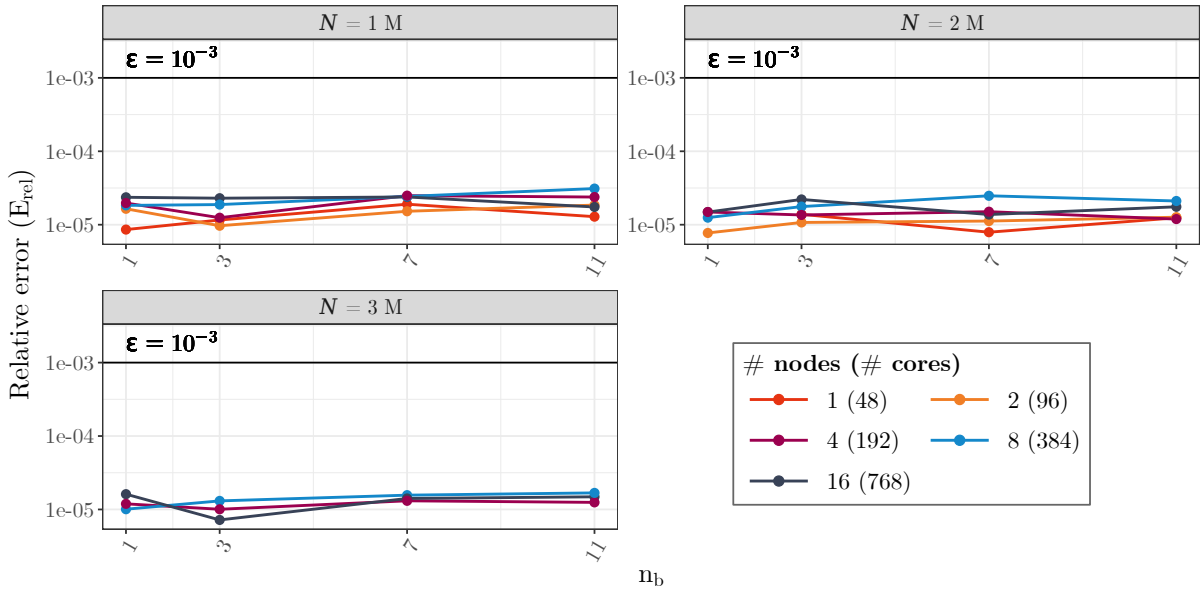


FIG. 30: Relative error (E_{rel}) of *baseline multi-factorization* on coupled FEM/BEM linear systems of varying number of unknowns and for varying values of n_b. Parallel runs on multiple `skylake` nodes. For MUMPS, the precision parameter ϵ was set to 10^{-3}. Compared to the fully compressed test cases relying on the MUMPS/HMAT coupling in Fig. 31, the relative error here is smaller as the dense part of the linear system is not compressed at all. The final result thus suffers less from the loss of accuracy due to the compression.

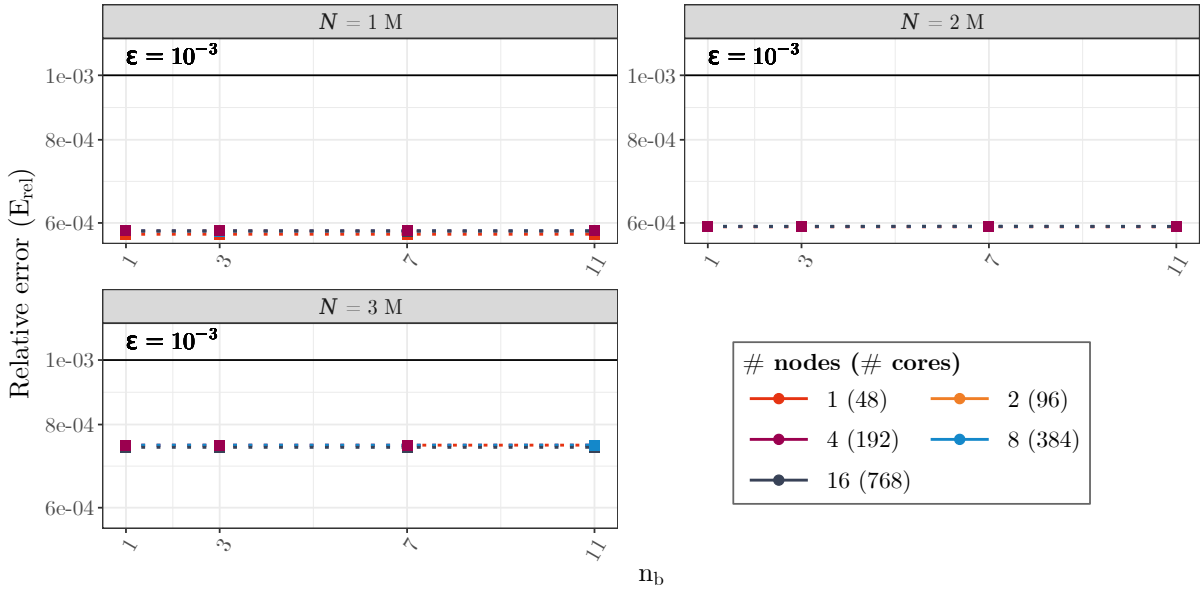


FIG. 31: Relative error (E_{rel}) of *compressed Schur multi-factorization* on coupled FEM/BEM linear systems of varying number of unknowns and for varying values of n_b. Parallel runs on multiple `skylake` nodes. For MUMPS and HMAT, the precision parameter ϵ was set to 10^{-3}.

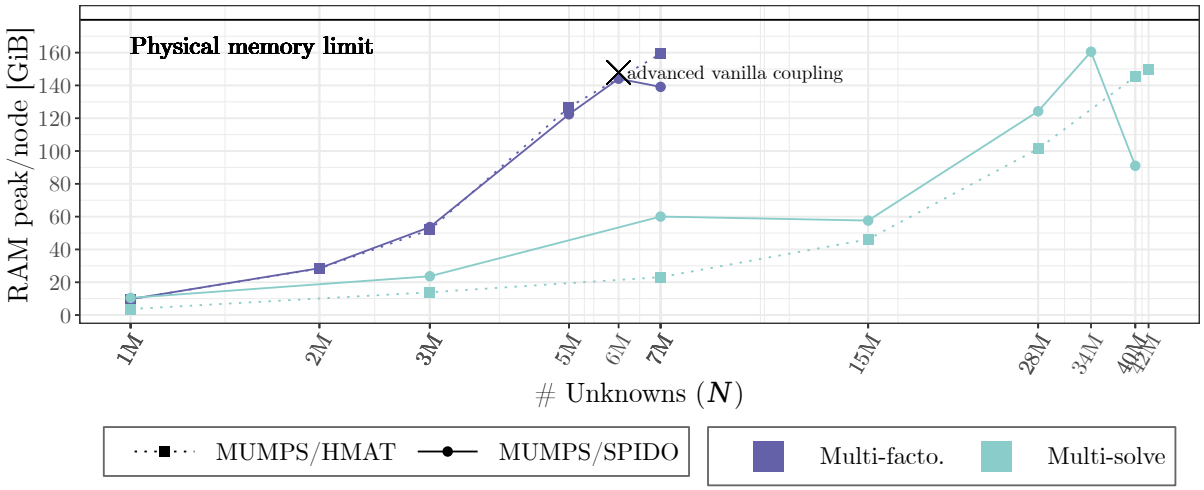


FIG. 32: Per-node RAM usage peaks, corresponding to the results in Figure 16, of **multi-solve** and **multi-factorization** for both MUMPS/HMAT and MUMPS/SPIDO with out-of-core optionally enabled. The state-of-the-art is represented by the advanced vanilla coupling. Parallel runs using 768 threads on 16 `skylake` nodes.

References

[1] E. Agullo, P. Amestoy, A. Buttari, A. Guermouche, J.-Y. L'Excellent, and F.-H. Rouet, *Robust memory-aware mappings for parallel multifrontal factorizations*, SIAM Journal on Scientific Computing, 38 (2016), pp. C256 – C279.

[2] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, *Implementing multifrontal sparse solvers for multicore architectures with Sequential Task Flow runtime systems*, ACM Transactions on Mathematical Software, (2016).

[3] E. Agullo, A. Falco, L. Giraud, and G. Sylvand, *Vers une factorisation symbolique hiérarchique de rang faible pour des matrices creuses*, in Conférence d'informatique en Parallélisme, Architecture et Système (ComPAS'17), Sophia Antipolis, France, June 2017.

[4] E. Agullo, M. Felšöci, A. Guermouche, H. Mathieu, G. Sylvand, and B. Tagliaro, *Study of the processor and memory power and energy consumption of coupled sparse/dense solvers*, in 2022 IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2022, pp. 211–220.

[5] E. Agullo, M. Felšöci, and G. Sylvand, *Direct solution of larger coupled sparse/dense linear systems using low-rank compression on single-node multi-core machines in an industrial context* , in 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Los Alamitos, CA, USA, 6 2022, IEEE Computer Society, pp. 25–35.

[6] 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, 37 (2015), pp. A1451–A1474.

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

[8] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK Users' Guide*, Society for Industrial and Applied Mathematics, 1997.

[9] F. Casenave, A. Ern, and G. Sylvand, *Coupled BEM–FEM for the convected Helmholtz equation with non-uniform flow in a bounded domain*, Journal of Computational Physics, 257 (2014), pp. 627–644.

[10] 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.

[11] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*, Oxford University Press, 2017.

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

[13] 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, 307 (2016), pp. 183–203. 1st Annual Meeting of SIAM Central States Section, April 11–12, 2015.

[14] 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.

[15] 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, 38 (2015).

[16] W. Gropp, *MPI: The Complete Reference*, no. v. 2 in Scientific and Engineering Computation, MIT Press, 1998.

[17] B. Lizé, *Résolution Directe Rapide pour les Éléments Finis de Frontière en Électromagnétisme et Acoustique : $\mathcal{H}$-Matrices. Parallélisme et Applications Industrielles.*, PhD thesis, Université Paris 13, 2014.

[18] 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, 52 (2012), pp. 446–467.

[19] A. Wang, N. Vlahopoulos, and K. Wu, *Development of an energy boundary element formulation for computing high-frequency sound radiation from incoherent intensity boundary conditions*, Journal of Sound and Vibration, 278 (2004), pp. 413–436.

[20] 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, 23 (1999), pp. 139–153.

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour

33405 Talence Cedex

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

ISSN 0249-6399