

AN ALGEBRAIC SPARSIFIED NESTED DISSECTION ALGORITHM USING LOW-RANK APPROXIMATIONS

LÉOPOLD CAMBIER*, CHAO CHEN†, ERIK G. BOMAN ‡, SIVASANKARAN
RAJAMANICKAM§, RAYMOND S. TUMINARO¶, AND ERIC DARVE||

Abstract. We propose a new algorithm for the fast solution of large, sparse, symmetric positive-definite linear systems, spaND — sparsified Nested Dissection. It is based on nested dissection, sparsification and low-rank compression. After eliminating all interiors at a given level of the elimination tree, the algorithm sparsifies all separators corresponding to the interiors. This operation reduces the size of the separators by eliminating some degrees of freedom but without introducing any fill-in. This is done at the expense of a small and controllable approximation error. The result is an approximate factorization that can be used as an efficient preconditioner. We then perform several numerical experiments to evaluate this algorithm. We demonstrate that a version using orthogonal factorization and block-diagonal scaling takes fewer CG iterations to converge than previous similar algorithms on various kinds of problems. Furthermore, this algorithm is provably guaranteed to never break down and the matrix stays symmetric positive-definite throughout the process. We evaluate the algorithm on some large problems and show it exhibits near-linear scaling. The factorization time is roughly $\mathcal{O}(N)$ and the number of iterations grows slowly with N .

Key words. sparse linear solver, hierarchical matrix, nested dissection, preconditioner, low-rank

AMS subject classifications. 65F05, 65F08, 65Y20

1. Introduction. We are interested in solving large symmetric, positive-definite (SPD) sparse linear systems

$$(1.1) \quad Ax = b, \quad A \in \mathbb{R}^{N \times N}.$$

In particular, we focus on linear systems with similar properties as those arising from the discretization of elliptic partial differential equations, using finite difference or finite elements for instance. Solving such systems is a crucial part of many scientific simulations.

Algorithms for solving Eq. 1.1 are traditionally divided into three categories. On one hand are direct methods. The naive Cholesky ($A = LL^T$) factorization can lead to a factorization cost of $\mathcal{O}(N^3)$ (with $\mathcal{O}(N^2)$ memory use) due to fill-in in the factor L . When the matrix A comes from the discretization of PDE's in 2D or 3D space, one usually uses the Nested Dissection [40] ordering to reduce fill-in. By doing so, the time complexity is typically reduced to $\mathcal{O}(N^{3/2})$ (in 2D) and $\mathcal{O}(N^2)$ (in 3D), with the memory complexity reduced to $\mathcal{O}(N \log N)$ (in 2D) and $\mathcal{O}(N^{4/3})$ (in 3D) [25, 40]. This is what most state-of-the-art direct solvers are built upon [17, 3, 35]. Those algorithms work very well for most moderate-size problems. However, the $\mathcal{O}(N^2)$ complexity in 3D makes them intractable on large scale problems.

An alternative is to use iterative algorithms like Krylov methods or multigrid. Multigrid [22, 9, 32] (and its algebraic version, [10, 50]) works very well on fairly regular elliptic PDEs, usually with a near-constant iteration count and $\mathcal{O}(N)$ memory

*Institute for Computational & Mathematical Engineering, Stanford University, lcambier@stanford.edu

†Institute for Computational Engineering and Sciences, The University of Texas at Austin, chenchao.nk@gmail.com

‡Center for Computing Research, Sandia National Laboratories, egboman@sandia.gov

§Center for Computing Research, Sandia National Laboratories, srajama@sandia.gov

¶Center for Computing Research, Sandia National Laboratories, rstumin@sandia.gov

||Department of Mechanical Engineering, Stanford University, darve@stanford.edu

use regardless of the problem size. However, it can solve only a fairly limited range of problems and its iteration count can start growing when the problem becomes ill-conditioned. Krylov methods, such as CG [36, 53], MINRES [42] or GMRES [47] can be applied to a very wide range of problems, necessitating only sparse matrix-vector products. However, to converge at all, one needs to always couple them with an efficient preconditioner. This is typically a very problem dependent task.

One way, however, to build preconditioners is using incomplete factorizations and low-rank approximations. Incomplete factorization algorithms are built on top of a classical matrix factorization algorithm. Incomplete LU (ILU) for instance starts with a classical LU algorithm and ignores some of the fill-in based on thresholding and on an artificially prescribed maximum number of non-zeros in every row & column [45]. Block versions [48] are sometimes used because of better robustness (with possible pivoting) and practical properties (cache-friendly algorithm, use of BLAS, etc.). Once an incomplete LU factorization has been computed, it can be used as a preconditioner for a CG or GMRES algorithm for instance.

Matrices arising from PDE discretization also typically have low-rank off-diagonal blocks [7, 6, 13]. More precisely, the fill-in arising when factoring the matrix typically has small numerical rank, with weak dependence on N . This is closely related to the existence of a smooth Green’s function for the underlying PDE and to the Fast Multipole Method [5, 29, 24]. Matrices built using this property are broadly called Hierarchical (\mathcal{H}) matrices [31]. Many formats exist, depending on when and how off-diagonal blocks are compressed into low-rank format. The Hierarchical Off-Diagonal Low Rank (HODLR) [1] format compresses all off-diagonal blocks. If the off-diagonal are compressed using a nested basis, we obtain Hierarchically Semi-Separable (HSS) matrices [14, 12, 15, 57]. Finally, the broader category of \mathcal{H}^2 matrices also uses nested basis but only compresses well-separated (i.e., far-field) interactions ([33, 34, 60], [43] with LoRaSp and [51] with the “Compress and Eliminate” solver). All of those representations lead to a data-sparse representation of the matrix with tunable accuracy (by making the low-rank approximations more or less accurate) and fast inverse computations. This can then be used to construct preconditioners. These constructions, while asymptotically efficient, sometimes have fairly large constants.

Attempts to improve the practical performance rely on exploiting sparsity as well as the low rank structure. Most approaches up to date have focused on incorporating fast (i.e., \mathcal{H} -) algebra into the classical Nested Dissection algorithm [28] in order to decrease the cost of working with large fronts. Other works have taken the similar approach of incorporating rank structured matrices into a multifrontal factorization in order to compress the large dense frontal matrices. HSS is often used to compress the large frontal matrices [56, 49, 54, 55, 26]. The last one was incorporated into the Strumpack package. [2] uses Block Low-Rank approximation to compress the frontal matrices in the MUMPS solver [3]. Finally, [21] studies the use of \mathcal{H} -matrices using HODLR in the PaStiX solver [35].

The Hierarchical Interpolative Factorization (HIF) [37] proposes a different approach. Instead of storing the full dense fronts in some low-rank format, it uses low-rank approximation to directly sparsify (i.e., eliminate part of) the Nested Dissection separators without introducing any fill-in. As a result, the algorithm never deals with large edges (in low-rank format or not) but instead constantly reduces the size of all edges and separators. This is the approach we take.

We finally mention some recent work by J. Xia & Z. Xin [59] and J. Feliu-Fabà et al. [23] where, in both cases, a scale-then-compress approach is taken. Our algorithm shares similarities with those, as we also scale the matrix block using the

Cholesky factorization of the pivot. As we will see, this significantly improves the preconditioner’s accuracy.

1.1. Contribution. Our approach is based on the idea of HIF described in [37]. However, there are several differences, improvements and novel capabilities:

- Our algorithm is completely general and can be applied to any (SPD) matrix. The only required input is the sparse matrix itself. If geometry information is available, it can be used to improve the quality of the ordering and clustering.
- We incorporate an additional diagonal block scaling step in the algorithm, greatly improving the accuracy of the preconditioner for only a small additional cost;
- We use an orthogonal (instead of interpolative) transformation, improving stability and guaranteeing that the preconditioner stays SPD when A is SPD;
- We test the algorithm on more and larger test problems.

In a nutshell, our algorithm is based on a couple of key ideas. First, we start with a nested dissection (ND) ordering. Then following the idea introduced in [37], after each elimination step, we sparsify the interfaces between just-eliminated interiors, effectively reducing the size of *all* ND separators. This is done using low-rank approximation, allowing to sparsify the separators without introducing any fill-in. We then merge clusters and proceed to the next level.

A natural consequence of the above algorithm is that, if the compression fails to reduce the size of the separators, the algorithm reverts to a (slower, but still relatively efficient) Nested Dissection algorithm.

1.2. Contrast with fast algebra based algorithms. We emphasize that the HIF approach [37] and ours are different from the classical way of accelerating sparse direct solvers. Consider for instance the top separator of a Nested Dissection elimination. At the end of the elimination, the corresponding (very large) block in the matrix is typically dense. MUMPS [2] and PaStiX [21] for instance use fast \mathcal{H} -algebra to compress this block. This allows for fast factorization, inversion, etc.

As indicated above, we take a different approach. Instead of storing large blocks (corresponding to large separators) in low-rank format (typically using \mathcal{H} -matrices), we eliminate part of the separators *right from the beginning*, effectively reducing their size. We do so without introducing any fill-in, but at the expense of an approximate factorization. As a result, the top separator remains dense but is much smaller than at the beginning.

Both approaches use some sort of hierarchical clustering of the unknowns. The difference lies in the order of operations. In the first category (large blocks using fast \mathcal{H} -algebra) elimination is delayed until the end. The result are large and dense but hierarchically low-rank fronts. In our approach (like in [37]), fronts are kept small throughout the factorization by eliminating unknowns related to low-rank interactions as soon as possible.

1.3. Organization of the paper. This paper is organized in three sections. First, [section 2](#) introduces and motivates the algorithm, starting at a high level and later introducing the details. Then, [section 3](#) proves the stability of the scheme, discusses the choice of the low-rank approximation and provides a complexity analysis. Finally, [section 4](#) shows numerical experiments on medium and large scale matrices.

2. Sparsified Nested Dissection. This section describes the algorithm in detail. We start by discussing Nested Dissection and some of its characteristics. Then, building upon it, we introduce our algorithm, and then detail all the various parts.

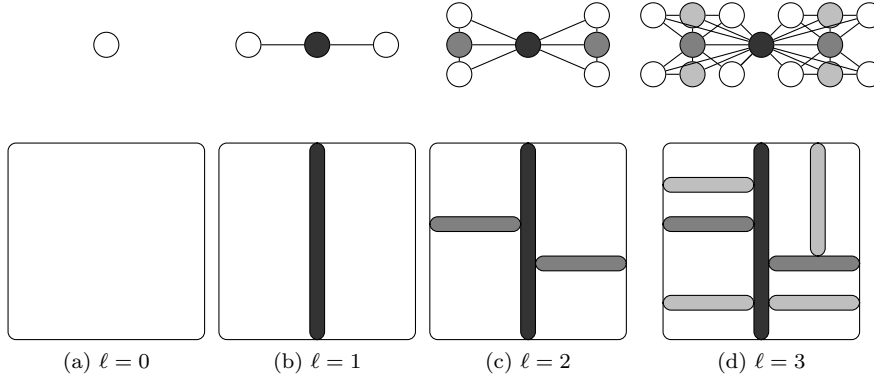


Fig. 2.1: Classical Nested Dissection ordering.

2.1. Classical Nested Dissection Ordering. Nested Dissection (ND) [25] is an ordering strategy to factor sparse matrices. Consider a sparse symmetric matrix $A \in \mathbb{R}^{N \times N}$ and its graph $G_A = (V, E)$ defined as $V = \{1, \dots, N\}$ and $E = \{(i, j) : A_{ij} \neq 0\}$. Notice that since A is symmetric, the graph is undirected. The basic building block of ND is the computation of *vertex-separators*. Starting with the full graph, one finds a cluster of vertices, a vertex-separator, separating the graph into two disconnected clusters (a cluster is a subset of V).

Fig. 2.1b gives an example of such a separator. The idea is then applied recursively as indicated in Fig. 2.1c and Fig. 2.1d. That is, disconnected clusters are further sub-divided using separators. This recursive process is repeated until cluster sizes are small enough to be factored using some direct dense method. A matrix factorization can begin by eliminating unknowns in all disconnected clusters defined by the last recursive level of the nested dissection process. Thus, the only non-eliminated unknowns correspond to degrees-of-freedom (dofs) associated with all of the separators. We then proceed to eliminate all dofs associated with the last set of separators (e.g., those defined in the $\ell = 3$ level of Fig. 2.1). Once these have been eliminated, we proceed by eliminating the second to last set of separator unknowns (e.g., those defined on $\ell = 2$ in Fig. 2.1). The process continues eliminating unknowns associated with successively lower levels. This process can be viewed as an elimination tree, illustrated in Fig. 2.2.

The elimination tree indicates dependencies between operations. Each node is a cluster in the graph of A (separator or leaf-interior), and a cluster can only be eliminated once all its descendants have been eliminated. The clusters are then eliminated from bottom to top. This follows from the fact that eliminating a parent before a child would create edges between clusters previously separated, breaking the purpose of the ordering. ND is an ordering that limits fill-in: by eliminating clusters from bottom to top, one never creates edges (i.e., fill-in) between clusters previously separated.

The elimination procedure can also be represented in matrix form. Denote the total number of levels by L (where the leaves correspond to $\ell = L$ and the root to $\ell = 1$). Define $A^{(L)}$ as the entire matrix and let $A^{(\ell)}$ (for $\ell < L$) be the Schur complement operator (trailing matrix) obtained by eliminating levels $\ell+1, \dots, L$. The

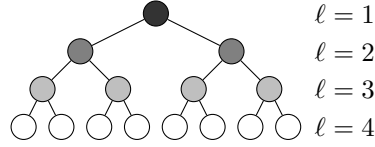


Fig. 2.2: The elimination tree associated to the ND ordering on Fig. 2.1

matrix obtained after eliminating the $\ell + 1$ level can be written in a block-arrowhead form

$$A^{(\ell)} = \begin{bmatrix} A_{11}^{(\ell)} & & & A_{1q}^{(\ell)} \\ & \ddots & & \vdots \\ & & A_{mm}^{(\ell)} & A_{mq}^{(\ell)} \\ A_{q1}^{(\ell)} & \dots & A_{qm}^{(\ell)} & A_{qq}^{(\ell)} \end{bmatrix}$$

where $m = 2^{\ell-1}$ and $q = m + 1$. Here, $A_{qq}^{(\ell)}$ refers to the matrix associated with all separators at levels $1, \dots, \ell - 1$. The $A_{ii}^{(\ell)}$ (for $i \leq m$) are the matrices associated with non-eliminated unknowns within the i^{th} disconnected separators on the ℓ^{th} level. The Schur complement can now be written as

$$A^{(\ell-1)} = A_{qq}^{(\ell)} - \sum_{i=1}^m A_{qi}^{(\ell)} \left(A_{ii}^{(\ell)} \right)^{-1} A_{iq}^{(\ell)}.$$

This new matrix can then be interpreted as another block-arrowhead matrix associated with level $\ell - 1$ and so the elimination procedure can be repeated.

While limited, the fill-in is still significant. For instance, once all descendants of the top separators have been eliminated, the top separator is typically completely filled (dense). For problems arising from the discretization of PDE's in 3D with $\mathcal{O}(N) = \mathcal{O}(n^3)$ degrees of freedom (dofs), the top separator typically has size $\mathcal{O}(N^{2/3}) = \mathcal{O}(n^2)$. For instance in a regular $n \times n \times n$ cube with $N = n^3$ dofs and a 7-points stencil (or any other stencil with only "local" connections), the top separator is a plane (see Fig. 2.3) of size $n \times n = n^2 = N^{2/3}$. Hence, its factorization will cost $\mathcal{O}(N^{2/3 \times 3}) = \mathcal{O}(N^2)$, leading to quadratic or near-quadratic algorithms. While this is only formally valid on regular cubic-shaped graphs, the issue extends beyond those problems [40]: the separators in 3D graphs are typically very large, leading to large Schur complements and an expensive factorization, with complexity well above $\mathcal{O}(N)$.

Our algorithm addresses this specific concern by continually decreasing the size of all separators to keep fill-in to a minimum. It does so using low-rank approximations, and the factorization is then only approximate. In most cases under consideration, the separator size is typically decreased to $\mathcal{O}(n) = \mathcal{O}(N^{1/3})$ so that its factorization costs $\mathcal{O}(N)$.

2.2. Sparsified Nested Dissection. As noted, the sub-blocks created by the repeated Schur complement process become denser. To further limit fill-in, we introduce a sparsification algorithm that is invoked after all eliminations associated with a particular level have been performed. To motivate the sparsification, let us first

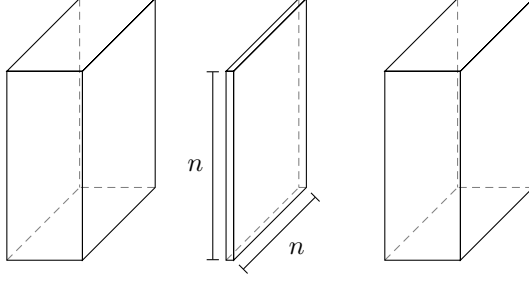


Fig. 2.3: Classical ND in 3D with $N = n^3$ nodes: the top separator is of size $\mathcal{O}(n^2)$. Once the left and right clusters have been eliminated the top separator becomes completely dense, making its elimination alone cost $\mathcal{O}((n^2)^3) = \mathcal{O}(N^2)$.

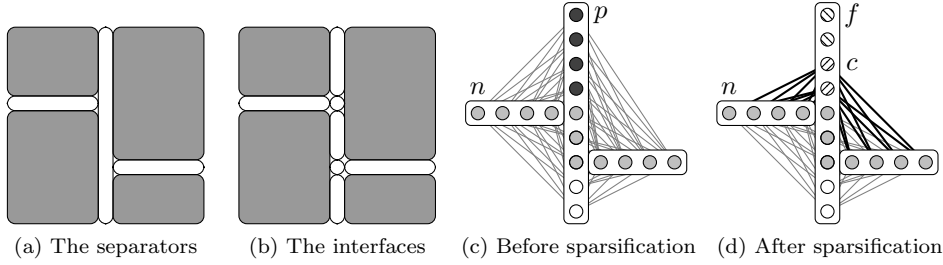


Fig. 2.4: Separator sparsification process. The first picture depicts the usual ND separator. The grey boxes have been eliminated. The second picture shows the different interfaces to be sparsified. The third and fourth picture focus on a given interface defined by p and connected to n . On the fourth picture, we have transformed p into f and c through a change of basis (not shown here) and up to a small $\mathcal{O}(\varepsilon)$ error. f is now disconnected from n and can be eliminated, without introducing any fill-in. Dark edges are edges updated by the sparsification.

consider a very simple case. Suppose we are about to eliminate level ℓ and that there exist a subset of the j^{th} separator such that the corresponding rows in $A_{jq}^{(\ell)}$ are relatively small. An inexact or incomplete factorization can be defined by simply ignoring these small rows. This effectively decouples those unknowns from the rest of the system so that they can be eliminated right away, without causing an increase in the number of non-zeros in the next recursive Schur complement. We can think of this as decreasing the size of the j^{th} separator. While $A_{jq}^{(\ell)}$ will not generally have small rows, we instead seek a transformation that produces the desired small rows without altering the nonzero structure of the matrix. Further, this transformation will not only be applied to the off-diagonal blocks in the level ℓ arrowhead matrix, but to all the non-eliminated degrees of freedom (dofs) (including those in $A_{qq}^{(\ell)}$). This means we will effectively decrease the size of all remaining separators at levels $1, \dots, \ell$.

To understand the transformation, it is best to switch to a different block structure

for the matrix. Specifically, consider the matrix

$$A^{(\ell)} = \begin{bmatrix} \hat{A}_{11}^{(\ell)} & \dots & \hat{A}_{1M}^{(\ell)} \\ \vdots & \ddots & \vdots \\ \hat{A}_{M1}^{(\ell)} & \dots & \hat{A}_{MM}^{(\ell)} \end{bmatrix}$$

This matrix is equivalent to $A^{(\ell)}$, the use of the *hat* accent symbol is only to emphasize the different block structure. Each $\hat{A}_{ii}^{(\ell)}$ corresponds to the sub-matrix associated with an *interface* (instead of a separator). Specifically, an interface is defined as a *subset* of a separator for which the left and right neighbors correspond to a given pair of separators at level ℓ . These M interfaces are subsets of all the separators associated with non-eliminated levels (i.e., levels k with $k \leq \ell$). Notice that many of the $\hat{A}_{ij}^{(\ell)}$ are zero as only neighboring interfaces are coupled. Fig. 2.4 shows the distinction between ND separators (Fig. 2.4a) and interfaces (Fig. 2.4b). The top-level (root) separator has been cut into 5 pieces, each associated to a pair of left and right neighbors.

Fig. 2.4c and Fig. 2.4d illustrates a typical situation and the effect of the sparsification. Let p be a subset of a ND separator (in dark grey) at the interface between two interiors and n be all its neighbors (in light grey). The remaining nodes are disconnected from p and can be ignored for the purpose of this discussion. In this situation, the leaf-level interiors (dark grey clusters on Fig. 2.4a and Fig. 2.4b) have been eliminated and only higher-levels separators are left. The greyed edges represent connections between degrees of freedom. Notice that edges never cross separators.

Let \hat{A}_{pn} denote all the edges from p to n . Assuming $\hat{A}_{pp} = I$ (this is not a restriction, see subsection 2.5), we can then consider this sub-matrix of $A^{(\ell)}$

$$\begin{bmatrix} I & \hat{A}_{pn} \\ \hat{A}_{np} & \hat{A}_{nn} \end{bmatrix}$$

Then, compute a low-rank approximation

$$\hat{A}_{pn} = \underbrace{[Q_{pf} \quad Q_{pc}]}_Q \begin{bmatrix} W_{fn} \\ W_{cn} \end{bmatrix} \text{ with } \|W_{fn}\| = \mathcal{O}(\varepsilon)$$

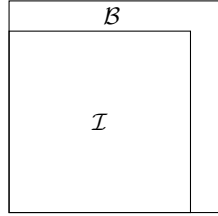
and verify that

$$\begin{bmatrix} Q^\top & \\ & I \end{bmatrix} \begin{bmatrix} I_{pp} & \hat{A}_{pn} \\ \hat{A}_{np} & \hat{A}_{nn} \end{bmatrix} \begin{bmatrix} Q & \\ & I \end{bmatrix} = \begin{bmatrix} I_{ff} & & \mathcal{O}(\varepsilon) \\ & I_{cc} & W_{cn} \\ \mathcal{O}(\varepsilon) & W_{cn}^\top & \hat{A}_{nn} \end{bmatrix}$$

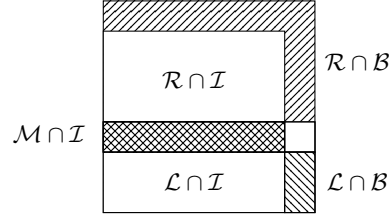
The matrix Q is a change of variables, transforming p into f (“fine”) and c (“coarse” — following AMG’s terminology, [50]). If we then ignore the $\mathcal{O}(\varepsilon)$ edges, we have effectively decoupled the f variables from the rest, i.e., we have eliminated f . Notice that this didn’t create any fill-in: \hat{A}_{nn} is unchanged. As a result, we effectively *decreased the size of the separator*, without altering the nested dissection ordering. In the following, we will drop the hat notation, as it should be clear from the context whether we are referring to separators or interfaces.

The algorithm then alternates between classical “interiors” elimination (using a block Cholesky factorization) and “interfaces” sparsification as explained above. Algorithm 2.1 presents a high-level version of the algorithm. We name the algorithm spaND, referring to “sparsified Nested Dissection”.

The subsequent sections explain in detail the ordering & clustering (i.e., how we define the “interfaces”), the elimination and sparsification.

Algorithm 2.1 High-level description of the spaND algorithm**Require:** Sparse matrix A , SPD, Maximum level L Compute a ND ordering for A , infer interiors, separators and interfaces (see [subsection 2.3](#))**for all** $\ell = L, \dots, 1$ **do** **for all** \mathcal{I} interior **do** Eliminate \mathcal{I} (see [subsection 2.4](#)) **end for** **for all** \mathcal{B} interface between interiors **do** Sparsify \mathcal{B} (see [subsection 2.5](#) and [subsection 2.6](#)) **end for****end for**

(a) An interior and its boundary before the modified ND step.



(b) The resulting left and right interiors and their boundaries.

Fig. 2.5: The clustering & ordering building block. On the left, an initial interior \mathcal{I} and its boundary \mathcal{B} . We then compute a vertex separator separating $\mathcal{I} \cup \mathcal{B}$ into left \mathcal{L} , right \mathcal{R} and separator \mathcal{M} . On the right, the resulting separated interiors and their boundaries, as well as the actual ND separator.

2.3. Ordering and Clustering. In addition to ordering, an appropriate clustering of the dofs has to be performed to define the various interfaces between interiors. That is, a simple ND ordering, by itself, does not give any indication about what the interfaces between different interiors are. To see this, consider [Fig. 2.1](#) (bottom row). This figure illustrates a classical ND ordering process. At every step, interiors are further separated by computing vertex separators. However, there is no clear way to define interfaces between interiors as shown on [Fig. 2.4](#) for instance. This cannot readily be calculated or even properly defined with a “usual” ND ordering.

To solve this issue, we have to keep track of the boundary of each interior during the ordering process. We do so by modifying the usual ND algorithm. In the classical algorithm, a set of vertices is separated by a vertex-separator, and the algorithm then recurses on the “left” and “right” clusters (interiors). We modify this by separating an interior *and* its boundary using vertex separators. This lets keep track of the interfaces. [Fig. 2.5](#) shows the high-level idea. For every interior \mathcal{I} we keep track of its boundary \mathcal{B} and we then separate their union $\mathcal{I} \cup \mathcal{B}$.

In practice, each node in the graph keeps track of its “left” and “right” neighboring separators, in addition to keeping track of the separator it belongs to. We encode this by associating to each vertex v a 3-tuple (S, L, R) . S is the usual ND separator (ℓ, k) where ℓ is its level and $1 \leq k \leq 2^{\ell-1}$. L and R are the ND separators of v ’s left and

right neighbors, respectively. [Algorithm 2.2](#) formalizes this idea. Notice how the only building block is a vertex-separator routine, as available in Metis [\[38\]](#).

Algorithm 2.2 Ordering and clustering algorithm. The algorithm is similar to a classical ND algorithm, except that it keeps track of the interfaces between interiors/separators, and recursively dissects interiors and their interfaces. ND separators are encoded as (ℓ, k) where ℓ is the level and $1 \leq k \leq 2^{\ell-1}$.

Require: V , vertices, E , edges, L levels

% Initialize the top separator (everyone), left and right neighbors (undefined)

$C[v] = (S : (1, 1), L : \text{none}, R : \text{none})$ for all $v \in V$

for all $\ell = 1, \dots, L - 1$ **do**

for all $k = 1, \dots, 2^{\ell-1}$ **do**

% Find interior to separate \mathcal{I} and its boundary \mathcal{B}

$\mathcal{I} = \{v \in V : C[v]_S = (\ell, k)\}$

$\mathcal{B} = \{v \in V : C[v]_L = (\ell, k) \text{ or } C[v]_R = (\ell, k)\}$

% Find vertex separator \mathcal{M} , left and right interiors \mathcal{L} and \mathcal{R}

$(\mathcal{L}, \mathcal{M}, \mathcal{R}) = \text{vertex-separator}(\mathcal{I} \cup \mathcal{B})$

% Update separator, left and right interiors

$C[v]_S = (\ell, k)$ for all $v \in \mathcal{M} \setminus \mathcal{B}$

$C[v]_S = (\ell + 1, 2k - 1)$ for all $v \in \mathcal{L} \setminus \mathcal{B}$

$C[v]_S = (\ell + 1, 2k)$ for all $v \in \mathcal{R} \setminus \mathcal{B}$

% Update neighbors of separator

for all $v \in \mathcal{M} \cap \mathcal{I}$ **do**

$C[v]_L = (\ell + 1, 2k - 1)$

$C[v]_R = (\ell + 1, 2k)$

end for

% Update neighbors of left and right boundaries

for all $v \in \mathcal{L} \cap \mathcal{B}$ **do**

if $C[v]_L = (\ell, k)$ **then**

$C[v]_L = (\ell + 1, 2k - 1)$

else

$C[v]_R = (\ell + 1, 2k - 1)$

end if

end for

for all $v \in \mathcal{R} \cap \mathcal{B}$ **do**

if $C[v]_L = (\ell, k)$ **then**

$C[v]_L = (\ell + 1, 2k)$

else

$C[v]_R = (\ell + 1, 2k)$

end if

end for

end for

end for

return C

[Algorithm 2.2](#) returns C that gives for each vertex v in the graph its ND separator, $C[v]_S$, as well as a tuple $(C[v]_L, C[v]_R)$ indicating its left and right neighboring interiors. We then cluster together vertices v with the same $C[v]$. This algorithm is naturally recursive and defines, for each separator, a tree of clusters.

[Fig. 2.6](#) illustrates the effect of [Algorithm 2.2](#). On the top row, we illustrate the

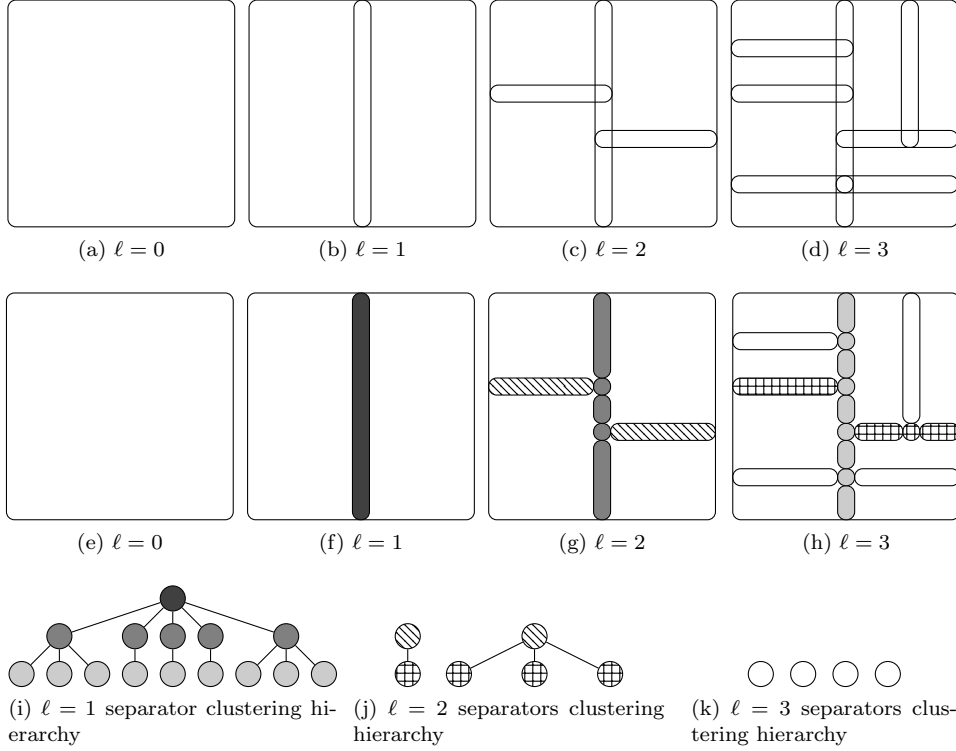


Fig. 2.6: A modified ND ordering & clustering ([Algorithm 2.2](#)). The top row indicates the separators computed at each step by separating interiors & boundaries. The middle row illustrates the clustering of dofs in each separator creating the interfaces between interiors. The bottom row shows the clusters hierarchy within each ND separator.

separators at every step (ℓ) of the algorithm. The important distinctions with [Fig. 2.1](#) is that the computed vertex-separators overlap with the boundaries to keep track of interfaces, and each separator is further divided into clusters. On the middle row, we illustrate the actual clusters at each level and how the ND separators are broken into pieces. Separators at each level are depicted in a different color. Each separator is associated a hierarchy of clusters. The bottom row shows such a hierarchy within each separator and how those have to be merged when going from a lower to higher level.

In practice (see [section 4](#)), we implement this algorithm in two ways. If geometry information is available, the **vertex-separator** subroutine of [Algorithm 2.2](#) is implemented using a recursive coordinate bisection. The subgraph is partitioned into two equal parts along the dimension with the largest span, and the nodes in the first part adjacent to the second form the middle separator. If no geometry information is available, we use the **nodeND** routine of Metis [\[38\]](#).

2.4. Separators Elimination using Block Cholesky. Now that the matrix has been ordered and that dofs have been grouped into clusters defining various in-

terfaces, the next step is to eliminate the interiors or separators at a given level ℓ of the ND tree, as in a usual direct solver (see [Algorithm 2.1](#)). This section describes this elimination step, which is simply a standard block Cholesky reinterpreted with our notation. Consider A into the “block-arrowhead” form following the ND ordering

$$A = \begin{bmatrix} A_{ss} & & A_{sn} \\ & A_{ww} & A_{wn} \\ A_{ns} & A_{nw} & A_{nn} \end{bmatrix}$$

We indicate the separator or interior of interest by s , its neighbors by n and all disconnected nodes by w . By symmetry, $A_{ab} = A_{ba}^\top$.

Let $L_s L_s^\top = A_{ss}$ the Cholesky factorization of A_{ss} . Then, define

$$E_s = \begin{bmatrix} L_s^{-1} & & \\ & I & \\ -A_{ns} A_{ss}^{-1} & & I \end{bmatrix}$$

Then, applying E_s on the left and right of A leads to

$$E_s A E_s^\top = \begin{bmatrix} I & & \\ & A_{ww} & A_{wn} \\ & A_{nw} & A_{nn} - A_{ns} A_{ss}^{-1} A_{sn} \end{bmatrix} = \begin{bmatrix} I & & \\ & A_{ww} & A_{wn} \\ & A_{nw} & B_{nn} \end{bmatrix}$$

We notice that this may introduces (potentially many) new n_i - n_j edges not present before, a fill-in. However, there was no modification involving w . This is key in the ND ordering: there are no edges s - w , so no fill-in outside the neighbors.

2.5. Interfaces Scaling. Once that the separators or interiors at a given level have been eliminated, the algorithm goes through each interface and sparsifies it. However, a critical step before this is the proper scaling of each of those clusters. The goal is to scale (what is left of) A such that each diagonal block corresponding to a given interface is the identity. This provides theoretical guarantees ([section 3](#)) and significantly improves the accuracy of the preconditioner ([section 4](#)).

Consider the matrix

$$A = \begin{bmatrix} A_{pp} & A_{pn} \\ A_{np} & A_{nn} \end{bmatrix}$$

We define the block-scaling operation over p as

$$S_p = \begin{bmatrix} L_p^{-1} & \\ & I \end{bmatrix}$$

The result is

$$S_p A S_p^\top = \begin{bmatrix} I & L_p^{-1} A_{pn} \\ A_{np} L_p^{-\top} & A_{nn} \end{bmatrix} = \begin{bmatrix} I & C_{pn} \\ C_{np} & A_{nn} \end{bmatrix}$$

2.6. Interface Sparsification using Low-Rank Approximations. Now that interiors have been eliminated and each interface scaled, the final step is the sparsification. At this stage, the algorithm will go through each interface, p , and sparsify it, using low-rank approximations. Consider again

$$A = \begin{bmatrix} A_{pp} & A_{pn} \\ A_{np} & A_{nn} \end{bmatrix}$$

2.6.1. Using orthogonal transformations. Let us assume $A_{pp} = I$. This is not a loss of generality, as it can always be obtained by scaling p , as described in the previous section. Let us also assume that A_{pn} can be well approximated by a low-rank matrix, i.e.,

$$A_{pn} = Q_{pc}W_{cn} + Q_{pf}W_{fn}, \quad \|W_{fn}\|_2 = \mathcal{O}(\varepsilon)$$

where Q_{pc} is a thin orthogonal matrix and Q_{pf} its complement. This can be computed using a rank-revealing QR (RRQR) or a singular value decomposition (SVD) [27, 11, 30]. We use the letters c to denote the “coarse” (also known as “skeleton” or “relevant”, [37]) dofs, and f the “fine” (“redundant” or “irrelevant”) dofs. Let Q_{pp} be a square orthogonal matrix built as $Q_{pp} = [Q_{pf} \quad Q_{pc}]$. This implies

$$Q_{pc}^\top A_{pn} = W_{cn}, \quad Q_{pf}^\top A_{pn} = W_{fn} = \mathcal{O}(\varepsilon)$$

Then, define

$$(2.1) \quad Q_p = \begin{bmatrix} Q_{pp} & \\ & I \end{bmatrix}$$

We see that

$$Q_p^\top A Q_p = \begin{bmatrix} I & & W_{fn} \\ & I & W_{cn} \\ W_{fn}^\top & W_{cn}^\top & A_{nn} \end{bmatrix} = \begin{bmatrix} I & & \mathcal{O}(\varepsilon) \\ & I & W_{cn} \\ \mathcal{O}(\varepsilon) & W_{cn}^\top & A_{nn} \end{bmatrix}$$

After the orthogonal transformation, f only has very “weak” connections to n . If we ignore the $\mathcal{O}(\varepsilon)$ term, this is the same as dropping the n – f edge. This effectively means f has been eliminated.

However, note that this did *not* introduce any new edge with any of the neighbors of p . This is the key difference with a “regular” elimination as described previously: we can eliminate part of a cluster, here f , *without forming new edges between its neighbors*. The n – n edge is unaffected by this operation (i.e., there is no fill-in). A regular elimination, on the other hand, would have changed the edges n – n .

2.6.2. Variant using Interpolative Transformations. The previous section details the sparsification process using orthogonal transformations. However, this can also be done using other transformations. In this section we explain one variant using interpolative factorization, which was the original idea in [37].

Assume we can *partition* $p = c \cup f$ (so in this case c and f are subsets of p) such that

$$A_{nf} = A_{nc}T_{cf} + \mathcal{O}(\varepsilon).$$

This is often called “interpolative decomposition”. It can be computed for instance using a rank-revealing QR (RRQR) factorization [18] (note that the RRQR is computed over A_{np} instead of A_{pn} in [subsection 2.6.1](#)): computing a RRQR over A_{np} leads to (with P the permutation, and $R_{22} = \mathcal{O}(\varepsilon)$)

$$\begin{aligned} [A_{nc} \quad A_{nf}] &= A_{np}P = [Q_1 \quad Q_2] \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix} \\ \Rightarrow A_{nf} &= \underbrace{Q_1 R_{11}}_{A_{nc}} \underbrace{R_{11}^{-1} R_{12}}_{T_{cf}} + \underbrace{Q_2 R_{22}}_{=\mathcal{O}(\varepsilon)} \end{aligned}$$

Note that this factorization can also be computed using randomized methods [39]. This technique is referred to as “interpolative” because it is exact on A_{nc} : only A_{nf} is approximated and T_{cf} acts as an interpolation operator (i.e., as a set of Lagrange basis functions).

Now, consider

$$T_p = \begin{bmatrix} I & & \\ -T_{cf} & I & \\ & & I \end{bmatrix}$$

Notice how T_p is a lower-triangular matrix, while Q_p in Eq. 2.1 was orthogonal. Both can be efficiently inverted; however, working with orthogonal matrices brings stability guarantees (see section 3). Then,

$$T_p^\top A T_p = \begin{bmatrix} C_{ff} & C_{fc} & \mathcal{O}(\varepsilon) \\ C_{cf} & A_{cc} & A_{cn} \\ \mathcal{O}(\varepsilon) & A_{nc} & A_{nn} \end{bmatrix}$$

with

$$C_{ff} = A_{ff} - A_{fc}T_{cf} - T_{cf}^\top A_{cf} + T_{cf}^\top A_{cc}T_{cf}, \quad C_{cf} = A_{cf} - A_{cc}T_{cf}, \quad C_{fc} = C_{cf}^\top$$

The final result is the same as using orthogonal transformation. The differences are that

- A_{pp} is not required to be identity;
- A_{cn} is simply a subset of A_{pn} .

However, as we will see later on, there is a significant accuracy loss when using this technique without block scaling as opposed to orthogonal transformations with block scaling. Furthermore, it does not guarantee that the approximation stays SPD.

2.7. Clusters merge. Finally, once we have eliminated all separators at a given level, we need to merge the interfaces of every remaining ND separator. Consider for instance Fig. 2.6. After having eliminated the leaf (level $\ell = 4$) and the level $\ell = 3$ separators, we need to merge the clusters in each separator. This is done following the cluster trees. Merging children clusters p_1, \dots, p_k into a parent cluster p simply means concatenating their dofs:

$$p = [p_1 \quad p_2 \quad \dots \quad p_k].$$

Then, all block rows and columns corresponding to p_1, \dots, p_k get concatenated into p .

2.8. Sparsified Nested Dissection. Now that we have introduced all the required building blocks (block elimination, scaling and sparsification), we can present the complete algorithm. Given a matrix A , appropriately ordered and clustered, the algorithm simply consists of applying a sequence of eliminations E_s (subsection 2.4), scalings S_p (subsection 2.5) and sparsification Q_p (subsection 2.6) (plus potentially some re-orderings and permutations to take care of the fine nodes f and the merge), at each level ℓ , effectively reducing A to (approximately) I :

$$M^\top A M \approx I \text{ with } M = \prod_{\ell=1}^L \left(\prod_{s \in S_\ell} E_s^\top \prod_{p \in C_\ell} S_p^\top \prod_{p \in C_\ell} Q_p \right)$$

In this expression, S_ℓ are all the ND separators at level ℓ and C_ℓ are all the clusters (interfaces) in the graph right after level ℓ elimination. Since M is given as a product of elementary transformations, it can easily be inverted. We refer to this algorithm as spaND, which stands for “sparsified Nested Dissection”. [Algorithm 2.3](#) presents the algorithm.

Algorithm 2.3 The spaND algorithm (**OrthS**).

Require: $A \succ 0$; $L > 0$; ε

$M = []$ (empty list)

Compute a L -levels modified ND ordering of A using [Algorithm 2.2](#). Infer clusters hierarchy in each ND separator.

for all $\ell = L, \dots, 1$ **do**

for all s separator at level ℓ **do** {Eliminate separators at level ℓ }

 Eliminate s , get E_s ([subsection 2.4](#))

 Append E_s to M

end for

for all p interfaces **do** {Scale interface}

 Scale p , get S_p ([subsection 2.5](#))

 Append S_p to M

end for

for all p interface **do** {Sparsify interfaces}

 Sparsify p with accuracy ε , get Q_p ([subsection 2.6](#))

 Append Q_p to M

end for

for all s separator **do** {Merge clusters}

 Merge interfaces of s one level following clusters hierarchy ([subsection 2.7](#))

end for

end for

return $M = \prod_{\ell=1}^L \left(\prod_{s \in S_\ell} E_s^\top \prod_{p \in C_\ell} S_p^\top \prod_{p \in C_\ell} Q_p \right)$ (such that $M^\top A M \approx I$)

We illustrate the effect of all the E_s , S_p and Q_p^\top in A (i.e., the trailing matrix) on [Fig. 2.7](#). The two top rows show the actual trailing matrix, while the two bottom rows show the evolution of the matrix graph’s clusters as the elimination and sparsification proceeds.

3. Theoretical results. We here discuss a couple of facts related to the above factorizations.

3.1. Sparsification and Error on the Schur Complement. Consider a framework where

$$A = \begin{bmatrix} A_{pp} & A_{pn} \\ A_{np} & A_{nn} \end{bmatrix}.$$

Without loss of generality, we do not include the w - w and w - n blocks, as they are completely disconnected from p and unaffected by the sparsification. Then, consider a general low-rank approximation

$$A_{pn} = X_1 Y_1^\top + X_2 Y_2^\top$$

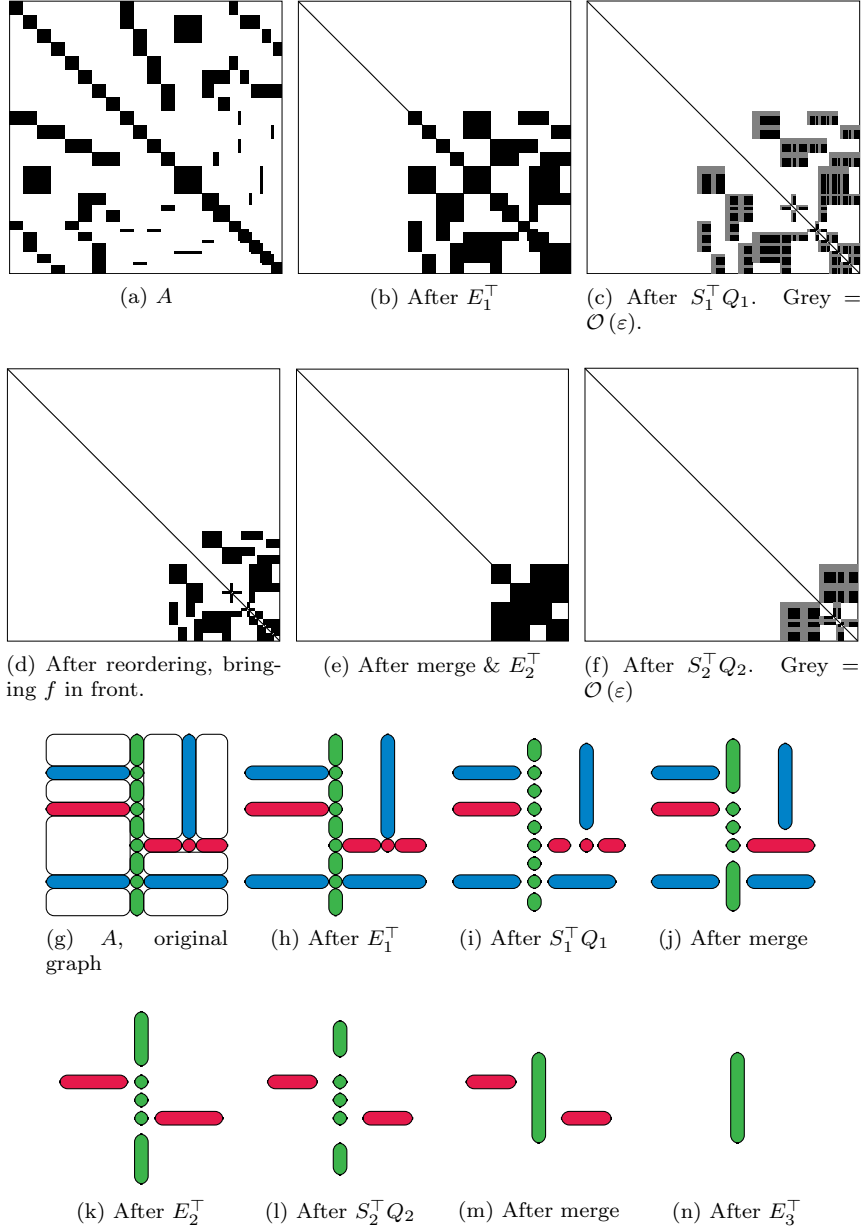


Fig. 2.7: Illustration of the spaND algorithm. Given A , create a ND tree of depth 4 and cluster A accordingly, as shown on Fig. 2.7g. This cartoon shows clusters of vertices of A , where the edges (not shown) should be thought as connecting close neighbors (like on a regular 2D grid). Denote by E_ℓ , S_ℓ and Q_ℓ all eliminations, scalings and sparsifications at level ℓ . Then, we have $E_4 E_3 (Q_2^\top S_2 E_2) (Q_1^\top S_1 E_1) A (E_1^\top S_1^\top Q_1) (E_2^\top S_2^\top Q_2) E_3^\top E_4^\top \approx I$. The top rows show the evolution of the trailing matrix; bottom rows show the evolution of the matrix graph after eliminations, sparsifications and merges. We represent the sparsification process by shrinking the size of the clusters.

Version	Error on n - n	Cost
In	$\mathcal{O}(\varepsilon^2) \ C_{ff}^{-1}\ _2$	$\mathcal{O}(p n c)$ C_{ff} arbitrary
InS	$\mathcal{O}(\varepsilon^2) \ C_{ff}^{-1}\ _2$	$\mathcal{O}(p n c)$ $C_{ff} = I + T_{cf}^\top T_{cf}$
OrthS	$\mathcal{O}(\varepsilon^2)$	$\mathcal{O}(p n c)$

Table 3.1: Error for various approximations. The left column indicate the sparsification variant: **In** means interpolative and no scaling; **InS** means interpolative and scaling; **OrthS** means orthogonal and scaling.

where $\|Y_2\| = \mathcal{O}(\epsilon)$. Using $X = \begin{bmatrix} X_1 & X_2 \end{bmatrix}$ as a change of variable, A becomes

$$\begin{bmatrix} X^{-1} & \\ & I \end{bmatrix} \begin{bmatrix} A_{pp} & A_{pn} \\ A_{np} & A_{nn} \end{bmatrix} \begin{bmatrix} X^{-\top} & \\ & I \end{bmatrix} = \begin{bmatrix} B_{pp} & Y^\top \\ Y & A_{nn} \end{bmatrix} = \begin{bmatrix} B_{11} & B_{12} & Y_1^\top \\ B_{21} & B_{22} & Y_2^\top \\ Y_1 & Y_2 & A_{nn} \end{bmatrix}$$

The sparsification process then assumes $Y_2 = 0$ and eliminates the 2-2 block. The true n - n Schur complement is

$$S = A_{nn} - Y_2 B_{22}^{-1} Y_2^\top$$

while the approximate one, ignoring Y_2 , is simply A_{nn} . The error is then

$$E_{nn} = Y_2 B_{22}^{-1} Y_2^\top.$$

We can now consider the different variants proposed in [subsection 2.6](#):

- (**In**) spaND using interpolative factorization and no diagonal block scaling. This gives $B_{22} = C_{ff}$ and $Y_2 = A_{nf} - A_{nc} T_{cf}$, so that

$$\|E_{nn}\|_2 \leq \|Y_2\|_2^2 \|B_{22}^{-1}\|_2 = \mathcal{O}(\varepsilon^2) \|C_{ff}^{-1}\|.$$

- (**InS**) spaND using interpolative factorization and diagonal block scaling. This leads to

$$\|E_{nn}\|_2 \leq \|Y_2\|_2^2 \|B_{22}^{-1}\|_2 = \mathcal{O}(\varepsilon^2) \|C_{ff}^{-1}\|.$$

However, since $A_{ss} = I$, $C_{ff} = I + T_{cf}^\top T_{cf}$, we can expect, if T_{cf} is small (which happens if the right algorithm is employed, see [\[41\]](#)), $\|C_{ff}^{-1}\|$ to be much closer to 1.

- (**OrthS**) spaND using orthogonal factorization and diagonal block scaling. In this case, we simply have $B_{22} = I$ and $Y_2 = W_{fn}^\top$, and so,

$$\|E_{nn}\|_2 \leq \|Y_2\|_2^2 = \mathcal{O}(\varepsilon^2).$$

Table [3.1](#) summarizes the results. We notice that those three variants have roughly the same cost, since they require a RRQR over A_{pn} or A_{np} , and their cost is proportional to $\mathcal{O}(|p||n||c|)$ with $|c|$ the resulting rank [\[27, Algorithm 5.4.1\]](#)

The key is that the interpolative error bound (without and to some extent with scaling) includes the potentially large $\|C_{ff}^{-1}\|_2$ term, which is not present with the **OrthS** version. This indicates that we can expect the versions with diagonal scaling to have smaller errors E_{nn} . This will be verified in [section 4](#).

3.2. Stability of the Block Scaling & Orthogonal Transformations Variant. In addition to a smaller n - n error as explained previously, the `OrthS` version provides stability guarantees.

THEOREM 3.1. *Let*

$$A = \begin{bmatrix} I & A_{pn} \\ A_{pn}^\top & A_{nn} \end{bmatrix}$$

be a SPD matrix. For any low-rank approximation

$$A_{pn} = Q_{pf}W_{fn} + Q_{pc}W_{cn}$$

where $Q_p = [Q_{pf} \quad Q_{pc}]$ is a square orthogonal matrix,

$$B_p = \begin{bmatrix} I & W_{cn} \\ W_{cn}^\top & A_{nn} \end{bmatrix}$$

is SPD.

Proof. The $n - n$ Schur Complement of B_p (when eliminating c) is

$$S_B = A_{nn} - W_{cn}^\top W_{cn}.$$

On the other hand, the $n - n$ Schur Complement of A (when eliminating p) is

$$S_A = A_{nn} - A_{np}^\top A_{pn} = A_{nn} - W_{cn}^\top W_{cn} - W_{fn}^\top W_{fn}$$

which implies

$$S_B = S_A + W_{fn}^\top W_{fn}.$$

Since A is SPD, so is S_A , and since $W_{fn}^\top W_{fn} \succeq 0$, we find that S_B is SPD. Since the $c - c$ block of B_p is identity, we conclude that B_p is SPD. \square

COROLLARY 3.2. *For any SPD matrix and $\varepsilon \geq 0$, the sparsified matrices of the spaND algorithm using block diagonal scaling and orthogonal low-rank approximations (`OrthS`) remain SPD. In other words, the algorithm never breaks down.*

Note that the above corollary does *not* depend on the quality of the low-rank approximation, i.e., it works even for $\varepsilon = 0$. It merely relies on the fact that the truncated error ($Q_{pf}W_{fn}$) is orthogonal to what is retained ($Q_{pc}W_{cn}$) and that the scheme is using a “weak admissibility” criterion (*all* edges of p are compressed). Finally, note that the above proof also shows that

$$S_B = S_A + \mathcal{O}(\varepsilon^2), \quad S_B \succeq S_A.$$

This is a classical result in the case of low-rank approximation using weak admissibility (see [58, 59] for instance).

3.3. Complexity analysis. We discuss the complexity of spaND and contrast it with the usual ND algorithm.

Classical ND. Nested dissection leads to a binary tree decomposition of the graph of A (although n -ary trees are possible). In the literature, the nested dissection tree is often defined as a tree of separators. Here for convenience, we take a slightly different viewpoint where each node is a subgraph of G . Both viewpoints are equivalent. We start with the root node that corresponds to the full graph G of size N . We define the children nodes as the subgraphs that are disconnected by the separator. This process is applied recursively to define the entire tree.

In our complexity analysis, we are going to assume that all the graphs for sparse matrices satisfy the following nested dissection property. We assume that leaf nodes contain at most N_0 nodes, where $N_0 \in \mathcal{O}(1)$. Consider a node i in the tree, of size n_i . Consider the set D_i of all nodes j such that they are descendant of i and they contain at least $n_i/2$ nodes. We assume that $|D_i| \in \mathcal{O}(1)$, that is the size of this set is bounded for all i and N . This property is satisfied for β -balanced trees in which all children subgraphs have size βn_i , for some $0 < \beta < 1$ independent of i and N . In that case we have $|D_i| \leq 1 + \log 2 / \log \beta^{-1}$.

We assume that all separators are minimal in the sense that each node in a separator is connected to the two children subgraphs in the nested dissection partitioning (otherwise this node can be moved to one of the subgraphs).

Finally, we assume that a subgraph of size n_i is connected to at most $\mathcal{O}(n_i^{2/3})$ nodes in G .

As far as the authors know, all matrices that arise in the discretization of partial differential equations in 3D using a local stencil satisfy this property.

Consider now a node i of size $2^{-\ell}N \leq n_i < 2^{-\ell+1}N$ (see Fig. 2.3). The associated separator has size at most

$$c_\ell \in \mathcal{O}\left(2^{-2\ell/3}N^{2/3}\right)$$

Further, the fill-in results in at most $\mathcal{O}\left(2^{-2\ell/3}N^{2/3}\right)$ non-zero entries in each row. The cost of eliminating a separator in that size range is bounded by

$$e_\ell \in \mathcal{O}\left(\left(2^{-2\ell/3}N^{2/3}\right)^3\right) = \mathcal{O}\left(2^{-2\ell}N^2\right)$$

From our assumption, the number of clusters of size $2^{-\ell}N \leq n_i < 2^{-\ell+1}N$ is bounded by 2^ℓ . Hence, the overall factorization cost is bounded by

$$t_{\text{ND,fact}} \in \mathcal{O}\left(\sum_{\ell=0}^L 2^\ell e_\ell\right) = \mathcal{O}\left(\sum_{\ell=0}^L 2^{-\ell}N^2\right) = \mathcal{O}\left(N^2\right), \quad L \in \Theta(\log(N/N_0))$$

We recover the usual computational cost of nested dissection for 3D meshes. Most of the computational expense is at the top of the nested dissection tree, with the final separator of size $N^{2/3}$.

The complexity of applying the factorization can be derived similarly. Since for each cluster of size n_i , its separator has $\mathcal{O}\left(2^{-2\ell/3}N^{2/3}\right)$ fill-in entries in its rows, the related solve cost is $\mathcal{O}\left(2^{-4\ell/3}N^{4/3}\right)$ and the cost of one solve is

$$t_{\text{ND,apply}} \in \mathcal{O}\left(\sum_{\ell=0}^L 2^{-\ell/3}N^{4/3}\right) = \mathcal{O}\left(N^{4/3}\right)$$

spaND. On the other hand, assume that the sparsification is able to decrease each separator size before elimination from c_ℓ to

$$s_\ell \in \mathcal{O}\left(2^{-\ell/3}N^{1/3}\right)$$

This means that the rank scales roughly like the diameter of the separators. This is also the rank of the off-diagonal blocks for separators in the original matrix A . The assumption in some sense is that the rank of far-away fill-ins is $\mathcal{O}(1)$. This is comparable with complexity assumptions in the fast multipole method for example.

We now discuss a few additional assumptions regarding the construction of the interfaces, in order to guarantee the final $\mathcal{O}(N \log N)$ cost. Recall that interfaces are used for sparsification and correspond to a multilevel partitioning of the separators. We will say that two nodes (i, j) at the same level in the nested dissection tree are neighbors if there is a node in G that belongs to a separator at this level or above, and that is connected to i and j , in the graph G . We will assume that each node has only $\mathcal{O}(1)$ neighbors. Under this assumption, each interface is connected to $\mathcal{O}(1)$ interfaces at the same level.

Considering the computational cost, for all nodes of size $2^{-\ell}N \leq n_i < 2^{-\ell+1}N$, the cost can be divided into:

- eliminating separators. With the same reasoning as previously, and since an interface is connected to $\mathcal{O}(1)$ interfaces, the cost is bounded by

$$\mathcal{O}\left((2^{-\ell/3}N^{1/3})^3\right) = \mathcal{O}(2^{-\ell}N)$$

- scaling and sparsifying the remaining interfaces. By construction the size of each interface is in $\mathcal{O}(2^{-\ell/3}N^{1/3})$. Since sparsification has cost $\mathcal{O}(m^2n)$ for a matrix block of size $m \times n$, the cost of sparsifying one interface is bounded similarly by $\mathcal{O}(2^{-\ell}N)$.

Hence, under our assumptions, the overall factorization cost for spaND is

$$t_{\text{spaND, fact}} \in \mathcal{O}\left(\sum_{\ell=1}^L 2^{\ell} 2^{-\ell}N\right) = \mathcal{O}(N \log N)$$

The complexity of applying the factorization can be derived like previously. A direct calculation leads to

$$t_{\text{spaND, apply}} \in \mathcal{O}\left(\sum_{\ell=0}^L 2^{\ell} \left(2^{-\ell/3}N^{1/3}\right)^2\right) = \mathcal{O}\left(\sum_{\ell=1}^L 2^{\ell/3}N^{2/3}\right) = \mathcal{O}(N)$$

Finally, notice that in both cases the memory complexity scales like the factorization application. Section 4.2 presents some experimental results regarding separator sizes as a function of N .

4. Numerical Experiments. This section presents applications of the algorithm on various problems. All matrices are symmetric, real and positive-definite.

We use the following notation throughout this section:

- t_F is the factorization time (in seconds), not including partitioning;
- t_P is the partitioning time (in seconds);
- t_S is the total time (in seconds) required for CG to reach a relative residual $\|Ax - b\|_2 / \|b\|_2$ of 10^{-12} . It is the total time to reach convergence. While this is quite a small value, being able to reach those tolerances is a good indication of the numerical stability of the algorithm (i.e., that the preconditioner does not prevent CG from converging to a small tolerance);
- n_{CG} is the associated number of CG steps;
- size_{Top} is the size of the top separator right before elimination;
- mem_F is the number of non-zero entries in the factorization;

On top of this, at some point we compare spaND to classical “exact” ND (using spaND with no compression & scaling; “Direct”) and to a classical ILU(0) [46] (“ILU(0)”).

All tests were run on a machine with 300 GB of RAM and a Intel(R) Xeon(R) Gold 5118 CPU at 2.30GHz. The algorithm is sequential and was written in C++. We

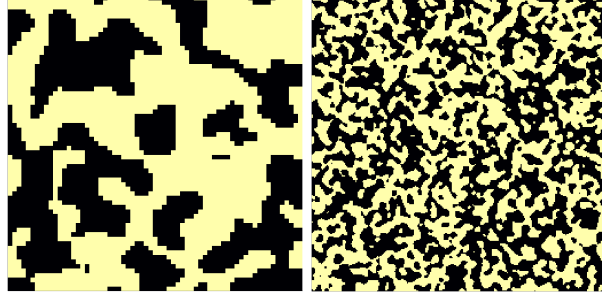


Fig. 4.1: A quantized high contrast field with lows of ρ^{-1} and highs of ρ for $n = 32$ (left) and $n = 128$ (right). The features sizes are roughly constant as we increase the mesh size n .

use GCC 8.1.0 and Intel(R) MKL 2019 for Linux for the BLAS & LAPACK operations. When no geometry information is available, we use Metis 5.1 [38] for the vertex-separator routine. We use Ifpack2 [44] for ILU(0). Low-rank approximations are performed using LAPACK's geqp3 [4]. The truncation uses a simple rule, truncating based on the absolute value of the diagonal entries of the R factor. This means that, given R , we select the first r rows, where $\frac{|R_{ii}|}{|R_{11}|} \geq \varepsilon$ for $1 \leq i \leq r$.

4.1. Impact of Diagonal Scaling & Orthogonal Transformations. In this first set of experiments we compare, empirically, the three variants of the algorithm:

- (In) spaND using interpolative factorization and no diagonal block scaling;
- (InS) spaND using interpolative factorization and diagonal block scaling;
- (OrthS) spaND using orthogonal factorization and diagonal block scaling.

This should be contrasted with prior work [37] where the algorithm was using the interpolative only variant (with no scaling).

4.1.1. High contrast 2D Laplacians. We first consider 2D elliptic equations

$$(4.1) \quad \nabla(a(x) \cdot \nabla u(x)) = f \quad \forall x \in \Omega = [0, 1]^2, \quad u|_{\partial\Omega} = 0$$

where $a(x)$ is a quantized high contrast field with high of ρ and low of ρ^{-1} and where 4.1 is discretized with a 5-points stencil. This leads to the following discretization

$$(a_{i-1/2,j} + a_{i+1/2,j} + a_{i,j-1/2} + a_{i,j+1/2})u_{ij} - a_{i-1/2,j}u_{i-1,j} - a_{i+1/2,j}u_{i+1,j} - a_{i,j-1/2}u_{i,j-1} - a_{i,j+1/2}u_{i,j+1} = h^2 f_{ij}$$

The field a is built in the following way:

- create a random $(0, 1)$ array \hat{a}_{ij} ;
- smooth \hat{a} by convolving it with a unit-width Gaussian;
- quantize \hat{a}

$$a_{ij} = \begin{cases} \rho & \text{if } \hat{a}_{ij} \geq 0.5 \\ \rho^{-1} & \text{else} \end{cases}$$

Fig. 4.1 gives an example of high contrast field for $n = 32$ and $n = 128$.

We compare the number of iterations CG [36] needs to reach a residual of 10^{-12} . In all those experiments, a missing value indicates the factorization was not SPD and, at some point, Cholesky (subsection 2.4) failed. Given that the problem is defined

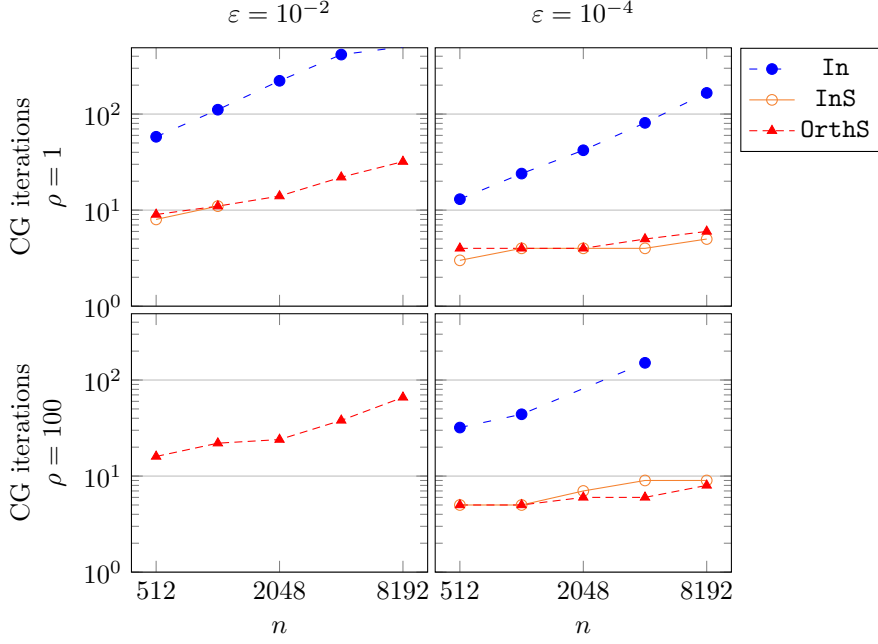


Fig. 4.2: 2D $n \times n$ Laplacians: each line represent a given variant: **In** (interpolative and no scaling), **InS** (interpolative and scaling) and **OrthS** (orthogonal and scaling), at a given accuracy ε . Each dot gives the CG iteration count, and we run the experiments on various problem of size $N = n^2$, for various ρ . The conditioning is roughly proportional to ρ . A missing data point means Cholesky broke down and the preconditioner is not SPD. This shows that, in general, **InS** and **OrthS** are much more accurate than **In** at a given ε and **OrthS** never breaks down. In addition, for small enough ε , the accuracy is roughly independent from the problem size N . Finally, when ρ is not too extreme, there is little dependency with the condition number.

on a regular mesh, we use a variant of [Algorithm 2.2](#) where the vertex-separator used is based on geometry. This leads to a more regular clustering and, in general, to slightly better performances (in terms of time or memory — CG iterations and the preconditioner accuracy are usually unaffected).

[Fig. 4.2](#) gives results for $\rho = 1$ to $\rho = 1000$. We compare the three variants for various ε and problem size $N = n^2$. We observe three things from the experiment. First, the number of iterations, particularly at moderate accuracies ($\varepsilon = 10^{-1}$ or 10^{-2}), is greatly reduced using **InS**. Further, the **OrthS** variant is usually the most accurate. This is likely due to the improved robustness and accuracies of the orthogonal transformations versus the interpolative ones. Finally, we see that, while the **In** and **InS** variants may fail due to non-SPD approximations, the **OrthS** never fails and can always be run, even at $\varepsilon \approx 1$. We finally note that the small target residual of 10^{-12} in CG illustrates the good numerical properties of the preconditioner. Previous work [\[37\]](#) was focused on the interpolative only variant. Both the scaling and orthogonal transformations greatly improve the algorithm: they reduce the CG iteration count and guarantee that the preconditioner stays SPD for SPD problems.

4.1.2. Non-Regular Problems. Fig. 4.3 gives results for three variants on many¹ of the SPD (real & square) problems from the SuiteSparse matrix collection [20] with more than 50 000 rows and columns.

Most of these problems come from PDE discretization, but not all. **G2circuit** for instance comes from a circuit simulation problem and **finan512** comes from a portfolio optimization problem.

For most problems, an accuracy of $\varepsilon = 10^{-2}$ leads to a number of iterations usually less than 100, while an accuracy of $\varepsilon = 10^{-4}$ leads usually to less than 10 iterations. Only the **Botonakis/thermomech_TK** problem needs more than 100 iterations for $\varepsilon = 10^{-6}$.

Fig. 4.4 shows a performance profile regarding the CG iteration count. Each plot compares the three variants for a given accuracy. For a given problem p and a variant v , let $CG_{p,v}$ be the CG count and CG_p^* the best result among the three variants (**In**, **InS** and **OrthS**), for a problem p . Then each curve is defined as

$$T_v(t) = \frac{\#\left\{p \in P \mid \frac{CG_{p,v}}{CG_p^*} \leq t\right\}}{\#P}$$

Each value $T_v(t)$ basically represent the fraction of problems where variant v is within t times the best algorithm. Problems for which the factorization broke down are given $CG_{p,v} = \infty$ and for the others the CG count was capped at 500.

On $\varepsilon = 10^{-1}$ and $\varepsilon = 10^{-2}$, **InS** and **In** often break down, so **OrthS** is significantly better. When **InS** does not break down however, it has similar performances as **OrthS**. On $\varepsilon = 10^{-4}$, **InS** rarely breaks down, and performances are very similar to **OrthS** throughout all the runs. On $\varepsilon = 10^{-6}$, most cases converge in a couples iterations, so the three variants have similar performances. The plots clearly shows that **OrthS** is the optimal strategy, being within at most 2 of the optimal in the worst case, and being often the winning algorithm.

Further, using orthogonal transformations guarantees that the approximation stays SPD, allowing the algorithm to not break down even for high ε 's. The number of iterations of **OrthS** is not always strictly smaller than the **InS** variant, while it is for the regular Laplacian examples. However, the extra robustness (no need for pivoting) of the orthogonal transformations make them quite attractive in practice for SPD problems.

We also point out that previous work [37] was restricted to standard elliptic model problems. To the best of our knowledge, this is the first application of this algorithm to a wide range of problems.

4.2. Scalings with problem size. We now consider scalings, i.e., how does the algorithm perform as N grows. Fig. 4.5 shows the evolution of the top separator size right before elimination (top) and the number of CG iterations (bottom) for $\rho = 1$ and $\rho = 100$ for 3D problems generated as in subsection 4.1.1 with a classic 7-points stencil. From now on, we will only consider the scaling & orthogonal method (**OrthS**).

This figure shows two properties of the algorithm:

- the top separator size (size_{Top}) typically grows like $\mathcal{O}(N^{1/3})$, regardless of ε ;
- for small enough ε , the number of CG iterations is roughly $\mathcal{O}(1)$.

¹We only excluded **Queen4147** and **Bump2911** (for which the solver ran out of memory) as well as the **Andrews** and **denormal** cases (which are so ill-conditioned that spaND never converges in less than 500 iterations).

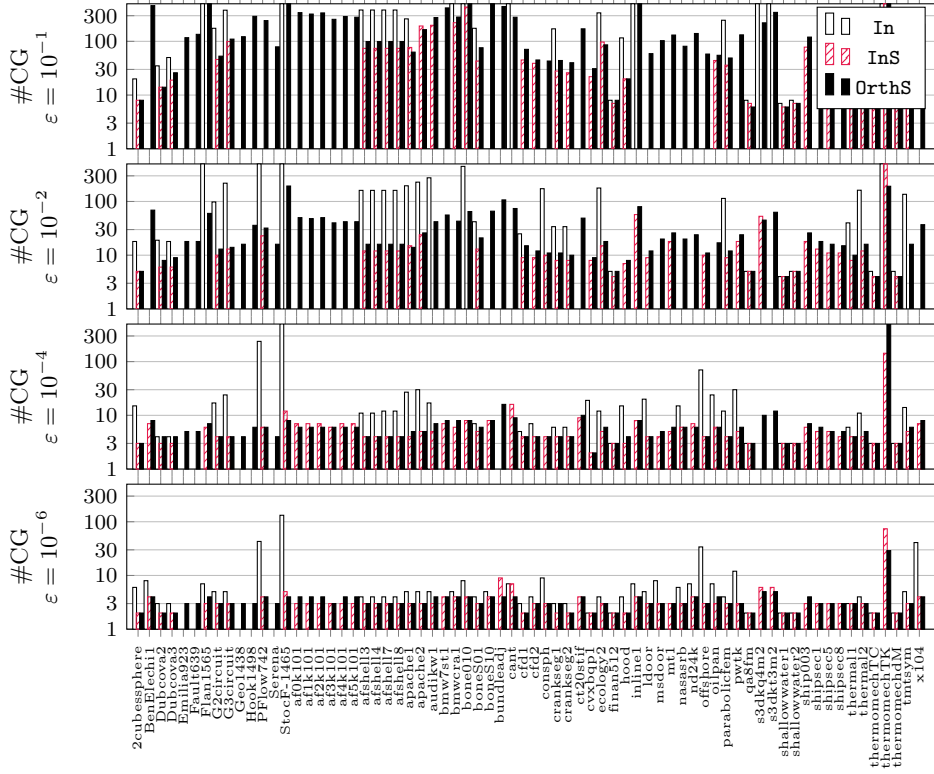


Fig. 4.3: SuiteSparse matrix collection: result on many SPD problems of the SuiteSparse matrix collections with $N \geq 50\,000$. The partitioning is only graph-based. Each bar represents the number of CG iterations for a given problem with a given variant of the algorithm: In (interpolative and no scaling), InS (interpolative and scaling) and OrthS (orthogonal and scaling) at a given accuracy ε . The absence of a bar means the algorithm broke down in the face of a non-SPD pivot. This shows that, as $\varepsilon \rightarrow 0$, the algorithm converges on a wide range of problems. This also shows that the scaling is beneficial in almost all cases. The orthogonal transformations, while not always better (in terms of accuracy) than the interpolative transformations, do guarantee that the preconditioner stays SPD and the factorization never breaks down because of indefinite pivots.

Combining those two properties, we can expect (see [subsection 3.3](#)), for small enough ε ,

- a factorization time of $\mathcal{O}(N \log N)$;
- a solve time of $\mathcal{O}(N \cdot 1) = \mathcal{O}(N)$,

which implies that the algorithm scales roughly linearly with N .

4.3. Timings and Memory Usage. We now study the efficiency of the algorithm in terms of time (factorization and solve time) and memory usage on “real-life” problems. To evaluate our algorithm, we use the following two metrics:

- the factorization and solve time (t_F and t_S);
- the memory footprint (mem_F , the number of non-zeros in the preconditioner

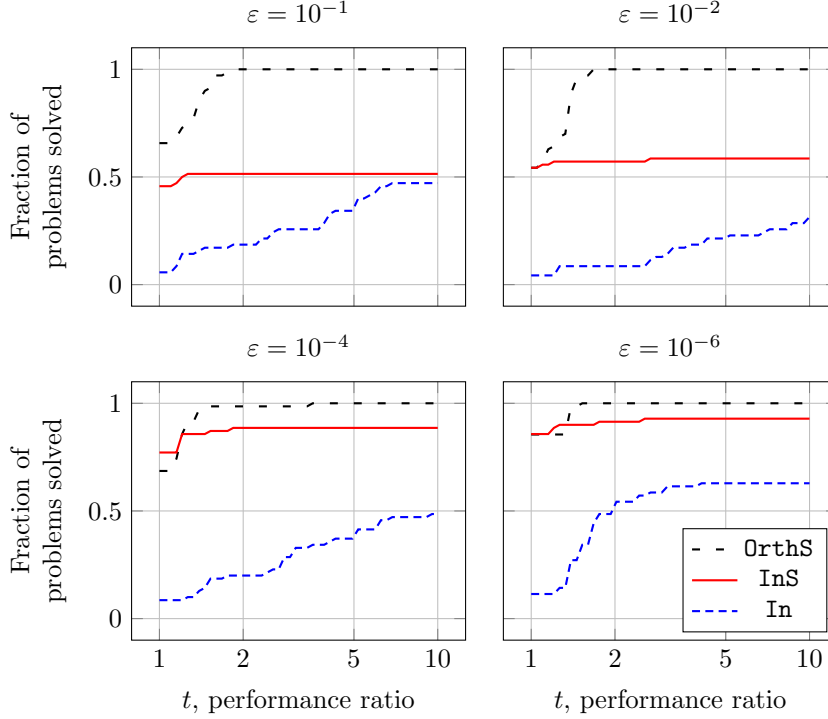


Fig. 4.4: Performance profile for all the SuiteSparse experiments (Fig. 4.3). Higher is better. The performance criterion is $\#CG$, the number of CG iterations. Each point $T_v(t)$ gives the fraction of problems for which variant v completed with a CG iterations count less than t times the best variant. An excellent method is one that starts at $t = 1$ close to 1 and quickly reaches 1 as t increases. This means that this method outperforms the other methods in almost all cases. **InS** and **OrthS** typically have the same number of iterations, but **InS** sometimes leads to a non-SPD preconditioner, hence the large difference in performances. **In** typically leads to a much larger iteration count. Looking at the bottom left figure ($\varepsilon = 10^{-4}$) for example, we see that for half of the problems **In** has a CG count more than 10 times greater than the best variant. For all cases, the **OrthS** is within a factor of 2 of the optimal CG count. This shows the importance of both the scaling and the orthogonal transformations.

M).

SuiteSparse. Table 4.1 shows the results on two specific problems from the SuiteSparse collection [20], inline and audikw. For both problems, we see that the “sweet-spot” in terms of minimal time-to-solution is not for high ε , but for much smaller ε . For the audikw problem, the optimal is when using $\varepsilon = 10^{-2}$, and for inline, 10^{-4} gives optimal result. The size size_{Top} for inline are overall much smaller than for audikw. This is usually an indication that the problem is near 2D, for which size_{Top} is typically $\mathcal{O}(1)$. Those problems are of fairly small size and, as such, direct solvers (with smaller constants and better implementations) remain competitive.

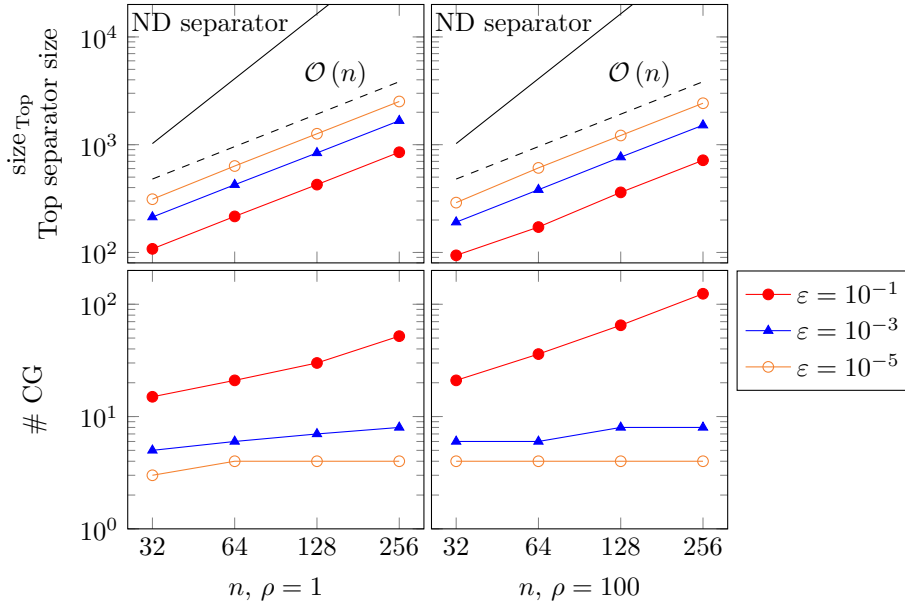


Fig. 4.5: 3D $n \times n \times n$ Laplacian results for $\rho = 1$ (left) and $\rho = 100$ (right) using `OrthS`. We see that size_{Top} (top separator final size, i.e., right before elimination) scales like $\mathcal{O}(n)$, and that increasing the accuracy (decreasing ε) essentially adds a constant; it does not change the scaling. This should be compared with the classical ND separator size (solid line), equal to n^2 . In addition, for a small enough ε , the CG iteration count becomes virtually constant. Both those facts mean the algorithm can be expected to have complexity $\mathcal{O}(N)$ (see [subsection 3.3](#)).

Ice-sheet modeling problem. [Table 4.2](#) gives the result on an ice-sheet modeling problem [52]. This problem comes from the modeling of ice flows on Antarctica using a finite-element discretization. The problem is challenging because of the high variations in background field and the near-singular blocks in the matrix, leading to a condition number of more than 10^{11} . This problem is nearly 2D. The graph in the x, y plane is regular but non-square. It is then extruded in the z direction.

We illustrate the partitioning (top-left) and one layer of the solution (top-right, with a random right-hand side) on a log scale. Note the high variations in scales in the solution. This makes the problem very ill conditioned and hard to solve with classical preconditioners. Since the problem is (nearly) 2D and we are given the geometry, we partition the matrix in the xy plane and extrude the partitioning in the z direction. The partitioning uses a classical recursive coordinate bisection approach [8].

We use two sequences of matrices with a different number of layers in the z direction. We see that size_{Top} grows very slowly, close (but not exactly) like $\mathcal{O}(1)$ for each set of problems. This is typical of 2D or near-2D problems. The memory use is roughly linear for each set of problems, and the factorization time is growing almost linearly. This validates the effectiveness of the algorithm.

We also compared the algorithm against a direct method (simply using `spaND` with no compression but otherwise with the same parameters). The results are in the “Direct” column. We note the very poor scaling of the direct method; our algorithm,

Problem (N)	ε	t_P (s.)	t_F (s.)	t_S (s.)	n_{CG}	$size_{Top}$	mem_F (10^9)
audikw_1 943 695	10^{-1}	96	128	512	277	322	0.46
	10^{-2}	95	268	103	42	606	0.73
	10^{-4}	95	500	18	7	1175	1.08
inline_1 503 712	10^{-1}	40	8	> 224	> 500	1	0.11
	10^{-2}	41	13	44	80	13	0.13
	10^{-4}	41	18	5	8	19	0.16

Table 4.1: Some SuiteSparse performance results using `OrthS`. Completely general partitioning (no geometry information used) using [Algorithm 2.2](#) with Metis as a vertex-separation routine. We see that the algorithm does converge when $\varepsilon \rightarrow 0$. The sweet spot varies for both problems. Notice that `inline_1` has a very small $size_{Top}$, characteristic of near-2D problems, while the top separator has a much larger size for `audikw_1`.

on the other hand, performs much better. In addition, we also compared the algorithm to out-of-the-box algebraic multigrid (AMG, a classical AMG) and Incomplete LU(0). On this specific problem, AMG simply did not converge in less than 500 iterations, the residual stalling around 1.0. While specifically designed AMG can and does solve this problem well [52], this illustrates that out-of-box algorithms cannot always efficiently solve very ill-conditioned problems. Because of this, we do not report those results. We finally tested Ifpack2’s ILU(0) [44] with GMRES. We tested two orderings, horizontal (layer-wise) and vertical (column-wise). The layer-wise ordering gave (by far) the best performances and we report only this one. However, while it is competitive for small problems, it cannot solve large problems because the number of iterations grows quickly, making the algorithm too expensive. This illustrates the strong advantage of spaND: with a nearly constant number of iterations, we do not suffer from this deterioration of the preconditioner and can solve larger problems.

We note that those results can also be compared with recent work using LoRaSp [43, 16] on the same matrices. Overall, while the scaling with N is similar, spaND exhibits better constants.

SPE benchmark. [Table 4.3](#) gives the results on a cubic slide of the SPE (Society of Petroleum Engineering) benchmark [19], a classical benchmark to evaluate oil & gas exploration codes. This matrix models a porous media flow. This problem is particularly challenging for direct methods since it resembles a 3D cubic problem and leads to a high complexity. On the other hand, it can be solved quite efficiently with classical preconditioners like AMG or ILU.

We use various values of n , and the problem is then of size $N = n^3$. The bottom pictures show the $size_{Top}$ and mem_F scaling with N . We see that $size_{Top}$ grows roughly like $\mathcal{O}(N^{1/3})$; this is typical of 3D problems. The memory use grows linearly with N . Furthermore, the number of CG iterations is constant for all resolutions. This serves as another validation of the ability of spaND to solve large-scale problems. In the last column, we show the result using the direct solver. Since it is a direct solver, the memory use is too great, and we cannot solve the 8M problem. Furthermore, the time to solve the 2M problem is about 10 times more than using spaND. This shows the limitations of direct solvers for solving large 3D problems for which the fill-in is too significant.

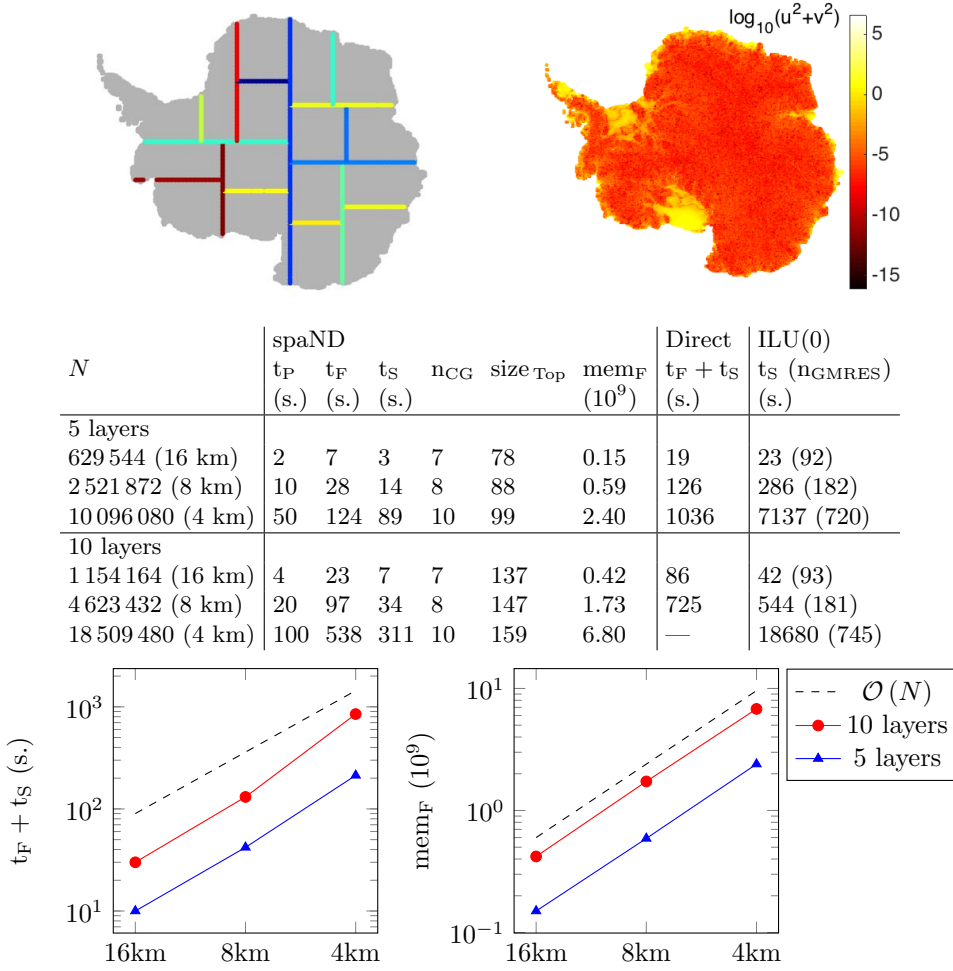
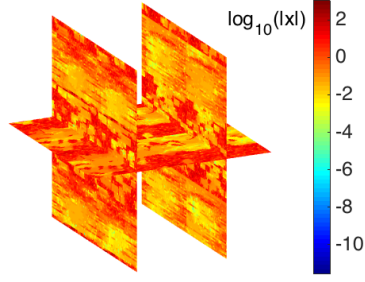


Table 4.2: Ice Sheet results. Unregular geometric partitioning, $\varepsilon = 10^{-2}$, **OrthS**. The top left picture illustrates the separators (for the top 5 levels) and the top right picture shows the solution (for a random right-hand side b) on log scale. — indicates the direct method ran out of memory. We run ILU with 2 ordering: layer-wise ordering and vertical column-wise ordering. The later lead to very poor convergence and is not shown here. This problem is very ill-conditioned and typically very hard to solve using out-of-the-box preconditioners. spaND, on the other hand, solves the problem well and scales near-linearly with the problem size.

4.4. Profiling. Fig. 4.6 shows the (cumulative) memory taken by M in spaND, compared to the direct method. This shows clearly the effect of the approximation. At the beginning, memory increases slowly. Then, we keep eliminating and going up the tree and elimination becomes more and more expansive. The sparsification, however, allows us to greatly decrease the memory use by reducing the separator's sizes. In this specific example, sparsification is skipped for the first four levels. This can be seen on Fig. 4.6, where spaND's level 5 memory use is slightly greater than



n	$N = n^3$	spaND						Direct.
		t_P (s.)	t_F (s.)	t_S (s.)	n_{CG}	$size_{Top}$	mem_F (10^9)	$t_F + t_S$ (s.)
128	2 097 152	7	55	18	13	504	0.62	743
160	4 096 000	18	118	44	14	635	1.2	3677
200	8 000 000	40	254	102	16	962	2.5	—
252	16 003 008	87	650	256	14	891	5.0	—

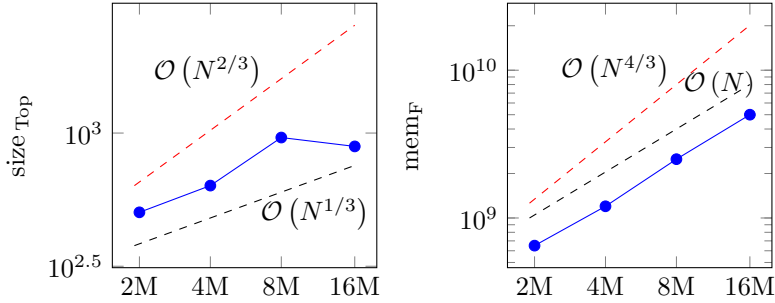


Table 4.3: SPE results. Regular geometric partitioning, $\varepsilon = 10^{-2}$, **OrthS**. “—” indicates the direct method ran out of memory. This problem is a regular cube, and hence very hard to solve using a direct method, since the separators are very large. spaND does not suffer from this problem and can solve this problem well, with a near (but not exactly) linear scaling with the problem size.

the Direct method. After that, however, it remains almost constant.

Fig. 4.7 shows profiling (traces) when solving a larger 16M SPE problem. This clearly shows the advantage of the algorithm. When using a direct method, elimination becomes excessively slow when reaching the top of the tree, and the time spent at the last level usually dominates. For instance in this specific problem, the last elimination would require factoring a matrix of size approximately $252^2 \times 252^2 = 63\,504 \times 63\,504$ (approx. 32GB!) that is completely dense. Our algorithm, on the other hand, spends more time at the early levels in the tree eliminating dofs and sparsifying separators (see the large brown bar at level 5). As a result, the time actually *decreases* as we reach higher levels in the tree. This makes for a much more efficient solver.

Notice that in this example, we start the sparsification at level 5 (i.e., we skip it for four levels). In our experiments, this gives the best results. Starting earlier leads to very high ranks (i.e., there is not much to compress), while delaying it too much leads to too large matrices A_{pn} for which RRQR becomes excessively slow.

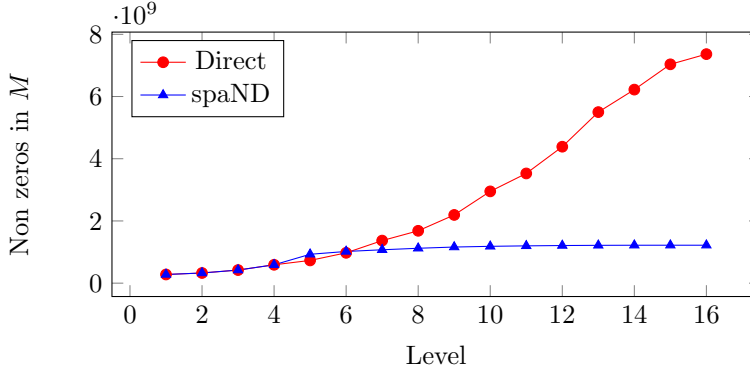


Fig. 4.6: Memory profiling of the SPE 4M problem. Each dot shows the total (cumulative) memory used by the partial preconditioner up to this level in the elimination. We compare spaND to a direct method using Nested Dissection. Thanks to the sparsification (started at level 5), the memory stays well under control, while a direct method takes more and more memory as the elimination proceeds.

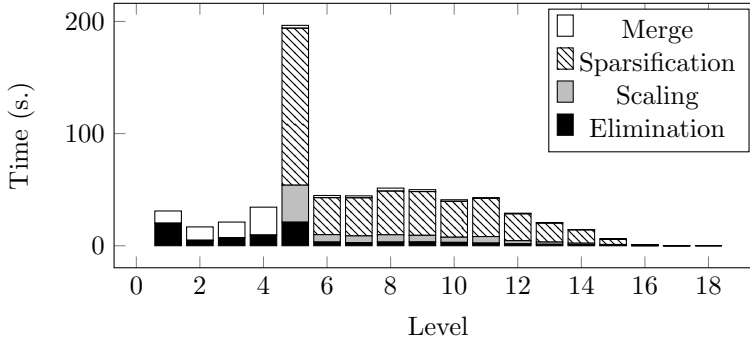


Fig. 4.7: Time profiling of the SPE 16M problem. Each bar represents the time spent at each level in the elimination. Unlike direct methods, most of the compute time is spent at the first levels (near the leaves), where we have to solve many small problems. A direct method would likely be faster at the beginning, but much slower near the end, where the fronts become very large and have to be factored exactly. Sparsification time spikes at level 5 when it is triggered. Starting sparsification sooner is inefficient since the blocks are not low-rank enough, and the time spent in the low-rank factorizations is then wasted.

5. Conclusion. In this paper we developed a sparsified Nested Dissection algorithm. The algorithm combines ideas from Nested Dissection (a fast direct method) and low-rank approximations to reduce the separator sizes. The result is an approximate factorization that can be computed in near-linear time and results in an efficient preconditioner.

We note that it differs from the “classical” way of accelerating sparse direct solvers (like MUMPS with BLR and Pastix with HODLR). Instead of using \mathcal{H} -algebra to compress large fronts, it simply keeps the fronts small throughout the algorithm by

sparsifying them at each step of the algorithm.

Prior work in this area included the HIF algorithm [37]. While our work resembles it, HIF is limited to $n \times n \times n$ regular problems [37] and does not use either the block diagonal scaling or orthogonal transformations. The LoRaSp algorithm [43] is also similar. LoRaSp’s performances however may degrade when the ranks at the leaf level are not small and does not have the same sparsity guarantees [16]. The ordering and the ability to skip compression for some levels fixes this.

We discuss three variants of the algorithm, depending on the low-rank approximation methods (interpolative or orthogonal) and the prior use, or not, of scaling. We showed through extensive numerical experiments that the scaling has a large impact on the preconditioner’s accuracy. In addition, the use of orthogonal transformation implies that the algorithm does not break down even when $\varepsilon \approx 1$.

We then tested the algorithm on both ill-conditioned problems (typically hard for preconditioners) and “cubic” problems (typically hard for direct methods). On these problems, spaND is very efficient, with very favorable scaling for the factorization and near-constant CG iteration count.

Multiple research directions remain unexplored. The compression algorithm used was a simple (but still quite expensive) RRQR algorithm. Other fast algorithms could be used, like randomized methods or skeletonized interpolation (where the c of the interpolative factorization are picked a-priori using some heuristic). These techniques could greatly accelerate the compression step. The loss of accuracy remains to be studied.

Expanding the algorithm to non-SPD or non-symmetric systems is conceptually straightforward. If A is not SPD, one can simply use the LDL^T factorization in the elimination step. Note that in this case, (symmetric) pivoting may be required and the algorithm may break down. If A is not symmetric, we need to compress A_{pn} and A_{np} and use the obtained basis on both the left and the right. The resulting preconditioner can then be coupled with GMRES instead of CG.

The partitioning algorithm is well-suited for matrices arising from the discretization of elliptic PDE’s, where we know that well-separated clusters have low-numerical rank. It would be interesting to explore other partitioning algorithms, for instance for indefinite matrices coming from Maxwell’s equations.

Finally, we mention that spaND exhibits more parallelism than direct methods. Indeed, most of the work occurs near the leaves of the tree. This means less synchronization and more parallelism. This is in contrast with direct methods based on Nested Dissection where the bottleneck is usually the factorization of the top separator at the root of the tree.

Acknowledgement. Some of the computing for this project was performed on the Sherlock cluster. We would like to thank Stanford University and the Stanford Research Computing Center for providing computational resources and support that contributed to these research results.

This work was partly funded by the U.S. Department of Energy National Nuclear Security Administration under Award Number DE-NA0002373-1 and partly funded by an LDRD research grant from Sandia National Laboratories. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-NA-0003525.

We thank Cindy Orozco for the idea of the modified Nested Dissection algorithm,

and Bazyli Klockiewicz for his insightful comments on the numerical algorithms.

REFERENCES

- [1] S. AMBIKASARAN, *Fast algorithms for dense numerical linear algebra and applications*, PhD thesis, Stanford University, 2013.
- [2] P. AMESTOY, C. ASHCRAFT, O. BOITEAU, A. BUTTARI, J.-Y. L'EXCELLENT, AND C. WEISBECKER, *Improving multifrontal methods by means of block low-rank representations*, SIAM Journal on Scientific Computing, 37 (2015), pp. A1451–A1474.
- [3] P. R. AMESTOY, I. S. DUFF, J.-Y. L'EXCELLENT, AND J. KOSTER, *MUMPS: a general purpose distributed memory sparse solver*, in International Workshop on Applied Parallel Computing, Springer, 2000, pp. 121–130.
- [4] E. ANDERSON, Z. BAI, C. BISCHOF, L. S. BLACKFORD, J. DEMMEL, J. DONGARRA, J. DU CROZ, A. GREENBAUM, S. HAMMARLING, A. MCKENNEY, ET AL., *LAPACK Users' guide*, SIAM, 1999.
- [5] J. BARNES AND P. HUT, *A hierarchical $\mathcal{O}(N \log N)$ force-calculation algorithm*, Nature, 324 (1986), p. 446.
- [6] M. BEBENDORF, *Efficient inversion of the galerkin matrix of general second-order elliptic operators with nonsmooth coefficients*, Mathematics of Computation, 74 (2005), pp. 1179–1199.
- [7] M. BEBENDORF AND W. HACKBUSCH, *Existence of \mathcal{H} -matrix approximants to the inverse FE-matrix of elliptic operators with L_∞ -coefficients*, Numerische Mathematik, 95 (2003), pp. 1–28.
- [8] M. J. BERGER AND S. H. BOKHARI, *A partitioning strategy for nonuniform problems on multiprocessors*, IEEE Transactions on Computers, (1987), pp. 570–580.
- [9] J. H. BRAMBLE, *Multigrid methods*, vol. 294, CRC Press, 1993.
- [10] A. BRANDT, S. MCCORMICK, AND J. RUGE, *Algebraic multigrid (AMG) for automatic multigrid solutions with application to geodetic computations*, Report, Inst. for computational Studies, Fort Collins, Colorado, (1982).
- [11] T. F. CHAN, *Rank revealing qr factorizations*, Linear Algebra and its Applications, 88 (1987), pp. 67–82.
- [12] S. CHANDRASEKARAN, P. DEWILDE, M. GU, T. PALS, X. SUN, A.-J. VAN DER VEEN, AND D. WHITE, *Some fast algorithms for sequentially semiseparable representations*, SIAM Journal on Matrix Analysis and Applications, 27 (2005), pp. 341–364.
- [13] S. CHANDRASEKARAN, P. DEWILDE, M. GU, AND N. SOMASUNDERAM, *On the numerical rank of the off-diagonal blocks of schur complements of discretized elliptic PDEs*, SIAM Journal on Matrix Analysis and Applications, 31 (2010), pp. 2261–2290.
- [14] S. CHANDRASEKARAN, M. GU, AND T. PALS, *Fast and stable algorithms for hierarchically semiseparable representations*, Submitted for publication, (2004).
- [15] S. CHANDRASEKARAN, M. GU, AND T. PALS, *A fast ULV decomposition solver for hierarchically semiseparable representations*, SIAM Journal on Matrix Analysis and Applications, 28 (2006), pp. 603–622.
- [16] C. CHEN, L. CAMBIER, E. G. BOMAN, S. RAJAMANICKAM, R. S. TUMINARO, AND E. DARVE, *A robust hierarchical solver for ill-conditioned systems with applications to ice sheet modeling*, Journal of Computational Physics, 396 (2019), pp. 819–836.
- [17] Y. CHEN, T. A. DAVIS, W. W. HAGER, AND S. RAJAMANICKAM, *Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate*, ACM Transactions on Mathematical Software (TOMS), 35 (2008), p. 22.
- [18] H. CHENG, Z. GIMBUTAS, P.-G. MARTINSSON, AND V. ROKHLIN, *On the compression of low rank matrices*, SIAM Journal on Scientific Computing, 26 (2005), pp. 1389–1404.
- [19] M. CHRISTIE, M. BLUNT, ET AL., *Tenth SPE comparative solution project: A comparison of upscaling techniques*, in SPE reservoir simulation symposium, Society of Petroleum Engineers, 2001.
- [20] T. A. DAVIS AND Y. HU, *The University of Florida sparse matrix collection*, ACM Transactions on Mathematical Software (TOMS), 38 (2011), p. 1.
- [21] M. FAVERGE, G. PICHON, P. RAMET, AND J. ROMAN, *On the use of \mathcal{H} -matrix arithmetic in PaStiX: a preliminary study*, in Workshop on Fast Direct Solvers, Toulouse, France, June 2015, <https://hal.inria.fr/hal-01187882>.
- [22] R. P. FEDORENKO, *A relaxation method for solving elliptic difference equations*, Computational Mathematics and Mathematical Physics, 1 (1962), pp. 1092–1096.
- [23] J. FELIU-FABÀ, K. L. HO, AND L. YING, *Recursively preconditioned hierarchical interpolative factorization for elliptic partial differential equations*, arXiv preprint arXiv:1808.01364,

- (2018).
- [24] W. FONG AND E. DARVE, *The black-box fast multipole method*, Journal of Computational Physics, 228 (2009), pp. 8712–8725.
 - [25] A. GEORGE, *Nested dissection of a regular finite element mesh*, SIAM Journal on Numerical Analysis, 10 (1973), pp. 345–363.
 - [26] P. GHYSSELS, X. S. LI, F.-H. ROUET, S. WILLIAMS, AND A. NAPOV, *An efficient multicore implementation of a novel hss-structured multifrontal solver using randomized sampling*, SIAM Journal on Scientific Computing, 38 (2016), pp. S358–S384.
 - [27] G. H. GOLUB AND C. F. VAN LOAN, *Matrix computations*, vol. 4, JHU Press, 2013.
 - [28] L. GRASEDYCK, R. KRIEMANN, AND S. LE BORNE, *Domain decomposition based \mathcal{H} -LU preconditioning*, Numerische Mathematik, 112 (2009), pp. 565–600.
 - [29] L. GREENGARD AND V. ROKHLIN, *A fast algorithm for particle simulations*, Journal of Computational Physics, 73 (1987), pp. 325–348, [https://doi.org/10.1016/0021-9991\(87\)90140-9](https://doi.org/10.1016/0021-9991(87)90140-9).
 - [30] M. GU AND S. C. EISENSTAT, *Efficient algorithms for computing a strong rank-revealing qr factorization*, SIAM Journal on Scientific Computing, 17 (1996), pp. 848–869.
 - [31] W. HACKBUSCH, *A Sparse Matrix Arithmetic Based on \mathcal{H} -Matrices. Part I: Introduction to \mathcal{H} -Matrices*, Computing, 62 (1999), pp. 89–108.
 - [32] W. HACKBUSCH, *Multi-grid methods and applications*, vol. 4, Springer Science & Business Media, 2013.
 - [33] W. HACKBUSCH AND S. BÖRM, *Data-sparse approximation by adaptive \mathcal{H}^2 -matrices*, Computing, 69 (2002), pp. 1–35.
 - [34] W. HACKBUSCH AND S. BÖRM, *\mathcal{H}^2 -matrix approximation of integral operators by interpolation*, Applied Numerical Mathematics, 43 (2002), pp. 129–143.
 - [35] P. HÉNON, P. RAMET, AND J. ROMAN, *Pastix: a high-performance parallel direct solver for sparse symmetric positive definite systems*, Parallel Computing, 28 (2002), pp. 301–321.
 - [36] M. R. HESTENES AND E. STIEFEL, *Methods of conjugate gradients for solving linear systems*, vol. 49, NBS Washington, DC, 1952.
 - [37] K. L. HO AND L. YING, *Hierarchical interpolative factorization for elliptic operators: differential equations*, Communications on Pure and Applied Mathematics, 69 (2016), pp. 1415–1451.
 - [38] G. KARYPIS AND V. KUMAR, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM Journal on scientific Computing, 20 (1998), pp. 359–392.
 - [39] E. LIBERTY, F. WOOLFE, P.-G. MARTINSSON, V. ROKHLIN, AND M. TYGERT, *Randomized algorithms for the low-rank approximation of matrices*, Proceedings of the National Academy of Sciences, 104 (2007), pp. 20167–20172.
 - [40] R. J. LIPTON, D. J. ROSE, AND R. E. TARJAN, *Generalized nested dissection*, SIAM Journal on Numerical Analysis, 16 (1979), pp. 346–358.
 - [41] L. MIRANIAN AND M. GU, *Strong rank revealing lu factorizations*, Linear Algebra and its Applications, 367 (2003), pp. 1–16.
 - [42] C. C. PAIGE AND M. A. SAUNDERS, *Solution of sparse indefinite systems of linear equations*, SIAM Journal on Numerical Analysis, 12 (1975), pp. 617–629.
 - [43] H. POURANSARI, P. COULIER, AND E. DARVE, *Fast hierarchical solvers for sparse matrices using extended sparsification and low-rank approximation*, SIAM Journal on Scientific Computing, 39 (2017), pp. A797–A830.
 - [44] A. PROKOPENKO, C. M. SIEFERT, J. J. HU, M. HOEMMEN, AND A. KLINVEX, *Ifpack2 Users Guide 1.0*, Tech. Report SAND2016-5338, Sandia National Labs, 2016.
 - [45] Y. SAAD, *ILUT: A dual threshold incomplete LU factorization*, Numerical Linear Algebra with Applications, 1 (1994), pp. 387–402.
 - [46] Y. SAAD, *Iterative methods for sparse linear systems*, vol. 82, SIAM, 2003.
 - [47] Y. SAAD AND M. H. SCHULTZ, *GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing, 7 (1986), pp. 856–869.
 - [48] Y. SAAD AND J. ZHANG, *BILUM: block versions of multielimination and multilevel ILU preconditioner for general sparse linear systems*, SIAM Journal on Scientific Computing, 20 (1999), pp. 2103–2121.
 - [49] P. G. SCHMITZ AND L. YING, *A fast direct solver for elliptic problems on general meshes in 2d*, Journal of Computational Physics, 231 (2012), pp. 1314–1338.
 - [50] K. STÜBEN, *A review of algebraic multigrid*, in Partial Differential Equations, Elsevier, 2001, pp. 281–309.
 - [51] D. A. SUSHNIKOVA AND I. V. OSELEDETS, *“Compress and Eliminate” solver for symmetric positive definite sparse matrices*, SIAM Journal on Scientific Computing, 40 (2018), pp. A1742–A1762.

- [52] I. K. TEZAUR, M. PEREGO, A. G. SALINGER, R. S. TUMINARO, AND S. F. PRICE, *Albany/FELIX: a parallel, scalable and robust, finite element, first-order Stokes approximation ice sheet solver built for advanced analysis*, Geoscientific Model Development (Online), 8 (2015).
- [53] H. A. VAN DER VORST, *Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems*, SIAM Journal on Scientific and Statistical Computing, 13 (1992), pp. 631–644.
- [54] J. XIA, *Efficient structured multifrontal factorization for general large sparse matrices*, SIAM Journal on Scientific Computing, 35 (2013), pp. A832–A860.
- [55] J. XIA, *Randomized sparse direct solvers*, SIAM Journal on Matrix Analysis and Applications, 34 (2013), pp. 197–227.
- [56] J. XIA, S. CHANDRASEKARAN, M. GU, AND X. S. LI, *Superfast multifrontal method for large structured linear systems of equations*, SIAM Journal on Matrix Analysis and Applications, 31 (2009), pp. 1382–1411.
- [57] J. XIA, S. CHANDRASEKARAN, M. GU, AND X. S. LI, *Fast algorithms for hierarchically semiseparable matrices*, Numerical Linear Algebra with Applications, 17 (2010), pp. 953–976.
- [58] J. XIA AND M. GU, *Robust approximate cholesky factorization of rank-structured symmetric positive definite matrices*, SIAM Journal on Matrix Analysis and Applications, 31 (2010), pp. 2899–2920.
- [59] J. Z. XIA, *Effective and robust preconditioning of general spd matrices via structured incomplete factorization*, SIAM Journal on Matrix Analysis and Applications, 38 (2017), pp. 1298–1322.
- [60] K. YANG, H. POURANSARI, AND E. DARVE, *Sparse hierarchical solvers with guaranteed convergence*, arXiv preprint arXiv:1611.03189, (2016).