Scalable Domain Decomposition Preconditioners for Heterogeneous Elliptic Problems

Pierre Jolivet, Frédéric Hecht, Frédéric Nataf, Christophe Prud'Homme

▶ **To cite this version:**

Pierre Jolivet, Frédéric Hecht, Frédéric Nataf, Christophe Prud'Homme. Scalable Domain Decomposition Preconditioners for Heterogeneous Elliptic Problems. SC13 - International Conference for High Performance Computing, Networking, Storage and Analysis, Nov 2013, Denver, United States. pp.80:1–80:11, 10.1145/2503210.2503212 . hal-00939957

HAL Id: hal-00939957
https://hal.science/hal-00939957

Submitted on 31 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Scalable Domain Decomposition Preconditioners

Pierre Jolivet[‡*†]
[‡]Laboratoire J. Kuntzmannn
Université J. Fourier
Grenoble Cedex 9, France
jolivet@ann.jussieu.fr

Frédéric Hecht[*†]
[*]Laboratoire J.-L. Lions
Université P. et M. Curie
Paris, France
hecht@ann.jussieu.fr

Frédéric Nataf[*†]
[†]INRIA
ALPINES research team
Rocquencourt, France
nataf@ann.jussieu.fr

Christophe Prud'homme[§]
[§]IRMA
Université de Strasbourg
Strasbourg Cedex, France
prudhomme@unistra.fr

ABSTRACT

Domain decomposition methods are, alongside multigrid methods, one of the dominant paradigms in contemporary large-scale partial differential equation simulation. In this paper, a lightweight implementation of a theoretically and numerically scalable preconditioner is presented in the context of overlapping methods. The performance of this work is assessed by numerical simulations executed on thousands of cores, for solving various highly heterogeneous elliptic problems in both 2D and 3D with billions of degrees of freedom. Such problems arise in computational science and engineering, in solid and fluid mechanics.

While focusing on overlapping domain decomposition methods might seem too restrictive, it will be shown how this work can be applied to a variety of other methods, such as non-overlapping methods and abstract deflation based preconditioners. It is also presented how multilevel preconditioners can be used to avoid communication during an iterative process such as a Krylov method.

Categories and Subject Descriptors

G.1.8 [**Numerical analysis**]: Partial Differential Equations—*domain decomposition methods, finite element methods*; D.2.13 [**Software engineering**]: Reusable Software—*Reusable libraries*

General Terms

Algorithms, Performance

Keywords

Linear solvers, divide and conquer, scalability

1. INTRODUCTION

Discretizations of partial differential equations used to model physical phenomena typically lead to larger and larger systems that cannot be solved directly and require advanced preconditioning techniques to ensure a fast convergence of iterative methods. As the current trend in high performance computing evolve towards more and more concurrency, recent results using domain decomposition preconditioners [20] and multigrid methods [3, 30] clearly show why these are the methods of choice for achieving high-throughput finite element simulations. Both are examples of the divide and conquer paradigm. On the one hand, for scalar equations such as the simulation of flows through porous media, multigrid methods are often the preferred methods. One the other hand, domain decomposition methods are widely used in solid mechanics, in multiphysics or for code coupling. While these methods can offer high efficiency for solving PDE in parallel, special care must be taken when solving highly heterogeneous and complex systems for ensuring convergence rates independent of the number of computing nodes and heterogeneities involved in the simulation, as shown in figure 1.

There are two main families of domain decomposition algorithms: overlapping Schwarz methods and iterative substructuring methods. While this work is more focused towards the former, the key idea is the same for both methods: to obtain convergence rates which are independent of the number of subdomains, also referred to as substructures, an additional component to provide global transfer of numerical information across all subproblems is needed. In this work, the construction of a so-called coarse operator is presented. In section 2, some important notions about domain decomposition methods are recalled, in section 3, the construction of the coarse operator is presented. Numerical results are gathered in section 3.4, where problems of up to 22 billions unknowns in 2D, and 2 billions unknowns in 3D are solved on more than 16K threads. Finally, it is shown in section 3.5 how one can use coarse grid correction to avoid communication during the iterations of a Krylov method.

2. DOMAIN DECOMPOSITION PRECONDITIONERS

Let $\Omega \subset \mathbb{R}^d$ ($d = 2$ or 3) be a domain whose associated mesh can be partitioned into N non-overlapping meshes

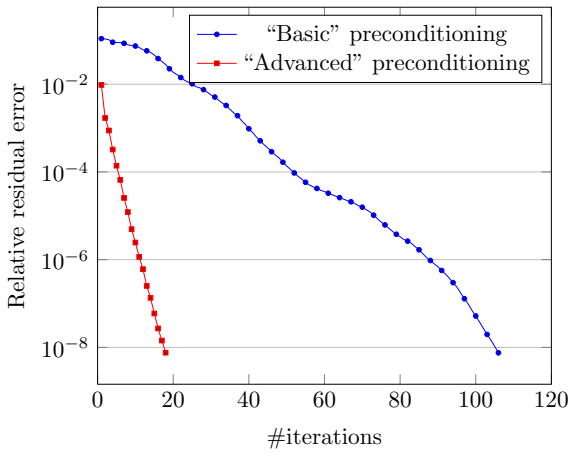


Figure 1: Convergence of the GMRES preconditioned by two domain decomposition methods on 16 subdomains. The "basic" method is oblivious to the heterogeneities in the domain, while the "advanced" method take them into account in the preconditioner.

$\{\mathcal{T}_i\}_{1\leqslant i\leqslant N}$ using graph partitioners such as METIS [18] or SCOTCH [8]. Let V be the finite element space spanned by the finite set of n basis functions $\{\phi_i\}_{1\leqslant i\leqslant n}$ defined on Ω. Typical finite element discretizations of a symmetric, coercive bilinear form $a : V \times V \to \mathbb{R}$ yield the following system to solve:

$$Ax = b, \tag{1}$$

where $(A_{ij})_{1\leqslant i,j\leqslant n} = a(\phi_j, \phi_i)$, and $(b_i)_{1\leqslant i\leqslant n} = (f, \phi_i)$, f being in the dual space V^*. Now let $\{V_i\}_{1\leqslant i\leqslant N}$ be the local finite element spaces defined on the domains associated to each $\{\Omega_i\}_{1\leqslant i\leqslant N}$ where Ω_i is the subdomain defined as the union of all mesh elements in $\mathcal{T}_i$, for all $1 \leqslant i \leqslant N$. If δ is a positive integer, the overlapping decomposition $\left\{\mathcal{T}_i^\delta\right\}_{1\leqslant i\leqslant N}$ is defined recursively as follow: $\mathcal{T}_i^\delta$ is obtained by including all elements of $\mathcal{T}_i^{\delta-1}$ plus all adjacent elements of $\mathcal{T}_i^{\delta-1}$. For $\delta = 0$, $\mathcal{T}_i^\delta = \mathcal{T}_i$. An example of such a construction is given figure 2, for $\delta = 2$.

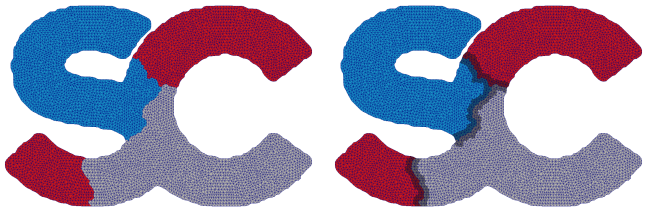


Figure 2: Decomposition of the SC conference logo (no copyrights infringement intended) into three color-coded subdomains. On the left, the decomposition is non-overlapping, $\delta = 0$. On the right, two consecutive extensions are performed, $\delta = 2$, and represented in black overlay, to yield $\left\{\Omega_i^2\right\}_{1\leqslant i\leqslant 3}$.

Now consider the restrictions $\{R_i\}_{1\leqslant i\leqslant N}$ from V to $\left\{V_i^\delta\right\}_{1\leqslant i\leqslant N}$, the local finite element spaces on $\left\{\Omega_i^\delta\right\}_{1\leqslant i\leqslant N}$, and a local partition of unity $\{D_i\}_{1\leqslant i\leqslant N}$ such that

$$\sum_{j=1}^{N} R_j^T D_j R_j = I_{n\times n}\,. \tag{2}$$

Algebraically speaking, if $\{n_i\}_{1\leqslant i\leqslant N}$ denotes the number of degrees of freedom in each local finite element spaces, then R_i is a boolean matrix of size $n_i \times n$, and D_i is a diagonal matrix of size $n_i \times n_i$ for all $1 \leqslant i \leqslant N$. In our experiments, we use a pretty simple partition of unity, already used for example in [19]. Let $\widetilde{\chi}_i$ be a continuous piecewise linear function (hence mesh elements vertices and degrees of freedom coincide) defined on Ω_i^δ as such:

$$\widetilde{\chi}_i = \begin{cases} 1 & \text{on all nodes of } \mathcal{T}_i^0 \\ 1 - \dfrac{m}{\delta} & \text{on all nodes of } \mathcal{T}_i^m \setminus \mathcal{T}_i^{m-1}\ \forall m \in [1;\delta] \end{cases}$$

The local partition of unity is defined as:

$$\chi_i = \frac{\widetilde{\chi}_i}{\displaystyle\sum_{j=1}^{N} \widetilde{\chi}_j|_{V_i^\delta \cap V_j^\delta}}\,,$$

so that the support of the non negative function χ_i is V_i^δ and

$$\sum_{i=1}^{N} \chi_i = 1.$$

Using a linear interpolant from the finite element space of continuous piecewise linear function from Ω_i^δ to V_i^δ (which is typically of higher order), D_i can be obtained from χ_i for all $1 \leqslant i \leqslant N$. For more complex partitions of unity, see for example [6]. Using the partition of unity, a common one-level preconditioner for system (1) introduced in [7] is:

$$\mathcal{P}_{\text{RAS}}^{-1} = \sum_{i=1}^{N} R_i^T D_i (R_i A R_i^T)^{-1} R_i\,. \tag{3}$$

Equation (3) clearly shows the need of the globally assembled matrix, or to be more precise, the need of N assembled submatrices (commonly referred to as "Dirichlet" matrices). These are typically hard to assemble directly and independently with classical finite element packages, in comparison with unassembled matrices (or "Neumann" matrices). Two approaches can be considered to build each "Dirichlet" matrix.

1. Build A, then extract each assembled subproblem. While this is the natural approach, it usually require some communications to build a parallel structure capable of handling distributed degrees of freedom, with ghost elements.

2. Build the stiffness matrices $A_i^{\delta+1}$ yielded by the discretization of a on $V_i^{\delta+1}$, then, remove the columns and rows associated to degrees of freedom lying on elements of $\mathcal{T}_i^{\delta+1} \setminus \mathcal{T}_i^\delta$. This yields $A_i := R_i A R_i^T$, the global assembled matrix A is never assembled.

The second approach does not require any additional parallel information or communication: there is no need for a global ordering, associated to a global partition of the degrees of

freedom. Local matrices A_i are symmetric positive definite, as A is.
Usually, one subdomain is mapped to a single MPI process. For that reason, a given MPI process has a natural access to $u_i := R_i u$ for any function $u \in V$. In all the following, we will assume that a finite element function in V_i^δ (resp. V) can be interpreted as vector of $\mathbb{R}^{n_i}$ (resp. $\mathbb{R}^n$) for all $1 \leqslant i \leqslant N$. In order to compute a global sparse matrix-vector product Ax, one has to notice that, thanks to the partition of unity and the duplication of unknowns on the overlap, it can be proven that:

$$R_i A R_j^T D_j x_j = R_i R_j^T R_j A R_j D_j x_j \,, \tag{4}$$

so that

$$\begin{aligned}(Ax)_i = R_i Ax &= R_i A \sum_{j=1}^{N} R_j^T D_j R_j x \\ &= R_i \sum_{j=1}^{N} R_j^T R_j A R_j D_j R_j x \\ &= \sum_{j=1}^{N} R_i R_j^T A_j D_j x_j \,.\end{aligned} \tag{5}$$

Applying $R_i R_j^T$ to a vector x_j of V_j^δ for all $1 \leqslant i \neq j \leqslant N$ is equivalent to restricting x_j to the degrees of freedom in V_j^δ that are duplicated within V_i^δ, sending the resulting vector to subdomain i, which then prolongates by 0 the received vector outside of the overlap $V_j^\delta \cap V_i^\delta$. In the following, $\mathcal{O}_i$ will be the set of neighboring subdomains to i, i.e. $\left\{j \,:\, j \neq i \text{ and } V_j^\delta \cap V_i^\delta \neq \emptyset\right\}$, and $\overline{\mathcal{O}_i} = \mathcal{O}_i \cup \{i\}$. There is no need to assemble each R_i, one only needs to know the action of $\{R_i R_j^T\}_{1 \leqslant i \leqslant N, j \in \mathcal{O}_i}$.

2.1 Problem specification

It is well known that one-level domain decomposition methods as depicted in the introduction of this section, see (3), do suffer from poor conditioning when used with many subdomains, [25, 28, 32]. Indeed, if the subdomains are assumed to be of size $\mathcal{O}(H)$, then the condition number of the preconditioner grows as $1/H$ for overlapping methods. For a given global mesh, increasing the number of subdomains leads to decreasing H, hence increasing the number of iterations needed for the preconditioned iterative method to converge. To overcome this recurrent problem in overlapping and non-overlapping methods, one must introduce a so-called coarse operator. In this work, an already established coarse operator whose theoretical foundations are presented in [29] is used. From a practical point of view, after building each local solver A_i, three dependent operators are needed :

1. a deflation matrix Z of size $n \times m$, with $m \ll n$,
2. a coarse operator $E = Z^T A Z$ of size $m \times m$,
3. the actual preconditioner
$$\mathcal{P}^{-1}_{\text{A-DEF1}} = \mathcal{P}^{-1}_{\text{RAS}}(I - AZE^{-1}Z^T) + ZE^{-1}Z^T \,, \tag{6}$$
thoroughly studied in [31].

The choice of using $\mathcal{P}^{-1}_{\text{A-DEF1}}$ instead of, for example,

$$\mathcal{P}^{-1}_{\text{A-DEF2}} = (I - AZE^{-1}Z^T)\mathcal{P}^{-1}_{\text{RAS}} + ZE^{-1}Z^T \,, \tag{7}$$

which is also studied in [31] is pretty simple. While both preconditioners have similar numerical properties, applying $\mathcal{P}^{-1}_{\text{A-DEF1}}$ to a vector u requires only one coarse problem solution (used in two different operations afterwards):

$$ZE^{-1}Z^T u \,.$$

On the other hand, applying $\mathcal{P}^{-1}_{\text{A-DEF2}}$ requires two coarse problem solutions:

$$ZE^{-1}Z^T u \qquad \text{and} \qquad ZE^{-1}Z^T \mathcal{P}^{-1}_{\text{RAS}} u \,.$$

Because applying a coarse correction is the most communication-intensive operation when preconditioning an iterative method, as shown in section 3.2, it is best to compute only 1 correction per iteration for scalability purposes.

In overlapping and non-overlapping decomposition methods, the deflation matrix is usually defined as:

$$Z = \begin{bmatrix} R_1^T W_1 & R_2^T W_2 & \cdots & R_N^T W_N \end{bmatrix} \in \mathbb{R}^n \times \mathbb{R}^{\sum_{i=1}^N \nu_i}$$

The question is then, how to choose all those W_i, for all $1 \leqslant i \leqslant N$, to build a numerically scalable preconditioner ? In [29], they are built as:

$$\left\{ W_i = \begin{bmatrix} D_i \Lambda_{i_1} & D_i \Lambda_{i_2} & \cdots & D_i \Lambda_{i_{\nu_i}} \end{bmatrix} \in \mathbb{R}^{n_i} \times \mathbb{R}^{\nu_i} \right\}_{1 \leqslant i \leqslant N} \tag{8}$$

A threshold criterion is used to select the ν_i eigenvectors $\{\Lambda_{i_j}\}_{1 \leqslant j \leqslant \nu_i}$ associated to the smallest eigenvalues in magnitude of the following local generalized eigenvalue problems:

$$A_i^\delta \Lambda_i = \lambda_i D_i R_{i,0}^T R_{i,0} A_i^\delta D_i \Lambda_i \tag{9}$$

where A_i^δ is the matrix yielded by the discretization of a on V_i^δ (unassembled submatrix), and $R_{i,0}$ is the restriction operator from V_i^δ to the overlap $V_i^\delta \cap \left(\cup_{j \in \mathcal{O}_i} V_j^\delta\right)$. If $\Omega_i^\delta \cap \partial\Omega = \emptyset$, Ω_i^δ is commonly referred to as a floating subdomain. Hence, the bilinear form a lacks essential boundary conditions on V_i^δ, so that A_i^δ is now symmetric positive indefinite (compared to A_i, the assembled submatrix, which is always symmetric positive definite under the assumption that A is). These independent and local eigenproblems are solved concurrently in order to find low eigenvalues that are in some sense close the lowest eigenvalues of the preconditioned system. It is well known that those are hurting the convergence rate of traditional Krylov methods, see [23, 11] for further details. With this construction of the coarse operator, it can be proven that the condition number of the preconditioned system is now independent of the size of the subdomains, of the number of subdomains, and of the heterogeneities in the physical coefficients. Once again, interested readers are referred to [29].

3. DESIGN OF THE COARSE OPERATOR

The goal of this section is to explain how a coarse operator built using deflation vectors can be efficiently assembled for large-scale simulations. This work is implemented in a light and versatile C++ framework that is not directly linked to domain decomposition methods, meaning that it is possible to use it to assemble coarse operator with other abstract deflation vectors, for example as defined in [14] for simulations in cosmology. When applicable, the MPI calls and operations related to linear algebra (either from dense BLAS [5] or sparse BLAS such as Intel Math Kernel Library or cuSPARSE) are provided in typewriter font.

3.1 Assembling the coarse operator

Looking at the general formulation of E introduced in the previous section, two global sparse matrix-matrix products must be computed to assemble the coarse operator:

$$AZ \quad \text{then} \quad Z^T(AZ)\,.$$

However, it is possible to exploit the sparsity pattern of Z to build E more efficiently than with these two consecutive computations. By construction, the deflation matrix Z is made of block of dense matrices of size $n_i \times \nu_i$ as displayed in figure 3.

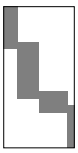


Figure 3: Sparsity pattern of the deflation matrix Z (grey blocks represent nonzero entries) with 4 subdomains.

If there is more than one block for which there are non zeros in a given row of Z, it means that one is dealing with a duplicated unknown (those which are in the overlap). From the simple example in figure 3 with 4 subdomains, one can infer that $\mathcal{O}_1 = \{2\}, \mathcal{O}_2 = \{1,3\}, \mathcal{O}_3 = \{2,4\}$, and $\mathcal{O}_4 = \{3\}$. Just like the globally assembled matrix A, the global deflation matrix Z is never assembled, but its representation above is useful to understand the assembly of E. Instead, each subdomain has access to its local dense matrix W_i. The block (i,j) of E of size $\nu_i \times \nu_j$ is then equal to $W_i^T R_i A R_j^T W_j$. From that algebraic definition, one can see that the sparsity pattern of E is linked to the connectivity between subdomains, because:

$$R_i A R_j^T = 0_{n_i \times n_j} \iff V_i^\delta \cap V_j^\delta = \emptyset\,.$$

In the context of overlapping domain decomposition methods, using equations (4)-(8), the block (i,j) of E can be computed as

$$\begin{aligned} W_i^T R_i A R_j^T W_j &= W_i^T R_i R_j^T R_j A R_j^T W_j \\ &= W_i R_i R_j^T A_j W_j\,. \end{aligned} \tag{10}$$

As in equation (5), the previous equation shows how it is possible to take advantage of the duplicated unknowns on the overlap to compute a global product with only local computations and peer-to-peer transfers : the assembled matrices A and Z are not needed. Noticing that $R_i R_i^T = I_{n_i \times n_i}$, the subdomain i needs to perform three tasks so that all blocks of E can be computed:

1. compute locally $T_i = A_i W_i$ (`csrmm`) and $E_{i,i} = W_i^T T_i$ (`gemm`),
2. send to each neighboring subdomain $j \in \mathcal{O}_i$, $S_j = R_j R_i^T T_i$, and receive from each neighboring subdomain $U_j = R_i R_j^T T_j$,
3. compute locally $E_{i,j} = W_i^T U_j$ (`gemm`).

The cost of step 2 is approximately the same as one global sparse matrix-vector product. Using the same toy problem as in figure 3, the following figure is a representation of the sparsity pattern of E. Two colors are used to differentiate blocks that can be computed without any communication and those that cannot.

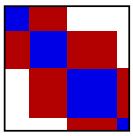


Figure 4: Sparsity pattern of the coarse operator E. Blue blocks involve only local computations (step 1) while red blocks also involve peer-to-peer transfers (steps 2 and 3).

The three tasks to assemble all blocks of E can be customized to suit one's needs using C++ polymorphism. For example, in the context of non-overlapping methods, the sparsity pattern of E is typically more dense: a block (i,j) of E is not null if and only if $j \in \mathcal{O}_j$, but also if there exists $k \in \mathcal{O}_j$ such that $k \in \mathcal{O}_i$. This can be handled by our framework. Likewise, each local computations T_i as introduced in the first step, can be modified to involve more complex operations and communications patterns.

Stopping at that point in the construction of E, solving systems involving the coarse operator would require either to factorize a relatively small matrix on a large number of MPI processes, or to call an iterative method with once again, a distribution of data that would lead to a too fine-grained granularity (blocks of rows of E are typically of size ν_i ranging from 1 to 30). In both cases, the increased communication overhead would lead to bad performances when solving multiple systems involving the coarse operator. Another approach would consist in replicating E over all subdomains, and then performing independent factorizations, but that is simply not feasible for large decompositions, with arbitrary numbers of deflation vectors. In the following subsections, a more efficient data distribution is explained. While it will imply more communications during the setup of the coarse operator, it will cover all possible issues stated in this paragraph, regarding communication overhead, memory consumption and fast computation of coarse solutions.

3.1.1 *Master-slave approach*

The idea is to use only a "small" group of processes that will be in charge of factorizing the coarse operator and that will afterwards be called for computing solutions of systems involving E^{-1} using a distributed sparse direct solver. This is inspired by the famous *master-slave* approach. For the rest of the paper, the following notations will be thoroughly used (for the sake of completeness, we provide their type in our implementation):

- `unsigned short P`: the number of masters, chosen at runtime by the user,
- `MPI_Comm masterComm`: a communicator between all masters, set to `MPI_COMM_NULL` on slaves, on which will be instantiated the distributed solver,
- `MPI_Comm splitComm`: a communicator between a master and its slaves in which the rank of the master is always 0, and the ranks of the slave follow the same order as in `MPI_COMM_WORLD`.

A representation of such communicators is given figure 5. Prior to factorizing E, the first step is to assemble it in a

distributed matrix on the masters. Each master will be in charge of assembling all the values of its slaves. It is assumed that the format in which the distributed matrix is stored is a simple global CSR or global COO — the standard format for most linear solvers available nowadays, meaning that for each non-zero value, one must know the absolute row and column indices of the given value in the global matrix E. For a process i, the global row indices I_{E_i} of all the blocks $\{E_{i,j}\}_{j\in\overline{\mathcal{O}_i}}$ range from r_i to $r_i+\nu_i$, where $r_i = \sum_{j=1}^{i-1}\nu_j$, and the global column indices J_{E_j} of all the blocks $\{E_{j,i}\}_{j\in\overline{\mathcal{O}_i}}$ range from r_i to $r_i+\nu_i$. The simplest approach would then be to:

1. call `MPI_Allgather`(ν_i) on `MPI_COMM_WORLD` to be able to compute the cumulative sums $\{r_j\}_{j\in\overline{\mathcal{O}_i}}$ and to allocate the buffers S_j and U_j for all $j \in \mathcal{O}_i$,
2. assemble locally $\{E_{i,j}\}_{j\in\overline{\mathcal{O}_i}}$ as previously and store the values in CSR (or COO): $(I_{E_i}, J_{E_i}, val_{E_i})$,
3. call `MPI_Gatherv`(I_{E_i}), `MPI_Gatherv`(J_{E_i}), and `MPI_Gatherv`(val_{E_i}) on `splitComm` with rank 0 as root.

The number of non-zero values for a process i is:

$$\texttt{size}(val_{E_i}) = \underbrace{\nu_i \times \nu_i}_{\text{Blue values in figure 4}} + \underbrace{\nu_i \times \sum_{j\in\mathcal{O}_i} \nu_j}_{\text{Red values in figure 4}}, \quad (11)$$

meaning that prior to the three `MPI_Gatherv` in step 3, a call to `MPI_Gatherv`($\mathcal{O}_i$) on `splitComm` with rank 0 as root must be made to allocate the right buffers for the distributed format I_E, J_E, and val_E on each master. While this approach is somehow natural for assembling the distributed matrix on the masters — because of the ordering of the rank of the slaves, calling `MPI_Gatherv` is similar to concatenating all local chunks of E — it implies a lot of (unnecessary) communications. In particular, why should slaves send to masters the global row and column indices ? Indeed, at the end of assembly, only the masters have access to the distributed coarse operator, so it is their responsability to compute the indices. The slaves should not have to store or compute anything related to the distributed format. The following approach has the advantage of transferring only what is needed from one slave i to its master: the array of `double` val_{E_i}. The indices will be computed after reception by each master, meaning that the memory overhead on the slaves is null (no integers are allocated for storing any index). The new workflow is now:

1. perform a neighborhood collective operation `MPI_Ineighbor_alltoall`(ν_i)[1] on the communicator to which the distributed graph topology information of the connectivity between subdomains is attached (`MPI_Dist_graph_create_adjacent`). Then allocate accordingly the buffers S_j and U_j for all $j \in \mathcal{O}_i$
2. call `MPI_Gather`($[\nu_i, |\mathcal{O}_i|]$) (array of 2 integers) on `splitComm` with rank 0 as root so that masters can preallocate the distributed CSR (I_E, J_E, val_E),
3. assemble locally $\{E_{i,j}\}_{j\in\overline{\mathcal{O}_i}}$ as previously and send the values to the master. Prepend to the beginning of the message, the values of $\mathcal{O}_i$, i.e. the final size of the message is $|\mathcal{O}_i| + (11)$.

[1] new to the MPI-3 standard

Additionally, the masters must concatenate all ν_i gathered in step 2, using `MPI_Allgatherv` on `commMaster` to be able to compute all cumulative sums r_i, for all $i \in$ `splitComm`. This call is equivalent to the `MPI_Allgather` in step 1 of the "natural" algorithm, but this time it doesn't involve any slave. When a master receives a message from a slave i, it knows that the global row index ranges from r_i to $r_i+\nu_i$, and because the first values of the received message are a copy of $\mathcal{O}_i$, it can compute the correct global column indices for the neighbors of this slave. The complete algorithm for assembling the coarse operator is summarized in algorithms 1 (construction of all blocks of E) and 2 (distributed assembly on the masters).

3.1.2 *Electing the masters*

In the previous paragraph, the masters are defined in a rather abstract way, as the processes that have a rank equal to 0 in `splitComm`. We have two ways to define the aforementioned MPI communicator. The first is the natural distribution: the process are spread uniformly and contiguously into P groups, the masters are of rank $i \cdot N/P$, for all $0 \leqslant i \leqslant P-1$. The second distribution is a little more advanced and better suited for assembling symmetric coarse operators. In that case, one only needs to assemble the upper part of the distributed CSR or COO, so that only the following blocks are computed and assembled:

$$E_{i,j},\ \forall 1 \leqslant i \leqslant N,\ \forall j \in \overline{\mathcal{O}_i}\ :\ j \geqslant i\,.$$

Moreover, only the upper parts of the dense diagonal blocks $E_{i,i}$ for all $1 \leqslant i \leqslant N$ are needed. To ensure load balancing between masters, the processes are now spread contiguously but non-uniformly, with masters of rank p_i, where p_i is defined by the following sequence to ensure heuristically that the number of values within each quadrilateral in figure 5 is the same:

$$p_0 = 0$$
$$p_i = \lfloor N - \sqrt{(p_{i-1}-N)^2 - N^2/P} + 0.5 \rfloor,\ 1 \leqslant i \leqslant P-1$$

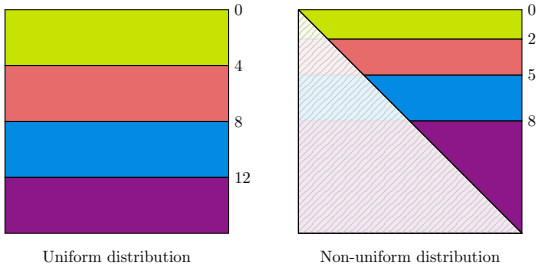


Figure 5: Distribution of E when build with 16 subdomains using 4 masters. Each color represents a different `splitComm`, each number represents the rank of the master (in `MPI_COMM_WORLD`) of a given `splitComm`. On the right, the number of values per `splitComm` is roughly the same if the values below the diagonal are dropped (symmetric coarse operator).

Algorithm 1 Schematic construction of $E_{i,j}$, for all $1 \leqslant i \leqslant N$ and $j \in \overline{\mathcal{O}_i}$

```
   MPI_Ineighbor_alltoall(ν_i)
 2: MPI_Gather([ν_i, |O_i|], splitComm, 0)
   compute(T_i)                                   ▷ T_i = A_i W_i
 4: for j ∈ O_i do                          ▷ After completion of line 1
     MPI_Isend(S_j, j, MPI_COMM_WORLD)        ▷ S_j = R_j R_i^T T_i
 6:  MPI_Irecv(U_j, j, MPI_COMM_WORLD, rq[j])
   end for
 8: compute(E_{i,i})                                ▷ Diagonal block
   for j ∈ O_i do
10:  MPI_Waitany(rq, &index)
     compute(E_{i,index})                          ▷ Off-diagonal block
12: end for
```

Algorithm 2 Schematic assembly of E on the masters

```
   buildComm(P, splitComm, masterComm)
14: if masterComm != MPI_COMM_NULL then                    ▷ Master
     MPI_Allgatherv(ν_i)
16:                  ▷ Now receive all messages from the slaves
     for j = 1, ..., MPI_Comm_size(splitComm) do
18:    MPI_Irecv(msgFromSlave[j], j − 1, splitComm, rq[j])
     end for
20:  assemble(E_{i,i})
     for k = 1, ..., |O_i| do
22:    assemble(E_{i,k})
     end for
24:    ▷ Blocks local to the masters have been assembled
     for j = 1, ..., MPI_Comm_size(splitComm) do
26:    MPI_Waitany(rq, &index)
       assemble(E_{index,index});
28:    for k = 1, ..., |O_index| do
         assemble(E_{index,msgFromSlave[index][k]})
30:    end for
     end for
32:    ▷ Blocks from the slaves have been assembled
     numericalFactorization(E)
34: else                                                   ▷ Slave
     msgToMaster = O_i
36:  concatenate(msgToMaster, E_{i,i})
     for k ∈ O_i do
38:    concatenate(msgToMaster, E_{i,k})
     end for
40:  MPI_Isend(msgToMaster, 0, splitComm)
     ▷ Send the index of the neighbors as well as the local
   rows of E computed in algorithm 1
42: end if
```

The procedure `compute` in algorithm 1 is in charge of returning a dense array of values corresponding to the block of E given in argument, while the procedure `assemble` in algorithm 2 is in charge of computing the indices of the values of the block of E given in argument and store them in the distributed matrix representation.

3.2 Applying a coarse operator correction

Once E has been assembled, it is involved in the solution of the following problem for a given vector u: $ZE^{-1}Z^Tu$. It is assumed that the right-hand side Z^Tu and the solution $E^{-1}Z^Tu$ of the coarse problem are kept distributed on `masterComm` at all time, as input and ouput of the distributed solver. The computation of the correction can obviously be broken down into three basic operations:

1. compute $Z^Tu = w \in \mathbb{R}^{\sum_{i=1}^{N}\nu_i}$. Once again, the structure of Z makes it possible to compute this sparse matrix-vector product without any explicit global representation of Z. Indeed,

$$Z^Tu = \sum_{i=1}^{N} W_i^T R_i u$$

   This is evaluated by having all subdomains i compute locally $w_i = W_i^T u_i$ (`gemv`), and calling `MPI_Gather(v)` on each `splitComm` with rank 0 as root to assemble w with all local contributions of size ν_i, for all i in `splitComm`, on the masters.

2. compute $E^{-1}w = y \in \mathbb{R}^{\sum_{i=1}^{N}\nu_i}$. This operation must be as fast and reliable as possible, since it is carried out by only few masters. The numerical factorization of E as computed in the previous paragraph during the assembly phase is reused for each forward elimination and back substitution.

3. compute $Zy = z \in \mathbb{R}^{\sum_{i=1}^{N} n_i}$. This is the exact dual of step 1. This operation reads:

$$Zy = \sum_{i=1}^{N} R_i^T W_i y_i$$

   First, a call to `MPI_Scatter(v)` is made on each `splitComm` with rank 0 as root so that each subdomain i can retrieve its y_i of size ν_i from its master. Then they all compute $z_i = W_i y_i$ (`gemv`). Finally,

$$(Zy)_i = \sum_{j=1}^{N} R_i R_j^T z_j \tag{12}$$

   is computed using the same communication procedure as for the sparse matrix-vector product, see (5).

Using this construction, it is clear that using a non-uniform criterion ν_i for each subdomain leads to using MPI communications with varying counts of data from each process (hence the `v` in parentheses). Because these communications scale as $\mathcal{O}(N)$, it is preferable to call prior to assembling the coarse operator `MPI_Allreduce(`ν_i`,MPI_MAX)`. That way, it is possible to use MPI communications with equal counts of data, which typically scale as $\mathcal{O}(\log(N))$. Moreover, the theoretical estimate on the condition number of the preconditioner system remains valid. There are also multiple simplifications possible during the assembly of E, because using a uniform criterion means that any process i already knows the value of ν_j, for all $j \in \mathcal{O}_i$ without needing the call to `MPI_Ineighbor_alltoall` and such.

3.3 Cost analysis

As a consequence of the observation made in the previous paragraph, it is assumed in this section that the number of deflation vectors per subdomain is chosen uniformly and set to ν. This will also be the case in section 3.4 for our scaling

experiments, and we now propose a comprehensive summary of the cost of our two-level preconditioner, compared to a simple one-level method. As a quick reminder, the cost of applying a one-level preconditioner such as $\mathcal{P}_{\text{RAS}}^{-1}$ in (3) is the same as computing N independent solutions on each subdomain – application of $(R_i A R_i^T)^{-1}$, then performing one global matrix-vector product.

Memory footprint. Because it is clear that no global structure is needed for the construction and the use of Z and E in our implementation, the memory footprint is straightforward to evaluate. For slaves, a dense array of size $\nu \times n_i$ is needed to store W_i. For masters, the same array must be stored, plus the factors of E^{-1} (whose size depends on the distributed solver used).

Messages. During the construction of E, each process i must send and receive a single message from each neighboring subdomains $j \in \mathcal{O}_i$. The size of theses messages are $\nu \times$ `size of overlap between` V_i^δ `and` V_j^δ, see S_j and U_j in algorithm 1. Then each slave i must send a single message of `double`, of size $|\mathcal{O}_i| + |\overline{\mathcal{O}_i}| \times \nu \times \nu$ to its master, see `msgToMaster` in algorithm 2, and each master must reciprocally receive these message for all slaves in `splitComm`. During the solution, at each iteration, a `MPI_Gather` and a `MPI_Scatter` are called on each `splitComm`. The messages passed at the end of the correction step in equation (12) are of the exact same sizes as for the global matrix-vector product for the one-level method, c.f. (5).

Computational intensity. The major contribution is the solution of each local and independant eigenvalue problem. Afterwards, for the setup of E, the BLAS computations stated in section 3.1 are negligible with respect to the time spent transferring messages, and at each iteration the costly operation in section 3.2 is the solution of the coarse correction – application of E^{-1} using a distributed direct solver. Note that for big subdomains, this cost is negligible compared to the one of computing each local solution for the one-level preconditioner $\mathcal{P}_{\text{RAS}}^{-1}$, c.f. (3).

3.4 Numerical results

Results in this section were obtained on Curie, a Tier-0 system for PRACE[2] composed of 5040 nodes made of 2 eight-core Intel Sandy Bridge processors clocked at 2.7 GHz. The interconnect is an InfiniBand QDR full fat tree and the MPI implementation used was BullxMPI version 1.1.16.5. Intel compilers and Math Kernel Library in their version 13.1.0.146 were used for all binaries and shared libraries, and as the linear algebra backend for both dense and sparse computation in our framework. LU or LDL^T decomposition of the local problems are computed with either instances of MUMPS [1, 2] or PaStiX [17] on `MPI_COMM_SELF`, or preferably using Intel MKL PARDISO, University of Lugano PARDISO [26, 27], or WSMP [16, 15] which are by design more suited for multithreaded computations. LU or LDL^T decomposition of the coarse operator is computed using MUMPS, PaStiX, or PWSMP. Eigenvalue problems to compute the deflation vectors as defined in (9) are solved using ARPACK [21]. Finite element matrices are obtained from FreeFem++, but the design of our domain decomposition framework makes it simple to port to other Domain-Specific (Embedded) Language such as Feel++ [24], FEniCS [22] or finite element libraries like deal.II [4], GetFem++ and such. We display the speedup and efficiency in terms of number of MPI processes. In these experiments, each MPI process is assigned a single subdomain, and 2 OpenMP threads. The GMRES is stopped when a relative 10^{-6} decrease of the residual is reached.

[2]Partnership for Advanced Computing in Europe

First, the system of linear elasticity with highly heterogeneous elastic moduli is solved with a minimal geometric overlap of one mesh element. Its variational formulation reads:

$$a(u,v) = \int_\Omega \lambda \nabla \cdot u \nabla \cdot v + 2\mu\varepsilon(u) : \varepsilon(v) + \int_\Omega f \cdot v + \int_{\partial\Omega} g \cdot v$$

where

- λ and μ are the Lamé parameters such that $\mu = \dfrac{E}{2(1+\nu)}$ and $\lambda = \dfrac{E\nu}{(1+\nu)(1-2\nu)}$ (E being Young's modulus and ν Poisson's ratio). They are chosen to vary between two sets of values, $(E_1, \nu_1) = (2 \cdot 10^{11}, 0.25)$, and $(E_2, \nu_2) = (10^7, 0.45)$.
- ε is the linearized strain tensor, f are the body forces (in this case, only the gravity), and g are the surface force (in this case, a vertical loading is imposed on some parts of the geometries).

Such an equation typically arises in computational solid mechanics, for modeling small deformations of bodies. In 2D, we use piecewise cubic basis functions ($\sim$ 33 nnz per row). The system is of constant size equal to approximately 2 billions unknowns. In 3D, piecewise quadratic basis functions are used ($\sim$ 83 nnz per row). The system is of constant size equal to approximately 300 millions unknowns. Both geometries are displayed in figure 6 and were meshed by Gmsh [12] and partitioned with METIS.
After the partitioning step, each local mesh is refined concurrently by splitting each triangle or tetrahedron into multiple smaller elements. This means that we start the simulation with a relatively "coarse" global mesh (26 million triangles in 2D, 10 million tetrahedra in 3D), which is then refined in parallel (thrice in 2D, twice in 3D). We get a nice speedup from $1024 \times 2 = 2048$ to $8192 \times 2 = 16384$ threads as shown in figure 8. According to the table in figure 8, the costly operations in the construction of the preconditioner are the solution of each local eigenvalue problem (9) (column *deflation*), and the factorization of each local solver A_i (column *factorization*). In 3D, the complexity of such operations typically grows superlinearly with respect to the number of unknowns. That explains why we can achieve superlinear speedup. At peak performance, on 16384 threads, the speedup relative to the runtime on 2048 threads equals $\dfrac{530.56}{51.76} \approx 10$. In 2D, the computation costs are lower, and tend to scale better with the number of unknowns, which

Figure 6: Tripod used for our 3D and cantilever used for our 2D strong scaling experiments. Black and light grey are used to represent the variations of the Young's modulus (200 GPa and 0.01 GPa) and Poisson's ratio (0.25 and 0.45).

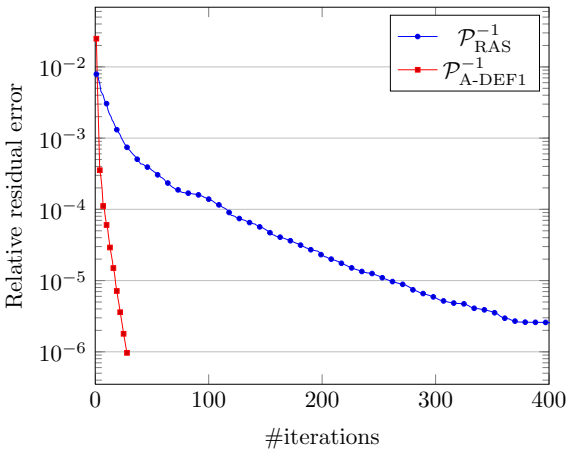


Figure 7: Convergence of the GMRES(40) preconditioned by $\mathcal{P}^{-1}_{\text{RAS}}$ and $\mathcal{P}^{-1}_{\text{A-DEF1}}$ for the problem of linear elasticity in 2D using 1024 subdomains. Timings for the setup and solution phases using $\mathcal{P}^{-1}_{\text{A-DEF1}}$ are available in figure 8. Using $\mathcal{P}^{-1}_{\text{RAS}}$, the convergence is not reached after 600 seconds.

makes it harder to achieve high speedup for larger number of subdomains. At peak performance, on 16384 threads, the speedup relative to the runtime on 2048 threads equals $\frac{213.20}{34.54} \approx 6$. In both cases, the solution of the eigenproblem is the limiting factor for achieving better speedups. This can be explained by the fact that the Arnoli method, which ARPACK is based on, tends to perform better for larger ratios $\frac{n_i}{\nu_i}$, but these values decrease as the subdomains get smaller. We will get back to this issue in the conclusion. For larger but fewer subdomains, the time to compute the solution (column *solution*), i.e. the time for the GMRES to converge, is almost equal to the forward eliminations and back substitutions in the subdomains times the number of iterations. When the decompositions become bigger, subdomains are smaller, hence each local solution is computed faster and global communications have to be taken into account. For a comparison similar to the one in figure 1 in this bigger configuration, we display in figure 7 the convergence histogram of a simple one-level method versus this two-level method. One can easily understand that, while the cost of building the preconditioner cannot be neglected, it is necessary to ensure the convergence of the Krylov method.

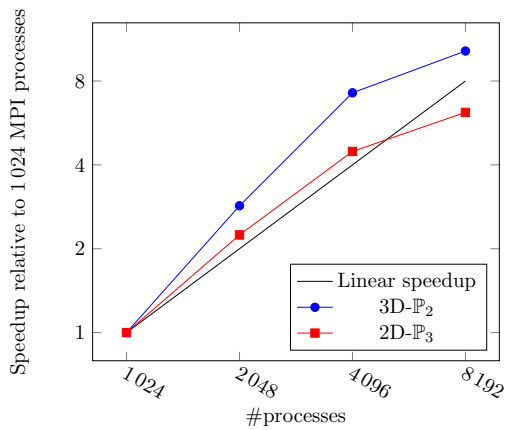
Moving on to the weak scaling properties of our framework, the problem we now solve is a scalar equation of diffusivity with highly heterogeneous coefficients (varying from 1 to $3 \cdot 10^6$ as displayed in figure 9) on $[0;1]^d$ ($d = 2$ or 3) with piecewise quartic basis functions in 2D (~ 23 nnz per row), and piecewise quadratic basis functions in 3D (~ 27 nnz per row). Its variational formulation reads:

$$a(u,v) = \int_\Omega \kappa \nabla u \cdot \nabla v + \int_\Omega f \cdot v$$

where f is a source term. Such an equation typically arises for modeling flows in porous media, or in computational fluid dynamics. No change is needed in our framework, since all operations are algebraic. The only work needed outside of our framework is changing the mesh used for computations, as well as the variational formulation of the problem in the FreeFem++ DSL.

On average, there is a constant number of degrees of freedom per subdomain equal to 280K in 3D, and near 2.7 millions in 2D. As for the strong scaling experiment, after building and partitioning a global "coarse" mesh (with few millions of elements), each local mesh is refined independently to ensure a constant size system per subdomain as the decomposition becomes bigger. The efficiency remains near the 90% mark, thanks to almost no variability in the factorization of the local problems and the construction of the deflation vectors. In 3D, the initial problem of 74 millions unknowns is solved in 200 seconds on 512 threads. Using 16384 threads, the problem is now made of approximately 2.3 billions unknowns, and it is solved in 215 seconds, which yields an efficiency of $\frac{200 \cdot 2.3 \cdot 10^9}{215 \cdot 7.4 \cdot 10^7 \cdot 32} \approx 90\%$. In 2D, the initial problem of 695 millions unknowns is solved in 175 seconds on 512 threads. Using 16384 threads, the problem is now made of approximately 22.3 billions unknowns, and it is solved in 187 seconds, which yields an efficiency of $\frac{175 \cdot 2.2 \cdot 10^{10}}{187 \cdot 6.7 \cdot 10^8 \cdot 32} \approx 96\%$. At such scales, the most penalizing step in the algorithm is the construction of the coarse operator, specially in 3D, with a non negligible increase in the time spent to assemble E.

Finally, we present in this last paragraph the performances of our framework to assemble and factorize the coarse operator E for all the previous simulations. Tables in figures 8



	N	Factorization	Deflation	Solution	#it.	Total	#d.o.f.
3D	1 024	177.86 s	264.03 s	77.41 s	28	530.56 s	$293.98 \cdot 10^6$
	2 048	62.69 s	97.29 s	20.39 s	23	186.04 s	
	4 096	19.64 s	35.70 s	9.73 s	20	73.12 s	
	8 192	6.33 s	22.08 s	6.05 s	27	51.76 s	
2D	1 024	37.01 s	131.76 s	34.29 s	28	213.20 s	$2.14 \cdot 10^9$
	2 048	17.55 s	53.83 s	17.52 s	28	95.10 s	
	4 096	6.90 s	27.07 s	8.64 s	23	47.71 s	
	8 192	2.01 s	20.78 s	4.79 s	23	34.54 s	

Figure 8: Strong scaling experiments.

and 10 already included these timings in their last column *total* (> *factorization* + *deflation* + *solution*), but for more in depth analysis, they are reported next separately.
The figure 11 includes all timing relative to algorithms 1 and 2 described in section 3.1.1: the construction of the communicators, the assembly of E, and its numerical factorization. The most consuming part of the algorithm is the actual transfer and the assembly by the masters. Especially in 3D, when the coarse operator is becoming less and less sparse (directly linked with the average value of $|\mathcal{O}_i|$), it is likely to become a problem for even larger decomposition. Note that the MPI implementation used for these experiments is not thread compliant (and in particular, does not support a level of thread support equal to `MPI_THREAD_MULTIPLE`), meaning that some "unnecessary" `#pragma omp critical` had to be used during the assembly by the masters. At these scales, another problem is the factorization of E. Indeed, increasing the number of masters P does not always have a beneficial effect for this concern, because distributed solvers have difficulties scaling beyond ~ 128 processes.

3.5 Communication avoiding multilevel preconditioners

Classical Krylov methods require global synchronizations during the orthogonalization and the normalization of the basis vectors at each iteration that can hamper parallel scalability. Efforts have been made to reduce the number of such global operations [10, 9], and recent alternatives, such as the p^1-GMRES introduced in [13], predict considerable improvement in speedups because they can overlap computations of sparse matrix-vector products and global synchronizations. For further references in this paper, it is best to quote the

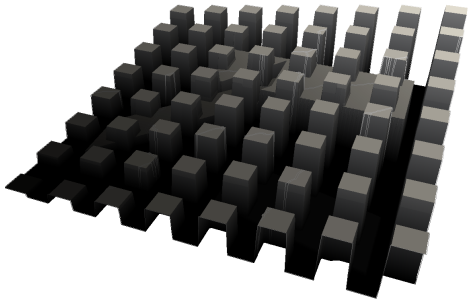


Figure 9: Diffusivity κ used for our 2D weak scaling experiment with channels and inclusions. Black and light grey are used to represent the variations of κ from 1 to $3 \cdot 10^6$.

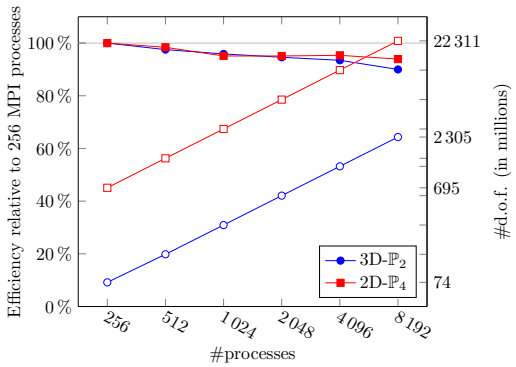
actual computational loop of the p^1-GMRES.

Algorithm Main loop of the p^1-GMRES introduced in [13]

1: **for** $i = 0, \dots, m$ **do**
2: $w \leftarrow Av_i$
3: **if** $i > 1$ **then**
4: $v_{i-1} \leftarrow v_{i-1}/h_{i-1,i-2}$ $z_i \leftarrow z_i/h_{i-1,i-2}$
5: $w \leftarrow w/h_{i-1,i-2}$ $h_{i-1,i-1} \leftarrow h_{i-1,i-1}/h^2_{i-1,i-2}$
6: $h_{j-1,i-1} \leftarrow h_{j-1,i-1}/h^2_{i-1,i-2},\ j = 0, \dots, i-2$
7: **end if**
8: $z_{i+1} \leftarrow w - \sum_{j=0}^{i-1} h_{j,i-1} z_{j+1}$
9: **if** $i > 0$ **then**
10: $v_i \leftarrow z_i - \sum_{j=0}^{i-1} h_{j,i-1} v_j$ $h_{i,i-1} \leftarrow ||v_i||_2$
11: **end if**
12: $h_{j,i} \leftarrow \langle z_{i+1}, v_j \rangle,\ j = 0, \dots, i$
13: **end for**

When using a multigrid or domain decomposition based preconditioner, line 2 becomes $w \leftarrow \mathcal{P}^{-1}_{\text{A-DEF1}} A v_i$ and it gets harder to hide the latency of the global reductions line 10 and 12 because applying the preconditioner might require a global synchronization (induced by the solution of a coarser problem).
Applying the preconditioner $\mathcal{P}^{-1}_{\text{A-DEF1}}$ built in this section requires one `MPI_Gather(v)` on each `splitComm`, followed by a coarse solution, followed by one `MPI_Scatter(v)` on each `splitComm` with rank 0 as root. While this shows that there is no global communication, data dependencies introduce a "virtual" global synchronization (using non-blocking collective operations is not possible). Moreover, the two global communications on each `splitComm` involve mainly short messages if the number of deflation vectors is low (typically $\nu_i < 30$ for each subdomain), meaning that passing extra data is unlikely to hurt too much the execution time of these operations. This is why we propose a *fused* p^1-GMRES, where:

- no global reduction is performed line 10 and 12 (only local computations),
- during the call to `MPI_Gather(v)` at the i-th iteration of the GMRES line 2 on each `splitComm` with rank 0 as root, each process also sends to its master local values of $h_{i-1,i-2}$ and $h_{j,i-1}$ for $j = 0, \dots, i-1$ (these are the non-reduced values of the previous iteration),



	N	Factorization	Deflation	Solution	#it.	Total	#d.o.f.
3D	256	64.24 s	117.74 s	15.81 s	13	200.57 s	$74.62 \cdot 10^6$
	512	63.97 s	112.17 s	19.93 s	18	199.41 s	$144.70 \cdot 10^6$
	1 024	63.22 s	118.58 s	16.18 s	14	202.40 s	$288.80 \cdot 10^6$
	2 048	59.43 s	117.59 s	21.34 s	17	205.26 s	$578.01 \cdot 10^6$
	4 096	58.14 s	110.68 s	27.89 s	20	207.47 s	$1.15 \cdot 10^9$
	8 192	54.96 s	116.64 s	23.64 s	17	215.15 s	$2.31 \cdot 10^9$
2D	256	29.40 s	111.35 s	25.71 s	29	175.85 s	$695.96 \cdot 10^6$
	512	29.60 s	111.52 s	27.99 s	28	179.07 s	$1.39 \cdot 10^9$
	1 024	29.43 s	112.18 s	33.63 s	28	185.16 s	$2.79 \cdot 10^9$
	2 048	29.18 s	112.23 s	33.74 s	28	185.20 s	$5.58 \cdot 10^9$
	4 096	29.80 s	113.69 s	31.02 s	26	185.38 s	$11.19 \cdot 10^9$
	8 192	29.83 s	113.81 s	30.67 s	25	187.57 s	$22.31 \cdot 10^9$

Figure 10: Weak scaling experiments.

- prior to calling the coarse solver, each master reduces the values received after the call to `MPI_Gather(v)` and then launches a `MPI_Iallreduce` on `masterComm`. It is then time for them to compute the solution of the coarse problem, after which they wait for the non-blocking reduction to finish,
- eventually, the reduced values of $h_{i-1,i-2}$ and $h_{j,i-1}$ for $j = 0, \ldots, i-1$ are scattered back with the coarse solution using a single `MPI_Scatter(v)` on each `splitComm` with rank 0 as root.

Each iteration of this *fused* p^1-GMRES can be seen as a coarse operator correction, without a single additional global communication or synchronization and only a single additional call to `MPI_Iallreduce` between the masters on `masterComm`, whose size is greatly smaller than the number of subdomains N. The communications induced by the `MPI_Iallreduce` can be overlapped with the computation of the coarse solution. While this approach looks promising, our test cases are too small and the numbers of iterations are relatively small, so that both pipelined GMRES are performing approximately the same as the reference GMRES. Such a direction of research is however becoming essential as the concurrency of modern supercomputers is increasing tremendously, and minimizing synchronizations is becoming essential for large decompositions, as shown by theoretical models in [13] for the unpreconditioned GMRES.

4. CONCLUSION AND OUTLOOK

In this paper, we assess the efficiency of our implementation of a new adaptative two-level preconditioner suited for various problems such as linear elasticity or Darcy's law with high-contrast coefficients. The preconditioner was recently theoretically introduced in [29] and it provides a sound background for evaluating the performance of our portable framework. Using state of the art multithreaded direct linear solvers and eigensolvers and distributed direct linear solvers, we show experimentally that our approach is well suited for large-scale simulations in both 2D and 3D with high-order finite element methods, on up to 16384 threads, enabling simulations for two classes of elliptic problems of more than 22 billion unkowns in 2D and 2 billion unkowns in 3D.
For building such a preconditioner, the deflation vectors used in our coarse operator are currently determined *a priori* by solving independently local eigenvalue problems. It is also possible to retrieve them *a posteriori* during the convergence of the iterative method, using for example approximations of the Ritz vectors. Using new theoretical results, we are currently evaluating a new *a posteriori* construction that doesn't require the solution of the aforementioned eigenvalue problems. This will hopefully improve our results for more fine-grained granularity in our strong scaling experiments. Finally, we are currently investing other types of systems for which domain decomposition methods have been proven to be efficient, namely nonlinear problems in computational solid mechanics, using non-overlapping methods. Thanks to the versatile design of our framework, and the flexibility of current finite element Domain-Specific (Embedded) Languages such as FreeFem++, new experiments should be soon possible.

3D

N	P		dim(E)		$\lvert\mathcal{O}_i\rvert$ (average)		nnz(E^{-1})		Time	
256	4		5120		11.5		$2.6 \cdot 10^6$		2.78 s	
512	6		10240		12.4		$7.7 \cdot 10^6$		3.35 s	
1024	8	8	20480	22528	13.0	12.0	$24.5 \cdot 10^6$	$14.6 \cdot 10^6$	4.42 s	11.25 s
2048	12	12	40960	40960	13.8	12.9	$74.6 \cdot 10^6$	$34.5 \cdot 10^6$	6.91 s	5.68 s
4096	18	22	73728	73728	14.2	13.7	$143.0 \cdot 10^6$	$85.3 \cdot 10^6$	10.75 s	8.04 s
8192	64	48	131072	131072	14.7	14.6	$285.4 \cdot 10^6$	$196.7 \cdot 10^6$	19.92 s	17.30 s

2D

N	P		dim(E)		$\lvert\mathcal{O}_i\rvert$ (average)		nnz(E^{-1})		Time	
256	2		5376		5.5		$959.7 \cdot 10^3$		9.39 s	
512	4		10240		5.6		$2.1 \cdot 10^6$		9.96 s	
1024	10	8	20480	24576	5.7	5.5	$5.0 \cdot 10^6$	$5.5 \cdot 10^6$	9.92 s	10.14 s
2048	14	12	38912	40960	5.8	5.7	$10.8 \cdot 10^6$	$9.3 \cdot 10^6$	10.05 s	6.20 s
4096	22	18	81920	73728	5.9	5.8	$28.4 \cdot 10^6$	$17.7 \cdot 10^6$	10.87 s	5.10 s
8192	36	36	163840	122880	5.9	5.8	$73.3 \cdot 10^6$	$30.4 \cdot 10^6$	13.27 s	6.96 s

Figure 11: Timings for assembling the coarse operator. Results are gathered two-by-two, the first column is for the diffusivity problem, the second is for the elasticity problem.

5. ACKNOWLEDGMENTS

This work has been supported in part by ANR through COSINUS program (project PETALh no. ANR-10-COSI-0013 and project HAMM no. ANR-10-COSI-0009). It was granted access to the HPC resources of TGCC@CEA made available within the Distributed European Computing Initiative by the PRACE-2IP, receiving funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement RI-283493. The first author would like to thank A. Gupta and O. Schenk for providing him suitable licenses of their respective software (WSMP and PARDISO).

6. REFERENCES

[1] P. Amestoy, I. Duff, J.-Y. L'Excellent, and J. Koster. A fully asynchronous multifrontal solver using distributed dynamic scheduling. *SIAM Journal on Matrix Analysis and Applications*, 23(1):15–41, 2001.

[2] P. Amestoy, A. Guermouche, J.-Y. L'Excellent, and S. Pralet. Hybrid scheduling for the parallel solution of linear systems. *Parallel computing*, 32(2):136–156, 2006.

[3] A. Baker, R. Falgout, T. Kolev, and U. Yang. Scaling *hypre*'s multigrid solvers to 100,000 cores. In *High-Performance Scientific Computing*, pages 261–279. Springer, 2012.

[4] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II — a general-purpose object-oriented finite element library. *ACM Transactions on Mathematical Software*, 33(4):24–27, 2007.

[5] L. Blackford, A. Petitet, R. Pozo, K. Remington, R. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[6] X. Cai, M. Dryja, and M. Sarkis. Restricted additive Schwarz preconditioners with harmonic overlap for symmetric positive definite linear systems. *SIAM Journal on Numerical Analysis*, 41(4):1209–1231, 2003.

[7] X. Cai and M. Sarkis. Restricted additive Schwarz preconditioner for general sparse linear systems. *SIAM Journal on Scientific Computing*, 21(2):792–797, 1999.

[8] C. Chevalier and F. Pellegrini. PT-Scotch: a tool for efficient parallel graph ordering. *Parallel Computing*, 34(6):318–331, 2008.

[9] A. Chronopoulos and C. Gear. *s*-step iterative methods for symmetric linear systems. *Journal of Computational and Applied Mathematics*, 25(2):153–168, 1989.

[10] E. De Sturler and H. Van der Vorst. Reducing the effect of global communication in GMRES(m) and CG on parallel distributed memory computers. *Applied Numerical Mathematics*, 18(4):441–459, 1995.

[11] J. Frank and C. Vuik. On the construction of deflation-based preconditioners. *SIAM Journal on Scientific Computing*, 23(2):442–462, 2002.

[12] C. Geuzaine and J.-F. Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

[13] P. Ghysels, T. Ashby, K. Meerbergen, and W. Vanroose. Hiding global communication latency in the GMRES algorithm on massively parallel machines. *SIAM Journal on Scientific Computing*, 1(35):48–71, 2013.

[14] L. Grigori, R. Stompor, and M. Szydlarski. A parallel two-level preconditioner for cosmic microwave background map-making. In *Proceedings of the 2012 ACM/IEEE conference on Supercomputing*, SC12. IEEE Computer Society, 2012.

[15] A. Gupta. WSMP: Watson sparse matrix package — part II: Direct solution of general systems. Technical Report 21888, IBM T.J. Watson Research Center, 2000.

[16] A. Gupta and H. Avron. WSMP: Watson sparse matrix package — part I: Direct solution of symmetric systems. Technical Report 21886, IBM T.J. Watson Research Center, 2000.

[17] P. Hénon, P. Ramet, and J. Roman. PaStiX: a high performance parallel direct solver for sparse symmetric positive definite systems. *Parallel Computing*, 28(2):301–321, 2002.

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

[19] J. Kimn and M. Sarkis. Restricted overlapping balancing domain decomposition methods and restricted coarse problems for the Helmholtz problem. *Computer Methods in Applied Mechanics and Engineering*, 196(8):1507–1514, 2007.

[20] A. Klawonn and O. Rheinbach. Highly scalable parallel domain decomposition methods with an application to biomechanics. *ZAMM - Journal of Applied Mathematics and Mechanics / Zeitschrift für Angewandte Mathematik und Mechanik*, 90(1):5–32, 2010.

[21] R. Lehoucq, D. Sorensen, and C. Yang. *ARPACK users' guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*, volume 6. Society for Industrial and Applied Mathematics, 1998.

[22] A. Logg, K.-A. Mardal, and G. Wells. *Automated solution of differential equations by the finite element method*, volume 84. Springer, 2012.

[23] R. Nicolaides. Deflation of conjugate gradients with applications to boundary value problems. *SIAM Journal on Numerical Analysis*, 24(2):355–365, 1987.

[24] C. Prud'homme, V. Chabannes, V. Doyeux, M. Ismail, A. Samake, and G. Pena. Feel++: A computational framework for Galerkin methods and advanced numerical methods. In *ESAIM: Proceedings*, volume 38, pages 429–455, 2012.

[25] A. Quarteroni and A. Valli. *Domain decomposition methods for partial differential equations*, volume 10. Clarendon Press, 1999.

[26] O. Schenk and K. Gärtner. Solving unsymmetric sparse systems of linear equations with PARDISO. *Future Generation Computer Systems*, 20(3):475–487, 2004.

[27] O. Schenk and K. Gärtner. On fast factorization pivoting methods for sparse symmetric indefinite systems. *Electronic Transactions on Numerical Analysis*, 23:158–179, 2006.

[28] B. Smith, P. Bjørstad, and W. Gropp. *Domain decomposition: parallel multilevel methods for elliptic partial differential equations*. Cambridge University Press, 2004.

[29] N. Spillane, V. Dolean, P. Hauret, F. Nataf, C. Pechstein, and R. Scheichl. A robust two-level domain decomposition preconditioner for systems of PDEs. *Comptes Rendus Mathematique*, 349(23):1255–1259, 2011.

[30] H. Sundar, G. Biros, C. Burstedde, J. Rudi,

O. Ghattas, and G. Stadler. Parallel geometric-algebraic multigrid on unstructured forests of octrees. In *Proceedings of the 2012 ACM/IEEE conference on Supercomputing*, SC12. IEEE Computer Society, 2012.

[31] J. Tang, R. Nabben, C. Vuik, and Y. Erlangga. Comparison of two-level preconditioners derived from deflation, domain decomposition and multigrid methods. *Journal of Scientific Computing*, 39(3):340–370, 2009.

[32] A. Toselli and O. Widlund. *Domain decomposition methods — algorithms and theory*, volume 34 of *Series in Computational Mathematics*. Springer, 2005.