New implementations for the Simultaneous-FETI method

Roberto Molina, François-xavier Roux

► To cite this version:

Roberto Molina, François-xavier Roux. New implementations for the Simultaneous-FETI method. International Journal for Numerical Methods in Engineering, 2019, 118 (9), pp.519-535. 10.1002/nme.6024 . hal-02294320

HAL Id: hal-02294320

<https://hal.science/hal-02294320>

Submitted on 23 Sep 2019

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.

New implementations for the Simultaneous-FETI method.

Roberto Molina^{1*} | François-Xavier Roux^{2*}

¹Laboratoire Jacques Louis Lions, Sorbonne Université, Paris, 75005, France

²DTIS/ONERA, Université Paris Saclay, F-91123 Palaiseau, France

Correspondence

Roberto Molina, Applied Mathematics Department, National Laboratory for Scientific Computing - LNCC, Petrópolis, RJ, 25651-070, Brazil
Email: rmolina@lncc.br

Present address

* National Laboratory for Scientific Computing - LNCC, Petrópolis, RJ, 25651-070, Brazil

[†]DTIS/ONERA, Université Paris Saclay, F-91123 Palaiseau, France

Funding information

The author is supported by CONICYT/Chile

In this work, we present alternative implementations for the Simultaneous-FETI (S-FETI) method. Developed in recent years, this method has shown to be very robust for highly heterogeneous problems. However, the memory cost in S-FETI is greatly increased and can be a limitation to its use. Our main objective is to reduce this memory usage without losing significant time consumption. The algorithm is based on the exploitation of the sparsity patterns found on the block of search directions, allowing to store less vectors per iteration in comparison to a full storage scheme. In addition, different variations for the S-FETI method are also proposed, including another treatment for the possible dependencies between directions and the use of the Lumped preconditioner. Several tests will be performed in order to establish the impact of the modifications presented in this work compared to the original S-FETI algorithm.

KEYWORDS

domain decomposition, FETI, S-FETI, scientific computation, HPC, heterogeneous problems

1 | INTRODUCTION

In the context of numerical resolution of partial differential equations, there has been and still exists a permanent increase in the size and complexity of real engineering applications. This phenomenon is present in problems coming

---

Abbreviations: DDM, Domain Decomposition Methods.; FETI, Finite Element Tearing and Interconnecting.

* Equally contributing authors.

from different fields such as fluid dynamics, electromagnetism, structural analysis, etc. Even if a lot has already been done, there is still room for improvement for the efficient exploitation of massively parallel systems to solve these problems. If we look, for example in the framework of solid mechanics, the complexity of composite materials leads to simulations in different scales on the same numerical model, which is a numerically challenging situation. Such problems can lead to ill conditioned systems with dimension ranging from several millions to hundreds of millions.

In order to use these new architectures in the context of PDE solutions, the domain decomposition methods have been a largely used option [1],[2],[3]. The finite element tearing and interconnecting method (FETI) [4] and the balanced domain decomposition method (BDD) [5] are among the most popular non-overlapping domain decomposition methods. These non-overlapping methods are very well suited for supercomputers or computer clusters because they are based on the solution of local problems easily parallelizable on a multi-core compute node. In particular, the FETI method combines the solution of these local problems with the iterative solution of interface matching conditions in order to find the global solution.

The FETI methods have been largely developed, leading to faster and more robust versions [6]. However, there are some numerically stiff problems where FETI does not perform with the same robustness as usual. We have, for example, the cases with extreme heterogeneities along the interface where the FETI operator is defined, i.e., the intersection of domains made in the decomposition. This leads to very ill-conditioned problems that FETI with the classic preconditioners does not assure a fast convergence.

For these stiffer type of problems the Simultaneous-FETI (S-FETI) method was developed, showing very good performances, see [7] and [8]. It is based on the construction of several search directions, to be used simultaneously in the iterative solver for the interface problem. This construction adds some extra time and memory consumption cost for S-FETI in comparison with the classic FETI method. The reduction in the time cost of the extra computations of S-FETI is treated in [8]. However, this method also includes the storage of a much larger number of search directions at each iteration and significant memory issues derive from this. In the case of larger 3D problems, this may be a limitation in the use of this method.

In this paper, we propose different strategies to generalize the application of the S-FETI method to a larger class of problems by reducing the memory use. We first extend the presentation of S-FETI to generate even more search directions per iteration. We include the use of the alternative Lumped preconditioner instead of the usual Dirichlet. We show an alternative way of sorting the search directions and we introduce a new implementation that reduces the memory consumption by storing these vectors in a sparse form, i.e, before the projection and full reorthogonalization processes.

This paper will be organized as follows. First, we recall the definitions of FETI and S-FETI. We propose then, a new way of sorting the search directions in each step of the iterative method. Next, we show in detail the new sparse storage implementation for the S-FETI method to finally make some numerical validation of all the ideas proposed in here.

1.1 | FETI Definition

Let us consider a linear mechanical problem defined in a global domain Ω . Let us also consider the discretization of it by some finite element method, leading to a symmetric and positive definite system of equations $Ku = f$. The domain is now partitioned into N_s non-overlapping subdomains $\Omega^{(s)}$, $s = 1, \dots, N_s$ and new local Neumann problems are set up. The FETI method introduces a Lagrange multipliers field λ that connects these subdomains. The following systems are

then formed

$$\begin{aligned} K^{(s)} u^{(s)} &= f^{(s)} + t^{(s)T} B^{(s)T} \lambda \\ \sum_{s=1}^{N_s} B^{(s)} t^{(s)} u^{(s)} &= 0 \end{aligned} \quad (1)$$

where $t^{(s)} : \Omega^{(s)} \rightarrow \partial\Omega^{(s)}$ are trace operators (restrictions of a field into the local boundary) and $B^{(s)} : \partial\Omega^{(s)} \rightarrow \Gamma$ is a signed boolean assembly operator on the global interface Γ , with $\Gamma = \bigcup_{1 \leq s < q \leq N_s} \Gamma^{(sq)}$ with $\Gamma^{(sq)} = \partial\Omega^{(s)} \cap \partial\Omega^{(q)}$. The first line represents the local equilibrium in every subdomain $\Omega^{(s)}$, $s = 1, \dots, N_s$ and the second is the continuity of the solution through each interface $\Gamma^{(sq)}$ between each neighboring pair of subdomains, $\Omega^{(s)}$ and $\Omega^{(q)}$.

The next part consists in reformulating the local equilibrium in terms of the interface unknowns, by using the Schur complements in each subdomain. The classical notation are used

$$S_{bb}^{(s)} = K_{bb}^{(s)} - K_{bi}^{(s)} K_{ii}^{(s)-1} K_{ib}^{(s)}, \quad F^{(s)} = t^{(s)} K^{(s)+} t^{(s)T}, \quad R^{(s)} = \ker(K^{(s)}) \quad (2)$$

the subscripts i and b stand for internal and boundary degrees of freedom respectively. $F^{(s)} = (S_{bb}^{(s)})^+$ is the dual of the Schur complement and $R^{(s)}$ is a basis of rigid body modes of $\ker(K^{(s)})$. With these considerations, plus the continuity condition, the classic FETI system is formed

$$\begin{pmatrix} F & G \\ G^T & 0 \end{pmatrix} \begin{pmatrix} \lambda \\ \alpha \end{pmatrix} = \begin{pmatrix} d \\ e \end{pmatrix} \quad (3)$$

where

$$\begin{aligned} e &= -(\dots, f^{(s)T} R^{(s)}, \dots)^T, & G &= (\dots, B^{(s)} t^{(s)} R^{(s)}, \dots)^T \\ F &= \sum_s B^{(s)} F^{(s)} B^{(s)T}, & d &= -\sum_s B^{(s)} t^{(s)} K^{(s)+} f^{(s)T} \end{aligned} \quad (4)$$

The operator F is not explicitly assembled, in practice to multiply it by a vector, a local Neumann problem is solved in each subdomain and the jump of the local solutions across each through each interface $\Gamma^{(sq)}$ is computed. The use of the pseudo-inverse in the definition of $F^{(s)}$ is associated to the constraint $G^T \lambda = e$ which enforces the global equilibrium on the interface for the floating subdomains. This condition is handled by an initialization-projection strategy with the definition of

$$\lambda_0 = AG(G^T AG)^{-1} e \quad \text{Initialization} \quad (5)$$

$$P = I - AG(G^T AG)^{-1} G^T \quad \text{Projection} \quad (6)$$

so that $G^T \lambda_0 = e$ and $G^T P = 0$. The solution is then obtained as $\lambda = \lambda_0 + P \tilde{\lambda}$ where $\tilde{\lambda}$ is found via the solution of

$$P^T F P \tilde{\lambda} = P^T (d - F \lambda_0) \quad (7)$$

This equation is solved via a projected Conjugate Gradient method, where the following classic preconditioner is used

$$\tilde{S} = \sum_s \tilde{B}^{(s)} \tilde{S}^{(s)} \tilde{B}^{(s)T} \quad (8)$$

Several choices of $\tilde{S}^{(s)}$ can be made to define the preconditioner. $S_{bb}^{(s)}$, $K_{bb}^{(s)}$ and $diag(K_{bb}^{(s)})$ define the *Dirichlet*, *Lumped* and *Superlumped* preconditioners respectively.

The matrix A introduced in 5 and 6 can be taken as the identity or any of the preconditioner previously defined. In practice it is not necessary to take the Dirichlet or the Lumped preconditioner for A , the Super Lumped works well and is much easier to implement. It will be used in all the numerical examples of the paper. Finally, we also note that unlike $B^{(s)}$, the operator $\tilde{B}^{(s)}$ is a signed scaled assembly operator, meaning that it includes weighted values to better represent the physics of the problem, allowing the original FETI method to treat with several heterogeneities [9].

Remark: The Dirichlet preconditioner greatly reduces the convergence ratio and is proven to be mathematically optimal. The Lumped preconditioner, at every iteration, is less expensive in terms of memory requirement and computing time than the Dirichlet preconditioner, however usually needs more iterations for convergence. The use of the Lumped preconditioner in S-FETI will be tested in the numerical part of this paper.

1.2 | The algorithm

Introduced first in [7] for a two-subdomain case, then generalized in [8] to any domain partition, the Simultaneous-FETI method exploits the additive structure of the preconditioner in FETI, in order to generate several search directions at each iteration of the Conjugate Gradient algorithm in FETI. Following the definition 8 we know the preconditioned residual in FETI has the additive form $z = \tilde{S}g = \sum_s \tilde{B}^{(s)} \tilde{S}^{(s)} \tilde{B}^{(s)T} g$. In S-FETI the local components $\tilde{B}^{(s)} \tilde{S}^{(s)} \tilde{B}^{(s)T} g$ are considered as different search direction, meaning that the residual is minimized with respect to the spanned subspace $Span(\dots, \tilde{B}^{(s)} \tilde{S}^{(s)} \tilde{B}^{(s)T} g, \dots)$.

This definition forms a number of directions equal to the number of subdomains in the partition, however if we consider the local subdivision of the interface in each subdomain, we can in a straightforward way, extend this definition to generate an even bigger number of directions.

Formally, we have that the local interface for a subdomain $\Omega^{(s)}$ is defined as

$$\Gamma^{(s)} = \bigcup_{1 \leq q \leq N_s} \Gamma^{(sq)} \quad (9)$$

with $\Gamma^{(sq)} = \partial\Omega^{(s)} \cap \partial\Omega^{(q)}$. So we can also naturally define a subdivision of this local interface in so-called *interface edges*

$$\Gamma^{(sq)} = \partial\Omega^{(s)} \cap \partial\Omega^{(q)} \quad (10)$$

for every neighboring subdomain $\Omega^{(q)}$ of subdomain $\Omega^{(s)}$.

The operators that define the search directions in S-FETI consider using the information contained in the local interface as a single vector. This fact represents a reduction of the total information, since directions can be consider as the sum of the restrictions onto the interface edges. In this paper we will use the definition of S-FETI considering the contribution of each interface edge independently, so that a bigger number of directions will be generated at

each iteration. The underlying idea is to keep the good performance even when the edges of a subdomain behave in very different ways one of the others. This can happen, for example, in changing material for different neighboring subdomains.

The computation of the new search directions goes as follows, we first multiply the gradient g by the assembling scaled matrix $\tilde{B}^{(s)T}$, then we compute the forces needed to have the corresponding displacements on the boundary of subdomain $\Omega^{(s)}$, and scale them back on the interface Γ :

$$\delta f^{(s)} = \tilde{B}^{(s)} \tilde{S}^{(s)} \tilde{B}^{(s)T} g \quad (11)$$

The forces $\delta f^{(s)}$ are non zero only on the interface $\Gamma^{(s)}$ of the subdomain $\Omega^{(s)}$. We know that the directions and magnitudes of these forces may change a lot from one edge of the interface to the other, depending on the neighboring subdomain, so we create new search directions that reflect these interactions separately by considering the restriction of these forces on each interface edge $\Gamma^{(sq)}$ and extending them by zero on the other edges of Γ . Note $\delta f^{(sq)}$ the vector defined as:

$$\delta f^{(sq)} = (0, \dots, 0, \delta f^{(s)}|_{\Gamma^{(sq)}}^T, 0, \dots, 0)^T \quad (12)$$

Note that on the same interface $\Gamma^{(sq)} = \Gamma^{(qs)}$ the only two such vectors which are non zero are $\delta f^{(sq)}$ and $\delta f^{(qs)}$. The first one is a contribution of subdomain $\Omega^{(s)}$, the second one is a contribution of subdomain $\Omega^{(q)}$.

Finally the search space in CG will now be generated by the block of columns:

$$Z = \left[\dots, \delta f^{(sq)}, \dots \right], \quad s = 1, \dots, N_s, \quad q = 1, \dots, n^{(s)} \quad (13)$$

In this case there is not only one column coming from each subdomain. Instead there is one for each interface edge of each subdomain. This will allow to span an even bigger search space at the cost of computing all the extra directions.

As done in the regular S-FETI to compute the final search directions, each of the columns of Z needs to be projected and then reorthogonalized with the previous directions. Then a rank revealing strategy is used to compute the inverse of $W_p^T F W_p$ and eliminate the redundant directions. This strategy is not unique, but we show the one presented in [8], more details about this part will be given in the following section.

The following algorithm 1 is the same as the original, with the difference in the construction of the block Z .

Augmenting the number of search directions in that way is not so expensive a priori. For both the splitting per subdomain or per interface, there is only one Dirichlet solve or sparse matrix product to perform in each subdomain for the preconditioner. The number of vectors of Z which are non zero in subdomain $\Omega^{(s)}$ is equal to $n^{(s)} + 1$ in the first case (single contribution of $\Omega^{(s)}$ and one contribution per neighbor), and only $2n^{(s)}$ in the second case (one contribution of $\Omega^{(s)}$ and one contribution per neighbor on each edge of the interface). So the ratio is less than 2. But the total number of columns of Z is nevertheless much larger, specially in 3D problems where the total number of interface edges is several times the number of subdomains. For these type of problems storing all the directions can create the memory issues that we want to treat in this paper.

In the next section, we will give details of the already used rank revealing strategy to detect which vectors are actually independent, and we will present a more robust alternative that tries to also reduce the number of directions stored, without losing information.

Algorithm 1 S-FETI algorithm with local interface subdivision

---

```

1: Initialization
2:  $\lambda_0 = AG[G^T AG]^{-1}(-R^T e)$ 
3:  $g_0 = P^T(F\lambda_0 - d)$ 
4:  $\delta f_0^{(s)} = \tilde{B}^{(s)}\tilde{S}^{(s)}\tilde{B}^{(s)T}g_0, \quad s = 1, N_s$ 
5:  $Z_0 = [\dots, \delta f_0^{(sq)}, \dots], \quad s = 1, N_s \quad q = 1, \dots, n^{(s)}$ 
6:  $W_0 = PZ_0$ 
7: loop Iterate  $p = 0, 1, 2, \dots$  until convergence
8:    $NLL^T N^T = W_p^T F W_p$  ▷ Rank revealing factorization
9:    $W_p = W_p N L^{-T}$ 
10:   $\rho_p = -W_p^T g_p$ 
11:   $\lambda_{p+1} = \lambda_p + W_p \rho_p$ 
12:   $g_{p+1} = g_p + P^T F W_p \rho_p$ 
13:   $\delta f_{p+1}^{(s)} = \tilde{B}^{(s)}\tilde{S}^{(s)}\tilde{B}^{(s)T}g_{p+1}, \quad s = 1, N_s$  ▷ Apply local preconditioner
14:   $Z_{p+1} = [\dots, \delta f_{p+1}^{(sq)}, \dots], \quad s = 1, N_s \quad q = 1, \dots, n^{(s)}$ 
15:   $W_{p+1} = PZ_{p+1}$  ▷ Project coarse space
16:  for  $i = 0$  to  $p$  do ▷  $F$ -orthogonalization against previous sets of directions
17:     $\Phi_i = -W_i^T F W_{p+1}$ 
18:     $W_{p+1} = W_{p+1} + W_i \Phi_i$ 
19:     $F W_{p+1} = F W_{p+1} + F W_i \Phi_i$ 
20:  end for
21: end loop

```

---

2 | SORTING SEARCH DIRECTIONS IN S-FETI

In the algorithm of previous section, we point out the need of factorizing the matrix $W_p^T F W_p$ in order to build an F -orthonormal block of directions at each CG step. We can say, a priori, that is a symmetric matrix positive semi definite because of the linear dependency that may occur between the directions built, allowing only a partial factorization. However, the partial factorization with symmetric pivoting can also be used to sort the vectors between the most significant ones and the rest, reducing the number of directions stored with only a tiny loss in the information.

The fact that a linear dependency between directions exists may be explained for two main reasons, the first is that we are working in limited precision arithmetics, so at a certain point when two directions are close of each other the difference between them may be smaller than the precision of the computation of the various operators, so in practice we are not able to differentiate them. The second phenomenon which can cause a linear dependency is that some local interfaces have reached convergence, and so the directions built from these local interfaces do not add any new information.

A first attempt to overcome this issue was the use of more precise arithmetics, changing from the usual *float* or *double precision* implementation to a *quadruple precision*. This means that, we should be able to better distinguish between directions at the cost of an expensive time of computation. Unfortunately this method has not given convincing results, because the main cause of arithmetics error does not lie in the factorization itself but in the computation of the $W_p^T F W_p$ matrix itself, since a product per F requires the solution via forward-backward substitution of local Neumann problems which can be extremely ill-conditioned. This will be illustrated by a test case in the numerical results section.

Our goal is to accurately build directions that are linearly independent, in a robust way and at the smaller possible cost. For this reason the first method proposed in the original S-FETI algorithm was a rank-revealing Cholesky factorization (symmetric pivoting). In this paper, we propose the alternative of using an Eigenvalue decomposition of the matrix $W_p F W_p$, that can sort the directions by building a new block, hopefully smaller than the one from Cholesky, but with the same amount of information and less dependant on the parameters used in this algorithm.

2.1 | Cholesky factorization with complete pivoting

Presented as a cheap alternative to sort directions, the first approach is the Cholesky decomposition with complete pivoting (symmetric pivoting) described in [10]. The algorithm for this factorization is proved to work as a rank revealing procedure. This is achieved by computing, in the p iteration, an square permutation matrix N_p . In this case the decomposition can be written as

$$N_p^T W_p^T F W_p N_p = L_p L_p^T \quad (14)$$

and the construction of the F -orthonormal blocks is done via the computation of

$$W_p \leftarrow (W_p N) \begin{bmatrix} \tilde{L}_p^{-1} & 0 \end{bmatrix}^T \quad (15)$$

with $\tilde{L}_p \in \mathbb{R}^{r \times r}$ and r being the rank or number of independent directions.

2.2 | Diagonalization of search directions block

Here we propose a second way to treat the sorting of independent search directions. It involves the use of a diagonalization process. We start by computing the eigenvalues and eigenvectors of $W_p^T F W_p$, we know that this procedure leads to the decomposition

$$W_p^T F W_p = E_p D_p E_p^{-1} \quad (16)$$

Where E_p is the matrix of the eigenvectors in the columns, and D_p is a diagonal matrix with the eigenvalues in the diagonal. In our case, the matrix is real and symmetric so we have that $E_p^{-1} = E_p^T$.

The common algorithms for the computation of eigenvalues can be used to obtain them in descending (or ascending) order. This fact will allow us to build an usually smaller F -orthonormal block of search directions. We should consider only the eigenvalues strictly greater than zero, so the rank of the matrix is also computed. In practice, in order to reduce the numerical noise due to round-off errors, we keep only the eigenvalues which are greater than a positive threshold ε .

Consider $\tilde{E}_p$ as the block of r eigenvectors associated to the eigenvalues which are greater than ε and if we write these eigenvalues in the diagonal matrix $\tilde{D}_p$, then the construction of the new block can be form by updating W_p

$$W_p \leftarrow W_p \tilde{E}_p \tilde{D}_p^{-1/2} \quad (17)$$

In the use of the Cholesky decomposition, a number of directions are eliminated, introducing a possible loss of the information contained in the starting block of directions. It depends on a parameter $\varepsilon > 0$ and can have a great impact

Subdomains	Iterations	Max local Memory (Gb)	Directions stored
125	7	3.872	11176

TABLE 1 Direction stored in a $5 \times 5 \times 5$ cube configuration

in the final results (see Section 4). With this new decomposition, even if we still have a parameter ε , we have a bigger control in the information loss, because few elimination of directions is required. This is due to the properties of the eigenvalue decomposition, that allows an ordering and hence a construction of a smaller block with some new vectors that are the most relevant.

The impact in time of the diagonalization process is expected to be small, because the size of the matrix $W_p^T F W_p$ depends on the number of subdomains, so it is moderate and does not involve any communication between processes. Also the use of LAPACK algorithms with BLAS optimizations, allows a fast computation simultaneously in every process. A parallelization of this diagonalization process could also be considered if it happened to become non negligible.

3 | MEMORY USAGE IN S-FETI

From the definition of the S-FETI algorithm it is known that one of the differences between this method and the classical FETI lies in the total memory use in both of them. For the classical FETI, only two vectors are stored at each iteration, they correspond to the search direction and its multiplication by F . On the other hand, in S-FETI the number of search directions built in every iteration is increased, making mandatory the storage of blocks W_p and $F W_p$. This difference is in fact one main drawback in the implementation of S-FETI.

The total number of directions stored depends and grows linearly as the number of iterations, this value is in every step usually the number of columns in Z_p (in practice, due to linearity of the directions, we expect a smaller number that correspond to $\text{Rank}(W_p F W_p^T)$). In 2D problems this is not much of an issue, but in 3D even in small cases this needs to be considered, for example if we consider our domain to be a cube divided into $5 \times 5 \times 5$ smaller cubes, we see in Table 1 that the number of stored directions at each step is of 1600 directions approximately, a fact that comes only from the geometry of the problem. For the total memory needed, this will depend on the size of the triangulation of the mesh, in this case, of about 10^5 elements in each subdomain.

To be able to use this method in bigger configurations we will change the usual, full storage and use a different strategy. We know that the time of a parallel application is limited by the exchange of information between processes, so if we add some (not too expensive) local extra computations we hope to have an implementation as fast as the original one, but without the memory constraint.

3.1 | New sparse storage

The idea of this new storage is to reconstruct the search directions at every iteration, that is, to compute the projection and the full reorthogonalization of the columns in Z_p , by storing matrices that are either sparse or small ones.

This is done by saving at each iteration the sparse matrices, Z_p and $F Z_p$, instead of the full $W_p = P Z_p$ and $F W_p = P F Z_p$. Along with these sparse blocks, we also compute and store some extra coefficient matrices that will allow the reconstruction of the full directions. The size of this new coefficient matrices will depend on the number of used search directions at each iteration, and will be independent of the size of the mesh, so in general, it will be a small number compared to the size of the problem.

The memory needed for the storage will now be limited in a more important way on the problem configurations, that determines the maximum total search directions in every iteration, rather than the size of the discretization. Compared with the total unknowns of the problem, this number (N_T) is usually a small one, because the number of neighbours of each subdomain is also limited, so the problems that we can solve can increase in its total size and have a smaller impact in the memory used by it, allowing us to use the S-FETI method in bigger problems.

The fact that we no longer store the complete search directions will also have an impact in the precision achieved with this implementation, but we hope to keep the good accuracy of the S-FETI method or at least be close enough to make this a useful algorithm, we will later discuss this fact in the numerical results.

3.2 | Reconstruction of search directions

3.2.1 | Formal construction

At iteration p iteration, the projection of Z_p appears as a low-rank correction using G : $PZ_p = Z_p + (AG)D_p$ with $D_p = -(G^T AG)^{-1} G^T Z_p$ by definition of the projection. Since the successive W_j blocks are constructed from the projected blocks PZ_j plus an F -orthogonalization procedure against the previous W_i blocks, it can easily be induced that the W_j blocks can be reconstructed from the previous projected PZ_i blocks and finally be decomposed in the following way:

$$W_j = Z_j + \sum_{i=1}^{j-1} Z_i B_{ij} + (AG)\Delta_j \quad (18)$$

with B_{ij} and Δ_j to be computed later, recursively. We replace this equation in the construction of the orthogonalized and projected search directions

$$\begin{aligned} W_p &= PZ_p + \sum_{j=1}^{p-1} (W_j)\Phi_{jp} \\ &= Z_p + (AG)D_p + \sum_{j=1}^{p-1} \left(Z_j + \sum_{i=1}^{j-1} Z_i B_{ij} + (AG)\Delta_j \right) \Phi_{jp} \\ &= \left\{ Z_p + \sum_{j=1}^{p-1} Z_j \Phi_{jp} + \sum_{j=1}^{p-1} \sum_{i=1}^{j-1} Z_i B_{ij} \Phi_{jp} \right\} + (AG) \left\{ \sum_{j=1}^{p-1} \Delta_j \Phi_{jp} + D_p \right\} \end{aligned} \quad (19)$$

Developing the construction of the first term of W_p , which depends only on the blocks Z_i , we obtain:

$$\begin{aligned} Z_p + \sum_{j=1}^{p-1} Z_j \Phi_{jp} + \sum_{j=1}^{p-1} \sum_{i=1}^{j-1} Z_i B_{ij} \Phi_{jp} &= Z_p + \sum_{j=1}^{p-1} Z_j \Phi_{jp} + \sum_{i=1}^{p-2} Z_i \left(\sum_{j=i+1}^{p-1} B_{ij} \Phi_{jp} \right) \\ &= Z_p + \sum_{i=1}^{p-1} Z_i \Phi_{ip} + \sum_{i=1}^{p-2} Z_i \left(\sum_{j=i+1}^{p-1} B_{ij} \Phi_{jp} \right) \\ &= Z_p + \sum_{i=1}^{p-1} Z_i B_{ip} \end{aligned} \quad (20)$$

with B_{ip} being computed in the following way:

$$B_{ip} = \Phi_{ip} + \sum_{j=i+1}^{p-1} B_{ij}\Phi_{jp} \quad (21)$$

In the same way the second term which depends only of (AG) can be written:

$$(AG) \left\{ \sum_{j=1}^{p-1} \Delta_j \Phi_{jp} + D_p \right\} = (AG)\Delta_p, \quad \text{where } \Delta_p = \sum_{j=1}^{p-1} \Delta_j \Phi_{jp} + D_p \quad (22)$$

We can now write the formula which allow to reconstruct the W_p and FW_p blocks from the sparse Z_i , FZ_i , (AG) and $F(AG)$ blocks which will be the only ones to be stored:

$$W_p = Z_p + \sum_{i=1}^{p-1} Z_i B_{ip} + (AG)\Delta_p \quad ; \quad FW_p = FZ_p + \sum_{i=1}^{p-1} FZ_i B_{ip} + F(AG)\Delta_p \quad (23)$$

The matrices of coefficients B_{ip} and Δ_p are computed using the recurrence formula (21) and (22). The Φ_{jp} matrices come from the F -orthogonalization of block PZ_p against the previous W_j blocks:

$$(FW_j)^T W_p = (FW_j)^T (PZ_p + \sum_{j=1}^{p-1} W_j \Phi_{jp}) = 0, \quad \text{for } j < p \quad (24)$$

which implies

$$(FW_j)^T (W_j) \Phi_{jp} = -(FW_j)^T PZ_p, \quad \text{for } j < p \quad (25)$$

The left hand side matrix correspond to the symmetric positive semi-definite matrix which needs to be pseudo inverted at each iteration. To simplify the computations we apply the cholesky decomposition to it, giving

$$L_j L_j^T \Phi_{jp} = -(FW_j)^T PZ_p \quad (26)$$

The L_j matrices need to be stored at every iteration, but then again these are small triangular matrices of size $rank(Z_j)$. The right-hand side matrix can also be computed using the recurrence formula (23):

$$\begin{aligned} (FW_j)^T (PZ_p) &= \left[FZ_j + \sum_{i=1}^{j-1} FZ_i B_{ij} + (FAG)\Delta_j \right]^T \left[Z_p + (AG)D_p \right] \\ &= Z_j^T (FZ_p) + \sum_{i=1}^{j-1} B_{ij}^T Z_i^T (FZ_p) + \Delta_j^T (FAG)^T Z_p + \\ &\quad + Z_j^T (FAG)D_p + \sum_{i=1}^{j-1} B_{ij}^T Z_i^T (FAG)D_p + \Delta_j^T (AG)^T (FAG)D_p \end{aligned} \quad (27)$$

Algorithm 2 S-FETI with Sparse Storage

```

1: Initialization
2:  $\lambda_0 = (\mathbf{AG})[\mathbf{G}^T \mathbf{AG}]^{-1}(-R^T c)$ 
3:  $\mathbf{g}_0 = F\lambda_0 - d$ 
4: loop Iterate  $p = 0, 1, 2, \dots$  until convergence
5:    $\delta f_p^{(s)} = \tilde{B}^{(s)} \tilde{S}^{(s)} \tilde{B}^{(s)T} \mathbf{g}_p, \quad s = 1, N_s$ 
6:    $\mathbf{Z}_p = [\dots, \delta f_p^{(sq)}, \dots], \quad s = 1, N_s \quad q = 1, \dots, n^{(s)}$ 
7:    $\mathbf{Q}_p = F\mathbf{Z}_p$ 
8:    $\mathcal{D}_p = -[\mathbf{G}^T \mathbf{AG}]^{-1} \mathbf{G}^T \mathbf{Z}_p$ 
9:   for  $j = 0$  to  $p - 1$  do
10:     $\mathbf{S}_{jp} = \mathbf{Z}_j^T \mathbf{Q}_p$ 
11:   end for
12:    $\mathcal{T}_p = \mathbf{Z}_p^T (F\mathbf{AG})$ 
13:   for  $j = 0$  to  $p - 1$  do
14:     $\Phi = \mathbf{S}_{jp}^T + \sum_{i=0}^{j-1} \mathbf{S}_{ip}^T \mathcal{B}_{ij} + \mathcal{T}_p \Delta_j$ 
15:     $\Phi = \Phi^T + \mathcal{T}_j \mathcal{D}_j + \sum_{i=0}^{j-1} \mathcal{B}_{ij}^T \mathcal{T}_i \mathcal{D}_p + \mathcal{U}_j^T \mathcal{D}_p$ 
16:     $\Phi_{jp} = -(\mathcal{L}_j \mathcal{L}_j^T)^{-1} \Phi$ 
17:   end for
18:    $\Delta_p = \mathcal{D}_p + \sum_{j=0}^{p-1} \Phi_{jp}$ 
19:    $\mathcal{U}_p = (\mathbf{AG})^T (F\mathbf{AG}) \Delta_p$ 
20:   for  $j = 0$  to  $p - 1$  do
21:     $\mathcal{B}_{jp} = \Phi_{jp} + \sum_{i=j+1}^{p-1} \mathcal{B}_{ji} \Phi_{ip}$ 
22:   end for
23:    $\mathbf{W} = \mathbf{Z}_p + \sum_{j=0}^{p-1} \mathbf{Z}_j \mathcal{B}_{jp} + (\mathbf{AG}) \Delta_p$  ▷ Compute projected and reorthogonalized blocks
24:    $\mathbf{FW} = \mathbf{Q}_p + \sum_{j=0}^{p-1} \mathbf{Q}_j \mathcal{B}_{jp} + (F\mathbf{AG}) \Delta_p$ 
25:    $N_p \mathcal{L}_p \mathcal{L}_p^T N_p^T = \mathbf{W}^T (\mathbf{FW})$ 
26:    $\rho = -(\tilde{\mathcal{L}}_p \tilde{\mathcal{L}}_p^T)^{-1} (\mathbf{W} N_p)^T \mathbf{g}_p$  ▷  $\tilde{\mathcal{L}}_p \in \mathbb{R}^{r \times r}$  is full ranked
27:    $\lambda_{p+1} = \lambda_p + (\mathbf{W} N_p) \rho$ 
28:    $\mathbf{g}_{p+1} = \mathbf{g}_p + (\mathbf{FW} N_p) \rho$ 
29: end loop

```

Next we present a summary of the computations needed and the matrices stored in this new algorithm for the S-FETI method. At each iteration p and for $j < p$ we have the sparse matrices, which means matrices whose columns are interface vectors, each of them being non zero only on a limited set of interface edges:

$$(\mathbf{AG})^T (F\mathbf{AG}); \quad \mathbf{Z}_j^T F\mathbf{Z}_p; \quad \mathbf{Z}_p^T (F\mathbf{AG}) \quad (28)$$

and for the full (but small) coefficient matrices

$$D_p; \Delta_p; (AG)^T (FAG) \Delta_p; \Phi_{jp}; B_{jp} \quad (29)$$

With all these considerations, we can describe the S-FETI method with this new search directions storage in Algorithm 2. For clarity, We use a different font for the coefficient matrices and bold for the coarse ones.

Remark: Compared with the S-FETI method, we just replaced the projection and full reorthogonalization processes to build W and FW , the rest is analogous, including the use of different preconditioners as well as the rank revealing strategy.

3.3 | Implementation details and exploitable parallelism

In the previous section, we introduced several extra computations to the basic S-FETI algorithm. They can be implemented in a straightforward way, but to achieve a similar performance as in S-FETI we will give some considerations that can have an important time boost to this method.

3.3.1 | Optimization of the code

With the structures in mind, we can now give the details of several possible optimization of Algorithm 2. These changes come from the advantages of BLAS3 when doing block computations and also by adding some extra parallelization. These changes can be summarized as:

- Concatenated allocation in memory of the terms Z_j with $j = 1, \dots, p - 1$ so that the computation of $Z_j^T (FZ_p)$ can now be done as a single block for all j .

A priori, the computation of this matrix is done using the sparsity properties of Z_p and FZ_p . This implies the computation, in each subdomain $\Omega^{(s)}$, of several dot products of the form $z_j^{(i)T} F^{(s)} z_p^{(k)}$, $\forall i, k \in \text{neighbour}(s) \cup \{s\}$. Then a global reduction allows the computation of the total matrix that is now shared by all the processes.

We allocatall the $j < p$ vectors $z_j^{(i)}$ as a single block, giving

$$\begin{bmatrix} z_1^{(i)}, \dots, z_{p-1}^{(i)} \end{bmatrix}^T F^{(s)} z_p^{(k)}, \quad \forall i, k \in \text{neighbour}(s) \cup \{s\} \quad (30)$$

This new matrix-vector multiplication allows the block optimization of the BLAS3 implementations [11], and the global reductions needed are reduced from $p - 1$ to a single one of greater size.

Remark: Let us recall that in the implementation of the FETI methods, the reduction operations can have a huge impact in the ellapsed time, due to the fact that the start-up time for a global communication is huge compared to the time for an arithmetic operation.

- If we look at the term $\sum_{i=1}^{j-1} B_{ij}^T T_i D_p$ in the loop to compute Φ_{jp} we can see that the matrix D_p is independent of the addition, so instead of doing computations of the type $(FAG)D_p$, we can compute and save

$$V_p \leftarrow \begin{bmatrix} \sum_{i=1}^{p-1} B_{ip}^T T_i \end{bmatrix} \quad (31)$$

FIGURE 1 Cube divided into 125 subdomains.

This computation is performed after the matrices B_{ip} have been calculated. We store this sum as a single matrix to use it directly in the next iterations. This term is another small matrix, so the memory cost is still controlled.

- As explained before, the matrices needed for the computations of the terms Φ_{jp} for all $j < p$, are global and shared in each subdomain, including the matrices coming from the Cholesky factorization. Hence, a priori, each process compute the same terms.

This involves the computation of a number of small matrix products, which increases at each iteration. To avoid these increasing computation costs, the calculation of Φ_{jp} for each $j = 1, p - 1$, can be done in parallel by a different process.

Each process j can now be in charge of each matrix Φ_{jp} , up to a limit of N_s matrices (the total number of processes). This means that after the iteration $p > N_s$ the first process will need to computation another matrix Φ_{jp} and then for each extra iteration, another process will be in charge of the new computations. The number of iterations of the S-FETI method is expected to be low, so this will seldom happen.

Remark: A priori, this parallelization will speed up the total time, but we can not forget that a global exchange is added after, so that every process can have access to the new computed Φ_{jp} matrices. The impact of this optimization will be tested in the numerical examples.

4 | NUMERICAL RESULTS

The following tests were made using a Fortran-MPI implementation, on a machine SGI UV 2000 with 32 CPUs Intel Xeon 64 bits EpyBridge E4650 of 10 cores each one, with a frequency of up to 2.4 GHz. This will allow for memory and time measurements. We note that no local parallelization (OpenMP) was used in any of the tests performed here.

4.1 | S-FETI basics

We use as a test case the cube $\Omega = [0, 1]^3$ that its divided in subdomains of smaller cubes of equal size, see Figure 1 for the $5 \times 5 \times 5$ subdivision. The problem to solve is a finite element discretization of a highly heterogeneous Poisson problem $-\nu \Delta u = f$ in Ω , $u = 0$ in $\partial\Omega$, leading to a system of the form $Ku = f$. We include a parameter ν that represents the differences in the materials of each subdomain. We use a sparse direct solver called DISSECTION for solving the local Neumann problem in each subdomain. This solver allows a more robust computation of the kernels and recently it

S-FETI decomposition	Total Search Directions	Iterations
Local subdomain	1125	9
Local interface	6356	4

TABLE 2 Difference in subdomain division versus local interface division

has shown an improved general performance, compared to the popular Intel PARDISO solver, specially for multiple forward-backward substitutions, which are used on S-FETI, see [12].

We consider as stopping criteria the global relative residual and the relative solution jump norm across the interfaces. In some cases precision measures will be performed, if that is not the case the iterations are stopped when both are smaller than 10^{-4} .

We consider the "Super lumped" scaling for the projection, i.e $P = P_A = I - AG(G^T AG)^{-1}G^T$ with $A = \tilde{S}$ as in equation (8), and the weighted (scaled) Dirichlet preconditioner as the global preconditioner.

The first test was made to show the difference between the subdomain and local interface subdivisions. In Table 2 we note the reduction in the number of iterations at the price of augmenting several times the number of directions used and stored. This will be the default for the next part, mainly because we are able to reduce communications and iterations at the price of more memory resources. In the following, we will show how this extra memory use can also be reduced.

4.2 | Decomposition of $W^T FW$

We change the test case to an heterogeneous cube as in Figure 2 with 10^0 and 10^3 being the values for the parameter ν for the red and blue subdomains respectively, also $f = 1$ and 10^5 elements per subdomain are used. The idea is to create a problem to test the differences in the precision achieved between the different strategies, as well as the number of directions stored.

The first comparison was done between a double and a quadruple precision implementation of the rank revealing Cholesky factorization of the $W^T FW$ matrix. Only this computation is performed in extended precision. By using an extended floating-point arithmetic we reduce the rounding error produced in this part, although the cost of itself can be a mayor drawback. We tested for different values of "zero", represented by the parameter ε .

In Table 3 we see that as ε goes to zero, the number of search directions increases and the iterations are reduced, this is thanks to the full reorthogonalization. However, all the extra directions are not necessarily independent. For this reason, if we consider values of ε not so small we are in a "losing directions" approach where more iterations are needed but a higher precision is achieved. All the directions eliminated this way can be automatically recovered in future iterations.

The differences exposed between double and quadruple implementations are not enough to consider the use of the much slower quadruple arithmetic, so from now on, only the regular double will be considered. These results show clearly that the lack of accuracy does not come from the Cholesky factorization but from the forming of the $W^T FW$ matrix, and specially from the computation of the products by F .

Using the same test case we are going to state the differences between the Cholesky and the Eigendecomposition for the pseudo-inversion of the $W^T FW$ matrix. One of the objectives in changing the decomposition is to make the method less sensitive to parameter variations and to reduce the number of search directions stored. We will test for

FIGURE 2 Checkerboard cube for 125 subdomains with two different values of ν .

ε	Implementation	Search directions per iteration	Minimal error	Iterations to error
10^{-4}	QUAD	1247	$6.0537 * 10^{-5}$	8
10^{-4}	DOUBLE	1245	$9.0064 * 10^{-5}$	8
10^{-6}	QUAD	1356	$1.7085 * 10^{-3}$	6
10^{-6}	DOUBLE	1356	$1.7131 * 10^{-3}$	6
10^{-10}	QUAD	1498	$6.4209 * 10^{-3}$	5
10^{-10}	DOUBLE	1494	$1.6876 * 10^{-3}$	5

TABLE 3 Quadruple vs Double comparison

different values of the zero parameter ε and we will establish the maximal precision achieved by both methods.

In Table 4 we show the number of directions stored with each zero value until max precision for each method is achieved. We see that in this case both decompositions are equivalent in achieving a stopping criteria, but with Eigen using less directions. Here we can see that the bigger values of ε reduces the number of directions kept, but the convergence is still assured, as the important information is contained in the directions associated to the bigger eigenvalues. The Eigenvalue decomposition in this case is less sensitive to variations in the zero parameters and in all the cases shows a reduction in the number of directions kept, meaning that at a small cost (in parallel computation it is negligible) we can have a method with an improved precision and at the same time fewer memory charge.

We also insist in the fact that a much more precise method is obtained when using values of ε not so close to zero, just $\varepsilon = 10^{-4}$ is more than enough to reduce the minimal error and at the same time reduce the stored directions.

4.3 | S-FETI with sparse storage

In this section we want to test the new sparse storage implementation for the S-FETI method, regarding both time and memory usage.

The model problem is again the 3D Poisson equations, solved in the cube $\Omega = [0, 1]^3$, with the same boundary conditions and source as the previous examples. The stopping criteria for the global error in this case is also 10^{-4} .

ε	Implementation	Search directions per iteration	Minimal error	Iterations to CHOLESKY minimal error	Iterations to minimal error
10^{-4}	CHOLESKY	1245	$9.0064 * 10^{-5}$	8	8
10^{-4}	EIGEN	1183	$1.7590 * 10^{-8}$	8	11
10^{-6}	CHOLESKY	1356	$1.7131 * 10^{-3}$	6	6
10^{-6}	EIGEN	1299	$1.7082 * 10^{-5}$	6	8
10^{-10}	CHOLESKY	1494	$1.6876 * 10^{-3}$	5	5
10^{-10}	EIGEN	1474	$2.2169 * 10^{-3}$	5	5

TABLE 4 Iteration and Search Direction number for different values of ε .

Implementation	Elements per subdomain	Iterations	Time (s)	Max local memory (Gb)
SPARSE-S-FETI	103823	8	571.1	4.91
SPARSE-OPT-S-FETI	103823	8	289.4	4.63
SPARSE-S-FETI	148877	8	648.1	6.57
SPARSE-OPT-S-FETI	148877	8	339.8	6.13

TABLE 5 SPARSE vs SPARSE-OPT Performance

We also use 125 subdomains with local interface subdivisions, to have at least the same amount of search directions as previous case. The parameter ν will represent the homogeneous case, meaning that we will have one material as shown in Figure 1. The time, memory and precision measures will be more important, rather than the number of iterations. This term should not change between the two sparse implementations, and we hope to have similar results for the sparse and the full versions.

We start by testing the speed and precision achieved by the two versions of the SPARSE-S-FETI method. The first with a straightforward implementation when operating with the sparse blocks and the second one with optimizations in memory allocation. We use BLAS3 routines, and we reduce the bottleneck of doing several matrix-vector operations by doing just one matrix-matrix product. We also change the full reorthogonalization process by making only one process to orthogonalize each saved block and then sending the information to the rest of the processes, see the last point in subsection 3.3.1. The results in terms of memory and time are shown in Table 5, where we measured the total time used and also the maximum memory used by each process. We can see an speed-up of roughly two times for both examples with 100 and 150 thousand elements approximately in each subdomain.

From here on, the optimized implementation will be the one used, for three main reasons. First the total time of computation was greatly reduced, second no precision was lost (both have the exact same), and finally we slightly reduced the memory used, as seen in the memory column.

FIGURE 3 Max local memory usage for 5×10^5 (left) and 1×10^6 (right) elements per subdomain in a $4 \times 4 \times 4$ cube subdivision.

4.3.1 | SPARSE-OPT vs FULL Implementations

The objective of the sparse storage implementation is to keep a memory usage controlled in order to be able to use this method for the biggest applications. In this part we will compare the memory usage of the SPARSE-OPT implementation versus the regular FULL one.

In the Figure 3 we can see the maximum memory usage per process for both methods, but with two different number of total elements in each one. We are interested in memory performance so we forced to save all the search directions because we want an upper estimate in terms of memory versus number of iterations. In this example we were using 936 directions at each iteration. These results show that the difference between the SPARSE storage and the FULL implementation is clearer when the size of each subdomain increases. For the FULL-S-FETI method, we have a linear relation between the number of iterations (starting from 2) and the memory usage (or the number of search directions stored). For the sparse storage, this iteration/memory relation is linear for the storage of the coarse spaces Z_p and $F Z_p$ and quadratic for the storage of the coefficient matrices which explains the behavior of the curve of SPARSE-OPT-S-FETI. As already said, these coefficient matrices are dense and their dimension does not depend on the size of the mesh (the size of the full vectors), but only on the number of subdomains and interface edges.

The following test to do, once the memory advantages of the sparse storage were established, is a time consumption test between the SPARSE-OPT and regular method to see the cost of all the extra computations.

The problem to solve is the same checkerboard configuration as previous with two materials, $\nu_1 = 10^0$ and $\nu_2 = 10^4$. The results are shown in Table 6 where we see that the differences in precision and number of iterations are negligible. In the same way, the time difference is small, and it even goes down whenever the problem gets bigger. We can say then that the new implementation is suitable to test big cases where the memory is a limitation.

4.4 | General comparison

Finally we want to test the difference between the two existent preconditioners for S-FETI, the Dirichlet and Lumped in a more general case. We test for memory, number of iterations and total time for the computations.

Added to this, we want to compare the SPARSE and FULL storage implementations with these preconditioners,

Method	Elements per subdomain	Iterations	Time (s)	Precision at stopping criteria
FULL-S-FETI	103823	11	620.1	$4.3295 * 10^{-5}$
SPARSE-OPT-S-FETI	103823	11	624.8	$2.0870 * 10^{-4}$
FULL-S-FETI	148877	11	755.1	$1.9688 * 10^{-4}$
SPARSE-OPT-S-FETI	148877	11	756.9	$1.8155 * 10^{-4}$

TABLE 6 FULL vs SPARSE-OPT Time performance

as well as the two decomposition for sorting the search direction. The idea is to have global view that can gives some notion of which method should we use in different situations.

- Type of implementation: FULL-S-FETI or SPARSE-OPT-S-FETI
- Preconditioner: Dirichlet or Lumped.
- Decomposition method for $W^T F W$: Cholesky or Eigenvalue.

The performance in this case, due to memory limitations, was tested on a different machine, the Santos Dumont Supercomputer at the National Laboratory for Scientific Computing - LNCC in Brazil. Each node of this computer have 2 x CPU Intel Xeon E5-2695v2 Ivy Bridge at 2,4GHZ. We note that, again, no local parallelization (OpenMP) is used here, so the times can only improve when using several threads to solve the local problems.

In this last test, we use the same checkerboard heterogeneities described in Figure 2 with $E_{red} = 10^0$ and $E_{blue} = 10^3$, but this time we use an automatic subdivision for the subdomains, as in Figure 4. The subdivision is done by the mesh partition algorithm METIS [13]. In here, we are closer to a real engineering problem where usually a global problem is divided automatically and numerical issues due to interface hetrogeneities are more present.

Furthermore, in all the tests we performed up to now, with a regular cartesian splitting, the size of the interface was minimal. For real problems and less regular splitting, the ratio of the numebr of interface nodes on the number of inner nodes tend to be larger. So the storage of the full vectors is more expensive. Reversely, irregular splitting tend to give less neighbors per subdomain and smaller number of interface edges, which determine the size of the coefficient matrices of the SPARSE implementation. So all the tests performed up to now had a bias which benefited to the method with full storage.

One of the crucial differences that we want to test is the one produced by the different preconditioners available, namely the weighted Dirichlet and Lumped preconditioners. If we look at both definitions in Equation 8, we see that one involves the multiplication by the Schur complement, which requires the solution of a local Dirichlet problem. Then, we can realize that there are differences between them regarding computational and memory cost. In both characteristics the Dirichlet is much more expensive; however, we expect a reduced number of iterations for this preconditioner, because the local contribution of each subdomain for the preconditioner is exactly the inverse of its contribution to the dual operator for FETI.

FIGURE 4 Cube metis with 125 subdomains.

FIGURE 5 Max local memory allocation in a cube for FULL and SPARSE implementations of S-FETI.

The results are shown in the Table 7. In line to what we expected a significant reduction in memory was achieved when using SPARSE implementation, compared to its FULL version in the different S-FETI variations. We also note a remarkable speedup when using the SPARSE implementation, more test are needed here, but using local parallelization.

If we now compare the Dirichlet versus Lumped preconditioners, we note that, the extra iterations needed in Lumped make this alternative slower and more memory consuming due to these extra iterations.

As far as the decomposition, we see a small memory reduction in the FULL version, associated to the less directions

Implementation	Preconditioner	Decomposition	Iterations	Time (s)	Max local memory (Gb)
FULL	Dirichlet	Cholesky	15	3282	23.11
FULL	Dirichlet	Eigen	15	3314	22.96
FULL	Lumped	Cholesky	25	7597	24.38
FULL	Lumped	Eigen	25	7336	23.25
SPARSE	Dirichlet	Cholesky	15	1578	18.46
SPARSE	Dirichlet	Eigen	15	1582	18.46
SPARSE	Lumped	Cholesky	25	3065	19.78
SPARSE	Lumped	Eigen	25	3065	19.79

TABLE 7 Performance of S-FETI implementations with METIS subdivisions. 125 subdomains and 5×10^5 elements per subdomain.

stored. In the SPARSE version, since the memory is driven by the coefficient matrices, no memory reduction is seen.

All these results need a further analysis to be able to predict or give lines of what implementation use in some determined problem. As a first approach, we present some theoretical approximation of the memory allocation made in both implementations for a cube divided in smaller cubes of equal mesh size. The idea is to show where the SPARSE implementation is more suited than the FULL one.

In Figure 5 we note the linear and quadratic relation in memory allocation for both methods and how they compare to each other. This shows that in cases with moderate number of large subdomains, the SPARSE method is better, conversely the SPARSE is less good for large numbers of small subdomains.

In most of practical implementations of FETI for real engineering problems, the first option, namely moderate number of large subdomains, is preferred. The first reason is that sparse direct solvers perform very well, both in terms of computation time and memory requirement, for finite element problems with some hundreds thousands degrees of freedom. So there is no practical need to cut in smaller subdomains. Furthermore, splitting a complex geometry with heterogeneous materials in large number of subdomains tend to create very badly shaped and highly heterogeneous interfaces which are bad for the convergence of FETI.

4.4.1 | S-FETI with reduced directions

Finally in our tests, when looking to the Eigendecomposition, we tried a different approach to reduce the memory usage of the FULL version. In this case we saved a reduced number of search directions in the form of the block used to update the solution at each iteration, meaning that instead of storing W_p we stored a part of $W_p \rho$.

We can save this complete vector to greatly reduce the memory use (we reduced to the cost of FETI), but implying a slower convergence. Because the Eigendecomposition reorders this block we can say that the first ones are the more important, so storing just a percentage may keep the good convergence (similar to the idea of use bigger values of ε in Table 4) but greatly reducing the memory use. In Figure 6 we show the iterations needed to converge when reducing the directions stored. We see that storing even a 50% of the total search directions can keep a similar convergence ratio and

FIGURE 6 Iterations when storing less directions in S-FETI.

that maybe we do not need to store all the directions. In any case more tests are needed to see the extends of this claim.

5 | CONCLUSION

Several implementation changes have been introduced in this paper, most of them showing an improved performance compared to the primary S-FETI. We added a new way of sorting search directions. The Eigenvalue decomposition has proved to be a more robust option, and at the same time, it reduces the number of directions stored in each iteration. In terms of memory usage, we developed a new sparse storage that allows using this method in bigger applications without losing significant performance. The Lumped preconditioner for this method is an attractive alternative to reduce memory and computation efforts, even if the results shown here are not definitive to say in which cases should be used. Further testing will be done to this preconditioner in future work.

Finally, a few new ideas were seen when doing this publication, such as reduce the number of stored directions when using the Eigendecomposition, giving some new lines of research to be done.

ACKNOWLEDGEMENTS

We like to thank to the National Laboratory for Scientific Computing - LNCC for the use of the Santos Dumont super-computer, used in some of the test cases of this paper.

REFERENCES

- [1] Farhat C, Lesoinne M, Pierson K. A scalable dual-primal domain decomposition method. *Numerical Linear Algebra with Applications* 2000;7(7-8):687–714.
- [2] de La Bourdonnaye A, Farhat C, Macedo A, Magoules F, Roux FX. A Non-Overlapping Domain Decomposition Method for the Exterior Helmholtz Problem. *Contemporary Mathematics* 1998;218:42–66.

- [3] Dureisseix D, Farhat C. A numerically scalable domain decomposition method for the solution of frictionless contact problems. *International Journal for Numerical Methods in Engineering* 2001;50(12):2643–2666.
- [4] Farhat C, Roux FX. A method of finite element tearing and interconnecting and its parallel solution algorithm. *International Journal for Numerical Methods in Engineering* 1991;32(6):1205–1227.
- [5] Mandel J. Balancing domain decomposition. *Communications in Numerical Methods in Engineering* 1993;9(3):233–241.
- [6] Farhat C, Chen PS, Mandel J, Roux FX. The two-level FETI method Part II: Extension to shell problems, parallel implementation and performance results. *Computer Methods in Applied Mechanics and Engineering* 1998;155(1):153 – 179.
- [7] D R. Substructuring and dual methods in structural analysis. PhD thesis, Université de Liège, Belgium, Collection des Publications de la Faculté des Sciences appliquées, n.175; 1997.
- [8] Gosselet P, Rixen D, Roux FX, Spillane N. Simultaneous FETI and block FETI: Robust domain decomposition with multiple search directions. *International Journal for Numerical Methods in Engineering* 2015;104(10):905–927.
- [9] Rixen DJ, Farhat C. A simple and efficient extension of a class of substructure based preconditioners to heterogeneous structural mechanics problems. *International Journal for Numerical Methods in Engineering* 1999;44(4):489–516.
- [10] Hammarling S, Higham NJ, Lucas C. In: Kågström B, Elmroth E, Dongarra J, Waśniewski J, editors. *LAPACK-Style Codes for Pivoted Cholesky and QR Updating* Berlin, Heidelberg: Springer Berlin Heidelberg; 2007. p. 137–146.
- [11] Dongarra J, Du Croz J, Duff I, Hammarling S. A Proposal for a Set of Level 3 Basic Linear Algebra Subprograms. *SIGNUM News* 1987 Jul;22(3):2–14.
- [12] Suzuki A, Roux FX. A dissection solver with kernel detection for symmetric finite element matrices on shared memory computers. *International Journal for Numerical Methods in Engineering* 2014;100(2):136–164.
- [13] Karypis G, Kumar V. METIS, a Software Package for Partitioning Unstructured Graphs and Computing Fill-Reduced Orderings of Sparse Matrices. University of Minnesota, Department of Computer Science 1998;180.