Overlapping domain decomposition methods with `FreeFem++`

Pierre Jolivet[1,3], Frédéric Hecht[1], Frédéric Nataf[1], and Christophe Prud'homme[2]

1 Introduction

Developping an efficient and versatile framework for finite elements domain decomposition methods can be a hard task because of the mathematical genericity of finite element spaces, the complexity of handling arbitrary meshes and so on. The purpose of this note is to present one way to implement such a framework in the context of overlapping decompositions. In section 2, the basics for one-level overlapping methods is introduced, in section 3, a second level is added to the original framework to ensure scalability using a portable `C++` library, and section 4 gathers some numerical results. `FreeFem++` will be used for the computations of finite element matrices, right hand side and mesh generation, but the work here is also applicable to other Domain-Specific (Embedded) Language such as `deal.II` [3], `Feel++` [12], `GetFem++`...

2 One-level methods

Let $\Omega \subset \mathbb{R}^d$ ($d = 2$ or 3) be a domain whose associated mesh can be partitioned into N non-overlapping meshes $\{\mathcal{T}_i\}_{1\leqslant i\leqslant N}$ using graph partitioners such as METIS [10] or SCOTCH [5]. 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, and $\{V_i\}_{1\leqslant i\leqslant N}$ be the local finite element spaces defined on the domains associated to each $\{\mathcal{T}_i\}_{1\leqslant i\leqslant N}$. 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^*. Let an integer δ be the level of overlap: $\left\{\mathcal{T}_i^\delta\right\}_{1\leqslant i\leqslant N}$ is an overlapping decomposition and if we 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 $\left\{\mathcal{T}_i^\delta\right\}_{1\leqslant i\leqslant N}$, and a local partition of unity $\{D_i\}_{1\leqslant i\leqslant N}$ such that

[1] Laboratoire Jacques-Louis Lions, CNRS UMR 7598, Université Pierre et Marie Curie, 75005 Paris, France, e-mail: `{jolivet}{hecht}{nataf}@ann.jussieu.fr` · [2] Institut de Recherche Mathématique Avancée, CNRS UMR 7501, Université de Strasbourg, 7 rue René Descartes, 67084 Strasbourg Cedex, France, e-mail: `prudhomme@unistra.fr` · [3] Laboratoire Jean Kuntzmann, CNRS UMR 5224, Université Joseph Fourier, 51 rue des Mathématiques, BP53, 38041 Grenoble Cedex 9, France, e-mail: `jolivet@imag.fr`

$$\sum_{j=1}^{N} R_j^T D_j R_j = I. \tag{2}$$

Then a common one-level preconditioner for system (1) introducted in [4] is

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

The global matrix A is never assembled, instead, we build locally $A_i^{\delta+1}$ the stiffness matrix yielded by the discretization of a on $V_i^{\delta+1}$, and we remove the columns and rows associated to degrees of freedom lying on elements of $\mathscr{T}_i^{\delta+1} \setminus \mathscr{T}_i^{\delta}$, this yields $A_i = R_i A R_i^T$. The distributed sparse matrix-vector product Ax for $x \in \mathbb{R}^n$ can be computed using point-to-point communications and the partition of unity without having to store the global *distributed* matrix A. Indeed, using (2), if one looks at the local components of Ax, that is $R_i Ax$, then one can write, introducing $\mathscr{O}_i$ the set of neighboring subdomains to i, i.e. $\left\{ j \,:\, \mathscr{T}_i^{\delta} \cap \mathscr{T}_j^{\delta} \neq \emptyset \right\}$:

$$\begin{aligned} R_i Ax &= \sum_{j=1}^{N} R_i A R_j^T D_j R_j x && (4)\\ &= A_i D_i R_i x + \sum_{j \in \mathscr{O}_i} R_i R_j^T A_j D_j R_j x. && (5) \end{aligned}$$

since it can be checked that

$$\forall x \in \mathbb{R}^n,\ R_i A R_j^T D_j R_j x = R_i R_j^T R_j A R_j^T D_j R_j x \tag{6}$$

The sparse matrix-sparse matrix products $R_i R_j^T$ are nothing else than point-to-point communications from neighbors j to i.
In `FreeFem++`, stiffness matrices such as $A_i^{\delta+1}$ and right-hand sides are assembled as follows (a simple 2D Laplacian is considered here):

```
mesh Th;                    // Th is a local 2D mesh (for example T_i^{δ+1})
fespace Vh(Th, Pk);         // Vh is a local finite element space
varf a(u, v) = int2d(dx(u) * dx(v) + dy(u) * dy(v))
             + int2d(f * v) + BC;
matrix A = a(Vh, Vh);       // A is a sparse matrix stored in the CSR format
Vh rhs;                     // rhs is a function lying in the FE space Vh
rhs[] = a(0, Vh);           // Its values are set to solve Ax = rhs
```

The mesh `Th` can either be created on the fly by `FreeFem++`, or it can be loaded from a file generated offline by Gmsh [6], for example when dealing with complex geometries. By default, `FreeFem++` handles continuous piecewise linear, quadratic, cubic, quartic finite elements, and other traditionnal FE like Raviart-Thomas 1, Morley, ... The boundary conditions depend on the label set on the mesh. For example, if one wants to impose penalized homogeneous Dirichlet boundary conditions on the label `1` of the boundary of `Th`, then one just has to add

`+ on(1, u = 0)` in the definition of the `varf`. For a more detailed introduction to FreeFem++ with abundant examples, interested readers should visit `http://www.freefem.org/ff++` or see [9].
The partition of unity D_i is built using a continuous piecewise linear approximation of

$$\chi_i = \frac{\widetilde{\chi}_i}{\widetilde{\chi}_i + \sum_{j \in \mathscr{O}_i} \widetilde{\chi}_j\big|_{\mathscr{T}_i^\delta \cap \mathscr{T}_j^\delta}}, \tag{7}$$

where $\widetilde{\chi}_i$ is defined as

$$\widetilde{\chi}_i = \begin{cases} 1 & \text{on all vertices of } \mathscr{T}_i \\ 1 - \dfrac{m}{\delta} & \text{on all vertices of } \mathscr{T}_i^m \setminus \mathscr{T}_i^{m-1} \ \forall m \in [1;\delta]\,. \end{cases} \tag{8}$$

3 Two-level methods

It is well known that one-level domain decomposition methods as depicted in section 2 do suffer from poor conditioning when used with many subdomains, [16]. In this section, we present a new C++ library, independent of the finite element backend used, that assembles efficiently a coarse operator that will be used in section 4 to ensure scalability of our framework. The theoretical foundations for the construction of the coarse operator are presented in [14]. From a practical point of view, after building each local solver A_i, three dependent operators are needed :

1. the deflation matrix Z,
2. the coarse operator $E = Z^T A Z$,
3. the actual precontioner $\mathscr{P}^{-1}_{\text{A-DEF1}} = \mathscr{P}^{-1}_{\text{RAS}}(I - AZE^{-1}Z^T) + ZE^{-1}Z^T$, thorougly studied in [15].

In [14], the deflation matrix is defined as :

$$Z = \begin{bmatrix} R_1^T W_1 & R_2^T W_2 & \cdots & R_N^T W_N \end{bmatrix} \in \mathbb{R}^n \times \mathbb{R}^{\sum_{i=1}^N \nu_i} \tag{9}$$

where

$$\left\{ W_i = \begin{bmatrix} D_i \Lambda_{i_1} & D_i \Lambda_{i_2} & \cdots & D_i \Lambda_{i_{\nu_i}} \end{bmatrix} \in \mathbb{R}^{n_i} \times \mathbb{R}^{\nu_i} \right\}_{1 \leqslant i \leqslant N} \tag{10}$$

ν_i is a threshold criterion used to select the eigenvectors Λ_i associated to the smallest eigenvalues in magnitude of the following *local* generalized eigenvalue problem:

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

where A_i^δ is the matrix yielded by the discretization of a on V_i^δ, and $R_{i,0}$ is the restriction operator from $\mathscr{T}_i^\delta$ to the overlap $\mathscr{T}_i^\delta \cap \left(\cup_{j \in \mathscr{O}_i} \mathscr{T}_j^\delta \right)$. In FreeFem++, sparse eigenvalue problems are solved either with SLEPc [8] or ARPACK [11]. The latter seems to yield better performance in our simulations. Given, for each MPI process,

the local matrix A_i, the local partition of unity D_i, the set of eigenvalues $\left\{\Lambda_{i_j}\right\}_{1\leqslant j\leqslant \nu_i}$ and the set of neighboring subdomains $\mathcal{O}_i$, our library assembles E without having to assemble A and to store Z, and computes its LU or LDL^T factorization using either MUMPS [1, 2], PARDISO [13] or PaStiX [7]. Moreover, all linear algebra related computations (e.g. sparse matrix-vector products) within our library are performed using Intel MKL, or can use user-supplied functions, for example those from within the finite element Domain-Specific (Embedded) Language. Assembling E is done in two steps: local computations and then renumbering.

- first, compute *local* vector-sparse matrix-vector triple products which will be used to assemble the diagonal blocks of E. For a given row in E, off-diagonal values are computed using *local* sparse matrix-vector products coupled with point-to-point communications with the neighboring subdomains: the sparsity pattern of the coarse operator is similar to the dual graph of the mesh partitioning (hence it is denser in 3D than in 2D),
- then, renumber the *local* entries computed previously in the *distributed* matrix E.

Only few processes are in charge of renumbering entries into E. Those processes will be refered to in the rest of this note as *master processes*. Any non master process has to send the rows it has previously computed to a specific master process. The master processes are then able to place the entries received at the right row and column indices. To allow an easy incremental matrix construction, E is assembled using the COO format. If need be, it is converted afterwards to the CSR format. Note here that MUMPS only supports the COO format while PARDISO and PaStiX work with the CSR format.
After renumbering, the master processes are also the one in charge of computing the factorization of the coarse operator. The number of master processes is a runtime constant, and our library is in charge of creating the corresponding MPI communicators. Even with "large" coarse operators of sizes of around $100\,000 \times 100\,000$, less than few tens of master processes usually perform the job quite well: computing all entries, renumbering and performing numerical factorization take around 15 seconds when dealing with thousands of slave processes.
A routine is then callable to solve the equation $Ex = y$ for an arbitrary $y \in \mathbb{R}^{\sum_{i=1}^{N} \nu_i}$, which in our case is used at each iteration of our Krylov method preconditioned by $\mathscr{P}^{-1}_{\text{A-DEF1}}$. Once again, the deflation matrix Z is not stored as the products $Z^T x \in \mathbb{R}^{\sum_{i=1}^{N} \nu_i}$ and $Zy \in \mathbb{R}^n$ can be computed explicitely with a *global* matrix-free method (we only use the *local* W_i plus point-to-point communications with neighboring subdomains).

4 Numerical results

Results in this section were obtained on Curie, a Tier-0 system for PRACE 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 network. We want here to assess the capability of our framework to scale:

1. strongly: for a given *global* mesh, the number of subdomains increases while *local* mesh sizes are kept constant (i.e. local problems get smaller and smaller),
2. weakly: for a given *global* mesh, the number of subdomains increases while *local* mesh sizes are refined (i.e. local problems have a constant size).

We don't time the generation of the mesh and partition of unity. Assembly and factorization of the local stiffness matrices, resolution of the generalized eigenvalue problems, construction of the coarse operator and time elapsed for the convergence of the Krylov method are the important procedures here. The Krylov method used is the GMRES, it is stopped when the relative residual error is inferior to $\varepsilon = 10^{-6}$ in 2D, and 10^{-8} in 3D. All the following results where obtained using a LDL^T factorization of the local solvers A_i^δ and the coarse operator E using MUMPS (with a MPI communicator set to respectively `MPI_COMM_SELF` or the communicator created by our library binding master processes).

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:

$$\int_\Omega \lambda \nabla \cdot u \nabla \cdot v + 2\mu\varepsilon(u)^T\varepsilon(v) + \int_\Omega f \cdot v + \int_{\partial\Omega} g \cdot v \tag{11}$$

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 $(E_2, \nu_2) = (10^8, 0.4)$.
- ε is the linearized strain tensor and f the volumetric forces (here, we just consider gravity).

Because of the overlap and the duplication of unkowns, increasing the number of subdomains means that the number of unknowns increases also slightly, even though the number of mesh elements (triangles or tetrahedra in the case of `FreeFem++`) is the same. In 2D, we use piecewise cubic basis functions on an unstructured *global* mesh made of 110 million elements, and in 3D, piecewise quadratic basis functions on an unstructured *global* mesh made of 20 million elements. This yields a symmetric system of roughly 1 billion unkowns in 2D and 80 million unknowns in 3D. The geometry is a simple $[0;1]^d \times [0;10]$ beam ($d = 1$ or 2) partitioned with METIS.

Solving the 2D problem initially on 1 024 processes takes 227 seconds, on 8192 processes, it takes 31 seconds (quasioptimal speedup). With that many subdomains, the coarse operator E is of size $121\,935 \times 121\,935$. It is assembled and factorized in 7 seconds by 12 master processes. For the 3D problem, it takes initially 373 seconds. At peak performance, near 6 144 processes, it takes 35 seconds (superoptimal speedup). This time, the coarse operator is of size $92\,160 \times 92\,160$ and is assembled and factorized by 16 master processes in 11 seconds.

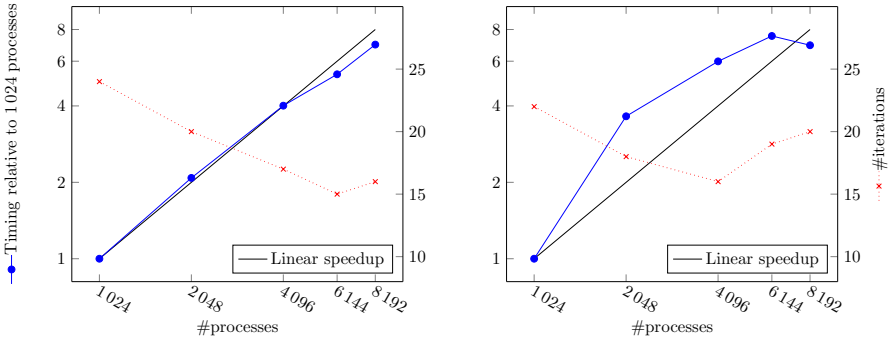

Fig. 1 Linear elasticity test cases. 2D on the left, 3D on the right. Strong scaling

Moving on to the weak scaling propreties of our framework, the problem we now solve is a scalar equation of diffusivity with highly heterogeneous coefficients (varying from 1 to 10^5) on $[0;1]^d$ ($d = 2$ or 3). Its variational formulation reads:

$$\int_\Omega \kappa \nabla u \cdot \nabla v + \int_\Omega f \cdot v \tag{12}$$

The targeted number of unkowns per subdomains is kept constant at approximately 800 thousands in 2D, and 120 thousands in 3D (once again with $\mathbb{P}_3$ and $\mathbb{P}_2$ finite elements respectively).

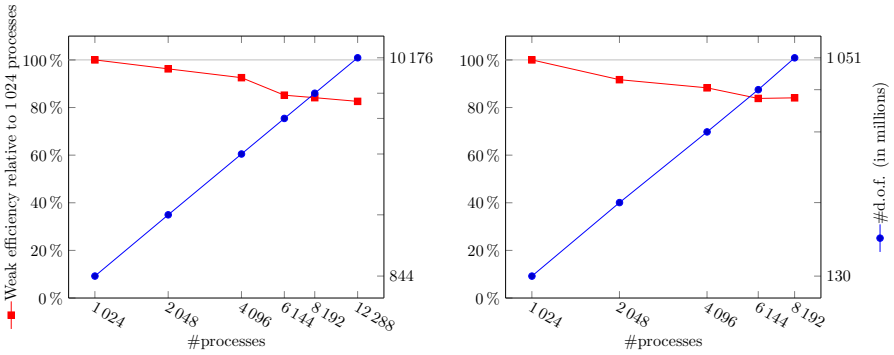

Fig. 2 Diffusion equation test cases. 2D on the left, 3D on the right. Weak scaling

In 2D, the initial extended system (with the duplication of unkowns) is made of 800 million unkowns and is solved in 141 seconds. Scaling up to 12 288 processes yields a system of 10 billion unkowns solved in 172 seconds, hence an efficiency of $\frac{141}{172} \approx 82\%$. In 3D, the initial system is made of 130 million unkowns and is solved in 127 seconds. Scaling up to 8192 processes yields a system of 1 billion unkowns solved in 152 seconds, hence an efficiency of $\frac{127}{152} \approx 83\%$.

5 Conclusion

This note clearly shows that our framework scales on very large architectures for solving linear positive definite systems using overlapping decompositions with many subdomains. It is currently being extended to support nonlinear problems (namely in the field of nonlinear elasticity) and we should be able to provide similar functionalities for non-overlapping decompositions. It should be noted that the heavy use of threaded (sparse) BLAS and LAPACK routines (via Intel MKL, PARDISO, and the Reverse Communication Interface of ARPACK) has already helped us to get a quick glance at how the framework performs using hybrid parallelism. We are confident that using this novel paradigm, we can still improve our scaling results in the near future by switching the value of `OMP_NUM_THREADS` to a value greater than 1.

Acknowledgements This work has been supported in part by ANR through COSINUS program (project PETALh no. ANR-10-COSI-0013 and projet 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.

References

1. Amestoy, P., Duff, I., L'Excellent, J.Y., Koster, J.: A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM Journal on Matrix Analysis and Applications **23**(1), 15–41 (2001)
2. Amestoy, P., Guermouche, A., L'Excellent, J.Y., Pralet, S.: Hybrid scheduling for the parallel solution of linear systems. Parallel computing **32**(2), 136–156 (2006)
3. Bangerth, W., Hartmann, R., Kanschat, G.: deal.II — a general-purpose object-oriented finite element library. ACM Transactions on Mathematical Software **33**(4), 24–27 (2007)
4. Cai, X.C., Sarkis, M.: Restricted additive Schwarz preconditioner for general sparse linear systems. SIAM Journal on Scientific Computing **21**(2), 792–797 (1999)
5. Chevalier, C., Pellegrini, F.: PT-Scotch: a tool for efficient parallel graph ordering. Parallel Computing **34**(6), 318–331 (2008)
6. Geuzaine, C., Remacle, J.F.: 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)
7. Hénon, P., Ramet, P., Roman, J.: PaStiX: a high performance parallel direct solver for sparse symmetric positive definite systems. Parallel Computing **28**(2), 301–321 (2002)
8. Hernandez, V., Roman, J., Vidal, V.: SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. ACM Transactions on Mathematical Software **31**(3), 351–362 (2005)
9. Jolivet, P., Dolean, V., Hecht, F., Nataf, F., Prud'homme, C., Spillane, N.: High performance domain decomposition methods on massively parallel architectures with FreeFem++. Journal of Numerical Mathematics **20**(4), 287–302 (2012)
10. Karypis, G., Kumar, V.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing **20**(1), 359–392 (1998)
11. Lehoucq, R., Sorensen, D., Yang, C.: ARPACK users' guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods, vol. 6. Society for Industrial and Applied Mathematics (1998)

12. Prud'homme, C., Chabannes, V., Doyeux, V., Ismail, M., Samake, A., Pena, G.: Feel++: A computational framework for Galerkin methods and advanced numerical methods. In: ESAIM: Proceedings, vol. 38, pp. 429–455 (2012)
13. Schenk, O., Gärtner, K.: Solving unsymmetric sparse systems of linear equations with PARDISO. Future Generation Computer Systems **20**(3), 475–487 (2004)
14. Spillane, N., Dolean, V., Hauret, P., Nataf, F., Pechstein, C., Scheichl, R.: A robust two-level domain decomposition preconditioner for systems of PDEs. Comptes Rendus Mathematique **349**(23), 1255–1259 (2011)
15. Tang, J., Nabben, R., Vuik, C., Erlangga, Y.: Comparison of two-level preconditioners derived from deflation, domain decomposition and multigrid methods. Journal of Scientific Computing **39**(3), 340–370 (2009)
16. Toselli, A., Widlund, O.: Domain decomposition methods — algorithms and theory, *Series in Computational Mathematics*, vol. 34. Springer (2005)