DOMAIN-DECOMPOSITION-TYPE METHODS FOR COMPUTING THE DIAGONAL OF A MATRIX INVERSE *

JOK M. TANG † AND YOUSEF SAAD †

Abstract. This paper presents two methods based on domain decomposition concepts for determining the diagonal of the inverse of a sparse matrix. The first uses a divide-and-conquer principle and the Sherman-Morrison-Woodbury formula, and assumes that the matrix can be decomposed into a 2×2 block-diagonal matrix and a low-rank matrix. The second method is a standard domain decomposition approach in which local solves are combined with a global correction. Both methods can be succesfully combined with iterative solvers and sparse approximation techniques. Results of numerical experiments are reported to illustrate the performance of the proposed methods.

Key words. Matrix diagonal extraction, domain decomposition methods, divide-and-conquer method, Sherman-Morrison-Woodbury formula, Schur complement, iterative methods, sparse approximate inverse.

AMS subject classifications. 65F05, 65F10, 65F50, 65N55.

1. Introduction. Extracting diagonal entries of a matrix inverse is important in many practical applications. Examples include examining inverse covariance matrices in Uncertainty Quantification [3], finding a rational approximation for Fermi-Dirac functions in the Density Functional Theory [18], and evaluating Green's functions in the Dynamic Mean-Field Theory (DMFT) [4]. The main purpose of this paper is to present domain-decomposition-type methods for computing the diagonal entries of a matrix inverse.

We consider a large and sparse matrix, $A \in \mathbb{C}^{n\times n}$, and assume that A is nonsingular. In the main applications of our interest, A is also complex symmetric. In this paper, we are interested in finding the diagonal matrix whose diagonal entries are the same as those of A^{-1}. This matrix is denoted by $\mathcal{D}(A^{-1})$.

Extracting $\mathcal{D}(A^{-1})$ is considered as a challenging task, in part because it cannot be expressed easily in the form of a system of equations. The problem is usually harder to solve than a linear system with the same matrix A. Several methods have been proposed to compute $\mathcal{D}(A^{-1})$, see, for example, [19, Sect. 2], [9, Sect. 2], and [12, Sect. 1.2] for a literature overview. The major methods proposed in [9, 12, 19] are briefly sketched next.

If A is diagonally dominant and/or positive definite, A^{-1} may consist of many small entries. In this case, a probing method can be used to find $\mathcal{D}(A^{-1})$ efficiently [19]. In this method, A^{-1} is sparsified by neglecting the small entries, then the sparsity pattern of the sparsified A^{-1} is determined followed by a process to find appropriate probing vectors using graph coloring arguments, and, finally, linear systems are solved by a direct or iterative method. For more general matrices, the method named Fast Inverse using Nested Dissection (FIND) has been advocated by Li *et al.* [9]. It is based on nested dissection, where an elimination process is modeled by a graph that is decomposed into a tree structure. An upward and downward traversal of the tree is used to find $\mathcal{D}(A^{-1})$ efficiently. Related to this approach is the hierarchical Schur complement method of Lin *et al.* [12]. Here, a hierarchical decomposition of the computational domain is adopted. It first constructs hierarchical Schur complements of the interior points for the blocks of the domain in a bottom-up pass, and

*Work supported in part by DOE under grant DE-FG 08ER 25841, by NSF under grant OCI-0904587, and by the Minnesota Supercomputing Institute.

*Department of Computer Science and Engineering, University of Minnesota, 200 Union Street S.E, Minneapolis, MN 55455, USA. Email: {jtang, saad}@cs.umn.edu.

then extracts the diagonal entries of $\mathcal{D}(A^{-1})$ efficiently in a top-down pass by exploiting the hierarchical local dependence of inverse matrices.

This paper describes two methods based on domain decomposition concepts to compute $\mathcal{D}(A^{-1})$. The first method is based on a recursive application of the following formula:

$$\left(Y - ZZ^T\right)^{-1} = Y^{-1} + Y^{-1}Z\left(I_k - Z^TY^{-1}Z\right)^{-1}Z^TY^{-1}, \tag{1.1}$$

where $Y \in \mathbb{C}^{n\times n}$, $Z \in \mathbb{C}^{n\times k}$ with $k \leq n$, I_k is the $k \times k$ identity matrix, and both Y and $I_k - Z^TY^{-1}Z$ are assumed to be nonsingular. Eq. (1.1) is a special case of the well-known Sherman-Morrison-Woodbury formula, see, e.g., [5, Sect. 2.1.3] and [6, 7]. The formula implies that a rank-k correction to a matrix results in a rank-k correction to its inverse. Formula (1.1) will be applied recursively resulting in a divide-and-conquer approach. The second method we describe is related to the first method, and is based on standard domain decomposition ideas. By dividing the computational domain into subdomains, and ordering the grid points first by the subdomains and then by the interface points, $\mathcal{D}(A^{-1})$ can be found by solving the local problems on each subdomain and updating them by a global correction. Although we focus on finding $\mathcal{D}(A^{-1})$ in this paper, we note that the two methods we propose can also be used to extract other entries of a matrix inverse.

One can view the methods proposed in [9, 12] as belonging to the class of domain decomposition methods as well. However, there are major differences between our approaches and theirs. The methods in [9, 12] are essentially direct methods based on a sort of multi-frontal technique to obtain the desired entries of the inverse. The methods proposed in this paper are more classical domain decomposition techniques and do not emphasize exactness.

Recently, Amestoy *et al.* [2] have proposed another method for computing $\mathcal{D}(A^{-1})$. Given an LU or LDL^T factorization of A held in out-of-core storage, they have shown how to compute $\mathcal{D}(A^{-1})$ efficiently by accessing one part of the factors. The method is tested by using the MUMPS software package [1], and it has shown promising results. Related to this approach is the SelInv method of Lin *et al.* [11], which is also recently proposed. Similar to [2], the SelInv method employs a given LDL^T factorization of A to compute selected elements of the inverse of a symmetric A, where the use of supernodes and block algorithms are key to achieve high performance. The methods we propose in this paper differ from [2, 11] in the sense that we do not need an explicit matrix factorization of A and we allow the use of iterative methods and sparse approximation techniques as components of the methods.

The outline of this paper is as follows. Section 2 is devoted to the description of the divide-and-conquer method. Section 3 describes the domain decomposition method. A further discussion of the methods is provided in Section 4. Results from numerical experiments are presented in Section 5. Tentative conclusions are stated in Section 6.

2. Divide-and-Conquer Method. In this section, we present a divide-and-conquer (D&C) method for finding $\mathcal{D}(A^{-1})$. The main idea of the method is described in §2.1. We discuss its implementation in §2.2, and its application to discretized partial differential equations (PDEs) in §2.3. Finally, we consider how the method can be further improved in §2.4.

2.1. The main idea. Suppose that a nonsingular and complex symmetric matrix, $A \in \mathbb{C}^{n\times n}$, is given and can be decomposed into a 2×2 block-diagonal matrix, C, and a low-rank matrix, $-L$. If the dimension of the first block of C is denoted by m and the rank of L is denoted by q, we can write

$$A = C - L, \quad C := \begin{pmatrix} C_1 & \\ & C_2 \end{pmatrix}, \quad L := EE^T, \tag{2.1}$$

where $C_1 \in \mathbb{C}^{m\times m}$, $C_2 \in \mathbb{C}^{(n-m)\times(n-m)}$, and $E \in \mathbb{C}^{n\times q}$ satisfying $0 < q < m < n$.

The inverse of A can now be extracted by substituting $Y := C$ and $Z := E$ into the Sherman-Morrison-Woodbury formula (1.1). This yields

$$A^{-1} = C^{-1} + UR^{-1}U^T, \tag{2.2}$$

with $U \in \mathbb{C}^{n\times q}$ and $R \in \mathbb{C}^{q\times q}$ defined as

$$U := C^{-1}E, \quad R := I_q - E^T U,$$

where we assume that C and R are nonsingular. Therefore, $\mathcal{D}(A^{-1})$ can be found via

$$\mathcal{D}(A^{-1}) = \mathcal{D}(C^{-1}) + \mathcal{D}(UR^{-1}U^T). \tag{2.3}$$

2.2. Implementation. For the implementation of the D&C method, it is useful to write

$$E = \begin{pmatrix} E_1 \\ E_2 \end{pmatrix},$$

where $E_1 \in \mathbb{C}^{m\times q}$ and $E_2 \in \mathbb{C}^{(n-m)\times q}$.

By solving a sequence of q linear systems, $CU = E$, we can obtain the matrix U. Since C is a block-diagonal matrix, $CU = E$ can be decoupled into two sequences of smaller linear systems,

$$C_1U_1 = E_1, \quad C_2U_2 = E_2, \tag{2.4}$$

where $U^T = (U_1^T, U_2^T)$. Depending on the rank q and the matrix properties of C_1 and C_2, the two sequences of linear systems (2.4) can be solved best by a direct or an iterative method, see [5, 14] and Section 5 for an illustration.

Subsequently, R can be rewritten as

$$R = I_q - E_1^T U_1 - E_2^T U_2, \tag{2.5}$$

and can be readily obtained if both E_1 and E_2 are sparse. Additionally, if q is relatively small, $R^{-1} \in \mathbb{C}^{q\times q}$, can be computed easily. For a large q, the explicit inverse can be avoided by (factoring R and) solving $RW = U^T$, which is a sequence of n linear systems with a coefficient matrix of dimension q, i.e.,

$$RW_1 = U_1^T, \quad RW_2 = U_2^T, \tag{2.6}$$

where $W^T = (W_1^T, W_2^T) \in \mathbb{C}^{n\times q}$. The same solution strategy as for Eq. (2.4) can be adopted here, although R is smaller and may have a different sparsity pattern compared with C_1 or C_2.

The matrix-matrix product of U and W is usually expensive to compute (i.e., it scales with $\mathcal{O}(qn^2)$, as both matrices are dense). Fortunately, we are only interested in $\mathcal{D}(UW)$, which can be found by regarding it as it were n inner products of vectors of length q. More specifically, if we write $U^T = (u_1, \ldots, u_n)$ and $W = (w_1, \ldots, w_n)$ with $u_j, w_j \in \mathbb{C}^q$, and denote the j-th diagonal entry of UW by $\mathcal{D}_j(UW)$, then we have

$$\mathcal{D}_j(UW) = \mathcal{D}_j(\begin{pmatrix} U_1W_1 \\ U_2W_2 \end{pmatrix}) = u_jR^{-1}u_j^T = u_jw_j, \quad j = 1, \ldots, n. \tag{2.7}$$

It can be noticed from (2.7) that, if u_j is sparse, just a few entries of w_j are required to compute $\mathcal{D}_j(UW)$. This observation may be used to solve (2.6) approximately, while $\mathcal{D}(UW)$ remains accurate.

Next, finding $\mathcal{D}(C^{-1})$ is equivalent to finding $\mathcal{D}(C_1^{-1})$ and $\mathcal{D}(C_2^{-1})$. These latter problems are similar to the original problem of finding $\mathcal{D}(A^{-1})$, but they are of smaller dimensions. If direct inversion of C_1 and C_2 is too expensive, we can apply the above described solution procedure for computing $\mathcal{D}(A^{-1})$ also to finding $\mathcal{D}(C_1^{-1})$ and $\mathcal{D}(C_2^{-1})$. To do so, we assume that both C_1 and C_2 can be decomposed into a 2×2 block-diagonal matrix and a low-rank matrix (cf. Eq. (2.1)). The procedure can be further repeated for computing diagonals of smaller matrix inverses, so that we eventually end up with a recursive application of Eq. (2.3). The recursion ends when it is efficient to invert the smallest matrices explicitly.

Because of the recursive way of applying Eq. (2.3) to find $\mathcal{D}(A^{-1})$, the resulting method can be interpreted as a divide-and-conquer (D&C) method. The corresponding algorithm is presented in Algorithm 1.

Algorithm 1: A Divide-and-Conquer (D&C) method for computing $\mathcal{D}(A^{-1})$

input : $A \in \mathbb{C}^{n \times n}$
output: $\mathcal{D}(A^{-1}) \in \mathbb{C}^{n \times n}$

1 **if** $n < \rho$ for a small $\rho \in \mathbb{N}$ or A does not satisfy (2.1) **then**
2 Compute A^{-1} explicitly, and determine $\mathcal{D}(A^{-1})$
3 **else**
4 Choose $m \in \mathbb{N}$ such that $m < n$
5 Partition A as in Eq. (2.1)
6 Determine $C = \begin{pmatrix} C_1 & \\ & C_2 \end{pmatrix}$ and E
7 Compute $U := C^{-1}E$ and $R := I_q - E^T U$
8 Compute $W := R^{-1}U^T$ and $M_1 := \mathcal{D}(UW)$
9 Find $M_2^{(1)} := \mathcal{D}(C_1^{-1})$ by applying Algorithm 1 to C_1
10 Find $M_2^{(2)} := \mathcal{D}(C_2^{-1})$ by applying Algorithm 1 to C_2
11 Set $M_2 := \mathcal{D}(C^{-1}) = \begin{pmatrix} M_2^{(1)} & \\ & M_2^{(2)} \end{pmatrix}$
12 Set $\mathcal{D}(A^{-1}) := M_1 + M_2$

A few remarks on Algorithm 1 can be made. In Line 2, A^{-1} is computed explicitly, but more efficient alternatives might be used to find $\mathcal{D}(A^{-1})$ for a small n. Furthermore, we suppose that the decomposition (2.1) exists for any choice of m. In this case, there is indeed some freedom to choose m in Line 4. The problem can be divided recursively into approximately equal sizes (i.e, $m \approx \frac{n}{2}$), but this is not essential for the D&C method. Another possibility is to choose m as small as possible during the whole procedure, so that the recursion proceeds only in the direction of $\mathcal{D}(C_2^{-1})$. However, some effort is required to end up with an efficient approach, as the linear system $C_2 U_2 = E_2$ is often expensive to solve. Finally, it should be checked at all recursion levels of the algorithm whether the matrices R and C are nonsingular. In practice, it is rare that those matrices are singular, but if they were, it is often possible to force them to be nonsingular. This can be accomplished by taking other choices for the corresponding E_1 and E_2, since the decomposition (2.1) is usually not

unique.

2.3. Application to discretized PDEs. It is not difficult to show that, in general, any banded matrix can be decomposed into (2.1), where q depends on the bandwidth of that matrix. Below, we show that this particularly holds for specific discretization matrices derived from PDEs. In this case, the D&C method is suitable to find the diagonal of the inverse of those discretization matrices.

2.3.1. 2-D problems. When a 5-point centered discretization scheme is employed to discretize a 2-D elliptic PDEs with n_x grid points in the x-direction and n_y grid points in the y-direction, the resulting matrix A often takes the following form:

$$A = \begin{pmatrix} A_1 & D_1 & & & \\ D_1 & A_2 & D_2 & & \\ & \ddots & \ddots & \ddots & \\ & & D_{n_y-2} & A_{n_y-1} & D_{n_y-1} \\ & & & D_{n_y-1} & A_{n_y} \end{pmatrix}, \tag{2.8}$$

where $\{A_j\}$ and $\{D_j\}$ are sets of n_y tridiagonal and $n_y - 1$ diagonal matrices of dimension n_x, respectively, so that the size of the matrix is $n = n_x n_y$. The matrix A can be written as

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & \\ & A_{22} \end{pmatrix} + \begin{pmatrix} & A_{12} \\ A_{21} & \end{pmatrix}, \tag{2.9}$$

with $A_{11} \in \mathbb{C}^{m\times m}$ and $A_{22} \in \mathbb{C}^{(n-m)\times(n-m)}$, where we assume that m is a multiple of n_x, i.e., $m = \phi n_x$ where ϕ is an integer such that $0 < \phi < n_y$. An observation is that the submatrix $A_{12} = A_{21}^T \in \mathbb{C}^{m\times(n-m)}$ has rank n_x. This is because

$$A_{12} = A_{21}^T = \begin{pmatrix} & \\ & \\ Q & \end{pmatrix} = -E_1 E_2^T, \tag{2.10}$$

with $E_1 \in \mathbb{C}^{m\times n_x}$ and $E_2 \in \mathbb{C}^{(n-m)\times n_x}$ defined as

$$E_1 := \begin{pmatrix} & \\ & \\ Q_1 \end{pmatrix}, \quad E_2 := \begin{pmatrix} Q_2 \\ & \\ & \end{pmatrix}, \tag{2.11}$$

where $Q, Q_1, Q_2 \in \mathbb{C}^{n_x\times n_x}$ are diagonal matrices obeying $Q_1 Q_2 = -Q$. In the experiments of Section 5, we take $Q_1 = -Q$ and $Q_2 = I_{n_x}$. Eq. (2.9) can now be rewritten as

$$A = \begin{pmatrix} A_{11} + E_1 E_1^T & \\ & A_{22} + E_2 E_2^T \end{pmatrix} - \begin{pmatrix} E_1 E_1^T & E_1 E_2^T \\ E_2 E_1^T & E_2 E_2^T \end{pmatrix}, \tag{2.12}$$

which is exactly in the form of Eq. (2.1), where we have $q := n_x$, and C_1 and C_2 are given by

$$C_1 := A_{11} + E_1 E_1^T, \quad C_2 := A_{22} + E_2 E_2^T.$$

Notice that the diagonal matrix $E_1 E_1^T$ perturbs the last n_x diagonal entries of A_{11}, while the diagonal matrix $E_2 E_2^T$ perturbs the first n_x diagonal entries of A_{22}. Consequently, C_1 and C_2 may be better conditioned than A_{11} and A_{22}, which is usually a favorable property if iterative

methods are exploited to solve (2.4). By recalling that $U := C^{-1}E$ and $U^T = (U_1^T, U_2^T)$, we also notice that $U_1 \in \mathbb{C}^{m\times n_x}$ corresponds to the scaled last n_x columns of C_1^{-1}, whereas $U_2 \in \mathbb{C}^{(n-m)\times n_x}$ corresponds to the scaled first n_x columns of C_2^{-1}.

The next example shows how the matrices at different recursion levels of the D&C approach look like for a specific problem.

EXAMPLE 2.1. *Suppose that we have the following discretization matrix:*

$$A = \begin{pmatrix} A_1 & D_1 & & & \\ D_1 & A_2 & D_2 & & \\ & D_2 & A_3 & D_3 & \\ & & D_3 & A_4 & D_4 \\ & & & D_4 & A_5 \end{pmatrix}.$$

$\mathcal{D}(A^{-1})$ can be found by Algorithm 1, where we need to decompose A into (2.1). This can be done by choosing $\rho := q = n_x$, and $m := \lceil \frac{n}{2} \rceil$. This leads to

$$C_1 = \begin{pmatrix} A_1 & D_1 & \\ D_1 & A_2 & D_2 \\ & D_2 & A_3 + D_3^2 \end{pmatrix}, \quad C_2 = \begin{pmatrix} A_4 + I_{n_x} & D_4 \\ D_4 & A_5 \end{pmatrix}, \quad E_1 = \begin{pmatrix} \\ \\ -D_3 \end{pmatrix}, \quad E_2 = \begin{pmatrix} I_{n_x} \\ \\ \end{pmatrix}.$$

Finding $\mathcal{D}(C^{-1})$ is equivalent to computing the two expressions $\mathcal{D}(C_1^{-1})$ and $\mathcal{D}(C_2^{-1})$ by Algorithm 1. We next show how this can be done for $\mathcal{D}(C_1^{-1})$. The case with $\mathcal{D}(C_2^{-1})$ can be treated in an analogous way. For the sake of clarity, we will put a bar on matrices when they act on the second recursion level. Likewise, a double bar will be used when matrices act on the third recursion level.

We set $\bar{A} := C_1$ in Eq. (2.1), and derive

$$\bar{C}_1 = \begin{pmatrix} A_1 & D_1 \\ D_1 & A_2 + D_2^2 \end{pmatrix}, \quad \bar{C}_2 = A_3 + D_3^2 + I_{n_x}, \quad \bar{E}_1 = \begin{pmatrix} \\ \\ -D_2 \end{pmatrix}, \quad \bar{E}_2 = \begin{pmatrix} I_{n_x} \\ \\ \end{pmatrix}.$$

Next, direct inversion can be used to determine $\mathcal{D}(\bar{C}_2^{-1})$, while one more recursion level of the D&C approach is required to find $\mathcal{D}(\bar{C}_1^{-1})$. For this third recursion level, we obtain

$$\bar{\bar{C}}_1 = A_1 + D_1^2, \quad \bar{\bar{C}}_2 = A_2 + D_2^2 + I_{n_x}, \quad \bar{\bar{E}}_1 = -D_1, \quad \bar{\bar{E}}_2 = I_{n_x}.$$

Both $\mathcal{D}(\bar{\bar{C}}_1^{-1})$ and $\mathcal{D}(\bar{\bar{C}}_2^{-1})$ can now be found by direct inversion.

2.3.2. 3-D problems. The D&C method can also be used to solve specific 3-D PDE problems. In this case, we assume that matrix A is derived from a 7-point stencil discretization, so that it can still be written in a similar way as Eq. (2.8), i.e.,

$$A = \begin{pmatrix} A_1 & D_1 & & & & \\ D_1^T & A_2 & D_2 & & & \\ & \ddots & \ddots & \ddots & & \\ & & D_{n_y n_z - 2}^T & A_{n_y n_z - 1} & D_{n_y n_z - 1} \\ & & & D_{n_y n_z - 1}^T & A_{n_y n_z} \end{pmatrix}. \tag{2.13}$$

In Eq. (2.13), $\{A_j\}$ is a set of $n_y n_z$ tridiagonal matrices of dimension n_x, and $\{D_j\}$ is a set of $n_y n_z - 1$ rectangular matrices with dimensions n_x by at most $n_x n_y$ of the following form:

$$D_j = \begin{cases} \begin{pmatrix} \tilde{D}_j & 0_{n_y - 2} & \hat{D}_j \end{pmatrix}, & \text{if } j = 1, \ldots, n_y n_z - n_y; \\ \begin{pmatrix} \tilde{D}_j & 0_{n_y n_z - j - 1} \end{pmatrix}, & \text{if } j = n_y n_z - n_y + 1, \ldots, n_y n_z - 2; \\ \tilde{D}_j, & \text{if } j = n_y n_z - 1, \end{cases}$$

where $\tilde{D}_j, \hat{D}_j \in \mathbb{C}^{n_x \times n_x}$ are diagonal matrices, and 0_l is the n_x-by-ln_x zero matrix.

The same procedure as presented in §2.3.1 can now be adopted to find $\mathcal{D}(A^{-1})$ based on Eq. (2.13). The most noticeable difference compared with the 2-D problem is the fact that $A_{12} = A_{21}^T$ now has rank $n_x n_y$, yielding $E, U \in \mathbb{C}^{n \times (n_x n_y)}$ and $R \in \mathbb{C}^{(n_x n_y) \times (n_x n_y)}$. This usually requires a larger computational effort to solve the sequence of $n_x n_y$ linear systems (2.4) and to compute $\mathcal{D}(UR^{-1}U^T)$ as in (2.7). Hence, the cost of finding $\mathcal{D}(A^{-1})$ for the 3-D problem can be significantly higher than in the 2-D case.

2.4. Possible improvements. The most time-consuming tasks of the D&C method are usually solving $C_j U_j = E_j$ and $RW_j = U_j^T$. Below, we show how those computations can be carried out more efficiently.

Because U_j is mostly a dense matrix, it is not straightforward to improve the solution procedure of $RW_j = U_j^T$ and the computation of $\mathcal{D}(U_j W_j)$. However, if just an approximation of $\mathcal{D}(A^{-1})$ suffices, we may employ a dropping strategy when computing U_j to reduce the computational cost of obtaining $U_j W_j$. The matrix U_j is replaced by $\tilde{U}_j$, which is identical to U_j except for the entries that are smaller than a relative tolerance $0 < \epsilon \ll 1$ in modulus. These small entries of U_j are discarded. More specifically, by writing $U_j = [u_{kl}]$ and $\tilde{U}_j = [\tilde{u}_{kl}]$, we define each entry of $\tilde{U}_j$ as follows:

$$\tilde{u}_{kl} = \begin{cases} u_{kl}, & \text{if } |u_{kl}| < \epsilon ||U||_2; \\ 0, & \text{otherwise.} \end{cases} \tag{2.14}$$

If R^{-1} consists of entries with values around $\mathcal{O}(1)$, then the error of computing $\mathcal{D}(U_j W_j)$ is $\mathcal{O}(\epsilon^2)$, while the cost can be significantly reduced.

The procedure for solving the sequence $C_j U_j = E_j$ can be avoided explicitly by noting that E_j is sparse, and, therefore, $U_j = C_j^{-1} E_j$ just involves the n_x nonzero columns of C_j^{-1} associated with the nonzero rows of E_j. These nonzero columns of C_j^{-1} can be computed via a recursive application of Eq. (2.2). To be more specific, suppose that the r_j-th to r_{j+n_x}-th rows of E_j are nonzero. Then, we define $\mathcal{P}(C_j^{-1})$ as a matrix with the same dimension as C_j^{-1}, where the r_j-th to the r_{j+n_x}-th columns of $\mathcal{P}(C_j^{-1})$ and C_j^{-1} are identical, and the remaining columns of $\mathcal{P}(C_j^{-1})$ are zero. The nonzero entries of $\mathcal{P}(C_j^{-1})$ can be found at the next recursion level when $\mathcal{D}(C_j^{-1})$ is computed, i.e., from Eq. (2.2), we obtain

$$\mathcal{P}(C_j^{-1}) = \mathcal{P}(\bar{C}_j^{-1}) + \mathcal{P}(\bar{U}_j \bar{W}_j), \tag{2.15}$$

where a bar is put on the matrices to stress that those are from a higher recursion level (cf. Example 2.1). If $\mathcal{P}(\bar{C}^{-1})$ is found by applying Eq. (2.15) again, we eventually end up with a recursive procedure. Note that solving $U_j = C_j^{-1} E_j$ based on Eq. (2.15) may not be practical, because $\bar{U}_j$ is dense, and, therefore, the computation of $\mathcal{P}(\bar{U}_j \bar{W}_j)$ can be expensive. However, if we combine the above strategy with an additional dropping procedure to $\bar{U}_j$, we may end up with a successful and improved variant of the original D&C method. This is further examined in §5.1.2.

3. Domain decomposition method. By taking a closer look at the matrix splitting (2.1) and the inverse expression (2.2), we observe that the physical domain in the D&C approach is divided artificially into two subdomains, Ω_1 and Ω_2. The block-diagonal matrix C^{-1} can be interpreted as a result of a discretization of the problem in which the interaction between Ω_1 and Ω_2 is neglected. A correction based on the interaction of these subdomains is characterized by the term $\mathcal{D}(UR^{-1}U^T)$, which is added to $\mathcal{D}(C^{-1})$ in order to form $\mathcal{D}(A^{-1})$. This idea of decoupling subdomains is then used at all recursion levels.

The approach of finding $\mathcal{D}(A^{-1})$ by decoupling the problem and correcting the solution by the interaction of subdomains can also be viewed from a domain decomposition point of view. In fact, we can derive another approach to compute $\mathcal{D}(A^{-1})$, which is more general than the D&C approach seen earlier. This domain decomposition (DD) method for finding $\mathcal{D}(A^{-1})$ is presented in this section. The main idea is described in §3.1. Some implementation issues are discussed in §3.2. Subsequently, the method is applied to problems derived from PDEs in §3.3, and some possible improvements of the DD method are provided in §3.4.

3.1. The main idea. We assume that, after possible row and column permutations, the nonsingular and complex symmetric matrix A has the following form:

$$A = \begin{pmatrix} B_1 & & & & F_1 \\ & B_2 & & & F_2 \\ & & \ddots & & \vdots \\ & & & B_p & F_p \\ F_1^T & F_2^T & \cdots & F_p^T & G \end{pmatrix} =: \begin{pmatrix} B & F \\ F^T & G \end{pmatrix}, \tag{3.1}$$

where $B_j \in \mathbb{C}^{n_j \times n_j}$, $F_j \in \mathbb{C}^{n_j \times n_G}$, $G \in \mathbb{C}^{n_G \times n_G}$, and $n_G + n_B = n$ with $n_B := \sum_{j=1}^{p} n_j$. Taking the inverse of Eq. (3.1) yields (cf. [14, Eq. (14.7)])

$$A^{-1} = \begin{pmatrix} B^{-1} + B^{-1}FS^{-1}F^TB^{-1} & -B^{-1}FS^{-1} \\ -S^{-1}F^TB^{-1} & S^{-1} \end{pmatrix}, \quad S := G - F^TB^{-1}F, \tag{3.2}$$

where both B and S are assumed to be nonsingular. Notice that the upper-diagonal block of Eq. (3.2) corresponds to the original Sherman-Morrison-Woodbury formula, and is in the form of Eq. (1.1) when $G = I_{n_G}$. Now, $\mathcal{D}(A^{-1})$ can be obtained via

$$\mathcal{D}(A^{-1}) = \begin{pmatrix} \mathcal{D}(B^{-1}) + \mathcal{D}(HS^{-1}H^T) & \\ & \mathcal{D}(S^{-1}) \end{pmatrix}, \quad H := B^{-1}F. \tag{3.3}$$

3.2. Implementation. Since B is a block-diagonal matrix, $H \in \mathbb{C}^{n_B \times n_G}$ can be computed by solving p sequences of n_G linear systems (cf. Eq. (2.4)), i.e.,

$$B_jH_j = F_j, \quad j = 1, 2, \ldots, p, \tag{3.4}$$

where $H^T = (H_1^T, H_2^T, \ldots, H_p^T)$ and $H_j \in \mathbb{C}^{n_j \times n_G}$.

The global Schur complement, S, can be computed via (cf. Eq. (2.5))

$$S = G - \sum_{j=1}^{p} F_j^T H_j, \tag{3.5}$$

where each term of the sum can be regarded as a local contribution, and the sparsity of F_j can be further exploited.

If n_G is relatively small, S^{-1} can be formed explicitly from which $\mathcal{D}\left(S^{-1}\right)$ easily follows. For large values of n_G, alternative methods are required to determine $\mathcal{D}\left(S^{-1}\right)$. It is known that the Schur complement, S, is generally well-approximated by a sparse matrix, and, in this situation, its diagonal can be obtained recursively. Ideas of this type are common when linear systems are solved by domain-decomposition-type methods, see, e.g., [16].

Subsequently, the term $\mathcal{D}(HS^{-1}H^T)$, which appears in the upper-diagonal block of Eq. (3.3), can be obtained in two stages. First, we solve a sequence of n_B linear systems

of the form $SX = H^T$ with $X \in \mathbb{C}^{n_G \times n_B}$. This computation can be carried out locally by distributing S to each subdomain and solving (cf. Eq. (2.6))

$$SX_j = H_j^T, \quad j = 1, 2, \ldots, p, \tag{3.6}$$

by a direct method (using, e.g., a matrix factorization of S) or an iterative method, and setting $X^T = (X_1^T, X_2^T, \ldots, X_p^T)$ where $X_j \in \mathbb{C}^{n_G \times n_j}$. Second, $\mathcal{D}(HX)$ can be formed similarly to Eq. (2.7). This means that each block of $\mathcal{D}(HX)$ can be computed locally by $\mathcal{D}(H_j X_j)$, in which each diagonal component is determined by

$$\mathcal{D}_k(H_j X_j) = h_k S^{-1} h_k^T = h_k x_k, \quad k = 1, 2, \ldots, n_j, \quad j = 1, 2, \ldots, p, \tag{3.7}$$

where $H_j^T = (h_1, \ldots, h_{n_j})$ and $X_j = (x_1, \ldots, x_{n_j})$ with $h_j, x_j \in \mathbb{C}^{n_G}$.

Finding $\mathcal{D}(B^{-1})$ is equivalent to finding all $\mathcal{D}(B_j^{-1})$ for $j = 1, 2, \ldots, p$. Each of $\mathcal{D}(B_j^{-1}) \in \mathbb{C}^{n_j \times n_j}$ can be computed locally, and the best method to use depends on the exact value of n_j and the sparsity of B_j. The above analysis for $\mathcal{D}\left(S^{-1}\right)$ is also applicable here: if n_j is small, B_j can be directly inverted, otherwise an alternative method, such as a recursive application of the domain decomposition approach, can be used to determine $\mathcal{D}(B_j^{-1})$.

Finally, we can combine the above results to determine $\mathcal{D}(B^{-1}) + \mathcal{D}(HS^{-1}H^T)$, which is the upper-diagonal block of Eq. (3.3). Each sub-block can be found locally via $\mathcal{D}(B_j^{-1}) + \mathcal{D}(H_j X_j)$ for $j = 1, 2, \ldots, p$. This yields

$$\mathcal{D}(A^{-1}) = \begin{pmatrix} \mathcal{D}(B_1^{-1}) + \mathcal{D}(H_1 X_1) & & & \\ & \ddots & & \\ & & \mathcal{D}(B_p^{-1}) + \mathcal{D}(H_p X_p) & \\ & & & \mathcal{D}\left(S^{-1}\right) \end{pmatrix}. \tag{3.8}$$

The resulting DD method for extracting $\mathcal{D}(A^{-1})$ is sketched in Algorithm 2.

Of course, if the matrix A is already based on a domain decomposition ordering, the first part of Line 1 (permute A) and Line 13 (repermute A) of Algorithm 3 are not needed. Moreover, we note that the algorithm is originally not a recursive one like Algorithm 1, although it can be made recursive by employing Algorithm 3. This recursive procedure of the DD method is used in the experiments of Section 5. Finally, we note that, in specific situations, the implementation of the DD method can be made more efficient by, e.g., exploiting the sparsity pattern of A, see §3.3.

3.3. Application to discretized PDEs. For a coefficient matrix derived from discretized PDEs, it is well-known that Eq. (3.1) can be obtained by rearranging grid points in the original physical domain, Ω. This Ω can be partitioned into p subdomains, $\{\Omega_j\}$, where interface points are defined explicitly. This is in contrast to the D&C approach, where it is implicitly assumed that the interface is between grid points, see Figure 3.1 for an illustration.

Next, suppose that there is no interaction between the open subdomains (i.e., subdomains excluding their interfaces). Then, under the usual ordering of interior points of the subdomains followed by the interface points, we obtain a matrix A of the form (3.1). A typical example of such a matrix based on an original and a domain decomposition ordering is presented in Figure 3.2.

Each F_j represents a set of edges from the interior points of subdomain Ω_j to its interface points. The number of nonzero columns of F_j, denoted by n_{F_j}, is equal to the number of these interface points that are adjacent to Ω_j. Therefore, for a relatively large p, we have

Algorithm 2: A Domain Decomposition (DD) method for computing $\mathcal{D}(A^{-1})$

input : $A \in \mathbb{C}^{n\times n}$
output: $\mathcal{D}(A^{-1}) \in \mathbb{C}^{n\times n}$

Permute variables such that A is in the form of Eq. (3.1)
Determine $\{B_j\}, \{F_j\}$ and G
Set $S := G$
for $j = 1, 2, \ldots, p$ **do**
    Compute $M_1^{(j)} := \mathcal{D}(B_j^{-1})$
    Compute $H_j := B_j^{-1} F_j$
    Update $S := S - F_j^T H_j$
for $j = 1, 2, \ldots, p$ **do**
    Compute $X_j := S^{-1} H_j^T$
    Update $M_1^{(j)} := M_1^{(j)} + \mathcal{D}(H_j X_j)$
Find $M_2 := \mathcal{D}\left(S^{-1}\right)$
Set $\mathcal{D}(A^{-1}) := \begin{pmatrix} M_1^{(1)} & & & \\ & \ddots & & \\ & & M_1^{(p)} & \\ & & & M_2 \end{pmatrix}$
Repermute variables in $\mathcal{D}(A^{-1})$

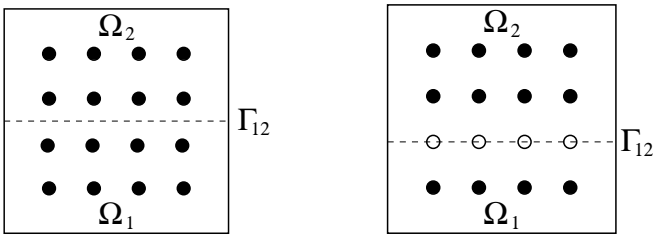
Algorithm 3: A recursive procedure for computing $\mathcal{D}(B_j^{-1})$

input : $B_j \in \mathbb{C}^{n_j\times n_j}$ from Algorithm 2
output: $\mathcal{D}(B_j^{-1}) \in \mathbb{C}^{n_j\times n_j}$

if $n_j < \psi$ for a small $\psi \in \mathbb{N}$ **then**
    Compute B_j^{-1} explicitly, and determine $\mathcal{D}(B_j^{-1})$
else
    Find $\mathcal{D}(B_j^{-1})$ by the domain decomposition method (Algorithm 2)

$\sum_j n_{F_j} \ll n_G$. This can be exploited in the implementation of the DD method, which is explained next.

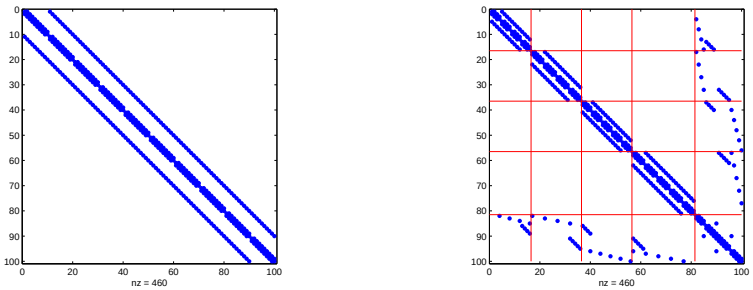
We first note that each H_j can interpret each H_j as a 'trace' of the local inverse on the interface points, and has at least n_{F_j} nonzero columns. Accordingly, the number of linear systems to be solved in Eq. (3.4) can be reduced from n_G to n_{F_j} for each j. Additionally, the sparsity of the right-hand side of each of these linear systems can also be exploited in the corresponding solver. Next, we note that the matrix X_j is usually a dense matrix, in contrast to H_j. In other words, the number of linear systems in the sequence of (3.6) cannot be easily reduced, which is different from the case of Eq. (3.4). Subsequently, by exploiting the zero columns in Eq. (3.7), the number of inner products for each j can be reduced from n_j to n_{F_j}. In addition, it can be noticed from the sparsity pattern of H_j that, for the computation of $\mathcal{D}(H_j X_j)$, it is not necessary to form X_j explicitly, since not all but at most $n_{F_j}^2$ entries of S^{-1} are required for each j. These entries correspond to the interface points of Ω_j. In an



(a) D&C method with $m = \frac{1}{2}n_x n_y$.

(b) DD method with $p = 2$, $n_G = n_1 = \frac{1}{2}n_2 = n_x$.

FIG. 3.1. *Illustration of the geometry (of the first recursion level) of the DD and D&C method based on a discretization matrix derived from PDEs. The underlying domain, Ω, with $n_x = n_y = 4$ grid points is divided into two subdomains, Ω_1 and Ω_2. Black and white dots represent interior and interface points, respectively.*



(a) Standard ordering.

(b) Domain decomposition ordering with $p = 4$ subdomains.

FIG. 3.2. *Illustration of the sparsity pattern of a matrix A with two different types of ordering of grid points. The matrix is derived from a five-point finite-difference discretization of a standard 2-D Laplacian equation with $n_x = n_y = 10$.*

efficient implementation, this might be further exploited.

Note that, except for the assumption that the open subdomains do not interact, the DD method does not make further assumptions on the underlying physical problem or its dimensions, so the method is generally applicable. Furthermore, the parallel implementation of Algorithm 2 can be done in a natural way. One or more subdomains are assigned to a processor and the computations of the Schur complements are similar to those encountered when solving linear systems of equations [16]. Increasing the number of subdomains, p, leads to smaller dimensions of each B_j, but also to a larger G. As is well-known, the dimensions of G may be quite large in the 3-D case, since interfaces between subdomains become surfaces in this case. As a consequence, the cost of Algorithm 2 is usually significantly larger for 3-D problems relative to 2-D problems.

3.4. Possible improvements. The DD approach can be improved in a way similar to the D&C method: a dropping procedure can be added to create a sparser matrix H_j, while both the computation of $B_j H_j = F_j$ and $SX_j = H_j^T$ can be carried out more efficiently by using information from higher recursion levels. The latter issue is not as straightforward as for the D&C approach, since a different domain decomposition ordering of variables is

usually used at every recursion level. Careful book keeping is required to retrieve the relevant information from higher recursion levels, see also §4.3.

Note that there is some freedom in choosing the parameters of the DD method. For instance, the choice of interface points in the computational domain can be varied if the matrix is derived from PDEs. As depicted in Figure 3.2, we assume that subdomains may share the same interface, while this is not essential for the method. For a fixed number of subdomains, the number of interface points in the domain decomposition setting can be increased such that each interface is connected to only one subdomain, and, therefore, the subdomains are fully disjoint. Although the dimension of the Schur complement matrix, S, becomes larger in this case, it may yield better properties of this matrix, so that corresponding computations can be carried out more efficiently, see, e.g., [10] for more details. Furthermore, the number of subdomains, p, can also be varied, each requiring a different efficient implementation. For example, when p is small, attention should be paid to reducing the cost of $B_j H_j = F_j$ rather than $SX_j = H_j^T$, while the opposite is true for a large value of p.

4. Further discussion. The D&C and DD methods are strongly related to each other. As can be seen in their derivation and implementation, similar expressions are used to find $\mathcal{D}(A^{-1})$. The cost of the related computations are, however, rather different. We examine this issue in §4.1. Subsequently, we explain in §4.2 which considerations should be made to decide which method to select. Finally, §4.3 discusses some relations with existing methods in the literature.

4.1. Comparison of computational efforts. Here, we explain that the major computations in the D&C and DD approaches are similar but have different costs. For this, we assume that the methods are applied to solve problems derived from 2-D PDEs. Additionally, suppose first that p is small, so that R and S have comparable dimensions.

One of the major expenses in the D&C method is when solving $C_j U_j = E_j$ for $j = 1, 2$ (see Eq. (2.4)) on each recursion level. The number of columns of U_j is fixed to n_x for each level, and corresponds to the interface connection of the underlying subdomain. On the initial recursion level, the dimension of C_j may be relatively large, say $m \approx \frac{n}{2}$. On the other hand, we have to solve the DD equivalence $B_j H_j = F_j$ for $j = 1, \ldots, p$ (see Eq. (3.4)). The number of columns of H_j is usually larger than that of U_j (i.e., $n_G \gg n_x$), but, as noticed in §3.3, H_j has only n_{F_j} nonzero columns, which are associated with the interface connection of the underlying subdomain like in the D&C method. If a recursive procedure of the DD approach is applied, the number of nonzero columns of H_j usually becomes smaller at each lower recursion level, which is in contrast to the D&C method. In addition, it can be noticed that the size of B_j is usually smaller than that of C_j, especially if $p \gg 1$. Hence, the total computations with $B_j H_j = F_j$ are often cheaper to perform than those with $C_j U_j = E_j$.

The other major expense in the D&C method is when solving $RW_j = U_j^T$ for $j = 1, 2$ (see Eq. (2.6)), or, equivalently, when computing $W_j = R^{-1} U_j^T$ if R^{-1} is available. In both cases, the corresponding cost is around $\mathcal{O}(n_x^2 n)$, since U is a dense matrix. The DD equivalent is $SX_j = H_j^T$ (see Eq. (3.6)), which is solved for $j = 1, \ldots, p$. Although S has a larger dimension than R, the sequence of linear systems $SX_j = H_j^T$ is easier to evaluate than $RW_j = U_j^T$ for a small p. This is because H_j may have a significant number of zero rows while U_j is dense. In addition, if S^{-1} is available, $X_j = S^{-1} H_j^T$ can be formed with just a selected number of entries of S^{-1}. In either case, the cost of computing each X_j is around $\mathcal{O}(n_G^2 n_{F_j})$, which is usually smaller than $\mathcal{O}(n_x^2 n)$ in 2-D problems. The cost can be even further reduced for the DD case, when the sparsity pattern of H_j is fully exploited. As mentioned in §3.3, we can often compute $\mathcal{D}(H_j X_j)$ by just using a selected number of entries of X_j.

If we apply the DD method with $p = 2$ and a recursive way of solving local problems, the resulting approach would be close to the D&C method. By using a similar implementation of their components, both methods would have a comparable cost. However, even though both methods belong to the same broad class of domain decomposition techniques, they are not mathematically equivalent, not even after permuting rows and columns. If the problem is derived from a discretization of PDEs, one difference becomes clearly apparent by considering the underlying geometry. In the DD approach, explicit interface points are created, while this is not the case in the D&C approach, see also Figure 3.1. We prefer to choose $p > 2$ in the DD method in order to alleviate aforementioned computational problems associated with the D&C method. This is further illustrated in Section 5.

If p is relatively large, solving $SX_j = H_j^T$ and especially computing $\mathcal{D}\left(S^{-1}\right)$ dominate the cost in the DD method. The implementation of the resulting approach is further discussed in §4.3.

4.2. Choice of the method. One advantage of the D&C approach is that its implementation is straightforward for a matrix A that is based on a standard ordering of grid points. In addition, the D&C method is a black-box approach, where obvious choices for m, E_1 and E_2 can be taken. The main drawbacks of the D&C method, as it is formulated here, is that it is only restricted to matrices that can be decomposed into the form of (2.1), its parallel implementation is not straightforward, and the computational cost may be higher than that of the DD method.

The advantage of the DD method is that it is independent of the matrix structure of A, and can be applied to basically any sparse matrix A that has the form (3.1) after permutations. In other words, it is not necessary to assume that A can be decomposed into a 2×2 block-diagonal matrix and a low-rank matrix as shown in Eq. (2.1), so that Algorithm 2 is more generally applicable. In addition, the DD method is straightforward to parallelize by associating one or more subdomains with each available processor. Parallel efficiency can be quite good, since we can perform most computations locally. Finally, as mentioned above, the computational effort for the DD method seems to be lower than the D&C method for specific choices of the parameters. On the other hand, the DD method is often a nested method, because it usually requires another method to solve $\mathcal{D}(B_j^{-1})$ and $\mathcal{D}\left(S^{-1}\right)$ efficiently. The best choice of this inner method depends on the exact value of n and p, so that some fine-tuning of parameters in the DD method is often required to obtain the best implementation in each practical situation. Furthermore, for $p \gg 1$, the number of interface points in the DD approach is much larger than for the D&C approach at the first recursion level. Consequently, when sparsity is not fully exploited, computations with S and S^{-1} might be more expensive than those with their D&C counterparts R and R^{-1}.

4.3. Related work. As mentioned in the introduction, the D&C and DD methods are somewhat related to the methods that are proposed in [9, 12]. The starting point of the DD method is quite similar to that of the hierarchical Schur complements (HSC) method [12] and the Fast Inverse using Nested Dissection (FIND) method [9]. The DD method can be viewed as a general method in which it is allowed to optimize various components in order to obtain an efficient implementation for specific problems.

The HSC method [12] starts with a hierarchical Schur complements of the interior points for the subdomains, and then extracts $\mathcal{D}(A^{-1})$ efficiently by exploiting the hierarchical local dependence of inverse matrices. The first level in the hierarchical domain decomposition of the HSC method is basically the first recursion level in the DD method, where p is chosen as large as possible, so that the number of interior points of each subdomain is small. In this case,

each B_j^{-1} can be computed with a low cost, so that $\mathcal{D}(B_j^{-1})$ and $H_j = B_j^{-1}F$ can be readily obtained. The major challenge here is to compute $\mathcal{D}(S^{-1})$ and $\mathcal{D}(HS^{-1}H^T)$ in an efficient way. This is achieved by applying the DD method recursively to find $\mathcal{D}(S^{-1})$, where, at each recursion level, the relevant entries of S^{-1} that are required for $\mathcal{D}(HS^{-1}H^T)$ are computed. We note that this latter procedure is not straightforward, and requires careful book keeping in the implementation. The principle of the FIND method [9] is essentially the same as the HSC method. The method can be regarded as an approach in which many LU factorizations on A are performed to compute $\mathcal{D}(A^{-1})$. By a proper reordering of A, the fill-in of the factorizations can be minimized, while every entry of $\mathcal{D}(A^{-1})$ can be found relatively easily. The cost to compute new LU factorizations is limited, since many intermediate results of the factorizations are identical. As in the case of the HSC method, the practical implementation is based on hierarchical Schur complements and hierarchical local dependences of inverses.

In the D&C and DD methods, exactness of the computations is not necessary, so that iterative methods and sparse approximation techniques can be exploited to accelerate the methods. In addition, we recall that there is some freedom to implement the different components especially in the DD method, so that it is flexible to use.

5. Numerical experiments. This section presents the results of some numerical experiments to illustrate the divide-and-conquer (D&C) and domain decomposition (DD) method for computing $\mathcal{D}(A^{-1})$. Three test problems are considered in the experiments: a toy problem based on a complex-shifted Laplacian, and a small instance of the more realistic applications of the Uncertainty Quantification and Dynamic Mean-Field Theory mentioned in the introduction. The computations are performed in MATLAB 7.4.0 on a sequential LINUX machine (Dell Precision T5500 with a Quad-core Intel Xeon 5500 series processor and 4 GB memory).

The D&C method is based on Algorithm 1, where we have $q = n_x$ and take $Q_1 := -Q$, $Q_2 := I$, $\rho := \gamma q$ with $\gamma := 3$, $m = \phi q$ with $\phi := \text{round}(\frac{n_y}{2})$ (i.e., m is the nearest integer of $\frac{n_y}{2}$ multiplied by q) at all recursion levels, unless stated otherwise. Here, n_x and n_y denote the grid sizes in the x- and y-direction of the specific 2-D problem, respectively. At the highest recursion level, A^{-1} is directly inverted by the MATLAB command `inv(A)`. Moreover, the computations $U := C^{-1}E$ and $W := R^{-1}U^T$ are done via the MATLAB commands `U = C \ E` and `W = R \ U'`, respectively.

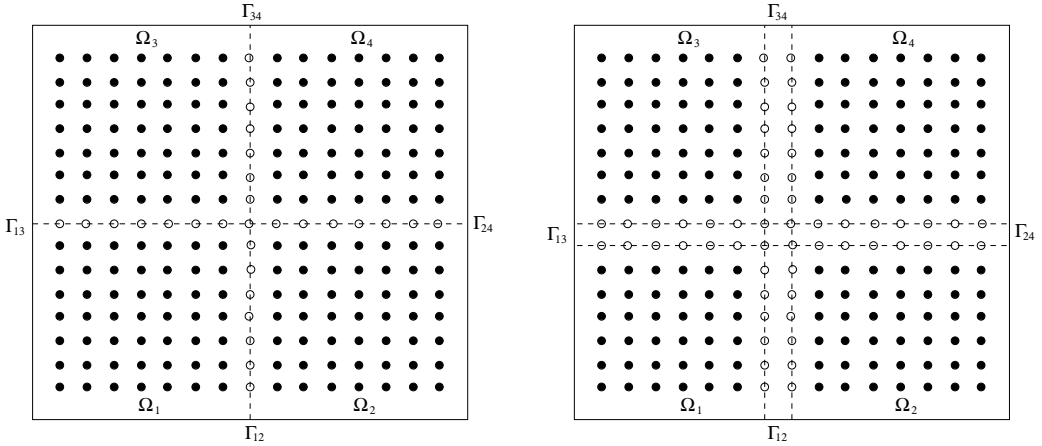
The DD method is based on Algorithm 2. We take $p = 4$ subdomains for every recursion level, unless stated otherwise. A recursive procedure of the DD method is applied to find $\mathcal{D}(B_j^{-1})$ for each j, i.e., Algorithm 3 is applied to extract $\mathcal{D}(B_j^{-1})$. As we only consider uniform and structured meshes in the test problems, squares and rectangles can be taken as the subdomains $\{\Omega_j\}$, where $n_j \approx \frac{n-n_G}{p}$ for all j. More specifically, we define $(n_x)_j$ and $(n_y)_j$ as the horizontal and vertical grid sizes of a discretized subdomain Ω_j, respectively. Assume that there are p_x and p_y subdomains in each horizontal and vertical direction, respectively, so that $p = p_x p_y$. We define the horizontal and vertical thickness of the interface as θ_x and θ_y, see Figure 5.1 for an illustration. We usually take $\theta_x = \theta_y = 1$, but the value is increased when it is required for the specific test problem. Then, we choose

$$(n_x)_j = \begin{cases} \lfloor \frac{n_x - \theta_x(p_x-1)}{p_x} \rfloor, & \text{if } j = 1, 2, \ldots, p_x - 1; \\ n_x - \sum_{j=1}^{p_x-1} \left[(n_x)_j + \theta_x \right], & \text{otherwise,} \end{cases}$$

and, likewise,

$$(n_y)_j = \begin{cases} \lfloor \frac{n_y - \theta_y(p_y-1)}{p_y} \rfloor, & \text{if } j = 1, 2, \ldots, p_y - 1; \\ n_y - \sum_{j=1}^{p_y-1} \left[(n_y)_j + \theta_y \right], & \text{otherwise.} \end{cases}$$

An illustration of the domain decomposition setting for the first recursion level and a part of the second recursion level is given in Figure 5.2. We note that, of course, more advanced (automatic) domain decomposition, such as METIS [8], can be adopted, especially if the DD method is applied to problems with an irregular and unstructured mesh.



(a) $\theta_x = \theta_y = 1$.

(b) $\theta_x = \theta_y = 2$.

(c) $\theta_x = \theta_y = 3$.

FIG. 5.1. *Geometry of a domain decomposition setting with $p = 4$, $n_x = n_y = 15$, and various choices for θ_x and θ_y. Black and white dots represent interior and interface points, respectively.*

The termination criterion of the recursion procedure in the DD method is given in Line 1 of Algorithm 3. Based on the above choice of the domain decomposition setting, this criterion can be replaced by the following expression:

$$(n_x)_j < \psi\left(\theta_x(p_x - 1) + p_x\right) \quad \text{or} \quad (n_y)_j < \psi\left(\theta_y(p_y - 1) + p_y\right),$$

where we choose $\psi = 2$ in the experiments. In addition, as p is kept small in the DD method, S is small as well, and, hence, $\mathcal{D}(S^{-1})$ can be extracted relatively easily by using the MATLAB command `invS = inv(S)`. Moreover, the nonzero columns of $H_j = B_j^{-1}F_j$ are computed

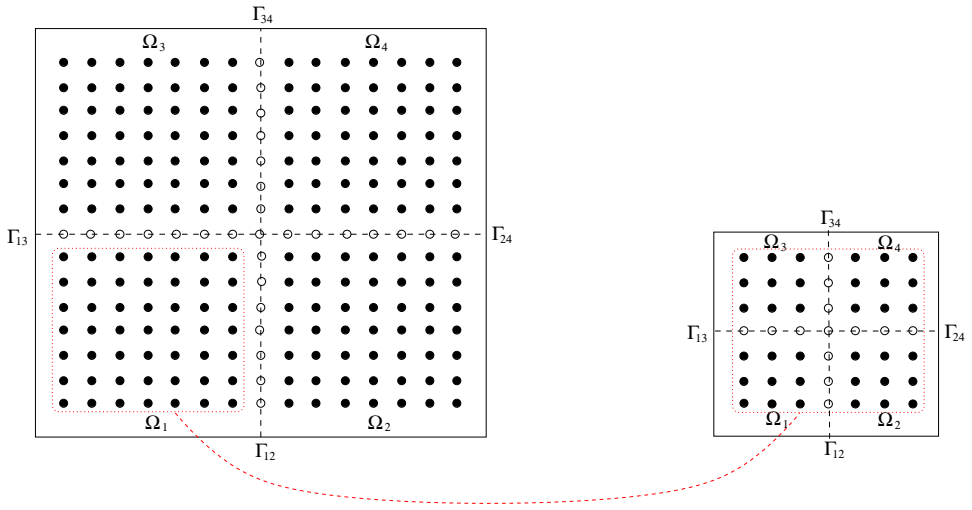


FIG. 5.2. *An example of the domain decomposition setting at the first and a part of the second recursion level of the DD method. Black and white dots represent interior and interface points, respectively. The parameters in this example are* $p = 4$, $\theta_x = \theta_y = 1$ *(so that* $n_G = n_x + n_y - 1$*), and* $n_x = n_y = 15$.

by first determining the nonzero columns of F_j via `any(Fj)`, followed by `Hsj = Bj \ Fsj`, where `Fsj` and `Hsj` represent F_j and H_j in which the zero columns are omitted. In addition, because H_j is quite sparse in our experiments, it is not necessary to compute all entries of the dense matrix X_j in order to obtain $\mathcal{D}(H_j X_j)$. Instead, we only compute the relevant entries of X_j by `Xrj = invS * Hrj`, where `Xrj` and `Hrj` denote the matrices containing the relevant entries of H_j and X_j with respect to $\mathcal{D}(H_j X_j)$.

We compare the performance of the D&C and DD method to that of a direct inversion (INV) method and a probing (PROBE) method [19]. For the INV method, $\mathcal{D}(A^{-1})$ is extracted from a direct inversion of A^{-1} by using `inv(A)`. For the probing method, we determine a probing matrix, $V \in \{0,1\}^{n\times s}$, with s probing vectors, based on the pattern of a sparsified matrix inverse (which is the same as the inverse matrix, but where entries smaller than 10^{-10} in absolute sense are dropped), and coloring the corresponding adjacency graph by a greedy algorithm. We then solve $AX = V$ by `X = A \ V`, and find $\mathcal{D}(A^{-1}) := \mathcal{D}(XV^T)$, see [19] for more details.

We emphasize that the numerical examples are chosen to highlight the proposed approaches without focusing on their optimal parameters for large and realistic problems (in a parallel environment). The latter is left for future work. Moreover, we note that our particular implementation of the DD approach is different from those of [9, 12] in the sense that we do not take a large p and just apply a simple procedure to determine $\mathcal{D}(S^{-1})$ and $\mathcal{D}(H_j X_j)$, where the use of iterative methods and sparse approximation techniques can be added to accelerate computations.

5.1. Shifted-Laplacian problem. In the first experiment, we consider the operator

$$-\Delta - \tau(1+i), \quad \tau \in \mathbb{R}, \quad i^2 = -1,$$

which is the Laplacian with a complex shift. This shifted-Laplacian (SL) operator is discretized by a standard finite difference scheme on a uniform 2-D grid with homogeneous Dirichlet boundary conditions and mesh size equal to $h = 1$. The resulting coefficient ma-

trix, A, is sparse and complex symmetric. We examine different choices of τ: $\tau = 10$ (giving a strongly diagonally dominant and negative-definite A), $\tau = 1$ (yielding a diagonally dominant and indefinite A), and $\tau = 0.1$ (giving a non-diagonally dominant and indefinite A).

5.1.1. Comparison of methods for various τ and n. We compare the performance of the INV, PROBE, D&C, and DD methods for various values of τ and n. The results of this experiment are presented in Table 5.1.

(a) $\tau = 10$.

$\sqrt{n}$	INV	PROBE	D&C	DD
25	.7	.1 (52)	.1	.1
50	12	.6 (53)	1.5	.6
75	66	1.7 (53)	5.5	2.1
100	n/a	3.7 (54)	16	5.9
150	n/a	11 (54)	64	23
200	n/a	30 (54)	238	64

(b) $\tau = 1$.

$\sqrt{n}$	INV	PROBE	D&C	DD
25	.7	.3 (165)	.1	.1
50	11	1.7 (170)	1.3	.6
75	64	4.9 (171)	5.3	2.1
100	n/a	9.9 (169)	14	5.8
150	n/a	73 (171)	64	23
200	n/a	n/a	173	62

(c) $\tau = 0.1$.

$\sqrt{n}$	INV	PROBE	D&C	DD
25	.6	n/a	.1	.1
50	11	n/a	1.2	.6
75	62	n/a	5.0	2.0
100	n/a	n/a	14	6.1
150	n/a	n/a	66	24
200	n/a	n/a	189	67

TABLE 5.1

Results of the SL experiment with different values of τ and n. The computational time (in seconds) is presented for the direct inversion (INV), probing (PROBE), divide-and-conquer (D&C), and domain decomposition (DD) methods. Here, 'n/a' means that the CPU time exceeds 1000 seconds, the method has memory problems, or the found solution is inaccurate. For the PROBE method, the number of required probing vectors is presented in brackets.

As can be observed in Table 5.1, the PROBE method outperforms the other methods when τ is large. However, when the matrix becomes indefinite and less diagonally dominant (by choosing a smaller τ), this method requires more probing vectors and becomes less efficient. The choice of τ hardly affects the performance of the other methods. Hence, the performance of the INV, D&C, and DD method does, generally, not depend directly on the definiteness of A and the exact values of its nonzeros, but mainly on the sparsity pattern of A. However, as we will see in the next experiments, a higher degree of indefiniteness will clearly impact performance of the D&C and DD method, when sparse approximations and iterative solvers are used.

Moreover, we notice that the INV method becomes impractical for $n \geq 100$. For a relatively large n and small τ, the D&C and DD methods are the methods of choice. Additionally, the DD method is faster than D&C method, which confirms the analysis seen in §4.1.

We note that the solutions provided by the D&C and DD approaches are accurate, as no iterative solvers or sparse approximations are used in the methods. Later on, we will show that adding these additional components may increase the efficiency of the approaches substantially, without losing much accuracy. Furthermore, no singular matrices are detected

in the D&C and DD methods, so the two methods are well-defined for the considered test cases.

5.1.2. Discussion of the D&C method. We examine the D&C method by varying the parameters in the method, and comparing several implementations of this method in the next experiments.

Previously, we have taken $\rho = \gamma n_x$ with $\gamma = 3$ and $m = \phi n_x$ with $\phi = \text{round}(n_y/2)$ in the D&C method. The parameter γ controls the number of recursion levels, while ϕ determines the sizes of the blocks C_1 and C_2 of the matrix C. In the next experiment, we examine whether the 'standard' parameters are optimal by varying them for a specific test case. The results can be found in Table 5.2.

ϕ	$\gamma = 3$	$\gamma = 5$	$\gamma = 7$
$\text{round}(n_y/2)$	14	16	18
$\text{round}(n_y/3)$	16	16	18
$\text{round}(n_y/4)$	18	18	20

TABLE 5.2

Results of the SL experiment with $n = 100$ and $\tau = 0.1$. The computational time (in seconds) is presented for the original D&C method with different values of $m = \phi n_x$ and $\rho = \gamma n_x$. Similar results are obtained for different values of n and τ.

From Table 5.2, it can be observed that increasing γ usually leads to worse results, as it is inefficient to apply direct inversion to relatively large matrices. Moreover, $\phi = \text{round}(n_y/2)$ seems to be the optimal choice, which means that C_1 and C_2 should have approximately equal dimensions. Decreasing ϕ yields an easier solve of $C_1 U_1 = E_1$, while solving $C_2 U_2 = E_2$ become the dominating part of the computations in the D&C method.

In the next experiment, we test two implementations of the D&C method The original D&C method as used above (with $\gamma = 3$ and $\phi = \text{round}(n_y/2)$) is denoted by D&C1, while D&C2 denotes the variant in which $\mathcal{P}(C_j^{-1})$ is computed on higher recursive levels instead of solving $C_j U_j = E_j$ explicitly (see §2.4). Both methods are also extended with a dropping procedure according to Eq. (2.14), which is applied to all recursion levels as long as $n_x \geq 25$. We take $\epsilon = 0$ (i.e., no dropping), $\epsilon = 10^{-6}$, and $\epsilon = 10^{-3}$ in the test cases. Table 5.3 gives the results of this experiment.

From Table 5.3, we can observe that the original D&C1 implementation is faster than the alternative D&C2 variant, when n is relatively large and no dropping is applied. On the other hand, by applying a dropping procedure, the D&C2 method becomes more favorable while the accuracy is still reasonable. This is in agreement with the arguments given in §2.4. We also notice that the dropping percentage depends on the choice of τ, and it can approach 100% for $\tau \geq 1$. When A is less favorable (i.e., A is less diagonally dominant and more strongly indefinite), fewer matrix entries can be dropped, so that a D&C approach with a dropping procedure becomes less effective.

Moreover, by comparing the results in Table 5.1 and 5.3, we notice that, for almost all n and τ, the DD method (without dropping) is still faster than all D&C implementations tested here. The major bottleneck is solving $U := C^{-1}E$, especially when n is large. An efficient implementation of the D&C method should address this problem, for example by adopting iterative methods and/or domain decomposition solvers.

5.1.3. Discussion of the DD method. To examine the DD method, we perform several experiments in which different implementations of the DD method are used. First, we test the original DD method for a different number of subdomains, p, see Table 5.4.

(a) $\tau = 10$.

	$\epsilon = 0$		$\epsilon = 10^{-6}$		$\epsilon = 10^{-3}$	
$\sqrt{n}$	D&C1	D&C2	D&C1	D&C2	D&C1	D&C2
50	1.3	1.1	1.4 (1E-9; 97%)	0.6 (7E-10; 93%)	1.4 (3E-3; 100%)	0.5 (2E-3; 99%)
100	14	18	15 (1E-9; 98%)	6.6 (7E-10; 96%)	14 (3E-3; 100%)	5.1 (2E-3; 99%)
200	238	601	169 (1E-9; 99%)	82 (7E-10; 98%)	165 (3E-3; 100%)	67 (2E-3; 100%)

(b) $\tau = 1$.

	$\epsilon = 0$		$\epsilon = 10^{-6}$		$\epsilon = 10^{-3}$	
$\sqrt{n}$	D&C1	D&C2	D&C1	D&C2	D&C1	D&C2
50	1.5	1.0	1.4 (3E-8; 66%)	1.0 (4E-7; 66%)	1.3 (7E-4; 96%)	0.7 (2E-3; 90%)
100	16	17	14 (3E-8; 91%)	9.9 (4E-7; 82%)	13 (7E-4; 99%)	6.4 (2E-3; 95%)
200	189	326	155 (3E-8; 98%)	105 (5E-7; 90%)	151 (7E-4; 100%)	77 (2E-3; 97%)

(c) $\tau = 0.1$.

	$\epsilon = 0$		$\epsilon = 10^{-6}$		$\epsilon = 10^{-3}$	
$\sqrt{n}$	D&C1	D&C2	D&C1	D&C2	D&C1	D&C2
50	1.2	1.1	1.8 (2E-7; 50%)	1.4 (3E-5; 50%)	1.3 (5E-3; 84%)	0.7 (9E-2; 87%)
100	15	17	20 (2E-7; 34%)	17 (3E-5; 35%)	14 (5E-3; 96%)	6.7 (9E-2; 93%)
200	173	304	202 (2E-7; 80%)	174 (4E-5; 81%)	159 (6E-3; 99%)	77 (1E-1; 97%)

TABLE 5.3

Results of the SL experiment for various τ, n, and implementations of the D&C method. The computational time (in seconds) is presented for the D&C1 and D&C2 methods. When $\epsilon > 0$, the relative error and the average dropping percentage at the lowest recursion level are presented in brackets. Denote the solution for $\epsilon > 0$ and $\epsilon = 0$ by $\mathcal{D}_\epsilon$ and $\mathcal{D}_0$, respectively. Then, the relative error is defined by $\frac{||\mathcal{D}_\epsilon - \mathcal{D}_0||_2}{||\mathcal{D}_0||_2}$. The dropping percentage is defined as the ratio of the number of dropped entries to the number of nonzeros in the specific matrix multiplied by 100%.

$\sqrt{n}$	$p = 4$	$p = 9$	$p = 16$
25	.1	.2	.2
50	.6	.6	1.9
100	5.8	5.8	15.3
200	62	48	139

TABLE 5.4

Results of the SL experiment with $\tau = 1$ and various values of n. The computational time (in seconds) is presented for the original DD method with different values of p. Similar results are obtained for different choices of τ.

In Table 5.4, it can be seen that, with the current implementation, the DD method performs best for $p = 4$ or $p = 9$. For a large n, the DD method with $p = 9$ seems to be the optimal choice, because $H_j := B_j^{-1} F_j$ can be computed with a relative ease (as each B_j is relatively small) while the effort to compute S^{-1} directly is relatively moderate (as the number of interface points is not too large).

Next, we perform an experiment in which a dropping procedure is added to the DD method, similarly to what is done in §5.1.2. When $n_j \geq 25^2$, entries of H_j that are smaller

than $\epsilon||H_j||$ in modulus are dropped. The results of this experiment are presented in Table 5.5 (cf. Table 5.3).

(a) $\tau = 10$.

$\sqrt{n}$	$\epsilon = 0$	$\epsilon = 10^{-6}$	$\epsilon = 10^{-3}$
50	.6	.6 (6E-11; 97%)	.6 (6E-5; 100%)
100	5.9	4.6 (8E-11; 99%)	4.5 (8E-5, 100%)
200	64	40 (9E-11; 99%)	39 (8E-5, 100%)

(b) $\tau = 1$.

$\sqrt{n}$	$\epsilon = 0$	$\epsilon = 10^{-6}$	$\epsilon = 10^{-3}$
50	.6	.6 (4E-11; 35%)	.6 (3E-5, 90%)
100	5.8	4.9 (8E-11; 80%)	4.4 (4E-5; 97%)
200	62	41 (1E-10; 95%)	39 (5E-5; 99%)

(c) $\tau = 0.1$.

$\sqrt{n}$	$\epsilon = 0$	$\epsilon = 10^{-6}$	$\epsilon = 10^{-3}$
50	.6	.6 (6E-11; 0%)	.6 (2E-4; 42%)
100	6.1	7.3 (4E-10; 1%)	4.7 (2E-4; 83%)
200	67	71 (6E-10; 57%)	43 (3E-4, 95%)

TABLE 5.5

Results of the SL experiment for different values of τ and n. The computational time (in seconds) is presented for the DD method with and without dropping. When $\epsilon > 0$, the relative error and the average dropping percentage at the lowest recursion level are presented in brackets.

Table 5.5 suggests that the addition of a dropping procedure is beneficial to the DD method, especially when τ and n are large. The dropping percentage grows for an increasing τ, and a more significant reduction of the computing time can be realized for the DD method with dropping as we increase n. We also observe that the accuracy of the solution scales with roughly ϵ^2, which is a favorable property of the method.

Subsequently, we perform an experiment in which we fix τ but vary p and ϵ in the DD method. We make a comparison with a standard LU factorization of A (by using the MATLAB command `lu(A)`). This is done in order to give an idea on how the DD method performs relative to other methods mentioned in Section 1 where a matrix factorization is required.

		$\epsilon = 10^{-6}$		$\epsilon = 10^{-3}$	
$\sqrt{n}$	LU	$p = 4$	$p = 9$	$p = 4$	$p = 9$
100	1.6	4.9 (8E-11; 80%)	5.6 (8E-11; 59%)	4.4 (4E-5; 97%)	5.0 (5E-5; 94%)
200	23	41 (1E-10; 95%)	35 (8E-11; 88%)	39 (5E-5; 99%)	32 (4E-5; 98%)
300	111	148 (1E-10; 92%)	129 (1E-10; 95%)	138 (5E-5; 100%)	117 (6E-5; 99%)

TABLE 5.6

Results of the SL experiment with $\tau = 1$ and a varying n. The computational time (in seconds) is presented for the LU factorization and the DD method, where various values for p and ϵ are taken. When $\epsilon > 0$, the relative error and the average dropping percentage at the lowest recursion level are presented in brackets.

Observe in Table 5.6 that for a large n, the DD method with $p = 9$ is faster than

the same method with $p = 4$, although the dropping percentage is usually smaller for a larger p. More interestingly, for an increasing n, the difference between the DD method and the LU factorization becomes smaller. A conclusion from this observation, is that the DD method seems to be at least competitive to methods based on matrix factorization, since those methods usually require a factorization as a first step and then additional computations to extract the diagonal $\mathcal{D}(A^{-1})$. This is at least as far as what an experiment with MATLAB allows to conclude. A full-fledged experiment with a production-level language, such as C or FORTRAN, is still needed and will be performed in the future.

We conclude the discussion on the DD method with an experiment in which we apply iterative solvers as an tool in the DD method, with or without the use of a dropping strategy. The Krylov-subspace method GMRES [15] is employed with a simple diagonal preconditioner to solve each sequence of linear systems, $B_j H_j = F_j$. The iterative process is terminated when the norm of the relative residual is smaller than a threshold, δ. The results of this experiment are provided in Table 5.7. In contrast to the previous tables, we only present the average number of GMRES iterations and the accuracy of the final solution, and omit the corresponding computational time. This is because the computational advantage of using iterative methods is small in MATLAB, and iterative methods do not provide a significant advantage over direct methods for 2-D problems.

(a) $\epsilon = 0$.

τ	$\delta = 10^{-12}$	$\delta = 10^{-6}$	$\delta = 10^{-3}$
10	15 (2E-16)	7 (2E-10)	3 (7E-7)
1	72 (3E-14)	35 (2E-9)	16 (5E-6)
0.1	115 (5E-13)	69 (8E-7)	30 (8E-4)

(b) $\epsilon = 10^{-6}$.

τ	$\delta = 10^{-12}$	$\delta = 10^{-6}$	$\delta = 10^{-3}$
10	15 (6E-11)	7 (1E-10)	3 (7E-7)
1	72 (4E-11)	35 (2E-9)	16 (5E-6)
0.1	115 (5E-13)	69 (8E-7)	30 (8E-4)

(c) $\epsilon = 10^{-3}$.

τ	$\delta = 10^{-12}$	$\delta = 10^{-6}$	$\delta = 10^{-3}$
10	15 (6E-5)	7 (6E-5)	3 (6E-5)
1	72 (3E-5)	35 (3E-5)	16 (3E-5)
0.1	115 (2E-4)	69 (2E-4)	30 (8E-4)

TABLE 5.7

Results of the SL experiment with $n = 50^2$ and a various τ. The DD method is examined, where the dropping tolerance, ϵ, and the GMRES termination tolerance, δ, are varied. The average number of GMRES iterations on the lowest recursion level is presented. The relative error of the computed $\mathcal{D}(A^{-1})$ is given in brackets. Similar results are obtained for different values of n.

A few interesting observations can be made from Table 5.7 A smaller τ leads to more GMRES iterations in the DD method. In an efficient implementation, this can be alleviated by choosing a more appropriate preconditioner. Moreover, a stringent GMRES termination tolerance, δ, is not always necessary; an accurate $\mathcal{D}(A^{-1})$ might be found without a high accuracy of the GMRES solver. In addition, the best choice of δ depends on the value of ϵ and the accuracy required for the final solution. For example, when we consider the test case with $\tau = 10$ and $\epsilon = 10^{-6}$, and we require that $\mathcal{D}(A^{-1})$ has a maximum relative error of 10^{-6}, then solving each $B_j H_j = F_j$ iteratively with a GMRES tolerance $\delta = 10^{-3}$ is sufficient according to Table 5.7.

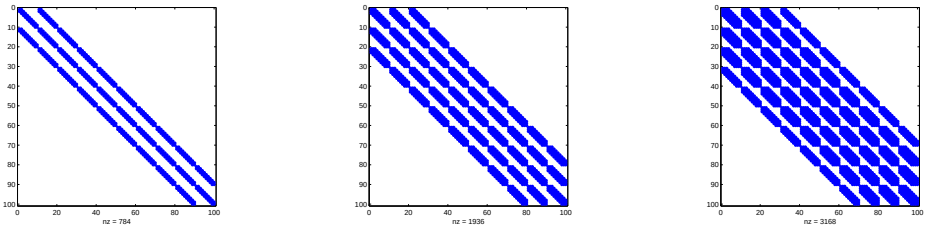
5.2. Uncertainty Quantification: covariance matrices. Uncertainty Quantification (UQ) in risk analysis has become a key issue in, e.g., geology, signal processing, and

portfolio management. In this setting, inverse covariance matrices hold a central role. A crucial question in data analysis for risk management is the degree of confidence that we can have in the quality of data. A highly useful measure of this quality is provided by the diagonal entries of the inverse covariance matrix, see also [3] and references therein.

Here, $A = [a_{jk}]$ denotes a model covariance matrix with entries computed via a decaying covariance function, see [17, Eq. (28)] and [13, Chapter 4.2]. This real and positive-definite function is based on a uniform 2-D Cartesian grid with $\sqrt{n}$ grid points in each direction (i.e., $n_x = n_y$), the Euclidean distance between grid points j and k (denoted by $d(j,k)$), the compact threshold $\alpha > 0$, and the function smoothness $\beta > 0$, i.e.,

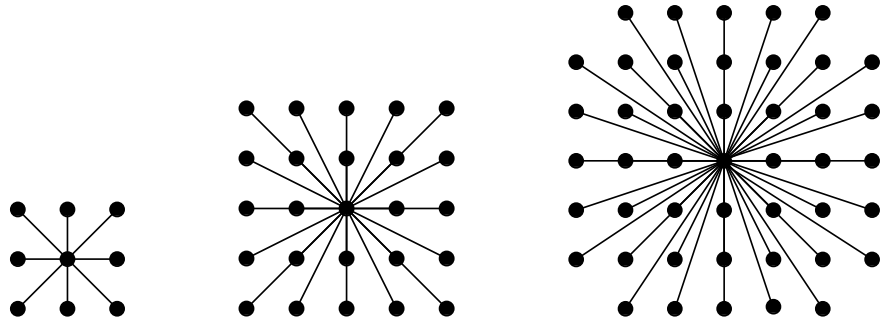
$$a_{j,k} := \begin{cases} \left(1 - \frac{d(j,k)}{\alpha}\right)^{\beta}, & \text{if } d(j,k) \le \alpha; \\ 0, & \text{otherwise.} \end{cases} \tag{5.1}$$

When α is not too large, the resulting matrix A is sparse, since the covariance between grid points is zero if their distance exceeds α. Moreover, α determines the exact sparsity pattern of A and the corresponding stencil, see Figures 5.3 and 5.4. The parameter β controls the diagonal dominance of A.



(a) $\alpha = 2$. (b) $\alpha = 3$. (c) $\alpha = 4$.

FIG. 5.3. *Typical sparsity patterns of A for $n = 10^2$ and various α derived from Eq. (5.1).*



(a) $\alpha = 2$ (9-point stencil). (b) $\alpha = 3$ (25-point stencil). (c) $\alpha = 4$ (45-point stencil).

FIG. 5.4. *Stencils associated with the sparsity patterns as depicted in Figure 5.3.*

Our aim is to compute $\mathcal{D}(A^{-1})$ for a matrix A with various choices of α and β. In [19], it has already been shown that the PROBE method is efficient, especially for a relatively small α and large β. The implementation of the D&C method as used in §5.1 can also be adopted

here for the UQ problem. The efficiency of this method will depend on α, as the rank of the low-rank matrix L and, hence, the number of columns of E increase as α grows. For example, we now have $q = n_x + 1, 2n_x + 2, 3n_x + 2$ for $\alpha = 2, 3, 4$, respectively. The implementation of the DD method also remains the same as before, except the choice of the interfaces in the domain decomposition setting. From Figure 5.4, it can be easily seen that if we choose $\theta_x = \theta_y = 1$ for $\alpha > 2$, the resulting matrix B is not block-diagonal, while that is essential in the DD method. To preserve disjoint subdomains in the domain decomposition setting and to guarantee an accurate solution, the thickness of the interfaces should be related to the choice of α. In our experiments, we therefore choose $\theta_x = \theta_y = \alpha - 1$. As the number of interface points grows with an increasing value of α, we expect that the DD method will become less efficient for a large α.

In the first experiment, we test the methods for a fixed n, and a varying α and β. The results are given in Table 5.8.

(a) $\alpha = 2$.

β	INV	PROBE	D&C	DD
6	18	2.0 (38)	2.9	1.6
4	16	3.2 (83)	2.5	1.6
2	17	21 (363)	2.4	1.6

(b) $\alpha = 3$.

β	INV	PROBE	D&C	DD
6	26	22 (363)	6.6	4.1
4	26	53 (627)	6.8	4.2
2	26	450 (2211)	6.9	4.2

(c) $\alpha = 4$.

β	INV	PROBE	D&C	DD
6	41	136 (1209)	13	8.1
4	41	399 (2232)	14	8.2
2	41	n/a	13	8.1

TABLE 5.8

Results of the UQ experiment with $n = 100^2$ and various values of α and β. The computational time (in seconds) is presented for the INV, PROBE, D&C, and DD method (without dropping). Furthermore, 'n/a' means that the CPU time exceeds 1000 seconds. Similar results can be found for different values of n.

Table 5.8 indicates that, with the current implementation, the PROBE method becomes less efficient for an increasing α and/or decreasing β, which is in line with the results presented in [19]. The performance of all methods suffers from a growing α, while the INV, D&C, and DD methods seem to be more-or-less insensitive to β. In addition, the fastest method in all test cases is the DD method.

In our next experiment, we focus on the DD method, where we vary the number of subdomains, p, and apply an extra dropping procedure to sparsify the matrix H. The results of this experiment are presented for the test case with $\alpha = 4$ and a varying β, see Table 5.9.

Table 5.9 shows that the DD method performs better with $p = 4$ than with $p = 9$. This can be explained by the fact that the computation of S^{-1} costs more for the case of $p = 9$. In addition, the dropping procedure is highly effective for $p = 4$, as many entries of H can be dropped without losing much accuracy of the solution.

5.3. Dynamic Mean-Field Theory: Green's function. Dynamic Mean-Field Theory (DMFT) has recently emerged as an important tool in the investigation of lattice models of highly correlated electrons in a quantum many-body system, see [4] and the references therein. For the inhomogeneous DMFT method, the computation of the diagonal of the

	$\epsilon = 0$		$\epsilon = 10^{-6}$		$\epsilon = 10^{-3}$	
β	$p = 4$	$p = 9$	$p = 4$	$p = 9$	$p = 4$	$p = 9$
6	8.1	29	4.0 (7E-10; 99%)	26 (1E-9; 98%)	3.9 (3E-5; 100%)	26 (4E-5; 100%)
4	8.2	29	4.1 (6E-10; 97%)	26 (9E-10; 94%)	3.9 (2E-5; 100%)	26 (2E-5; 100%)
2	8.1	31	4.9 (8E-11; 85%)	30 (9E-11; 68%)	4.0 (3E-5; 98%)	28 (4E-5; 95%)

TABLE 5.9

Results of the UQ experiment for $n = 100^2$ and $\alpha = 4$. The computational time (in seconds) is presented for the DD method with different values of p and ϵ. When $\epsilon > 0$, the relative error and the average dropping percentage at the lowest recursion level are presented in brackets. Similar results can be found for a different n and α.

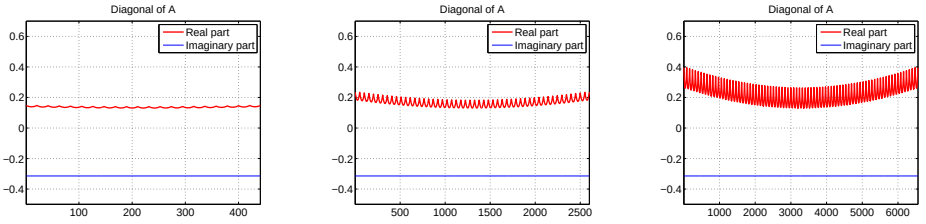
inverse of a large collection of sparse matrices is required. These inverses correspond to problems associated with the real and imaginary time Green's functions, respectively. Both problems are related by the chemical potential, $\mu \in \mathbb{R}$, and are solved for different values of the frequency, $\omega \in \mathbb{R}$. A typical form of the complex symmetric matrix A is

$$A := K + D, \quad K \in \{0, 1\}^{n \times n}, \ D \in \mathbb{C}^{n \times n}. \tag{5.2}$$

The matrix K is known as a 2-D hopping matrix, which has a similar sparsity pattern as a standard 2-D Laplacian matrix. The entries of the main diagonal of K are zero, and the entries of the four nonzero off-diagonals are equal to -1. The matrix D is a sum of various diagonal matrices, and varies for each ω. For the imaginary time Green's function, D is given such that A is often diagonally dominant and positive-definite. Consequently, A^{-1} shows an exponential decay property, so that many entries are small. In this case, the PROBE method is highly efficient, see [19]. In the specific case of the real time Green's function, the diagonal matrix D is given as follows:

$$D = (\mu + \omega) I_n - V - \Sigma, \quad V \in \mathbb{R}^{n \times n}, \ \Sigma \in \mathbb{C}^{n \times n}, \tag{5.3}$$

where V and Σ represent the trap potential and local self-energy, respectively. A detailed explanation and analysis can be found in [4]. The resulting matrix A is often indefinite and not diagonally dominant, see Figure 5.5 where a typical diagonal of A is presented for various values of n. Consequently, the PROBE method is not efficient for this case, as it requires too many probing vectors. Instead, we test the performance of the D&C and DD method for this problem. The results of the experiment with A based on Figure 5.5 can be found in Table 5.10.



(a) $n = 21^2$. (b) $n = 51^2$. (c) $n = 81^2$.

FIG. 5.5. *The real and imaginary parts of $\mathcal{D}(A)$ in the DMFT experiment for various values of n. The sparsity pattern and the off-diagonals of A are the same as in the SL problem.*

$\sqrt{n}$	INV	D&C	DD
21	.3	.1	.1
51	12	1.4	.7
81	88	7.7	2.8

TABLE 5.10

Results of the DMFT experiment for different values of n. The computational time (in seconds) is presented for the INV, D&C, and DD method (without dropping). The PROBE method is not suitable in this experiment, as it requires too many probing vectors.

The results as presented in Table 5.10 are similar to those of Table 5.1. For larger n, the cost of direct inversion grows fast, while the D&C and DD approaches still perform reasonably well. The fastest method in the experiment is the DD approach.

As in the experiments of the previous test problems, we can also combine the D&C and DD method with a dropping procedure, see Table 5.11 for some results.

ϵ	D&C	DD
0	7.7	2.8
10^{-6}	12 (8E-9; 26%)	3.4 (3E-11; 5%)
10^{-3}	8.5 (1E-4; 84%)	2.6 (1E-5; 67%)

TABLE 5.11

Results of the DMFT experiment for $n = 81^2$, where the D&C and DD method are examined for different values of ϵ. The computational time (in seconds) is presented. When $\epsilon > 0$, the relative error and the average dropping percentage at the lowest recursion level are presented in brackets.

Since the matrix A is less favorable than those seen in previous test problems, it can be observed in Table 5.11 that the dropping procedure is less advantageous for both the D&C and DD method, especially when a high accuracy of the solution is desired. Solving sequences of linear systems dominates the computations in both methods. Future work will examine how to mitigate this by incorporating more advanced and efficient direct or iterative solvers.

6. Conclusions and future work. Two methods have been presented to extract the diagonal entries of the inverse of a matrix. The divide-and-conquer method decouples the physical domain recursively, where a decoupled problem is solved and corrected at each recursion level. The domain decomposition method relies on local subdomain solves and a Schur complement correction, but it can also be extended such that it becomes a recursive procedure. Both methods rely on a simple algorithm, while an efficient implementation usually requires a careful book keeping of computations and more advanced computational tools. In the experiments, we show that a standard implementation of the divide-and-conquer and domain decomposition methods work well for general problems. These methods also prove to benefit from the use of sparse approximation and iterative methods when the considered matrix is (nearly) diagonally dominant. In the specific case when the matrix is strongly diagonally dominant, the proposed methods are good competitors to the probing method [19].

Both the divide-and-conquer and domain decomposition methods appear to be promising for a parallel computing environment. Future research should explore ideas to improve the different components of the proposed methods with a focus on their parallel implementation and their application to large and realistic 3-D problems.

Acknowledgments. The authors are indebted to Jie Chen for providing them with the covariance matrix generator and Pierre Carrier for his help with the DMFT code. They also

wish to thank the referees for their constructive comments which helped improve the quality of the paper.

REFERENCES

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

[2] P. R. Amestoy, I. S. Duff, Y. Robert, F.-H. Rouet, and B. Uçar. On computing inverse entries of a sparse matrix in an out-of-core environment. Technical Report TR/PA/10/59, CERFACS, Toulouse, France, 2010.

[3] C. Bekas, A. Curioni, and I. Fedulova. Low cost high performance uncertainty quantification. In *WHPCF '09: Proc. of the 2nd Workshop on High Performance Computational Finance*, pages 1–8, New York, NY, USA, 2009. ACM.

[4] J. K. Freericks. *Transport in Multilayered Nanostructures. The Dynamical Mean-Field Theory Approach*. Imperial College, London, UK, 2006.

[5] G. H. Golub and C. F. van Loan. *Matrix Computations*. Johns Hopkins Univ. Press, Baltimore, MD, 1996. Third edition.

[6] W. W. Hager. Updating the inverse of a matrix. *SIAM Rev.*, 31(2):221–239, 1989.

[7] H. V. Henderson and S. R. Searle. On deriving the inverse of a sum of matrices. *SIAM Rev.*, 23(1):53–60, 1981.

[8] G. Karypis and V. Kumar. METIS 4.0: Unstructured graph partitioning and sparse matrix ordering system. Technical report, Department of Computer Science, University of Minnesota, 1998. Available at `http://www.cs.umn.edu/~metis`.

[9] S. Li, S. Ahmed, G. Klimeck, and E. Darve. Computing entries of the inverse of a sparse matrix using the FIND algorithm. *J. Comput. Phys.*, 227(22):9408–9427, 2008.

[10] Z. Li, Y. Saad, and M. Sosonkina. pARMS: a parallel version of the algebraic recursive multilevel solver. *Num. Lin. Alg. Appl.*, 10:485–509, 2003.

[11] L. Lin, J. Lu, R. Car, and W. E. SelInv – an algorithm for selected inversion of a sparse symmetric matrix. *ACM Trans. Math. Software*, 2010 (to appear).

[12] L. Lin, J. Lu, L. Ying, R. Car, and W. E. Fast algorithm for extracting the diagonal of the inverse matrix with application to the electronic structure analysis of metallic systems. *Commun. Math. Sci.*, 7:755–777, 2009.

[13] C.E. Rasmussen and C.K.I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, Cambridge, MA, 2006.

[14] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelphia, PA, USA, 2003. Second edition.

[15] Y. Saad and M. H. Schultz. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 7(3):856–869, 1986.

[16] Y. Saad and B. Suchomel. ARMS: An algebraic recursive multilevel solver for general sparse linear systems. *Num. Lin. Alg. Appl.*, 9, 2002.

[17] K.P. Schmitt, M. Anitescu, and D. Negrut. Efficient sampling for spatial uncertainty quantification in multibody system dynamics applications. *Int. J. Numer. Meth. Eng.*, 80(5):537–564, 2009.

[18] R. B. Sidje and Y. Saad. Rational approximation to the Fermi-Dirac function with applications in density functional theory. Technical Report UMSI 2008/279, Minnesota Supercomputing Institute, University of Minnesota, 2008.

[19] J. M. Tang and Y. Saad. A probing method for computing the diagonal of the matrix inverse. Technical Report UMSI 2010/42, Minnesota Supercomputing Institute, University of Minnesota, 2010.