# FAST AND SCALABLE HIERARCHICAL LINEAR SOLVERS

A DISSERTATION
SUBMITTED TO THE INSTITUTE FOR COMPUTATIONAL AND
MATHEMATICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Léopold Cambier
November 2020

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

**Eric Darve, Primary Adviser**

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

**Juan Alonso**

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

**Erik Boman**

Approved for the Stanford University Committee on Graduate Studies.

**Stacey F. Bent, Vice Provost for Graduate Education**

*This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.*

# Abstract

Linear solvers are a key component of scientific computing. In Chapter 2 we develop a new algorithm for the fast solution of large, sparse linear systems, spaND (sparsified Nested Dissection). spaND uses low-rank approximations to reduce the size of the Nested Dissection separators without creating any fill-in. The result is an approximate factorization that can be used as an efficient preconditioner. Unlike many preconditioners, spaND works on a large class of matrices. We 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 the previously developed Hierarchical Interpolative Factorization algorithm on nearly all SPD problems from the SuiteSparse matrix collection. Furthermore, we prove that this algorithm is guaranteed to never break down if the input matrix is SPD. We evaluate the algorithm on some large problems, both SPD and unsymmetric, and show that it exhibits near-linear scalings.

In Chapter 3, we tackle the problem of task-based parallel computing. Runtime systems, where the user expresses computations as tasks with inputs and outputs, offer a solution to programming increasingly large and complex modern computers. However, their adoption has been so far very limited. In this work, we present TaskTorrent, a lightweight distributed task-based runtime in C++. TaskTorrent uses a parametrized task graph to express the task DAG, and one-sided active messages to trigger remote tasks asynchronously. TaskTorrent is very easy to learn, easy to integrate into existing codes, and has minimal overhead. We explain the API and the implementation. We perform a series of benchmarks against StarPU, Regent and ScaLAPACK. Both TaskTorrent and StarPU outperform ScaLAPACK, and TaskTorrent exhibits good strong and weak scalings. We then combine TaskTorrent with spaND in Chapter 4, explain the implementation, and run the resulting algorithm on a few large problems, efficiently using up to 9000 cores. This shows that spaND scales very well when the ranks grow slowly with the problem size and that TaskTorrent has no

problems exploring very large DAGs.

In Chapter 5, we study the problem of kernel matrix factorization. In this work, we develop the Skeletonized Interpolation algorithm, a new way to build a low-rank factorization of kernel matrices. This is done by first sampling the kernel function at new interpolation points, then selecting a subset of those using a CUR decomposition, and finally using this reduced set of points as pivots for a rank-revealing LU-type factorization. We explain how this implicitly builds an optimal interpolation basis for the kernel under consideration and show the asymptotic convergence of the algorithm. Finally, we demonstrate on numerical examples that it performs very well in practice, allowing us to obtain ranks nearly equal to the optimal rank at a fraction of the cost of the naive algorithm.

Finally, in Chapter 6, we study a parametrized least-square problem $\operatorname{argmin}_{x\in\mathcal{S}} \|(A+\omega I)^{-1/2}(Ax-b)\|_2$ for $A = A^*$, $\omega > -\lambda_{\min}(A)$ and $\mathcal{S}$ a subspace. We show that the solutions $x_{b,\omega}$ belong to a low-dimensional subspace, independent from $b$ and $\omega$, of dimension $\mathbf{Ind}_A(\mathcal{S}) = \dim(\mathcal{S} + A\mathcal{S}) - \dim(\mathcal{S})$, a quantity we call the index of invariance. This result implies the low-dimensionality result from Hallman & Gu (2018). We furthermore study the tightness of the bound, and show that $\{x_{b,\omega} - x_{b,\mu} \mid \omega, \mu\}$ can have dimension 1 for all $b$ even if $\mathbf{Ind}_A(\mathcal{S}) > 1$. Finally, we exhibit sufficient conditions on $A$ (such as $A$ being SPD) such that $\{x_{b,\omega} - x_{b,\mu} \mid b, \omega, \mu\}$ has dimension $\mathbf{Ind}_A(\mathcal{S})$.

# Acknowledgments

First and foremost, I want to thank my advisor, Prof. Eric Darve. Prof. Darve has always put a lot of trust in me, letting me explore any direction. He has taught me about research, academic writing, and given me much relevant advice regarding my professional career. None of this would have been possible without him. I also want to thank Erik G. Boman and Prof. Juan J. Alonso for taking the time to read this manuscript and for all their recommendations, as well as Prof. Lexing Ying and Prof. Michael Saunders for being part of my oral committee.

My experience at Stanford would have been very different without the numerous friends I made during my time here, friends who all played some role in this work. Thank you all for being so great. In particular, thank you to my wonderful wife Cindy who I also met during my time at Stanford. She always showed tremendous support, provided me with many interesting ideas, and made everything worth it. Many thanks to my parents, Edouard and Frédérique, for always being behind me, supporting me, and being available to talk with me when I needed it.

I also want to thank all the ICME staff. In particular, Prof. Margot Gerritsen and Prof. Gianluca Iaccarino for being tremendous directors, and Indira Choudhury for running ICME so smoothly in the background. Finally, thank you to Total and all the affiliated people for providing me so many opportunities through the Total fellowship.

**Note on shared authorship** Chapter 2, Chapter 3 and Chapter 5 present work performed by the author in collaboration with other people [33, 32, 34]. Whenever appropriate, shared authorship is clearly indicated. Chapter 6 and Appendix A present shared work between the author and Rahul Sarkar [35]. Both authors contributed equally to that work, and this chapter presents select sections of this paper. Some of the text was originally written by Rahul Sarkar.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Scientific computing is a field of science combining mathematical models and computer simulations to analyze or predict natural phenomena. This encompasses many disciplines, such as finance, weather predictions, computational biology, computational fluid dynamics, and many more.

**Sparse linear systems** At the core of many of those applications, one can often find a linear system such as

$$Ax = b \tag{1.1}$$

(where $A \in \mathbb{R}^{N \times N}$ and $b \in \mathbb{R}^N$ are given and $x \in \mathbb{R}^N$ is the unknown). A frequent reason is that the model requires solving a partial differential equation (PDE). For instance, in its simplest form, one can model the diffusion of heat by solving $(\nabla \cdot (\nabla u))(x) = f(x)$ for all $x \in \Omega$ with suitable boundary conditions. This is an example of an elliptic PDE. One way to solve this is to discretize it using finite differences or finite-elements [105]. This results in a linear system such as (1.1) where $A$ is sparse, symmetric, and positive definite (SPD). Many other PDEs are frequently encountered in practice. For instance, in Section 2.4 we consider a Stokes formulation for the movement of ice in Antarctica (an ice-sheet model) discretized using a finite-element method. Those equations are non-linear because of a modeling assumption on the viscosity. Solving those equations using Newton's method leads to a sequence of sparse linear systems which are SPD. In Section 4.3 we study similar equations in the context of a subsonic flow around an airfoil. Those non-linear equations lead to a sequence of non-symmetric linear systems. In addition to PDEs, linear systems

also occur in inverse problems [141], optimization [115], ordinary differential equations and dynamical systems [105], graph theory [48], etc. Solving sparse linear systems is at the core of most scientific applications. In this thesis, we focus on linear systems coming from PDEs since they exhibit specific properties that allow for fast algorithms. Solving them often takes most of the overall simulation time, especially on modern machines [10]. As such, this requires efficient algorithms.

Much effort has been dedicated to finding efficient algorithms to solve (1.1) when $A$ is large and sparse. Linear solvers can roughly be divided into three categories (see Figure 1.1):

1. Sparse direct methods (Figure 1.1, left), such as the sparse LU or sparse Cholesky [52], with a suitable ordering such as Nested Dissection (see Section 2.2.1). These methods are attractive because they can factor almost any matrix exactly into a product of triangular $L$ and $U$ such that $A = LU$. Solving (1.1) can then be done efficiently through forward and backward substitution. However, when $A \in \mathbb{R}^{N \times N}$ comes from a 3D PDE discretized with a local stencil, fill-in can be significant. This usually leads to a $\mathcal{O}(N^2)$ time complexity, making them prohibitively expensive even for moderate $N$.

2. Iterative methods, such as CG [91], MINRES [118], or GMRES [132], combined with a problem-dependent preconditioner (such as Multigrid [28, 29, 143] for elliptic PDEs — Figure 1.1, right). Iterative methods are attractive because they only require matrix-vector products $Av$. However, when $A$ is ill-conditioned, as it is often the case in practice, they converge very slowly. Hence, a problem-dependent preconditioner is needed to keep the iteration count low. The difficulty with this approach is that it requires domain knowledge to design an efficient preconditioner, and what works on a particular problem may not work on a different but similar one.

3. Incomplete or approximate factorizations (Figure 1.1, center). This approach consists of computing an approximate factorization of $A$. For instance, incomplete LU (ILU, [133]) computes $A \approx LU$ using a classical LU algorithm but drops some of the fill-ins to not have more than $k$ (for some $k$) entries per row or column of $L$ and $U$. The approximate factorization can then be used as a preconditioner within an iterative method. Incomplete factorizations are appealing because they don't rely on an expert user and can in theory be applied to any matrix. However, methods such as ILU, when used as a preconditioner, typically lead to an iteration count that grows significantly

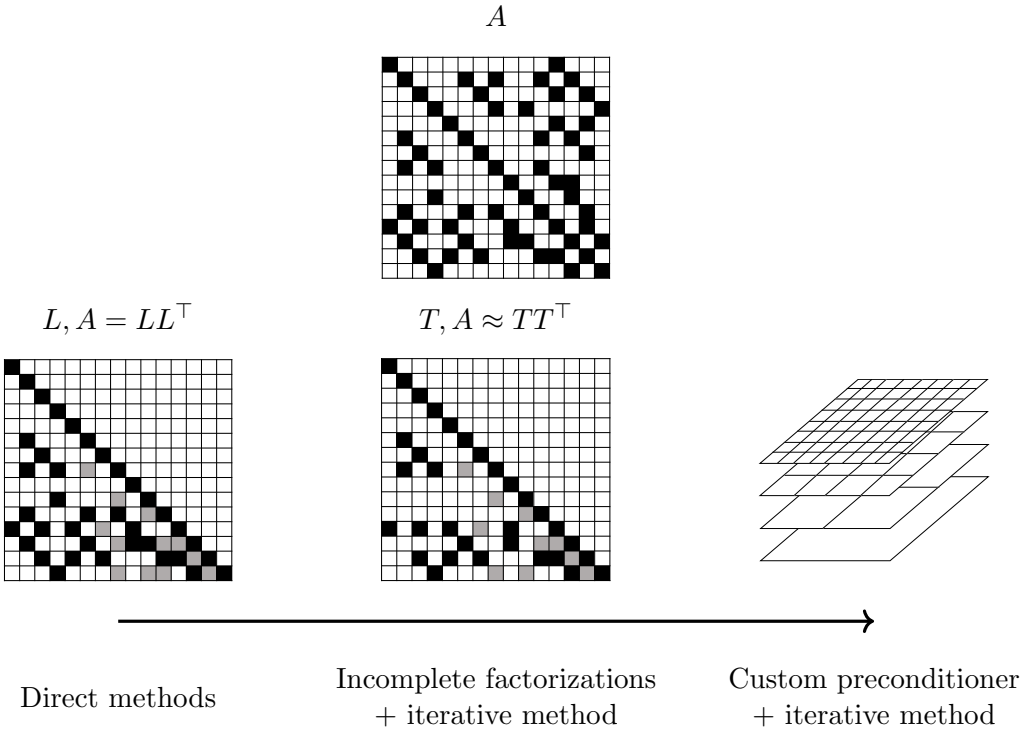



Figure 1.1: Various methods to solve linear systems. We here assume the matrix is SPD. Left: direct methods, that compute $L$ such that $A = LL^\top$. Center: incomplete factorizations, similar to direct methods but with appropriate approximations to make the algorithm faster. Computes $L$ such that $A \approx LL^\top$. Usually combined with an iterative method. Right: custom preconditioners (Multigrid depicted), combined with an iterative method. Usually very problem dependent.

with $N$ and, as such, do not scale well with problem size.

In this work, we follow the third approach. The goal is to design an approximate factorization $M$ of $A$ such that $M \approx A$ and where $M$ is sparse, fast to compute, and fast to invert (i.e., solving $Mx = b$ should be fast). By fast we refer to $\mathcal{O}(N)$ or $\mathcal{O}(N \log N)$ time complexities. The approximation should also be accurate enough to lead to a low and slowly-growing iteration count when combined with an iterative method such as CG or GMRES.

To do so, we rely on the properties of the underlying PDE. By using appropriate low-rank approximations, we directly eliminate, without creating *any fill-ins*, many unknowns in the system. The algorithm does not use any particular hierarchical matrix format (see Section 2.1.2) and is a conceptually simple but fast algorithm. We call this algorithm spaND

(Sparsified Nested Dissection) and explain it in detail in Chapter 2. In Chapter 4 we study the parallel properties of the algorithm when combined with a task-based runtime system.

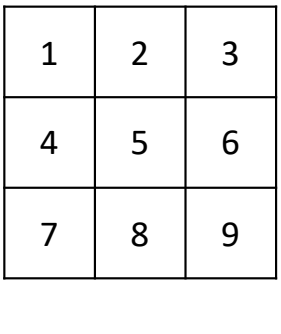
**Dense linear systems** While many linear systems are sparse, dense linear systems do also arise. For instance, the equation

$$\phi(x) = \frac{\mu}{4\pi} \int_{\mathcal{Y}} \mathscr{K}(x,y)k(y)\mathrm{d}y \text{ for all } x \in \mathcal{X}, \tag{1.2}$$

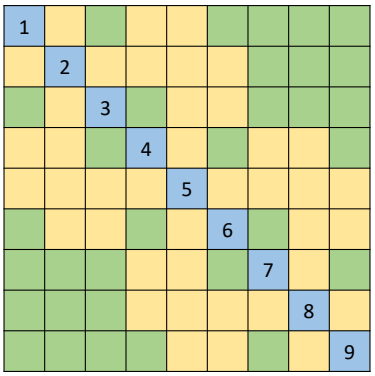
with $\mathscr{K}(x,y) = 1/\|x-y\|_2$ (the kernel function) is a classical boundary integral equation arising in electromagnetics [104]. Similar integral equations also arise in acoustics [152] and other applications. Those equations may also involve $\mathscr{K}(x,y) = \log(\|x-y\|_2)$, $\mathscr{K}(x,y) = 1/\|x-y\|_2^2$ or other kernel functions. Given the infinite support of $\mathscr{K}$ and the integral, discretization of (1.2) leads to linear systems $Au = b$ where $A \in \mathbb{R}^{N\times N}$ is *dense*. Upon discretization we have $A_{ij} = \mathscr{K}(x_i, y_j)$ for $x_i \in \mathcal{X}$ and $y_j \in \mathcal{Y}$. However, while dense, this matrix has a lot of structure. For properly chosen subsets $X \subseteq \mathcal{X}$ and $Y \subseteq \mathcal{Y}$, the corresponding block $K$ of $A$ has a low numerical rank.

This low-rank property leads to efficient data-sparse representations of $A$. Figure 1.2a shows a clustering of a square mesh $\mathcal{X}$, in which each square indicates a cluster of points. Figure 1.2b shows $A$, in which points belonging to the same cluster are ordered consecutively in the matrix. Each block corresponds to the interactions between a pair of clusters of $\mathcal{X}$. Blocks in green correspond to interactions between well-separated clusters and are low-rank. Blocks in yellow and blue are not. We see that the low-rank property lets us store the matrix at a reduced cost since each block of size $n \times n$ and rank $r$ can be stored using $\mathcal{O}(nr)$ memory instead of $\mathcal{O}\left(n^2\right)$. This also allows for faster factorizations (such as LU) or matrix-vector computations. Finally, note that other more complex representations are possible. Using hierarchical matrices ($\mathcal{H}$-matrices) leads to a matrix like in Figure 1.2b but where not all blocks have the same size.

Regardless of the format used, all those data-sparse representations require the low-rank blocks to be first factored into a low-rank form. This is the focus of this chapter. To simplify the discussion, let us focus on a specific block $K \in \mathbb{R}^{n\times n}$ of rank $r$ of $A$, defined as $K_{ij} = \mathscr{K}(x_i, y_j)$ for $x_i \in X$, $y_j \in Y$. We could use the SVD to compute a low-rank approximation of $K$, at a cost of $\mathcal{O}\left(n^2 r\right)$. However, in this work, we seek to avoid the $\mathcal{O}\left(n^2\right)$ time complexity. This implies that we *cannot even build* $K$. To do so, we designed the




(a) A clustering of the mesh $X$ over which (1.2) is discretized.




(b) A block low-rank representation of $A$. Green blocks correspond to interactions between well-separated clusters and are low-rank.

Skeletonized Interpolation (SI) algorithm, which can build a low-rank approximation of $K$ with $\mathcal{O}(nr)$ time complexity. SI is completely independent of the actual point distributions within the clusters associated with $K$ and only relies on their geometry. Chapter 5 explains in detail the algorithm, includes a proof of correctness under some assumptions, and finally contrasts it to existing approaches, such as ACA, demonstrating its improved robustness.

**Iterative methods** Solving sparse linear systems often requires combining multiple techniques. Algorithms like spaND can be used to approximately solve $Ax = b$, by computing an approximate factorization $A \approx LU$ and then solving $LUx = b$. But iterative methods need to be used to fully solve the system. Those algorithms repeatedly solve related problems where $x$ belongs to a larger and larger subspace, hence computing an increasingly accurate solution to (1.1). Various methods exist. CG [91] and MINRES [118] are amongst the most popular methods when $A$ is symmetric.

In Chapter 6 we perform a theoretical study of a family of iterative algorithms —a family which includes CG and MINRES, showing that the solutions belong to a smaller space than what can normally be expected. More precisely, for a particular $A \in \mathbb{F}^{n\times n}$, $b \in \mathbb{F}^n$, and $\mathcal{S} \subseteq \mathbb{F}^n$ a subspace (with $\mathbb{F} = \mathbb{R}$ or $\mathbb{C}$), we consider the problem

$$x_{b,\omega} := \operatorname{argmin}_{x\in S} \|(A + \omega I)^{-1/2}(Ax - b)\|_2 \tag{1.3}$$

with $\omega > \omega_{\min} := -\lambda_{\min}(A)$. Note that if $\mathcal{S} = K(A, b, k)$ is the $k^{\text{th}}$ Krylov subspace, as

$\omega = 0$, one recovers the CG iterate, and as $\omega \to \infty$, one recovers MINRES.

We study the iterates $x_{b,\omega}$ in detail and show that they actually belong not only to $\mathcal{S}$, as expected, but also to a low-dimensional subspace $\mathcal{Y}$, independent from $b$ and $\omega$, with $\dim(\mathcal{Y}) \leq \mathbf{Ind}_A(\mathcal{S}) := \dim(\mathcal{S} + A\mathcal{S}) - \dim(\mathcal{S})$, a quantity we call the index of invariance of a matrix $A$ with respect to a subspace $\mathcal{S}$. This result generalizes and explains a surprising result in [89], where the authors studied (1.3) when $\mathcal{S} = K(A, b, k)$ and where they found that the iterates belong to a one-dimensional subspace. Our result goes further, by generalizing the subspace into consideration. Finally, we also study the converse, i.e., when do the iterates fill the space. We show that it does not happen in general, but show sufficient conditions under which $\{x_{b,\omega} \mid b, \omega > \omega_{\min}\}$ fills the space. This result could be used to efficiently solve related problems, such as weighted least squares problems with linear constraints.

**Parallel Computing** Finally, efficiently exploiting computer resources is critical to solving large problems. It is now well known that CPU clocks have stopped increasing according to Moore's law [114] for many years. As such, programmers and scientists cannot rely on forever-increasing single-threaded performance to run larger and larger problems. The only way to run large computations is then to leverage parallel machines with many cores and other computing units (GPUs, etc.). Those can be on one physical machine or distributed across an entire cluster. Efficiently using all those available resources is the topic of parallel computing.

Parallel computing is not new. MPI 1.0 was formally published in 1994 [66], to program parallel algorithms using a distributed memory model. Similarly, the OpenMP 1.0 Fortran API specification was published in 1997 [21], to program multi-core shared-memory machines. However, both methods typically lead to fork-join (for OpenMP) or bulk-synchronous (for MPI) algorithms and programs. This typically creates many unnecessary synchronization points, where the algorithm is artificially waiting.

In addition, modern machines are becoming increasingly complex. They have deep memory hierarchies, from various levels of caches to separate NUMA nodes. Computing clusters have many nodes, equipped with CPUs with an increasing number of cores. Finally, accelerators (such as GPUs) are also becoming more and more common, supplementing multicore CPUs. As such, it is becoming increasingly complex to program algorithms for such architectures.

Runtime systems propose one solution to those problems. Instead of a fork-join or bulk-synchronous approach, computations are expressed as small tasks, with dependencies between them. The runtime system is then in charge of scheduling those tasks on the machine, avoiding synchronization altogether, and exploiting available resources as much as possible. Some runtime systems can even automatically generate code to leverage accelerators, such as GPUs, as well.

Many solutions have been proposed in the last 10 to 15 years. Section 3.2 highlights some of them. However, none has yet gained enough popularity to replace something like MPI and/or OpenMP. We believe the following features are often lacking and are required for the wide adoption of new solutions.

- It needs an easy-to-use API, not requiring the user to give away control of data structures and not requiring the user to learn a new programming language.
- It should only use standard tools such as MPI and C++ and be designed to enable incremental adoption in existing codebases. For instance, one should be able to use the runtime only in select portions of the code.
- It should be lightweight, to not lose performance when tasks are short or granularity is not optimal.

To address those requirements, we designed TaskTorrent (`TTor`). `TTor` addresses the above concerns in the following way.

- It has an easy-to-use API, is entirely written in C++, and is compatible with any user data.
- It requires only MPI and C++ and is based on the message-passing paradigm. So it is compatible with most existing codes and can be introduced step by step, in select portions of the code only.
- It uses a Parametrized Task Graph (PTG) approach with Active Messages (AM). As such, the task graph (the DAG) is entirely distributed and explored completely in parallel. Hence the runtime is lightweight and the code scales very well even with small tasks.

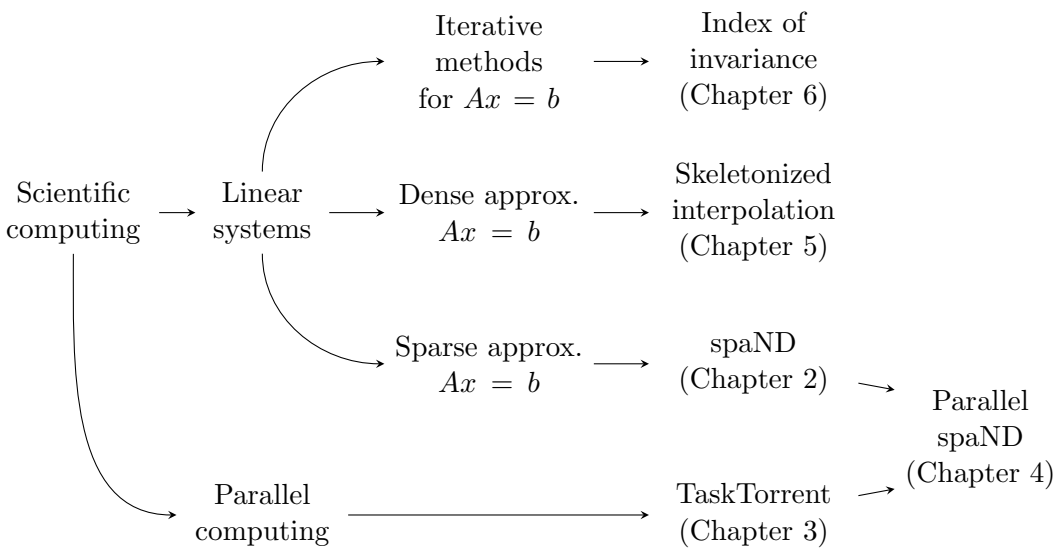



Figure 1.2: The five chapters of this thesis and their connections to each other.

We explain `TTor`'s API and implementation in detail in Chapter 3. We then show that this approach of combining PTG and AMs scales well and is competitive with existing state-of-the-art runtimes such as StarPU and Regent, by comparing them on a few large linear algebra problems. This shows that a simple approach, with a friendly API, can perform well in practice. Finally, we combine TaskTorrent with spaND in Chapter 4. This allows us to study spaND's efficiency on very large problems with up to 300M degrees of freedom.

**Structure of this thesis** It can be seen that this work contains related, but distinct, topics. The central thread is the efficient solution of linear systems, from both an algorithmic (exploiting sparsity and low-rank approximations) and practical (using parallel algorithms) standpoints. Figure 1.2 shows how each topic relates to each other.

# Chapter 2

# Sparsified Nested Dissection



## 2.1 Introduction

We are interested in solving large sparse linear systems

$$Ax = b,\ A \in \mathbb{R}^{N \times N} \tag{2.1}$$

where $A$ is SPD, symmetric, or unsymmetric, but with a symmetric (or near-symmetric) sparsity pattern. In particular, we focus on linear systems with similar properties as those arising from the discretization of 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 Equation (2.1) are traditionally divided into three categories. On one hand are direct methods. The naive Cholesky or LU 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 [109] 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) [71, 109]. This is what most state-of-the-art direct solvers are built upon [44, 8, 90]. 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 [62, 28, 84] (and its algebraic version, [29, 143]) works very well on fairly regular elliptic PDEs, usually with a near-constant iteration count and $\mathcal{O}(N)$ memory 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 [91, 150], MINRES [118] or GMRES [132] 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 an artificially prescribed maximum number of non-zeros in every row & column [133]. Block versions [135] 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 elliptic PDE discretization also typically have low-rank off-diagonal blocks [17, 16, 39]. More precisely, the fill-in arising when factoring the matrix typically has a small numerical rank with a 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 [12, 78, 65]. Matrices built using this property are broadly called Hierarchical ($\mathcal{H}$) matrices [83]. Many formats exist, depending on when and how off-diagonal blocks are compressed into low-rank format. The Hierarchical Off-Diagonal Low Rank (HODLR) [6] format compresses all off-diagonal blocks. If the off-diagonal blocks are compressed using a nested basis, we obtain Hierarchically Semi-Separable (HSS) matrices [37, 38, 40, 157]. Finally, the broader category of $\mathcal{H}^2$ matrices also uses nested basis but only compresses well-separated (i.e., far-field) interactions ([85, 86, 160], [125] with LoRaSp and [144] 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 [77] 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 [156, 138, 154, 155, 73]. The last one was incorporated into the Strumpack package. [7] uses Block Low-Rank approximation to compress the frontal matrices in the MUMPS solver [8]. Finally, [61] studies the use of $\mathcal{H}$-matrices using HODLR in the PaStiX solver [90].

The Hierarchical Interpolative Factorization (HIF) [95] proposes a different approach. Instead of storing the full dense fronts in some low-rank format, it uses low-rank approximations 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 [159] and J. Feliu-Fabà et al. [63] 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.

### 2.1.1 Contribution

Our approach is based on the idea of HIF described in [95]. However, there are several differences, improvements, and novel capabilities:

- Our algorithm is completely general and can be applied to any matrix, either SPD (with strong stability guarantees), symmetric or unsymmetric (in which case, the sparsity pattern is assumed to be symmetric or near-symmetric). 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 [95], 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.

### 2.1.2 Contrast with $\mathcal{H}$-matrix-based algorithms

We emphasize that the HIF approach [95] 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 [7] and PaStiX [61] for instance use fast $\mathcal{H}$-algebra (specifically block low-rank (BLR) matrices) to compress this block [7, 124]. 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 results are large and dense but hierarchically low-rank fronts. In our approach (like in [95]), fronts are kept small throughout the factorization by eliminating unknowns related to low-rank interactions as soon as possible.

## 2.2 Sparsified Nested Dissection

This section describes the algorithm in detail. We call it spaND for Sparsified Nested Dissection. We start by discussing sparse direct methods and Nested Dissection. Then, building upon them, we introduce our algorithm and then detail all the various parts.

### 2.2.1 Sparse direct methods and Nested Dissection

Let us assume $A$ is SPD (this will be relaxed later on). The first approach to solve (2.1) consists of direct methods. Those compute an exact Cholesky factorization of $A$

$$A = LL^\top$$

where $L \in \mathbb{R}^{N\times N}$ is lower triangular.

Since $A$ is sparse, so is $L$. To understand how sparse $L$ is, consider the following. Let us perform a partial Cholesky factorization of $A$ by eliminating the first row and column —degree of freedom (dof). $A$ can then be written as

$$A = \begin{bmatrix} a_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

with $a_{11} \in \mathbb{R}$, $A_{22} \in \mathbb{R}^{N-1\times N-1}$, and with compatible sizes for the other blocks. With $l_{11}^2 = a_{11} > 0$ (because $A$ is SPD), we have

$$\begin{bmatrix} l_{11}^{-1} & \\ -A_{21}a_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} a_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} l_{11}^{-1} & -a_{11}^{-1}A_{12} \\ & I \end{bmatrix} = \begin{bmatrix} I & \\ & A_{22} - A_{21}a_{11}^{-1}A_{12} \end{bmatrix}. \tag{2.2}$$

We say that the first dof has been eliminated. At this point, the algorithm recurses on $B_{22} = A_{22} - A_{21}A_{11}^{-1}A_{12}$, called the Schur complement or *trailing matrix*. Note that if $A$ is SPD, so is $B_{22}$. However, notice that the sparsity pattern of $B_{22}$ is not the same as $A_{22}$ because of the $-A_{21}a_{11}^{-1}A_{12}$ term. In general, it may contain more non-zero entries.

To understand this, let us define $G_A$ to be the graph of $A \in \mathbb{R}^{N\times N}$, $G_A = (V, E)$ with $V = \{1, \ldots, N\}$ and $E = \{(i, j) \mid A_{ij} \neq 0\}$. We see that, ignoring fortuitous cancellations, $B_{22,ij} \neq 0$ either if $A_{22,ij} \neq 0$ or $(A_{21}a_{11}^{-1}A_{12})_{ij} \neq 0$. The first condition corresponds to original non-zero entries in $A$ or existing edges in $G_A$. The second condition can be equivalently formulated as $A_{21,i} \neq 0$ and $A_{21,j} \neq 0$. In $G_A$, this indicates that if vertex 1

has an edge to $1+i$ and $1+j$, then $B_{22,ij} \neq 0$. Those additional non zero entries in $B_{22}$ are called fill-in. This argument can be repeated up to when every vertex has been eliminated. At every step, eliminating a vertex $v$ fills the trailing matrix with edges corresponding to a clique between all of $v$'s non-eliminated neighbors. This argument can be generalized to a group (or cluster) of dofs. Eliminating a set $S$ of vertices creates fill-ins corresponding to a clique involving all of $S$'s non-eliminated neighbors in $A$. (Note that this is formally only an upper-bound.) Assuming $S$ comes first in the matrix, one can write

$$\begin{bmatrix} L_{11}^{-1} & \\ -A_{21}A_{11}^{-1} & I \end{bmatrix} \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} L_{11}^{-1} & -A_{11}^{-1}A_{12} \\ & I \end{bmatrix} = \begin{bmatrix} I & \\ & A_{22} - A_{21}A_{11}^{-1}A_{12} \end{bmatrix}$$

which is very similar to Equation (2.2), with blocks instead of scalars for $l_{11}$ and $a_{11}$. We say that vertices in $S$ have been eliminated. Like before, we see that $B_{22} := A_{22} - A_{21}A_{11}^{-1}A_{12}$ is, in general, denser than $A_{22}$ because of the second term.

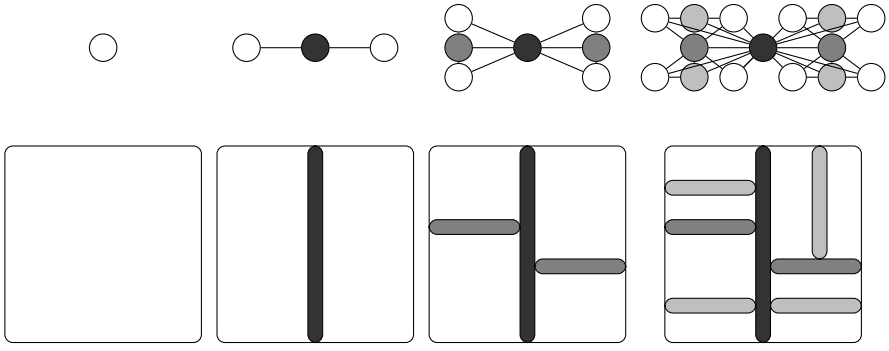
What this indicates is that not all elimination orders are equal. Indeed, one could equivalently factor

$$PAP^\top = LL^\top. \tag{2.3}$$

This merely corresponds to renumbering rows (equations) and columns (unknowns). In (2.3), there are potentially many choices of $P$ leading to different Schur complements, trailing matrices, and eventually $L$'s, all with different sparsity patterns. Those choices are called *orderings*, since they amount to reordering the unknowns, through the permutation $P$.

Finding the $P$ minimizing the amount of fill-in is known to be NP-complete in general [161]. As such, many heuristics exist. One of the most common heuristics, in particular when $A$ comes from the discretization of a PDE, is the *Nested-Dissection* (ND) ordering [71, 109, 74]. ND is a divide-and-conquer algorithm where the graph of $A$ is recursively divided into three sets of dofs: a separator $s$, a left interior $l$, and a right interior $r$. $l$ and $r$ should be picked in such a way that they are disconnected in the graph of $A$. This implies that the matrix can be written in an *arrow-head* fashion

$$A = \begin{bmatrix} A_{ll} & & A_{sl} \\ & A_{rr} & A_{sr} \\ A_{sl} & A_{sr} & A_{ss} \end{bmatrix}$$



(a) $\ell = 3$ (b) $\ell = 2$ (c) $\ell = 1$ (d) $\ell = 0$

Figure 2.1: Classical Nested Dissection ordering for $L = 4$ levels.

We then see that the elimination of $l$ does not produce fill-in on $r$ (and vice-versa). They only produce fill-in on $s$, the separator. This also implies that they can be eliminated in parallel. The algorithm is then recursively applied on $L$ and $R$ until interiors are small enough to be factored directly using a dense direct method. Elimination then follows a bottom-up approach, where interiors are eliminated before their parent separator.

For the remainder of this work, it will be convenient to consider a slightly more restrictive framework in which interiors and separators form a binary tree of $L$ levels. This means that the recursive process described before stops after exactly $L$ steps. Every level $\ell$ will have exactly $2^{L-\ell-1}$ separators (note that some separators may be empty, and they may have different sizes). We then denote the leaf level by $\ell = 0$ and the top-level by $\ell = L - 1$. Figure 2.1 illustrates the separator construction processor for $L = 4$. We see how the graph of $A$ is recursively separated by separators (in black and grey). Figure 2.2 shows the associated binary tree of separators and interiors. Elimination then proceeds level by level from the bottom ($\ell = 0$) to the top ($\ell = L - 1$). At a given level $\ell$, we often refer to the dofs to be eliminated as *interiors*.

We can also write the elimination in matrix form. Define $A^{(0)}$ as the entire matrix and let $A^{(\ell)}$ (for $\ell > 0$) be the Schur complement operator (trailing matrix) obtained by eliminating levels $0, \ldots, \ell - 1$. The matrix obtained after eliminating the $\ell - 1$ level can be

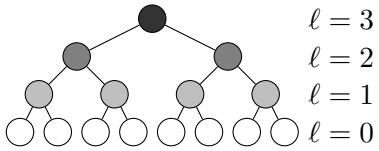



Figure 2.2: The elimination tree associated to the ND ordering on Figure 2.1 for $L = 4$ levels.

written in a block-arrowhead form

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

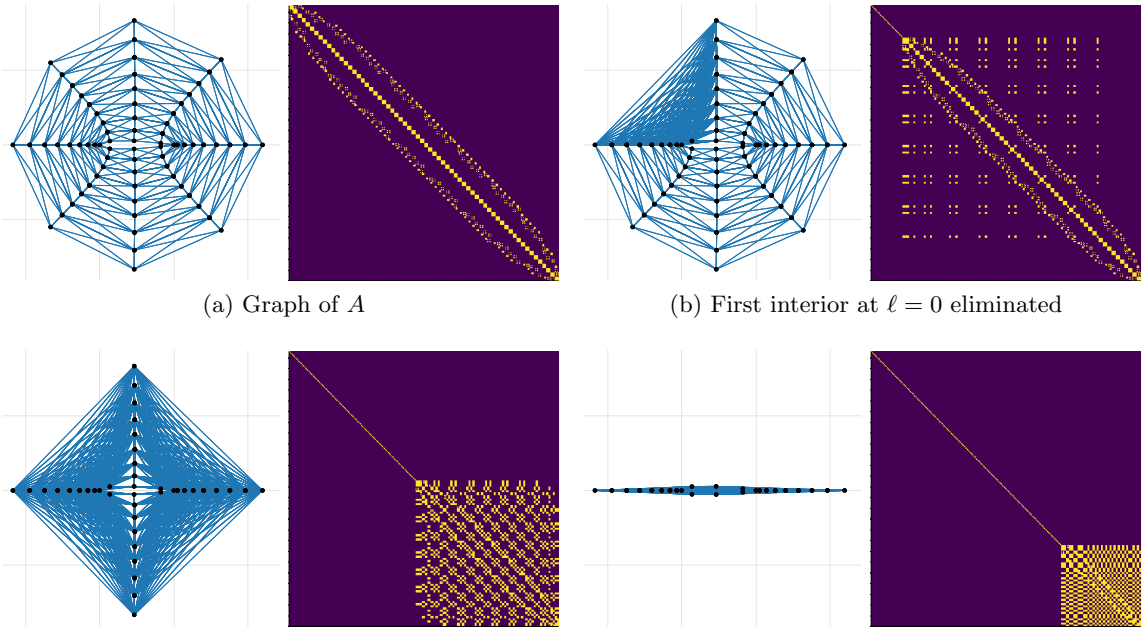
where $m = 2^{L-\ell-1}$ and $q = m + 1$. Here, $A_{qq}^{(\ell)}$ refers to the matrix associated with all separators at levels $\ell + 1, \dots, L - 1$. The $A_{ii}^{(\ell)}$ (for $i \leq m$) are the matrices associated with non-eliminated unknowns within the $i^{th}$ disconnected separators on the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

Figure 2.3 illustrates the elimination of a sparse matrix $A$ using ND ordering with $L = 3$. The matrix graph is shown on the left and the trailing matrix on the right. The trailing matrix is shown in its natural, or topological, ordering. So it is not in arrowhead form. We observe the phenomenon described earlier: when an interior is eliminated, in general, all nodes at its boundaries are connected.

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 entirely 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



(a) Graph of $A$

(b) First interior at $\ell = 0$ eliminated

(c) $\ell = 0$ eliminated

(d) $\ell = 1$ eliminated, with only the top separator left

Figure 2.3: Elimination using ND ordering with $L = 3$ levels

connections), the top separator is a plane (see Figure 2.4) of size $n \times n = n^2 = N^{2/3}$. Hence, its factorization will cost $\mathcal{O}\big(N^{2/3\times 3}\big) = \mathcal{O}\big(N^2\big)$, leading to quadratic or near-quadratic algorithms. While this is only formally valid on regular cubic-shaped graphs, the issue extends beyond those problems[109]: 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 $\mathcal{O}(n) = \mathcal{O}\big(N^{1/3}\big)$ so that its factorization costs $\mathcal{O}(N)$.

### 2.2.2 Sparsified Nested Dissection

We now explain an algorithm used to further decrease the amount of fill-in arising during the factorization. Consider again the Cholesky factorization of $A$ using a ND ordering, with $L = 3$, and assume level $\ell = 0$ has been eliminated.

Consider the example on Figure 2.5. Figure 2.5a shows the original matrix and graph.

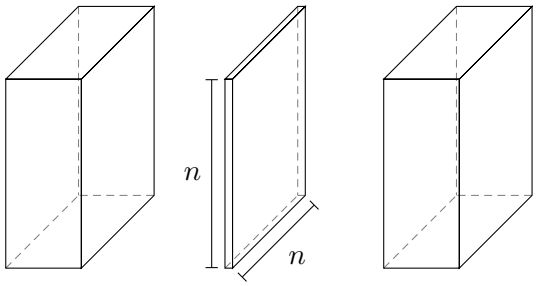



Figure 2.4: Classical ND in $3D$ with $N = n^3$ nodes: the top separator is of size $\mathcal{O}(n^2)$. Once the left and right clusters have been eliminated the top separator becomes completely dense, making its elimination alone cost $\mathcal{O}((n^2)^3) = \mathcal{O}(N^2)$.

We then perform one level of ND elimination and obtain Figure 2.5b. Now consider the dofs depicted in red on Figure 2.5c, and call those $p$. Their neighbors, $n$ are depicted in green. Notice how *most* of the connections from red to green are through fill-in. In particular, for nodes in $p$ at the top, most of those fill-ins are very long-range interactions, meaning their neighbors were originally very distant in the graph of $A$.

Now consider the matrix $A_{pn}$ corresponding to edges from $p$ to $n$. Since $p$ has eight dofs, $A_{pn}$ has eight rows. Also, since $A$ is SPD, $A_{np} = A_{pn}^\top$, so considering $A_{pn}$ is enough in the following analysis. Let $\sigma_i$ be the singular values of $A_{pn}$ $(1 \leq i \leq 8)$. $\sigma_1/\sigma_8$ are depicted on Figure 2.5c. Notice that $\sigma_i$ decays quickly with $i$. *This observation is crucial.* This implies that $A_{pn}$ can be well-approximation by a low-rank matrix. Let

$$A_{pn} = Q_c W_{cn} + Q_f W_{fn} \tag{2.4}$$

where $Q = \begin{bmatrix} Q_c & Q_f \end{bmatrix}$ is orthogonal and $\|W_{fn}\|_2 \leq \varepsilon$ where $\varepsilon$ is small, for instance $10^{-2}\|A_{pn}\|_2$. We call *rank* the number of columns of $Q_c$ (i.e., the $\varepsilon$-rank of $A_{pn}$). Given the fast decay of $\sigma_i$, we can expect this approximation to be very accurate even for a small rank.

Now consider the trailing matrix at that stage, reordered so that $p$ come first, followed by $n$ and where $w$ denotes the remaining dofs. In future sections, $A_{pp}$ will be replaced by the identity. But to keep the discussion short we don't consider this here. The trailing matrix is then

$$\begin{bmatrix} A_{pp} & A_{pn} & \\ A_{np} & A_{nn} & A_{nw} \\ & A_{wn} & A_{ww} \end{bmatrix}.$$

For convenience, let

$$Q^\top A_{pp} Q = \begin{bmatrix} B_{ff} & B_{fc} \\ B_{cf} & B_{cc} \end{bmatrix} = \begin{bmatrix} Q_f^\top A_{pp} Q_f & Q_f^\top A_{pp} Q_c \\ Q_c^\top A_{pp} Q_f & Q_c^\top A_{pp} Q_c \end{bmatrix}.$$

Given (2.4), we have

$$\begin{bmatrix} Q^\top & & \\ & I & \\ & & I \end{bmatrix} \begin{bmatrix} A_{pp} & A_{pn} & \\ A_{np} & A_{nn} & A_{nw} \\ & A_{wn} & A_{ww} \end{bmatrix} \begin{bmatrix} Q & & \\ & I & \\ & & I \end{bmatrix} = \left[\begin{array}{c|c|cc} B_{ff} & B_{fc} & \mathcal{O}(\varepsilon) & \\ \hline B_{cf} & B_{cc} & W_{cn} & \\ \hline \mathcal{O}(\varepsilon) & W_{nc}^\top & A_{nn} & A_{nw} \\ & & A_{wn} & A_{ww} \end{array}\right]$$

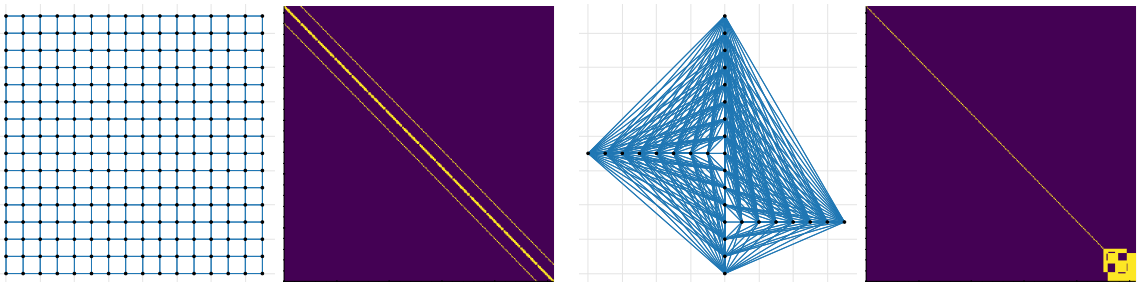
$$\approx \begin{bmatrix} B_{ff} & B_{fc} & & \\ B_{cf} & B_{cc} & W_{cn} & \\ & W_{nc}^\top & A_{nn} & A_{nw} \\ & & A_{wn} & A_{ww} \end{bmatrix}$$

So we see that after transformation $Q$, the dofs $f$ ("fine", following AMG's terminology [143]) are approximately disconnected from $n$ (and $w$, as before). The dofs $c$ ("coarse") are connected to the original neighbors of $p$. In addition, this transformation did not introduce any fill-in on $n$ or $w$. So dofs $f$ can be immediately eliminated, and the trailing matrix becomes

$$\begin{bmatrix} C_{cc} & W_{cn} & \\ W_{nc}^\top & A_{nn} & A_{nw} \\ & A_{wn} & A_{ww} \end{bmatrix}$$

with $C_{cc} = B_{cc} - B_{cf} B_{ff}^{-1} B_{fc}$. This result is shown in Figure 2.5d. We see that $p$ shrunk in size, with only one dof left. Similarly, the trailing matrix lost seven rows and columns, without creating any new fill-in (there is no fill-in between the left and right separators on Figure 2.5d).

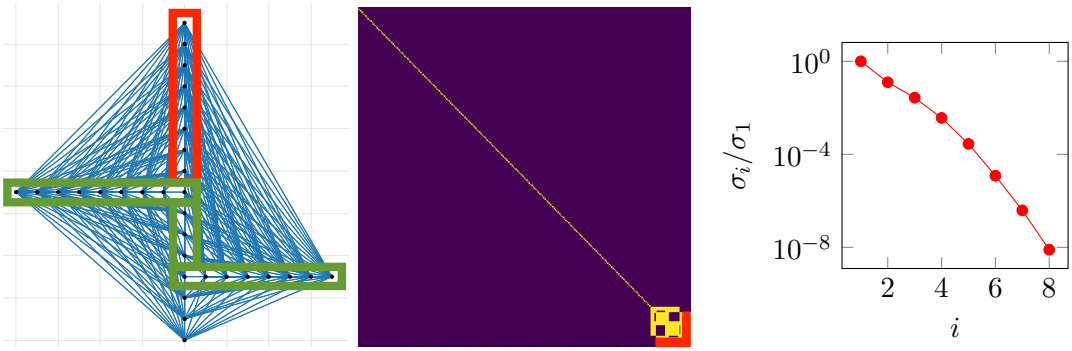
This idea is the key behind the spaND, explained in detail in the next section. This sparsification procedure is applied at each level of the ND elimination, after interior elimination. As hinted on Figure 2.5, spaND requires identifying clusters ($p$, on Figure 2.5) originally connected to their neighbors through as long-range connections as possible. We call those clusters *interfaces*, and the algorithm simply sparsified all remaining interfaces at every level. Algorithm 2.1 presents a high-level version of the algorithm.



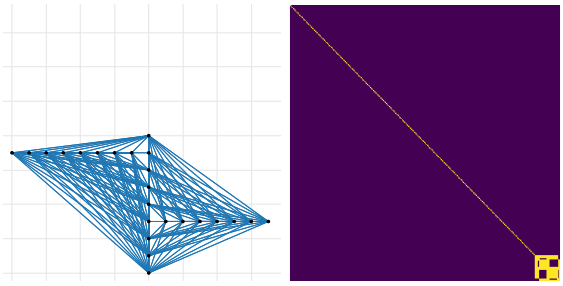
(a) Original matrix and graph



(b) After interiors at $\ell = 0$ are eliminated




(c) Vertices selected for sparsification ($p$, red) and their neighbors ($n$, green)



(d) After sparsification. $p$ decreased in size while creating no new fill-ins between its neighbors $n$.

Figure 2.5: Low-rank structure arising from elimination using ND ordering

**Algorithm 2.1** High-level description of the spaND algorithm

**Require:** Sparse matrix $A$, Maximum level $L$
  Compute a ND ordering for $A$, infer interiors, separators and interfaces (see Section 2.2.3)
  **for all** $\ell = 0, \ldots, L-1$ **do**
    **for all** $\mathcal{I}$ interior **do**
      Eliminate $\mathcal{I}$ (see Section 2.2.4)
    **end for**
    **for all** $\mathcal{B}$ interface between interiors **do**
      Sparsify $\mathcal{B}$ (see Section 2.2.5 and Section 2.2.6)
    **end for**
  **end for**

The result is that separators sizes are greatly reduced, at the cost of the controllable approximation error directly related to $\varepsilon$. As $\varepsilon \to 0$, the algorithm simply reverts to a classical ND elimination. Typically, in 2D, separator sizes decay so that the algorithm complexity becomes $\mathcal{O}(N)$ and, in 3D, $\mathcal{O}(N \log N)$.

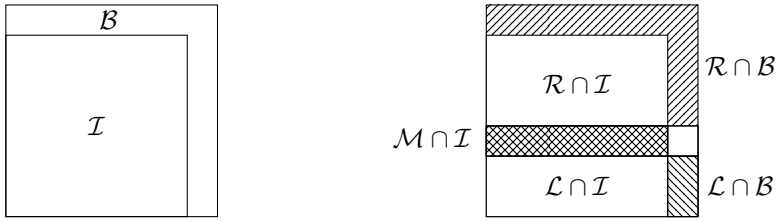
The subsequent sections explain in detail the ordering & clustering (i.e., how we define the "interfaces"), the elimination and sparsification.

### 2.2.3 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 Figure 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 (like the red box on Figure 2.5c for instance).

In principle, one could cluster separators individually, for instance using K-way or any other clustering algorithm. However, in this work, we seek a clustering algorithm that clusters separators in such a way that every cluster is adjacent to a given pair of "left" and "right" interiors. This will let us extract low-rank structure in the matrix, using the fact that most of the nodes in every cluster will be connected to outside nodes mostly through fill-ins.

To do so, 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




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

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

Figure 2.6: 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.

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 let us keep track of the interfaces. Figure 2.6 shows the high-level idea. For every interior $\mathcal{I}$ we keep track of its boundary $\mathcal{B}$ and we then separate their union $\mathcal{I} \cup \mathcal{B}$.

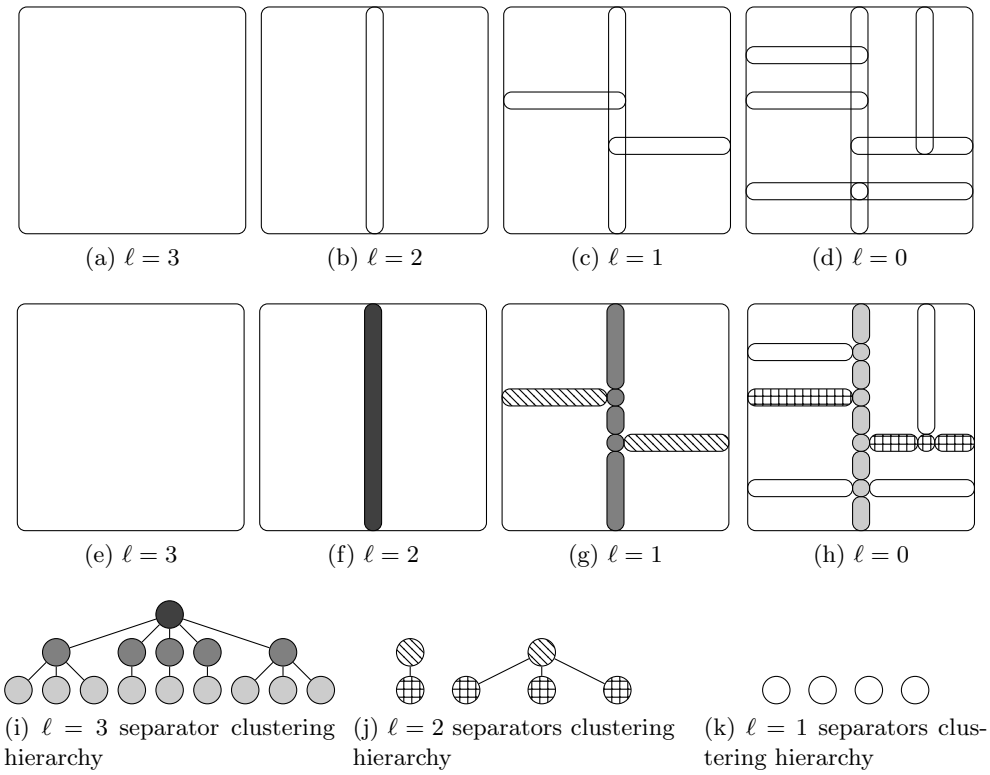
In practice, each node in the graph keeps track of its "left" and "right" neighboring separators, in addition to keeping track of the separator it belongs to. We encode this by associating to each vertex $v$ a 3-tuple $(S, L, R)$. $S$ is the usual ND separator $(\ell, k)$ where $\ell$ is its level ($0 \leq \ell < L$, with $L-1$ the root and 0 the leaves) and $k$ its separator ($0 \leq k < 2^{L-\ell-1}$). $L$ and $R$ are the ND separators of $v$'s left and right neighbors, respectively. Algorithm 2.2 formalizes this idea. We call this algorithm Modified Nested Dissection (MND). Notice how the only building block is a vertex-separator routine, as available in Metis [100].

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.

Figure 2.7 illustrates the effect of Algorithm 2.2 On the top row, we illustrate the separators at every step ($\ell$) of the algorithm. The important distinction with Figure 2.1 is that the computed vertex-separators overlap with the boundaries to keep track of interfaces and that each separator is further divided into clusters. In the middle row, we illustrate the actual clusters at each level and how the ND separators are broken into pieces. Separators

**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 $(\ell, k)$ where $\ell$ 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)*
  $C[v] = (S : (L-1, 0), L : \texttt{none}, R : \texttt{none})$ for all $v \in V$
  **for all** $\ell = L-1, \ldots, 1$ **do**
    **for all** $k = 1, \ldots, 2^{L-\ell-1}$ **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 $\mathcal{R}$*
      $(\mathcal{L}, \mathcal{M}, \mathcal{R}) = \texttt{vertex-separator}(\mathcal{I} \cup \mathcal{B})$
      *% Update separator, left and right interiors*
      $C[v]_S = (\ell, k)$ for all $v \in \mathcal{M} \backslash \mathcal{B}$
      $C[v]_S = (\ell-1, 2k-1)$ for all $v \in \mathcal{L} \backslash \mathcal{B}$
      $C[v]_S = (\ell-1, 2k)$ for all $v \in \mathcal{R} \backslash \mathcal{B}$
      *% 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** $C$



(a) $\ell = 3$ (b) $\ell = 2$ (c) $\ell = 1$ (d) $\ell = 0$

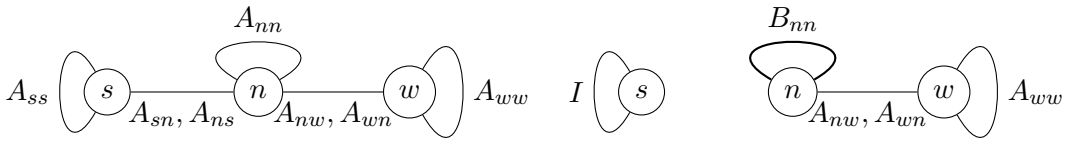
(e) $\ell = 3$ (f) $\ell = 2$ (g) $\ell = 1$ (h) $\ell = 0$

(i) $\ell = 3$ separator clustering hierarchy (j) $\ell = 2$ separators clustering hierarchy (k) $\ell = 1$ separators clustering hierarchy

Figure 2.7: A modified ND ordering & clustering (Algorithm 2.2) for $L = 4$. 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.

at each level are depicted in a different color. Each separator is associated with 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 a higher level.

In practice (see Section 2.4), 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 [100].




(a) Before eliminating $s$ (b) After eliminating $s$

Figure 2.8: Before and after the elimination of the separator $s$ using block Cholesky. Eliminating $s$ disconnects it from the rest, but requires updating the $A_{nn}$ edges, connecting all neighbors $n$ of $s$

### 2.2.4 Elimination of separators

Now that the matrix has been ordered and that dofs have been grouped into clusters defining various interfaces, the next step is to eliminate the interiors or separators at a given level $\ell$ of the ND tree, as in a usual direct solver (see Algorithm 2.1). This section describes this elimination step.

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 $w$.

Assume $A_{ss}$ is invertible, and let $L_s U_s^\top = A_{ss}$ be *some* factorization of $A_{ss}$ where $L_s$ and $U_s$ are easy to invert (like triangular, permutation, orthogonal, or any product of those). When $A$ is SPD, we use Cholesky, i.e., $A_{ss} = L'_s L_s'^\top$, so $L_s = L'_s$ and $U_s = L_s'^\top$. In general, one can use row pivoted LU or full pivoted LU, i.e., $A_{ss} = P'_s L'_s U'_s Q'_s$ and $L_s = P'_s L'_s$, $U_s = U'_s Q'_s$.

Then, define

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

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

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

Notice that $E_s A F_s$ does not depend on the choice of $L_s$ and $U_s$, only on $A_{ss}^{-1}$. This may introduce (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.

Figure 2.8 shows the elimination process from the matrix' graph perspective. We see that $s$ is now isolated from the rest of the graph. We say that $s$ has been *eliminated*. Furthermore, the effect on the *rest* of the graph was to update the self-edge $n$–$n$, $A_{nn}$. Separated nodes ($w$) remain untouched.

### 2.2.5 Block scaling of the interfaces

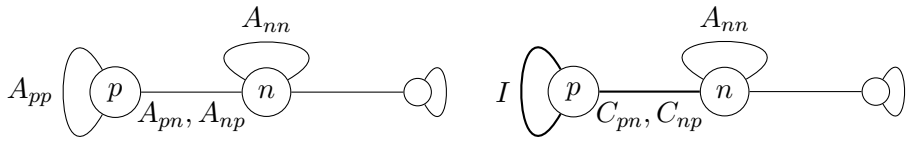
Once that the separators or interiors at a given level have been eliminated, the algorithm goes through each interface and sparsifies it. However, a critical step before this is the proper scaling of each of those clusters. The goal is to scale (what is left of) $A$ such that each diagonal block corresponding to a given interface is the identity. This provides theoretical guarantees in the SPD case (Section 2.3) and significantly improves the accuracy of the preconditioner (Section 2.4).

Consider the matrix

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

Like in Section 2.2.4, let $A_{pp} = L_p U_p^\top$. Notice in this case that it may be desirable to *balance* $L_p$ and $U_p$, so that $\|L_p^{-1}\| \approx \|U_p^{-1}\|$. This is done to keep $A_{np}$ and $A_{pn}$ of similar magnitudes. When using Cholesky, this is trivially the case since $U_p = L_p$. We define the block-scaling operation over $p$ as

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




(a) Before scaling $p$ (b) After scaling $p$

Figure 2.9: Scaling of an interface $p$

The result is

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

also depicted in Figure 2.9.

We see that the scaling of $p$ requires modifying the edges involving $p$ itself, but it has no other effect than that.

### 2.2.6 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

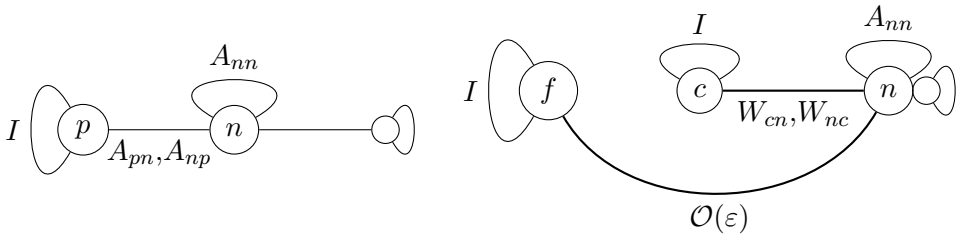
$$A = \begin{bmatrix} A_{pp} & A_{pn} \\ A_{np} & A_{nn} \end{bmatrix}$$

#### 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 $\begin{bmatrix} A_{pn} & A_{np}^\top \end{bmatrix}$ (if $A = A^\top$, the second term is not needed) can be well approximated by a low-rank matrix, i.e.,

$$\begin{bmatrix} A_{pn} & A_{np}^\top \end{bmatrix} = Q_{pc} \begin{bmatrix} W_{cn} & W_{nc}^\top \end{bmatrix} + Q_{pf} \begin{bmatrix} W_{fn} & W_{nf}^\top \end{bmatrix}, \quad \left\| \begin{bmatrix} W_{fn} & W_{nf}^\top \end{bmatrix} \right\|_2 \leq \varepsilon$$

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) [75, 36, 80].




(a) Before sparsification of $p$ (b) After sparsification of $p$

Figure 2.10: Sparsification of $p$ using orthogonal transformation.

We use the letters $c$ to denote the "coarse" (also known as "skeleton" or "relevant", [95]) dofs, and $f$ the "fine" ("redundant" or "irrelevant") dofs. Let $Q_{pp}$ be a square orthogonal matrix built as $Q_{pp} = \begin{bmatrix} Q_{pf} & Q_{pc} \end{bmatrix}$. This implies

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

Then, define

$$Q_p = \begin{bmatrix} Q_{pp} & \\ & I \end{bmatrix} \tag{2.5}$$

We see that

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

Figure 2.10 shows the effect of the sparsification on the matrix graph. This is the key picture. 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 $n$–$n$.

**Why not having different $Q_{pp}$ for the left and the right?** In the previous paragraph, $Q_{pp}$ was computed as a basis for $\begin{bmatrix} A_{pn} & A_{np}^\top \end{bmatrix}$. One can naturally ask why we did not compute a basis $Q_{pp}^l$ for $A_{pn}$ and a basis $Q_{pp}^r$ for $A_{np}^\top$, when $A$ is general. But notice that in this case

$$Q_p^{l\top} A Q_p^r = \begin{bmatrix} Q_{pf}^{l\top} Q_{pf}^r & Q_{pf}^{l\top} Q_{pc}^r & \mathcal{O}(\varepsilon) \\ Q_{pc}^{l\top} Q_{pf}^r & Q_{pc}^{l\top} Q_{pc}^r & W_{cn} \\ \mathcal{O}(\varepsilon) & W_{nc} & A_{nn} \end{bmatrix}$$

where there is no guarantee that $Q_{pf}^{l\top} Q_{pf}^r$ is invertible. In particular, if $Q_{pf}^l \perp Q_{pf}^r$, then $Q_{pf}^{l\top} Q_{pf}^r = 0$, and the algorithm would break down, as the fine dofs cannot be eliminated.

**Which interfaces should be sparsified?** After elimination, the remaining dofs are all considered interfaces. However, in general, not all interfaces should be sparsified. Only interfaces with (relatively to their size) few original connections (i.e. connection existing in $A$) should be sparsified. Using the framework of Algorithm 2.2, this means only interfaces for which their left and right separators are eliminated should be sparsified.

### 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 [95].

Assume we can *partition* $p = c \cup f$ (so, in this case, $c$ and $f$ are subsets of $p$) such that (if $A = A^\top$, the bottom row is ignored)

$$\begin{bmatrix} A_{nf} \\ A_{fn}^\top \end{bmatrix} = \begin{bmatrix} A_{nc} \\ A_{cn}^\top \end{bmatrix} T_{cf} + \mathcal{O}(\varepsilon)\,.$$

This is often called "interpolative decomposition". It can be computed for instance using a rank-revealing QR (RRQR) factorization [45] (note that the RRQR is computed over $A_{np}$ instead of $A_{pn}$ in Section 2.2.6): computing a RRQR over $\begin{bmatrix} A_{np} \\ A_{pn}^\top \end{bmatrix}$ leads to (with $P$ the permutation, and $R_{22} = \mathcal{O}(\varepsilon)$)

$$\begin{bmatrix} A_{nc} & A_{nf} \\ A_{cn}^\top & A_{fn}^\top \end{bmatrix} = \begin{bmatrix} A_{np} \\ A_{pn}^\top \end{bmatrix} P = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} A_{nf} \\ A_{fn}^\top \end{bmatrix} = \underbrace{Q_1 R_{11}}_{\begin{bmatrix} A_{nc} \\ A_{cn}^\top \end{bmatrix}} \underbrace{R_{11}^{-1} R_{12}}_{T_{cf}} + \underbrace{Q_2 R_{22}}_{=\mathcal{O}(\varepsilon)}$$

Note that this factorization can also be computed using randomized methods [107]. This technique is referred to as "interpolative" because it is exact on $A_{nc}$ and $A_{cn}$: only $A_{nf}$ and $A_{fn}$ are 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 Equation (2.5) was orthogonal. Both can be efficiently inverted; however, working with orthogonal matrices brings stability guarantees (see Section 2.3). Then,

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

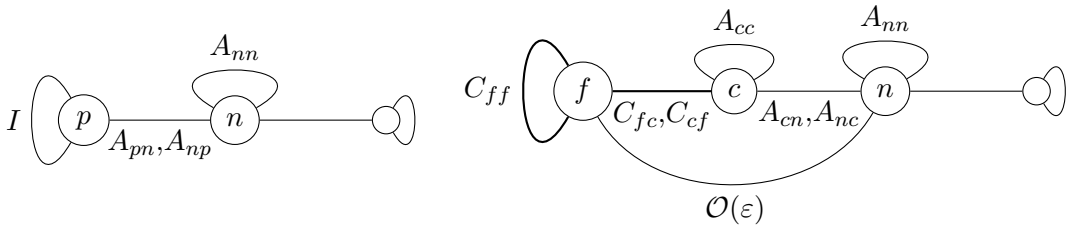
with

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

Figure 2.11b shows the matrix' graph after the sparsification of $p$ using interpolative factorization (and without prior scaling). Notice how $f$ and $c$ are still connected; however, $f$ is (almost) disconnected from the rest of the matrix. We see that by ignoring the $\mathcal{O}(\varepsilon)$ term, we can eliminate $f$ immediately (see Section 2.2.4). The result is (note the updated $c$–$c$ edge) shown in Figure 2.11c.

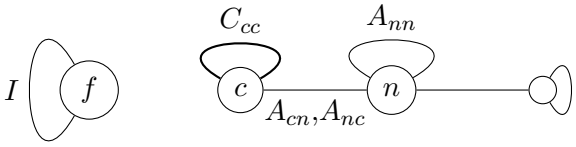
The final result is the same as using orthogonal transformation. The differences are that

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




(a) Before sparsification of $p$

(b) After sparsification of $p$




(c) After elimination of $f$

Figure 2.11: Sparsification of $p$ using interpolative factorization

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.

### 2.2.7 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 Figure 2.7. After having eliminated the leaf (level $\ell = 0$) and the level $\ell = 1$ separators, we need to merge the clusters in each separator. This is done following the cluster trees. Merging children clusters $p_1, \ldots, p_k$ into a parent cluster $p$ simply means concatenating their dofs:

$$p = \begin{bmatrix} p_1 & p_2 & \ldots & p_k \end{bmatrix}.$$

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

### 2.2.8 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 $E_s$ and $F_s$ (Section 2.2.4), scalings $S_p$ and $R_p$ (Section 2.2.5) and sparsification $Q_p$ (Section 2.2.6) (plus potentially some re-orderings and permutations to take care of the fine nodes $f$ and the merge), at each level $\ell$, effectively reducing $A$ to (approximately) $I$:

$$M_l A M_r \approx I \text{ with } M_l = \prod_{\ell=0}^{L-1} \left( \prod_{p \in C_\ell} Q_p^\top \prod_{p \in C_\ell} S_p \prod_{s \in S_\ell} E_s \right), \; M_r = \prod_{\ell=L-1}^{0} \left( \prod_{s \in S_\ell} F_s \prod_{p \in C_\ell} T_p \prod_{p \in C_\ell} Q_p \right)$$

In this expression, $S_\ell$ is a set containing all the ND separators at level $\ell$ and $C_\ell$ contains all the clusters (interfaces) in the graph right after level $\ell$ elimination. Since $M_l$ and $M_r$ are given as a product of elementary transformations, they can easily be inverted. Algorithm 2.3 presents the algorithm.

We illustrate the effect of all the $E_s$, $F_s$, $S_p$, $T_p$, and $Q_p^\top$ in $A$ (i.e., the trailing matrix) in Figure 2.12. 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.

## 2.3 Theoretical results

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

### 2.3.1 Sparsification and error in the Schur complement

Consider a framework where

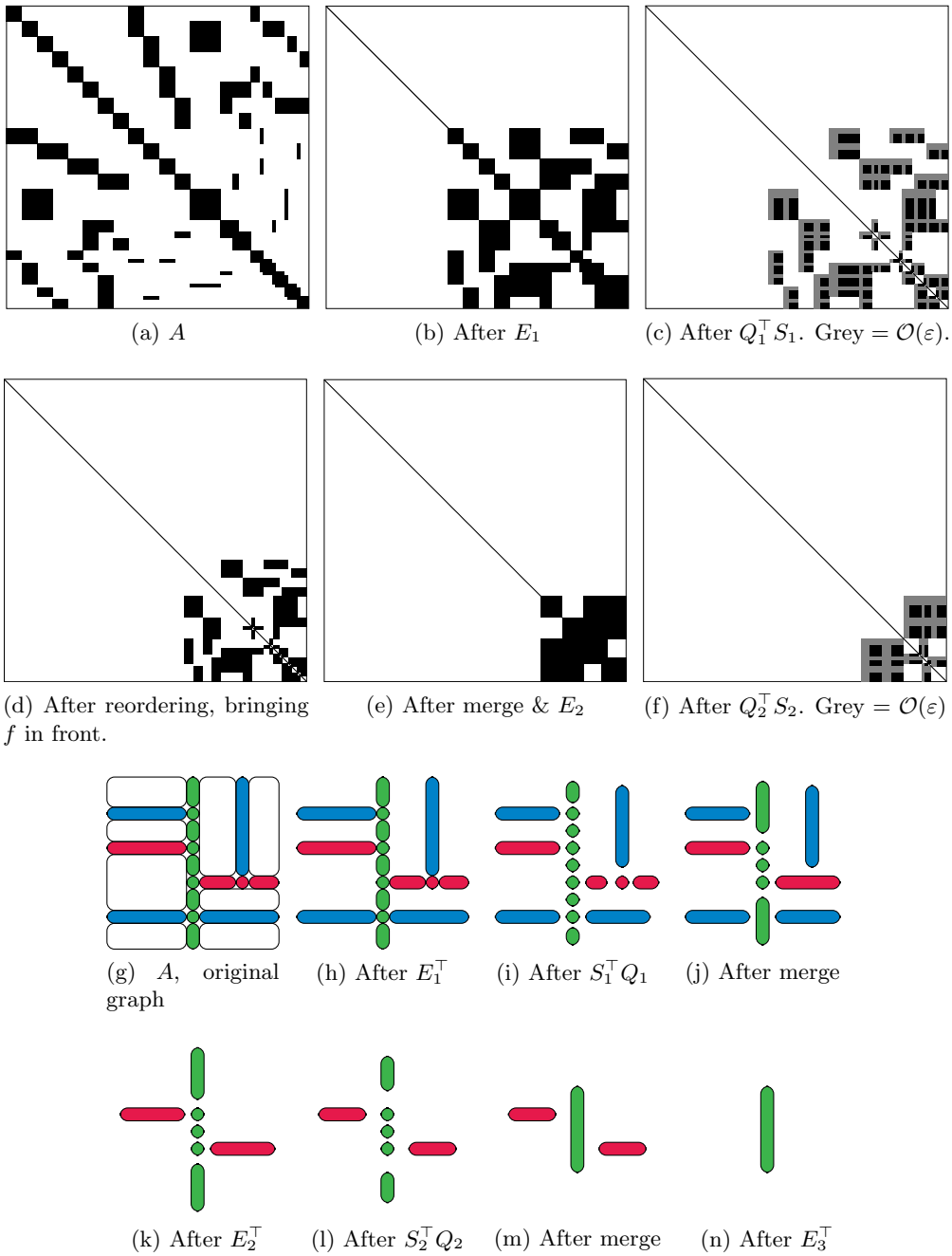
$$A = \begin{bmatrix} A_{pp} & A_{pn} \\ A_{np} & A_{nn} \end{bmatrix}.$$

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

$$\begin{bmatrix} A_{pn} & A_{np}^\top \end{bmatrix} = X_1 \begin{bmatrix} Y_{1pn} & Y_{1np}^\top \end{bmatrix} + X_2 \begin{bmatrix} Y_{2pn} & Y_{2np}^\top \end{bmatrix}$$

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

**Require:** $A$ square; $L > 0$; $\varepsilon$
  $M_l = [], M_r = []$ (empty list)
  Compute a $L$-levels modified ND ordering of $|A| + |A|^\top$ using Algorithm 2.2.
  Infer clusters hierarchy in each ND separator.
  **for all** $\ell = 0, \ldots, L-1$ **do**
    **for all** $s$ separator at level $\ell$ **do** ▷ Eliminate separators at level $\ell$
      Eliminate $s$, get $E_s$ and $F_s$ (Section 2.2.4)
      Append $E_s$ to $M_l$ and $F_s$ to $M_r$
    **end for**
    **for all** $p$ interfaces at level $\ell$ **do** ▷ Scale interface
      Scale $p$, get $S_p$ and $T_s$ (Section 2.2.5)
      Append $S_p$ to $M_l$ and $T_p$ to $M_r$
    **end for**
    **for all** $p$ interface at level $\ell$ between eliminated interiors **do** ▷ Sparsify interfaces
      Sparsify $p$ with accuracy $\varepsilon$, get $Q_p$ (Section 2.2.6)
      Append $Q_p$ to $M_l$ and $M_r$
    **end for**
    **for all** $s$ separator at level $\ell$ **do** ▷ Merge clusters
      Merge interfaces of $s$ one level following clusters hierarchy (Section 2.2.7)
    **end for**
  **end for**
  **return** $M_l$, $M_r$ (such that $M_l A M_r \approx I$)



(a) $A$ (b) After $E_1$ (c) After $Q_1^\top S_1$. Grey $= \mathcal{O}(\varepsilon)$.

(d) After reordering, bringing $f$ in front. (e) After merge & $E_2$ (f) After $Q_2^\top S_2$. Grey $= \mathcal{O}(\varepsilon)$

(g) $A$, original graph (h) After $E_1^\top$ (i) After $S_1^\top Q_1$ (j) After merge

(k) After $E_2^\top$ (l) After $S_2^\top Q_2$ (m) After merge (n) After $E_3^\top$

Figure 2.12: Illustration of the spaND algorithm, where for simplicity *all* interfaces are sparsified at each level. Given $A$, create a ND tree of depth 4 and cluster $A$ accordingly, as shown in Figure 2.12g. This cartoon shows clusters of vertices of $A$, where the edges (not shown) should be thought of as connecting close neighbors (like on a regular 2D grid). Denote by $E_\ell$ and $F_\ell$, $S_\ell$ and $T_\ell$ and $Q_\ell$ all eliminations, scalings and sparsifications at level $\ell$. Then, we have $E_4E_3(Q_2^\top S_2E_2)(Q_1^\top S_1E_1)A(F_1T_1Q_1)(F_2T_2Q_2)F_3F_4 \approx I$. The top rows show the evolution of the trailing matrix; the 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.

where $\left\| \begin{bmatrix} Y_{2pn} & Y_{2np}^\top \end{bmatrix} \right\| \leq \varepsilon$. Using $X = \begin{bmatrix} X_1 & X_2 \end{bmatrix}$ as a change of variable, $A$ becomes

$$\begin{bmatrix} X^{-1} & \\ & I \end{bmatrix} \begin{bmatrix} A_{pp} & A_{pn} \\ A_{np} & A_{nn} \end{bmatrix} \begin{bmatrix} X^{-\top} & \\ & I \end{bmatrix} = \begin{bmatrix} B_{11} & B_{12} & Y_{1pn} \\ B_{21} & B_{22} & Y_{2pn} \\ Y_{1np} & Y_{2np} & A_{nn} \end{bmatrix}$$

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

$$S = A_{nn} - Y_{2np} B_{22}^{-1} Y_{2pn}$$

while the approximate one, ignoring $Y_{2pn}$ and $Y_{2np}$, is simply $A_{nn}$. The error is then

$$E_{nn} = Y_{2np} B_{22}^{-1} Y_{2pn}.$$

We can now consider the different variants proposed in Section 2.2.6:

- (`In`) spaND using interpolative factorization and no diagonal block scaling. This gives $B_{22} = C_{ff}$, so that

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

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

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

  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 [112]), $\|C_{ff}^{-1}\|$ 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 so,

$$\|E_{nn}\|_2 \leq \|Y_{2np}\|_2 \|Y_{2pn}\|_2 = \mathcal{O}\left(\varepsilon^2\right).$$

Table 2.1 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 [75, Algorithm 5.4.1]

| 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 2.1: Error for various approximations. The left column indicates the sparsification variant:`In` means interpolative and no scaling; `InS` means interpolative and scaling; `OrthS` means orthogonal and scaling.

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 2.4.

### 2.3.2 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 when $A$ is SPD.

**Theorem 2.1.** *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* $Q_p = \begin{bmatrix} Q_{pf} & Q_{pc} \end{bmatrix}$ *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 $W_{fc}^\top W_{fc} \succeq 0$, we find that $S_B$ is SPD. Since the $c - c$ block of $B_p$ is identity, we conclude that $B_p$ is SPD. □

**Corollary 2.1.** *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 $\epsilon = 0$. 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

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

This is a classical result in the case of low-rank approximation using weak admissibility (see [158, 159] for instance).

### 2.3.3 Complexity analysis

We discuss the complexity of spaND and contrast it with the usual ND algorithm. In the following, let $\delta = L - \ell - 1$, $0 \leq \delta < L$ be the *depth* of the separator in the ND tree, so the top separator has $\delta = 0$ while the leaves have $\delta = L - 1$.

**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 $\beta$-balanced trees in which all children subgraphs have size $\beta n_i$, 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 $O(n_i^{2/3})$ 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 $2^{-\delta}N \leq n_i < 2^{-\delta+1}N$ (see Figure 2.4). The associated separator has size at most

$$c_\delta \in \mathcal{O}\Big(2^{-2\delta/3}N^{2/3}\Big)$$

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

$$e_\delta \in \mathcal{O}\bigg(\Big(2^{-2\delta/3}N^{2/3}\Big)^3\bigg) = \mathcal{O}\Big(2^{-2\delta}N^2\Big)$$

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

$$t_{\mathrm{ND,fact}} \in \mathcal{O}\left(\sum_{\delta=0}^{L-1} 2^\delta e_\delta\right) = \mathcal{O}\left(\sum_{\delta=0}^{L-1} 2^{-\delta}N^2\right) = \mathcal{O}\big(N^2\big)\,, \quad L \in \Theta(\log(N/N_0))$$

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 $\mathcal{O}\big(2^{-2\delta/3}N^{2/3}\big)$ fill-in entries in its rows, the related solve

cost is $\mathcal{O}\big(2^{-4\delta/3}N^{4/3}\big)$ and the cost of one solve is

$$t_{\text{ND,apply}} \in \mathcal{O}\left(\sum_{\delta=0}^{L-1} 2^{-\delta/3}N^{4/3}\right) = \mathcal{O}\Big(N^{4/3}\Big)$$

**spaND** On the other hand, assume that the sparsification can decrease each separator size before elimination from $c_\delta$ to

$$s_\delta \in \mathcal{O}\Big(2^{-\delta/3}N^{1/3}\Big)$$

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 $O(1)$. 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 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 $2^{-\delta}N \leq n_i < 2^{-\delta+1}N$, 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

$$\mathcal{O}\Big((2^{-\delta/3}N^{1/3})^3\Big) = \mathcal{O}\Big(2^{-\delta}N\Big)$$

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

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

$$t_{\text{spaND,fact}} \in \mathcal{O}\left(\sum_{\delta=0}^{L-1} 2^{\delta}\ 2^{-\delta} N\right) = \mathcal{O}(N \log N)$$

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

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

Finally, notice that in both cases the memory complexity scales like the factorization application. Section 2.4.2 presents some experimental results regarding separator sizes as a function of $N$.

### 2.3.4 SPD, symmetric and unsymmetric cases

We finish with a discussion regarding the SPD case, the symmetric case (i.e., symmetric but not SPD), and the general case (neither SPD nor symmetric). We only consider spaND using orthogonal transformations.

As indicated in the previous section, if $A$ is SPD, the trailing matrix provably remains SPD, and the algorithm never breaks down. In the general case, the trailing matrix is unsymmetric from the start. In addition, the block elimination and block scaling may both break down, since there is no guarantee that the pivot is invertible. Pivoting techniques could alleviate this issue but are outside the scope of this work.

Symmetric matrices pose a specific issue. Consider $A$, $A = A^\top$, and assume we are about to sparsify an interface $p$. Since the original matrix is symmetric, we wish for the trailing matrix to remain symmetric during the algorithm (if not, then we can simply consider the general algorithm). Prior to scaling, the trailing matrix can be written as (with $A_{np}^\top = A_{pn}$)

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

Since $A$ is not SPD, there are no guarantees that $A_{pp}$ is invertible, but let us assume it is. In general, $A_{pp}$ can be factored as $A_{pp} = L_p S_p L_p^\top$ where $S_p$ is a diagonal sign matrix (with $+1$ or $-1$ entries). This can be achieved using the eigenvalue decomposition or an LDLT

factorization for instance. Given this, we can then scale $A_{pp}$ so that

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

Notice that it is *not* possible to scale $A$ so that both the following are true:

- $A_{pp}$ becomes the identity
- The trailing matrix remains symmetric.

Hence, $A_{pp}$ has to become $S_p$. We then sparsify $L_p^{-1}A_{pn}$, obtaining the change of basis $Q_{pp}$ so that

$$\begin{bmatrix} Q_{pp}^\top & \\ & I \end{bmatrix} \begin{bmatrix} S_p & B_{pn} \\ B_{np} & A_{nn} \end{bmatrix} \begin{bmatrix} Q_{pp} & \\ & I \end{bmatrix} = \begin{bmatrix} Q_{pp}^\top S_p Q_{pp} & W_{pn} \\ W_{np} & A_{nn} \end{bmatrix}$$

However, notice how $Q_{pp}^\top S_p Q_{pp} \neq I$. In particular, there are *no guarantees* that the $f$–$f$ block of $Q_{pp}^\top S_p Q_{pp}$ is invertible, unlike in the SPD or general case (this is a similar issue as in Section 2.2.6). We see how this adds another challenge, on top of the already lack of guarantees that the pivots are even invertible in the first place. As such, symmetric matrices are usually treated as unsymmetric ones.

## 2.4 Numerical Experiments (SPD)

This section presents applications of the algorithm on various problems. We begin by considering only SPD cases.

We use the following notation throughout this section:

- $\mathrm{t_{fact}}$ is the factorization time (in seconds), not including partitioning;
- $\mathrm{t_{part}}$ is the partitioning time (in seconds);
- $\mathrm{t_{solve}}$ 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);
- $\mathrm{n_{CG}}$ is the associated number of CG steps;

- $\text{size}_{\text{top}}$ is the size of the top separator right before elimination;
- $\text{mem}_{\text{fact}}$ 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) [134] ("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 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 [100] for the vertex-separator routine. We use Ifpack2 [126] for ILU(0). Low-rank approximations are performed using LAPACK's geqp3 [9]. 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 $1 \leq i \leq r$.

### 2.4.1 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 [95] where the algorithm was using the interpolative only variant (with no scaling).

#### High contrast $2D$ Laplacians

We first consider $2D$ elliptic equations

$$\nabla(a(x) \cdot \nabla u(x)) = f \quad \forall x \in \Omega = [0,1]^2,\ u|_{\partial\Omega} = 0 \tag{2.6}$$

where $a(x)$ is a quantized high contrast field with highs of $\rho$ and lows of $\rho^{-1}$ and where 2.6 is discretized with a 5-points stencil. This leads to the following discretization

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

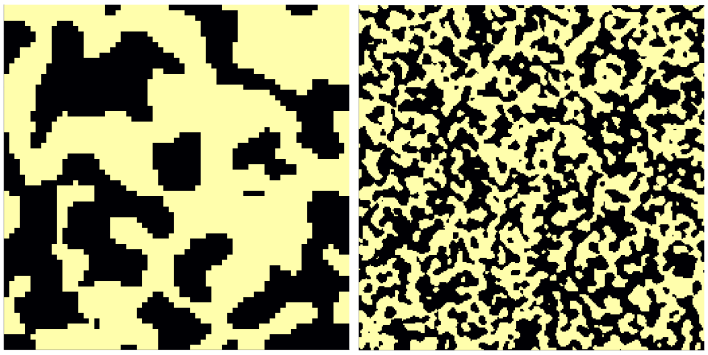


Figure 2.13: A quantized high contrast field with lows of $\rho^{-1}$ and highs of $\rho$ for $n = 32$ (left) and $n = 128$ (right). The features sizes are roughly constant as we increase the mesh size $n$.

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 $\hat{a}$

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

Figure 2.13 gives an example of a high contrast field for $n = 32$ and $n = 128$.

We compare the number of iterations CG [91] 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 (Section 2.2.4) failed. Given that the problem is defined 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).

Figure 2.14 gives results for $\rho = 1$ to $\rho = 1000$. We compare the three variants for various $\varepsilon$ 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

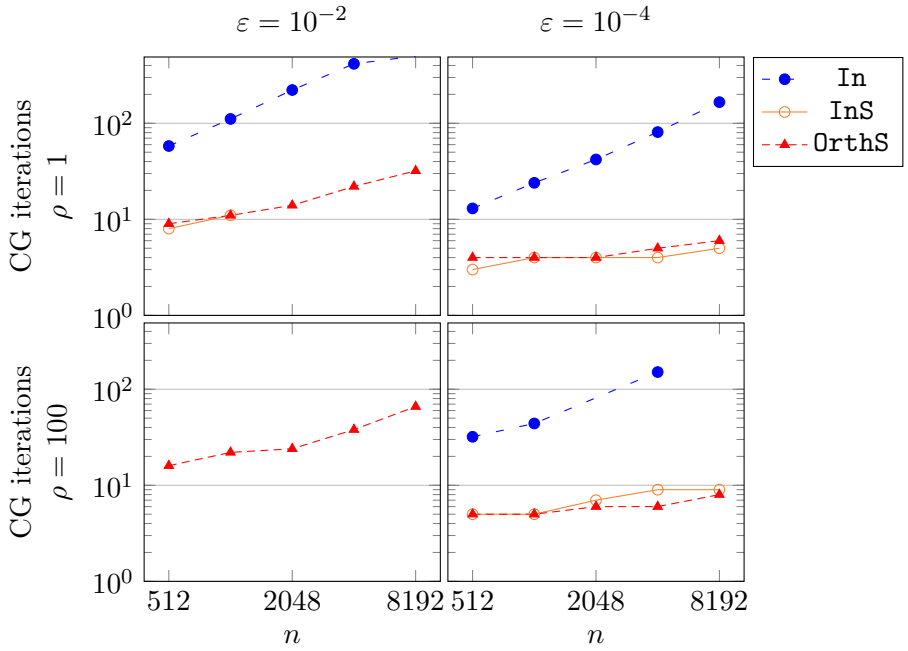



Figure 2.14: 2D $n \times n$ Laplacians: each line represent a given variant: `In` (interpolative and no scaling), `InS` (interpolative and scaling) and `OrthS` (orthogonal and scaling), at a given accuracy $\varepsilon$. Each dot gives the CG iteration count, and we run the experiments on various problems of size $N = n^2$, for various $\rho$. The conditioning is roughly proportional to $\rho$. 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 $\varepsilon$ and `OrthS` never breaks down. In addition, for small enough $\varepsilon$, the accuracy is roughly independent of the problem size $N$. Finally, when $\rho$ is not too extreme, there is little dependency on the condition number.

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 [95] 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.

### Non-Regular Problems

Figure 2.15 gives results for three variants on many[1] of the SPD (real & square) problems

[1]We 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 fewer than 500

from the SuiteSparse matrix collection [53] 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 $\varepsilon = 10^{-6}$.

Figure 2.16 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

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

Each value $T_v(t)$ represents 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 couple of iterations, so the three variants have similar performances. The plots clearly show 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 $\varepsilon$'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 makes them quite attractive in practice for SPD problems.

We also point out that previous work [95] 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.

---

iterations).

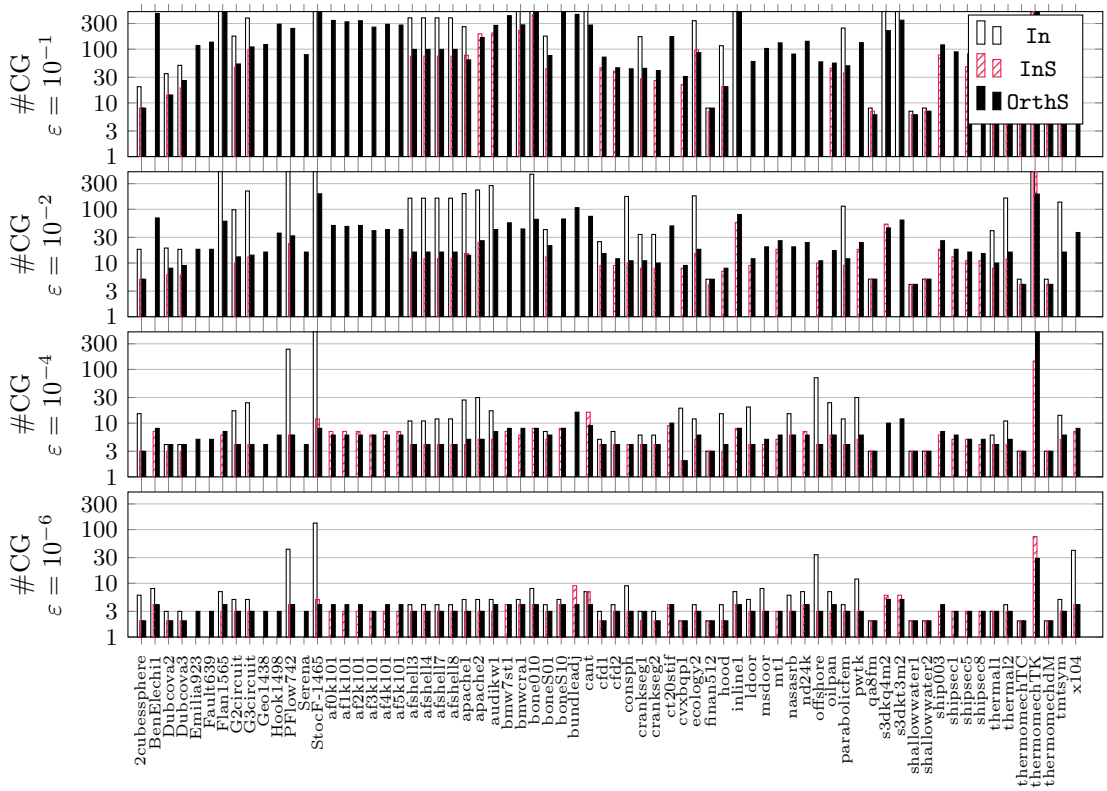



Figure 2.15: SuiteSparse matrix collection: results on many SPD problems of the SuiteSparse matrix collections with $N \geq 50\,000$. The partitioning is only graph-based. Each bar represents the number of CG iterations for a given problem with a given variant of the algorithm: `In` (interpolative and no scaling), `InS` (interpolative and scaling) and `OrthS` (orthogonal and scaling) at a given accuracy $\varepsilon$. The absence of a bar means the algorithm broke down in the face of a non-SPD pivot. This shows that, as $\varepsilon \to 0$, 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.

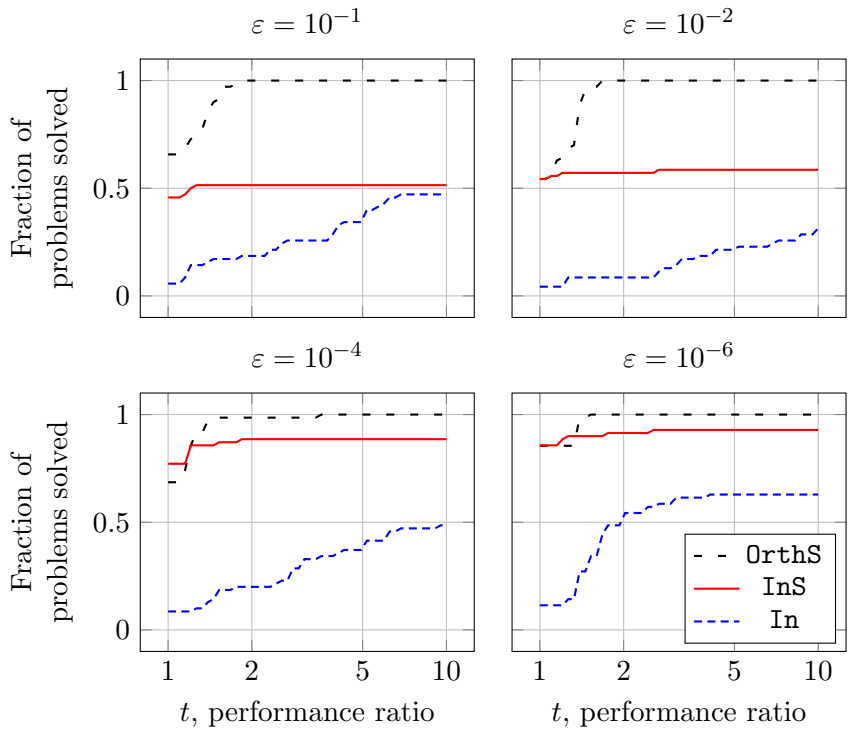



Figure 2.16: Performance profile for all the SuiteSparse experiments (Figure 2.15). Higher is better. The performance criterion is $\#CG$, the number of CG iterations. Each point $T_v(t)$ gives the fraction of problems for which variant $v$ completed with a CG iterations count less than $t$ times the 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.

### 2.4.2 Scalings with problem size

We now consider scalings, i.e., how does the algorithm perform as $N$ grows. Figure 2.17 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 Section 2.4.1 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 ($\text{size}_{\text{top}}$) typically grows like $\mathcal{O}\left(N^{1/3}\right)$, regardless of $\varepsilon$;
- for small enough $\varepsilon$, the number of CG iterations is roughly $\mathcal{O}(1)$.

Combining those two properties, we can expect (see Section 2.3.3), for small enough $\varepsilon$,

- 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 $N$.

### 2.4.3 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 ($\text{t}_{\text{fact}}$ and $\text{t}_{\text{solve}}$);
- the memory footprint ($\text{mem}_{\text{fact}}$, the number of non-zeros in the preconditioner $M$).

**SuiteSparse** Table 2.2 shows the results on two specific problems from the SuiteSparse collection [53], inline and audikw. For both problems, we see that the "sweet-spot" in terms of minimal time-to-solution is not for high $\varepsilon$, but for much smaller $\varepsilon$. For the audikw problem, the optimal is when using $\varepsilon = 10^{-2}$, and for inline, $10^{-4}$ gives optimal results. The $\text{size}_{\text{top}}$ for inline are overall much smaller than for audikw. This is usually an indication that the problem is near 2D, for which $\text{size}_{\text{top}}$ 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.

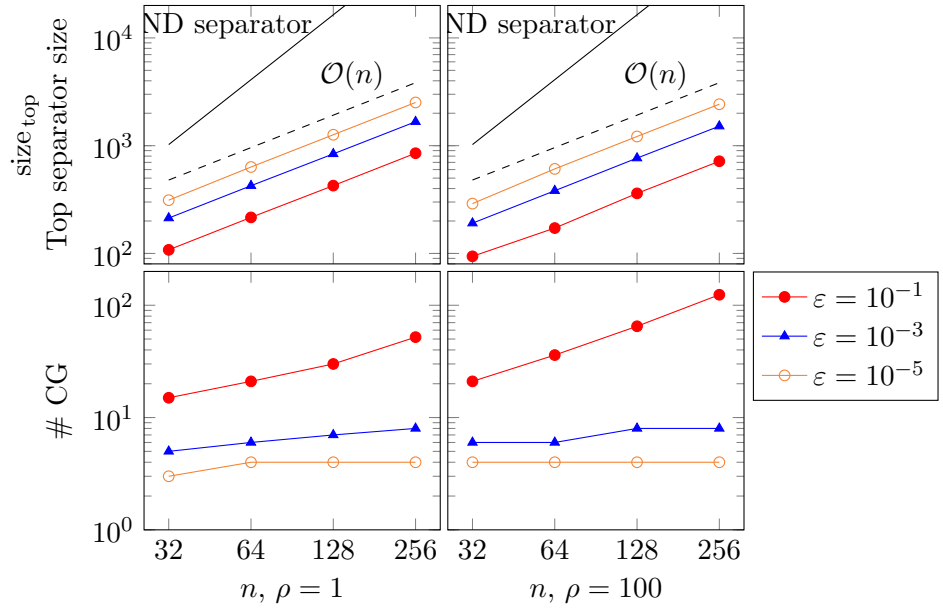



Figure 2.17: 3D $n \times n \times n$ Laplacian results for $\rho = 1$ (left) and $\rho = 100$ (right) using `OrthS`. We see that $\text{size}_{\text{top}}$ (top separator final size, i.e., right before elimination) scales like $\mathcal{O}(n)$, and that increasing the accuracy (decreasing $\varepsilon$) 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 $\varepsilon$, the CG iteration count becomes virtually constant. Both those facts mean the algorithm can be expected to have complexity $\mathcal{O}(N)$ (see Section 2.3.3).

| Problem ($N$) | $\varepsilon$ | $t_{\text{part}}$ (s.) | $t_{\text{fact}}$ (s.) | $t_{\text{solve}}$ (s.) | $n_{\text{CG}}$ | $\text{size}_{\text{top}}$ | $\text{mem}_{\text{fact}}$ ($10^9$) |
|---|---|---|---|---|---|---|---|
| audikw_1 | $10^{-1}$ | 96 | 128 | 512 | 277 | 322 | 0.46 |
| 943 695 | $10^{-2}$ | 95 | 268 | 103 | 42 | 606 | 0.73 |
| | $10^{-4}$ | 95 | 500 | 18 | 7 | 1175 | 1.08 |
| inline_1 | $10^{-1}$ | 40 | 8 | $> 224$ | $> 500$ | 1 | 0.11 |
| 503 712 | $10^{-2}$ | 41 | 13 | 44 | 80 | 13 | 0.13 |
| | $10^{-4}$ | 41 | 18 | 5 | 8 | 19 | 0.16 |

Table 2.2: 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 $\varepsilon \to 0$. The sweet spot varies for both problems. Notice that inline_1 has a very small $\text{size}_{\text{top}}$, characteristic of near-2D problems, while the top separator has a much larger size for audikw_1.

**Ice-sheet modeling problem** Table 2.3 gives the result on an ice-sheet modeling problem [145]. 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 the 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 [19].

We use two sequences of matrices with a different number of layers in the $z$-direction. We see that $\text{size}_{\text{top}}$ 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, 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 [145], 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) [126] 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 [125, 42]

on the same matrices. Overall, while the scaling with $N$ is similar, spaND exhibits better constants.

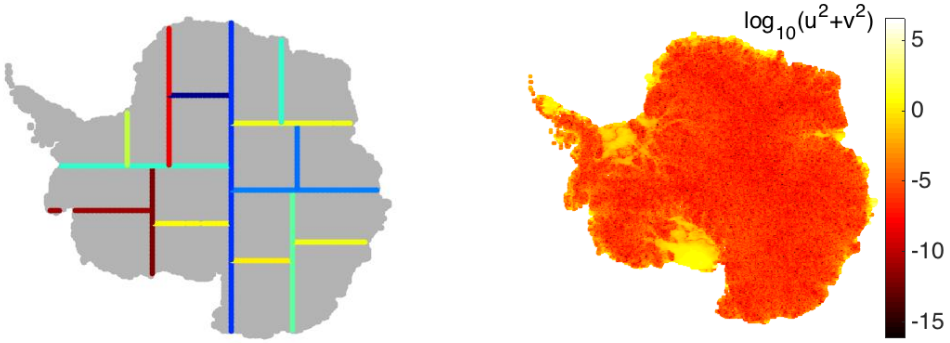
**SPE benchmark** Table 2.4 gives the results on a cubic slide of the SPE (Society of Petroleum Engineering) benchmark [47], a classical benchmark to evaluate oil & gas exploration codes. This matrix models a porous media flow. This problem is particularly challenging for direct methods since it resembles 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 $\text{size}_{\text{top}}$ and $\text{mem}_{\text{fact}}$ scaling with $N$. We see that $\text{size}_{\text{top}}$ 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.

### 2.4.4 Profiling

Figure 2.18 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 in Figure 2.18, where spaND's level 5 memory use is slightly greater than the Direct method. After that, however, it remains almost constant.

Figure 2.19 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.




| $N$ | spaND $t_{part}$ (s.) | $t_{fact}$ (s.) | $t_{solve}$ (s.) | $n_{CG}$ | $size_{top}$ | $mem_{fact}$ ($10^9$) | Direct $t_{fact} + t_{solve}$ (s.) | ILU(0) $t_{solve}$ ($n_{GMRES}$) (s.) |
|---|---|---|---|---|---|---|---|---|
| 5 layers | | | | | | | | |
| 629 544 (16 km) | 2 | 7 | 3 | 7 | 78 | 0.15 | 19 | 23 (92) |
| 2 521 872 (8 km) | 10 | 28 | 14 | 8 | 88 | 0.59 | 126 | 286 (182) |
| 10 096 080 (4 km) | 50 | 124 | 89 | 10 | 99 | 2.40 | 1036 | 7137 (720) |
| 10 layers | | | | | | | | |
| 1 154 164 (16 km) | 4 | 23 | 7 | 7 | 137 | 0.42 | 86 | 42 (93) |
| 4 623 432 (8 km) | 20 | 97 | 34 | 8 | 147 | 1.73 | 725 | 544 (181) |
| 18 509 480 (4 km) | 100 | 538 | 311 | 10 | 159 | 6.80 | — | 18680 (745) |

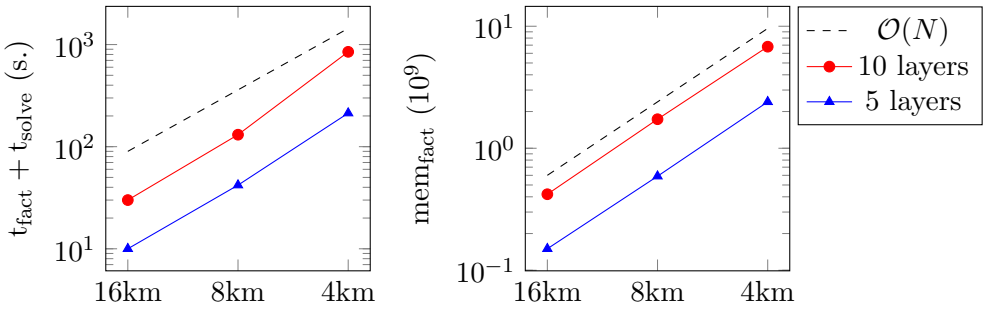
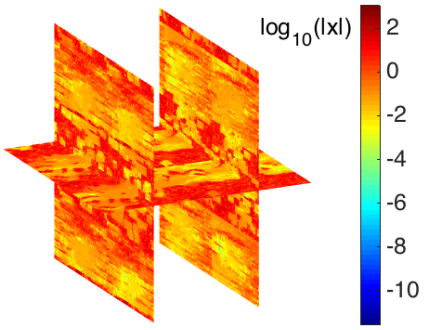



Table 2.3: Ice Sheet results. Unregular geometric partitioning, $\varepsilon = 10^{-2}$, `OrthS`. The top left picture illustrates the separators (for the top 5 levels) and the top right picture shows the solution (for a random right-hand side $b$) on log scale. — indicates the direct method ran out of memory. We run ILU with 2 ordering: layer-wise ordering and vertical column-wise ordering. The later leads 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.




| $n$ | $N = n^3$ | spaND $t_{\text{part}}$ (s.) | $t_{\text{fact}}$ (s.) | $t_{\text{solve}}$ (s.) | $n_{\text{CG}}$ | $\text{size}_{\text{top}}$ | $\text{mem}_{\text{fact}}$ ($10^9$) | Direct. $t_F + t_{\text{solve}}$ (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 | — |

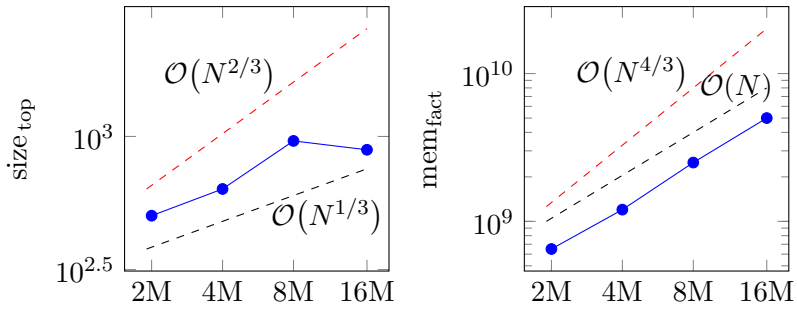



Table 2.4: SPE results. Regular geometric partitioning, $\varepsilon = 10^{-2}$, `OrthS`. "—" indicates the direct method ran out of memory. This problem is a regular cube, and hence very hard to solve using a direct method, since the separators are very large. spaND does not suffer from this problem and can solve this problem well, with a near (but not exactly) linear scaling with the problem size.

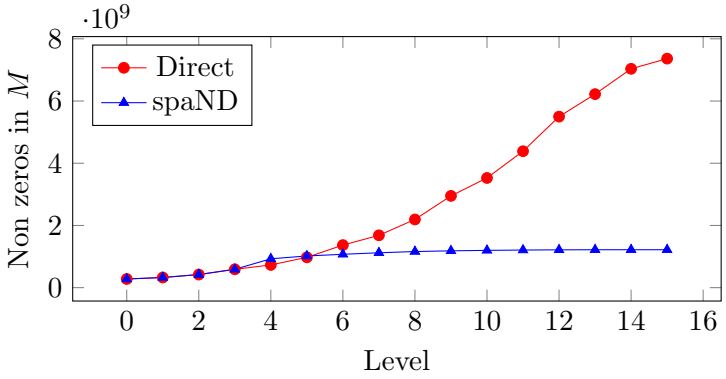



Figure 2.18: Memory profiling of the SPE 4M problem. Each dot shows the total (cumulative) memory used by the partial preconditioner up to this level in the elimination. We compare spaND to a direct method using Nested Dissection. Thanks to the sparsification (started at level 5), the memory stays well under control, while a direct method takes more and more memory as the elimination proceeds.

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.

## 2.5 Numerical Experiments (non-SPD)

We now present some results on non-SPD problems.

### Advection-diffusion

We begin with an advection-diffusion equation

$$-\nabla \cdot (a(x) \cdot \nabla u(x)) + b(x) \cdot \nabla u(x) = f$$

discretized using centered finite differences over $[0,1]^3$ with $a = 10^{-2}$ and $b = 1$, both constant. The resulting matrix is unsymmetric. As such, we use partial pivoted LU to

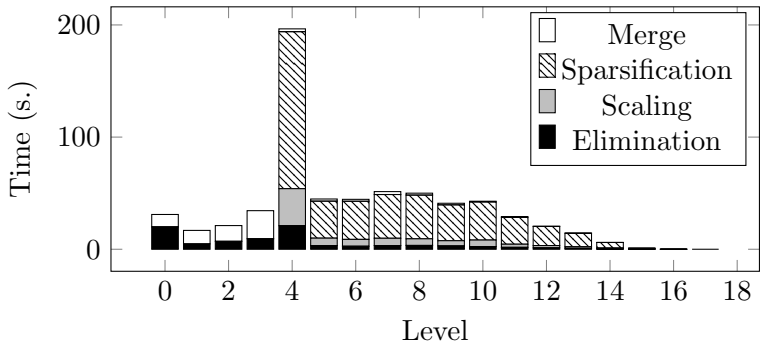



Figure 2.19: Time profiling of the SPE 16M problem. Each bar represents the time spent at each level in the elimination. Unlike direct methods, most of the compute time is spent at the first levels (near the leaves), where we have to solve many small problems. A direct method would likely be faster at 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.

eliminate and scale pivots.

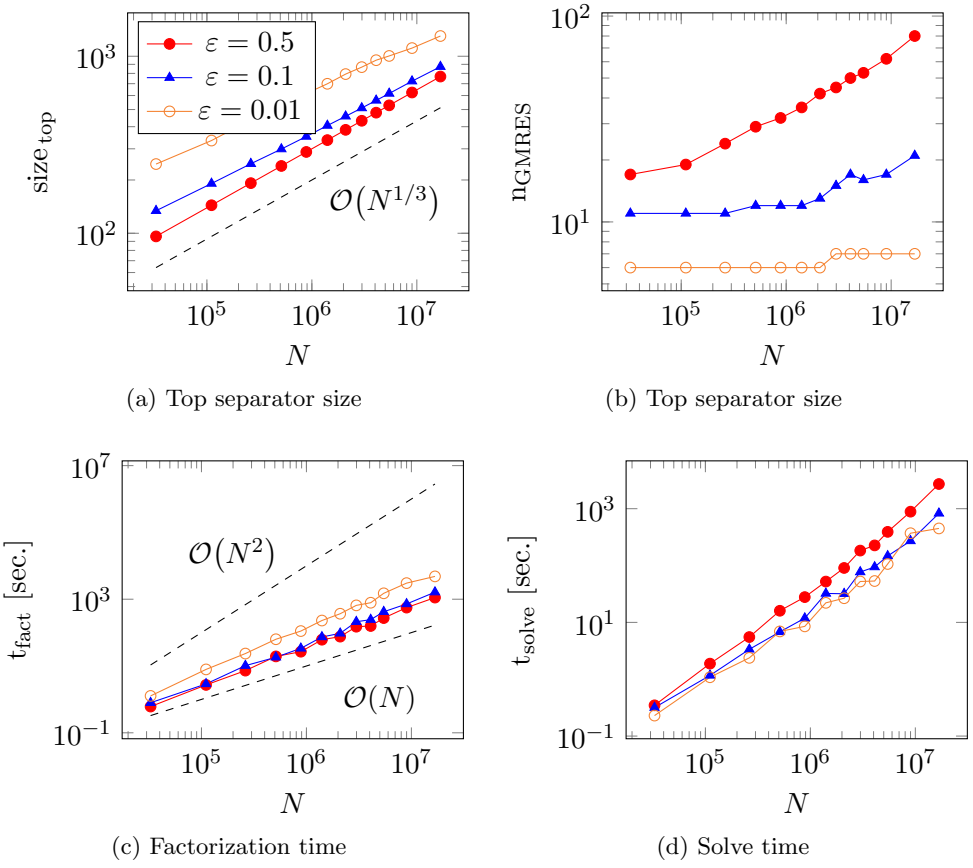
We use GMRES, a random right-hand side, and stop GMRES when the residual $\|Ax - b\|/\|b\|$ reaches $10^{-12}$. Figure 2.20 shows the results. We see that the ranks behave slightly *better* than expected, with a complexity slightly under $\mathcal{O}(N^{1/3})$. Furthermore, for a tight enough tolerance ($10^{-1}$ or under), the number of GMRES steps is almost constant.

### Quasi-definite problems

In this section, we consider symmetric matrices of the form

$$A = \begin{bmatrix} K & B \\ B^\top & C \end{bmatrix} \tag{2.7}$$

where $K \succ 0$ and $C \prec 0$. Notice that there is no constraints on $B$. Such matrices are called quasi-definite [72]. In the following, we show that a particular sequence of eliminations, scalings, and sparsifications does not destroy the quasi-definite character of $A$. We refer to sets of rows and columns associated with $K$ as $K$-clusters and to sets of rows and columns associated with $C$ as $C$-clusters.




(a) Top separator size

(b) Top separator size

(c) Factorization time

(d) Solve time

Figure 2.20: Advection-diffusion results

**Lemma 2.1.** *Assume*[2] *$A$ is quasi-definite and of the form (2.7). Then the following operations do not alter the quasi-definite character of $A$ or keep the trailing matrix quasi-definite:*

- *eliminating any $K$ or $C$-clusters;*
- *scaling any $K$ or $C$-clusters;*
- *sparsifying any previously scaled $K$ or $C$-clusters.*

*Proof.* Let us first consider eliminating a $K$-cluster. Denote it by $p$ and the other $K$-dofs by $q$. We have

$$A = \begin{bmatrix} K_{pp} & K_{pq} & B_p \\ K_{pq}^\top & K_{qq} & B_q \\ B_p^\top & B_q^\top & C \end{bmatrix}$$

After eliminating $p$, the Schur complement is

$$\begin{bmatrix} K_{qq} - K_{pq}^\top K_{pp}^{-1} K_{pq} & B_q - K_{pq}^\top K_{pp}^{-1} B_p \\ B_q^\top - B_p^\top K_{pp}^{-1} K_{pq} & C - B_p^\top K_{pp}^{-1} B_p \end{bmatrix}$$

where we notice that $K_{qq} - K_{pq}^\top K_{pp}^{-1} K_{pq} \succ 0$ since it is the Schur complement of an SPD matrix, and $C - B_p^\top K_{pp}^{-1} B_p \prec C \prec 0$ since $K_{pp} \succ 0$. Now assume we are eliminating $C$-dofs. We can apply the same reasoning on $-A$ and we conclude immediately.

Now consider scaling a $K$-cluster. With $K_{pp} = L_p L_p^\top$, this leads to

$$\begin{bmatrix} I & L_p^{-1} K_{pq} & L_p^{-1} B_p \\ K_{pq}^\top L_p^{-\top} & K_{qq} & B_q \\ B_p^\top L_p^{-\top} & B_q^\top & C \end{bmatrix}$$

which is quasi-definite since

$$\begin{bmatrix} I & L_p^{-1} K_{pq} \\ K_{pq}^\top L_p^{-\top} & K_{qq} \end{bmatrix}$$

is SPD and $C$ is unchanged. Now consider scaling a $C$-cluster. One can apply the above reasoning on $-A$ and we conclude.

Finally, consider sparsifying some $K$-cluster, $p$. Assume $p$ has been scaled such that $K_{pp} = I$. We consider sparsification involving all of $p$'s $K$-neighbors, but possibly only a

[2] Both Léopold Cambier and Bazyli Klockiewicz contributed to this proof

subset of its $C$-neighbors. This means that after sparsification, $A$ has the form

$$\left[\begin{array}{ccc|cc} I & & W_{cq} & \times & W_c \\ & I & \varepsilon & \times & \varepsilon \\ W_{cq}^\top & \varepsilon & K_{qq} & \times & \times \\ \hline \times & \times & \times & \times & \times \\ W_c^\top & \varepsilon & \times & \times & \times \end{array}\right]$$

where $\times$ indicates unchanged entries and $\varepsilon$ indicates entries that will be dropped. Dropping the $\varepsilon$ blocks lead to a trailing matrix such as

$$\left[\begin{array}{ccc|cc} I & & W_{cq} & \times & W_c \\ & I & & \times & \\ W_{cq}^\top & & K_{qq} & \times & \times \\ \hline \times & \times & \times & \times & \times \\ W_c^\top & & \times & \times & \times \end{array}\right]$$

From Theorem 2.1 we know that the top $3 \times 3$ block is SPD. In addition, the bottom $2 \times 2$ block is unchanged, and as such, is still negative definite. We conclude that the trailing matrix is quasi-definite. If we sparsify instead $C$-dofs, one can repeat the above argument on $-A$ and the conclusion follows. □

Given this result, we propose the following spaND algorithm for quasi-definite matrices. This algorithm relies on the fact that, if $K$ and $C$-dofs are eliminated and sparsified separately, then $A$ remains quasi-definite. As such we can guarantee pivots to always be non-singular in exact arithmetic. Note that since $A$ is not SPD, this does not imply stability. We then first perform a classical MND ordering and clustering of $A$, but then further divide every cluster into $K$ and $C$-dofs. We call $K$ and $C$ clusters originating from the same interface siblings, and we never merge $K$ and $C$-clusters. Hence, $K$ and $C$-dofs remain separated throughout the algorithm. The algorithm then sparsifies $K$-interfaces and $C$-interfaces separately, considering every edge except edges to their $C$ or $K$-sibling. Algorithm 2.4 shows the complete algorithm.

We now present results on the Biot problem. This problem models a deformable porous medium, and the resulting matrix has the quasi-definite structure as described before. The $K$-block corresponds to velocities and $C$-block to pressure unknowns (see [54, Section 5.3]).

**Algorithm 2.4** spaND for quasi-definite matrices

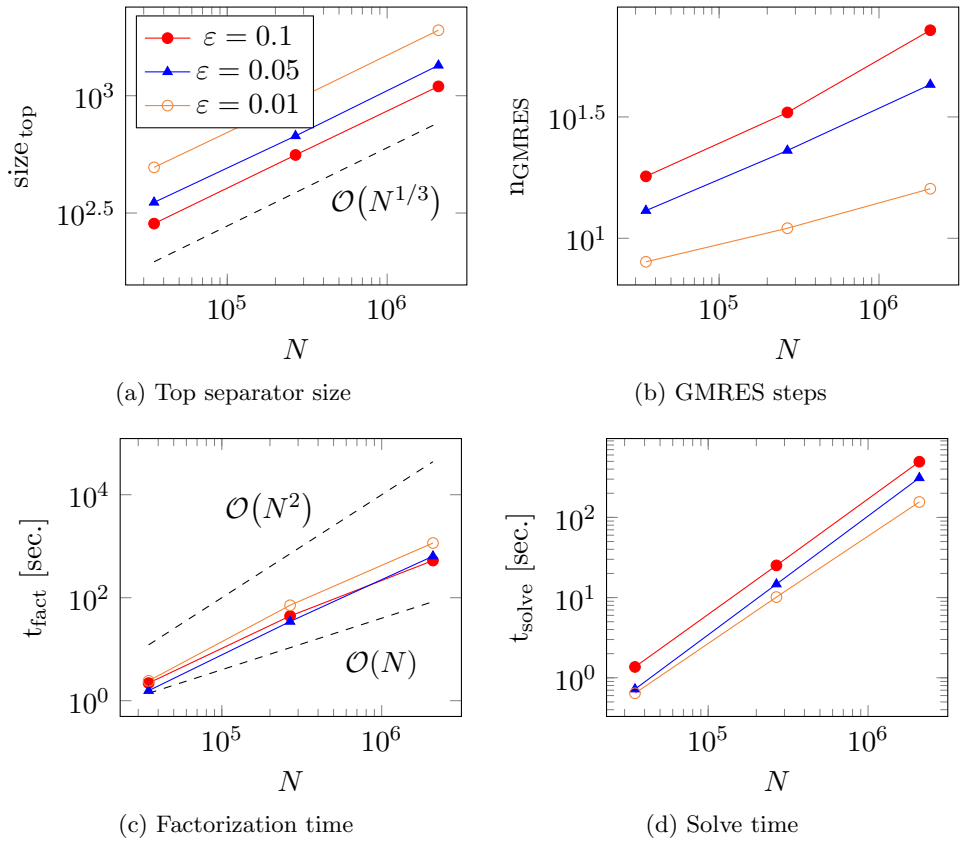
**Require:** Quasi-definite $A$, Maximum level $L$
  Compute a ND ordering for $A$, infer interiors, separators and interfaces (see Section 2.2.3)
  Further divide every interface into $K$ and $C$-clusters
  **for all** $\ell = 0, \ldots, L-1$ **do**
    **for all** $\mathcal{I}$ interior **do**
      Eliminate $\mathcal{I}$
    **end for**
    **for all** $\mathcal{B}$ interface between interiors **do**
      Scale $\mathcal{B}$
    **end for**
    **for all** $\mathcal{B}$ $K$-interface between interiors **do**
      Sparsify $\mathcal{B}$, not including the edge from $\mathcal{B}$ to its sibling
    **end for**
    **for all** $\mathcal{B}$ $C$-interface between interiors **do**
      Scale $\mathcal{B}$
    **end for**
    **for all** $\mathcal{B}$ $C$-interface between interiors **do**
      Sparsify $\mathcal{B}$, not including the edge from $\mathcal{B}$ to its sibling
    **end for**
  **end for**

The matrices are generated using the Sfepy [49] multiphysics example, available at `http://sfepy.org/doc/examples/multi_physics-biot.html`. Figure 2.21 shows the results. We see that the top separator size, again, follows closely $\mathcal{O}(N^{1/3})$. The GMRES iteration count grows slowly for $\varepsilon = 10^{-2}$. The factorization time is well below $\mathcal{O}(N^2)$, though slightly larger than $\mathcal{O}(N)$.

## 2.6 Conclusion

In this chapter, 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.




(a) Top separator size

(b) GMRES steps

(c) Factorization time

(d) Solve time

Figure 2.21: Biot problem results

Prior work in this area included the HIF algorithm [95]. While our work resembles it, HIF is limited to $n \times n \times n$ regular problems [95] and does not use either the block diagonal scaling or orthogonal transformations. The LoRaSp algorithm [125] is also similar. LoRaSp's performances however may degrade when the ranks at the leaf level are not small and it does not have the same sparsity guarantees [42]. 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 also discuss the algorithm when used on non-SPD matrices. We showed through extensive numerical experiments that the scaling has a large impact on the preconditioner's accuracy. In addition, the use of orthogonal transformations implies that the algorithm does not break down even when $\varepsilon \approx 1$ when the matrix is SPD. gr 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 $c$ unknowns 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.

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 ranks. 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.

# Chapter 3

# TaskTorrent

Part of this chapter contains the full text of [34]. This work is © 2020 IEEE. Reprinted, with permission, from Léopold Cambier, Yizhou Qian, Eric Darve. TaskTorrent: a Lightweight Distributed Task-Based Runtime System in C++. 2020 IEEE/ACM 3rd Annual Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM). 2020.

## 3.1 Introduction

### 3.1.1 Parallel runtime systems

Classical parallel computing has traditionally followed a fork-join (as in OpenMP) or bulk-synchronous (MPI) approach. (Figure 3.1a shows the skeleton of a typical MPI program). This has many advantages, including ease of programming and predictable performance. It has however a key downside: many points of synchronization during execution are added, even when not necessary.

Runtime systems take a different approach. The key concept is to express computations as a graph of tasks with dependencies between them (Figure 3.1b). This graph is directed and acyclic, and we will later refer to it as the task DAG. Given the DAG, the runtime system can extract parallelism by identifying which tasks can run in parallel. Tasks are then assigned to processors (either individual cores, nodes, accelerators, etc). The advantage of this method is that it removes all unnecessary synchronization points.

### 3.1.2 Existing approaches to describe the DAG

A key design choice in runtime systems is how to express the DAG. At a high-level, two approaches have been primarily used.

#### Sequential Task Flow (STF)

In this approach, the graph is discovered by the runtime using a sequential semantics, that is, typically, on each node, a single thread is responsible for building the DAG. Different mechanisms to compute task dependencies can be used. Often, this takes the form of inferring dependencies based on specifying data sharing rules (e.g., READ, WRITE, READWRITE).

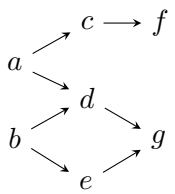
This is the approach taken by Legion/Regent [13] [1] and StarPU [11]. In both, the user first defines data regions and tasks operating on those regions (as inputs or outputs). Regent maintains a global view of the data, and data regions correspond to a partitioning of the data. The user is also able to write mappers to indicate how to map and schedule tasks to the available hardware. StarPU uses data handles referring to distributed memory buffers. The program is then written in a sequential style (with for loops, if/else statements, etc.), creating tasks on previously registered data regions. The runtime system then discovers task dependencies, builds the DAG and executes tasks in parallel.

The key in the STF approach is that the DAG has to be discovered through sequential enumeration. This restriction may have performance implications but is attractive to the programmer since the program is easy to write and understand.

[1] Legion is the name of the lower-level C++ API, while Regent is the name of the higher-level language based on Lua.

```
for (auto i : local0)
  compute0(i);
if ([...])
  { MPI_Send(m, ...); }
else
  { MPI_Recv(m, ...); }
for (auto i : local1)
  compute1(i);
```

(a) A typical MPI program




(b) Example of DAG of tasks.

Figure 3.1: More parallelism can be extracted using a tasks DAG: task $d$ needs to wait for tasks $a$ and $b$. However, task $f$ can run as soon as task $c$ has finished.

```
/** Define task **/
void task(...) {...}
/** Register data **/
data = [...]
/** Process DAG **/
for(k ...)
    task(data[k], data[k1], ...)
```

(a) STF based program. Dependencies are inferred through data sharing rules.

```
/** Task deps. expressed
 *  as functions of K **/
in_deps  = (K k){...}
task     = (K k){...}
out_deps = (K k){...}
/** Seed tasks **/
for(k in kinit)
    start(k)
```

(b) PTG based program. Task dependencies are defined using functions over `K`. Computation is triggered by seeding the initial tasks.

Figure 3.2: Schematic of STF and PTG programs.

### Parametrized Task Graph (PTG)

The PTG approach is another method to express the DAG. Using some index space (`K`) to index all tasks, functions of `K` are used to express tasks and their dependencies. As an example, the DAG could be defined by specifying three functions of `K` (other choices are possible): one for the in-dependencies, one for the computational task itself, and one for the out-dependencies. By running these functions as needed, the runtime discovers the DAG dynamically.

PaRSEC [26] takes that approach, using a custom language (JDF) to express the PTG. In PaRSEC, in and out-dependencies specifications contain both tasks and data.

The PTG format has multiple advantages. Since task in/out-dependencies can be independently queried at any time, it simplifies task management, leading to minimal overhead during execution. It also naturally scales by parallelizing both the DAG creation and DAG execution. In contrast, an STF code uses, in its purest form, a single thread to discover the DAG. It also removes the need to store in memory large portions of the DAG of tasks. Instead, the runtime can query the relevant functions only as needed and discover the DAG piece by piece.

The main drawback of the PTG approach is that the program no longer has a sequential semantics, which makes it harder to understand the program's behavior at first sight. Figure 3.2 illustrates at a higher level the differences between the STF and the PTG approach.

## 3.2 Previous work

**Runtime systems** As mentioned in Section 3.1.2, other task-based runtime systems exist. We highlight some of their characteristics. PaRSEC [26] is a runtime system centered around dense linear algebra. It takes the PTG approach but uses a custom programming language, the JDF. This can make adoption harder for new users. Legion [13] is a general-purpose STF runtime. It has many features and can be used from C++ but requires the user to express everything using Legion's data structures. It is also intended to be used primarily with GASNet [23] and not MPI. Regent [139] proposes a higher-level language on top of Legion, making programming more productive. Unfortunately, obtaining high performance requires the user to program directly the mapper which is time-consuming and requires a detailed understanding of the inner workings of Legion. Finally, StarPU [11] uses C++ and is STF-based. The data is initially distributed by the user like a classical MPI code, and various scheduling strategies can be used to further improve performance. However, user data still has to be wrapped using StarPU's data structures.

In designing `TTor` we chose to focus on the following features. The message-passing paradigm requires the programmer to distribute data but makes it easy to reach good performance because it minimizes the need for global synchronization and communication. It also simplifies the library implementation. MPI and C++ make integration into other codes easier. Active messages are necessary because of the asynchronous nature of computations. Finally, the PTG approach leads to a minimal runtime overhead. Note however that the choice of PTG has drawbacks: it can be difficult for the programmer to reason about task dependencies. This can be easier in some applications (like linear algebra) than others. `TTor` also does not consider concepts like memory affinity or accelerators at the moment. This is reserved for future work.

**Task-based parallelism** Task-based parallelism is now a common feature of many parallel programming systems.

Cilk [98, 67] introduced a multi-threading component to C in 1996, and Cilk-5 introduced `spawn` and asynchronous computations. Many other efforts followed, including OpenMP [51] (with tasking introduced in version 3.0), Intel TBB [128] (where task DAGs can be expressed), Cilk Plus [130], XKaapi [70], OmpSs [56], Superglue [146], and the SMPSs programming model [121, 122]. The Plasma [3, 4] (for CPU) and Magma [147] (for CPU and GPU) libraries are replacements for multithreaded LAPACK, where parallelism is obtained

through tiled algorithms using a dynamic runtime, Quark [162].

Notice that all the previously mentioned work is typically only useful in a shared-memory context. In particular, there is no support to let one rank trigger (or fulfill the dependency of) a task on another rank.

**Distributed programming** An explicit goal of `TTor` is to provide support for distributed computing.

The most common distributed programming paradigm is using explicit message passing like in MPI. In MPI, ranks are completely independent and only communicate with each other through explicit message passing. Charm++ [99] takes an object-oriented approach. It exposes *chares* which are concurrent objects communicating through messages. We also mention `DARMA/vt` [108], a tasking and active message library in C++, with other features such as load balancing and asynchronous collectives. We decided to implement one-sided active messages on top of classical MPI non-blocking sends and receives given its wide and ubiquitous support. This facilitates integration with existing codes. Finally, in the PGAS (partitioned global address space) model (like GASNet [23]), each rank can access a global address space through read (get) and write (put) operations. Chapel [31], Fortran Co-arrays [116], UPC [58] and UPC++ [164] are examples of PGAS-based parallel programming languages. One of the arguments of the PGAS model is that it increases developer productivity by presenting a simpler programming model compared to explicit message passing.

**Active messages** One-sided active messages is another important feature of TaskTorrent. Von Eicken et al. [151] argued in 1992 that active messages are a powerful mechanism to hide latency and improve performance.Active messages are also a central part of UPC++ where they resemble the ones in `TTor`. In UPC++, however, remote data is referred to using global data structures, while `TTor` tends to use the C++ variable capture mechanism in lambda functions. `TTor` takes the same custom-compiler free approach as UPC++: it only requires a C++ compiler and no other custom compiler on top of it (like in UPC or PaRSEC).

**Hardware-aware programming** While this is not relevant to this work, we note that another important aspect of high-performance computing is to fully exploit the complex heterogeneous machines.

For instance, machines may have various number of CPUs, custom accelerators (GPUs), and deep memory hierarchies far from what is exposed by a flat memory model. Programming and taking advantage of this is a non-trivial task for the programmer.

Some runtime systems try to take advantage of this automatically, for instance, Legion and StarPU. Other works in this direction include X10 [41], Sequoia [60], HwLoc [30] (Hardware Locality, used by StarPU), and Phalanx [68].

### 3.2.1 Contributions

In this chapter, we present TaskTorrent (`TTor`). `TTor` is a lightweight, distributed task-based runtime that uses a PTG approach. Our main contributions are:

- We show how to combine a PTG approach with one-sided active messages.
- A mathematical proof is provided for the correctness of our implementation.
- We benchmark `TTor` and show that it matches or exceeds the performance of StarPU on sample problems.

`TTor` has a couple of notable features compared to existing solutions

- It is a C++14 library with no dependencies other than MPI.
- `TTor`'s implementation leads to a small overhead and handles well small task granularity (about 10 $\mu s$ and up). This means that `TTor` can be used on any existing code, without needing to fuse or redefine tasks, or change existing algorithms.
- Default options in `TTor` are designed to provide good performance "out-of-the-box" without requiring the user to tune or optimize internal parameters or functionalities of the library.
- The user can use their own data structures without having to wrap their data in opaque data structures.
- It is perfectly scalable in the following sense. Consider a provably scalable numerical algorithm (e.g., there exists an iso-efficiency curve). Assume that (1) the parallel computer is composed of nodes with a bounded number of cores, but with an unbounded number of nodes, and (2) that each node in the DAG has a bounded number

of dependencies. Then if the algorithm is executed using `TTor` it will remain scalable. Said more simply, `TTor` does not introduce any parallel bottleneck.

We emphasize that `TTor` is a general-purpose runtime system. The applications in this chapter are mostly in linear algebra, but there are no features or optimizations that are specific to linear algebra in this version of `TTor`.

## 3.3 TaskTorrent

`TTor` uses a PTG. The DAG is expressed by providing at least three functions: (1) one returning the number of in-dependencies of every task; (2) one that runs the computational task and fulfills dependencies on other tasks; (3) one returning the thread each task should be mapped to (an option is provided to bound the task to the thread or leave it stealable). When their dependencies are satisfied, tasks are inserted into a thread pool, where a work-stealing algorithm keeps the load balanced between the threads.

Tasks then run and fulfill other tasks' dependencies, locally (on the same rank) or remotely on a different rank. In the case of remote dependencies, since all computations are asynchronous, the receiver rank cannot explicitly wait for data to arrive. Hence, one-sided active messages are used. An active message (AM) is a pair (function, data). Once the AM arrives on the receiver, the function is run with the data passed as arguments. This is typically used to store the data and fulfill dependencies, eventually triggering more tasks.

This approach means `TTor` never needs to store the full DAG. Task dependencies are queried only when needed, and the DAG is discovered piece by piece. In particular, `TTor` becomes aware of the existence of a specific task **only** when a task fulfills its first dependency. This makes `TTor` scalable and lightweight. The full DAG is never stored or even explored by any specific thread or rank, and the task management overhead is minimal. Figure 3.3 illustrates this local DAG + AM model.

### 3.3.1 API Description

`TTor`'s API can be divided into two parts, a shared memory component (expressing the PTG) and a distributed component (used for AMs). The combination of those two features is what distinguishes `TTor` from other solutions and is one of the factors that makes `TTor` lightweight.

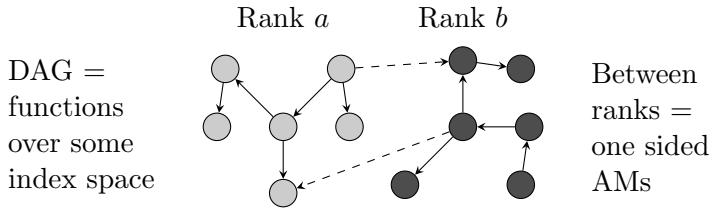



Figure 3.3: The model of `TTor`: a distributed graph of tasks expressed using a parametrized task graph (solid arrows), with explicit active messages (dashed arrows) between ranks to asynchronously insert/trigger tasks.

### Shared memory components

**Threadpool** A `Threadpool` is a fixed set of threads that receive and process tasks. A `Threadpool` with `n_threads` threads can be created by `Threadpool tp(n_threads, &comm)`. (`comm` is a `Communicator`; see Section 3.3.1). Tasks can be inserted directly in the thread pool, but typically this is done using a `Taskflow`. The thread pool joins when calling `tp.join()`. This returns when all the threads are idle and all communications have completed. Section 3.3.2 explains in detail the distributed completion mechanism.

**Taskflow** A `Taskflow<K> tf` (for some index space `K`, typically an integer or a tuple of integers) represents a Parametrized Task Graph. It is created using `Taskflow<K> tf(&tp)` where `tp` is a `Threadpool`. It is responsible for managing task dependencies and automatically inserting tasks in `tp` when ready. At least three functions have to be provided:

- `(int)indegree(K k)` returns the number of dependencies for task `k`.
- `(void)task(K k)` indicates what task `k` should be doing when running. Typically this is some computational routine followed by the trigger of other tasks. For instance, task `k1` can fulfill one dependency of task `k2` by `tf.fulfill_promise(k2)`.
- `(int)mapping(K k)` indicates what thread should task `k` be initially mapped to.

In general, tasks can be stolen between threads to avoid starvation. This is done using a work-stealing algorithm. `tf.set_binding(binding)` can be used to make some tasks bound to their thread. Optional priorities can also be provided through `tf.set_priority(priority)`. Finally, `tf.fulfill_promise(k)` is used to fulfill one of the dependencies of task `k` on Taskflow `tf`. See Figure 3.4.

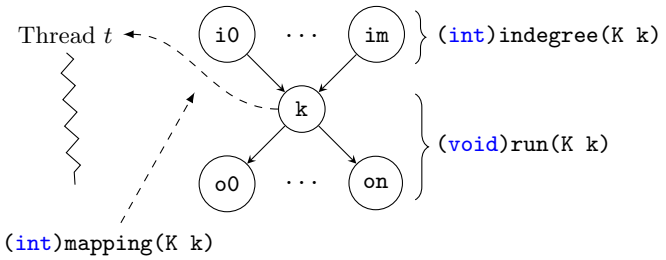



Figure 3.4: The `Taskflow<K>` API. `(int)indegree(K k)` returns the number of incoming dependencies of task `k`. `(void)run(K k)` indicates what function to run. `(int)mapping(K k)` returns what thread the task should be mapped (but not bound) to.

### Distributed memory components

Active Messages (AMs) are used to allow tasks on rank $a$ to trigger tasks on rank $b \neq a$ without rank $b$ explicitly waiting for messages.

An AM is a pair `(function, payload)`. When an AM is sent from rank $a$ to rank $b$, the payload is sent through the network, and upon arrival, the function (with the associated payload passed as argument) is run on the receiver rank. This allows for instance to store the payload at some location in local memory and then trigger tasks.

**Active message** An `ActiveMsg<Ps...> am` pairs a function `(void)fun(Ps... ps)` and a payload `ps`. Note that `Ps...` is a variadic template: different types can be used as arguments. A `view<T>` can be used to identify a memory buffer (i.e., a pointer and a length) and is built as `view<T> v(pointer, num_elements)`.

The AM can be sent to rank `dest` over the network using `am->send(dest, ps...)`. When sent, the payload is serialized on the sender, sent over the network, deserialized on the receiver, and the function is run as `fun(ps...)`. The payloads are always serialized in a temporary buffer by the library. As such, the user-provided arguments can be immediately reused or modified as soon as `send` returns. `am->send` is thread-safe and can be called by any thread.

`TTor` also provides *large* active messages. A large AM can be used to avoid temporarily copying large buffers. A large AM payload is made of one `view<T>` and a series of arguments `Ps...`. The view will be sent and received directly without any extra copy. It is associated with three functions: (1) a function to be run on the receiver rank that returns a pointer to a user-allocated buffer, where the data will be stored; (2) a function to be run on the

receiver rank to process the data upon arrival; (3) a function to be run on the sender rank when the buffer on the sender side can be reused. This is an important feature to avoid costly copies and/or when memory use is constrained.

**Communicator** A `Communicator comm` is a C++ factory to create AMs and is responsible for sending, receiving, and running AMs. `Communicator comm(mpi_comm)` creates a communicator using the `mpi_comm` MPI communicator. An AM can then be created by `am = comm.make_active_msg(f)` where `f` is a `(void)f(Ps...)` function. AMs always have to be created in the same order on all ranks because we need to create a consistent global indexing of all the AM that need to be run.

### Example

The following shows how the different components can be used together. This assumes `compute(k)` does the computation related to task `k`. In addition, `mapping(k)` returns a thread for task `k` (which is typically `k % n_threads`), `n_deps(k)` gives its number of in-dependencies, `deps(k)` iterates through its out-dependencies, and `task_2_rank(k)` returns the rank it is mapped to. `n_threads` is the desired number of threads to use. We assume that task outputs are stored in `data`. The execution of the DAG starts when the initial tasks are seeded and finishes when `tp.join()` returns.

```
/** Initialize structures **/
Communicator comm(MPI_COMM_WORLD);
Threadpool tp(n_threads, &comm);
Taskflow<int> tf(&tp);
/** Create active message **/
am = comm.make_active_msg(
  [&](int d, int k, payload pk) {
    data[k] = pk;
    tf.fulfill_promise(d);
  });
/** Define Taskflow **/
tf.set_mapping(mapping);
tf.set_indegree(n_deps);
tf.set_run([&](int k) {
  compute(k);
  for (auto d : deps(k)) {
    int dest = task_2_rank(d);
    if (dest == my_rank) {
      tf.fulfill_promise(d);
    } else {
      am->send(dest, d, k, data[k]);
    }
```

```
  }
});
/** Start initial tasks **/
for (auto k : initial_tasks)
  tf.fulfill_promise(k);
/** Wait for completion **/
tp.join();
```

### 3.3.2 Implementation Details

#### Taskflow and threadpool

The thread pool is implemented with two `std::priority_queue<Task*>` per thread, storing the ready-to-run tasks. Since some tasks can be stolen and others not, each thread has two queues. The priority queues are protected using `std::mutex` so that tasks can be inserted into a thread queue by any other thread.

One of the main goals of the `Taskflow<K>` implementation is to support arbitrary task flows with keys belonging to any domain. Hence, we store dependencies in a `std::unordered_map<K,int>`. Furthermore, to avoid having one central map storing all dependencies (whose access needs to be serialized), the map is distributed across threads. Task's dependencies are split among the threads using the mapping function: the dependency count of task `k` is stored in the map associated to thread `mapping(k)`. Each distributed map is always accessed by the same thread, preventing data races.

Figure 3.5 summarizes the relationships between the different components.

#### Active messages and communication thread

Active messages (AM) are implemented by registering functions on every rank in the same order. Each AM then has a unique ID shared across ranks. This ID is later used to retrieve the function on the receiver side. Communication is performed using MPI non-blocking sends and receives. The `Communicator` maintains three queues:

1. a queue of serialized and ready-to-send messages;
2. a queue of send messages, to be later freed when the associated send completes;
3. a queue of receive messages, to be later run and freed when the associated receive completes.

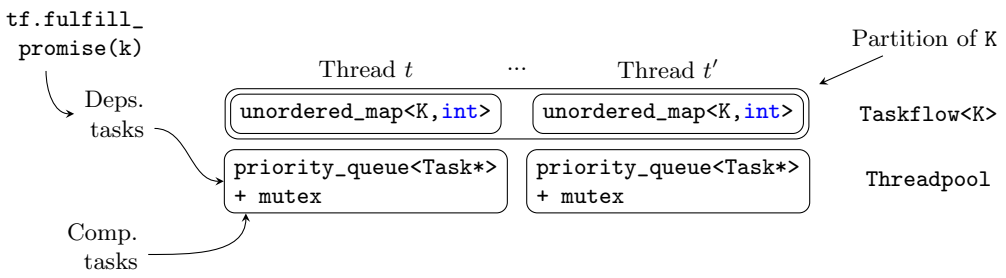



Figure 3.5: `Taskflow` and `Threadpool` implementation. Each thread has two queues (only one shown here) of ready tasks (`priority_queue<Task*>`) protected by a `mutex`. A distributed map (`unordered_map<K,int>`) is used to store task dependency counts. `tf.fulfill_promise(k)` is used to update (or initially create) the dependency count of task `k`. To do so, a task "Deps. task" (which cannot be stolen) decreasing its count is inserted on thread `mapping(k)`. When the count reaches 0, the computational task "Comp. task" is created and inserted into `mapping(k)`'s ready queue. In general, non-bound tasks can be stolen between threads.

On the sender side, when sending (thread-safe) an active message `am->send(dest, ps...)`, the various arguments `ps...` are first serialized into a buffer, along with the AM ID. The buffer is placed in a queue in the communicator. When calling `progress()`, that buffer will eventually be sent using `MPI_Isend` and later freed when the send has completed.

On the receiver side, calling `progress()` performs the following:

1. As long as it succeeds, it calls `MPI_Iprobe` to probe for incoming messages and (1) retrieves the message size using `MPI_Getcount`, (2) allocates a buffer, and (3) receives the message using `MPI_Irecv`.

2. It goes through all received messages and tests for completion with `MPI_Test`. If it succeeds (1) it retrieves the AM using the ID from the buffer, (2) deserializes the buffer, passes the arguments to the user function, and runs the user function.

MPI tags are used to distinguish (1) messages of size smaller or larger than $2^{31}$ bytes, and (2) regular and large AMs.

We say that an AM is queued when calling `am->send(...)` and processed once their associated function has finished running. Since AM are run as soon as their associated buffer is ready, they can be *processed* on the receiver in a different order than they are *queued* on the sender. However, messages are always *received* (calling `MPI_Irecv`) in the same order they are *sent* (calling `MPI_Isend`). As such, when an AM is processed, all previously sent AM are pending in the receiver's queues.

Figure 3.6 illustrates the one-sided AM implementation.

### Distributed completion algorithm

We now discuss the distributed algorithm to determine completion. We present the algorithm along with a proof of correctness. The difficulty in detecting completion lies in the fact that even if all taskflows are idle, the program may not be finished since active messages (AM) may still be in-flight. An example of a flawed strategy is to request that all ranks send an `IDLE` signal to one rank when they have no tasks running. This strategy will lead to early termination of the program in many cases. Hence, detecting completion is non-trivial in a distributed setting.

**Completion** In the following, we will consider a series of events such as queuing and processing messages, checking certain conditions, etc. Within a thread, we assume a total ordering between events which lets us associate each of them with a unique real number which we informally call "time". We consider a program with two threads per rank: a main (MPI) thread responsible for MPI communication (asynchronous sends and receives) and AMs, and a worker thread responsible for executing all the user-defined tasks (in practice, the worker thread may be a thread pool, but this is not relevant).

We say that an AM is **queued** on a sending rank when it is issued either by the worker or the main thread. When issued by a worker, we assume that queueing always finishes before the completion of the enclosing task. An AM is **processed** on the receiving rank by the main thread. We assume that if an AM results in a task being inserted in the task queue of the worker thread, this insertion must complete before the end of the enclosing AM.

To define our ordering between ranks, we assume that if a message is queued at time $t$ and processed at time $t'$ then $t' > t$. We assume that messages that are queued are eventually processed if the network and all ranks are idle except for handling these messages (progress guarantee), and that all communications are non-blocking (no deadlocks are possible). `TTor` satisfies those assumptions by construction.

**Definition 3.1** (Completion)**.** *We say that $\{t_a\}_a$ is completion time sequence if:*

- *Rank $a$ is idle at time $t_a$ for all $a$;*

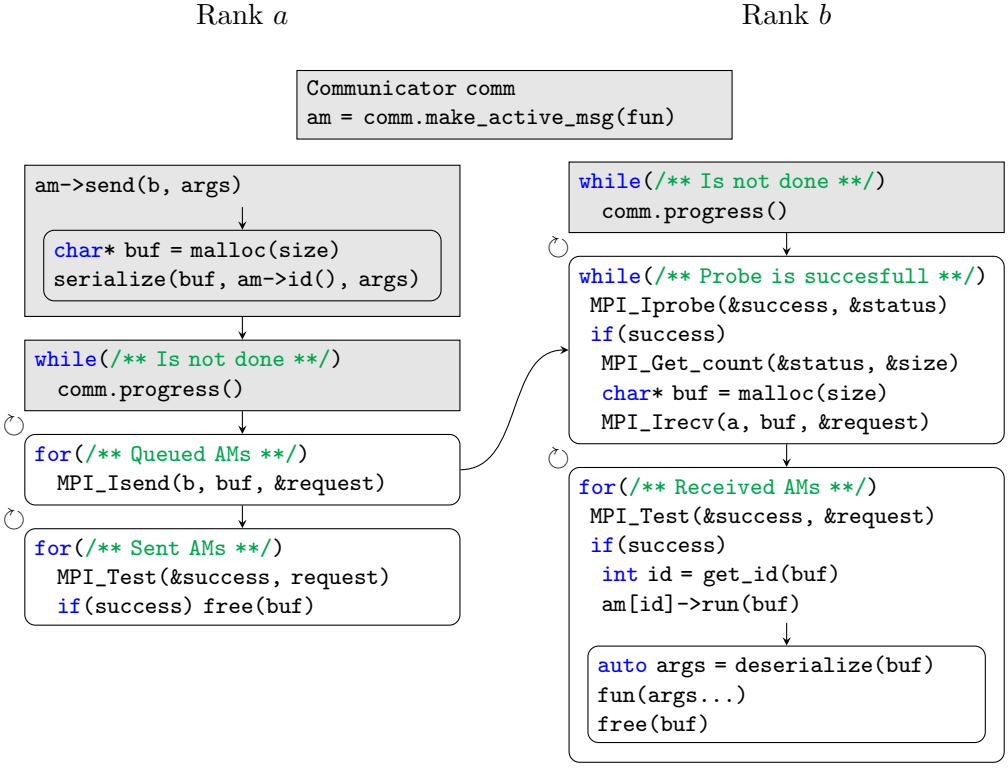



Figure 3.6: Active messages implementation. User's interface is in grey, `TTor`'s implementation in white. This shows the main code path through which an active message goes through. On the sender's side, after a call (from any thread) to `am->send`, the AM is serialized and queued in the communicator. The main thread then repeatedly call `progress()`, which internally call `MPI_Isent` and, when the send has completed, frees the buffer. The receiver also calls `progress()`, which internally probes for incoming messages (`MPI_Iprobe`) and calls receive (`MPI_Irecv`) when the probe is successful. When the receive has completed, the main thread on the receiver rank deserializes the payload and runs the active message. Circular arrows ⟳ indicate that the associated box is internally repeated as many times as necessary. Note that `progress()` has to be called until all communications have completed.

- *For any pair of ranks $(a, b)$ and all AMs from $a$ to $b$, all AMs queued before $t_a$ have been processed on $b$ before $t_b$.*

We can prove that this definition implies the intuitive definition of completion, which is that, after $t_a$, if we keep the program running, rank $a$ remains idle forever.

*Proof.* Assume that the conditions in the definition hold. Since at $t_a$, $a$ is idle, it can only resume activity after receiving a message. Assume that $a$ processes a message at time $t > t_a$. Pick $a$ such that $t$ is minimal. There must exist a rank $b$ that queued this message. Since $b$ has sent this message but $b$ was idle at time $t_b$, it must have processed a message at some time $t'$ with $t' > t_b$. Since $t' < t$, this contradicts our assumption that $t$ is minimal. Therefore no such message can exist. □

**Completion algorithm** The algorithm is based on making sure, after all ranks are idle, that the number of messages sent is equal to the number of messages received. For this verification to work, we need to proceed in two steps. We define $q_a(t)$ (resp. $p_a(t)$) to be the total number of queued (resp. processed) AMs on rank $a$ at time $t$. In step 1, once idle, rank $a$ will send to the main rank, at time $t^-$, the pair $(q_a(t^-), p_a(t^-))$. Then the main rank will request from $a$ a confirmation. In step 2, at time $t^+ > t^-$, if the value of $(q_a, p_a)$ has not changed on rank $a$, $a$ will send back a confirmation message. This leads to the following definition

**Definition 3.2** (Synchronization time)**.** *Assume that for all ranks $a$, we have defined a pair of times $(t_a^-, t_a^+)$ with $t_a^- < t_a^+$. We say that $\bar{t}$ is a synchronization time for $(t_a^-, t_a^+)$ if*

$$t_a^- < \bar{t} < t_a^+, \quad \text{for all } a$$

Before giving the exact algorithm, we prove a sufficient condition to establish completion.

**Lemma 3.1.** *Let $p_a(t)$ (resp. $q_a(t)$) be the number of processed (resp. queued) AMs on rank $a$ at time $t$. Assume that there exists a synchronization time $\bar{t}$ for $(t_a^-, t_a^+)$ and that for all $a$*

- *the worker thread on rank $a$ is idle at $t_a^-$;*
- *$\bar{p}_a = p_a(t_a^-) = p_a(t_a^+)$ (no new processed AM between $t_a^-$ and $t_a^+$);*
- *$\bar{q}_a = q_a(t_a^-) = q_a(t_a^+)$ (no new queued AM between $t_a^-$ and $t_a^+$);*

- $\sum_a \bar{q}_a = \sum_a \bar{p}_a$.

*Then the sequence $\{t_a^-\}_a$ is a completion time sequence for the execution.*

*Proof.* Let us first prove that rank $a$ is idle during the entire period $[t_a^-, t_a^+]$. Rank $a$ is idle at $t_a^-$. Since $p_a(t_a^-) = p_a(t_a^+)$, no AM was processed at any time $t \in [t_a^-, t_a^+]$. So no tasks may have been inserted in the worker task queue by the main thread. Hence, rank $a$ is idle during $[t_a^-, t_a^+]$.

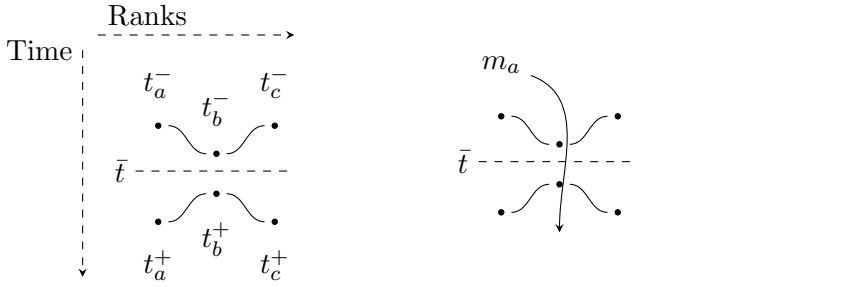
Now consider a message $m$ queued at $t_q$ on rank $q$ and processed at $t_p$ on rank $p$. Recall that $\sum_a \bar{q}_a = \sum_a \bar{p}_a$. Since no messages are queued or processed on $[t_q^-, t_q^+]$ and $[t_p^-, t_p^+]$, we have four possibilities:

- $t_q < t_q^-$ and $t_p < t_p^-$. Then $m$ contributes $+1$ to both sides of the equality $\sum_a \bar{q}_a = \sum_a \bar{p}_a$;
- $t_q > t_q^+$ and $t_p > t_p^+$. Them $m$ does not contribute to either side of the equality $\sum_a \bar{q}_a = \sum_a \bar{p}_a$;
- $t_q > t_q^+$ and $t_p < t_p^-$. Since $t_p^- < \bar{t} < t_q^+$ this would imply $t_p < t_q$ which is not possible (a message cannot be processed before it is queued);
- $t_q < t_q^-$ and $t_p > t_p^+$. Then $m$ contributes $+1$ to the left-hand-side of $\sum_a \bar{q}_a = \sum_a \bar{p}_a$ but 0 to the right-hande-side. Since this is the last remaining case, and that all other cases lead to $+1$ or $+0$ on each side of the equality, this case cannot happen if the equality is to hold.

We conclude that every message from $q$ to $p$ queued before $t_q^-$ has been processed before $t_p^-$, and hence $\{t_a^-\}_a$ is a completion time sequence. □

Figure 3.7 illustrates this lemma and the case of an AM "crossing" the $\{t_a^-\}_a$ and $\{t_a^+\}_a$ boundaries (the "envelope"). This lemma shows that any AM queued before the envelope has to be processed before. Since no tasks are running at any time within the envelope on any rank, the envelope represents a completion time. In particular, $\{t_a^-\}_a$ is a completion time sequence.

We now describe the algorithm. Rank 0 will be responsible to detect completion by synchronizing ($\bar{t}$) with other ranks $r > 0$. **When a rank is idle**, the main thread on all ranks does the following.




(a) Synchronization time $\bar{t}$, completion times $\{t_a^-\}_a$, and confirmation times $\{t_a^+\}_a$.

(b) The completion algorithm has to ensure that no message may cross the "envelope" created by $\{t_a^-\}_a$ and $\{t_a^+\}_a$ if it is to detect completion.

Figure 3.7: Completion algorithm (see Lemma 3.1).

1. All ranks $r$ continuously monitor $q_r(t)$ and $p_r(t)$ (which only contain the user's AM count and not the messages used in the completion algorithm). If at a time $t_r^-$ those values differ from the latest observed ones, rank $r$ sends a message `COUNT` $= (r, q_r(t_r^-), p_r(t_r^-))$ to rank 0 with those updated counts.

2. Rank 0 continuously observes the latest received counts. Since $q_r(\cdot)$ and $p_r(\cdot)$ are non-decreasing it is enough to consider the greatest received counts and discard the others. If at time $\tilde{t}$ (implemented as an always increasing integer counter), $\sum_r q_r(t_r^-) = \sum_r p_r(t_r^-)$ and that sum is different from the latest observed sum, rank 0 sends a `REQUEST` $= (q_r(t_r^-), p_r(t_r^-), \tilde{t})$ message back to all ranks $r > 0$.

3. All ranks $r$ continuously monitor the `REQUEST` messages from rank 0. They process the one with the largest $\tilde{t}$, and discard the others. At time $t_r^+$, if $q_r(t_r^-) = q_r(t_r^+)$ and $p_r(t_r^-) = p_r(t_r^+)$, they send a `CONFIRMATION` $= (\tilde{t})$ back to rank 0.

4. Rank 0 continuously observes the received `CONFIRMATION`. If all ranks replied with the latest $\tilde{t}$, the program has completed. Rank 0 then sends a `SHUTDOWN` message to all ranks.

5. All ranks $r$ continuously listen to the `SHUTDOWN` message. When received, the program has completed and rank $r$ terminates.

Note that although we write the algorithm as a sequence from 1 to 5, the word "continuously" indicates that this is implemented as a loop that keeps attempting to perform each step until `SHUTDOWN` is received.

We note that even when AM $m_1$ and $m_2$ are sent from rank $a$ to $b$ in that order, they may be processed in any order on rank $b$. This is because MPI messages are guaranteed to arrive in the order in which they are sent but they may complete at any time. The only guarantee is that when message $m_2$ is being processed, $m_1$ must be in the receive queue of rank $b$. So the algorithm needs to make sure to discard out-of-date values.

When all ranks have terminated, the worker threads are idle and there are no in-flight **user** messages. Because `SHUTDOWN` is sent last, all internal completion-related messages must be present in the queue on the receiver, and it is then safe to exit by repeatedly calling `progress()` on the communication thread on all ranks until all messages have been processed. Since those completion-related messages correspond to times $t < \bar{t}$, they can safely be discarded.

Figure 3.8 shows the completion algorithm in pseudo-code.

We finally state the following two properties, correctness and completion in finite time.

**Theorem 3.1** (Correctness)**.** *The `SHUTDOWN` message is sent if and only if completion has been reached.*

*Proof.* Assume first `SHUTDOWN` is sent as a response to a confirmation $\tilde{t}$. Denote $t_a^-$ the time at which `COUNT` is sent and $t_a^+$ the time at which `CONFIRMATION` is sent. Because of the intervening `REQUEST`, the synchronization time $\bar{t}$ exists. All assumptions in Lemma 3.1 are satisfied, hence the program has reached completion. Now assume we have reached completion at time $\{t_a^-\}_a$. Then all messages queued have been processed and all taskflows are idle. Hence all ranks $a > 0$ will send their latest $q_a(t_a^-)$ and $p_a(t_a^+)$ with a `COUNT` to rank 0 and those will be such that $\sum_a q_a(t_a^-) = \sum_a p_a(t_a^+)$. Rank 0 will reply with a `REQUEST` with time $\tilde{t}$, and since we have completed, the counts will no longer change and rank 0 will receive `CONFIRMATION` with matching times $\tilde{t}$. Rank 0 will then send `SHUTDOWN` messages and all ranks must eventually terminate. □

The second property guarantees that `CONFIRMATION` is sent in a finite time. For example, if the number of messages is potentially unbounded, messages from some ranks could always be prioritized, preventing any progress from other ranks, and the algorithm may never terminate.

**Theorem 3.2** (Finiteness)**.** *The completion protocol is guaranteed to send `CONFIRMATION` in a finite time.*

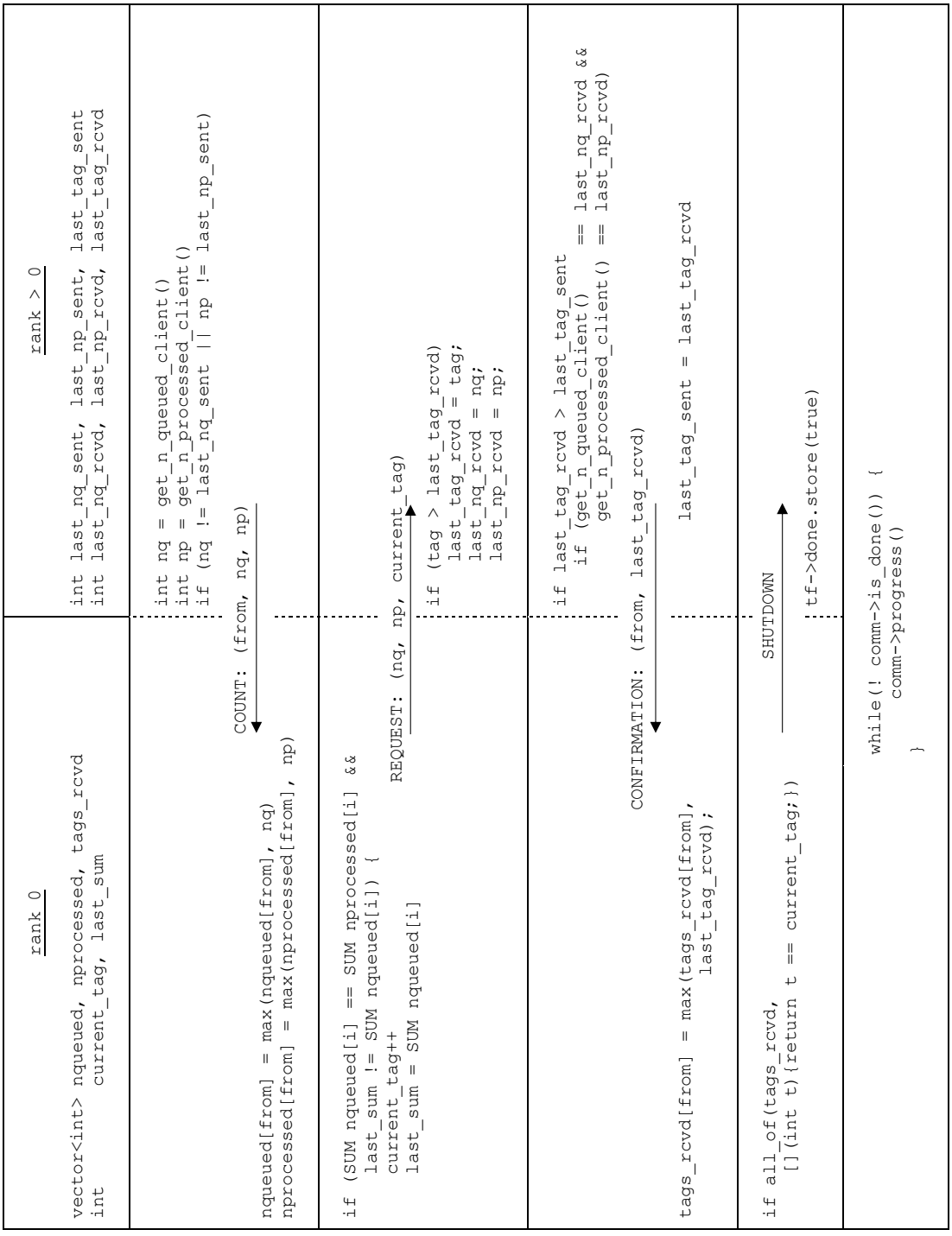



Figure 3.8: Completion algorithm

*Proof.* Denote $n$ the total number of **user messages** sent and received. That number is finite assuming the user code is correct, e.g., there is no hidden infinite loop. We prove that the completion algorithm requires sending no more than $O(n)$ messages. Let's consider each step:

- `COUNT`: since the message count can only increase, at most $2n$ messages may be sent (this is a pessimistic bound).
- `REQUEST`: since we only send a message when the count $\sum_a q_a(t_a^-)$ changes, we will broadcast at most $n$ messages.
- `CONFIRMATION`: the number of `CONFIRMATION` cannot exceed the number of `REQUEST`.
- `SHUTDOWN`: by construction, this message is sent exactly once.

Denote $n_p$ the number of processes. The number of messages used in the completion protocol is at most $2nn_p + 1$.

Assume that the user tasks as stalled because of missing data from other ranks. From our assumption, we are guaranteed that all messages will eventually arrive. This ensures that the user tasks will complete in finite-time. After that, the completion messages are also guaranteed to complete in finite-time. □

## 3.4 Benchmarks

In this section, we present benchmarks comparing `TTor` to OpenMP, StarPU, Regent, and ScaLAPACK.

We start with micro-benchmarks against OpenMP and StarPU to validate the low overhead of the shared memory component. This is only used to verify that the task-based management overhead is comparable, and sometimes better, than other runtime systems.

We then apply `TTor` (with its distributed component) to three classical linear algebra problems. In those sections, the goal is to compare a sequential enumeration of the DAG (STF) as implemented in StarPU and Regent versus the PTG approach as implemented in `TTor`. We note in particular that it is possible to modify the StarPU code such that the DAG is parallelized in a manner close to `TTor`. Similarly, several optimizations in `TTor` are possible but were not explored for this work (memory management, task insertion, communication).

Therefore, these benchmarks cannot be interpreted as measuring the peak performance of either runtime.

In particular, as we shall see, the results in Regent are very disappointing. Investigation shows that good results would only be possible with a custom mapper, which is something we did not pursue. As such, the Regent results should not be considered as representing Regent's peak performance.

In all cases, experiments are run on a cluster equipped with dual-sockets and 16 cores Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz with 32GB of RAM per node. Intel Compiler (`icpc (ICC) 19.1.0.166 20191121`) and Intel MPI are used with Intel MKL (version `2020.0.166`) for BLAS, LAPACK and ScaLAPACK. We use StarPU version `1.3.2` and Regent at commit `3418c32`. We assign one MPI rank per node. `TTor`'s code, including benchmarks, is available at `github.com/leopoldcambier/tasktorrent`. StarPU, Regent, and ScaLAPACK's benchmarks are available at `github.com/leopoldcambier/tasktorrent_paper_benchmarks`.

### 3.4.1 Micro-benchmarks

We first perform a series of micro-benchmarks to validate the low overhead of the shared memory component of the runtime. In the following, we average timings across 25 runs. In every case, the standard deviation was recorded as well, to estimate the variability of the measurement. In most cases, it was negligible and we don't report it. In all cases, we pick a number of tasks so that the total runtime is about 1 second.

#### No-dependencies overhead

We begin with an estimation of the "serial" overhead of `TTor`'s shared memory runtime. We start `ntasks` tasks, without any dependencies, and assign them in a round-robin fashion to the `nthreads` threads. Each task is only spinning for `spin_time` seconds. As such, the total ideal time is `spin_time` $\times$ `ntasks` / `nthreads`. Figure 3.9 shows the efficiency as a function of `nthreads` and `spin_time`. Given a total wall clock time of `run_time`, efficiency is defined as `run_time` $\times$ `nthreads` / (`spin_time` $\times$ `ntasks`). `ntasks` is chosen so that `run_time` is around 2 seconds.

Figure 3.9a shows results for `TTor`'s only, where we do **not** measure task insertion, i.e., we evaluate

```
for(int k = 0; k < n_tasks; k++) {
  tf.fulfill_promise(k);
```

```
}
tp.start(); // Start measuring time
tp.join();  // Stop measuring time
```

We see that the runtime has negligible impact for tasks $\approx 100\mu$s, and it becomes significant around 1 $\mu$s where overhead dominates.

We then compare it to OpenMP and StarPU in Figure 3.9b where, to make the comparison fair, insertion time **is** measured (which reduces the maximum possible efficiency, as the insertion is sequential).

```
tp.start(); // Start measuring time
for(int k = 0; k < n_tasks; k++) {
  tf.fulfill_promise(k);
}
tp.join();  // Stop measuring time
```

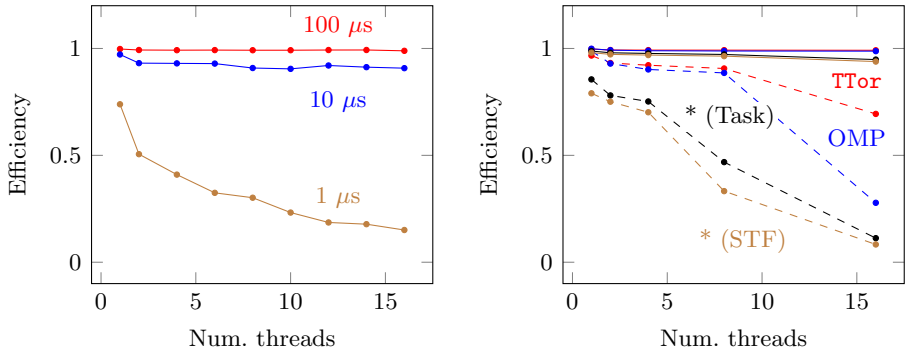
We note that this is a spurious consequence of creating tasks with no dependencies. In practice the insertion is done by other tasks, themselves executing in parallel. We evaluate StarPU both using "direct" task insertion ("Task"), as well as using the STF approach ("STF"). In the STF approach, each independent task is associated with an artificial independent read-write piece of data. We see that for very small tasks $< 10\mu s$, overhead is significant but comparable for all runtimes.

### Many dependencies overhead

We then estimate the overhead when dependencies are involved. Consider a 2D array of `nrows` $\times$ `ncols` tasks, with `ndeps` dependencies between task $(i, j)$ and $((i + k)\%\texttt{nrows}, j + 1)$ for $0 \leq k <$ `ndeps`. Again, tasks are spinning for `spin_time` seconds and, in `TTor`, task $(i, j)$ is assigned to thread $i\%$ `nthreads`.

Since this is not easily implementable in OpenMP, we only compare `TTor` with StarPU. In the "Task" version, tasks are directly inserted, and their dependencies are explicitly expressed. In the STF approach, we register data for every $(i, j)$ task and that data is used to create dependencies with the tasks in the next column. We note that StarPU STF has the constraint that the number of input data buffers for a given task should normally be known at compile-time, which makes it not well-suited for this benchmark.

Figure 3.10 shows the results with `nrows` set to 32. We see that `TTor` is between StarPU "Task" and StarPU "STF", with similar overhead. This validates the implementation.




(a) TTOR's overhead (no dependencies) measurement. Task insertion time is not included. Numbers indicate `spin_time`.

(b) Overhead (no dependencies) comparisons. Task insertion time is included. Solid is `spin_time` = 100 $\mu$s; dashed is 10 $\mu$s; * is StarPU with direct task insertion (Task) and STF semantics (STF).

Figure 3.9: Shared memory serial overhead, as a function of the number of threads `nthreads` (x-axis) and the task time `spin_time` (various lines). The plots show the mean across 25 runs.

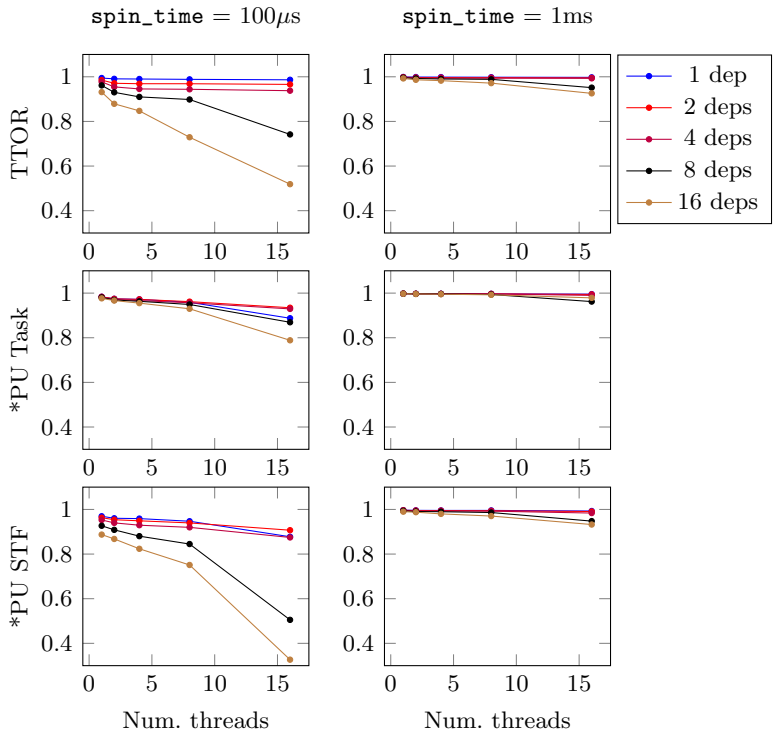



Figure 3.10: Efficiency vs. number of threads. Shared memory runtime dependency management overhead. The plots show the mean across 25 runs.

We conclude that the overhead of `TTor` is comparable (and sometimes better) to OpenMP and StarPU.

### 3.4.2 Distributed Matrix-matrix Product

We now consider a distributed matrix-matrix multiplication problem (GEMM), i.e., given $A, B \in \mathbb{R}^{N\times N}$ compute $C = AB$. We compare:

- `TTor` with an algorithm using a 2D block-cyclic mapping of blocks of size 256 to ranks, using the default ("small") and large AMs;
- `TTor` with an algorithm using a 3D mapping of blocks to ranks, tiled (every GEMM is single-threaded, with a block size of 256) or not (every GEMM is a single large multithreaded BLAS). We use the DNS algorithm (see Algorithm 3.1 and also for instance [76]) to map blocks to ranks.
- StarPU (with STF semantics, i.e., all ranks explore the full DAG) using a 2D block-cyclic mapping of blocks of size 256 to ranks. Various scheduling strategies have been tried, without significant variation in runtime; the default local work-stealing `lws` is then used.
- Regent, with a block size of 512 (which gives better results than 256), but *without* any custom mapper.
- ScaLAPACK using a 2D block-cyclic mapping (with a block size of 256) with multithreaded BLAS. We note that ScaLAPACK is not a runtime and is not actively managing a task graph.

The following code snippet shows the GEMM portion when using the 2D block-cyclic data distribution. In this case, contributions $A_{ik}B_{kj}$ are ordered with $k$, i.e., $A_{ik}B_{kj}$ happens before $A_{i(k+1)}B_{(k+1)j}$. Furthermore, because of the 2D data distribution, the products $A_{ik}B_{kj}$ are mapped to a rank function of $(i, j)$ only and, as such, always happen on the same node. The mapping of tasks to threads may be any deterministic function of `ikj`. In practice, something as simple as `ikj[0] % n_threads` can be used without any visible performance degradation. It is merely used to distribute task dependency management evenly across threads. `noalias()` is from the linear algebra library Eigen.

**Algorithm 3.1** The DNS 3D Gemm algorithm. We divide the processors into a 3D grid and refer to them as triplets $(i, j, k)$. The matrices $A$, $B$ and $C$ are divided into blocks of the same sizes.

**procedure** DNS($A, B$) ▷ Rank $(i, j, 0)$ owns $A_{ij}$ and $B_{ij}$
  Rank $(i, j, 0)$ sends $A_{ij}$ to rank $(i, j, j)$
  Rank $(i, j, 0)$ sends $A_{ij}$ to rank $(i, j, i)$
  Rank $(i, j, j)$ broadcasts $A_{ij}$ to rank $(i, *, j)$
  Rank $(i, j, i)$ broadcasts $B_{ij}$ to rank $(*, i, j)$
  Rank $(i, j, k)$ computes $\tilde{C}_{ijk} = A_{ik}B_{kj}$, sends to rank $(i, j, 0)$
  Rank $(i, j, 0)$ accumualtes all $\tilde{C}_{ijk}$ into $C_{ij}$
  **return** $C_{ij}$ ▷ Rank $(i, j, 0)$ owns $C_{ij}$
**end procedure**

```
gemm_Cikj.set_task([&](int3 ikj){
    int i = ikj[0];
    int k = ikj[1];
    int j = ikj[2];
    C_ij[i + j * num_blocks].noalias() +=
        A_ij[i + k * num_blocks] *
        B_ij[k + j * num_blocks];
    if(k < num_blocks-1) {
        gemm_Cikj.fulfill_promise({i,k+1,j});
    }
}).set_indegree([&](int3 ikj) {
    return (ikj[1] == 0 ? 2 : 3);
}).set_mapping([&](int3 ikj) {
    return (ikj[0] / nprows + ikj[2] / npcols
        * (num_blocks / nprows)) % n_threads;
});
```

Figure 3.11 presents strong and weak scalings results. Scalings are done multiplying the number of rows and columns by 2 and/or the number of nodes by 8, and the largest test cases are matrices of size 32 768. We make multiple observations:

- `TTor` benefits from the large messages (Figure 3.11c) over small ones, decreasing the total time by up to 30%.
- `TTor` with large messages and StarPU using the 2D mapping have similar performance (Figure 3.11c vs Figure 3.11e). `TTor` performs better than StarPU with small blocks (Figure 3.11h).
- `TTor` with the 3D mapping and the tiled algorithm has better performance than without (see Figure 3.11d as well as Figure 3.11a vs Figure 3.11b for results on 8 nodes). This shows the importance of having a small task granularity, to increase the overlap

between communications and computations. It has however similar performance to the 2D mapping.

- Regent (Figure 3.11f) does not exhibit good performances. Profiling and discussion with the library authors indicate that the reason is the lack of a custom mapper, which is something complex to write as a user and that we did not pursue.
- Runtime-based (`TTor` and StarPU) implementations outperform ScaLAPACK (Figure 3.11g), showing the benefits of a task-based runtime system.

Figure 3.11h shows the impact of the block size on the runtime. We see that `TTor` is about 2.5x faster than StarPU at small sizes. This highlights the advantages of a distributed DAG exploration. We note that in this case, small blocks are not optimal. However, GEMM is in some sense an "easy" benchmark since it offers a large amount of concurrency. Therefore, to stress the runtimes and observe measurable differences we need to deviate from the optimal GEMM settings. Although we could not investigate other algorithms for this chapter, more complex applications would probably reveal additional differences between `TTor` and StarPU. This is particularly important in less regular computations where the task granularity is harder to control (for instance in more complex scientific codes) or varies significantly.

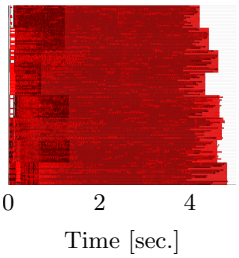
Finally, Figure 3.11i shows the efficiency of `TTor` (2D GEMM) as a function of the concurrency. Since the GEMMs are sequential as a function of $k$, `num_blocks^2/n_cores` indicates how much parallelism is available per core. This represents the number of blocks that are processed on each core between communication steps. We see that efficiency decreases sharply at around 16 blocks per core.

### 3.4.3 Distributed dense Cholesky factorization

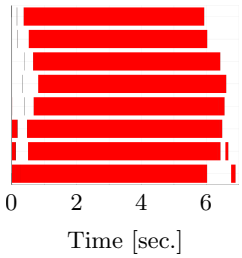
We now consider an implementation of the Cholesky algorithm, i.e., given a symmetric positive definite matrix $A \in \mathbb{R}^{N\times N}$, compute $L$ such that $A = LL^\top$. In its sequential and blocked form, the algorithm is described in Algorithm 3.2.

The algorithm is made of three main computational routines: potrf$(k)$, trsm$(i,k)$ and gemm$(k,i,j)$ (in practice syrk when $i = j$). We show a PTG formulation of Algorithm 3.2 in Figure 3.12. Large active messages are used.

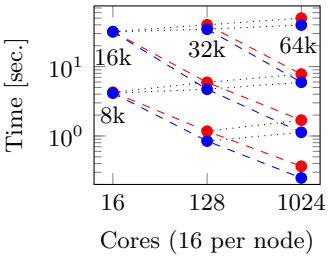
We compare `TTor`, StarPU (with STF semantics), Regent, and ScaLAPACK. A 2D block-cyclic data distribution is used with a block size of 256, except in Regent where the



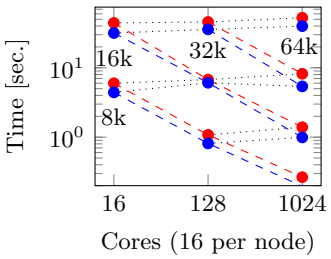

(a) Tiled 3D DNS GEMM trace for 8 nodes. Every red line is 1 `TTor` thread using single-threaded BLAS.



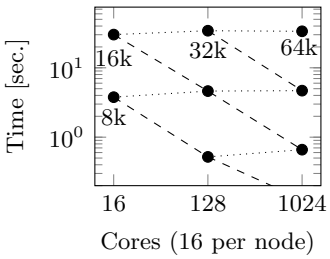

(b) Non-tiled 3D DNS GEMM trace for 8 nodes. Every red line is 1 `TTor` thread using multithreaded BLAS.



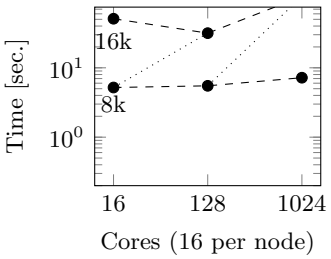

(c) `TTor` 2D GEMM. Red = small AMs, blue = large AMs.



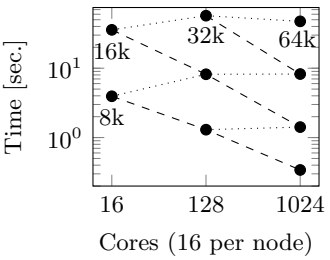

(d) `TTor` 3D GEMM. Red = non-tiled, blue = tiled.



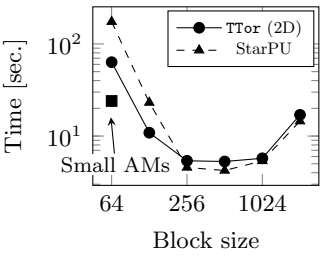

(e) StarPU 2D GEMM.



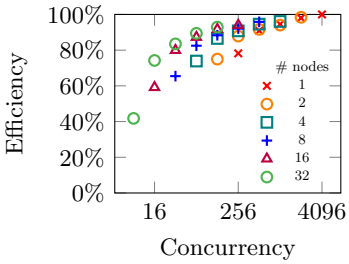

(f) Regent 2D GEMM (block size 512, default mapper).




(g) ScaLAPACK GEMM.




(h) Block size impact, $N = 32\,768$ with 1024 CPUs.

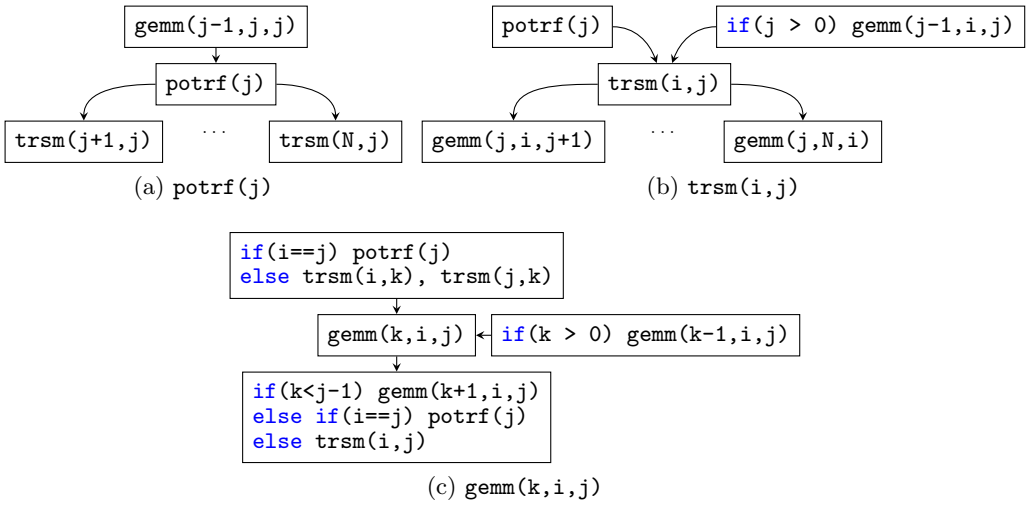



(i) `TTor` 2D GEMM. Concurrency = `num_blocks^2/n_cores`, $N = 16\,384$.

Figure 3.11: GEMM scalings. (a-b): impact of task granularity on 3D GEMM. Smaller tasks give a higher overlap of computation and communication. (c-g): weak (dotted) and strong (dashed) scalings. Numbers indicate the matrix size $N$. Largest test case is $N = 65\,536$. (h): optimal block size (i.e., task granularity) for the $N = 32\,768$ test case. The extra data point shows the improvement when using small AMs instead of large AMs on small block sizes. The decrease in the number of messages sent improves the runtime by 3x. (i): efficiency as a function of concurrency for $N = 16\,384$. Reference timing is with 1 core. Figure and data from Yizhou Qian.

**Algorithm 3.2** Block Cholesky algorithm

1: **procedure** CHOLESKY($A$, $n$) ▷ $A \succ 0$, $n \times n$ blocks
2: **for** $1 \leq k \leq n$ **do**
3: $L_{kk}L_{kk}^{\top} = A_{kk}$ ▷ potrf$(k)$
4: **for** $k+1 \leq i \leq n$ **do**
5: $L_{ik} = A_{ik}L_{kk}^{-\top}$ ▷ trsm$(i,k)$
6: **for** $k+1 \leq j \leq i$ **do**
7: $A_{ij} \leftarrow A_{ij} - L_{ik}L_{jk}^{\top}$ ▷ gemm$(k,i,j)$
8: **end for**
9: **end for**
10: **end for**
11: **end procedure**




(a) potrf(j) (b) trsm(i,j)

(c) gemm(k,i,j)

Figure 3.12: PTG description of Algorithm 3.2. In TTor, when out-dependencies are remote, an AM is sent to the remote rank, carrying the associated block and triggering remote tasks.

block size is 512 and no custom mapper is used. Task priorities in `TTor` are computed using [14]. As before, in ScaLAPACK the block size is related to the data distribution but there are no tasks per se.

Weak and strong scalings are performed by multiplying the number of rows and columns by 2 or the number of cores by 8. The larger test case is a matrix of size $N = 131\,072$. Figure 3.13 shows the results.

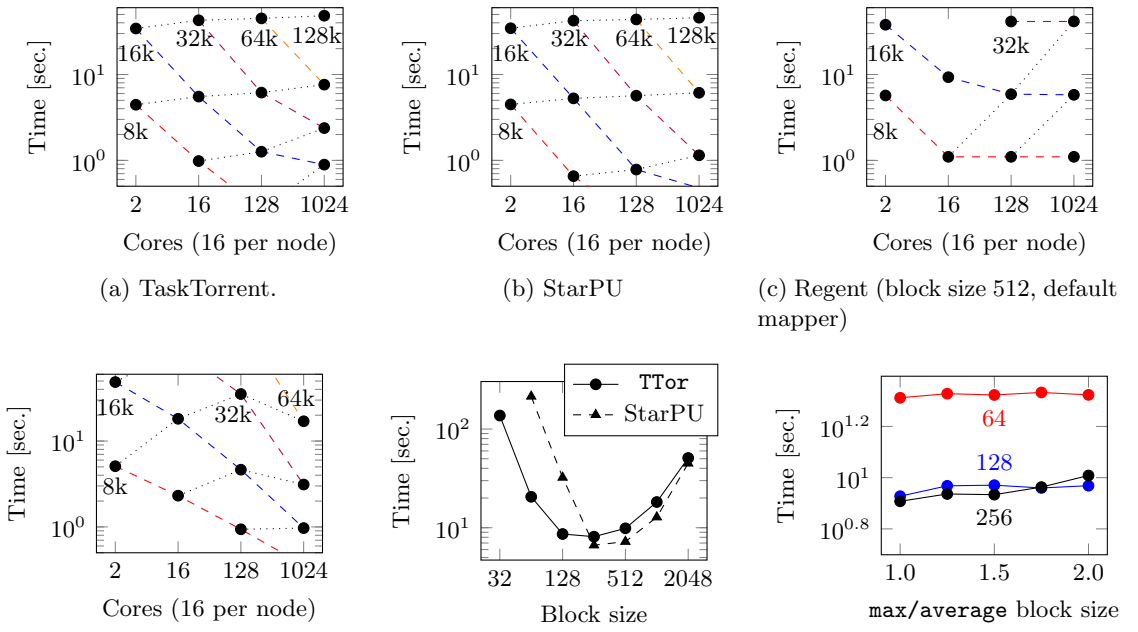
We see that on large problems, both `TTor` and StarPU reach very similar performances, both outperforming ScaLAPACK by far: for $N = 131\,072$ on 1024 cores, ScaLAPACK takes more than 125 secs (not shown). On the $N = 131\,072$ test case, `TTor` and StarPU differ by less than 10%. StarPU shows better strong scaling for small problems on many nodes. We conjecture that this may be due to a better task scheduler, memory management (thread-memory affinity), and mapping of the computation across nodes. Regent, on the other hand, while scaling on 1 node, does not reach good performances beyond that. We conjecture that the lack of a custom mapper prevents proper scaling, as the matrix is not well distributed across ranks.

Figure 3.13e shows the runtime as a function of the block size for a test case of size $65\,536 \times 65\,536$ on 64 nodes (1024 CPUs). We see that 256 gives the best results for both `TTor` and StarPU. Furthermore, we observe that for small task size, `TTor` degrades less quickly than StarPU. The small block size leads to many tasks and unrolling the DAG on one node becomes prohibitive, even for reasonably large tasks (block size of 128). For a block size of 64, `TTor` is about 10x faster. Thanks to its lightweight runtime and distributed DAG exploration, `TTor` suffers less from the small task size. For large task sizes, both degrade similarly. The poor performance at large sizes is caused by a lack of concurrency.

Figure 3.13f shows a load balancing test using random block sizes with a fixed number of blocks. $\rho$ is the ratio of the largest over the average block size. For $\rho = 1.5$, the ratio of flops from smallest to largest task is $(1.5/0.5)^3 = 27$. We see that `TTor` handles tasks of various granularity very well, with less than 25% degradation from $\rho = 1$ to $\rho = 2$ for an average block size of 256.

### 3.4.4 Sparse Cholesky

As a final application, we consider a sparse Cholesky algorithm. We use Scotch [46]'s algebraic nested dissection ordering and a 1D mapping of columns to ranks. We perform a strong scaling only, with an *average* block size of 256 (Scotch leads to some blocks having




(a) TaskTorrent.

(b) StarPU

(c) Regent (block size 512, default mapper)

(d) ScaLAPACK

(e) Block size impact, $N = 65\,536$ with 1024 CPUs.

(f) Load balancing test with random block sizes.

Figure 3.13: Cholesky scalings. (a-d): weak (dotted) and strong (dashed) scalings. Numbers indicate the matrix size $N$. Largest (top right) test case is $N = 131\,072$. (e): optimal block size (i.e., task granularity) for the $N = 65\,536$ test case. (f): load balancing test using random block sizes for the $N = 65\,536$ test case with 1024 CPUs. Block sizes are random uniform on $((2-\rho)b, \rho b)$ with $b$ the maximum block size and $\rho =$ `max_block_size/average_block_size`. Numbers indicate the average block size.

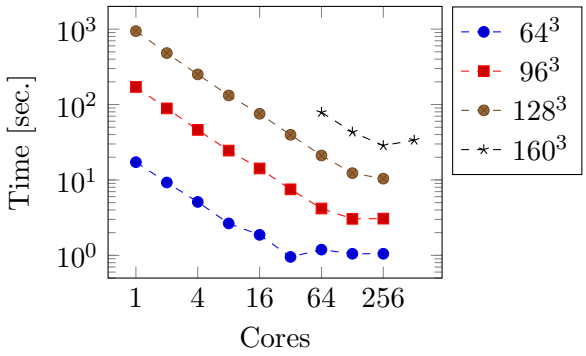



Figure 3.14: Sparse Cholesky. Weak and strong scalings over a 7-points stencil matrix. The legend indicates the matrix size.

larger and smaller sizes). The matrix corresponds to a 7-points stencil over a 3D cube. Figure 3.14 presents the factorization time as a function of the total number of cores, for various matrix sizes, indicated as $N = n^3$. We see that for large enough sizes, `TTor` strong scales very well. For instance, for $N = 128^3 \approx 2M$, the total factorization time goes from about 1000 seconds down to about 10 seconds.

At the moment, the code performs the ordering on a single node and runs out of memory on large problems. We note that an algorithm such as sparse Cholesky is very sensible to the block size, the ordering, the clustering within separators, and other factors. Those questions are however outside the scope of this chapter and, as such, we do not compare `TTor` with other sparse direct solvers.

## 3.5 Conclusion

We presented TaskTorrent (`TTor`), a lightweight distributed task-based runtime system in C++. It has a friendly API and relies on readily available tools (C++14 and MPI). It enables shared-memory task-based parallelism coupled with one-sided active messages. Those two concepts naturally work together to create a distributed task-based parallel computing framework. We showed that `TTor` is competitive with both StarPU (a state of the art runtime) and ScaLAPACK on large problems. Its lightweight nature allows it to be more forgiving when task granularity is not optimal, which is key to integrating this approach in legacy codes.

# Chapter 4

# Parallel Sparsified Nested Dissection

## 4.1 Introduction

In this work, we consider the parallelization of the sparsified nested dissection (spaND) algorithm. While spaND is a fast algorithm, parallelization is essential to run large problems. In the following, we describe a *distributed memory* and *task-based* parallel version of spaND.

### 4.1.1 Previous work

Task-based algorithms have already been used in linear algebra. Since the years 2010, most efforts have been focused on dense linear algebra algorithms (typically Cholesky, LU, and QR factorizations). The PLASMA (for multicore CPUs) and MAGMA (for hybrid CPU and GPU machines) projects [140, 3, 148] implement tiled task-based algorithms using a dynamic scheduler. DPLASMA [27] extends PLASMA to distributed memory machines and uses the PaRSEC runtime (closely related to the DAGuE compiler) [26]. More recently, the related SLATE project [69] aims at replacing ScaLAPACK's dense linear algebra algorithms with task-based algorithms. In [97] the authors introduced FLAME and the SuperMatrix data structure, to efficiently map dense linear algebra operations on heterogeneous systems.

Task-based parallelism has also been used in sparse direct solvers. In [102], the authors point out that DAG scheduling of sparse algorithms is challenging because of the large number of small and irregular tasks. Tasks are first created, by traversing the elimination

tree and using block algorithms. The algorithm only then executes tasks asynchronously using the previously computed dependencies. In [103] the authors integrated StarPU [11] and PaRSEC [26] in the PaStiX [90] sparse direct solver. Using a generic task-based runtime system led to similar performance compared to the original specialized scheduler used in PaStiX, with the additional benefit of leveraging accelerators such as GPUs. The authors in [1, 2] study the use of a hybrid (CPU+GPU) STF task-based runtime system for a sparse QR algorithm, where task granularity has to be large enough to saturate the GPU, but small enough to exhibit enough parallelism. In [5] the authors took an MPI+task approach. Instead of expressing the entire DAG with PaRSEC or StarPU, only the local DAG is provided to a runtime system. This is well-suited to hybrid CPU+GPU machines, where each subdomain (owned by one MPI rank) is assigned to a heterogeneous machine. Each domain can then be scheduled on the multicore CPU, possibly extended by accelerators such as GPUs.

Regarding spaND, to the best of our knowledge, the only distributed memory implementation of a spaND-like algorithm can be found in [106] where the authors consider a distributed HIF [95]. That work resembles ours. HIF is similar to spaND, but is restricted to 7-points stencils on regular 2D or 3D meshes, and use interpolative factorization. We consider a more general setting (spaND works on any sparse matrix), use a task-based approach, and we use a different algorithm using orthogonal transformations.

However, unlike in [106], we do not use distributed RRQR's to parallelize the large low-rank factorizations arising at the top of the tree. spaND is an $\mathcal{O}(N \log N)$ algorithm when used over matrices coming from the discretization of 3D PDEs. This indicates that every level in the algorithm has a similar $\mathcal{O}(N)$ complexity. However, while the leaves exhibit a lot of concurrency, the top levels only have a few large interfaces. In our approach, we associate each task with an interface. As such, concurrency is limited when the sparsified interfaces are large, as it is typically the case on large problems coming from 3D PDEs. As such, we will focus on applications coming from 2D PDEs when the matrix size is very large. In this case, the top separator size grows slowly with $N$ and the algorithm is expected to have an $\mathcal{O}(N)$ complexity. Hence, a task-based approach should scale well. This is what we indeed observe.

### 4.1.2 Contributions

In this work, we build a task-based algorithm for a fast sparse linear solver, spaND. To the best of our knowledge, this is the first task-based approach to a fast sparse solver. We perform the following.

- We describe a parallel ordering and clustering algorithm for spaND, that only relies on the graph of $A$ (Section 4.2.1);
- We design a sparsification method such that all interfaces can be sparsified simultaneously, and rigorously prove that it has the same error as the original method in Chapter 2 (Section 4.2.2);
- We describe the resulting DAG, and how to express it using `TTor` (Section 4.2.3);
- We run the resulting algorithm on large problems, with matrix sizes up to $300M$ and using more than 9000 cores (Section 4.3), and show that it performs well compared to other state-of-the-art fast solvers such as Hypre's Boomer AMG.
- We discuss the use of `TTor` for such an algorithm, its benefits, and its limitations.

## 4.2 Task-based parallel spaND

### 4.2.1 Parallel partitioning and ordering

We now describe the algorithm used to define separators, interfaces (a clustering of the vertices within separators), and to distribute the matrix across MPI ranks[1]. In the following, the notation // denotes the integer division (i.e., $i//j = \text{floor}(i/j)$).

The algorithm is based on a recursive bisection (a partitioning) of the graph of $A$ and proceeds in three steps. Let $L > 0$ be the number of desired levels and $P = 2^k$ ($k \geq 0$) the number of MPI ranks. In the following, we say that a set $1, \dots, N$ is distributed based on a map $i \to p_i$ with $0 \leq p_i < \rho$ over $P$ ranks if rank $0 \leq p < P$ owns $i$ such that $p_i \in [pK, (p+1)K - 1]$ with $K = (\rho + P - 1)//P$.

Given $L > 0$, we consider a binary tree of separators as shown in Figure 4.1a. We denote a separator by $(\ell, s)$ where $0 \leq \ell < L$ is the level (with 0 for the leaves and $L - 1$ for the top) and $0 \leq s < 2^{L-\ell-1}$.

[1] In this work, we use the term rank for the ranks resulting from the low-rank approximations, and MPI rank for a processor's position within a MPI communicator.

For $s, t$ two separators, $\text{top}(s, t)$ is defined as the separator $r$ on the path $s \to t$ the closest to the root $(L-1, 0)$. If $t = \texttt{none}$, then $\text{top}(s, t)$ returns $s$. For a separator $t = (\ell, s)$ we also define $\text{level}(t) = \ell$. Finally, let $N_i = \{j | A_{ij} \neq 0, j \neq i\}$ be the neighborhood of vertex $i$ in $A$.

The algorithm to build separators, interfaces, and to partition the matrix is the following.

- A recursive bisection of $A$ is computed, assigning to every vertex $i$ a partition $p_i$, $0 \leq p_i < 2^{L-1}$. The dissection is recursive, meaning the map $i \to p_i//2^\ell$ defines the partitioning at level $\ell$. This can be done entirely algebraically or using geometrical information. In practice, if geometry information is available, it is preferable to use it. The matrix is then distributed based on this $i \to p_i$ partitioning. Edges in the matrix are distributed using a 1D (block-) column partitioning: $A_{ij}$ is mapped to the MPI rank of vertex $j$.

- Separators are computed, assigning to each vertex $i$ a 2-tuple $(\ell, s)$ with $\ell$ the level in the ND tree and $s$ the separator. Informally, vertices on the left of a partition but adjacent to the right partition will form a separator. The algorithm proceeds level by level, computing first the level $L-1$ separator (the top). It then communicates that information (using a halo exchange) and then proceeds with level $L-2$, etc., until level 0. This is necessary since higher-level separator information is required to compute lower level separators.

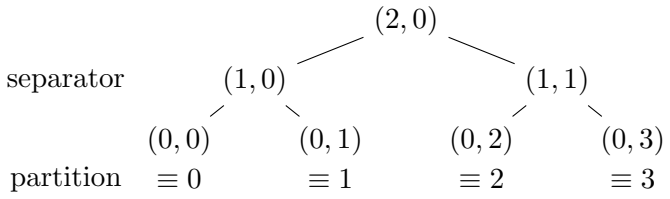
- Interfaces are computed by assigning to each vertex $i$ a tuple $(s_l, s_r)$ where $s_l$ is the highest separator on its left and $s_r$ is the highest separator on its right. The highest separator on the left is defined by repeatedly applying top between $s_l$ (initialized to $(0, p_i)$) and all the left neighbors of $i$. Similarly, the highest separator on the right is defined by repeatedly applying top between $s_r$ (initialized to $\texttt{none}$) and all the right neighbors of $i$ (which, by construction, form a non-empty set). This step can be done in one pass over the data as it only depends on previously computed separator information.

The complete algorithm is available in Algorithm 4.1 and illustrated on Figure 4.1. Each MPI rank only holds the piece of $s$ and $c$ carrying local and adjacent (halo) nodes information.

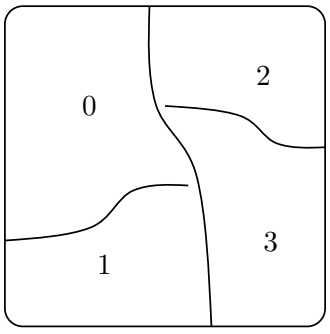
The output of Algorithm 4.1 is a mapping from vertices to a 5-tuples of integers and

**Algorithm 4.1** Partitioning algorithm. "Halo update" means communicating updated $p, s, c$ to neighboring MPI ranks to update their halo values, and vice-versa.

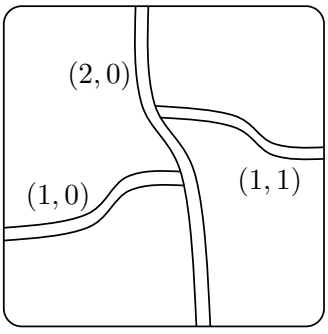
**Require:** $A$ symmetric, distributed, $L > 0$, $P > 2^{L-1}$ a power of 2
**Ensure:** $0 \leq p_i < 2^{L-1}$
**Ensure:** $s_i = (\ell, s)$ with $0 \leq \ell < L$, $0 \leq s \leq 2^{L-\ell-1}$
**Ensure:** $c_i = (s_l, s_r)$ are separators on the left and right of $i$
  $p \leftarrow RB(A, 2^{L-1})$ ▷ Recursively partition $A$ in $2^{L-1}$ pieces
  Distribute $A$, send vertex $i$ to MPI rank $p_i/(2^{L-1}/P)$.
  **for** $i$ local **do**
    $s_i \leftarrow (0, p_i)$
  **end for**
  Halo_update$(A, p, s, c)$
  **for** $\ell = L-1, \ldots, 1$ **do** ▷ Create top to bottom separators
    **for** $i$ local **do**
      $p'_i \leftarrow p_i // 2^\ell$ ▷ Partitioning at level $\ell$
      **if** $p'_i \% 2 = 0$ **then** ▷ If on the left...
        **for** $j \in N_i$ **do**
          $p'_j \leftarrow p_j // 2^\ell$ ▷ Partitioning at level $\ell$
          **if** $p'_j = p'_i + 1$ **then** ▷ But touches the right...
            $s_i \leftarrow (l, p'_i // 2)$ ▷ Set separator
            Break
          **end if**
        **end for**
      **end if**
    **end for**
    Halo_update$(A, p, s, c)$
  **end for**
  **for** $i$ local **do** ▷ Create interfaces
    **if** level$(s_i) > 0$ **then** ▷ If not a leaf
      $s_l, s_r, p'_i \leftarrow (0, p_i), \texttt{none}, p_i // 2^{\text{level}(s_i)}$
      **for** $j \in N_j$ **do**
        **if** level$(s_j) <$ level$(s_i)$ **then**
          $p'_j \leftarrow p_j // 2^{\text{level}(s_j)}$
          **if** $p'_j < p'_i$ **then**
            $s_l \leftarrow \text{top}(s_j, s_l)$
          **else**
            $s_r \leftarrow \text{top}(s_j, s_r)$
          **end if**
        **end if**
      **end for**
      $c_i \leftarrow (s_l, s_r)$
    **else**
      $c_i \leftarrow ((0, p_i), (0, p_i))$
    **end if**
  **end for**
  Halo_update(A,p,s,c)
  **return** $p$ the partitioning, $s$ the ordering, $c$ the interfaces clustering



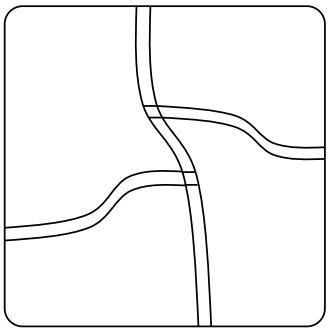

(a) Nested Dissection ordering binary tree for $L = 3$




(b) Recursive bisection. The matrix is distributed across MPI ranks using a 1D column partitioning based on the RB.




(c) Separators creation. Separators are vertices in the left partition adjacent to the right partition.



(d) Interfaces definition. Interfaces are built by clustering vertices within separators based on the $(p, s_l, s_r)$ tuple where $p$ is the partition, $s_l$ is the left separator and $s_r$ is the right separator.

Figure 4.1: Partitioning algorithm

separators

$$\text{map} : i \to \text{map}(i) = (p_i, s, s_l, s_r, 0)$$

defining interfaces at level 0. $p_i$ is the partition of vertex $i$ at level 0, $s$ its ND separator, $s_l$ and $s_r$ its left and right ND separators neighbors, and 0 indicate this is the partitioning at the very first level. Vertices $i$, $j$ such that $\text{map}(i) = \text{map}(j)$ are then clustered together and form an interface. Note that interfaces, by construction, belong to the same partition and, as such, to the same MPI rank.

From level $\ell$ to level $\ell+1$, merging is done by following the RB and ND trees up towards the root. Formally, a non-eliminated interface $\phi = (p_i, s, s_l, s_r, \ell)$ (i.e., $s$ is not a leaf in the ND tree at level $\ell$) is transformed into its parent interface $\psi = \text{parent}(\phi)$ using this map

$$\phi \to \text{parent}(\phi) : (p_i, s, s_l, s_r, \ell) \to (p_i//2, s, \text{up}(s_l, \ell), \text{up}(s_r, \ell), \ell + 1)$$

where

$$\text{up}(s, \ell) = \begin{cases} \text{parent}(s) & \text{if } s \text{ is an ND leaf at level } \ell \\ s & \text{otherwise} \end{cases}$$

and where $\text{parent}(s)$ is the parent of $s$ in the ND binary tree. Different interfaces $\phi, \phi'$ with the same parent $\psi$ are then merged together to form an interface at the next level. Interfaces are then distributed across MPI ranks based on their partition.

Notice that

- By construction, an interface is always entirely contained in a partition and, as such, is resident on a given MPI rank.
- In the above formula, the ND separator $s$ never changes.

We re-emphasize that, in this algorithm, the load balancing (i.e., which interface is mapped to which MPI rank) is purely based on the recursive bisection of $A$. In particular, this assumes that interface ranks are uniform across the domain. From experience, this is usually the case on elliptic PDEs. However, if this is not the case, this load balancing strategy may be far from the optimum.

### 4.2.2 Parallel sparsification

Sparsification is the central part of spaND. In this section, we describe a parallel version of sparsification. Let us consider a particular level, and assume that eliminations and block

scalings have all been performed. What is left to do is to sparsify all interfaces.

Let us denote the remaining interfaces by $s_1, s_2, \ldots, s_k$. We also denote by $n_1, \ldots, n_k$ their respective complement (i.e., if $I$, $|I| = n$, denote all the remaining dofs, $n_i \cup s_i = I$, $n_i \cap s_i = \emptyset$). At a given step of the algorithm, the trailing matrix is

$$A = \begin{bmatrix} A_{s_1 s_1} & A_{s_1 s_2} & \ldots & A_{s_1 s_k} \\ A_{s_1 s_2} & A_{s_2 s_2} & \ldots & A_{s_2 s_k} \\ \vdots & \vdots & \ddots & \vdots \\ A_{s_k s_1} & A_{s_k s_2} & \ldots & A_{s_k s_k} \end{bmatrix} \in \mathbb{R}^{n \times n}.$$

In practice note that most blocks in this matrix are effectively zero and $A_{s_i s_i} = I$. However, those do not affect the following analysis.

**Regular sparsification algorithm** Now consider sparsification following the order $s_1 \to s_2 \to \cdots \to s_k$ (see Algorithm 4.2). Starting from $A_0 := A$, we consider sparsifying $s_1$. This means computing a rank-revealing factorization such that

$$\begin{bmatrix} A_{0,s_1 n_1} & A_{0,n_1 s_1}^\top \end{bmatrix} = Q_1 \begin{bmatrix} W_{s_1 n_1} & W_{n_1 s_1}^\top \end{bmatrix} = \begin{bmatrix} Q_1^f & Q_1^c \end{bmatrix} \begin{bmatrix} W_{s_1 n_1}^f & W_{n_1 s_1}^{f,\top} \\ W_{s_1 n_1}^c & W_{n_1 s_1}^{c,\top} \end{bmatrix}$$

with $\left\| Q_i^{f,\top} \begin{bmatrix} A_{0,s_i n_i} & A_{0,n_i s_i}^\top \end{bmatrix} \right\|_F = \left\| \begin{bmatrix} W_{s_1 n_1}^f & W_{n_1 s_1}^f \end{bmatrix} \right\|_F \leq \varepsilon$. Note the Frobenius norm, as this simplifies some derivations. Similar results, with different constants, can be obtained in the 2-norm. Rows $W_{s_1 n_1}^f$ and columns $W_{n_1 s_1}^f$ are then set to zero, leading to

$$A_1 := \begin{bmatrix} Q_1^{f,\top} A_{s_1 s_1} Q_1^f & Q_1^{f,\top} A_{s_1 s_1} Q_1^c & & & \\ Q_1^{c,\top} A_{s_1 s_1} Q_1^f & Q_1^{c,\top} A_{s_1 s_1} Q_1^c & W_{s_1 s_2}^c & \ldots & W_{s_2 s_k}^c \\ & W_{s_2 s_1}^c & A_{0,s_2 s_2} & & A_{0,s_2 s_k} \\ & \vdots & & \ddots & \\ & W_{s_k s_1}^c & A_{0,s_k s_2} & & A_{0,s_k s_k} \end{bmatrix}.$$

With $U_1 = \begin{bmatrix} Q_1 & \\ & I \end{bmatrix}$ orthogonal we have $\|U_1^\top A_0 U_1 - A_1\|_F = \left\| \begin{bmatrix} W_{s_1 n_1}^f & W_{n_1 s_1}^{f,\top} \end{bmatrix} \right\|_F \leq \varepsilon$. The algorithm then proceeds with $s_2$, ..., $s_k$. At every step, we build $Q_i = \begin{bmatrix} Q_i^f & Q_i^c \end{bmatrix}$ such that $\left\| Q_i^{f,\top} \begin{bmatrix} A_{i-1,s_i n_i} & A_{i-1,n_i s_i}^\top \end{bmatrix} \right\|_F \leq \varepsilon$. Defining $U_i$ as a $k-$ diagonal block matrix with ones on

the diagonal, except for its $i^{\text{th}}$ diagonal block equal to $Q_i$, we then have $\|U_i^\top A_{i-1}U_i - A_i\|_F \leq \varepsilon$. Now assume that $\|U_i^\top \cdots U_1^\top A_0 U_1 \cdots U_i - A_i\|_F \leq i\varepsilon$ and $\|U_{i+1}^\top A_i U_{i+1} - A_{i+1}\|_F \leq \varepsilon$. We then have

$$\begin{aligned}
&\|U_{i+1}^\top \cdots U_1^\top A_0 U_1 \cdots U_{i+1} - A_{i+1}\|_F \\
&\quad = \|U_{i+1}^\top (U_i^\top \cdots U_1^\top A_0 U_1 \cdots U_i - A_i + A_i) U_{i+1} - A_{i+1}\|_F \\
&\quad \leq \|U_{i+1}^\top (U_i^\top \cdots U_1^\top A_0 U_1 \cdots U_i - A_i) U_{i+1}\|_F + \|U_{i+1}^\top A_i U_{i+1} - A_{i+1}\|_F \\
&\quad = \|U_i^\top \cdots U_1^\top A_0 U_1 \cdots U_i - A_i\|_F + \|U_{i+1}^\top A_i U_{i+1} - A_{i+1}\|_F \\
&\quad \leq (i+1)\varepsilon
\end{aligned}$$

using the invariance of the Frobenius norm to square orthogonal transformations and the triangle inequality. Let $\widetilde{A} := A_k$ be the final sparsified matrix. We conclude that

$$\|U_k^\top \cdots U_1^\top A U_1 \cdots U_k - \widetilde{A}\|_F \leq k\varepsilon.$$

**Algorithm 4.2** Sequential sparsification algorithm. $Q_i$ is computed based on $A_{i-1}$ and drop_fine drops entries corresponding to fine degrees of freedom

$A_0 \leftarrow A$
**for** $i = 1, \ldots, k$ **do**
  Sparsify $s_i$, i.e., compute $Q_i$ such that

$$\|Q_i^{f,\top} \begin{bmatrix} A_{i-1,s_i n_i} & A_{i-1,n_i s_i}^\top \end{bmatrix}\|_F \leq \varepsilon$$

  $A_i \leftarrow \text{drop\_fine}(U_i^\top A_{i-1} U_i)$
**end for**

**Simultaneous sparsification algorithm** We now consider an alternative algorithm (see Algorithm 4.3). Instead of sparsifying $s_1 \to s_2 \to \cdots \to s_k$, we sparsify all $s_i$ $(1 \leq i \leq k)$ at the same time, given the original $A$. This means that for all $i$, we *simultaneously* compute $Q_i = \begin{bmatrix} Q_i^f & Q_i^c \end{bmatrix}$ such that $\left\| Q_i^{f,\top} \begin{bmatrix} A_{s_i n_i} & A_{n_i s_i}^\top \end{bmatrix} \right\|_F \leq \varepsilon$. Define $U_i$ as a $k-$diagonal block matrix with $Q_i$ in its $i^{\text{th}}$ position and ones otherwise. Then consider $\overline{A} = U_k^\top \cdots U_1^\top A U_1 \cdots U_k$ and $\widetilde{A}$ defined as

$$\widetilde{A}_{s_i s_j} = \begin{bmatrix} 0 & 0 \\ 0 & Q_i^{c,\top} A_{s_i s_j} Q_j^c \end{bmatrix} \text{ if } i \neq j \text{ and } \widetilde{A}_{s_i s_i} = \overline{A}_{s_i s_i} \text{ otherwise.}$$

In short, off-diagonal block rows and columns corresponding to fine degrees of freedom are zeroed-out compared to $\overline{A}$, with the rest unchanged. We then find

$$\|\overline{A} - \widetilde{A}\|_F^2 = \sum_{i \neq j} \left\| \begin{bmatrix} Q_i^{f,\top} A_{s_i s_j} Q_j^f & Q_i^{f,\top} A_{s_i s_j} Q_j^c \\ Q_i^{c,\top} A_{s_i s_j} Q_j^f & 0 \end{bmatrix} \right\|_F^2 \tag{4.1}$$

$$\leq \sum_{i \neq j} \left\| \begin{bmatrix} Q_i^{f,\top} A_{s_i s_j} Q_j^f & Q_i^{f,\top} A_{s_i s_j} Q_j^c \end{bmatrix} \right\|_F^2 + \sum_{i \neq j} \left\| \begin{bmatrix} Q_i^{f,\top} A_{s_i s_j} Q_j^f \\ Q_i^{c,\top} A_{s_i s_j} Q_j^f \end{bmatrix} \right\|_F^2 \tag{4.2}$$

$$= \sum_{i \neq j} \left\| Q_i^{f,\top} A_{s_i s_j} \right\|_F^2 + \sum_{i \neq j} \left\| A_{s_i s_j} Q_j^f \right\|_F^2 \tag{4.3}$$

$$= \sum_{i \neq j} \left\| Q_i^{f,\top} A_{s_i s_j} \right\|_F^2 + \sum_{i \neq j} \left\| A_{s_j s_i} Q_i^f \right\|_F^2 \tag{4.4}$$

$$= \sum_{i \neq j} \left\| Q_i^{f,\top} A_{s_i s_j} \right\|_F^2 + \sum_{i \neq j} \left\| Q_i^{f,\top} A_{s_j s_i}^\top \right\|_F^2 \tag{4.5}$$

$$= \sum_i \sum_{j \neq i} \left( \left\| Q_i^{f,\top} A_{s_i s_j} \right\|_F^2 + \left\| Q_i^{f,\top} A_{s_j s_i}^\top \right\|_F^2 \right) \tag{4.6}$$

$$= \sum_i \left( \left\| Q_i^{f,\top} A_{s_i n_i} \right\|_F^2 + \left\| Q_i^{f,\top} A_{n_i s_i}^\top \right\|_F^2 \right) \tag{4.7}$$

$$= \sum_i \left\| Q_i^{f,\top} \begin{bmatrix} A_{s_i n_i} & A_{n_i s_i}^\top \end{bmatrix} \right\|_F^2 \tag{4.8}$$

$$\leq k\varepsilon^2 \tag{4.9}$$

(4.1) is by construction of $\widetilde{A}$ and $\overline{A}$. (4.2) is by property of the Frobenius norm, and the inequality is (possibly) loose since $\left\| Q_i^{f,\top} A_{s_i s_j} Q_f^f \right\|_2$ is counted twice. (4.3) is by invariance of the Frobenius norm to square orthogonal matrices, (4.4) follows by interchanging the role of $i$ and $j$, and (4.5) is by invariance to the Frobenius norm to the transpose. (4.6), (4.7) and (4.8) are by reordering and definition of $n_i$. Finally, (4.9) follows by construction of $Q_i^f$. We conclude that

$$\|U_k^\top \cdots U_1^\top A U_1 \cdots U_k - \widetilde{A}\|_F \leq \sqrt{k}\varepsilon.$$

We see that both approaches, Algorithm 4.2 and Algorithm 4.3, have a similar bound on the sparsification error. Namely, both approaches implicitly compute an orthogonal matrix $U$ such that, if sparsification is done with tolerance $\varepsilon$, and if $\widetilde{A}$ is the final matrix without

**Algorithm 4.3** Simultaneous sparsification algorithm. $Q_i$ is computed based on $A$, for all $i$. drop_fine drops entries corresponding to fine degrees of freedom.

**for** $i = 1, \ldots, k$ **do**
Sparsify $s_i$, i.e., compute $Q_i$ such that

$$\| Q_i^{f,\top} \begin{bmatrix} A_{s_i n_i} & A_{n_i s_i}^\top \end{bmatrix} \|_F \leq \varepsilon$$

**end for**
Compute $\overline{A} \leftarrow U_k^\top \cdots U_1^\top A U_1 \cdots U_k$
Compute $\widetilde{A} \leftarrow \text{drop\_fine}(\overline{A})$

the fine edges, $\|U^\top A U - \widetilde{A}\|_F = \mathcal{O}(\varepsilon)$ (note that the constant is smaller for Algorithm 4.3). Because transformations are orthogonal, there is no loss of accuracy by sparsifying all clusters at the same time. Other approaches, such as those using interpolative factorization [95, 106], don't use orthogonal transformations. As such, they don't benefit from this property.

But Algorithm 4.3 has a significant advantage: all sparsifications can happen in parallel. For Algorithm 4.2 to be parallel, one would have to compute a 1-coloring on the graph of $A$, mapping every interface $i$ to a color $c_i$ so that no two neighboring interfaces have the same color. Sparsification would then have to be ordered by color, i.e., $s_i$ is sparsified before $s_j$ if and only if $c_i < c_j$. This is similar to what is done in the parallel version of LoRaSp, see [43]. As a result, we decided to use Algorithm 4.3 to maximize concurrency. We note that Algorithm 4.3 has a slightly increased flop count. For neighboring clusters $i$, $j$ such that $i$ would have been sparsified before $j$, the sparsification of $s_j$ does not benefit from the reduced size of $s_i$. However, given the added concurrency and the simplicity (no coloring needed) of the simultaneous sparsification algorithm, this is the approach taken in this work.

### 4.2.3 Task-Based Algorithm using TaskTorrent

We now turn to the description of the algorithm. The algorithm is nearly identical to the sequential version except that:

- partitioning is done as explained in Section 4.2.1;
- sparsification is done using the simultaneous version as described in Section 4.2.2;

We then implement this using `TTor` and a PTG approach. We parallelize each level

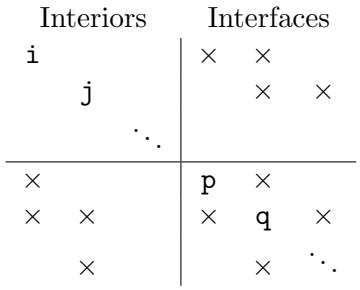



Figure 4.2: Example matrix

independently from each other. This means there is a synchronization point *between* each level, and in the following, we consider a specific level in the algorithm.

Using a PTG approach requires identifying tasks and all their dependencies. Consider a trailing matrix at a given level as illustrated in Figure 4.2. $i$ and $j$ are leaves (interiors) and $p$ and $q$ are interfaces (i.e., subsets of the remaining separators). $\times$ represent non-zero blocks in the matrix.

We create tasks by transforming every block matrix operation into a task:

- Elimination of an interior creates three different types of tasks: pivot factorization (`getrf`), panel update (`trsm`) and Schur complement updates (`gemm`);
- Block scaling of an interface creates two types of tasks: pivot factorization (`getrf`) and panel update (`trsm`);
- Sparsification of an interface creates two types of tasks: rank-revealing factorization (`geqp3`), i.e., compute $Q_i$, and trailing matrix update (`ormqr`), i.e., updating $A_{ij} \leftarrow Q_i^{c,\top} A_{ij} Q_j^c$.

Dependencies depend directly on the input and output of tasks. For instance, `geqp3` requires as input the blocks located in the row and column of the associated diagonal block. Its output, $Q_i$, is then needed for the `ormqr` associated with every block in its row and column.

Figure 4.3 illustrates the PTG formulation for the elimination, Figure 4.4 for the scaling, and Figure 4.5 for the sparsification. Every table shows the factorization (top), the local PTG (middle), and an illustration of the dependencies in terms of the trailing matrix (bottom).

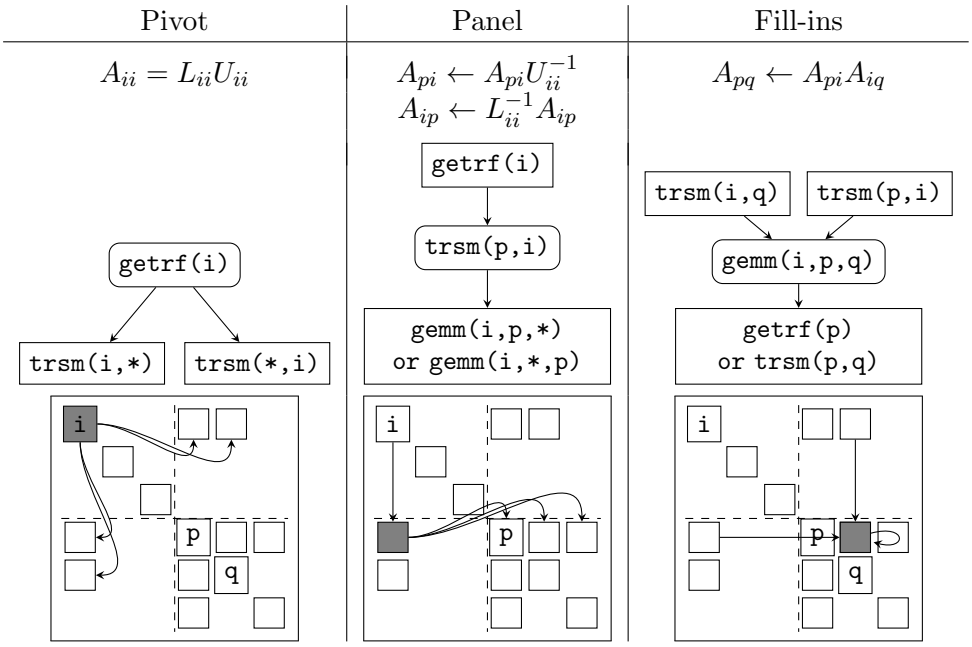



Figure 4.3: Task-based elimination. $L_{ii}U_{ii}$ represents a generic factorization where $L_{ii}$ and $U_{ii}$ can both be quickly inverted (Cholesky, partial pivoted LU, etc.). The `gemm(i,p,q)` tasks can happen in any order but are bound to a specific thread to prevent race conditions.

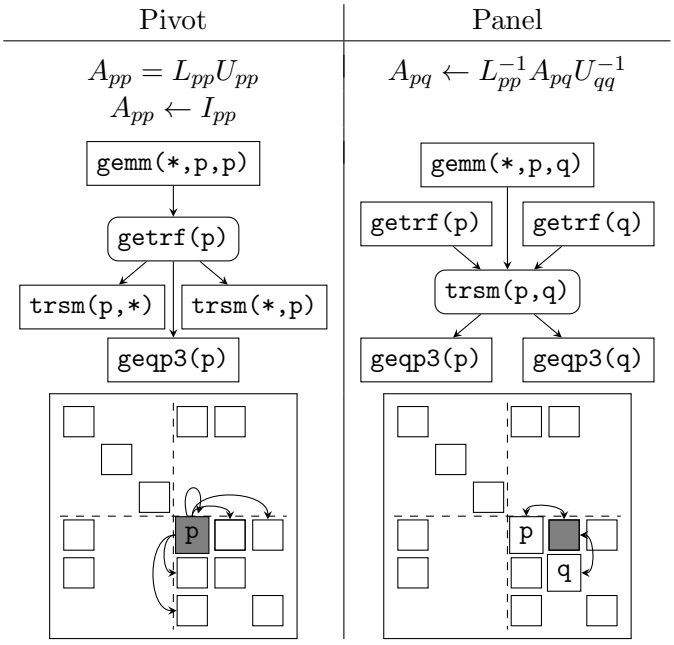



Figure 4.4: Task-based block scaling

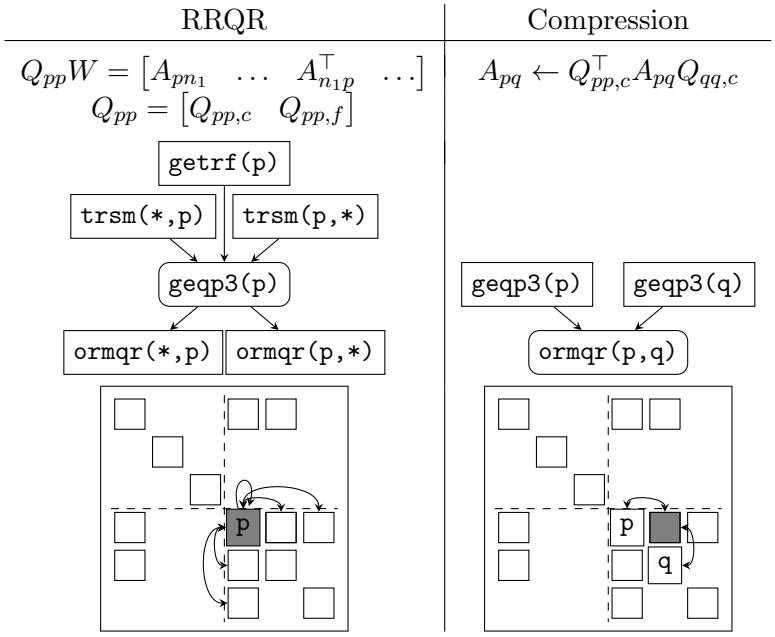



Figure 4.5: Task-based sparsification

Figure 4.6 shows a small section of the task DAG (arising in the Ice-Sheet benchmark, see Section 4.3.2). We recall that the parallelization is level-wise. Hence, the DAG is relatively shallow and wide, with a fixed depth of seven (three for elimination, two for block scaling, and two for sparsification). Figure 4.7 shows all the tasks at all levels of the algorithm, ordered from bottom levels (bottom) to top levels (top).

We finally present some of the `TTor` code regarding the `geqp3` task flow (compare with Figure 4.5). We recall the following.

- `geqp3(p)` requires, as inputs, the blocks $A_{pn}$ and $A_{np}$, for all neighbor $n$ of $p$;
- `geqp3(p)` performs a low-rank approximation of $\begin{bmatrix} A_{pn_1} & \dots & A_{pn_k} & A_{n_1p}^\top & \dots & A_{n_kp}^\top \end{bmatrix}$, leading to an orthogonal tall-and-skinny $Q_p^c$;
- `geqp3(p)` outputs $Q_p^c$, which is the input of the tasks `ormqr(p,n)` and `ormqr(n,p)` for all neighbor $n$ of $p$.

Now assume `Taskflow<int> geqp3` has been defined. We first express the number of incoming dependencies for a given diagonal block `k`.

```
geqp3.set_indegree([&](int k) {
    Cluster* self = get_cluster_local(k);
```

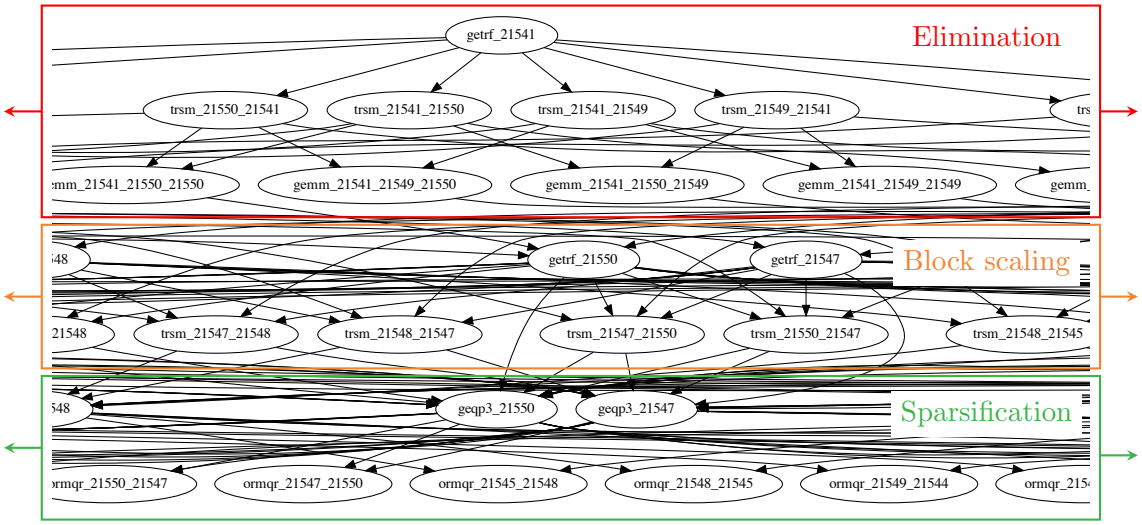



Figure 4.6: Example of task DAG for the ice-sheet problem at a particular level. This shows only a portion of the DAG, which expands further on the left and right.

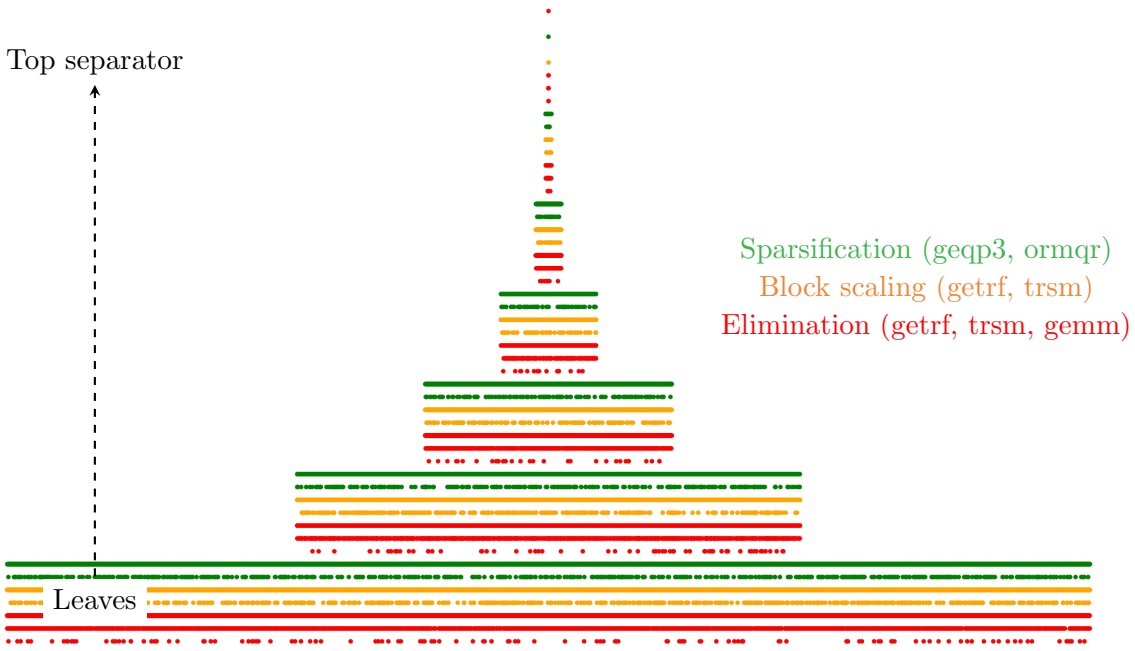



Figure 4.7: Illustration of all the DAGs for the ice-sheet problem. From lower levels (bottom) to top levels (top). Colors indicate the kind of task.

```
    return 1 + // Pivot
           self->edgesColNbrSparsification().size() + // Trsm's in column
           self->edgesRowNbrSparsification().size();  // Trsm's in row
});
```

Then, we provide a function to map task `k` to a thread.

```
geqp3.set_mapping([&](int p) {
    return p % ttor_threads;
});
```

Then, we indicate the computational routine to perform when task `k` is ready.

```
geqp3.set_task([&](int p) {
    this->sparsify_adaptive_only_Q(p);
});
```

Finally, we provide the function used to fulfill dependencies. This is the most complex part of using `TTor`. This function first collects all the neighboring MPI ranks and which blocks $A_{pn}$, $A_{np}$ are resident on those MPI ranks. Then, it sends one active message per neighboring MPI rank, fulfilling the dependencies of the `ormqr` task flow on those blocks $(p, n)$ and $(n, p)$. Note that the implementation computes $Q_p^c$ both as a tall-and-skinny orthogonal matrix (`Q` in the code), as well as a set of householder reflectors (referred to as `h` and `v` in the code).

```
geqp3.set_fulfill([&](int p) {
    Cluster* self = get_cluster_local(p);
    // Collect a map rank -> tasks to fulfill
    map<int,vector<int2>> ff;
    for(auto erow: self->edgesRowNbrSparsification()) {
        int dest = edge2rank(erow);
        if(dest == my_rank) {
            ormqr.fulfill_promise(erow);
        } else ff[dest].push_back(erow);
    }
    for(auto ecol: self->edgesColNbrSparsification()) {
        int dest = edge2rank(ecol);
        if(dest == my_rank) {
            ormqr.fulfill_promise(ecol);
        } else ff[dest].push_back(ecol);
    }
    // Send to all neighbors
    for(auto& r_ff: ff) {
        int size = self->get_v()->rows();
        int rank = self->get_v()->cols();
        auto ff_view = view(&r_ff.second);
        auto Q_view = view(self->get_Q());
        auto v_view = view(self->get_v());
        auto h_view = view(self->get_h());
        geqp3_am->send(r_ff.first, p, size, rank, ff_view,
```

```
                        Q_view, v_view, h_view);
    }
});
```

This function requires the definition of the `geqp3_am` active message, which we finally provide. This active message (1) stores $Q_p^c$ on the receiver and (2) fulfill the local `ormqr` tasks.

```
auto geqp3_am = comm.make_active_msg(
    [&](int &p, int& size, int& rank, ttor::view<EdgeGID>& ff,
        ttor::view<double>& Q_data, ttor::view<double>& v_data,
        ttor::view<double>& h_data) {
        GhostCluster* s = get_cluster(p);
        GhostEdge* pp = get_pivot(p);
        // Copy Q, v, h
        s->set_Q(make_matrix(Q_data, size, rank));
        s->set_v(make_matrix(v_data, size, rank));
        s->set_h(make_vector(h_data, rank));
        // Make pivot identity
        pp->make_pivot_identity(rank);
        // Fulfill dependencies
        for(auto& e: ff) {
            ormqr.fulfill_promise(e);
        }
    });
```

This is the entirety of the code used to define the `geqp3` task flow.

## 4.3 Numerical results

We now show some benchmarks of the proposed approach on large problems. The goal of this section is to demonstrate that `TTor`'s approach scales well on sparse, fast block algorithms.

We implemented parallel spaND in C++ using TaskTorrent [34]. We use Zoltan [22] for algebraic and geometric recursive bisection. The SPE and Naca0012 benchmarks were run on a Stanford HPC Center cluster equipped with dual-sockets and 16 cores Intel(R) Xeon(R) CPUE5-2670 0 @2.60GHz with 32GB of RAM per node. Intel Compiler (version 19.1.0.166) and Intel MPI are used with Intel MKL (version 2020.0) for BLAS and LAPACK. The Naca0012 benchmark comes from the SU2 flow code [57], in which spaND was integrated. The Ice-Sheet benchmarks were run on the LLNL Quartz machine. Each node has a dual-socket 36 cores Intel(R) Xeon(R) E5-2695 v4 with 128 GB of RAM. GCC (version 8.1.0) is used with OpenMPI (version 4.0.0) and Intel MKL (version 2020.0).

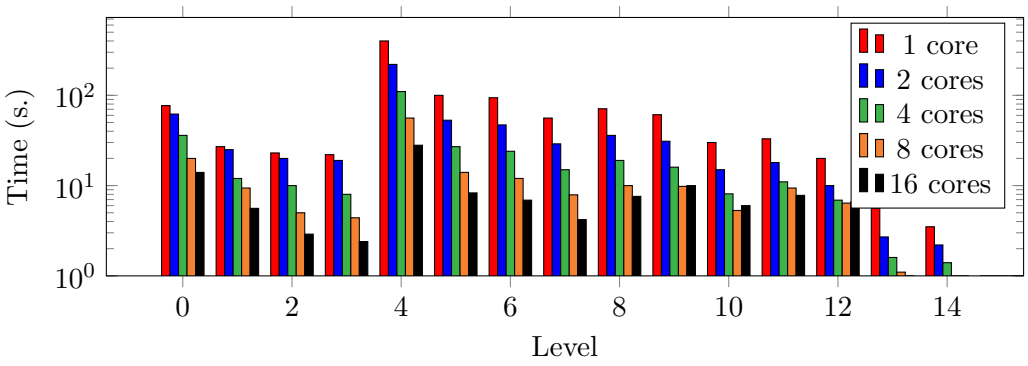



Figure 4.8: SPE profile (elimination, scaling, and sparsification combined), strong scalings, from 1 to 16 cores.

### 4.3.1 SPE

We start with the SPE benchmark [47]. This problem corresponds to the discretization of a scalar elliptic PDE in 3D with a 7-points stencil over a regular cube. As such, top separators are relatively large, with size $\approx 1000$ for problems of size $8M$. In addition, those get larger as the problem becomes larger. As such, we perform strong scalings on 1 node, using from 1 to 16 cores, on the problem of size 8M. We use $\varepsilon = 10^{-2}$, use $L = 17$, skip sparsification for 4 levels and use Cholesky for block pivot factorization. In that case, the trailing matrix provably remains SPD and spaND is guaranteed to never break down. Figure 4.8 shows the time spent on each level with various numbers of cores (i.e., `TTor`'s threads).

Since this problem has a moderate size, most of the time is spent in the first few sparsification levels. Hence, there is much concurrency available, and the algorithm strong scales relatively well. Larger test cases would exhibit larger ranks near the top, with worst strong scalings. Eventually, the top levels will start dominating.

### 4.3.2 Ice-Sheet

We continue with some results on an Ice-Sheet modeling problem. Those problems come from the modeling of the movement of ice in Antarctica [145]. Stokes equations are used, and a modeling assumption leads to a non-linear equation. The mesh is a vertically extruded 2D mesh. Finally, a Newton method is used, and we consider the matrices arising in the fourth step. We consider a mesh with 10 vertical layers and with horizontal resolutions from 16km (1M problem size) to 1km (296M problem size).

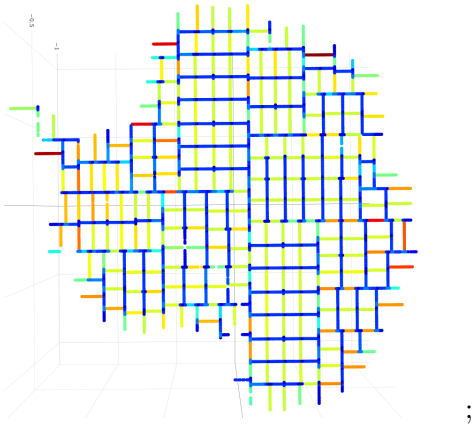
| | | spaND | | | | | AMG (Hypre) | |
|---|---|---|---|---|---|---|---|---|
| cores | $N$ | $t_{fact}$ | $t_{app}$ | $t_{solve}$ | $t_{fact}+t_{solve}$ | $n_{CG}$ | $t_{fact}+t_{solve}$ | $n_{CG}$ |
| 36 | 1M | 6.2 | 0.14 | 0.9 | 7.1 | 6 | 15 | 427 |
| 144 | 4M | 7.3 | 0.15 | 1.2 | 8.5 | 6 | 16 | 456 |
| 576 | 18M | 8.9 | 0.15 | 1.6 | 10.5 | 7 | 22 | 527 |
| 2304 | 74M | 9.8 | 0.17 | 1.9 | 11.7 | 8 | 29 | 627 |
| 9216 | 296M | 13.2 | 0.21 | 3.7 | 16.9 | 12 | 39 | 623 |

Table 4.1: Ice-sheet results, weak scalings, from 36 to 9216 cores.

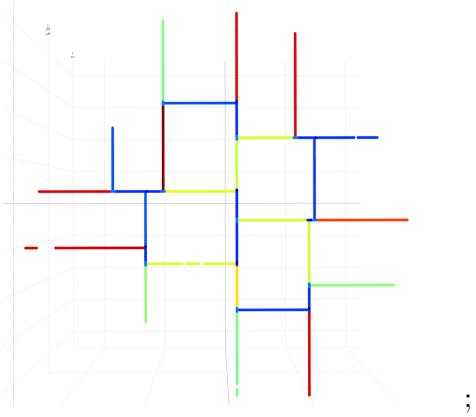
We perform weak scalings and compare spaND (using Cholesky for block scaling with $\varepsilon = 10^{-2}$) with Hypre [59] with Boomer AMG. For AMG, the matrix is distributed using ParMetis [101] K-way and Boomer AMG uses a strong threshold of 0.9. For spaND we bind 1 MPI rank per socket (with `TTor` using all cores through threads), while for Hypre, we bind one MPI rank per core. Both use CG with a residual tolerance of $10^{-8}$. We use from 36 to 9216 cores, with a matrix size from 1.1M to 296M. Table 4.1 shows the result. We observe that the factorization time scales well, albeit with a small uptick in time for the largest test case. The number of CG steps is also close to constant, only doubling from the smallest to largest test case. Overall spaND is scaling similarly to Hypre and is more competitive in terms of time-to-solution.

Figure 4.9 (top) shows the ranks across the physical domain at various levels. Our load balancing heuristic assigns vertices to MPI ranks based on the initial partitioning of $A$. In particular, it does not try to predict ranks, which are directly related to the number of flops. However, in this problem, we observe that the ranks are very uniform. Strong localization of some high ranks would lead to a poor load balancing.

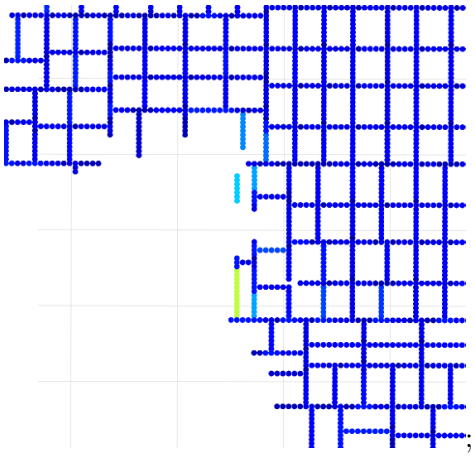
Figure 4.10 shows the rank across levels for all problem sizes. We display both the average and maximum ranks. As expected on 2D-like problems, we see that ranks grow very slowly with the problem size. Average ranks show less than a 10% increase between problems of size $N$ and $4N$. We however observe that the maximum ranks show large values towards the leaves. Figure 4.9 (bottom) shows that this is due to imperfections in the partitioning at the boundaries of the domain. However, this effect is limited to the first few levels and does not impact the overall runtime of the algorithm.



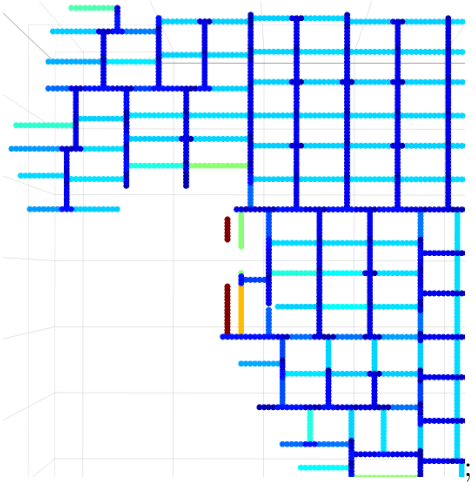
(a) Ranks, 16km case, $\ell = 4$. Ranks from 22 to 350, average of 98



(b) Ranks, 16km case, $\ell = 8$. Ranks from 1 to 291, average of 100



(c) Location of some of the high ranks, 8km case, $\ell = 2$ (after sparsification)



(d) Location of some of the high ranks, 8km case, $\ell = 3$ (before sparsification)

Figure 4.9: Ice-sheet ranks. (a-b): all ranks at various levels of the hierarchy. (c-d): Location of a few high ranks.

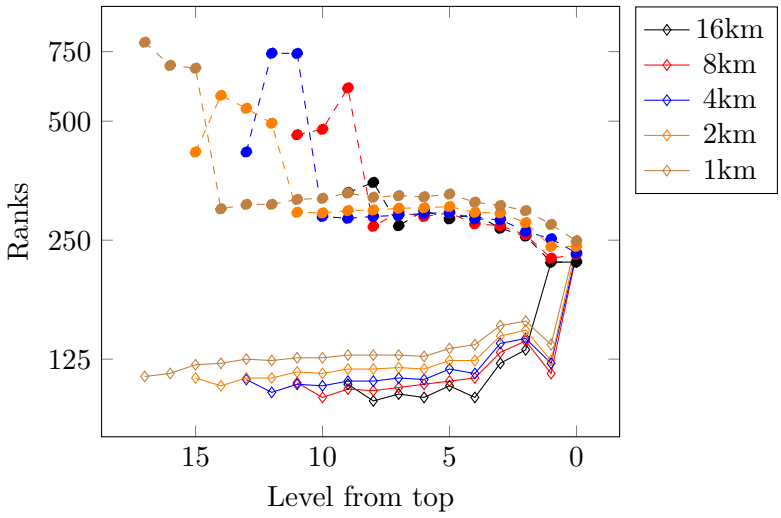
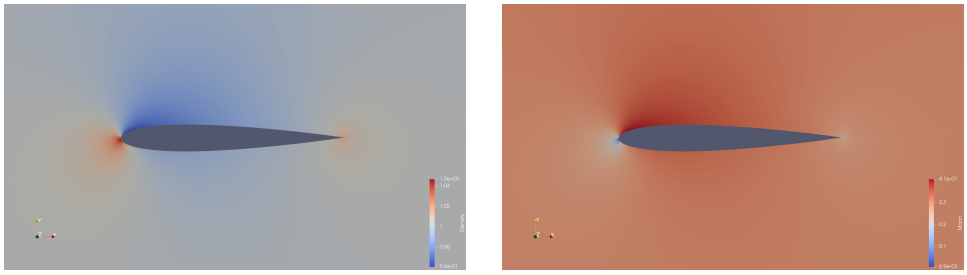



Figure 4.10: Ice-sheet ranks for all problem sizes. Averages (solid, diamond) and maximums (dashed, circle).

### 4.3.3 NACA airfoil

We finish with a test problem from fluid dynamics. This is the classical NACA0012 test case, with an airfoil at the center and a mesh stretching around. This is a classical, simple, benchmark used to evaluate flow solvers. The matrix comes from the implicit inviscid DG discretization of the Navier-Stokes equations solved using Newton's method and with second-order elements. We use the freely-available SU2 [57] solver. The matrix is unsymmetric but with a symmetric sparsity pattern and has a natural block structure, with blocks of size $16 \times 16$ corresponding to each element, and connected to the adjacent four elements. Finally, the mesh is regular (i.e., derived from a regular mesh) but highly non-uniform. As such, we assign to each element a tuple $(i, j)$ where $i$, $j$ are integers increasing monotonically with the radius and the angle, respectively. The matrix is then partitioned using those $(i, j)$ coordinates with a standard recursive bisection algorithm as described before. We consider test cases with a matrix size from $250k$ to $67M$, and perform again weak scalings. Figure 4.11 illustrates the solution of the PDE on the $N = 4M$ test case, zooming over the airfoil.

We then solve the linear systems using spaND with `TTor`. spaND and `TTor` have been integrated into the SU2 solver. Because `TTor` is based on message passing with MPI, integration into an existing MPI-based codebase is seamless. We use partial pivoted LU as



(a) Density around the airfoil (b) Mach number around the airfoil

Figure 4.11: Naca solution for $N = 4M$.

block scaling algorithm, $\varepsilon = 10^{-3}$, and GMRES with a tolerance of $10^{-3}$ on the residual.

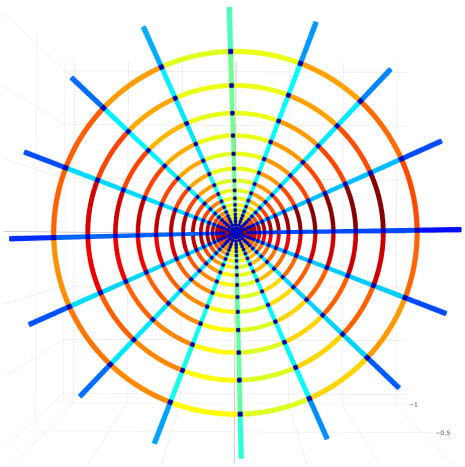
Figure 4.12 shows the ranks throughout the domain. This clearly shows a strong directionality, with ranks much higher ahead and behind the wing, but smaller above and below. This presents challenges since, as a consequence, the load balancing is much less favorable.

Finally, Figure 4.13 presents all the results throughout all Newton iterations. We make multiple observations:

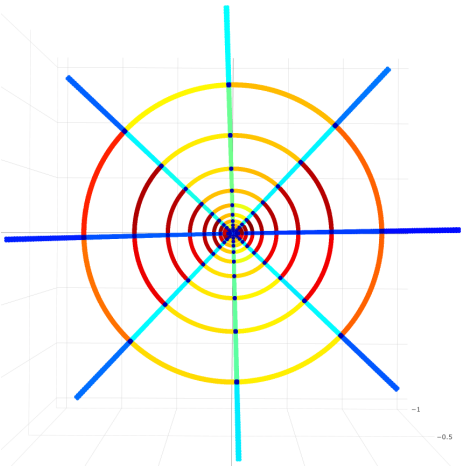
- The average ranks grow slowly with the problem size. On the other hand, maximum ranks grow quicker than expected, and are not localized to the leaves. This can slow down spaND significantly since high ranks lead to longer sparsification time and high ranks towards the top of the tree lead to poor concurrency.
- On the other hand, the number of GMRES steps remains very low at all problem sizes and does not vary much during the Newton iteration.
- The factorization time scales well with the problem size, albeit with an increase in the larger test case. We conjecture that the relatively high maximum ranks are degrading performance.
- The solve time scales roughly like the number of GMRES steps, indicating that applying the preconditioner scales well with the problem size.

## 4.4 Benefits and limitations of the `TTor` approach

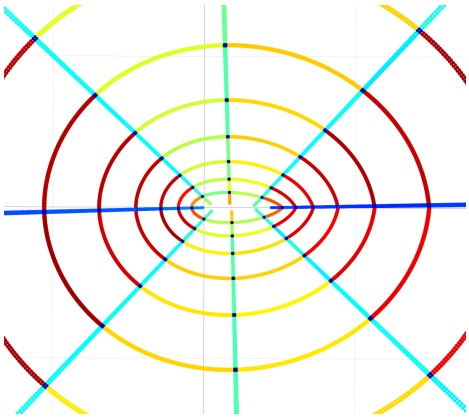
In this work, we parallelized spaND using `TTor`. This presents a couple of advantages but has some limitations. We here discuss some of those.



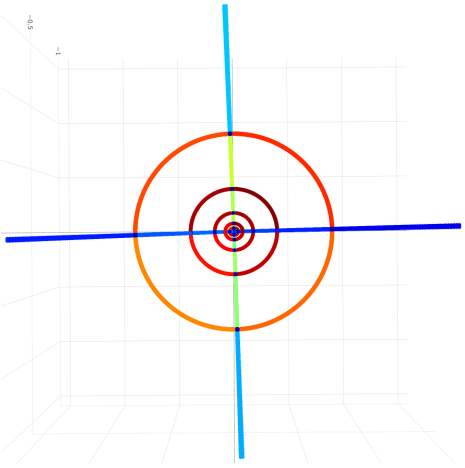
(a) $\ell = 4$. Ranks from 40 (blue) to 296 (brown), average of 162



(b) $\ell = 6$. Ranks from 47 (blue) to 353 (brown), average of 198

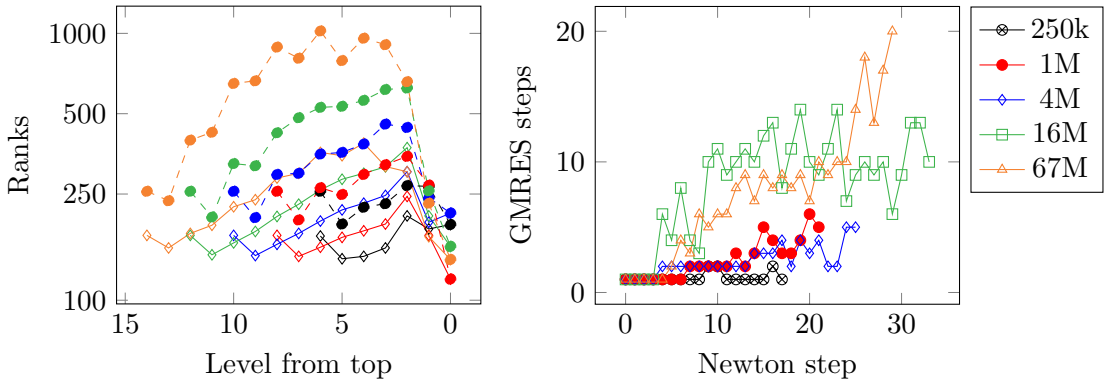


(c) $\ell = 6$, zoom on the airfoil.



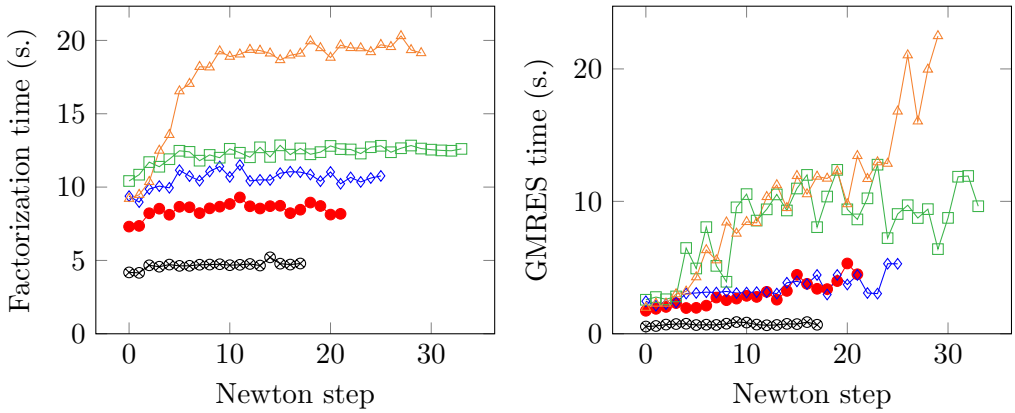
(d) $\ell = 8$. Ranks from 40 (blue) to 385 (brown), average of 231

Figure 4.12: Naca ranks at the last Newton step, step 25, with $\varepsilon = 0.001$ for $N = 4M$ and various level $\ell$ with $L = 13$.




(a) Ranks at the last Newton step. Average (diamond) and maximums (circle).

(b) GMRES steps




(c) Factorization time

(d) Solve time

Figure 4.13: Naca results, weak scalings, from 8 cores (1 node) for 250k to 2048 cores (128 nodes) for 67M. spaND using PLU, $\varepsilon = 10^{-3}$, and GMRES with $10^{-3}$ tolerance

- The approach is well suited for (sparse) tiled matrix algorithms, where every block operation generates a corresponding task in the DAG. This makes writing the PTG code simple since there is a one-to-one mapping between tasks and block operations. In this work, this is exactly what is done. Every block operation between interiors and/or interfaces generates a task.

- The code is made simpler if all distributed data structures are entirely precomputed. Tasks are then just filling (initially unallocated) dense blocks with the appropriate values at runtime. Changing data structures at runtime risks introducing data races that are difficult to detect.

- Similarly, `TTor` is easy to use when dependencies are simple to compute. In spaND, dependencies depend entirely on the matrix graph (in which there is an edge between $i$ and $j$ if and only if block $A_{ij}$ is non-zero). As such, knowledge of neighbors is enough to compute the in-degree of every task and to know what tasks to fulfill at the end of a task.

- In the largest test cases studied, the DAG is very large near the leaves. In this case, the PTG+AMs approach taken by `TTor` works well since the DAG is entirely distributed and explored in parallel.

- Since `TTor` is based around message passing and MPI, it is fully compatible with any MPI-based library. In this work, we integrated spaND into the SU2 [57] codebase without any particular hurdle. It was equivalent to integrating a regular MPI-based library. We were able to use `TTor` and spaND only in a portion of the code, before returning control to SU2. This is important to enhance the adoption of task-based runtime systems.

- `TTor`'s approach is based on message passing. This means every block in the matrix is resident on a particular MPI rank. There is no built-in dynamic load balancing that would migrate tasks and data across the system while computations are happening. This may pose problems when the load is difficult to predict (such as in the Naca0012 benchmark), and the runtime system does not provide a solution to this particular issue (to the best of our knowledge, neither do Legion, Regent, or StarPU).

- `TTor`'s approach may be difficult to use when there is no simple mapping between

tasks and block operations. We note that this is an issue in any task-based runtime system.

On the other hand, `TTor`'s design makes it possible to interface with external MPI-based libraries. In particular, any algorithm trying to address the issue of the large ranks will require some form of distributed RRQR. The simplest way to achieve this would be to use for instance ScaLAPACK, a classical MPI-based library. Since `TTor` is based on MPI and C++, this is possible and relatively easy.

- Finally, `TTor`'s approach of PTG+AMs typically requires the DAG to encapsulate *everything*. In particular, it is difficult to compose functions and to create abstractions without introducing synchronization. For instance, if one had a task-based distributed RRQR library implemented with `TTor`, it may be difficult to integrate that within the spaND DAG while avoiding synchronization during the call to the library. A way to compose functions using various DAGs would present a significant improvement. This would also allow for easier testing of various parts of the algorithm independently from each other. One way to do this would be by letting tasks on the first DAG fulfill dependencies on arbitrary tasks in another DAG in a generic way, without knowing precisely what that second DAG is.

## 4.5 Conclusions

In this work, we designed a task-based distributed version of the spaND algorithm using `TTor`. We explained the implementation and performed both strong and weak scalings experiments. Strong scalings exhibit good performances on problems of moderate sizes where the top ranks are not too large. Weak scalings exhibit excellent performances on very large problems where ranks grow slowly with $N$. This demonstrates the good performances of a task-based approach using `TTor`, and show that `TTor` scales very well on large problems.

Future work should be centered around addressing the large ranks arising on large 3D problems. This is the main limitation of our approach at the moment. While this is conceptually simple, it poses three practical issues. First, the partitioning needs to encode that information (i.e., what MPI rank participates in which RRQR). Second, tasks on various MPI ranks are usually independent in any task-based runtime system. The implementation will need to take this into account, by having all MPI ranks start the distributed RRQR together. Finally, the scaling of RRQR in a distributed setting is unclear.

However, results such as [106] indicate that random-sampling based approaches exhibit good performance.

# Chapter 5

# Skeletonized Interpolation



## 5.1 Introduction

In this work, we are interested in the low-rank approximation of kernel matrices, i.e., matrices $K_{ij}$ defined as

$$K_{ij} = \mathscr{K}(x_i, y_j)$$

for $x_i \in X = \{x_1, \ldots, x_m\} \subseteq \mathcal{X}$ and $y_j \in Y = \{y_1, \ldots, y_n\} \subseteq \mathcal{Y}$ and where $\mathscr{K}$ is a smooth function over $\mathcal{X} \times \mathcal{Y}$. A typical example is when

$$\mathscr{K}(x, y) = \frac{1}{\|x - y\|_2}$$

and $X \subset \mathcal{X}$ and $Y \subset \mathcal{Y}$ are two well-separated sets of points.

This kind of matrices arises naturally when considering integral equations like

$$a(x)u(x) + \int_{\tilde{\mathcal{Y}}} \mathscr{K}(x, y)u(y)\mathrm{d}y = f(x) \quad \forall x \in \tilde{\mathcal{X}}$$

where the discretization leads to a linear system of the form

$$a_i u_i + \sum_j K_{ij} u_j = f_i \tag{5.1}$$

where $K$ is a *dense* matrix. While this linear system as a whole is usually not low-rank, one can select subsets of points $X \subset \mathcal{X}$ and $Y \subset \mathcal{Y}$ such that $\mathscr{K}$ is smooth over $\mathcal{X} \times \mathcal{Y}$ and hence $\mathscr{K}(X, Y)$ is low-rank. This corresponds to a submatrix of the complete $K$. Being able to efficiently compute a low-rank factorization of such submatrix would lead to significant computational savings. By "smooth" we usually refer to a function with infinitely many continuous derivatives over its domain. Such a function can be well approximated by its interpolant at Chebyshev nodes for instance.

Low-rank factorization means that we seek a factorization of $K = \mathscr{K}(X, Y)$ as

$$K = USV^\top$$

where $U \in \mathbb{R}^{m \times r}, V \in \mathbb{R}^{n \times r}$, $S \in \mathbb{R}^{r \times r}$, and $r$ is the rank. In that factorization, $U$ and $V$ don't necessarily have to be orthogonal. One way to compute such a factorization is to first compute the matrix $K$ at a cost $\mathcal{O}(mn)$ and then to perform some rank-revealing factorization like SVD, rank-revealing QR or rank-revealing LU at a cost usually proportional to $\mathcal{O}(mnr)$. But, even though the resulting factorization has a storage cost of $\mathcal{O}((m+n)r)$, linear in the size of $X$ and $Y$, the cost would be proportional to $\mathcal{O}(mn)$, i.e., quadratic.

### 5.1.1 Notation

In the following, we will denote by $\mathscr{K}$ a function over $\mathcal{X} \times \mathcal{Y}$. $X$ and $Y$ are finite sequences of vectors such that $X \subset \mathcal{X}$ and $Y \subset \mathcal{Y}$ and $\mathscr{K}(X, Y)$ denotes the matrix $K_{ij} = \mathscr{K}(x_i, y_j)$. Small-case letters $x$ and $y$ denote arbitrary variables, while capital-case letters $\overline{X}$, $\widehat{X}$, $\check{X}$, $\widetilde{X}$ denotes sequences of vectors. We denote matrices like $A(X, Y)$ when the rows refer to the set $X$ and the columns to the set $Y$. Table 5.1 summarizes all the symbols used in this chapter.

### 5.1.2 Previous work

The problem of efficiently solving (5.1) has been extensively studied in the past. As indicated above, discretization often leads to a dense matrix $K_{ij}$. Hence, traditional techniques such as the LU factorization cannot be applied because of their $\mathcal{O}(n^3)$ time and even $\mathcal{O}(n^2)$ storage complexity. The now traditional method used to deal with such matrices is to use the fact that they usually present a (hierarchically) low-rank structure, meaning we can represent the matrix as a hierarchy of low-rank blocks. The Fast Multipole Method

| | |
|---|---|
| $\mathscr{K}$ | The smooth kernel function |
| $\mathcal{X}, \mathcal{Y}$ | The spaces over which $\mathscr{K}$ is defined, i.e., $\mathcal{X} \times \mathcal{Y}$ |
| $x$, $y$ | Variables, $x \in \mathcal{X}$, $y \in \mathcal{Y}$ |
| $X$, $Y$ | The mesh of points over which to approximate $\mathscr{K}$, i.e., $X \times Y$ |
| $K$ | The kernel matrix, $K = \mathscr{K}(X, Y)$, $K_{ij} = \mathscr{K}(x_i, y_j)$ |
| $m$, $n$ | $m = \lvert X \rvert$, $n = \lvert Y \rvert$ |
| $\overline{X}$, $\overline{Y}$ | The tensor grids of Chebyshev points |
| $\widehat{X}$, $\widehat{Y}$ | The subsets of $\overline{X}$ and $\overline{Y}$ output by the algorithm used to build the low-rank approximation |
| $\check{X}$, $\check{Y}$ | The complements of the above, i.e., $\check{X} = \overline{X} \setminus \widehat{X}$, $\check{Y} = \overline{Y} \setminus \widehat{Y}$ |
| $\overline{m}$, $\overline{n}$ | The number of Chebyshev tensor nodes, $\overline{m} = \lvert\overline{X}\rvert$, $\overline{n} = \lvert\overline{Y}\rvert$ |
| $r_0$ | The "interpolation" rank of $\mathscr{K}$, i.e., $r_0 = \min(\lvert\overline{X}\rvert, \lvert\overline{Y}\rvert)$ |
| $r_1$ | The Skeletonized Interpolation rank of $\mathscr{K}$, i.e., $r_1 = \lvert\widehat{X}\rvert = \lvert\widehat{Y}\rvert$ |
| $r$ | The rank of the continuous SVD of $\mathscr{K}$ |
| $S(x, \overline{X})$, $T(y, \overline{Y})$ | Row vectors of the Lagrange basis functions, based on $\overline{X}$ and $\overline{Y}$ and evaluated at $x$ and $y$, respectively. Each column is one Lagrange basis function. |
| $\widehat{S}(x, \widehat{X})$, $\widehat{T}(y, \widehat{Y})$ | Row vectors of Lagrange basis functions, based on $\widehat{X}$ and $\widehat{Y}$, built using the Skeletonized Interpolation and evaluated at $x$ and $y$, respectively. Each column is one function. |
| $w_k$, $w_l$ | Chebyshev integration weights |
| $\mathrm{diag}(\overline{W}_X)$, $\mathrm{diag}(\overline{W}_Y)$ | Diagonal matrices of integration weights when integration is done at nodes $\overline{X}$ and $\overline{Y}$ |
| $\mathrm{diag}(\widehat{W}_X)$, $\mathrm{diag}(\widehat{W}_Y)$ | Subset of $\mathrm{diag}(\overline{W}_X)$ and $\mathrm{diag}(\overline{W}_Y)$ corresponding to rows and columns $\widehat{X}$ and $\widehat{Y}$ |

Table 5.1: Notations used in this chapter

(FMM) [131, 79, 12] takes advantage of this fact to accelerate computations of matrix-vector products $Kv$ and one can then couple this with an iterative method. More recently, [65] proposed a kernel-independent FMM based on interpolation of the kernel function.

Other techniques compute explicit low-rank factorization of blocks of the kernel matrix through approximations of the kernel function. The Panel Clustering method [87] first computes a low-rank approximation of $\mathscr{K}(x,y)$ as

$$\mathscr{K}(x,y) \approx \sum_i \kappa_i(x;y_0)\phi_i(y)$$

by Taylor series and then uses it to build the low-rank factorization.

Bebendorf and Rjasanow proposed the Adaptive Cross Approximation [18], or ACA, as a technique to efficiently compute low-rank approximations of kernel matrices. ACA has the advantage of only requiring to evaluate rows or columns of the matrix and provides a simple yet very effective solution for smooth kernel matrix approximations. However, it can have convergence issues in some situations (see for instance [25]) if it cannot capture all necessary information to properly build the low-rank basis and lacks convergence guarantees.

In the realm of analytic approximations, [163] (and similarly [24], [25], [65] and [153] in the Fourier space) interpolate $\mathscr{K}(x,y)$ over $\mathcal{X}\times\mathcal{Y}$ using classical interpolation methods (for instance, polynomial interpolation at Chebyshev nodes in [65]), resulting in expressions like

$$\mathscr{K}(x,y) \approx S(x,\widetilde{X})\mathscr{K}(\widetilde{X},\widetilde{Y})T(y,\widetilde{Y})^\top = \sum_k \sum_l S_k(x)\mathscr{K}(\tilde{x}_k,\tilde{y}_l)T_l(y)$$

where $S$ and $T$ are Lagrange interpolation basis functions. Those expressions can be further recompressed by performing a rank-revealing factorization on the node matrix $\mathscr{K}(\widetilde{X},\widetilde{Y})$, for instance using SVD [65] or ACA [25]. Furthermore, [163] takes the SVD of a scaled $\mathscr{K}(\widetilde{X},\widetilde{Y})$ matrix to further recompress the approximation and obtain an explicit expression for $u_r$ and $v_r$ such that

$$\mathscr{K}(x,y) \approx \sum_s \sigma_s u_s(x) v_s(y)$$

where $\{u_s\}_s$ and $\{v_s\}_s$ are sequences of orthonormal functions in the usual $L_2$ scalar product.

Bebendorf [15] builds a low-rank factorization of the form

$$\mathscr{K}(x,y) = \mathscr{K}(x,\widetilde{Y})\mathscr{K}(\widetilde{X},\widetilde{Y})^{-1}\mathscr{K}(\widetilde{X},y) \tag{5.2}$$

where the nodes $\widetilde{X}$ and $\widetilde{Y}$ are interpolation nodes of an interpolation of $\mathscr{K}(x, y)$ built iteratively. Similarly, in their second version of the Hybrid cross approximation algorithm, Börm and Grasedyck [25] propose applying ACA to the kernel matrix evaluated at interpolation nodes to obtain pivots $\widetilde{X}_i$, $\widetilde{Y}_j$, and implicitly build an approximation of the form given in (5.2). Both those algorithms resemble our approach in that they compute pivots $\widetilde{X}, \widetilde{Y}$ in some way and then use (5.2) to build the low-rank approximation. In contrast, our algorithm uses weights and has stronger accuracy guarantees. We highlight those differences in Section 5.5.

Our method inserts itself amongst those low-rank kernel factorization techniques. However, with the notable exception of ACA, those methods often either rely on analytic expressions for the kernel function (and are then limited to some specific ones), or have suboptimal complexities, i.e., greater than $\mathcal{O}(nr)$. In addition, even though we use interpolation nodes, it is worth noting that our method differs from interpolation-based algorithms as we never explicitly build the $S(x, \widetilde{X})$ and $T(y, \widetilde{Y})$ matrices containing the basis functions. We merely rely on their existence.

$\mathcal{H}$-matrices [83, 85, 82] are one way to deal with kernel matrices arising from boundary integral equations that are Hierarchically Block Low-Rank. The compression criterion (i.e., which blocks are compressed as low-rank and which are not) leads to different methods, usually denoted as strongly-admissible (only compress well-separated boxes) or weakly-admissible (compress adjacent boxes as well). In the realm of strongly-admissible $\mathcal{H}$-matrices, the technique of Ho & Ying [94] as well as Tyrtyshnikov [149] are of particular interest to us. They use Skeletonization of the matrix to reduce storage and computation cost. In [94], they combine Skeletonization and Sparsification to keep compressing blocks of $\mathcal{H}$-matrices. [149] uses a somewhat non-traditional Skeletonization technique to also compress hierarchical kernel matrices.

Finally, extending the framework of low-rank compression, [50] uses tensor-train compression to re-write $\mathscr{K}(X, Y)$ as a tensor with one dimension per coordinate, i.e., $\mathscr{K}(x_1, \ldots, x_d, y_1, \ldots, y_d)$ and then compress it using the tensor-train model.

### 5.1.3 Contribution

#### Overview of the method

In this work, we present a new algorithm that performs this low-rank factorization at a cost proportional to $\mathcal{O}(m+n)$. The main advantages of the method are as follows:

1. The complexity of our method is $\mathcal{O}(r(m+n))$ (in terms of kernel function $\mathscr{K}$ evaluations) where $r$ is the target rank.
2. The method is robust and accurate, irrespective of the distribution of points $x$ and $y$.
3. We can prove both convergence and numerical stability of the resulting algorithm.
4. The method is very simple and relies on well-optimized BLAS3 (GEMM) and LAPACK (RRQR, LU) kernels.

Consider the problem of approximating $\mathscr{K}(x,y)$ over the mesh $X \times Y$ with $X \in \mathcal{X}$ and $Y \in \mathcal{Y}$. Given the matrix $\mathscr{K}(X,Y)$, one possibility to build a low-rank factorization is to do a rank-revealing LU. This would lead to the selection of

$$X_{\text{piv}} \subset X, \qquad Y_{\text{piv}} \subset Y$$

called the "pivots", and the low-rank factorization would then be given by

$$\mathscr{K}(X,Y) \approx \mathscr{K}(X,Y_{\text{piv}})\mathscr{K}(X_{\text{piv}},Y_{\text{piv}})^{-1}\mathscr{K}(X_{\text{piv}},Y)$$

In practice, however, this method may become inefficient as it requires assembling the matrix $\mathscr{K}(X,Y)$ first.

In this chapter, we propose and analyze a new method to select the "pivots" *outside* of the sets $X$ and $Y$. The key advantage is that this selection is independent of the sets $X$ and $Y$, hence the reduced complexity. Let us consider the case where $\mathcal{X},\mathcal{Y} = [-1,1]^d$. We will keep this assumption throughout this work. Then, within $[-1,1]^d$, one can build tensor grids of Chebyshev points $\overline{X},\overline{Y}$ and associated integration weights $\overline{W}_X,\overline{W}_Y$ and then consider the matrix

$$K_w = \text{diag}(\overline{W}_X)^{1/2}\mathscr{K}(\overline{X},\overline{Y})\text{diag}(\overline{W}_Y)^{1/2}$$

Denote $r_0 = \min(|\overline{X}|, |\overline{Y}|)$. Based on interpolation properties, we will show that this matrix is closely related to the continuous kernel $\mathscr{K}(x, y)$. In particular, they share a similar spectrum. Then, we select the sets $\widehat{X} \subset \overline{X}$, $\widehat{Y} \subset \overline{Y}$ by performing strong rank-revealing QRs [80] over, respectively, $K_w^\top$ and $K_w$ (this is also called a CUR decomposition):

$$K_w P_y = Q_y R_y$$

$$K_w^\top P_x = Q_x R_x$$

and build $\widehat{X}$ by selecting the elements of $P_x$ associated to the largest rows of $R_x$ and similarly for $\widehat{Y}$ (if they differ in size, extend the smallest). We denote the rank of this factorization $r_1 = |\widehat{X}| = |\widehat{Y}|$, and in practice, we observe that $r_1 \approx r_{SVD}$, where $r_{SVD}$ is the rank the truncated SVD of $\mathscr{K}(X, Y)$ would provide. The resulting factorization is

$$\mathscr{K}(X, Y) \approx \mathscr{K}(X, \widehat{Y}) \mathscr{K}(\widehat{X}, \widehat{Y})^{-1} \mathscr{K}(\widehat{X}, Y) \tag{5.3}$$

Note that, in this process, at no point did we built any Lagrange basis function associated with $\overline{X}$ and $\overline{Y}$. We only evaluate the kernel $\mathscr{K}$ at $\overline{X} \times \overline{Y}$.

This method appears to be very efficient in selecting sets $\widehat{X}$ and $\widehat{Y}$ of minimum sizes. Indeed, instead, one could aim for a simple interpolation of $\mathscr{K}(x, y)$ over both $\mathcal{X}$ and $\mathcal{Y}$ separately. For instance, using the regular polynomial interpolation at Chebyshev nodes $\overline{X}$ and $\overline{Y}$, it would lead to a factorization of the form

$$\mathscr{K}(X, Y) \approx S(X, \overline{X}) \mathscr{K}(\overline{X}, \overline{Y}) T(Y, \overline{Y})^\top$$

In this expression, we collect the Lagrange basis functions (each one associated to a node of $\overline{X}$) evaluated at $X$ in the columns of $S(X, \overline{X})$ and similarly for $T(Y, \overline{Y})$. This provides a robust way of building a low-rank approximation. The rank $r_0 = \min(|\overline{X}|, |\overline{Y}|)$, however, is usually much larger than the true rank $r_{SVD}$ and than $r_1$ (given a tolerance). Note that even if those factorizations can always be further recompressed to a rank $\approx r_{SVD}$, they incur a high upfront cost because of the rank $r_0 \gg r_{SVD}$. See Section 5.1.3 for a discussion about this.

### Distinguishing features of the method

Since many methods resemble our approach, we point out its distinguishing features. The singular value decomposition (SVD) offers the optimal low-rank representation in the 2-norm. However, its complexity scales like $\mathcal{O}(n^3)$. In addition, we will show that the new approach is negligibly less accurate than the SVD in most cases.

The rank-revealing QR and LU factorization, and methods of random projections [88], have a reduced computational cost of $\mathcal{O}(n^2 r)$, but still scale quadratically with $n$.

Methods like ACA [18], the rank-revealing LU factorization with rook pivoting [75], and techniques that randomly sample from columns and rows of the matrix scale like $\mathcal{O}(nr)$, but they provide no accuracy guarantees. In fact, counterexamples can be found where these methods fail. In contrast, our approach relies on Chebyshev nodes, which offers strong stability and accuracy guarantees. The fact that new interpolation points, $\overline{X}$ and $\overline{Y}$, are introduced (the Chebyshev nodes) in addition to the existing points in $X$ and $Y$ is one of the key elements.

Analytical methods are available, like the fast multipole method, etc., but they are limited to specific kernels. Other techniques, which are more general, like Taylor expansion and Chebyshev interpolation [65], have strong accuracy guarantees and are as general as the method presented. However, their cost is much greater; in fact, the difference in efficiency is measured directly by the reduction from $r_0$ to $r_1$ in our approach.

### Low-rank approximation based on SVD and interpolation

Consider the kernel function $\mathscr{K}$ and its singular value decomposition [127, theorem VI.17]:

**Theorem 5.1** (Singular Value Decomposition)**.** *Suppose $\mathscr{K} : [-1,1]^d \times [-1,1]^d$ is square integrable. Then there exist two sequences of orthogonal functions $\{u_i\}_{i=1}^{\infty}$ and $\{v_i\}_{i=1}^{d}$ and a non-increasing sequence of non-negative real number $\{s_i\}_{i=1}^{\infty}$ such that*

$$\mathscr{K}(x,y) = \sum_{s=1}^{\infty} \sigma_s u_s(x) v_s(y) \tag{5.4}$$

As one can see, under relatively mild assumptions, any kernel function can be expanded into a singular value decomposition. Hence, from any kernel function expansion, we find a low-rank decomposition for the matrix $\mathscr{K}(X,Y)$ (which is *not* the same as the matrix

SVD):

$$\mathscr{K}(X,Y) \approx \sum_{s=1}^{r} u_s(X)\sigma_s v_s(Y) = \begin{bmatrix} u_1(X) & \cdots & u_r(X) \end{bmatrix} \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_r \end{bmatrix} \begin{bmatrix} v_1^\top(Y) \\ \vdots \\ v_r^\top(Y) \end{bmatrix} \tag{5.5}$$

where the sequence $\{s_i\}_{i=1}^{\infty}$ was truncated at an appropriate index $r$. As a general rule of thumb, the smoother the function $\mathscr{K}(x,y)$, the faster the decay of the $\sigma_s$'s and the lower the rank.

If we use a polynomial interpolation method with Chebyshev nodes, we get a similar form:

$$\mathscr{K}(X,Y) \approx S(X,\overline{X})\ \mathscr{K}(\overline{X},\overline{Y})\ T(Y,\overline{Y})^\top \tag{5.6}$$

The interpolation functions $S(x,\overline{X})$ and $T(y,\overline{Y})$ have strong accuracy guarantees, but the number of terms required in the expansion is $r_0 \gg r \approx r_1$. This is because Chebyshev polynomials are designed for a broad class of functions. In contrast, the SVD uses basis functions $u_s$ and $v_s$ that are optimal for the chosen $\mathscr{K}$.

**Optimal interpolation methods**

We will now discuss a more general problem, then derive our algorithm as a special case. Let's start with understanding the optimality of the Chebyshev interpolation. With Chebyshev interpolation, $S(x,\overline{X})$ and $T(y,\overline{Y})$ are polynomials. This is often considered one of the best (most stable and accurate) ways to interpolate smooth functions. We know that for general polynomial interpolants we have:

$$f(x) - S(x,\overline{X})f(\overline{X}) = \frac{f^{(m)}(\xi)}{m!}\prod_{j=1}^{m}(x-\overline{X}_j) \tag{5.7}$$

If we assume that the derivative $f^{(m)}(\xi)$ is bounded, we can focus on finding interpolation points such that

$$\prod_{j=1}^{m}(x-\overline{X}_j) = x^m - r_{\overline{X}}(x)$$

is minimal, where $r_{\overline{X}}(x)$ is a degree $m-1$ polynomial. Since we are free to vary the interpolation points $\overline{X}$, then we have $m$ parameters (the location of the interpolation points)

and $m$ coefficients in $r_{\overline{X}}$. By varying the location of the interpolation points, we can recover any polynomial $r_{\overline{X}}$. Chebyshev points are known to solve this problem optimally. That is, they lead to an $r_{\overline{X}}$ such that $\max_x |x^m - r_{\overline{X}}(x)|$ is minimal.

Chebyshev polynomials are a very powerful tool because of their generality and simplicity of use. Despite this, we will see that this can be improved upon with relatively minimal effort. Let's consider the construction of interpolation formulas for a family of functions $\mathscr{K}(x, \lambda)$, where $\lambda$ is a parameter. We would like to use the SVD, but, because of its high computational cost, we rely on the cheaper rank-revealing QR factorization (RRQR, a QR algorithm with column pivoting). RRQR solves the following optimization problem:

$$\min_{\{\lambda_s, v_s\}} \max_{\lambda} \left\| \mathscr{K}(x, \lambda) - \sum_{s=1}^{m} \mathscr{K}(x, \lambda_s) v_s(\lambda) \right\|_2, \qquad v_s(\lambda_t) = \delta_{st}$$

where the 2-norm is computed over $x$—in addition RRQR produces an orthogonal basis for $\{\mathscr{K}(x, \lambda_s)\}_s$ but this is not needed in the current discussion. The vector space $\text{span}\{\mathscr{K}(x, \lambda_s)\}_{s=1,\dots,m}$ is close to $\text{span}\{u_s\}_{s=1,\dots,m}$ [see (5.4)], and the error can be bounded by $\sigma_{m+1}$.

Define $\widehat{\Lambda} = \{\lambda_1, \dots, \lambda_m\}$. From there, we identify a set of $m$ interpolation nodes $\widehat{X}$ such that the square matrix

$$\mathscr{K}(\widehat{X}, \widehat{\Lambda}) := \begin{bmatrix} \mathscr{K}(\widehat{X}, \lambda_1) & \cdots & \mathscr{K}(\widehat{X}, \lambda_m) \end{bmatrix}$$

is as well-conditioned as possible. We now define our interpolation operator as

$$\widehat{S}(x, \widehat{X}) = \mathscr{K}(x, \widehat{\Lambda}) \mathscr{K}(\widehat{X}, \widehat{\Lambda})^{-1}$$

By design, this operator is exact on $\mathscr{K}(x, \lambda_s)$:

$$\widehat{S}(x, \widehat{X}) \mathscr{K}(\widehat{X}, \lambda_s) = \mathscr{K}(x, \lambda_s)$$

It is also very accurate for $\mathscr{K}(x, \lambda)$ since

$$\widehat{S}(x, \widehat{X}) \mathscr{K}(\widehat{X}, \lambda) \approx \sum_{s=1}^{m} \widehat{S}(x, \widehat{X}) \mathscr{K}(\widehat{X}, \lambda_s) v_s(\lambda) = \sum_{s=1}^{m} \mathscr{K}(x, \lambda_s) v_s(\lambda) \approx \mathscr{K}(x, \lambda)$$

With Chebyshev interpolation, $S(x, \overline{X})$ is instead defined using order $m - 1$ polynomial

functions.

A special case that illustrates the difference between SI and Chebyshev, is with rank-1 kernels:

$$\mathscr{K}(x,\lambda) = u(x)v(\lambda)$$

In this case, we can pick any $x_1$ and $\lambda_1$ such that $\mathscr{K}(x_1,\lambda_1) \neq 0$, and define $\widehat{X} = \{x_1\}$ and

$$\widehat{S}(x,\widehat{X}) = \mathscr{K}(x,\lambda_1)\mathscr{K}(x_1,\lambda_1)^{-1}$$

$$\widehat{S}(x,\widehat{X})\mathscr{K}(\widehat{X},\lambda) = u(x)v(\lambda_1)\frac{1}{u(x_1)v(\lambda_1)}u(x_1)v(\lambda) = u(x)v(\lambda)$$

SI is exact using a single interpolation point $x_1$. An interpolation using Chebyshev polynomials would lead to errors, for any expansion order (unless $u$ is fortuitously a polynomial).

So, one of the key differences between SI and Chebyshev interpolation is that SI uses, as a basis for its interpolation, *a set of nearly optimal functions that approximate the left singular functions of* $\mathscr{K}$, rather than generic polynomial functions.

### Proposed method

In this work, we use the framework from Section 5.1.3 to build an interpolation operator for the class of functions $\mathscr{K}(x,y)$, which we view as a family of functions of $x$ parameterized by $y$ (and vice versa to obtain a symmetric interpolation method). The approximation (5.3) can be rewritten

$$\mathscr{K}(X,Y) \approx \left[\mathscr{K}(X,\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\right]\,\mathscr{K}(\widehat{X},\widehat{Y})\,\left[\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\mathscr{K}(\widehat{X},Y)\right]$$

and by comparing with (5.6), we recognize the interpolation operators:

$$\widehat{S}(x,\widehat{X}) = \mathscr{K}(x,\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}, \qquad \widehat{T}(y,\widehat{Y}) = \mathscr{K}(\widehat{X},y)^{\top}\mathscr{K}(\widehat{X},\widehat{Y})^{-T}$$

These interpolation operators are nearly optimal; because of the way these operators are constructed we call the method "Skeletonized Interpolation." The sets $\widehat{X}$ and $\widehat{Y}$ are the minimal sets such that if we sample $\mathscr{K}$ at these points we can interpolate $\mathscr{K}$ at any other point with accuracy $\varepsilon$. In particular, $\widehat{X}$ and $\widehat{Y}$ are much smaller than their Chebyshev-interpolant counterparts $\overline{X}$ and $\overline{Y}$ and their size, $r_1$, is very close to $r$ in (5.5). The approach we are proposing produces nearly-optimal interpolation functions for our kernel,

instead of generic polynomial functions.

Note that none of the previous discussions explains why the proposed scheme is stable; the inverse $\mathscr{K}(\widehat{X},\widehat{\Lambda})^{-1}$ as well as $\mathscr{K}(\widehat{X},\widehat{Y})^{-1}$ in (5.3) could become troublesome numerically. We will explain in detail in Section 5.3 why this is not an issue numerically, and we explore the connection with interpolation in more detail in Section 5.4.

## 5.2 Skeletonized Interpolation

### 5.2.1 The algorithm

Algorithm 5.1 provides the high-level version of the algorithm. It consists of 3 steps:

- Build grids $\overline{X}$ and $\overline{Y}$, tensor grids of Chebyshev nodes. Over $[-1,1]$ in 1D, the $\overline{m}$ Chebyshev nodes of the first kind are defined as

$$\bar{x}_k = \cos\left(\frac{2k-1}{2\overline{m}}\pi\right) \qquad k = 1,\ldots,\overline{m}$$

  In higher dimensions, they are defined as the tensor product of one-dimensional grids. The number of points in every dimension should be such that

$$\sum_{k=1}^{\overline{m}}\sum_{l=1}^{\overline{n}} S_k(x)\mathscr{K}(\bar{x}_k,\bar{y}_l)T_l(y) = S(x,\overline{X})\mathscr{K}(\overline{X},\overline{Y})T(y,\overline{Y})^\top$$

  provides an $\delta$ uniform approximation over $[-1,1]^d\times[-1,1]^d$ of $\mathscr{K}(x,y)$, i.e.,

$$|S(x,\overline{X})\mathscr{K}(\overline{X},\overline{Y})T(y,\overline{Y})^\top - \mathscr{K}(x,y)| \le \delta$$

  for all $(x,y)\in[-1,1]^d\times[-1,1]^d$. Denote $r_0 = \min(|\overline{X}|,|\overline{Y}|)$.

- Recompress the grid by performing a strong rank-revealing QR factorization [80] of

$$\mathrm{diag}(\overline{W}_X)^{1/2}\mathscr{K}(\overline{X},\overline{Y})\mathrm{diag}(\overline{W}_Y)^{1/2} \tag{5.8}$$

  and its transpose, up to accuracy $\varepsilon$ (i.e., the 2-norm error of the approximation is at most $\varepsilon$). This factorization is also named CUR decomposition [111, 45]. While our error estimates only hold for strong rank-revealing QR factorizations, in practice, a simple column-pivoted QR factorization based on choosing columns with large norms

works as well. In the case of Chebyshev nodes of the first kind in 1D over $[-1,1]$ the integration weights are given by

$$w_k = \frac{\pi}{\overline{m}}\sqrt{1-\bar{x}_k^2} = \frac{\pi}{\overline{m}}\sin\left(\frac{2k-1}{2\overline{m}}\pi\right)$$

The weights in $d$ dimensions are the products of the corresponding weights in 1D, and the $\text{diag}(\overline{W}_X)$ and $\text{diag}(\overline{W}_Y)$ matrices are simply the diagonal matrices of the integration weights. Denote

$$r_1 = |\widehat{X}| = |\widehat{Y}|$$

In case the sets $\widehat{X}$ and $\widehat{Y}$ output by those RRQR's are of slightly different sizes (which we rarely noticed in our experiments), extend the smallest to have the same size as the largest.

- Given $\widehat{X}$ and $\widehat{Y}$, the low-rank approximation is given by

$$\mathscr{K}(X,\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\mathscr{K}(\widehat{X},Y)$$

of rank $r_1 \approx r_{SVD}$.

Algorithm 5.1 summarizes the algorithm.

**Algorithm 5.1** Skeletonized Interpolation

**procedure** SKELETONIZED INTERPOLATION($\mathscr{K} : [-1,1]^d \times [-1,1]^d \to \mathbb{R}$, $X$, $Y$, $\varepsilon$, $\delta$)
  **require** $\varepsilon < \delta$
  Build $\overline{X}$ and $\overline{Y}$, sets of Chebyshev nodes over $[-1,1]^d$ that interpolate $\mathscr{K}$ with error $\delta$ uniformly
  Build $K_w$ as $K_w = \text{diag}(\overline{W}_X)^{1/2}\mathscr{K}(\overline{X},\overline{Y})\text{diag}(\overline{W}_Y)^{1/2}$.
  Extract $\widehat{Y} \subseteq \overline{Y}$ by performing a strong RRQR over $K_w$ with 2-norm tolerance $\varepsilon$.
  Extract $\widehat{X} \subseteq \overline{X}$ by performing a strong RRQR over $K_w^\top$ with 2-norm tolerance $\varepsilon$.
  If the sets have different size, extends the smallest to the size of the largest.
  **return** $\mathscr{K}(X,Y) \approx \mathscr{K}(X,\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\mathscr{K}(\widehat{X},Y)$.
**end procedure**

### 5.2.2 Theoretical Convergence

#### Overview

In this section, we prove that the error made during the RRQR is not too much amplified when evaluating the interpolant. We first recall that

1. From interpolation properties,

$$\mathscr{K}(x,y) = S(x,\overline{X})\mathscr{K}(\overline{X},\overline{Y})T(y,\overline{Y})^\top + E_{\text{INT}}(x,y)$$

   where $T$ and $S$ are small matrices (i.e., bounded by logarithmic factors in $r_0$) and $|E_{\text{INT}}(x,y)| \leq \delta$ (by construction).

2. From the strong RRQR properties,

$$K_w = \begin{bmatrix} I \\ \widehat{S} \end{bmatrix} \widehat{K}_w \begin{bmatrix} I & \widehat{T}^\top \end{bmatrix} + E_{\text{QR}}$$

   where $\widehat{K}_w$ has a spectrum similar to that of $K_w$ (up to a small polynomial), $\widehat{S}$ and $\widehat{T}$ are bounded by a small polynomial, and where $\varepsilon := \|E_{\text{QR}}\|_2$ (by construction).

Then, by combining those two facts and assuming $\delta < \varepsilon$, one can show

1. First, that the interpolation operators are bounded,

$$\|\mathscr{K}(x,\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\|_2 = \mathcal{O}(q(r_0,r_1)) \tag{5.9}$$

   where $q$ is a fixed degree polynomial.

2. Second, that the error $\varepsilon$ made in the RRQR is not too much amplified, i.e.,

$$|\mathscr{K}(x,y) - \mathscr{K}(x,\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\mathscr{K}(\widehat{X},y)| \leq \varepsilon r(r_0,r_1) \tag{5.10}$$

   where $r$ is another fixed degree polynomial.

Finally, if one assumes that $\sigma_i(K_w)$ decays exponentially fast, so does $\varepsilon$ and the resulting approximation in (5.10) converges as $r_0 \to \infty$.

We now present the main lemmas leading to the above results.

### Interpolation-related results

We first consider the interpolation itself. Consider $\overline{X}$ and $\overline{Y}$, constructed such as

$$\mathscr{K}(x,y) = S(x,\overline{X})\mathscr{K}(\overline{X},\overline{Y})T(y,\overline{Y})^\top + E_{\text{INT}}(x,y)$$

**Lemma 5.1** (Interpolation at Chebyshev Nodes)**.** *$\forall x \in \mathcal{X} \subset \mathbb{R}^d$ and $\overline{X}$ tensor grids of Chebyshev nodes of the first kind, let $\overline{m} = |\overline{X}|$. Then*

$$\|S(x,\overline{X})\|_2 = \mathcal{O}\Big(\log(\overline{m})^d\Big)$$

*In addition, the weights, collected in the weight matrix $diag(\overline{W}_X)$ are such that*

$$\|\,diag(\overline{W}_X)^{1/2}\|_2 \leq \frac{\pi^{d/2}}{\sqrt{\overline{m}}} = \mathcal{O}\left(\frac{1}{\sqrt{\overline{m}}}\right)$$
$$\|\,diag(\overline{W}_X)^{-1/2}\|_2 \leq \frac{\overline{m}}{\pi^{d/2}} = \mathcal{O}(\overline{m})$$

*Proof.* This bound on the Lagrange basis is a classical result related to the growth of the Lebesgue constant in polynomial interpolation. For $\overline{m}$ Chebyshev nodes of the first kind on $[-1,1]$ and the associated Lagrange basis functions $\ell_1,\ldots,\ell_{\overline{m}}$ we have the following result [96, equation 13]

$$\max_{x\in[-1,1]} \sum_{i=1}^{\overline{m}} |\ell_i(x)| \leq \frac{2}{\pi}\log(\overline{m}+1) + 0.974 = \mathcal{O}(\log(\overline{m}))$$

This implies that in one dimension,

$$\|S(x,\overline{X})\|_2 \leq \|S(x,\overline{X})\|_1 = \mathcal{O}(\log\overline{m})$$

Going from one to $d$ dimensions can be done using Kronecker products. Indeed, for $x \in \mathbb{R}^d$,

$$S(x,\overline{X}) = S(x_1,\overline{X}_1)\otimes\cdots\otimes S(x_d,\overline{X}_d)$$

where $x = (x_1,\ldots,x_d)$ and $\overline{X}_1,\ldots,\overline{X}_d$ are the one-dimensional Chebyshev nodes. Since

for all $a \in \mathbb{R}^m, b \in \mathbb{R}^n$, $\|a \otimes b\|_2 = \sqrt{\sum_{i,j}(a_i b_j)^2} = \|ab^\top\|_F = \|a\|_2\|b\|_2$, it follows that

$$\|S(x, \overline{X})\|_2 = \|S(x_1, \overline{X}_1) \otimes \cdots \otimes S(x_d, \overline{X}_d)\|_2 = \prod_{i=1}^{d} \|S(x_i, \overline{X}_i)\|_2 = \prod_{i=1}^{d} \mathcal{O}(\log \overline{m}_i)$$

This implies, using a fairly loose bound,

$$\|S(x, \overline{X})\|_2 = \mathcal{O}\Big(\log(|\overline{X}|)^d\Big)$$

The same argument can be done for $T(y, \overline{Y})$.

In 1D, the weights are

$$w_k = \frac{\pi}{\overline{m}} \sin\left(\frac{2k-1}{2\overline{m}}\pi\right)$$

for $k = 1, \ldots, \overline{m}$. Clearly, $w_k > 0$ and $w_k < \frac{\pi}{\overline{m}}$. Also, the minimum being reached at $k = 1$ or $k = \overline{m}$,

$$w_k \geq \frac{\pi}{\overline{m}} \sin\left(\frac{\pi}{2\overline{m}}\right) > \frac{\pi}{\overline{m}} \frac{2\pi}{2\pi\overline{m}} = \frac{\pi}{\overline{m}^2}$$

Since the nodes in $d$ dimensions are products of the nodes in 1D, it follows that

$$\|\mathrm{diag}(\overline{W}_X)\|_2 \leq \frac{\pi^d}{\overline{m}}$$
$$\|\mathrm{diag}(\overline{W}_X)^{-1}\|_2 \leq \frac{\overline{m}^2}{\pi^d}$$

The result follows. □

### Skeletonization results

We now consider the skeletonization step of the algorithm performed through the two successive rank-revealing QR factorizations.

**Rank-Revealing QR factorizations** Let us first recall what a rank-revealing QR factorization is. Given a matrix $A \in \mathbb{R}^{m \times n}$, one can compute a rank-revealing QR factorization [75] of the form

$$A\Pi = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} \\ & R_{22} \end{bmatrix}$$

where $\Pi$ is a permutation matrix, $Q$ an orthogonal matrix, and $R$ a triangular matrix. Both $R$ and $Q$ are partitioned so that $Q_1 \in \mathbb{R}^{m\times k}$ and $R_{11} \in \mathbb{R}^{k\times k}$. If $\|R_{22}\| \approx \varepsilon$, this factorization indicates that $A$ has an $\varepsilon$-rank of $k$. The converse, however, is not necessarily true [75] in general.

From there, one can also write

$$A\Pi = Q_1 R_{11} \begin{bmatrix} I & R_{11}^{-1}R_{12} \end{bmatrix} + E = A_1 \begin{bmatrix} I & T \end{bmatrix} + E$$

where $T$ is the *interpolation* operator, $A_1$ a set of $k$ columns of $A$, and $E$ the approximation error. This approximation can be achieved by a simple column-pivoted QR algorithm [75]. This algorithm, however, is not guaranteed to always work (i.e., even if $A$ has rapidly decaying singular values, this rank-revealing factorization may fail to exhibit it).

A *strong* rank-revealing QR, however, has more properties. It has been proven [80, 45] that one can compute in $\mathcal{O}\big(mn^2\big)$ a rank-revealing QR factorization that guarantees

$$\sigma_i(A_1) \geq \frac{\sigma_i(A)}{q_1(n,k)},\ \sigma_j(E) \leq \sigma_{k+j}(A) q_1(n,k) \text{ and } \|T\|_F \leq q_2(n,k) \tag{5.11}$$

where $q_1$ and $q_2$ are two small polynomials (with fixed constants and degrees). The existence of this factorization is a crucial part of our argument. Using the interlacing property of singular values [75], this implies that we now have both lower and upper bounds on the singular values of $A_1$

$$\frac{\sigma_i(A)}{q_1(n,k)} \leq \sigma_i(A_1) \leq \sigma_i(A) \tag{5.12}$$

From (5.11) we can directly relate the error $E$ and $\sigma_{k+1}$ from

$$\|E\|_2 = \sigma_1(E) \leq \sigma_{k+1}(A) q_1(n,k) \tag{5.13}$$

Finally, given a matrix $A$, one can apply the above result to both its rows and columns, leading to a factorization

$$\Pi_r^\top A \Pi_c = \begin{bmatrix} I \\ T_r \end{bmatrix} A_{rc} \begin{bmatrix} I & T_c \end{bmatrix} + E$$

with the same properties as detailed above (see also [45, theorem 3 and remark 5]).

**Skeletonized Interpolation** We can now apply this result to the $K_w$ and $\widehat{K}_w$ matrices.

**Lemma 5.2** (CUR Decomposition of $K_w$)**.** *The partition $\overline{X} = \widehat{X} \cup \check{X}$, $\overline{Y} = \widehat{Y} \cup \check{Y}$ of Algorithm 5.1, with $r_0 = |\overline{X}| = |\overline{Y}|$ and $r_1 = |\widehat{X}| = |\widehat{Y}|$, is such that there exist $\check{S}$, $\check{T}$, $E_{QR}(\overline{X}, \overline{Y})$ matrices (with $\varepsilon := \|E_{QR}(\overline{X}, \overline{Y})\|_2$) and polynomials $p_1(r_0, r_1), p_2(r_0, r_1)$ (with fixed degrees) such that*

$$K_w = \begin{bmatrix} I \\ \check{S} \end{bmatrix} \widehat{K}_w \begin{bmatrix} I & \check{T}^\top \end{bmatrix} + E_{QR}(\overline{X}, \overline{Y})$$

*and where*

$$\begin{aligned} \varepsilon &\leq p_1(r_0, r_1)\sigma_{r_1+1}(K_w) \\ \|\check{S}\|_2 &\leq p_2(r_0, r_1) \\ \|\check{T}\|_2 &\leq p_2(r_0, r_1) \end{aligned}$$

*Finally, we have*

$$\|\widehat{K}_w^{-1}\|_2 \leq \frac{p_1(r_0, r_1)^2}{\varepsilon}$$

*Proof.* The first three results are direct applications of [45, theorem 3 and remark 5] as explained in the previous paragraph (using the fact that the 2-norm and Frobenius norm are equivalent). The last result follows from the properties of the strong rank-revealing QR:

$$\|\widehat{K}_w^{-1}\|_2 = \frac{1}{\sigma_{r_1}(\widehat{K}_w)} \leq \frac{p_1(r_0, r_1)}{\sigma_{r_1}(K_w)} \leq \frac{p_1(r_0, r_1)}{\sigma_{r_1+1}(K_w)} \leq \frac{p_1(r_0, r_1)^2}{\varepsilon}$$

The first inequality follows from $\sigma_{r_1}(K_w) \leq \sigma_{r_1}(\widehat{K}_w)p_1(r_0, r_1)$ (5.12), the second from $\sigma_{r_1}(K_w) \geq \sigma_{r_1+1}(K_w)$ (by definition of singular values) and the last from $\sigma_{r_1+1}(K_w)^{-1} \leq p_1(r_0, r_1)\varepsilon^{-1}$ (5.13).

□

This leads to the first result

**Theorem 5.2.** *If $\widehat{X}$ and $\widehat{Y}$ are constructed following Algorithm 5.1, then there exists a*

*fixed degree polynomial* $q(r_0, r_1)$ *such that for any* $x \in \mathcal{X}, y \in \mathcal{Y}$,

$$\|\mathscr{K}(x, \widehat{Y})\mathscr{K}(\widehat{X}, \widehat{Y})^{-1}\|_2 = \mathcal{O}(q(r_0, r_1))$$
$$\|\mathscr{K}(\widehat{X}, \widehat{Y})^{-1}\mathscr{K}(\widehat{X}, y)\|_2 = \mathcal{O}(q(r_0, r_1))$$

*Proof.* We show the result for the second equation. This requires using, consecutively, the interpolation result and the CUR decomposition one. First, one can write from Lemma 5.1 and the interpolation,

$$\mathscr{K}(\widehat{X}, \widehat{Y})^{-1}\mathscr{K}(\widehat{X}, y) = \mathscr{K}(\widehat{X}, \widehat{Y})^{-1}\left[\mathscr{K}(\widehat{X}, \overline{Y})T(y, \overline{Y})^\top + E_{\text{INT}}(\widehat{X}, y)\right]$$

Then, introducing the weight matrices and applying Lemma 5.2 on the interpolation matrix,

$$\begin{aligned}
&\mathscr{K}(\widehat{X}, \overline{Y}) = \\
&= \text{diag}(\widehat{W}_X)^{-1/2}\text{diag}(\widehat{W}_X)^{1/2}\mathscr{K}(\widehat{X}, \overline{Y})\text{diag}(\overline{W}_Y)^{1/2}\text{diag}(\overline{W}_Y)^{-1/2} \\
&= \text{diag}(\widehat{W}_X)^{-1/2}\Big\{\text{diag}(\widehat{W}_X)^{1/2}\mathscr{K}(\widehat{X}, \widehat{Y})\text{diag}(\widehat{W}_Y)^{1/2}\begin{bmatrix} I & \breve{T}^\top \end{bmatrix} \\
&\qquad + E_{\text{QR}}(\widehat{X}, \overline{Y})\Big\}\text{diag}(\overline{W}_Y)^{-1/2}
\end{aligned}$$

Finally, combining and distributing all the factors gives us

$$\begin{aligned}
\mathscr{K}(\widehat{X}, \widehat{Y})^{-1}\mathscr{K}(\widehat{X}, y) &= \text{diag}(\widehat{W}_Y)^{1/2}\begin{bmatrix} I & \breve{T}^\top \end{bmatrix}\text{diag}(\overline{W}_Y)^{-1/2}T(y, \overline{Y})^\top \\
&\quad + \mathscr{K}(\widehat{X}, \widehat{Y})^{-1}\text{diag}(\widehat{W}_X)^{-1/2}E_{\text{QR}}(\widehat{X}, \overline{Y})\text{diag}(\overline{W}_Y)^{-1/2}T(y, \overline{Y})^\top \\
&\quad + \mathscr{K}(\widehat{X}, \widehat{Y})^{-1}E_{\text{INT}}(\widehat{X}, y)
\end{aligned}$$

Here, we can bound all terms:

- For the first term, Lemma 5.1 and Lemma 5.2 show that the expression is bounded by a fixed degree polynomial;
- For the second term, use the fact that, since $\text{diag}(\widehat{W}_X)^{1/2}$ and $\text{diag}(\overline{W}_Y)^{-1/2}$ are bounded by fixed degree polynomials,

  $$\|\widehat{K}_w^{-1}\|_2 \leq \frac{p_1^2(r_0, r_1)}{\varepsilon} \Rightarrow \|\mathscr{K}(\widehat{X}, \widehat{Y})^{-1}\|_2 \leq \frac{p'(r_0, r_1)}{\varepsilon}$$

  for some fixed degree polynomial $p'$. Hence, since $\|E_{QR}(\overline{X}, \overline{Y})\|_2 = \varepsilon$, the product is

again bounded by a polynomial since the $\varepsilon$ cancel out ;

- The last term can be bounded in a similar way using

$$E_{\mathrm{INT}}(x,y) \leq \delta < \varepsilon$$

We conclude that there exists a fixed degree polynomial $q$ such that

$$\|\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\mathscr{K}(\widehat{X},y)\|_2 = \mathcal{O}(q(r_0,r_1))$$

The proof is similar in $x$. □

The key ingredient is simply that $\|\widehat{K}_w^{-1}\|_2 \leq p_1(r_0,r_1)^2\varepsilon^{-1}$ from the RRQR properties; hence $\widehat{K}_w$ is ill-conditioned, but not arbitrarily. Its condition number grows like $\varepsilon^{-1}$. Then, when multiplied by quantities like $\varepsilon$ or $\delta < \varepsilon$, the factors cancel out and the resulting product can be bounded.

**Link between the node matrix and the continuous SVD**

In this section, we link the continuous SVD and the spectrum (singular values) of the matrix $\mathrm{diag}(\overline{W}_X)^{1/2}K_w\mathrm{diag}(\overline{W}_Y)^{1/2}$. This justifies the use of the weights.

Let us consider the case where interpolation is performed at *Gauss-Legendre* (*not* Chebyshev) nodes $\overline{X},\overline{Y}$ with the corresponding integration weights $\overline{W}_X,\overline{W}_Y$. (A more complete explanation can be found in [163].)

Take the classical discrete SVD of $K_w$,

$$K_w = \overline{U}\,\overline{\Sigma}\,\overline{V}^{\top}$$

We then have

$$\mathscr{K}(x,y) = \underbrace{S_w(x,\overline{X})\overline{U}\,\overline{\Sigma}\,\overline{V}^{\top}T_w(y,\overline{Y})^{\top}}_{=\overline{\mathscr{K}}(x,y)} + E_{\mathrm{INT}}(x,y)$$

Then, denote the sets of new basis functions

$$\overline{u}(x) = S_w(x,\overline{X})\overline{U} \qquad \overline{v}(y) = T_w(y,\overline{Y})\overline{V}$$

The key is to note that those functions are orthonormal. Namely, for $\overline{u}$,

$$\begin{aligned}\int_{\mathcal{X}} \overline{u}_i(x)\overline{u}_j(x)\mathrm{d}x &= \sum_{k=1}^{r_0} \overline{w}_k \overline{u}_i(\overline{x}_k)\overline{u}_j(\overline{x}_k) \\ &= \sum_{k=1}^{r_0} \overline{w}_k \left(\sum_{l=1}^{r_0} \overline{w}_l^{-1/2} S_l(\overline{x}_k)\overline{U}_{li}\right)\left(\sum_{l=1}^{r_0} \overline{w}_l^{-1/2} S_l(\overline{x}_k)\overline{U}_{lj}\right) \\ &= \sum_{k=1}^{r_0} \overline{w}_k \left(\sum_{l=1}^{r_0} \delta_{kl}\overline{w}_l^{-1/2}\overline{U}_{li}\right)\left(\sum_{l=1}^{r_0} \delta_{kl}\overline{w}_l^{-1/2}\overline{U}_{lj}\right) \\ &= \sum_{k=1}^{r_0} \overline{w}_k \overline{w}_k^{-1/2}\overline{U}_{ki}\overline{w}_k^{-1/2}\overline{U}_{kj} = \sum_{k=1}^{r_0} \overline{U}_{ki}\overline{U}_{kj} = \delta_{ij}\end{aligned}$$

The same result holds for $\overline{v}$. This follows from the fact that a Gauss-Legendre quadrature rule with $n$ points can exactly integrate polynomials up to degree $2n-1$. This shows that we are implicitly building a factorization

$$\mathscr{K}(x,y) = \sum_{s=1}^{\infty} \sigma_s u_s(x) v_s(y) = \underbrace{\sum_{s=1}^{r_0} \sigma_s(K_w)\overline{u}_s(x)\overline{v}_s(y)}_{=\overline{\mathscr{K}}(x,y)} + E_{\mathrm{INT}}(x,y) \tag{5.14}$$

where the approximation error is bounded by the interpolation error $E_{\mathrm{INT}}$ and where the sets of basis functions are orthogonal.

Assume now that the kernel $\mathscr{K}$ is square-integrable over $[-1,1]^d \times [-1,1]^d$. This is called a Hilbert-Schmidt kernel [129, Lemma 8.20]. This implies that the associated linear operator is compact [129, Theorem 8.83]. $\overline{\mathscr{K}}$ is compact as well since it is finite rank [129, Theorem 8.80]. Given the fact that $|E_{\mathrm{INT}}(x,y)| \leq \delta$ for all $x, y$, $\|\mathscr{K} - \overline{\mathscr{K}}\|_{L_2} \leq C\delta$ for some $C$ and hence, by compactness of both operators [81, Corollary 2.2.14],

$$|\sigma_i - \sigma_i(\overline{\mathscr{K}})| \leq C\delta$$

for some $C > 0$. Then, from the above discussion, we clearly have $\sigma_i(K_w) = \sigma_i(\overline{\mathscr{K}}) + \mathcal{O}(\delta)$ and hence

$$\sigma_i(K_w) = \sigma_i + \mathcal{O}(\delta)$$

This result only formally holds for Gauss-Legendre nodes and weights. However, this motivates the use of integration weights in the case of Chebyshev as well.

### Convergence of the Skeletonized interpolation

We now present the main result of this chapter:

**Theorem 5.3** (Convergence of Skeletonized Interpolation)**.** *If $\widehat{X}$ and $\widehat{Y}$ are constructed following Algorithm 5.1, then there exists a fixed degree polynomial $r(r_0, r_1)$ such that for any $x \in \mathcal{X}$ and $y \in \mathcal{Y}$,*

$$|\mathscr{K}(x,y) - \mathscr{K}(x,\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\mathscr{K}(\widehat{X},y)| \leq \varepsilon\, r(r_0,r_1)$$

The key here is that the error incurred during the CUR decomposition, $\varepsilon$, is amplified by at most a polynomial of $r_0$ and $r_1$. Hence, Theorem 5.3 indicates that if the spectrum decays fast enough (i.e., if $\varepsilon \to 0$ when $r_0, r_1 \to \infty$ faster than $r(r_0, r_1)$ grows), the proposed approximation should converge to the true value of $\mathscr{K}(x, y)$.

*Proof.* Combining interpolation and CUR decomposition results one can write

$$\begin{aligned}
&\mathscr{K}(x,y) = S(x,\overline{X})\mathscr{K}(\overline{X},\overline{Y})T(y,\overline{Y})^\top + E_{\text{INT}}(x,y)\\
&= S_w(x,\overline{X})\mathscr{K}_w(\overline{X},\overline{Y})T_w(y,\overline{Y})^\top + E_{\text{INT}}(x,y)\\
&= S_w(x,\overline{X})\left[\begin{bmatrix} I \\ \breve{S}\end{bmatrix}\mathscr{K}_w(\widehat{X},\widehat{Y})\begin{bmatrix} I & \breve{T}^\top\end{bmatrix} + E_{\text{QR}}(\overline{X},\overline{Y})\right]T_w(y,\overline{Y})^\top + E_{\text{INT}}(x,y)\\
&= S_w(x,\overline{X})\begin{bmatrix}\mathscr{K}_w(\widehat{X},\widehat{Y}) \\ \breve{S}\mathscr{K}_w(\widehat{X},\widehat{Y})\end{bmatrix}\mathscr{K}_w(\widehat{X},\widehat{Y})^{-1}\begin{bmatrix}\mathscr{K}_w(\widehat{X},\widehat{Y}) & \mathscr{K}_w(\widehat{X},\widehat{Y})\breve{T}^\top\end{bmatrix}T_w(y,\overline{Y})^\top\\
&\quad + S_w(x,\overline{X})E_{\text{QR}}(\overline{X},\overline{Y})T_w(y,\overline{Y})^\top + E_{\text{INT}}(x,y)\\
&= S_w(x,\overline{X})\left[\mathscr{K}_w(\overline{X},\widehat{Y}) + E_{\text{QR}}(\overline{X},\widehat{Y})\right]\mathscr{K}_w(\widehat{X},\widehat{Y})^{-1}\left[\mathscr{K}_w(\widehat{X},\overline{Y}) + E_{\text{QR}}(\widehat{X},\overline{Y})\right]\\
&\quad \times T_w(y,\overline{Y})^\top + S_w(x,\overline{X})E_{\text{QR}}(\overline{X},\overline{Y})T_w(y,\overline{Y})^\top + E_{\text{INT}}(x,y)\\
&= (\mathscr{K}(x,\widehat{Y}) + E_{\text{INT}}(x,\widehat{Y}))\mathscr{K}(\widehat{X},\widehat{Y})^{-1}(\mathscr{K}(\widehat{X},y) + E_{\text{INT}}(\widehat{X},y))\\
&\quad + S_w(x,\overline{X})E_{\text{QR}}(\overline{X},\widehat{Y})\mathscr{K}_w(\widehat{X},\widehat{Y})^{-1}\mathscr{K}_w(\widehat{X},\overline{Y})T_w(y,\overline{Y})^\top\\
&\quad + S_w(x,\overline{X})\mathscr{K}_w(\overline{X},\widehat{Y})\mathscr{K}_w(\widehat{X},\widehat{Y})^{-1}E_{\text{QR}}(\widehat{X},\overline{Y})T_w(y,\overline{Y})^\top\\
&\quad + S_w(x,\overline{X})E_{\text{QR}}(\overline{X},\widehat{Y})\mathscr{K}_w(\widehat{X},\widehat{Y})^{-1}E_{\text{QR}}(\widehat{X},\overline{Y})T_w(y,\overline{Y})^\top\\
&\quad + S_w(x,\overline{X})E_{\text{QR}}(\overline{X},\overline{Y})T_w(y,\overline{Y})^\top + E_{\text{INT}}(x,y)
\end{aligned}$$

Distributing everything, factoring the weights matrices, and simplifying, we obtain the

following, where we indicate the bounds on each term on the right,

$$
\begin{aligned}
\mathscr{K}(x,y) &= \mathscr{K}(x,\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\mathscr{K}(\widehat{X},y) && \text{Approximation}\\
&+ E_{\text{INT}}(x,\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\mathscr{K}(\widehat{X},y) && \mathcal{O}(\delta q(r_0,r_1))\\
&+ \mathscr{K}(x,\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}E_{\text{INT}}(\widehat{X},y) && \mathcal{O}(\delta q(r_0,r_1))\\
&+ E_{\text{INT}}(x,\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}E_{\text{INT}}(\widehat{X},y) && \mathcal{O}\big(\delta p'(r_0,r_1)\big)\\
&+ S(x,\overline{X})\text{diag}(\overline{W}_X)^{-1/2}E_{\text{QR}}(\overline{X},\widehat{Y})\text{diag}(\widehat{W}_Y)^{-1/2}\\
&\quad \mathscr{K}(\widehat{X},\widehat{Y})^{-1}\mathscr{K}(\widehat{X},\overline{Y})T(y,\overline{Y})^\top && \mathcal{O}\Big(\varepsilon(\log r_0)^{2d}q(r_0,r_1)r_0^2\Big)\\
&+ S(x,\overline{X})\mathscr{K}(\overline{X},\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\text{diag}(\widehat{W}_X)^{-1/2}\\
&\quad E_{\text{QR}}(\widehat{X},\overline{Y})\text{diag}(\overline{W}_Y)^{-1/2}T(y,\overline{Y})^\top && \mathcal{O}\Big(\varepsilon(\log r_0)^{2d}q(r_0,r_1)r_0^2\Big)\\
&+ S(x,\overline{X})\text{diag}(\overline{W}_X)^{-1/2}E_{\text{QR}}(\overline{X},\widehat{Y})\\
&\quad \text{diag}(\widehat{W}_Y)^{-1/2}\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\text{diag}(\widehat{W}_X)^{-1/2}\\
&\quad E_{\text{QR}}(\widehat{X},\overline{Y})\text{diag}(\overline{W}_Y)^{-1/2}T(y,\overline{Y})^\top && \mathcal{O}\Big(\varepsilon(\log r_0)^{2d}r_0^2p_1(r_0,r_1)\Big)\\
&+ S(x,\overline{X})\text{diag}(\overline{W}_X)^{-1/2}E_{\text{QR}}(\overline{X},\overline{Y})\\
&\quad \text{diag}(\overline{W}_Y)^{-1/2}T(y,\overline{Y})^\top && \mathcal{O}\Big(\varepsilon(\log r_0)^{2d}r_0^2\Big)\\
&+ E_{\text{INT}}(x,y) && \mathcal{O}(\delta)
\end{aligned}
$$

This concludes the proof. □

What is simply left is then linking $\varepsilon$, $r_0$, and $r_1$. We have, from the CUR properties,

$$\varepsilon \leq p_1(r_0,r_1)\sigma_{r_1+1}(K_w)$$

which implies

$$|\mathscr{K}(x,y) - \mathscr{K}(x,\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\mathscr{K}(\widehat{X},y)| = \mathcal{O}(\sigma_{r_1+1}(K_w)s(r_0,r_1))$$

for another fixed-degree polynomial $s$.

Then, following the discussion from Section 5.2.2, we expect (note that this is not proven)

$$\sigma_i(K_w) = \sigma_i + \mathcal{O}(\delta)$$

Hence, if $\mathscr{K}$ has rapidly-decaying singular values, so does $K_w$. Assuming the singular values of $K_w$ decay exponentially fast, i.e.,

$$\log \sigma_k(K_w) \approx \text{poly}(k),$$

we find

$$|\mathscr{K}(x,y) - \mathscr{K}(x,\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\mathscr{K}(\widehat{X},y)| \to 0$$

as $r_0, r_1 \to \infty$, or alternatively, as $\varepsilon \to 0$.

## 5.3 Numerical stability

### 5.3.1 The problem

The previous section indicates that, at least theoretically, we can expect convergence as $\varepsilon \to 0$. However, the factorization

$$\mathscr{K}(X,Y) \approx \mathscr{K}(X,\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\mathscr{K}(\widehat{X},Y) \tag{5.15}$$

seems to be numerically challenging to compute. Indeed, as we showed in the previous section, we can only really expect at best $\|\widehat{K}_w^{-1}\|_2 = \mathcal{O}\big(\varepsilon^{-1}\big)$ which indicates that, roughly,

$$\kappa(\mathscr{K}(\widehat{X},\widehat{Y})) = \mathcal{O}\left(\frac{1}{\varepsilon}\right)$$

i.e., the condition number grows with the desired accuracy, and convergence beyond a certain threshold (like $10^{-8}$ in double-precision) seems impossible. Hence, we can reasonably be worried about the numerical accuracy of computing (5.15) even with a stable algorithm.

Note that this is not a pessimistic upper bound; by construction, $\widehat{K}_w$ really is ill-conditioned, and experiments show that solving linear systems $\widehat{K}_w x = b$ with random right-hand sides is numerically challenging and leads to errors of the order $\varepsilon^{-1}$.

### 5.3.2 Error Analysis

Consider (5.15) and let for simplicity

$$K_x = \mathscr{K}(X,\widehat{Y}), \quad K = \mathscr{K}(\widehat{X},\widehat{Y}), \quad K_y = \mathscr{K}(\widehat{X},Y)$$

In this section, our goal is to show why one can expect this formula to be accurately computed if one uses backward stable algorithms. As proved in Section 5.2, we have the following bounds on the interpolation operators

$$\|K_x K^{-1}\|_2 \leq q(r_0, r_1)$$
$$\|K^{-1} K_y\|_2 \leq q(r_0, r_1)$$

for some polynomial $q$. The key is that there is no $\varepsilon^{-1}$ in this expression. Those bounds essentially follow from the guarantees provided by the strong rank-revealing QR algorithm.

Now, let's compute the derivative of $K_x K^{-1} K_y$ with respect to $K_x$, $K$ and $K_y$ [123]:

$$\begin{aligned}\partial(K_x K^{-1} K_y) &= (\partial K_x) K^{-1} K_y + K_x(\partial(K^{-1}))K_y + K_x K^{-1}(\partial K_y) \\ &= (\partial K_x) K^{-1} K_y - K_x K^{-1}(\partial K) K^{-1} K_y + K_x K^{-1}(\partial K_y)\end{aligned}$$

Then, consider perturbing $K_x$, $K$, $K_y$ by $\gamma$ (assume all matrices are of order $\mathcal{O}(1)$ for the sake of simplicity), i.e., let $\delta K_x$, $\delta K_y$ and $\delta K$ be perturbations of $K_x$, $K_y$ and $K$, respectively, with

$$\|\delta K_x\| = \mathcal{O}(\gamma)\,, \|\delta K_y\| = \mathcal{O}(\gamma)\,, \|\delta K\| = \mathcal{O}(\gamma)\,.$$

Then, using the above derivative as a first-order approximation, we can write

$$\begin{aligned}&\|K_x K^{-1} K - (K_x + \delta K_x)(K + \delta K)^{-1}(K_y + \delta K_y)\| \\ &\leq \|\delta K_x\|\|K^{-1} K_y\| + \|K_x K^{-1}\|\|\delta K\|\|K^{-1} K_y\| + \|K_x K^{-1}\|\|\delta K_y\| + \mathcal{O}\big(\gamma^2\big) \\ &\leq 2\gamma q(r_0, r_1) + \gamma q(r_0, r_1)^2 + \mathcal{O}\big(\gamma^2\big) \\ &= \gamma(2q(r_0, r_1) + q(r_0, r_1)^2) + \mathcal{O}\big(\gamma^2\big)\end{aligned}$$

We see that the computed result is independent of the condition number of $K = \mathscr{K}(\widehat{X}, \widehat{Y})$ and depends on $q(r_0, r_1)$ only.

Assume now that we are using backward stable algorithms in our calculations [92]. We then know that the computed result is the result of an exact computation where the inputs have been perturbed by $\gamma$. The above result indicates that the numerical result (with roundoff errors) can be expected to be accurate up to $\gamma$ times a small polynomial, hence stable.

## 5.4 Skeletonized Interpolation as a new interpolation rule

As indicated in the introduction, one can rewrite

$$\begin{aligned}\mathscr{K}(x,y) &\approx \mathscr{K}(x,\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\mathscr{K}(\widehat{X},y)\\ &= \left[\mathscr{K}(x,\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\right]\mathscr{K}(\widehat{X},\widehat{Y})\left[\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\mathscr{K}(\widehat{X},y)\right]\\ &= \widehat{S}(x,\widehat{X})\mathscr{K}(\widehat{X},\widehat{Y})\widehat{T}(y,\widehat{Y})^{\top}\end{aligned}$$

where we recognize two new "cross-interpolation" (because they are built by considering both the $\mathcal{X}$ and $\mathcal{Y}$ space) operators $\widehat{S}(x,\widehat{X}) = \mathscr{K}(x,\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}$ and $\widehat{T}(y,\widehat{Y}) = \mathscr{K}(\widehat{X},y)^{\top}\mathscr{K}(\widehat{X},\widehat{Y})^{-\top}$. In this notation, each column of $\widehat{S}(x,\widehat{X})$ and $\widehat{T}(y,\widehat{Y})$ is a Lagrange function associated with the corresponding node in $\widehat{X}$ or $\widehat{Y}$ and evaluated at $x$ or $y$, respectively.

This interpretation is interesting as it allows us to "decouple" $x$ and $y$ and analyze them independently. In particular, one can look at the quality of the interpolation of the basis functions $u_k(x)$ and $v_k(y)$ using $\widehat{S}$ and $\widehat{T}$. Indeed, if this is accurate, it is easy to see that the final factorization is accurate. Indeed,

$$\begin{aligned}\mathscr{K}(x,y) &\approx \sum_{k=1}^{r}\sigma_k u_k(x)v_k(y)\\ &\approx \sum_{k=1}^{r}\sigma_k(\widehat{S}(x,\widehat{X})u_k(\widehat{X}))(\widehat{T}(y,\widehat{Y})v_k(\widehat{Y}))^{\top}\\ &= \widehat{S}(x,\widehat{X})\left(\sum_{k=1}^{r}\sigma_k u_k(\widehat{X})v_k(\widehat{Y})^{\top}\right)\widehat{T}(y,\widehat{Y})^{\top}\\ &\approx \widehat{S}(x,\widehat{X})\mathscr{K}(\widehat{X},\widehat{Y})\widehat{T}(y,\widehat{Y})^{\top}\\ &\approx \mathscr{K}(x,\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\mathscr{K}(\widehat{X},y)\end{aligned}$$

To illustrate this, let us consider a simple 1-dimensional example. Let $x, y \in [-1,1]$ and consider

$$\mathscr{K}(x,y) = \frac{1}{4+x-y}.$$

Then approximate this function up to $\varepsilon = 10^{-10}$, and obtain a factorization of rank $r$.

Figure 5.1 illustrates the $4^{\text{th}}$ Lagrange basis function in $x$, i.e., $\widehat{S}(x,\widehat{X})_4$ and the classical Lagrange polynomial basis function associated with the same set $\widehat{X}$. We see that they are both 1 at $\widehat{X}_4$ and 0 at the other points. However, $\widehat{S}(x,\widehat{X})_4$ is much more stable and small than its polynomial counterpart. In the case of polynomial interpolation at equispaced nodes, the growth of the Lagrange basis function (or, equivalently, of the Lebesgue constant) is the reason for the inaccuracy and instability.

Figure 5.2 shows the effect of interpolating $u_r(x)$ using $\widehat{S}(x,\widehat{X})$ as well as using the usual polynomial interpolation at the nodes $\widehat{X}$. We see that $\widehat{S}(x,\widehat{X})$ interpolates very well $u_r(x)$, showing indeed that we implicitly build an accurate interpolant, even on the last (least smooth) eigenfunctions. The usual polynomial interpolation fails to capture any feature of $u_r$ on the other hand. Note that we could have reached a similar accuracy using interpolation at Chebyshev nodes but only by using many more interpolation nodes.

Finally, Figure 5.3 shows how well we approximate the various $r$ eigenfunctions. As one can see, interpolation is very accurate on $u_1(x)$, but the error grows for less smooth eigenfunctions. The growth is, roughly, similar to the growth of $\frac{\varepsilon}{\sigma_i}$. Notice how this is just enough so that the resulting factorization is accurate:

$$\begin{aligned}
\widehat{S}(x,\widehat{X})\mathscr{K}(\widehat{X},y) &= \sum_{s=1}^{r} \sigma_s \widehat{S}(x,\widehat{X}) u_s(\widehat{X}) v_s(y) + \mathcal{O}(\varepsilon) \\
&= \sum_{s=1}^{r} \sigma_s \left( u_s(x) + \mathcal{O}\left(\frac{\varepsilon}{\sigma_s}\right)\right) v_s(y) + \mathcal{O}(\varepsilon) \\
&= \sum_{s=1}^{r} \sigma_s u_s(x) v_s(y) + \sum_{s=1}^{r} \mathcal{O}(\varepsilon)\, v_s(y) + \mathcal{O}(\varepsilon) \\
&= \mathscr{K}(x,y) + \mathcal{O}(\varepsilon)
\end{aligned}$$

It is also consistent with the analysis of Section 5.3. This illustrates how the algorithm works: it builds an interpolation scheme that allows for interpolating the various eigenfunctions of $\mathscr{K}$ with just enough accuracy so that the resulting interpolation is accurate up to the desired accuracy.

## 5.5 Numerical experiments

In this section, we present some numerical experiments on various geometries. We study the quality (how far $r_1$ is from the optimal SVD-rank $r$ and how accurate the approximation

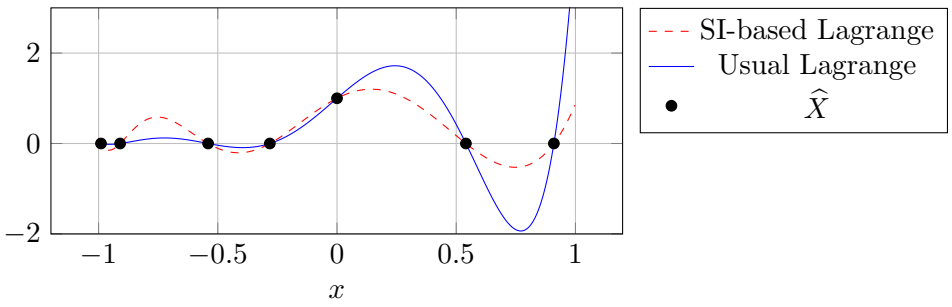



Figure 5.1: $4^{\text{th}}$ Lagrange basis function. We see that the SI-based Lagrange basis function is more stable than the polynomial going through the same interpolation nodes.

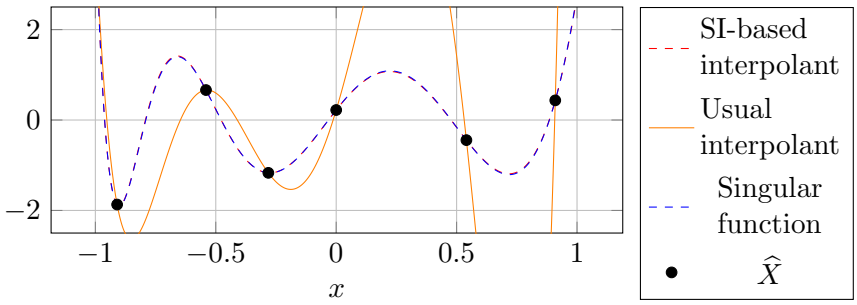



Figure 5.2: Interpolation of the last (and least smooth) eigenfunction. We see that the SI interpolant is much more accurate than the polynomial interpolant going through the same interpolation nodes.

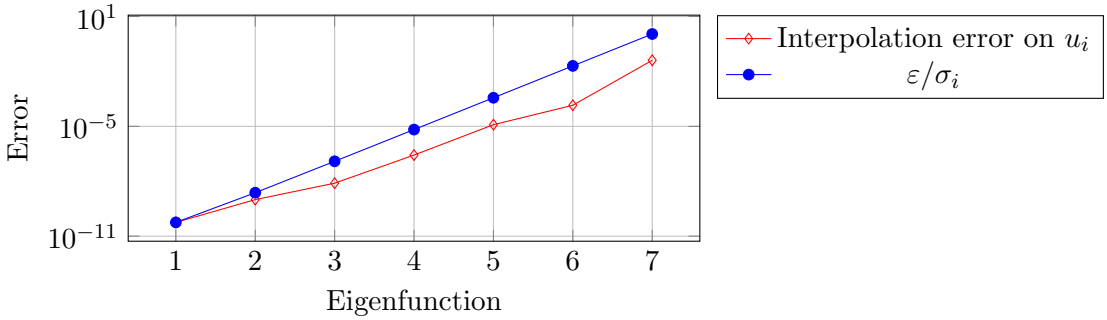



Figure 5.3: Interpolation error on the various eigenfunctions. The error grows just slowly enough with the eigenfunctions so that the overall interpolant is accurate up to the desired accuracy.

is) of the algorithm in Section 5.5.1 and Section 5.5.2. We illustrate in Section 5.5.3 the improved guarantees of RRQR and justify the use of weights in Section 5.5.4. Finally, Section 5.5.5 studies the algorithm computational complexity.

The experiments are done using Julia [20] and the code is sequential. For the strong rank-revealing QR algorithm, we use the `LowRankApprox.jl` Julia package [93]. The code can be downloaded from `https://stanford.edu/~lcambier/papers.html`.

### 5.5.1 Simple geometries

We begin this section with an elementary problem, as depicted in Figure 5.4b. In this problem, we consider the usual kernel $\mathscr{K}(x,y) = \|x-y\|_2^{-1}$ where $x, y \in \mathbb{R}^2$. $\mathcal{X}$ and $\mathcal{Y}$ are two squares of side of length 1, centered at $(0.5, 0.5)$ and $(2.5, 2.5)$ respectively. Finally, $X$ and $Y$ are two uniform meshes of $50 \times 50$ mesh points each, i.e., $n = 2500$.

We pick the Chebyshev grids $\overline{X}$ and $\overline{Y}$ using a heuristic based on the target accuracy $\varepsilon$. Namely, we pick the number of Chebyshev nodes in each dimension (i.e., $x_1$, $x_2$, $y_1$ and $y_2$) independently (by using the midpoint of $\mathcal{X}$ and $\mathcal{Y}$ as reference), such that the interpolation error is approximately less than $\varepsilon^{3/4}$. This value is heuristic but performs well for those geometries. Other techniques are possible. This choice is based in part on the observation that the algorithm is accurate even when $\delta > \varepsilon$, i.e., when the Chebyshev interpolation is *less accurate* than the actual final low-rank approximation through Skeletonized Interpolation.

Consider then Figure 5.4a. The $r_0$ line indicates the rank ($r_0 = \min(|\overline{X}|, |\overline{Y}|)$) of the low-rank expansion through interpolation. The $r_1$ line corresponds to the rank obtained after the RRQR over $\mathscr{K}(\overline{X}, \overline{Y})$ and its transpose, i.e., it is the rank of the final approximation

$$\mathscr{K}(X,Y) \approx \mathscr{K}(X,\widehat{Y})\mathscr{K}(\widehat{X},\widehat{Y})^{-1}\mathscr{K}(\widehat{X},Y)$$

Finally, "SVD rank" is the rank one would obtain by truncating the SVD of

$$\mathscr{K}(X,Y) = USV^\top$$

at the appropriate singular value, as to ensure

$$\|\mathscr{K}(X,Y) - \tilde{\mathscr{K}}(X,Y)\|_F \approx \varepsilon \|\mathscr{K}(X,Y)\|_F$$

Similarly, "RRQR" is the rank a rank-revealing QR on $\mathscr{K}(X,Y)$ would obtain. This is

usually slightly suboptimal compared to the SVD. Those two values are there to illustrate that $r_1$ is close to the optimal value.

The conclusion regarding Figure 5.4a is that Skeletonized Interpolation is nearly optimal in terms of rank. While the rank obtained by the interpolation is clearly far from optimal, the RRQR over $\mathscr{K}(\overline{X}, \overline{Y})$ allows us to find subsets $\widehat{X} \subset \overline{X}$ and $\widehat{Y} \subset \overline{Y}$ that are enough to represent well $\mathscr{K}(X, Y)$, and the final rank $r_1$ is nearly optimal compared to the SVD-rank $r$. We also see that the rank of a blind RRQR over $\mathscr{K}(X, Y)$ is higher than the SVD-rank and usually closer—if not identical—to $r_1$.

We want to re-emphasize that, in practice, *the error of the sets* $\overline{X}$, $\overline{Y}$ *—i.e., the error of the polynomial interpolation based on* $\overline{X} \times \overline{Y}$ *— can be larger than the required tolerance.* If they are large enough, the compressed sets $\widehat{X}$ and $\widehat{Y}$ will contain enough information to properly interpolate $\mathscr{K}$ and the final error will be smaller than the required tolerance. This is important, as the size of the Chebyshev grid for a given tolerance can be fairly large (as indicated in the introduction, and one of the main motivations of this work), even though the final rank is small.

As a sanity check, Figure 5.4c gives the relative error measured in the Frobenius norm

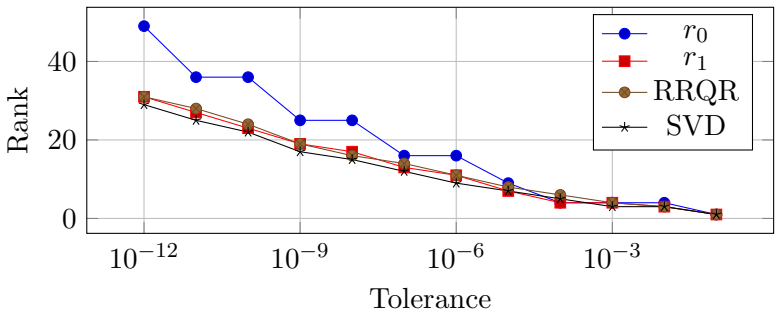
$$\frac{\|\mathscr{K}(X, Y) - \mathscr{K}(X, \widehat{Y})\mathscr{K}(\widehat{X}, \widehat{Y})^{-1}\mathscr{K}(\widehat{X}, Y)\|_F}{\|\mathscr{K}(X, Y)\|_F}$$

between $\mathscr{K}(X, Y)$ and its interpolation as a function of the tolerance $\varepsilon$.[1] We see that both lines are almost next to each other, meaning our approximation indeed reaches the required tolerance. This is important as it means that one can effectively *control* the accuracy.

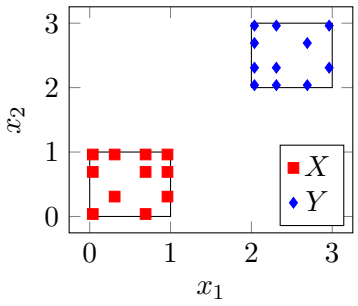
Finally, Figure 5.4b also shows the resulting $\widehat{X}$ and $\widehat{Y}$. It is interesting to notice how the selected points cluster near the close corners, as one could expect since this is the area where the kernel is the least smooth.

We then consider results for the same Laplacian kernel $\mathscr{K}(x, y) = \|x - y\|_2^{-1}$ between two plates in 3D (Figure 5.5b). We observe overall very similar results as for the previous case in Figure 5.5a, where the initial rank $r_0$ is significantly decreased to $r_1$ while keeping the resulting accuracy close to the required tolerance as Figure 5.5c shows. Finally, one can see in Figure 5.5b the selected Chebyshev nodes. They again cluster in the areas where smoothness is the worst, i.e., at the close edges of the plates.

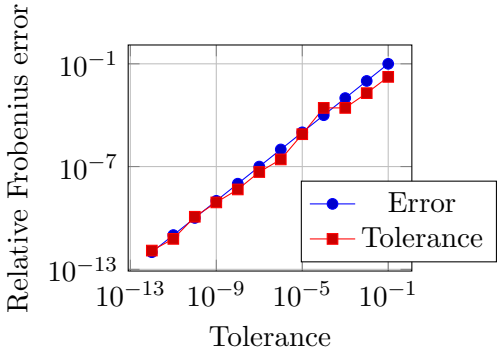
[1] Choosing the Frobenius norm is not critical—very similar results are obtained in the 2-norm for instance.




(a) Ranks as a function of the desired accuracy

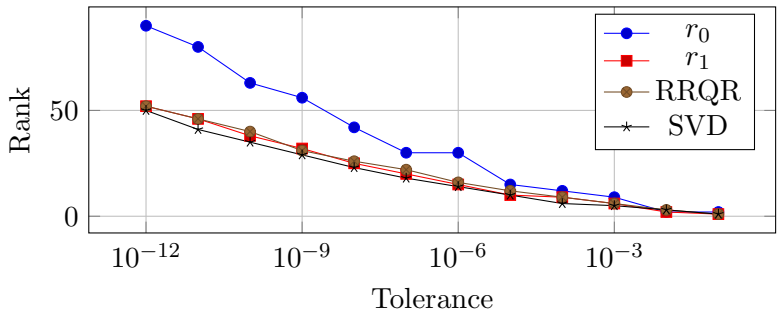



(b) The geometry used, and the resulting choice of $\widehat{X}$, $\widehat{Y}$ for a tolerance of $10^{-6}$.

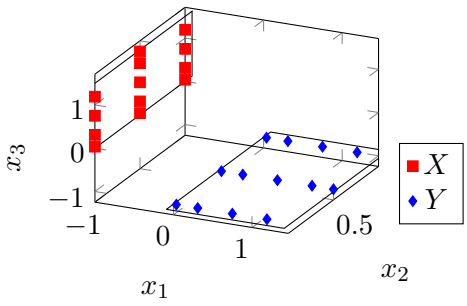



(c) Relative Frobenius-norm error as a function of the desired accuracy

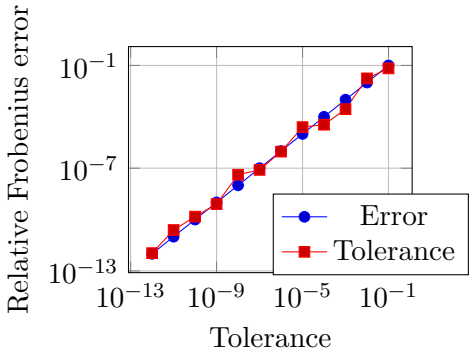
Figure 5.4: Results for the 2D-squares example. The rank $r_0$ before compression is significantly reduced to $r_1$, very close to the true SVD or RRQR-rank.




(a) Ranks as a function of the desired accuracy




(b) The geometry used, and the resulting choice of $\widehat{X}$, $\widehat{Y}$ for a tolerance of $10^{-6}$.




(c) Relative Frobenius-norm error as a function of the desired accuracy

Figure 5.5: Results for the perpendicular plates example. The rank $r_0$ before compression is significantly reduced to $r_1$, very close to the true SVD or RRQR-rank.

### 5.5.2 Comparison with ACA and Random Sampling

We then compare our method with other standard algorithms for kernel matrix factorization. In particular, we compare it with ACA [18] and 'Random CUR' where one selects, at random, pivots $\widetilde{X}$ and $\widetilde{Y}$ and builds a factorization

$$\mathscr{K}(X,Y) \approx \mathscr{K}(X,\widetilde{Y})\mathscr{K}(\widetilde{X},\widetilde{Y})^{-1}\mathscr{K}(\widetilde{X},Y)$$

based on those. As we are interested in comparing the *quality* of the resulting sets of pivots for a given rank, we compare those algorithms for sets $X$ and $Y$ with a variable distance between each other, for a fixed tolerance ($\varepsilon = 10^{-8}$), and kernel ($\mathscr{K}(x,y) = \|x-y\|_2^{-1}$). The geometry is two unit-length squares side-by-side with a variable distance between their closest edges.

The comparison is done in the following way:

1. Given $r_1$, build the ACA factorization of rank $r_1$ and compute its relative error in Frobenius norm with $\mathscr{K}(X,Y)$;

2. Given $r_1$, build the random CUR factorization by sampling uniformly at random $r_1$ points from $X$ and $Y$ to build $\widetilde{X}$, $\widetilde{Y}$. Then, compute its relative error with $\mathscr{K}(X,Y)$.

We then do so for sets of varying distance, and for a given distance, we repeat the experiment 25 times by building $X$ and $Y$ at random within the two squares. This allows us to study the variance of the error and to collect some statistics.

Figure 5.6 gives the resulting errors in relative Frobenius norm for the three algorithms using box-plots of the errors to show distributions. The rectangular boxes represent the distributions from the 25% to the 75% quantiles, with the median in the center. The thinner lines represent the complete distribution, except outliers depicted using large dots. We observe that the ($\widehat{X}$, $\widehat{Y}$) sets based on Chebyshev-SI are, *for a common size* $r_1$, *more accurate* than the Random or ACA sets. In addition, by design, they lead to more stable factorizations (as they have very small variance in terms of accuracy) while ACA for instance has a higher variance. We also see, as one may expect, that while ACA is still fairly stable even when the clusters get close, random CUR starts having a higher and higher variance. This is understandable as the kernel gets less and less smooth as the clusters get close.

Finally, we ran the same experiments with several other kernels ($r^{-2}$, $r^{-3}$, $\log(r)$, $\exp(-r)$, $\exp(-r^2)$) and observed quantitatively very similar results.

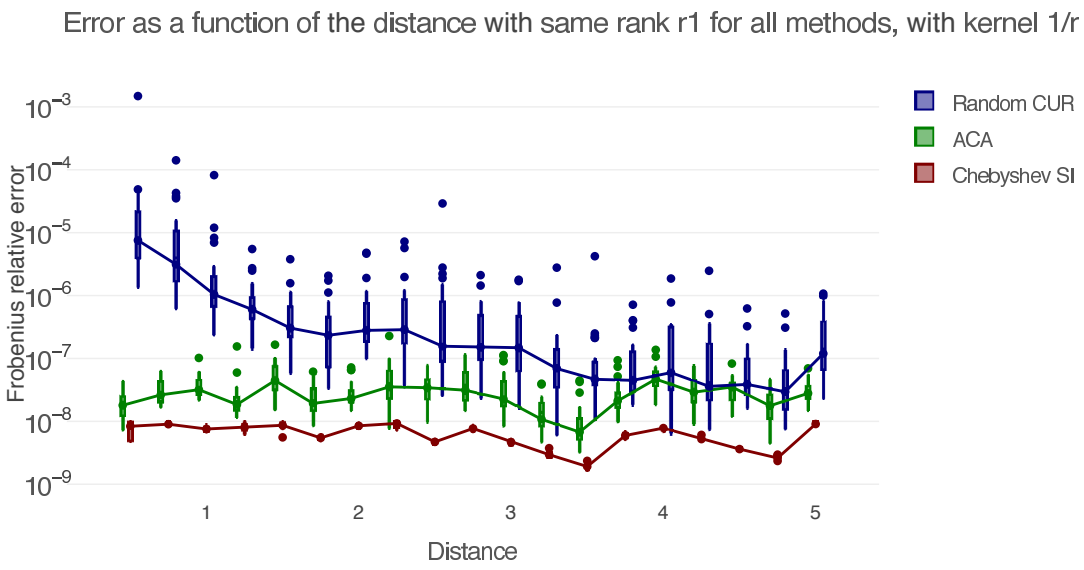



Figure 5.6: Comparison between different algorithms: Chebyshev-based SI, ACA and purely random CUR decomposition. We consider two 2D squares of sides 1 with a variable distance from each other; for each distance, we run Chebyshev-based SI and find the smallest sets $\overline{X}, \overline{Y}$ of rank $r_0$ leading to a factorization using $\widehat{X}, \widehat{Y}$ of sizes $r_1$ with relative error at most $10^{-8}$. Then $r_1$ is used as an a priori rank for ACA and Random CUR. We randomize the experiments by subsampling 500 points from a large $100 \times 100$ points grid in each square.

### 5.5.3 Stability guarantees provided by RRQR

In Algorithm 5.1, in principle, any rank-revealing factorization providing pivots could be used. In particular, ACA itself could be used. In this case, this is the HCAII (without the weights) algorithm as described in [25]. However, ACA is only a heuristic: unlike strong rank-revealing factorizations, it can't always reveal the rank. In particular, it may have issues when some parts of $X$ and $Y$ have strong interactions while others are weakly coupled. To highlight this, consider the following example. It can be extended to many other situations.

Let us use the rapidly decaying kernel

$$\mathscr{K}(x,y) = \frac{1}{\|x-y\|_2^3}$$

and the situation depicted in Figure 5.7 with $X = \begin{bmatrix} X_1 & X_2 \end{bmatrix}$ and $Y = \begin{bmatrix} Y_1 & Y_2 \end{bmatrix}$. We note that, formally, $X$ and $Y$ are not well-separated.

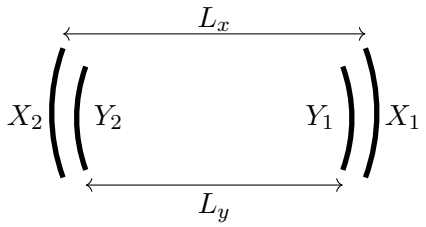
Since $\mathscr{K}$ is rapidly decaying and $X_1$ and $Y_2$ (resp. $X_2$ and $Y_1$) are far away from each other, the resulting matrix is *nearly* block diagonal, i.e.,

$$\mathscr{K}(X,Y) \approx \begin{bmatrix} \mathscr{K}(X_1,Y_1) & \mathcal{O}(\varepsilon) \\ \mathcal{O}(\varepsilon) & \mathscr{K}(X_2,Y_2) \end{bmatrix} \tag{5.16}$$

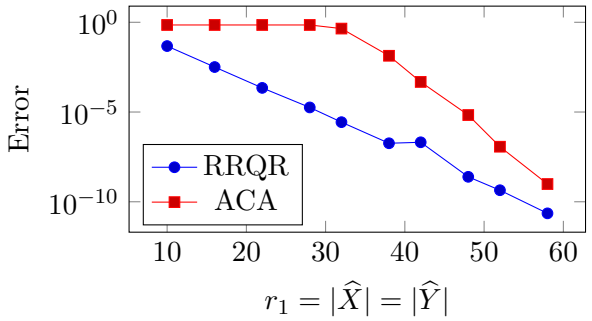
for some small $\varepsilon$. This is a challenging situation for ACA since it will need to sweep through the initial block completely before considering the other one. In practice heuristics can help alleviate the issue; see ACA+ [25] for instance. Those heuristics, however, do not come with any guarantees. Strong RRQR, on the other hand, does not suffer from this and picks optimal nodes in each cluster from the start. It guarantees stability and convergence.

### 5.5.4 The need for weights

Another characteristic of Algorithm 5.1 is the presence of weights. We illustrate here why this is necessary in general. Algorithm 5.1 uses $\overline{X}$ and $\overline{Y}$ both to select interpolation points (the "columns" of the RRQR) and to evaluate the resulting error (the "rows"). Hence, a non-uniform distribution of points leads to over- or under-estimated $L_2$ error and to a biased interpolation point selection. The weights, roughly equal to the (square root of) the inverse points density, alleviate this effect. This is a somewhat small effect in the case of




(a) The geometry. $L_y = 0.9L_x$, each domain $X_i, Y_i$ has 50 points uniformly distributed (hence, the weights are uniform) on an arc of angle $\pi/4$. $\overline{X} = X$ and $\overline{Y} = Y$, and $\widehat{X}$, $\widehat{Y}$ are $r_1$ points subsampled from $\overline{X}, \overline{Y}$ using Algorithm 5.1.



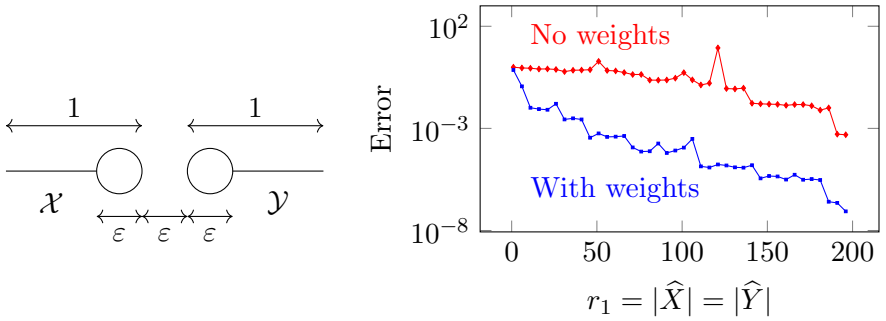

(b) Relative Frobenius error over $X \times Y$ using both RRQR and ACA to select $\widehat{X}, \widehat{Y}$ of size $r_1$ from $\overline{X}$, $\overline{Y}$ using Algorithm 5.1.

Figure 5.7: Failure of ACA. The geometry is such that the coupling between $X_1/Y_1$ and $X_2/Y_2$ is much stronger than between $X_1/Y_2$ and $X_2/Y_1$. This leads to ACA not selecting pivots properly. RRQR on the other hand has no issue and converges steadily.

Chebyshev nodes & weights as the weights have limited amplitudes.

To illustrate this phenomenon more dramatically, consider the situation depicted in Figure 5.8. We define $\overline{X}$ and $\overline{Y}$ in the following way. Align two segments of $N$ points, separated by a small interval of length $\varepsilon = 1/N$ with $N = 200$. At the close extremities, we insert $25N$ additional points inside small 3D spheres of diameter $\varepsilon$. As a result, $|\overline{X}| = |\overline{Y}| = 26N = 5,200$, and $\overline{X}$, $\overline{Y}$ are strongly non-uniform. We see that the small spheres hold a large number of points in an interval of length $N^{-1}$. As a result, their associated weight should be proportional to $N^{-1/2}$, while the weight for the points on the segments should be proportional to 1. Then we apply Algorithm 5.1 with and without weights, and evaluate the error on the segments using $|X| = |Y| = 10,000$ equispaced points as a proxy for the $L_2$ error.

When using a rank $r_0 = 200$, the CUR decomposition picks only 6 more points on the segments (outside the spheres) for the weighted case compared to the unweighted. However, this is enough to dramatically improve the accuracy, as Figure 5.8b shows. Overall, the presence of weights has a large effect, and this shows that in general, one should appropriately weigh the node matrix $K_w$ to ensure maximum accuracy.




(a) The geometry. Kernel is $1/r$, $\varepsilon = 0.01$.

(b) Error with and without weights in Algorithm 5.1.

Figure 5.8: Benchmark demonstrating the importance of using weights in the RRQR factorization. The setup for the benchmark is described in the text. The blue curve on the right panel, which uses weights, has a much improved accuracy.

### 5.5.5 Computational complexity

We finally study the computational complexity of the algorithm. It's important to note that two kinds of operations are involved: kernel evaluations and classical flops. As they may potentially differ in cost, we keep those separated in the following analysis.

The cost of the various parts of the algorithm is the following :

- $\mathcal{O}(r_0^2)$ kernel evaluations for the interpolation, i.e., the construction of $\overline{X}$ and $\overline{Y}$ and the construction of $\mathscr{K}(\overline{X}, \overline{Y})$
- $\mathcal{O}(r_0^2 r_1)$ flops for the RRQR over $\mathscr{K}(\overline{X}, \overline{Y})$ and $\mathscr{K}(\overline{X}, \overline{Y})^\top$
- $\mathcal{O}((m+n)r_1)$ kernel evaluations for computing $\mathscr{K}(X, \widehat{Y})$ and $\mathscr{K}(\widehat{X}, Y)$, respectively (with $m = |X|$ and $n = |Y|$)
- $\mathcal{O}(r_1^3)$ flops for $\mathscr{K}(\widehat{X}, \widehat{Y})^{-1}$ (through, say, an LU factorization)

So the total complexity of building the three factors is $\mathcal{O}((m+n)r_1)$ kernel evaluations. If $m = n$ and $r_1 \approx r$, the total complexity is

$$\mathcal{O}((m+n)r_1) \approx \mathcal{O}(nr)$$

Also note that the memory requirements are, clearly, of order $\mathcal{O}((m+n)r_1)$.

When applying this low-rank matrix on a given input vector $f(Y) \in \mathbb{R}^n$, the cost is

- $\mathcal{O}(r_1 n)$ flops for computing $w_1 = \mathscr{K}(\widehat{X}, Y) f(Y)$
- $\mathcal{O}\left(r_1^2\right)$ flops for computing $w_2 = \mathscr{K}(\widehat{X}, \widehat{Y})^{-1} w_1$ assuming a factorization of $\mathscr{K}(\widehat{X}, \widehat{Y})$ has already been computed
- $\mathcal{O}(m r_1)$ flops for computing $w_3 = \mathscr{K}(X, \widehat{Y}) w_2$

So the total cost is

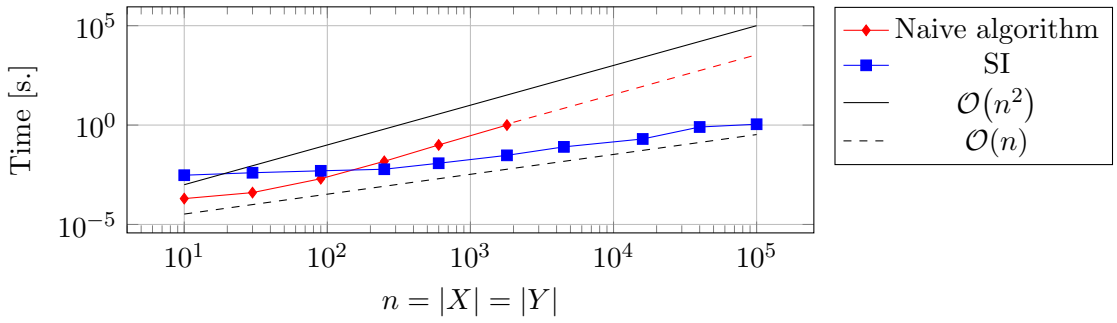
$$\mathcal{O}((m+n) r_1) \approx \mathcal{O}(nr)$$

flops if $m = n$ and $r_1 \approx r$.

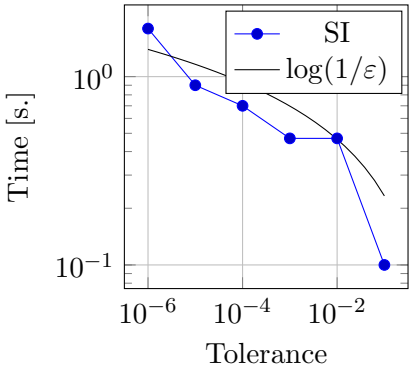
To illustrate those results, Figure 5.9a shows, using the same setup as in the 2D square example of Section 5.5.1, the time (in seconds) taken by our algorithm versus the time taken by a naive algorithm that would first build $\mathscr{K}(X, Y)$ and then perform a rank-revealing QR on it. Time is given as a function of $n$ for a fixed accuracy $\varepsilon = 10^{-8}$. One should not focus on the absolute values of the timing but rather the asymptotic complexities. In this case, the $\mathcal{O}(n)$ and $\mathcal{O}\left(n^2\right)$ complexities clearly appear, and our algorithm scales much better than the naive one (or that any algorithm that requires building the full matrix first). Note that we observe no loss of accuracy as $n$ grows. Also, note that the plateau at the beginning of the Skeletonized Interpolation curve is all the overhead involved in selecting the Chebyshev points $\overline{X}$ and $\overline{Y}$ using some heuristic. This is very implementation-dependent and could be reduced significantly with a better or more problem-tailored algorithm. However, since this is by design independent of $X$ and $Y$ (and, hence, $n$) it does not affect the asymptotic complexity.

Figure 5.9b shows the time as a function of the desired accuracy $\varepsilon$, for a fixed number of mesh points $n = 10^5$. Since the singular values of $\mathscr{K}(X, Y)$ decay exponentially, one has $r \approx \mathcal{O}\left(\log\left(\frac{1}{\varepsilon}\right)\right)$. The complexity of the algorithm being $\mathcal{O}(nr)$, we expect the time to be proportional to $\log\left(\frac{1}{\varepsilon}\right)$. This is indeed what we observe.

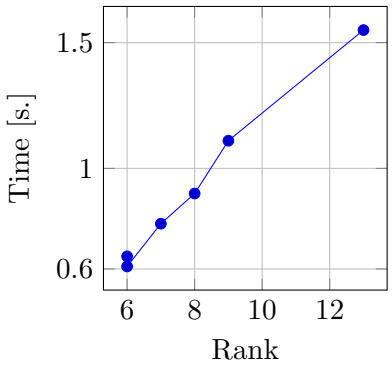
Figure 5.9c depicts the time as a function of the rank $r$ for a fixed accuracy $\varepsilon = 10^{-8}$ and for a number of mesh points $n = 10^5$. In that case, to vary the rank and keep $\varepsilon$ fixed, we change the geometry and observe the resulting rank. This is done by moving the top-right square (see Figure 5.4b) towards the bottom-left one (keeping approximately one cluster diameter between them) or away from it (up to 6 diameters). The rank displayed is the rank obtained by the factorization. As expected, the algorithm scales linearly as a function of $r$.




(a) Time as a function of $n$ for a fixed tolerance. The plateau is the overhead in the Skeletonized Interpolation algorithm that is independent of $n$.




(b) Time as a function of $\varepsilon$, for fixed clusters of points.




(c) Time as a function of $r$. The rank is varied by increasing the distance between the two clusters, for a fixed tolerance and number of points.

Figure 5.9: Timings experiments on Skeletonized Interpolation

## 5.6 Conclusions

In this work, we built a kernel matrix low-rank approximation based on Skeletonized interpolation. This can be seen as an optimal way to interpolate families of functions using a custom basis.

This type of interpolation, by design, is always at least as good as polynomial interpolation as it always requires the minimal number of basis functions for a given approximation error. We proved in this chapter the asymptotic convergence of the scheme for kernels exhibiting fast (i.e., faster than polynomial) decay of singular values. We also proved the numerical stability of general Schur-complement types of formulas when using a backward stable algorithm.

In practice, the algorithm exhibits a low computational complexity of $\mathcal{O}(nr)$ with small constants and is very simple to use. Furthermore, the accuracy can be set a priori and in practice, we observe nearly optimal convergence of the algorithm. Finally, the algorithm is completely insensible to the mesh point distribution, leading to more stable sets of "pivots" than Random Sampling or ACA.

# Chapter 6

# The Index of Invariance

As indicated on page vi, this chapter contains excerpts from [35]. Both authors were equally involved in this work, and some of the results in this chapter were originally written by Rahul Sarkar.

## 6.1 Introduction

In [89], the authors recently showed that $x_\omega$ ($\omega \geq 0$) defined as

$$x_\omega := \operatorname{argmin} \left\| (A^*A + \omega I)^{-\frac{1}{2}} A^*(b - Ax) \right\|_2 \quad \text{s.t. } x \in \mathcal{K}_k(A^*A, A^*b), \tag{6.1}$$

for some $b \in \mathbb{R}^m$ and $A \in \mathbb{R}^{m \times n}$ full-rank are convex combinations of the LSQR iterates [119] (equal to $x_0$) and LSMR iterates [64] (equal to $x_\omega$ as $\omega \to \infty$). $\mathcal{K}_k(A^*A, A^*b)$ is the $k^{\text{th}}$ Krylov subspace over which we minimize (6.1). This also shows that the set $\{x_\omega | \omega \geq 0\}$ lies within a one-dimensional subspace.

For any full rank matrix $A$, $A^*A$ is SPD. Conversely, any SPD matrix $B \in \mathbb{R}^{n \times n}$ can be decomposed as $B = L^*L$. Hence, we deduce that the solutions $\{x'_\omega \mid \omega \geq 0\}$ to the related problem

$$x'_\omega := \operatorname{argmin} \left\| (B + \omega I)^{-\frac{1}{2}} (b' - Bx) \right\|_2 \quad \text{s.t. } x \in \mathcal{K}_k(B, b'), \tag{6.2}$$

also lie on a line between $x_0$, the CG iterates [91], and $x_\omega$ as $\omega \to \infty$, the MINRES iterates [118].

In this chapter, we generalize the previously mentioned one-dimensional affine subspace

result by studying the minimization problem

$$x_{b,\omega} := \text{argmin}_{x\in\mathcal{S}} \|(A+\omega I)^{-\frac{1}{2}}(Ax-b)\|_2, \tag{6.3}$$

with $\mathcal{S}$ an arbitrary (not only Krylov) $p$-dimensional subspace of $\mathbb{F}^n$ ($\mathbb{F}=\mathbb{R}$ or $\mathbb{F}=\mathbb{C}$), $A$ a Hermitian invertible matrix, and $\omega_{\min} := -\lambda_{\min}(A) < \omega \in \mathbb{R}$, where $\lambda_{\min}(A) \in \mathbb{R}$ is the smallest eigenvalue of $A$. For any subspace $\mathcal{S}$ of $\mathbb{F}^n$, we also define $\mathbf{Ind}_A(\mathcal{S}) := \dim(\mathcal{S}+A\mathcal{S}) - \dim(\mathcal{S})$, which we call the index of invariance of $\mathcal{S}$ with respect to $A$.

One can see that the role of $\mathbf{Ind}_A(\mathcal{S})$ is to capture how far $\mathcal{S}$ is from being $A$-invariant. If $\mathbf{Ind}_A(\mathcal{S})=0$, then $\mathcal{S}$ is $A$-invariant. Non-invariant Krylov subspaces, for which $\mathbf{Ind}_A(\mathcal{S})=1$, are nearly invariant.

In the next section, we will show that $x_{b,\omega} := V\left(V^*AA_\omega^{-1}AV\right)^{-1}V^*AA_\omega^{-1}b$ (with $V$ an orthogonal basis for $\mathcal{S}$ and $A_\omega = A+\omega I$). From this, it can be seen that, when $\mathcal{S}$ is $A-$invariant, or equivalently when $\mathbf{Ind}_A(\mathcal{S})=0$, $x_{b,\omega}=x_{b,\mu}$ for all $b$ and $\omega,\mu>\omega_{\min}$. Hence, the solutions belong to a $0-$dimensional subspace. A natural question that arises is then whether this pattern can be extended to the case where $\mathbf{Ind}_A(\mathcal{S})>0$, namely, do the solutions belong to a $\mathbf{Ind}_A(\mathcal{S})$-dimensional subspace. We already know that this is the case in the particular case where $\mathcal{S}$ is a non-invariant Krylov subspace (for which $\mathbf{Ind}_A(\mathcal{S})=1$) [89], and this chapter is dedicated to generalizing this result.

We prove several results:

(i) We study (Section 6.2) the index of invariance and show its relationship to a block decomposition of $A$. This block decomposition can be seen as an extension of the block Schur decomposition (for an invariant subspace) or Hessenberg reduction of a matrix (for a Krylov subspace).

(ii) We show (Section 6.3) that there exists a subspace $\mathcal{Y}$ (depending on $A$ and $\mathcal{S}$ but not $b$, $\omega$ or $\mu$) such that for all $b\in\mathbb{F}^n$, $\omega,\mu>\omega_{\min}$, $x_{b,\omega}-x_{b,\mu}\in\mathcal{Y}$, where $\dim(\mathcal{Y})\leq\mathbf{Ind}_A(\mathcal{S})$. This immediately generalizes the result from [89], since $\mathbf{Ind}_A(\mathcal{S})=1$ for Krylov subspaces. We give an expression for $\mathcal{Y}$ as a function of $A$ and $\mathcal{S}$. This result indicates that the solutions to (6.3) belong to a much smaller space, in general, than $\mathcal{S}$. And the dimension of this space is directly related to how far $\mathcal{S}$ is from being $A$-invariant. This shows that this problem has more structure than what can be seen at first sight.

(iii) We then study (Section 6.4) the tightness of the previously mentioned bound. First,

we vary $\omega$, keeping $b$ fixed. Let $\mathcal{X}_b := \text{span}(\{x_{b,\omega} - x_{b,\mu} \mid \omega, \mu > \omega_{\min}\})$.

- We show that the 0-dimensional case is special, as $\dim(\mathcal{X}_b) = 0$ for all $b \in \mathbb{F}^n$, if and only if $\mathbf{Ind}_A(\mathcal{S}) = 0$.
- We show however that there exist $A$ and $\mathcal{S}$ such that, for all $b \in \mathbb{F}^n$, $\dim(\mathcal{X}_b) = 1$, while $\mathbf{Ind}_A(\mathcal{S})$ can be arbitrarily large. This indicates that the bound on $\dim(\mathcal{Y})$ is not tight if one can only vary $\omega$, with $b$ fixed.

Then, we continue by studying instead a related set $\mathcal{X} := \text{span}(\{x_{b,\omega} - x_{b,\mu} \mid b \in \mathbb{F}^n, \omega, \mu > \omega_{\min}\})$, where we find some sufficient conditions on $A$ and $\mathcal{S}$ ensuring $\dim(\mathcal{X}) = \mathbf{Ind}_A(\mathcal{S})$. In particular, if $A$ is SPD, then we prove that $\dim(\mathcal{X}) = \mathbf{Ind}_A(\mathcal{S})$. This shows that the previously derived bound on $\dim(\mathcal{Y})$ is now tight if one is free to vary both $\omega$ and $b$ in $x_{b,\omega}$.

## 6.2 Preliminaries

### 6.2.1 Notations

Table 6.1 summarizes the notations used throughout this chapter. We separate general notations (top) and notations specific to the problem under consideration (bottom).

### 6.2.2 Index of invariance

We begin by defining the index of invariance of a matrix $A$ with respect to a subspace $\mathcal{S}$.

**Definition 6.1.** *Let $A \in \mathbb{F}^{n\times n}$ and $\mathcal{S} \subseteq \mathbb{F}^n$ a subspace. We define the index of invariance of $\mathcal{S}$ with respect to $A$ as $\boldsymbol{Ind}_A(\mathcal{S}) := dim(\mathcal{S} + A\mathcal{S}) - dim(\mathcal{S})$.*

If $A \in \mathbb{F}^{n\times n}$, a subspace $\mathcal{S} \subseteq \mathbb{F}^n$ is called an *invariant subspace* of $A$ or simply $A-$*invariant* if $A\mathcal{S} \subseteq \mathcal{S}$. Thus it can be seen from Definition 6.1 that $\mathbf{Ind}(\mathcal{S}, A) = 0$ if and only if $\mathcal{S}$ is $A-$invariant. Another interesting example is that of a Krylov subspace $\mathcal{K}_k(A, b)$ that is not $A-$invariant, in which case it can be verified that $\mathbf{Ind}(\mathcal{K}_k(A, b), A) = 1$. We study several interesting properties of the index of invariance below in Section 6.2.4.

### 6.2.3 Problem statement

In this subsection, after we formally state the problem in the next paragraph, we will define a few quantities that will be used in its analysis and prove some facts.

| | |
|---|---|
| $\mathbb{F}$ | $\mathbb{R}$ or $\mathbb{C}$, depending on the context |
| Semi-unitary | A matrix $A \in \mathbb{F}^{m\times n}$ such that $A^*A = I$ |
| $\mathrm{Sym}(n)$ | The set of Hermitian matrices |
| $\mathrm{P}^+(n)$ | The set of SPD matrices |
| $\mathrm{GL}(n)$ | The set of invertible matrices |
| $\mathrm{Gr}^{\mathbb{F}}(p,n)$ | The set of $p$-dimensional subspaces of $\mathbb{F}^n$ |
| $\lambda_{\min}(A)$ | With $A \in \mathrm{Sym}(n)$, the smallest eigenvalue of $A$ |
| $A\mathcal{S}$ | With $A \in \mathbb{F}^{n\times n}$ and $S \subseteq \mathbb{F}^n$ a subspace, the subspace $\{Ax \mid x \in \mathcal{S}\}$ |
| $\mathcal{S}^\perp$ | With $\mathcal{S} \subseteq \mathbb{F}^n$ a subspace, the orthogonal complement of $S$ |
| $\mathcal{S} + \mathcal{S}'$ | With $S, S' \subseteq \mathbb{F}^n$ subspaces, the subspace $\{x + x' \mid x \in \mathcal{S}, x' \in \mathcal{S}'\}$ |
| $\mathcal{S} \oplus \mathcal{S}'$ | With $S, S' \subseteq \mathbb{F}^n$ subspaces and $\mathcal{S} \cap \mathcal{S}' = \{0\}$, a direct sum, equal to $\mathcal{S} + \mathcal{S}'$ |
| $\mathcal{K}_k(A, b)$ | With $A \in \mathbb{F}^{n\times n}$ and $b \in \mathbb{F}^n$, the Krylov subspace, $\mathrm{span}(\{b, Ab, \ldots, A^{k-1}b\})$ |
| $A$ | A Hermitian, invertible matrix |
| $A_\omega$ | Equal to $A + \omega I$ |
| $\mathcal{S}$ | A $p-$dimensional subspace of $\mathbb{F}^n$ |
| $p$ | The dimension of $\mathcal{S}$ |
| $\omega_{\min}$ | Equal to $-\lambda_{\min}(A)$ |
| $\mathbf{Ind}_A(\mathcal{S})$ | Equal to $\dim(\mathcal{S} + A\mathcal{S}) - \dim(\mathcal{S})$, also denoted $q$ |
| $q$ | Equal to $\mathbf{Ind}_A(\mathcal{S})$ |
| $x_{b,\omega}$ | The solution to $\min_{x\in\mathcal{S}} \|(A + \omega I)^{-1/2}(b - Ax)\|_2$ |
| $\mathcal{X}_b$ | $\mathrm{span}(\{x_{b,\omega} - x_{b,\mu} \mid \omega, \mu > \omega_{\min}\})$ |
| $\mathcal{X}$ | $\mathrm{span}(\{x_{b,\omega} - x_{b,\mu} \mid b \in \mathbb{F}^n, \omega, \mu > \omega_{\min}\})$ |
| $V$, $V'$, $V''$ | Orthogonal basis to $\mathcal{S}$, $\mathcal{S}^\perp \cap (\mathcal{S} + A\mathcal{S})$ and $(\mathcal{S} + A\mathcal{S})^\perp$. $\begin{bmatrix} V & V' & V'' \end{bmatrix}$ is square and unitary. |
| $c$, $c'$, $c''$ | Equal to $V^*b$, $V'^*b$, $V''^*b$ |
| $T$ | Equal to $V^*AV$ |
| $B$ | Equal to $V'^*AV$ |
| $H$ | Equal to $\begin{bmatrix} T \\ B \end{bmatrix}$ |
| $N$ | A basis for the nullspace of $H$ |
| $d_{b,\omega,\mu}$ | Equal to $V^*(x_{b,\omega} - x_{b,\mu})$ |
| $\mathbf{D}_A(\omega)$ | Equal to $V(V^*AA_\omega^{-1}AV)^{-1}V^*AA_\omega^{-1}$ |
| $\partial\mathbf{D}_A(\omega,\mu)$ | Equal to $\mathbf{D}_A(\omega) - \mathbf{D}_A(\mu)$ |

Table 6.1: Notations used throughout this chapter. Top shows general notations; bottom shows notations specific to our problem as defined in Section 6.2.3

Let $\mathbb{F}$ be $\mathbb{C}$ or $\mathbb{R}$. Let $A \in \mathrm{Sym}(n) \cap \mathrm{GL}(n)$ be a Hermitian invertible matrix, $\mathcal{S} \in \mathrm{Gr}^{\mathbb{F}}(p,n)$ be a $p$-dimensional subspace of $\mathbb{F}^n$ of dimension $1 \leq p \leq n$, and let $\omega_{\min} := -\lambda_{\min}(A)$. For any $b \in \mathbb{F}^n$ and $\omega \in (\omega_{\min}, \infty)$, we define $x_{b,\omega}$ to be the solution to the following minimization problem (the fact that this minimizer exists and is unique is proved in Lemma 6.1):

$$x_{b,\omega} := \mathrm{argmin}_{x \in \mathcal{S}} \|(A + \omega I)^{-\frac{1}{2}}(b - Ax)\|_2, \tag{6.4}$$

and in addition, we also define two subspaces

$$\begin{aligned} \mathcal{X}_b &:= \mathrm{span}(\{x_{b,\omega} - x_{b,\mu} \mid \omega, \mu > \omega_{\min}\}) \\ \mathcal{X} &:= \mathrm{span}(\{x_{b,\omega} - x_{b,\mu} \mid b \in \mathbb{F}^n, \omega, \mu > \omega_{\min}\}) \end{aligned} \tag{6.5}$$

Intuitively, $\mathcal{X}_b$ is the linear subspace containing all solutions for a fixed $b$, varying $\omega$ (up to a constant term). $\mathcal{X}$ contains all solutions as one vary $b$ and $\omega$.

We seek to resolve the following questions: what is the maximum dimension of $\mathcal{X}_b$ and $\mathcal{X}$? Conversely, does the dimensions of $\mathcal{X}_b$ and $\mathcal{X}$ say anything about the quantity $\mathbf{Ind}_A(\mathcal{S})$?

### Characterizing the solution

We start by giving an explicit solution for $x_{b,\omega}$.

**Lemma 6.1.** *Let $A \in Sym(n) \cap GL(n)$, $b \in \mathbb{F}^n$, $\omega_{min} = -\lambda_{min}(A)$, and $\mathcal{S} \in Gr^{\mathbb{F}}(p,n)$. Then for any $\omega \in (\omega_{min}, \infty)$, the problem*

$$\mathrm{argmin}_{x \in \mathcal{S}} \|A_\omega^{-1/2}(b - Ax)\|_2 \tag{6.6}$$

*has a unique solution $x_{b,\omega}$ given by*

$$x_{b,\omega} := V\left(V^* A A_\omega^{-1} A V\right)^{-1} V^* A A_\omega^{-1} b \tag{6.7}$$

*where $A_\omega := A + \omega I$, and $V \in \mathbb{F}^{n \times p}$ is any full rank matrix whose columns span $\mathcal{S}$. (6.7) is well defined as it is independent of the choice of $V$.*

The proof of this lemma is given in Appendix A.2, and we simply note here that $\omega > \omega_{\min}$ ensures that $A_\omega \in \mathrm{P}^+(n)$. Because of the freedom in the choice of $V$ in Lemma 6.1, from now on unless otherwise specified, we will always assume that $V$ is semi-unitary.

The expression for $x_{b,\omega}$ will be studied in some detail in this chapter, and so to make things easier we make the following definition:

**Definition 6.2.** *Using the notation and assumptions of Lemma 6.1, we define two maps* $\mathbf{D}_A : (\omega_{min}, \infty) \to \mathbb{F}^{n\times n}$, *and* $\partial\mathbf{D}_A : (\omega_{min}, \infty) \times (\omega_{min}, \infty) \to \mathbb{F}^{n\times n}$ *as*

$$\mathbf{D}_A(\omega) = V(V^*AA_\omega^{-1}AV)^{-1}V^*AA_\omega^{-1}, \;\; \partial\mathbf{D}_A(\omega,\mu) = \mathbf{D}_A(\omega) - \mathbf{D}_A(\mu). \tag{6.8}$$

*For any* $\omega, \mu \in (\omega_{min}, \infty)$, $\mathbf{D}_A(\omega)$ *and* $\partial\mathbf{D}_A(\omega,\mu)$ *represent linear maps* $\mathbb{F}^n \to \mathcal{S}$, *and are independent of the choice of* $V$ *(the proof of independence is essentially contained in the proof of Lemma 6.1). With those definitions, note that* $x_{b,\omega} = \mathbf{D}_A(\omega)b$ *and* $x_{b,\omega} - x_{b,\mu} = \partial\mathbf{D}_A(\omega,\mu)b$.

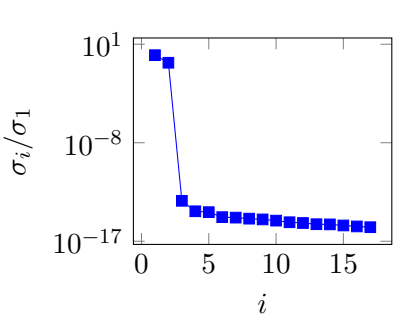
### Motivation

In order to gain some motivation about why we study the problem, we start with the following observation, that holds under the assumptions mentioned above.

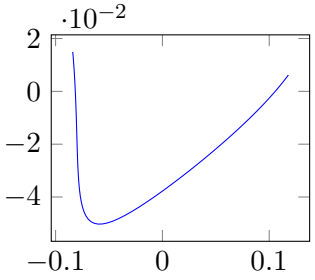
**Lemma 6.2.** *If* $\mathcal{S}$ *is* $A-$*invariant,* $x_{b,\omega}$ *given by (6.7) is independent of* $\omega$.

*Proof.* Since $A$ is invertible and $\mathbf{Ind}_A(\mathcal{S}) = 0$, $A\mathcal{S} = \mathcal{S}$; so applying the spectral theorem to the restriction map $A|_{\mathcal{S}}$ (which is Hermitian because $A$ is), we can conclude that $\mathcal{S}$ is spanned by eigenvectors of $A$. Thus, we can choose $V$ such that $AV = V\Lambda$, where the columns of $V$ are eigenvectors of $A$, and $\Lambda$ is a diagonal matrix with real entries (the eigenvalues). Thus for any $\omega \in \mathbb{R}$, $A_\omega V = V(\Lambda+\omega I)$, while the definition of $A_\omega^{-1}$ shows that $A_\omega^{-1}V = V(\Lambda+\omega I)^{-1}$ for all $\omega \in (\omega_{\min}, \infty)$, where the bounds on $\omega$ ensure that $A_\omega \in \mathrm{P}^+(n)$ so that $A_\omega^{-1}$ is well-defined. Plugging into (6.7) gives $x_{b,\omega} = V\left(\Lambda(\Lambda+\omega I)^{-1}\Lambda\right)^{-1}\Lambda(\Lambda+\omega I)^{-1}V^*b = V\Lambda^{-1}V^*b$. □

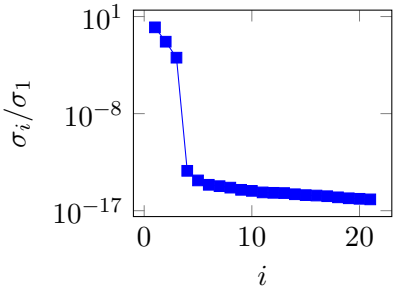
Lemma 6.2 suggests that when $\mathbf{Ind}_A(\mathcal{S}) = 0$, $\dim(\mathcal{X}_b) = 0$ for all $b \in \mathbb{F}^n$. A natural question that arises then is what happens when $\mathbf{Ind}_A(\mathcal{S}) > 0$. As indicated in Section 6.1, a simple consequence of the results in [89] is that in the case $\mathbb{F} = \mathbb{R}$, if $B \in \mathbb{R}^{n\times n}$ is an SPD matrix, $b \in \mathbb{R}^n$, and $\mathcal{S} = \mathcal{K}_k(B, b)$ is a real Krylov subspace, the set of solutions $\{x_{b,\omega} \mid \omega \geq 0\}$ with $x_{b,\omega}$ defined as the solution to (6.2) belong to a 1-dimensional affine subspace. But for a Krylov subspace $\mathcal{S} = \mathcal{K}_k(A, b)$ that is not $A-$invariant, $\mathbf{Ind}_A(\mathcal{S}) = 1$. Based on these two known results, we are faced with the possibility that the conjecture $\dim(\mathcal{X}_b) \leq \mathbf{Ind}_A(\mathcal{S})$ for all $b \in \mathbb{F}^n$, might be true for $\mathbb{F} = \mathbb{C}$ or $\mathbb{R}$.



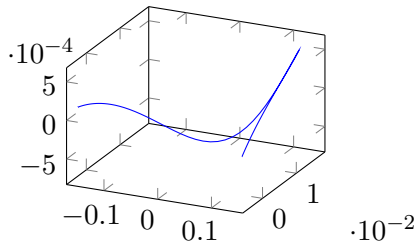

(a) Singular values of $Y$ for $\mathbf{Ind}_A(\mathcal{S}) = 2$




(b) Principal components of $x_{b,\omega_j}$ for $\mathbf{Ind}_A(\mathcal{S}) = 2$




(c) Singular values of $Y$ for $\mathbf{Ind}_A(\mathcal{S}) = 3$




(d) Principal components of $x_{b,\omega_j}$ for $\mathbf{Ind}_A(\mathcal{S}) = 3$

Figure 6.1: Illustration of the low dimensionality of the affine subspace $\mathcal{X}_b = \mathrm{Aff}(\{x_{b,\omega} \mid \omega \geq 0\})$. Left plots show the singular values of the centered matrix $Y$ computed as $Y_{ij} = X_{ij} - K^{-1} \sum_k X_{ik}$ where $X = [x_{b,\omega_1} \ldots x_{b,\omega_K}]$. A sharp drop in the singular values indicates that the set $\{x_{b,\omega_j}\}_{j=1}^K$ lives in a low dimensional affine subspace. Right plots show the projection of $\{x_{b,\omega_j}\}_{j=1}^K$ over that low dimensional subspace.

We now describe a numerical experiment that also illustrates and confirms our intuition. Working over $\mathbb{F} = \mathbb{R}$, given a positive matrix $A \in \mathbb{R}^{N \times N}$ (built as the finite difference discretization with a 5-points stencil of a Poisson equation on a square domain, with $N = 529$), we create two experiments by building $\mathcal{S}$ as the sum of two (resp. three) real Krylov subspaces, i.e. $\mathcal{K}_{11}(A, b_1) + \mathcal{K}_6(A, b_2)$ (resp. $\mathcal{K}_{11}(A, c_1) + \mathcal{K}_6(A, c_2) + \mathcal{K}_4(A, c_3)$). The vectors $b_1, b_2, c_1, c_2, c_3 \in \mathbb{R}^N$ were chosen as random Gaussian vectors, but such that $\mathbf{Ind}_A(\mathcal{S}) = 2$ (resp. 3), and $\dim(\mathcal{S}) = 17$ (resp. 21). The vector $b \in \mathbb{R}^N$ was also initialized as a random Gaussian vector.

To check the dimension of the solution set $\mathcal{X}_b$, we then build a matrix $X = [x_{b,\omega_1} \ldots x_{b,\omega_K}] \in \mathbb{R}^{N \times K}$, with the columns of $X$ computed using (6.7) and $K = 200$, and where $\omega_j =$

$10^{-3+6(j-1)/(K-1)}$, for all $1 \leq j \leq K$. We then perform a principal component analysis on $X$: we compute and subtract the mean across each $N$ dimensions, building $Y$ such that $Y_{ij} = X_{ij} - K^{-1}\sum_k X_{ik}$, for all $1 \leq i \leq N,\ 1 \leq j \leq K$. Figure 6.1a (resp. Figure 6.1c) shows the singular values of $Y$ in the $\mathbf{Ind}_A(\mathcal{S}) = 2$ (resp. $\mathbf{Ind}_A(\mathcal{S}) = 3$) cases. The sharp drop at the third (resp. fourth) singular value indicates that $Y$ is rank two (resp. three), which indicates that $\mathcal{X}_b$ may belong to a low dimensional affine subspace of dimension 2 (resp. 3). Figure 6.1b (resp. Figure 6.1d) shows the solution set $\{x_{b,\omega_j}\}_{j=1}^K$ projected over the leading two (resp. three) eigenvectors of $Y$ for the two experiments.

### 6.2.4 Properties of the index of invariance

We now prove some facts about the index of invariance, defined previously in Definition 6.1. Additional facts can be found in Appendix A.1. We start with two lemmas that characterize the relationship between the index of invariance and bases of the subspaces involved in its definition. Those results lead to the definition of a precise block-decomposition of $A$, critical to the results of this chapter.

This first result shows that there exist semi-unitary matrices $V$, spanning $\mathcal{S}$, and $V'$, spanning $\mathcal{S} + A\mathcal{S}$ together with $V$. The proof is elementary and provided in Appendix A.1.

**Lemma 6.3.** *Let $A \in \mathbb{F}^{n\times n}$, $\mathcal{S} \in Gr^{\mathbb{F}}(p,n)$ and $q = \boldsymbol{Ind}_A(\mathcal{S})$. Then*

*(i)* $\boldsymbol{Ind}_A(\mathcal{S}) \leq \min\{dim(\mathcal{S}),\ n - dim(\mathcal{S})\} \leq \lfloor n/2 \rfloor$.

*(ii) There exists semi-unitary $V \in \mathbb{F}^{n\times p}$ such that $Range(V) = \mathcal{S}$, and there exists $V' \in \mathbb{F}^{n\times q}$ such that $\begin{bmatrix} V & V' \end{bmatrix}$ is semi-unitary, $Range(\begin{bmatrix} V & V' \end{bmatrix}) = \mathcal{S} + A\mathcal{S}$, and $Range(V') = \mathcal{S}^{\perp} \cap (\mathcal{S} + A\mathcal{S})$.*

This second result now connects $V$ and $V'$ to $A$ and $\mathbf{Ind}_A(\mathcal{S})$.

**Lemma 6.4.** *Let $A \in \mathbb{F}^{n\times n}$, $\mathcal{S} \in Gr^{\mathbb{F}}(p,n)$. Let $\begin{bmatrix} V & V' \end{bmatrix}$ be semi-unitary such that $Range(V) = \mathcal{S}$, and $Range(\begin{bmatrix} V & V' \end{bmatrix}) = \mathcal{S} + A\mathcal{S}$. The following are equivalent:*

*(i)* $\boldsymbol{Ind}_A(\mathcal{S}) = q$.

*(ii) There exist $T \in \mathbb{F}^{p\times p}$, $B \in \mathbb{F}^{q\times p}$ and $\mathrm{rank}(B) = q$, such that $AV = VT + V'B$, and $T, B$ are uniquely determined by $A, V, V'$.*

*Proof.* Notice that from Lemma 6.3(ii), $V$ and $V'$ always exist. Let $\mathbf{Ind}_A(\mathcal{S}) = q$. We first prove (i)$\rightarrow$(ii). Since $\text{Range}(AV) = A\mathcal{S} \subseteq \mathcal{S} + A\mathcal{S} = \text{Range}(\begin{bmatrix} V & V' \end{bmatrix})$, we have $AV = VT + V'B$, for some $T \in \mathbb{F}^{p\times p}$ and $B \in \mathbb{F}^{q\times p}$, which are uniquely determined because $\begin{bmatrix} V & V' \end{bmatrix}$ is full rank. From Lemma 6.3(i) we have $q \leq p$. Now assume $B$ is not of full rank $q$. Then one can decompose $B$ (such as using the singular value decomposition) as $B = UW$, where $U \in \mathbb{F}^{q\times r}$, $W \in \mathbb{F}^{r\times p}$ with $r < q$. Then $AV = VT + (V'U)W$ from which it follows that $\text{Range}(AV) \subseteq \text{Range}(\begin{bmatrix} V & V'U \end{bmatrix})$, where $\text{rank}(V'U) \leq r < q$. But $\mathcal{S} = \text{Range}(V)$, and so $\mathcal{S} + A\mathcal{S} \subseteq \text{Range}(\begin{bmatrix} V & V'U \end{bmatrix}) = \mathcal{S} + \text{Range}(V'U)$. This implies that $\dim(\mathcal{S} + A\mathcal{S}) \leq \dim(\mathcal{S}) + r$, which is a contradiction.

Now suppose (ii) holds. Since $\mathcal{S} = \text{Range}(V)$, and $A\mathcal{S} = \text{Range}(AV)$, by assumption it follows that $\mathcal{S} + A\mathcal{S} = \{Vx + (VT + V'B)y \mid x, y \in \mathbb{F}^p\} = \{V(x + Ty) + V'By \mid x, y \in \mathbb{F}^p\} = \{Vw + V'z \mid w \in \mathbb{F}^p,\ z \in \mathbb{F}^q\} = \text{Range}(\begin{bmatrix} V & V' \end{bmatrix})$ (the second last equality follows because $B$ is full rank). Since $\begin{bmatrix} V & V' \end{bmatrix}$ is semi-unitary, we conclude that $\dim(\mathcal{S} + A\mathcal{S}) = p + q = \dim(\mathcal{S}) + q$. □

Lemma 6.4 has an important consequence that we state next, which will play a key role later in the proof of the main theorem of this chapter.

**Corollary 6.1.** *Let $A \in \mathbb{F}^{n\times n}$, $\mathcal{S} \in Gr^{\mathbb{F}}(p, n)$, and $\boldsymbol{Ind}_A(\mathcal{S}) = q$. Let $V \in \mathbb{F}^{n\times p}$, $V' \in \mathbb{F}^{n\times q}$, and $V'' \in \mathbb{F}^{n\times(n-p-q)}$ be such that $\begin{bmatrix} V & V' & V'' \end{bmatrix}$ is unitary, $\mathcal{S} = \text{Range}(V)$, and $\mathcal{S} + A\mathcal{S} = \text{Range}(\begin{bmatrix} V & V' \end{bmatrix})$. Then $A$ has the following block decomposition*

$$\begin{bmatrix} V^* \\ V'^* \\ V''^* \end{bmatrix} A \begin{bmatrix} V & V' & V'' \end{bmatrix} = \begin{array}{c} \begin{array}{ccc} p & q & \end{array} \\ \left[\begin{array}{c|c|c} T & P & Q \\ \hline B & C & R \\ \hline 0 & D & E \end{array}\right] \end{array} \begin{array}{c} p \\ , q \end{array} \tag{6.9}$$

*where $T \in \mathbb{F}^{p\times p}$, $C \in \mathbb{F}^{q\times q}$, $E \in \mathbb{F}^{(n-p-q)\times(n-p-q)}$, with the shapes of the other blocks being compatible, and $B$ is of full rank $q$. Additionally*

*(i) If $A \in GL(n)$, then $H := \begin{bmatrix} T \\ B \end{bmatrix}$ is of full rank $p$.*

*(ii) If $A \in Sym(n)$, one has $P = B^*$, $R = D^*$, $Q = 0$, and $T, C, E$ Hermitian.*

*Proof.* The decomposition follows from Lemma 6.4 (which also gives $\text{rank}(B) = q$), by noting that $V''^*(AV) = 0$, using the unitarity of $\begin{bmatrix} V & V' & V'' \end{bmatrix}$ and $AV = VT + V'B$. For

(i), let $\bar{A}$ be the right-hand-side of (6.9); so $A \in \mathrm{GL}(n)$ implies $\bar{A} \in \mathrm{GL}(n)$, which means the first $p$ columns of $\bar{A}$ are linearly independent. But if $\mathrm{rank}(H) < p$, the first $p$ columns of $\bar{A}$ are linearly dependent, giving a contradiction. For (ii), note that when $A \in \mathrm{Sym}(n)$, both sides of (6.9) are Hermitian, and so the conclusion follows. □

When $A \in \mathrm{Sym}(n)$, the decomposition given by Corollary 6.1 will be called the *tridiagonal block decomposition*. Note that (i) this decomposition exists regardless of whether $A$ is invertible or SPD, (ii) even if $A$ is invertible, the diagonal blocks $T$, $C$ and $E$ need not be, (iii) if however $A$ is SPD, then $T$, $C$ and $E$ are in fact SPD, but $D$ need not be full rank[1]. We also note that this decomposition is similar to the block Lanczos decomposition [75, page 567] as used in block Krylov methods (amongst many, [117, 137, 136, 55]).

It is worth noting some special cases. Consider the case when $\mathcal{S}$ is $A-$invariant, and $A \in \mathrm{Sym}(n)$. Then $\mathbf{Ind}_A(\mathcal{S}) = 0$, and so in the tridiagonal block decomposition (6.9), $V'$ has 0 columns (i.e., $q = 0$), and we can simply write

$$\begin{bmatrix} V^* \\ V''^* \end{bmatrix} A \begin{bmatrix} V & V'' \end{bmatrix} = \begin{bmatrix} T & 0 \\ 0 & E \end{bmatrix}. \tag{6.10}$$

This is the block-diagonal Schur decomposition of a Hermitian matrix for a given invariant subspace [75, page 443]. Consider similarly the case when $\mathcal{S} = \mathcal{K}_p(A, b)$, such that $\mathcal{S}$ is not $A-$invariant, and so $\mathbf{Ind}_A(\mathcal{S}) = 1$. We then know that $B \in \mathbb{F}^{1\times p}$ is rank-1. In fact, if we build $V$ by the Arnoldi process (that is the first $k$ columns of $V$ span $\mathcal{K}_k(A, b)$ for $1 \leq k \leq p$), then $B = V'^* AV = \beta e_p^*$, where $\beta \neq 0$ with $(e_p)_i = 0$ for $i < p$ and $(e_p)_p = 1$.

## 6.3 Proof of the main result

The main result of this chapter is the following theorem which we prove after. For completeness, we also redefine every quantity needed.

**Theorem 6.1.** *Let $\mathbb{F}$ denote the field $\mathbb{C}$ or $\mathbb{R}$. Let $A \in Sym(n) \cap GL(n)$ be an $n \times n$ invertible Hermitian matrix over $\mathbb{F}$, $\mathcal{S} \subseteq \mathbb{F}^n$ a $p$-dimensional subspace, and $b \in \mathbb{F}^n$. Define $\omega_{min} := -\lambda_{min}(A)$, $q := dim(\mathcal{S} + A\mathcal{S}) - dim(\mathcal{S})$, and for all $\omega \in (\omega_{min}, \infty)$*

$$x_{b,\omega} := \mathrm{argmin}_{x\in\mathcal{S}} \|(A + \omega I)^{-\frac{1}{2}}(Ax - b)\|_2.$$

[1] For example, if $\mathbf{Ind}_A(\mathcal{S} + A\mathcal{S}) = 0$ (i.e., $\mathcal{S} + A\mathcal{S}$ is invariant), then $D = 0$.

*Then there exists a $q$-dimensional subspace $\mathcal{Y} \subseteq \mathbb{F}^n$, independent of $b$, such that $x_{b,\omega} - x_{b,\mu} \in \mathcal{Y}$ for all $\omega, \mu \in (\omega_{min}, \infty)$. If $V \in \mathbb{F}^{n\times p}$, $V' \in \mathbb{F}^{n\times q}$ are chosen such that $\begin{bmatrix} V & V' \end{bmatrix}$ is semi-unitary, Range$(V) = \mathcal{S}$, and Range$(\begin{bmatrix} V & V' \end{bmatrix}) = \mathcal{S} + A\mathcal{S}$, then $\mathcal{Y} =$ Range$(V(H^*H)^{-1}B^*)$ (not depending on the choice of $V, V'$), where $H^* = \begin{bmatrix} T & B^* \end{bmatrix}$, with $T = V^*AV$, and $B = V'^*AV$.*

Let $\mathbf{Ind}_A(\mathcal{S}) = q$. Using the tridiagonal block decomposition (Corollary 6.1), we will choose $V' \in \mathbb{F}^{n\times q}$, and $V'' \in \mathbb{F}^{n\times(n-p-q)}$ such that $\begin{bmatrix} V & V' & V'' \end{bmatrix}$ is unitary, $\mathcal{S} + A\mathcal{S} =$ Range$(\begin{bmatrix} V & V' \end{bmatrix})$, and

$$\begin{bmatrix} V^* \\ V'^* \\ V''^* \end{bmatrix} A \begin{bmatrix} V & V' & V'' \end{bmatrix} = \begin{bmatrix} T & B^* & 0 \\ B & C & D^* \\ 0 & D & E \end{bmatrix}, \tag{6.11}$$

where $T \in \mathbb{F}^{p\times p}$, $C \in \mathbb{F}^{q\times q}$, $E \in \mathbb{F}^{(n-p-q)\times(n-p-q)}$ are all Hermitian, and the shapes of the other blocks are compatible, and we denote $H := \begin{bmatrix} T \\ B \end{bmatrix} \in \mathbb{F}^{(p+q)\times p}$ which is of full rank $p$. We let $b = Vc + V'c' + V''c''$, for some $c \in \mathbb{F}^p$, $c' \in \mathbb{F}^q$, and $c'' \in \mathbb{F}^{n-p-q}$, the representation being unique for the given choice of $V, V'$, and $V''$, and existing for any $b \in \mathbb{F}^n$, because Range$(\begin{bmatrix} V & V' & V'' \end{bmatrix}) = \mathbb{F}^n$.

To simplify the presentation of this section, we make a few observations. Let $A_\omega := A + \omega I$. Using (6.11) and the unitarity of $\begin{bmatrix} V & V' & V'' \end{bmatrix}$, we obtain

$$\begin{bmatrix} V^* \\ V'^* \\ V''^* \end{bmatrix} A_\omega \begin{bmatrix} V & V' & V'' \end{bmatrix} = \begin{bmatrix} T+\omega I & B^* & 0 \\ B & C+\omega I & D^* \\ 0 & D & E+\omega I \end{bmatrix}, \tag{6.12}$$

and since $A_\omega \in \mathrm{P}^+(n)$ for $\omega > \omega_{\min}$, the right-hand side of (6.12) is also SPD. Thus in particular $E + \omega I$ is SPD, which allows us to define $F_\omega \in \mathbb{F}^{q\times q}$ and $G_\omega \in \mathbb{F}^{(p+q)\times(p+q)}$ for any $\omega \in (\omega_{\min}, \infty)$, as follows

$$F_\omega := D^*(E+\omega I)^{-1}D, \text{ and } G_\omega := \begin{bmatrix} T & B^* \\ B & C - F_\omega \end{bmatrix} + \omega I. \tag{6.13}$$

The positivity of $E + \omega I$ directly ensures that $F_\omega \in \mathrm{P}^+(q)$, while $G_\omega \in \mathrm{P}^+(p+q)$ as it is

the Schur complement of the $E+\omega I$ block of the right-hand side of (6.12). Finally, we note a couple of key identities that follow from the 2-by-2 block matrix inversion formula [110], whenever $\omega > \omega_{\min}$:

$$\begin{aligned} G_\omega^{-1} &= \begin{bmatrix} V^* \\ V'^* \end{bmatrix} A_\omega^{-1} \begin{bmatrix} V & V' \end{bmatrix}, \\ -G_\omega^{-1} \begin{bmatrix} 0 \\ D^*(E+\omega I)^{-1} \end{bmatrix} &= \begin{bmatrix} V^* \\ V'^* \end{bmatrix} A_\omega^{-1} V''. \end{aligned} \tag{6.14}$$

We are now ready to prove the following lemma, which is the first step in proving Theorem 6.1.

**Lemma 6.5.** *Define $d_{b,\omega,\mu} := V^*(x_{b,\omega} - x_{b,\mu}) \in \mathbb{F}^p$, whenever $\omega, \mu \in (\omega_{min}, \infty)$, and let $N \in \mathbb{F}^{(p+q)\times q}$ be any full rank matrix[2] whose columns span the nullspace of $H^*$. Also define*

$$z_{b,\omega,\mu}(t) := D^*(E+\omega I)^{-1}c'' - D^*(E+\mu I)^{-1}c'' + \begin{bmatrix} B & C - F_\mu \end{bmatrix} Nt. \tag{6.15}$$

*Then there exists a unique $d \in \mathbb{F}^p$ and $t \in \mathbb{F}^q$ satisfying the system of equations*

$$\begin{cases} H^* G_\omega^{-1} \left( Hd + \mu N t + \begin{bmatrix} 0 \\ z_{b,\omega,\mu}(t) \end{bmatrix} \right) = 0 \\ Nt = G_\mu^{-1} \left( H(H^* G_\mu^{-1} H)^{-1} H^* G_\mu^{-1} - I \right) \begin{bmatrix} c \\ c' - D^*(E+\mu I)^{-1} c'' \end{bmatrix}, \end{cases} \tag{6.16}$$

*where the solution $d$ satisfies $d = d_{b,\omega,\mu}$.*

*Proof.* To show uniqueness, suppose $(d,t), (d',t') \in \mathbb{F}^p \times \mathbb{F}^q$ are two solutions of (6.16). Then from the second equation we get $N(t-t') = 0$; but since $N$ is of full rank $q$, we have $t = t'$. The first equation then gives $H^* G_\omega^{-1} H(d-d') = 0$. Now as $G_\omega \in \mathrm{P}^+(p+q)$, we have $H^* G_\omega^{-1} H \in \mathrm{P}^+(p)$, which gives $d = d'$ proving uniqueness.

To prove the existence of a solution to (6.16), we start by expressing $d_{b,\omega,\mu}$ using (6.7), and obtain $d_{b,\omega,\mu} = (V^* A A_\omega^{-1} A V)^{-1} V^* A A_\omega^{-1} b - (V^* A A_\mu^{-1} A V)^{-1} V^* A A_\mu^{-1} b$, which after multiplying both sides by $V^* A A_\omega^{-1} A V$ and rearranging is equivalent to

$$V^* A A_\omega^{-1} \left\{ A V d_{b,\omega,\mu} - b + A V (V^* A A_\mu^{-1} A V)^{-1} V^* A A_\mu^{-1} b \right\} = 0. \tag{6.17}$$

[2] Existence of $N$ is guaranteed as $\mathrm{rank}(H^*) = \mathrm{rank}(H) = p$, hence the nullspace of $H^*$ has dimension $q$.

Next observe that as $A$ is Hermitian, we can express $V^*AA_\mu^{-1}AV$ and $V^*AA_\mu^{-1}b$, as $(AV)^*A_\mu^{-1}(AV)$ and $(AV)^*A_\mu^{-1}b$ respectively, and so firstly using the fact that $AV = \begin{bmatrix} V & V' \end{bmatrix} H$ from (6.12), and secondly using the identities in (6.14) one obtains

$$V^*AA_\mu^{-1}AV = H^*G_\mu^{-1}H, \;\; V^*AA_\mu^{-1}b = H^*G_\mu^{-1}\begin{bmatrix} c \\ c' - D^*(E+\mu I)^{-1}c'' \end{bmatrix}, \tag{6.18}$$

with similar expressions holding for $\mu$ replaced by $\omega$. Using (6.18) one can then equivalently write (6.17) as

$$\begin{aligned} &H^*G_\omega^{-1}\left\{ Hd_{b,\omega,\mu} + \begin{bmatrix} 0 \\ D^*(E+\omega I)^{-1}c'' - D^*(E+\mu I)^{-1}c'' \end{bmatrix} \right\} \\ &+ H^*G_\omega^{-1}\left( H\left(H^*G_\mu^{-1}H\right)^{-1}H^*G_\mu^{-1} - I \right)\begin{bmatrix} c \\ c' - D^*(E+\mu I)^{-1}c'' \end{bmatrix} = 0. \end{aligned} \tag{6.19}$$

Now let $s := \left( H\left(H^*G_\mu^{-1}H\right)^{-1}H^*G_\mu^{-1} - I \right)\begin{bmatrix} c \\ c' - D^*(E+\mu I)^{-1}c'' \end{bmatrix}$. Then it follows that $H^*G_\mu^{-1}s = 0$, or equivalently $G_\mu^{-1}s = Nt$ for some $t \in \mathbb{F}^q$, as the columns of $N$ form a basis for the nullspace of $H^*$. But this then implies that

$$s = G_\mu Nt = \mu Nt + \begin{bmatrix} 0 \\ \begin{bmatrix} B & C - F_\mu \end{bmatrix} Nt \end{bmatrix}, \tag{6.20}$$

using the fact that $\begin{bmatrix} T & B^* \end{bmatrix} N = H^*N = 0$. Plugging $s$ back into (6.19) then shows that $(d_{b,\omega,\mu}, t)$ is a solution of (6.16), finishing the proof. □

We now prove this chapter's main result, Theorem 6.1.

*Proof of Theorem 6.1.* From Lemma 6.5 $(d_{b,\omega,\mu}, t)$ is the unique solution of (6.16), for some $t \in \mathbb{F}^q$. Hence there exist $t' \in \mathbb{F}^q$ such that

$$Hd_{b,\omega,\mu} + \mu Nt + \begin{bmatrix} 0 \\ z_{b,\omega,\mu}(t) \end{bmatrix} = G_\omega Nt' = \omega Nt' + \begin{bmatrix} 0 \\ \begin{bmatrix} B & C - F_\omega \end{bmatrix} Nt' \end{bmatrix}, \tag{6.21}$$

where we used the fact that $H^*N = \begin{bmatrix} T & B^* \end{bmatrix} N = 0$. So $Hd_{b,\omega,\mu} = N(\omega t' - \mu t) +$

$\begin{bmatrix} 0 \\ z'_{b,\omega,\mu}(t,t') \end{bmatrix}$ where $z'_{b,\omega,\mu}(t,t') = \begin{bmatrix} B & C - F_\omega \end{bmatrix} Nt' - z_{b,\omega,\mu}(t)$. Since $H$ is full column rank, $H^*H$ is invertible and $H^*Hd_{b,\omega,\mu} = H^* \begin{bmatrix} 0 \\ z'_{b,\omega,\mu}(t,t') \end{bmatrix}$, and we conclude that

$$d_{b,\omega,\mu} = (H^*H)^{-1}B^* z'_{b,\omega,\mu}(t,t'). \tag{6.22}$$

Noticing that $x_{b,\omega}, x_{b,\mu} \in \mathcal{S}$, we have $Vd_{b,\omega,\mu} = x_{b,\omega} - x_{b,\mu}$, and (6.22) then gives $x_{b,\omega} - x_{b,\mu} \in \text{Range}(V(H^*H)^{-1}B^*)$, for all $\omega, \mu > \omega_{\min}$ and $b \in \mathbb{F}^n$. Since $B^*$ has full column rank $q$, and $V(H^*H)^{-1}$ has full column rank $p$, $V(H^*H)^{-1}B^*$ has full column rank $q$[3], and so defining $\mathcal{Y} := \text{Range}(V(H^*H)^{-1}B^*)$ gives $\dim(\mathcal{Y}) = q$.

The theorem is proved if we can show that $\mathcal{Y}$ does not depend on the choice of $V, V'$. So suppose that $\overline{V} \in \mathbb{F}^{n\times p}$, $\overline{V'} \in \mathbb{F}^{n\times q}$ is a different choice of semi-unitary matrices such that $\begin{bmatrix} \overline{V} & \overline{V'} \end{bmatrix}$ is semi-unitary, $\text{Range}(\overline{V}) = \mathcal{S}$, and $\text{Range}(\begin{bmatrix} \overline{V} & \overline{V'} \end{bmatrix}) = \mathcal{S} + A\mathcal{S}$. Let $\overline{T} := \overline{V}^* A \overline{V}$, $\overline{B}^* := \overline{V}^* A \overline{V'}$, and $\overline{H}^* := \begin{bmatrix} \overline{T} & \overline{B}^* \end{bmatrix}$ be analogously defined. Then there exists $U \in \text{U}(p)$ and $U' \in \text{U}(q)$, such that $V = \overline{V}U$ and $V' = \overline{V'}U'$. A simple computation then shows that $V(H^*H)^{-1}B^* = (\overline{V}(\overline{H}^*\overline{H})^{-1}\overline{B}^*)U'$, and as $U'$ is unitary this shows that $\text{Range}(V(H^*H)^{-1}B^*) = \text{Range}(\overline{V}(\overline{H}^*\overline{H})^{-1}\overline{B}^*)$. □

This proof immediately gives us bounds on the dimensions of the affine subspace $\mathcal{X}_b$ and the subspace $\mathcal{X}$, as stated in the next corollary.

**Corollary 6.2.** *The sets $\mathcal{X}_b$ and $\mathcal{X}$ introduced in (6.5) satisfy*

*(i) For all $b \in \mathbb{F}^n$, $\mathcal{X}_b \subseteq$ Range$(V(H^*H)^{-1}B^*)$ and $\dim(\mathcal{X}_b) \leq \boldsymbol{Ind}_A(\mathcal{S})$.*

*(ii) $\mathcal{X} \subseteq$ Range$(V(H^*H)^{-1}B^*)$ and $\dim(\mathcal{X}) \leq \boldsymbol{Ind}_A(\mathcal{S})$.*

*Proof.* Both (i) and (ii) follow by applying Theorem 6.1, because

$$x_{b,\omega} - x_{b,\mu} \in \text{Range}(V(H^*H)^{-1}B^*),$$

for all $\omega, \mu > \omega_{\min}$, and for all $b \in \mathbb{F}^n$. □

As mentioned in Section 6.1, the particular case, when $\mathbb{F} = \mathbb{R}$ and $c = b$, follows from the results proved in [89]. One can ask whether the bounds in Corollary 6.2 are tight, or

[3] Multiplication of a $\mathbb{F}$ valued matrix from the left by a full column rank matrix does not change its rank.

whether they can be improved. This is the topic of the next section.

## 6.4 Tightness of the bounds

In this section, we explore the converse of the main theorem. In Section 6.4.1, we explore how tight is the bound $\dim(\mathcal{X}_b) \leq \mathbf{Ind}_A(\mathcal{S})$ for fixed $b \in \mathbb{F}^n$. In Section 6.4.2 we formulate some sufficient conditions under which $\dim(\mathcal{X}) = \mathbf{Ind}_A(\mathcal{S})$.

### 6.4.1 Bounds on $\dim(\mathcal{X}_b)$

The result that motivated this section is the following observation.

**Lemma 6.6.** *The following conditions are equivalent:*

*(i)* $\boldsymbol{Ind}(A, \mathcal{S}) = 0$.

*(ii) For all* $b \in \mathbb{F}^n$ *and for all* $\mu, \omega > \omega_{min}$, $x_{b,\mu} = x_{b,\omega}$.

*(iii) There exists distinct* $\mu, \omega > \omega_{min}$ *such that for all* $b \in \mathbb{F}^n$, $x_{b,\mu} = x_{b,\omega}$.

*Proof.* (i) $\rightarrow$ (ii) was proved in Lemma 6.2, and (ii) $\rightarrow$ (iii) is straightforward.

We prove (iii) $\rightarrow$ (i). If $\mathcal{S} = \mathbb{F}^n$ we have $\mathbf{Ind}(A, \mathbb{F}^n) = 0$, so assume that $\mathcal{S} \neq \mathbb{F}^n$. Pick any $b \in A_\omega A^{-1}\mathcal{S}^\perp$. Then $AA_\omega^{-1}b \in \mathcal{S}^\perp$, and so $V^*AA_\omega^{-1}b = 0$. Since $x_{b,\mu} = x_{b,\omega}$, it follows using (6.7) that $V^*AA_\mu^{-1}b = 0$, or $AA_\mu^{-1}b \in \mathcal{S}^\perp$. Now $AA_\omega^{-1} : A_\omega A^{-1}\mathcal{S}^\perp \rightarrow \mathcal{S}^\perp$ is an isomorphism as both $A, A_\omega \in \mathrm{GL}(n)$, and notice that $AA_\mu^{-1} = A_\mu^{-1}A_\omega(AA_\omega^{-1})$ using the fact that $A, A_\mu, A_\omega \in \mathrm{Sym}(n)$; so we have in fact proved that $\mathbf{Ind}(A_\mu^{-1}A_\omega, \mathcal{S}^\perp) = 0$. Finally notice that $A_\mu^{-1}A_\omega = I + (\omega - \mu)A_\mu^{-1}$, which means that for any $x \in \mathcal{S}^\perp$, $x + (\omega - \mu)A_\mu^{-1}x \in \mathcal{S}^\perp$, and so $A_\mu^{-1}x \in \mathcal{S}^\perp$ (as $\mu \neq \omega$). Thus we conclude that $\mathbf{Ind}(A_\mu^{-1}, \mathcal{S}^\perp) = 0$, and by applying Lemma A.2(iii), (iv), and (i) successively, we get $\mathbf{Ind}(A, \mathcal{S}) = 0$. □

The equivalence of the conditions (i), (ii), and (iii) of Lemma 6.6, leads to the following corollary.

**Corollary 6.3.** *The following statements are true.*

*(i)* $\boldsymbol{Ind}_A(\mathcal{S}) = 0$ *if and only if* $dim(\mathcal{X}) = 0$.

*(ii)* $\boldsymbol{Ind}_A(\mathcal{S}) = 0$ *if and only if* $dim(\mathcal{X}_b) = 0$, *for all* $b \in \mathbb{F}^n$.

*(iii) The map* $\mathbf{D}_A$ *is a constant map if* $\boldsymbol{Ind}_A(\mathcal{S}) = 0$*, and injective otherwise.*

*Proof.* Both (i) and (ii) follow from Lemma 6.6. For (iii), Lemma 6.2 implies that $\mathbf{D}_A$ is a constant map if $\mathbf{Ind}_A(\mathcal{S}) = 0$, while if $\mathbf{Ind}_A(\mathcal{S}) > 0$ and $\mathbf{D}_A$ is not injective, there exists distinct $\mu, \omega > \omega_{\min}$, such that $\mathbf{D}_A(\mu) = \mathbf{D}_A(\omega)$ implying that $x_{b,\mu} = x_{b,\omega}$ for all $b \in \mathbb{F}^n$, thereby contradicting Lemma 6.6. □

An interesting consequence of Corollary 6.3 is that when $\mathbf{Ind}_A(\mathcal{S}) = 1$, there must exist $b \in \mathbb{F}^n$ such that $\dim(\mathcal{X}_b) = 1$, since we know that $\dim(\mathcal{X}_b) \leq 1$ by Theorem 6.1. One can then ask whether this pattern holds in general, that is if $\mathbf{Ind}_A(\mathcal{S}) \geq 1$, whether there always exists $b \in \mathbb{F}^n$ such that $\dim(\mathcal{X}_b) = \mathbf{Ind}_A(\mathcal{S})$. However, this turns out to not be true as shown by the following example, which shows that one can have cases where $\dim(\mathcal{X}_b) \leq 1$ for all $b \in \mathbb{F}^n$, even though $\mathbf{Ind}_A(\mathcal{S})$ is arbitrarily large.

**Example 6.1.** *For* $\alpha \in \mathbb{R} \setminus \{1, -1\}$*, let* $p = q \geq 1$*,* $n = p + q$ *and consider*

$$A = \begin{bmatrix} \alpha I & I \\ I & \alpha I \end{bmatrix} \in \mathbb{F}^{n \times n} \tag{6.23}$$

*with* $\mathcal{S} = span(\{e_1, \ldots, e_p\})$*, where* $e_k \in \mathbb{F}^n$ *is given by* $(e_k)_i = \delta_{ik}$*. Notice that* $\det(A) = (\alpha^2 - 1)^p$*, so* $A \in GL(n)$*. Furthermore, for* $\alpha > 1$*,* $A \in P^+(n)$ *since its eigenvalues are given by* $\alpha \pm 1$*. With* $k(\alpha, \omega) := (\alpha + \omega)^2 - 1$ *(note that* $\omega_{min} = 1 - \alpha$*, so* $k(\alpha, \omega) > 0$ *for* $\omega > \omega_{min}$*), we find*

$$\begin{aligned} A_\omega^{-1} &= k(\alpha, \omega)^{-1} \begin{bmatrix} (\alpha + \omega) I & -I \\ -I & (\alpha + \omega) I \end{bmatrix}, \\ A A_\omega^{-1} A &= k(\alpha, \omega)^{-1} \begin{bmatrix} (\alpha^3 + \alpha^2 \omega - \alpha + \omega) I & (\alpha^2 + 2\alpha\omega - 1) I \\ (\alpha^2 + 2\alpha\omega - 1) I & (\alpha^3 + \alpha^2 \omega - \alpha + \omega) I \end{bmatrix}. \end{aligned} \tag{6.24}$$

*Since* $A A_\omega^{-1} A \in P^+(p)$*, first note that* $\alpha^3 + \alpha^2 \omega - \alpha + \omega > 0$ *for* $\omega > \omega_{min}$*, and it follows by choosing* $V^* = \begin{bmatrix} I & 0 \end{bmatrix}$*, and* $V'^* = \begin{bmatrix} 0 & I \end{bmatrix}$ *that*

$$\begin{aligned} V^* x_{b,\omega} &= \frac{(\alpha^2 + \alpha\omega - 1)c + \omega c'}{\alpha^3 + \alpha^2 \omega - \alpha + \omega}, \\ d_{b,\omega,\mu} &= V^*(x_{b,\omega} - x_{b,\mu}) = \frac{(\mu - \omega)(\alpha^2 - 1)(c - \alpha c')}{(\alpha^3 + \alpha^2 \omega - \alpha + \omega)(\alpha^3 + \alpha^2 \mu - \alpha + \mu)}. \end{aligned} \tag{6.25}$$

*It is clear that* $d_{b,\omega,\mu} \in$ *Range*$(c - \alpha c')$, *so* $dim(\mathcal{X}_b) \leq 1$ *(equality holds if and only if* $c - \alpha c' \neq 0$*), while* $\boldsymbol{Ind}_A(\mathcal{S}) = p$ *using Lemma A.1.*

### 6.4.2 Conditions when $\dim(\mathcal{X}) = \mathbf{Ind}_A(\mathcal{S})$

The purpose of this subsection is to provide sufficient conditions under which $\dim(\mathcal{X}) = \mathbf{Ind}_A(\mathcal{S})$. From Corollary 6.3(ii) we already know that $\dim(\mathcal{X}) = 0$ if and only if $\mathbf{Ind}_A(\mathcal{S}) = 0$, so it follows that $\dim(\mathcal{X}) \geq 1$ implies $\mathbf{Ind}_A(\mathcal{S}) \geq 1$. Thus, throughout this subsection, we will assume that $q \geq \dim(\mathcal{X}) \geq 1$.

We start with the observation that a sufficient condition to ensure $\dim(\mathcal{X}) = \mathbf{Ind}_A(\mathcal{S}) = q$ is that for every $y \in \mathcal{Y}$ (with $\mathcal{Y} = \text{Range}(V(H^*H)^{-1}B^*)$ as defined in Theorem 6.1), there exist $b \in \mathbb{F}^n$ and $\omega, \mu > \omega_{\min}$, such that $y = x_{b,\omega} - x_{b,\mu}$, or equivalently $V^*y = V^*(x_{b,\omega} - x_{b,\mu}) = d_{b,\omega,\mu}$. This is because $\dim(\mathcal{Y}) = q$, and $\mathcal{X} \subseteq \mathcal{Y}$. But since $\mathcal{Y} = \text{Range}(V(H^*H)^{-1}B^*)$, this is equivalent to showing that for every $u \in \mathbb{F}^q$, there exist $b, \omega, \mu$ such that $d_{b,\omega,\mu} = (H^*H)^{-1}B^*u$. For convenience, let us define for all $\omega > \omega_{\min}$, $J_\omega \in \mathbb{F}^{(p+q)\times(p+q)}$ and $E_\omega \in \mathbb{F}^{(n-p-q)\times(n-p-q)}$ as

$$J_\omega := G_\omega^{-1}\left(H(H^*G_\omega^{-1}H)^{-1}H^*G_\omega^{-1} - I\right), \quad E_\omega := E + \omega I. \tag{6.26}$$

Let $N$ be defined as in Lemma 6.5, and suppose $N = \begin{bmatrix} N_1 \\ N_2 \end{bmatrix}$ be a partitioning of $N$, where $N_1 \in \mathbb{F}^{p\times q}$ and $N_2 \in \mathbb{F}^{q\times q}$. Then by Lemma 6.5 and the quantities defined therein, we deduce the following sufficient condition:

**Lemma 6.7.** *Let* $u \in \mathbb{F}^q$ *be fixed. If there exists* $b \in \mathbb{F}^n$, $\omega, \mu \in (\omega_{min}, \infty)$, *and* $t, t' \in \mathbb{F}^q$ *satisfying the system*

$$\begin{cases} T(H^*H)^{-1}B^*u = N_1(\omega t' - \mu t) \\ B(H^*H)^{-1}B^*u = N_2(\omega t' - \mu t) + z'_{b,\omega,\mu}(t,t') \\ Nt = J_\mu \begin{bmatrix} c \\ c' - D^*E_\mu^{-1}c'' \end{bmatrix}, \end{cases} \tag{6.27}$$

*with* $z'_{b,\omega,\mu}(t,t') = \begin{bmatrix} B & C - F_\omega \end{bmatrix} Nt' - z_{b,\omega,\mu}(t)$ *(see (6.15)), then* $d_{b,\omega,\mu} = (H^*H)^{-1}B^*u$. *Conversely, for fixed* $(b,\omega,\mu)$, *if* $d_{b,\omega,\mu} = (H^*H)^{-1}B^*u$, *then there exists* $(t,t')$ *satisfying (6.27). Finally, if a* $(b,\omega,\mu,t,t')$ *exists for every* $u \in \mathbb{F}^q$ *solving (6.27), then* $dim(\mathcal{X}) = q$.

*Proof.* Combining the first two equations of (6.27) gives $H(H^*H)^{-1}B^*u+\mu Nt+\begin{bmatrix}0\\ z_{b,\omega,\mu}(t)\end{bmatrix} = G_\omega Nt'$. Since $N$ is a basis for the nullspace of $H^*$, this is equivalent to the first equation of (6.16) with $d = (H^*H)^{-1}B^*u$. Now let $u \in \mathbb{F}^q$ be fixed. By applying Lemma 6.5 we find $d_{b,\omega,\mu} = (H^*H)^{-1}B^*u$. For the converse, suppose $d_{b,\omega,\mu} = (H^*H)^{-1}B^*u$. Then by Lemma 6.5, $t$ exists such that (6.16) is satisfied. Since $N$ is a basis for the nullspace of $H^*$, there exist $t'$ such that $Hd + \mu Nt + \begin{bmatrix}0\\ z_{b,\omega,\mu}(t)\end{bmatrix} = G_\omega Nt'$ and the conclusion follows. Finally, we have already argued the last statement in the paragraph immediately before this lemma. □

Because of this result, our task now reduces to finding conditions that guarantee solutions to (6.27). It turns out that the first and third equations of (6.27) pose no obstructions, which we show in the next lemma, and recall that we denote $c = V^*b$, $c' = V'^*b$ and $c'' = V''^*b$. So a choice of $(c, c', c'')$ uniquely defines $b$, and vice-versa.

**Lemma 6.8.** *The following statements are true:*

*(i) For every $u \in \mathbb{F}^q$, there exists a unique $t'' \in \mathbb{F}^q$, such that $T(H^*H)^{-1}B^*u = N_1t''$, showing that the first equation of (6.27) also admits a solution for some $\omega, \mu > \omega_{min}$, and $t, t' \in \mathbb{F}^q$.*

*(ii) For every $\mu > \omega_{min}$, $t \in \mathbb{F}^q$, and $c'' \in \mathbb{F}^{n-p-q}$, there exists $c \in \mathbb{F}^p$, and $c' \in \mathbb{F}^q$ solving the third equation of (6.27). Conversely for every $\mu > \omega_{min}$, and $b \in \mathbb{F}^n$, there exists a unique $t \in \mathbb{F}^q$ solving the third equation of (6.27).*

*Proof.* (i) This proof relies on the results of Appendix A.3. Fix $u \in \mathbb{F}^q$. Firstly, it has already been argued in Corollary A.2(i) that $\operatorname{rank}(N_1) = q$, so it follows that $t''$ is unique, if it exists. Let $u' = (H^*H)^{-1}B^*u$, and so $(H^*H)u' = (T^2 + B^*B)u' = B^*u$, or equivalently $T^2u' = B^*(u - Bu')$. But this then implies that $T^2u' \in \text{Range}(T) \cap \text{Range}(B^*)$, and since $Tu' \in \text{Range}(T)$ also, we further deduce that $Tu' \in T^\dagger(\text{Range}(T) \cap \text{Range}(B^*))$, where $T^\dagger$ denotes the pseudoinverse of $T$, defined in Corollary A.2. Finally, Corollary A.2(ii) shows that $T^\dagger(\text{Range}(T) \cap \text{Range}(B^*)) \subseteq \text{Range}(N_1)$, and so $Tu' \in \text{Range}(N_1)$, proving the existence of $t''$. One can now choose $t = t' = t''$ and $\omega, \mu > \omega_{\min}$ such that $\omega - \mu = 1$, which ensures $\omega t' - \mu t = t''$, and solves the first equation of (6.27).

(ii) Fix any $\mu > \omega_{\min}$. Let $H_\mu = G_\mu^{-1/2} H$, and notice that since $\operatorname{rank}(H) = p$, we have $\operatorname{rank}(H_\mu) = p$, and $\dim(\text{Range}(H_\mu)^\perp) = q$. Also note that $J_\mu = -G_\mu^{-1/2}(I - H_\mu(H_\mu^* H_\mu)^{-1} H_\mu^*) G_\mu^{-1/2}$, from which we may observe that the inner factor is an orthogonal projector onto $\text{Range}(H_\mu)^\perp$, and so $\operatorname{rank}(J_\mu) = \dim(\text{Range}(J_\mu)) = q$ using invertibility of $G_\mu^{-1/2}$. Moreover, one can compute that $H^* J_\mu = 0$, which gives $\text{Range}(J_\mu) \subseteq \text{Range}(N)$, since the columns of $N$ span the null space of $H^*$. But $\operatorname{rank}(N) = q$, and so in fact

$$\text{Range}(J_\mu) = \text{Range}(N). \tag{6.28}$$

Now if $b \in \mathbb{F}^n$ is given, (6.28) implies the existence of $t \in \mathbb{F}^q$ satisfying the third equation of (6.27), and it is unique as $N$ is full column rank (by construction). Next suppose that $t$ and $c''$ are given. Then (6.28) again implies the existence of $v \in \mathbb{F}^{p+q}$ such that $Nt = J_\mu v$. We can then choose $c, c'$ such that $\begin{bmatrix} c \\ c' \end{bmatrix} = v + \begin{bmatrix} 0 \\ D^* E_\mu^{-1} c'' \end{bmatrix}$, and this is a solution to the third equation of (6.27). Since $\mu$ is arbitrary, this completes the proof.

□

We can now state the first condition that ensures that one can find for all $u \in \mathbb{F}^q$, $(b, \omega, \mu, t, t')$ satisfying (6.27).

**Lemma 6.9.** *If* $\mathit{Range}(T) \cap \mathit{Range}(B^*) = \{0\}$, *for each* $u \in \mathbb{F}^q$, *there exists a* $(b, \omega, \mu, t, t')$ *satisfying* (6.27).

*Proof.* The main ingredient of this proof is the characterization of $N$ in Lemma A.5(iii), by which if $\text{Range}(T) \cap \text{Range}(B^*) = \{0\}$, then one must choose $N_1$ so that $\text{Range}(N_1) = \text{Range}(T)^\perp$, and $N_2 = 0$. We claim that $BN_1 \in \mathbb{F}^{q \times q}$ is invertible, which we prove later. Now fix any $u \in \mathbb{F}^q$, $\omega, \mu > \omega_{\min}$ ($\omega \neq \mu$), and $c'' \in \mathbb{F}^{n-p-q}$. Then by Lemma 6.8(i) there exists a unique $t'' \in \mathbb{F}^q$ such that $T(H^*H)^{-1} B^* u = N_1 t''$. Let $\mathcal{A}_1 := \{(t, t') \mid t, t' \in \mathbb{F}^q, \omega t' - \mu t = t''\}$, and notice that it is non-empty. Also, since $N_2 = 0$, the second equation of (6.27) reduces to

$$BN_1(t' - t) = B(H^*H)^{-1} B^* u + D^*(E_\omega^{-1} - E_\mu^{-1}) c'', \tag{6.29}$$

and then by the invertibility of $BN_1$, there exists a unique $t''' \in \mathbb{F}^q$ such that the set $\mathcal{A}_2 := \{(t, t') \mid t, t' \in \mathbb{F}^q, t' - t = t''', (6.29) \text{ holds}\}$ is non-empty. So the set $\mathcal{A}_1 \cap \mathcal{A}_2$ contains

exactly one element $(t, t')$ which is the solution to the equation

$$\begin{bmatrix} \omega I & -\mu I \\ I & -I \end{bmatrix} \begin{bmatrix} t' \\ t \end{bmatrix} = \begin{bmatrix} t'' \\ t''' \end{bmatrix}, \tag{6.30}$$

because the matrix on the left-hand side has determinant $(\mu - \omega)^q \neq 0$ (by assumption). Finally, with the choices for $\mu, c''$ and the solution $t$ of (6.30), we can by Lemma 6.8(ii) find $c, c'$ satisfying the third equation of (6.27). Thus we have found $(b, \omega, \mu, t, t')$ satisfying (6.27), and this proves the lemma as $u$ is arbitrary.

Now we prove the claim $BN_1 \in \mathrm{GL}(q)$. Suppose for the sake of contradiction this is not true, and there exists $0 \neq y \in \mathbb{F}^q$ such that $BN_1 y = 0$. Then $N_1 y \in \mathrm{Range}(T)^\perp$ (as $\mathrm{Range}(N_1) = \mathrm{Range}(T)^\perp$), and $N_1 y \in \mathrm{Ker}(B) = \mathrm{Range}(B^*)^\perp$ simultaneously, so in fact $N_1 y \in (\mathrm{Range}(T) + \mathrm{Range}(B^*))^\perp$. Finally, we also know that $\mathrm{rank}\,(H^*) = p$, so $\mathbb{F}^p = \mathrm{Range}(H^*) = \mathrm{Range}(T) + \mathrm{Range}(B^*)$ implying $N_1 y \in (\mathbb{F}^p)^\perp$, hence $N_1 y = 0$. Since $N_1$ is full column rank by Corollary A.2(i), this implies $y = 0$ and gives a contradiction. □

A second condition that guarantees for all $u \in \mathbb{F}^q$ existence of $(b, \omega, \mu, t, t')$ satisfying (6.27), is that $T$ is invertible. In fact we state a much stronger theorem below, from which our result will follow. We also recall the definition of $\partial \mathbf{D}_A(\omega, \mu)$ from Definition 6.2, so $V d_{b,\omega,\mu} = x_{b,\omega} - x_{b,\mu} = \partial \mathbf{D}_A(\omega, \mu) b$, and we know that $\mathrm{Range}(\partial \mathbf{D}_A(\omega, \mu)) \subseteq \mathcal{Y}$ by Theorem 6.1. In the theorem below, we are concerned about when this can actually be an equality.

**Theorem 6.2.** *Suppose $T \in GL(p)$, and consider the open set $\mathcal{U} := (\omega_{min}, \infty) \times (\omega_{min}, \infty) \subseteq \mathbb{R}^2$. Then there exists a closed subset $\mathcal{V} \subseteq \mathcal{U}$ of 2-dimensional Lebesgue measure zero, such that for all $(\omega, \mu) \in \mathcal{U} \setminus \mathcal{V}$, we have $Range(\partial \mathbf{D}_A(\omega, \mu)) = \mathcal{Y}$ (with $\mathcal{Y}$ defined as in Theorem 6.1). Moreover, if $A \in P^+(n)$, then $Range(\partial \mathbf{D}_A(\omega, \mu)) = \mathcal{Y}$ for all $\omega, \mu \geq 0$ and $\omega \neq \mu$.*

The proof can be found in [35].

Combining Lemma 6.7, Lemma 6.9 and theorem 6.2 we have thus finished the proof of the following corollary:

**Corollary 6.4.** *$dim(\mathcal{X}) = \boldsymbol{Ind}_A(\mathcal{S})$ if any of the following conditions hold:*

*(i) $Range(T) \cap Range(B^*) = \{0\}$;*

*(ii)* $T \in GL(p)$;

*(iii)* $A \in P^+(n)$.

We state a surprising consequence of Theorem 6.1, and Corollaries 6.3 and 6.4. In the very special case of $\mathbf{Ind}_A(\mathcal{S}) = 1$, Theorem 6.2 can be strengthened significantly.

**Corollary 6.5.** *Suppose* $\boldsymbol{Ind}_A(\mathcal{S}) = 1$. *Then*

*(i) For all distinct* $\omega, \mu \in (\omega_{min}, \infty)$, *the matrix* $\partial \mathbf{D}_A(\omega, \mu)$ *has rank 1, and constant image* $\mathcal{Y}$ *defined in Theorem 6.1.*

*(ii)* $dim(\mathcal{X}) = 1$.

*Proof.* (i) Fix any $\omega, \mu \in (\omega_{\min}, \infty)$ such that $\omega \neq \mu$. Since $\mathbf{Ind}_A(\mathcal{S}) = 1$, by Corollary 6.3(iii), $\mathbf{D}_A$ is injective, so $\partial \mathbf{D}_A(\omega, \mu) \neq 0$. Thus $\partial \mathbf{D}_A(\omega, \mu)$ at least has rank 1. Moreover, by Theorem 6.1, $\text{Range}(\partial \mathbf{D}_A(\omega, \mu)) \subseteq \mathcal{Y}$ with $\dim(\mathcal{Y}) = 1$, and so $\text{Range}(\partial \mathbf{D}_A(\omega, \mu)) = \mathcal{Y}$.

(ii) We use result (i). Pick any $\omega \neq \mu \in (\omega_{\min}, \infty)$. Since $\text{Range}(\partial \mathbf{D}_A(\omega, \mu)) = \mathcal{Y}$, for any $y \in \mathcal{Y}$, there exist $b$ such that $y = \partial \mathbf{D}_A(\omega, \mu) b = x_{b,\omega} - x_{b,\mu}$. So $\dim(\mathcal{X}) = 1$.

□

## 6.5 Conclusion

In this chapter, we studied the following family of weighted least-squares problems

$$x_{b,\omega} = \text{argmin}_{x \in \mathcal{S}} \|(A + \omega I)^{-1/2}(b - Ax)\|_2 \tag{6.31}$$

for any $b \in \mathbb{F}^n$ and $\omega > \omega_{\min} := -\lambda_{\min}(A)$. We showed that

- There exist $\mathcal{Y}$ (dependent on $A$ and $\mathcal{S}$ but independent from $b$, $\omega$ and $\mu$), with $\dim(\mathcal{Y}) \leq \mathbf{Ind}_A(\mathcal{S}) = \dim(\mathcal{S} + A\mathcal{S}) - \dim(\mathcal{S})$ such that, for any $\omega, \mu > \omega_{\min}$, and $b$, $x_{b,\omega} - x_{b,\mu} \in \mathcal{Y}$.
- There exist $A$ and $\mathcal{S}$ such that $\mathbf{Ind}_A(\mathcal{S})$ is arbitrarily large and, for all $b$, the subspace $\text{span}(\{x_{b,\omega} - x_{b,\mu} \mid \omega, \mu > \omega_{\min}\})$ has dimension at most 1. This indicates that the above bound is not always tight.

- However, the subspace $\text{span}(\{x_{b,\omega} - x_{b,\mu} \mid \omega, \mu > \omega_{\min}, b \in \mathbb{F}^n\})$ has dimension $\mathbf{Ind}_A(\mathcal{S})$ if any of the following conditions is true:
  - $\text{Range}(T) \cap \text{Range}(B^*) = \{0\}$;
  - $T \in \text{GL}(p)$;
  - $A \in \text{P}^+(n)$.

  $T$ and $B$ are defined in (6.9).

Additional results can also be found in [35].

While this was not the focus on this work, we note that our result indicates that efficient algorithms for (6.31) are possible. If we express $x = Wy$ where $W$ is a basis for $\mathcal{Y}$, then the search space is reduced from $\mathcal{S}$ to $\mathcal{Y}$.

# Chapter 7

# Conclusion

This thesis is centered around the theme of scientific computing, linear systems, and parallel computing. It is about exploiting sparsity and low-rank properties.

In the first part, we introduced spaND, a sparse fast hierarchical linear solver. spaND is versatile and can be applied to a wide range of problems. It is guaranteed to never break down on SPD matrices. We applied spaND to several problems. On many, it exhibits a $\mathcal{O}(N \log N)$ complexity, with separator sizes scaling like $\mathcal{O}\big(N^{1/3}\big)$. When combined with an iterative method it leads to a fast algorithm to solve linear systems.

We then introduced TaskTorrent. TaskTorrent is a proof-of-concept of a lightweight, fast runtime system with an easy-to-learn API and good interoperability with legacy codes. We applied TaskTorrent on a couple of large linear algebra problems, showing it scales well on thousands of cores and has a low overhead.

We then combined spaND and TaskTorrent. This serves as a validation of the TaskTorrent approach on very large DAGs. While our current implementation of spaND has some limitations, we studied the performance of spaND on a few large problems. When the ranks grow slowly with the problem size, like on 2D problems, spaND shows good weak scalings. Further improvements will require a way to address larger ranks in 3D, such as using distributed RRQR or another sparsification approach.

We then tackled the problem of kernel matrix factorization. This is a critical step when using hierarchical matrices to compress dense matrices arising in integral equations. We introduced Skeletonized Interpolation, which is a fast and reliable algorithm to compute such low-rank approximations. We studied its convergence and showed that it is more robust than existing approaches such as ACA in some cases.

Finally, we studied a parametrized least-square problem. We show that the solutions to this problem lie on a lower-dimensional subspace than expected. We provide an explicit expression for this subspace, and study in which case this bound on the dimension is tight or not. Our result generalizes and explains a previously published result in [89].

# Appendix A

# Index of Invariance

## A.1 Properties of the Index of Invariance

*Proof of Lemma 6.3.* (i) Since $\dim(A\mathcal{S}) \leq \dim(\mathcal{S})$, $\dim(\mathcal{S}+A\mathcal{S}) \leq 2\dim(\mathcal{S})$, and so $\mathbf{Ind}_A(\mathcal{S}) = \dim(\mathcal{S}+A\mathcal{S}) - \dim(\mathcal{S}) \leq \dim(\mathcal{S})$. Furthermore, since $\dim(\mathcal{S}+A\mathcal{S}) \leq n$, $\mathbf{Ind}_A(\mathcal{S}) \leq n - \dim(\mathcal{S})$. We conclude by noting that $\lfloor n/2 \rfloor \geq \min\{\dim(\mathcal{S}),\ n - \dim(\mathcal{S})\} \in \mathbb{N}$.

(ii) Since $\mathcal{S}$ is of dimension $p$, the existence of $V$ follows from using the Gram-Schmidt process on any basis of $\mathcal{S}$. Now assume $q \geq 1$. Since $\dim(\mathcal{S}+A\mathcal{S}) = \dim(\mathcal{S}) + q$, one can find $q$ independent vectors $\{x_i\}_{i=1}^{q}$ in $\mathcal{S}+A\mathcal{S}$ not in $\mathcal{S}$, and let $X = \begin{bmatrix} x_1 & \dots & x_q \end{bmatrix} \in \mathbb{F}^{n\times q}$. Then, applying the Gram Schmidt process to $\begin{bmatrix} V & X \end{bmatrix}$ gives the semi-unitary matrix $\begin{bmatrix} V & V' \end{bmatrix}$. The columns of $V'$ are orthogonal to $\mathcal{S}$ because $\begin{bmatrix} V & V' \end{bmatrix}$ is semi-unitary, so $\text{Range}(V') \subseteq \mathcal{S}^\perp \cap (\mathcal{S}+A\mathcal{S})$. Also $\mathcal{S}+A\mathcal{S} = \mathcal{S} \oplus (\mathcal{S}^\perp \cap (\mathcal{S}+A\mathcal{S}))$, thus $\dim(\mathcal{S}^\perp \cap (\mathcal{S}+A\mathcal{S})) = q = \dim(\text{Range}(V'))$, so in fact $\text{Range}(V') = \mathcal{S}^\perp \cap (\mathcal{S}+A\mathcal{S})$. □

The next three results build upon Corollary 6.1.

**Lemma A.1.** *Let $A \in \mathbb{F}^{n\times n}$, $\mathcal{S} \in Gr^{\mathbb{F}}(p,n)$, and $\boldsymbol{Ind}_A(\mathcal{S}) = q$, with $1 \leq p < n$. Let $\begin{bmatrix} V_1 & V_2 \end{bmatrix}$ be invertible, such that Range$(V_1) = \mathcal{S}$, and Range$(V_2) = \mathcal{S}^\perp$. Then $AV_1 = V_1S_1 + V_2S_2$ for unique $S_1 \in \mathbb{F}^{p\times p}, S_2 \in \mathbb{F}^{(n-p)\times p}$, and* $\text{rank}(S_2) = q$.

**Remark A.1.** *Note that since there always exist $S_1$ and $S_2$ such that $AV_1 = V_1S_1 + V_2S_2$, Lemma A.1 is necessary and sufficient: if* $\text{rank}(S_2) = q$, $\boldsymbol{Ind}_A(\mathcal{S}) = q$.

*Proof.* Existence and uniqueness of $S_1$ and $S_2$, such that $AV_1 = V_1S_1+V_2S_2$, follows from the invertibility of $\begin{bmatrix} V_1 & V_2 \end{bmatrix}$, as the columns form a basis of $\mathbb{F}^n$. Now $A$ has the decomposition (6.9) by Corollary 6.1, where $\text{Range}(V) = \mathcal{S}$, and since $\begin{bmatrix} V & V' & V'' \end{bmatrix}$ is unitary, we also have $\text{Range}(\begin{bmatrix} V' & V'' \end{bmatrix}) = \mathcal{S}^\perp$. Thus there exist $M_1 \in \text{GL}(p)$ and $M_2 \in \text{GL}(n-p)$, such that $V_1 = VM_1$ and $V_2 = \begin{bmatrix} V' & V'' \end{bmatrix} M_2$, and so we have $AV = VM_1S_1M_1^{-1} + \begin{bmatrix} V' & V'' \end{bmatrix} M_2S_2M_1^{-1}$. But $AV = VT + \begin{bmatrix} V' & V'' \end{bmatrix} \begin{bmatrix} B \\ 0 \end{bmatrix}$ also, from which it follows that

$$T = M_1S_1M_1^{-1}, \text{ and } M_2S_2M_1^{-1} = \begin{bmatrix} B \\ 0 \end{bmatrix}. \tag{A.1}$$

The latter gives that $\text{rank}(S_2) = \text{rank}(B) = q$, as $M_1, M_2$ are invertible. □

**Corollary A.1.** *Let $A \in \mathbb{F}^{n\times n}$, and $\mathcal{S}$ be a subspace. Define the nested sequence of subspaces $\mathcal{S}_0 \subseteq \cdots \subseteq \mathcal{S}_i \subseteq \mathcal{S}_{i+1} \subseteq \ldots$, as $\mathcal{S}_0 = \mathcal{S}$, and $\mathcal{S}_{i+1} = \mathcal{S}_i + A\mathcal{S}_i$. Then $\boldsymbol{Ind}_A(\mathcal{S}_i) \geq \boldsymbol{Ind}_A(\mathcal{S}_{i+1})$ for all $i \geq 0$, and there exists $j \geq 0$ such that $\boldsymbol{Ind}_A(\mathcal{S}_j) = 0$.*

*Proof.* If $\mathbf{Ind}_A(\mathcal{S}_0) = 0$, then $\mathcal{S}_i = \mathcal{S}_0$ for all $i \geq 0$, and the statement follows. Now assume $\mathbf{Ind}_A(\mathcal{S}_0) \geq 1$. Let us just show that $\mathbf{Ind}_A(\mathcal{S}_0) \geq \mathbf{Ind}_A(\mathcal{S}_1)$; repeated application of the same argument proves that the sequence $\{\mathbf{Ind}_A(\mathcal{S}_i)\}_{i=0}^{\infty}$ is non-increasing. If $\mathcal{S}_1 = \mathbb{F}^n$ we are again done as $\mathcal{S}_i = \mathcal{S}_1$ for all $i \geq 1$, so assume this is not the case. Consider the decomposition of $A$ in (6.9), from which we have $\text{Range}(\begin{bmatrix} V & V' \end{bmatrix}) = \mathcal{S} + A\mathcal{S}$, and $\text{Range}(V'') = (\mathcal{S} + A\mathcal{S})^\perp$; thus defining $V_1 := \begin{bmatrix} V & V' \end{bmatrix}$ and $V_2 := V''$ we obtain $AV_1 = V_1S_1 + V_2S_2$, with $S_2 = \begin{bmatrix} 0 & D \end{bmatrix}$ (and $S_1$ similarly determined by (6.9)). Now $\text{rank}(S_2) = \text{rank}(D) \leq \mathbf{Ind}_A(\mathcal{S})$, and thus applying Lemma A.1 gives $\mathbf{Ind}_A(\mathcal{S} + A\mathcal{S}) \leq \mathbf{Ind}_A(\mathcal{S})$. To prove that there exists $j \geq 0$ such that $\mathbf{Ind}_A(\mathcal{S}_j) = 0$, notice that if this was false then there would exist $k \geq 0$ such that $\dim(\mathcal{S}_k) > n$, which would give a contradiction. □

**Lemma A.2.** *Let $A \in \mathbb{F}^{n\times n}$, $\mathcal{S} \in Gr^{\mathbb{F}}(p, n)$, and $\boldsymbol{Ind}_A(\mathcal{S}) = q$. Let us also define $\mathcal{S}' := \mathcal{S}^\perp \cap (\mathcal{S} + A\mathcal{S})$. Then we have the following.*

*(i) If $A_\omega := A + \omega I$ for $\omega \in \mathbb{F}$, then $\boldsymbol{Ind}_{A_\omega}(\mathcal{S}) = q$.*

*(ii) $\boldsymbol{Ind}_A(\mathcal{S}^\perp) \leq \min\{p,\ n-p\} \leq \lfloor n/2 \rfloor$, $\boldsymbol{Ind}_A(\mathcal{S}+A\mathcal{S}) \leq \min\{q,\ n-p-q\} \leq \lfloor (n-p)/2 \rfloor$, and $\boldsymbol{Ind}_A(\mathcal{S}') \leq q$.*

*(iii)* *If* $A \in GL(n)$, *then* $\boldsymbol{Ind}_{A^{-1}}(\mathcal{S}) = q$.

*(iv)* *If* $A \in Sym(n)$, *then* $\boldsymbol{Ind}_A(\mathcal{S}^\perp) = q$, *and* $\boldsymbol{Ind}_A(\mathcal{S}') = q$. *Thus if* $q = 0$, *both* $\mathcal{S}$ *and* $\mathcal{S}^\perp$ *are* $A-$*invariant, and if* $A \in Sym(n) \cap GL(n)$, *both* $\mathcal{S}$ *and* $\mathcal{S}^\perp$ *are also* $A^{-1}-$*invariant*[1].

*(v)* *If* $A \in P^+(n)$ *and* $\boldsymbol{Ind}_A(\mathcal{S}) = 0$, *then for any* $s \in \mathbb{R}$, $\boldsymbol{Ind}_{A^s}(\mathcal{S}) = 0$.

*Proof.* (i) This follows because $\mathcal{S} + A_\omega \mathcal{S} = \mathcal{S} + A\mathcal{S}$.

(ii) $\mathbf{Ind}_A(\mathcal{S}^\perp) \leq \min\{p,\ n-p\} \leq \lfloor n/2 \rfloor$ follows by applying Lemma 6.3(i) to $\mathcal{S}^\perp$, and noticing that $\dim(\mathcal{S}^\perp) = n - p$. $\mathbf{Ind}_A(\mathcal{S} + A\mathcal{S}) \leq q$ was proved in Corollary A.1. Applying Lemma 6.3(i) to $\mathcal{S} + A\mathcal{S}$ gives $\mathbf{Ind}_A(\mathcal{S} + A\mathcal{S}) \leq \min\{p+q,\ n-p-q\}$, as $\dim(\mathcal{S} + A\mathcal{S}) = p + q$; so combining gives $\mathbf{Ind}_A(\mathcal{S} + A\mathcal{S}) \leq \min\{q,\ p+q,\ n-p-q\} = \min\{q,\ n-p-q\}$. Finally $\min\{q,\ n-p-q\} \leq \lfloor (n-p)/2 \rfloor$. $\mathbf{Ind}_A(\mathcal{S}') \leq q$ follows from Lemma 6.3(i): $\mathbf{Ind}_A(\mathcal{S}') \leq \dim(\mathcal{S}') = \mathbf{Ind}_A(\mathcal{S}) = q$.

(iii) The $p = 0$ case is clear, so assume $p \geq 1$. Denote by $\hat{A}$ the right-hand side of (6.9). Since $A \in \mathrm{GL}(n)$, $\hat{A} \in \mathrm{GL}(n)$ and we have $A^{-1}\begin{bmatrix} V & V' & V'' \end{bmatrix} = \begin{bmatrix} V & V' & V'' \end{bmatrix} \hat{A}^{-1}$. We use the subscript 1 (resp. 2) to denote the first $p$ (resp. last $n-p$) rows or columns. From the nullity theorem (Theorem 2.1 in [142]), nullity $(\hat{A}^{-1})_{21}$ = nullity $\hat{A}_{21}$, and so rank $(\hat{A}^{-1})_{21}$ = rank $(\hat{A}_{21}) = q$. We then have $A^{-1}V = V(\hat{A}^{-1})_{11} + \begin{bmatrix} V' & V'' \end{bmatrix} (\hat{A}^{-1})_{21}$, and using Lemma A.1 we conclude $\mathbf{Ind}_{A^{-1}}(\mathcal{S}) = q$.

(iv) Assuming $A \in \mathrm{Sym}(n)$, (6.9) gives $P = B^*$, $Q = 0$, $\mathrm{Range}(\begin{bmatrix} V' & V'' \end{bmatrix}) = \mathcal{S}^\perp$, $\mathcal{S}' = \mathrm{Range}(V')$, and $\mathcal{S}'^\perp = \mathrm{Range}(\begin{bmatrix} V & V' \end{bmatrix})$. We also have $A\begin{bmatrix} V' & V'' \end{bmatrix} = \begin{bmatrix} V' & V'' \end{bmatrix} S_1 + VS_2$, and $AV' = V'\widetilde{S}_1 + \begin{bmatrix} V & V' \end{bmatrix} \widetilde{S}_2$, with $S_1, S_2, \widetilde{S}_1, \widetilde{S}_2$ determined by (6.9). In particular $S_2 = \begin{bmatrix} P & 0 \end{bmatrix}$ and $\widetilde{S}_2 = \begin{bmatrix} P \\ D \end{bmatrix}$, and note that $\mathrm{rank}(P) = q$, by Lemma 6.4. Now $\mathrm{rank}(S_2) = \mathrm{rank}(P)$ trivially, while $\mathrm{rank}(\widetilde{S}_2) = q$ as $\mathrm{rank}(\widetilde{S}_2) \geq \mathrm{rank}(P)$, and also $\mathrm{rank}(\widetilde{S}_2) \leq q$ since $\widetilde{S}_2 \in \mathbb{F}^{(n-q)\times q}$. So by Lemma A.1 $\mathbf{Ind}_A(\mathcal{S}^\perp) = \mathbf{Ind}_A(\mathcal{S}') = q$. Finally by (iii), if $A \in \mathrm{Sym}(n) \cap \mathrm{GL}(n)$ and $q = 0$, then $\mathbf{Ind}_{A^{-1}}(\mathcal{S}) = \mathbf{Ind}_{A^{-1}}(\mathcal{S}^\perp) = 0$.

(v) Note that from assumptions, $A\mathcal{S} = \mathcal{S}$, using both invertibility of $A$ and $\mathbf{Ind}_A(\mathcal{S}) = 0$. By an argument similar to that already used in Lemma 6.2 we see that $A^s\mathcal{S} = \mathcal{S}$ also (since $\mathcal{S}$ is spanned by eigenvectors of $A$, which are also eigenvectors of $A^s$), and the conclusion follows.

[1] The fact that $\mathcal{S}$ being $A-$invariant implies $\mathcal{S}^\perp$ is $A-$invariant for Hermitian $A$ is well known.

□

## A.2 Quadratic forms

**Lemma A.3.** *Let $m \geq n$, $A \in \mathbb{F}^{m\times n}$ a full column rank matrix, and $b \in \mathbb{F}^n$. The solution to $\min_{x\in\mathbb{F}^n} \|Ax - b\|_2$ is uniquely given by $x = (A^*A)^{-1}A^*b$.*

*Proof.* Rewrite $f(x) := \|Ax - b\|_2^2 = x^*A^*Ax - (x^*A^*b) - (b^*Ax) + b^*b$. Since $A$ is full column rank, $P = A^*A \in \mathrm{P}^+(n)$. Let $Q$ be such that $P = Q^2$, $Q \in \mathrm{P}^+(n)$. Such $Q$ always exists: let $U\Lambda U^* = P$ be the eigenvalue decomposition of $P$, and then one can choose $Q = U\Lambda^{1/2}U^*$. Then $f(x) = (Qx)^*(Qx) - ((Qx)^*(Q^{-1}A^*b)) - ((Q^{-1}A^*b)^*(Qx)) + b^*b = \|Qx - Q^{-1}A^*b\|_2^2 + b^*b - b^*AP^{-1}A^*b$. The minimum is obtained when $Qx - Q^{-1}A^*b = 0$, which happens uniquely (since $Q \in \mathrm{P}^+(n)$) when $Q^2x = A^*b$ or $A^*Ax = A^*b$. □

*Proof of Lemma 6.1.* Each $x \in \mathcal{S}$ can be uniquely written as $x = Vy$ for some $y \in \mathbb{F}^p$. Then rewrite the function to minimize in (6.6) as

$$\|A_\omega^{-1/2}(b - Ax)\|_2 = \|A_\omega^{-1/2}AVy - A_\omega^{-1/2}b\|_2. \tag{A.2}$$

In this expression, $A_\omega^{-1/2}AV$ is full rank since $A_\omega \in \mathrm{P}^+(n)$ (because of the choice of $\omega_{\min}$), $A \in \mathrm{GL}(n)$, and $V$ is full-rank. Then using Lemma A.3, the unique minimizer to (A.2) is given by $y = (V^*AA_\omega^{-1}AV)^{-1}V^*AA_\omega^{-1}b$ or $x = Vy = V(V^*AA_\omega^{-1}AV)^{-1}V^*AA_\omega^{-1}b$. For the last part, notice that $x_{b,\omega}$ does not depend on the choice of $V$, because if $V' \in \mathbb{F}^{n\times p}$ is another full rank matrix whose columns span $\mathcal{S}$, then $V' = VL$ for some $L \in \mathrm{GL}(p)$, from which it follows that $V(V^*AA_\omega^{-1}AV)^{-1}V^* = V'(V'^*AA_\omega^{-1}AV')^{-1}V'^*$. This completes the proof. □

## A.3 The nullspace of $H^*$

We present here a geometrical relationship between the images of $T$ and $B^*$, and the nullspace of $H^* = \begin{bmatrix} T & B^* \end{bmatrix}$, as defined in the statement of Theorem 6.1. Recall that $T \in \mathbb{F}^{p\times p}$ is Hermitian, and $H^*$ and $B^* \in \mathbb{F}^{p\times q}$ are full rank, with $p \geq q$. To do this we first establish a more general result below. In this appendix, for any matrix $M \in \mathbb{F}^{p\times q}$, and $\mathcal{A}$ a subspace of $\mathbb{F}^p$, we define $\mathrm{pre}_M(\mathcal{A}) := \{x \in \mathbb{F}^q \mid Mx \in \mathcal{A}\}$.

**Lemma A.4.** *Let* $M = \begin{bmatrix} M_1 & M_2 \end{bmatrix}$ *be a matrix such that* $M_1 \in \mathbb{F}^{s\times s_1}$, $M_2 \in \mathbb{F}^{s\times s_2}$, *with* $s \geq s_2$, *and suppose that* $\operatorname{rank}(M_2) = s_2$. *Let* $N$ *be any matrix such that the columns of* $N$ *span the nullspace of* $M$, *and let us partition* $N$ *as* $N = \begin{bmatrix} N_1 \\ N_2 \end{bmatrix}$, *where* $N_1$ *are the first* $s_1$ *rows of* $N$. *Then*

*(i)* $N_2 = -(M_2^* M_2)^{-1} M_2^* M_1 N_1$.

*(ii)* $\operatorname{rank}(N) = \operatorname{rank}(N_1)$.

*(iii)* $\mathit{Range}(N_1) = \mathit{pre}_{M_1}(\mathit{Range}(M_2))$.

*Proof.* (i) As the columns of $N$ span the nullspace of $M$, we have $M_1N_1 + M_2N_2 = 0$, and since $\operatorname{rank}(M_2) = s_2$, the matrix $M_2^*M_2$ is invertible, from which the result follows.

(ii) Let $N' := -(M_2^*M_2)^{-1}M_2^*M_1$. From (i), we have $N = \begin{bmatrix} I \\ N' \end{bmatrix} N_1$. Since $\begin{bmatrix} I \\ N' \end{bmatrix}$ is full column rank, the conclusion follows.

(iii) From (i) $M_1N_1 = -M_2N_2$; so each column of $N_1$ is in $\operatorname{pre}_{M_1}(\operatorname{Range}(M_2))$, and hence $\operatorname{Range}(N_1) \subseteq \operatorname{pre}_{M_1}(\operatorname{Range}(M_2))$. Now define $\mathcal{A} := \operatorname{Range}(M_1) \cap \operatorname{Range}(M_2)$, and observe that $\operatorname{pre}_{M_1}(\operatorname{Range}(M_2)) = \operatorname{pre}_{M_1}(\mathcal{A})$. To prove the result it suffices to show that $\dim(\operatorname{Range}(N_1)) = \dim(\operatorname{pre}_{M_1}(\mathcal{A}))$. Let $\operatorname{rank}(M) = r$, and suppose $\operatorname{rank}(N) = t$, so $\dim(\operatorname{Range}(N_1)) = t$ by part (ii). From the rank-nullity theorem we first have

$$\begin{aligned} t &= s_1 + s_2 - r, \\ \dim(\operatorname{Range}(M_1)) + \dim(\operatorname{Ker}(M_1)) &= s_1, \end{aligned} \tag{A.3}$$

and since $\operatorname{Range}(M_2) = \mathcal{A} \oplus (\mathcal{A}^\perp \cap \operatorname{Range}(M_2))$, and $\mathbb{F}^s = \operatorname{Range}(M_1) \oplus \operatorname{Range}(M_1)^\perp$ we also have

$$\begin{aligned} \dim(\mathcal{A}) + \dim(\mathcal{A}^\perp \cap \operatorname{Range}(M_2)) &= \dim(\operatorname{Range}(M_2)) = s_2, \\ \dim(\operatorname{Range}(M_1)) + \dim(\operatorname{Range}(M_1)^\perp) &= s. \end{aligned} \tag{A.4}$$

Next notice that $\operatorname{Range}(M_1) + \operatorname{Range}(M_2) = \operatorname{Range}(M_1) + \mathcal{A} \oplus (\mathcal{A}^\perp \cap \operatorname{Range}(M_2)) = \operatorname{Range}(M_1) + (\mathcal{A}^\perp \cap \operatorname{Range}(M_2))$. But $\operatorname{Range}(M_1) \cap (\mathcal{A}^\perp \cap \operatorname{Range}(M_2)) = (\operatorname{Range}(M_1) \cap$

$\text{Range}(M_2)) \cap \mathcal{A}^\perp = \mathcal{A} \cap \mathcal{A}^\perp = \{0\}$, so in fact

$$\dim(\text{Range}(M_1)) + \dim(\mathcal{A}^\perp \cap \text{Range}(M_2)) = \dim(\text{Range}(M_1) + \text{Range}(M_2)) = r. \tag{A.5}$$

Using (A.3), (A.4) and (A.5) we finally find

$$\dim(\mathcal{A}) + \dim(\text{Ker}(M_1)) = s_1 + s_2 - r = t. \tag{A.6}$$

Now consider the set $\mathcal{A}' := \text{Ker}(M_1)^\perp \cap \text{pre}_{M_1}(\mathcal{A})$, which is a subspace of $\text{pre}_{M_1}(\mathcal{A})$. Then the restriction to $\mathcal{A}'$ of the linear map given by $M_1$ is an isomorphism $M_1|_{\mathcal{A}'} : \mathcal{A}' \to \mathcal{A}$, which gives $\dim(\mathcal{A}) = \dim(\mathcal{A}')$. Notice that this proves the result as $\text{pre}_{M_1}(\mathcal{A}) = \mathcal{A}' \oplus \text{Ker}(M_1)$, since $\text{Ker}(M_1)$ is also a subspace of $\text{pre}_{M_1}(\mathcal{A})$, giving $\dim(\text{pre}_{M_1}(\mathcal{A})) = \dim(\mathcal{A}') + \dim(\text{Ker}(M_1)) = t$, using (A.6).

□

Let us apply Lemma A.4 to characterize the nullspace of $H^*$, which is of dimension $q$, and derive some consequences. If we choose $N \in \mathbb{F}^{(p+q)\times q}$ to be full column rank, as in Lemma 6.5, and write $N = \begin{bmatrix} N_1 \\ N_2 \end{bmatrix}$ as in Lemma A.4 with $N_1 \in \mathbb{F}^{p\times q}$, then we have $N_2 = -(BB^*)^{-1}BTN_1$, and $\text{Range}(N_1) = \text{pre}_T(\text{Range}(B^*))$. Denoting $\mathcal{A} := \text{Range}(T) \cap \text{Range}(B^*)$, and $\mathcal{A}' := \text{Ker}(T)^\perp \cap \text{pre}_T(\mathcal{A})$, the proof of Lemma A.4(iii) shows that $\text{Range}(N_1) = \text{pre}_T(\mathcal{A}) = \text{Ker}(T) \oplus \mathcal{A}'$. Now as $T \in \text{Sym}(p)$, we have $\text{Ker}(T) = \text{Range}(T)^\perp$, which means that $\mathcal{A}' \subseteq \text{Range}(T)$, and $\text{Range}(N_1) = \text{Range}(T)^\perp \oplus \mathcal{A}'$ (an orthogonal direct sum). Thus we have that both $\mathcal{A}, \mathcal{A}'$ are subspaces of $\text{Range}(T)$. Also $T\mathcal{A}' \subseteq \mathcal{A}$ from definition, and moreover $\dim(\mathcal{A}) = \dim(\mathcal{A}')$ from the proof of Lemma A.4(iii), which means $T\mathcal{A}' = \mathcal{A}$. Now by the spectral theorem, the restriction to the subspace $\text{Range}(T)$ of the linear operator $T$, i.e. $T|_{\text{Range}(T)}$, is invertible; thus in fact $T\mathcal{A}' = T|_{\text{Range}(T)}\mathcal{A}' = \mathcal{A}$, or $\mathcal{A}' = T|^{-1}_{\text{Range}(T)}\mathcal{A}$. Denoting the pseudoinverse [113, 120] of $T$ by $T^\dagger$, it is also easily checked that $T^\dagger|_{\text{Range}(T)} = T|^{-1}_{\text{Range}(T)}$, since $T \in \text{Sym}(n)$, and so we have proved the following corollary:

**Corollary A.2.** *Let $N \in \mathbb{F}^{(p+q)\times q}$ be a full column rank matrix, whose columns span the nullspace of $H^*$, and let $T^\dagger$ be the pseudoinverse of $T$. If we partition $N$ as $N = \begin{bmatrix} N_1 \\ N_2 \end{bmatrix}$, where $N_1 \in \mathbb{F}^{p\times q}$ and $N_2 \in \mathbb{F}^{q\times q}$. Then*

*(i)* $N_2 = -(BB^*)^{-1}BTN_1$, *and* $\text{rank}\,(N) = \text{rank}\,(N_1) = q$.

*(ii)* $Range(N_1) = Range(T)^{\perp} \oplus T^{\dagger}(Range(T) \cap Range(B^*))$.

Finally we note a couple of special cases in the next lemma that follow from Corollary A.2.

**Lemma A.5.** *Let* $N, N_1, N_2$ *be as in Corollary A.2. Then*

*(i)* $Range(B^*) \subseteq Range(T)$ *if and only if* $T \in GL(p)$. *In this case, one can choose* $N_1 = -T^{-1}B^*$, *and* $N_2 = I$.

*(ii)* $Range(T) \subseteq Range(B^*)$ *if and only if* $B^* \in GL(p)$. *In this case one can choose* $N_1 = Q$, *for any* $Q \in GL(p)$ *(for e.g.* $Q = I$*).*

*(iii) If* $Range(T) \cap Range(B^*) = \{0\}$, *then* $N_1$ *should be chosen such that* $Range(N_1) = Range(T)^{\perp}$, *and in this case* $N_2 = 0$.

*Proof.* (i) If $T \in \text{GL}(p)$, $\text{Range}(T) = \mathbb{F}^p$ and so $\text{Range}(B^*) \subseteq \text{Range}(T)$. If $\text{Range}(B^*) \subseteq \text{Range}(T)$, then $\text{Range}(T) = \mathbb{F}^p$ as $\text{rank}\,(H^*) = p$, so $T \in \text{GL}(p)$. In this case, $\text{Range}(T)^{\perp} = \{0\}$, $\text{Range}(T) \cap \text{Range}(B^*) = \text{Range}(B^*)$, and $T^{\dagger} = T^{-1}$. Thus from Corollary A.2(ii) we have $\text{Range}(N_1) = T^{-1}\text{Range}(B^*) = \text{Range}(T^{-1}B^*)$, and so we can choose $N_1 = -T^{-1}B^*$. With this choice, we get $N_2 = I$ by Corollary A.2(i).

(ii) Interchanging the roles of $T$ and $B^*$ in the proof of part (i) proves that $\text{Range}(T) \subseteq \text{Range}(B^*)$ if and only if $B^* \in \text{GL}(p)$. In this case $\text{Range}(T) \cap \text{Range}(B^*) = \text{Range}(T)$, and so $T^{\dagger}(\text{Range}(T) \cap \text{Range}(B^*)) = T^{\dagger}\text{Range}(T) = T|^{-1}_{\text{Range}(T)}\text{Range}(T) = \text{Range}(T)$. This gives using Corollary A.2(ii) that $\text{Range}(N_1) = \text{Range}(T)^{\perp} \oplus \text{Range}(T) = \mathbb{F}^p$. Thus $N_1$ must be chosen to be an invertible matrix in $\text{GL}(p)$.

(iii) It follows directly in this case that $\text{Range}(N_1) = \text{Range}(T)^{\perp}$ from Corollary A.2(ii). Thus $TN_1 = 0$ and this implies $N_2 = 0$.

□

# Bibliography


[1] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Task-based multifrontal QR solver for GPU-accelerated multicore architectures. In *22nd international conference on high performance computing (HiPC)*, pages 54–63. IEEE, 2015.

[2] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems. *ACM Trans. Math. Software*, 43(2):1–22, 2016.

[3] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

[4] Emmanuel Agullo, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julie Langou, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Asim YarKhan. PLASMA user's guide. Technical report, ICL, UTK, 2009.

[5] Emmanuel Agullo, Luc Giraud, and Stojce Nakov. Task-based sparse hybrid linear solver for distributed memory heterogeneous architectures. In *European Conference on Parallel Processing*, pages 83–95. Springer, 2016.

[6] Sivaram Ambikasaran. *Fast algorithms for dense numerical linear algebra and applications*. PhD thesis, Stanford University, 2013.

[7] Patrick Amestoy, Cleve Ashcraft, Olivier Boiteau, Alfredo Buttari, Jean-Yves L'Excellent, and Clément Weisbecker. Improving multifrontal methods by means of block low-rank representations. *SIAM J. Sci. Comput.*, 37(3):A1451–A1474, 2015.

[8] Patrick R. Amestoy, Iain S. Duff, Jean-Yves L'Excellent, and Jacko Koster. MUMPS: a general purpose distributed memory sparse solver. In *Internat. Workshop Appl. Parallel Comput.*, pages 121–130. Springer, 2000.

[9] Edward Anderson, Zhaojun Bai, Christian Bischof, L. Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. *LAPACK Users' guide*. SIAM, 1999.

[10] J. Ang, K. Evans, A. Geist, M. Heroux, P. Hovland, O. Marques, L. McInnes, E. Ng, and S. Wild. Report on the workshop on extreme-scale solvers: Transitions to future architectures. Technical report, Office of Advanced Scientific Computing Research, US Department of Energy, 2012.

[11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[12] Josh Barnes and Piet Hut. A hierarchical $\mathcal{O}(N \log N)$ force-calculation algorithm. *Nature*, 324(6096):446, 1986.

[13] Michael Edward Bauer. *Legion: Programming distributed heterogeneous architectures with logical regions*. PhD thesis, Stanford University, 2014.

[14] Olivier Beaumont, Julien Langou, Willy Quach, and Alena Shilova. A makespan lower bound for the scheduling of the tiled cholesky factorization based on ALAP schedule, 2020.

[15] Mario Bebendorf. Approximation of boundary element matrices. *Numer. Math.*, 86(4):565–589, 2000.

[16] Mario Bebendorf. Efficient inversion of the Galerkin matrix of general second-order elliptic operators with nonsmooth coefficients. *Math. Comp.*, 74(251):1179–1199, 2005.

[17] Mario Bebendorf and Wolfgang Hackbusch. Existence of $\mathcal{H}$-matrix approximants to the inverse FE-matrix of elliptic operators with $L_\infty$-coefficients. *Numer. Math.*, 95(1):1–28, 2003.

[18] Mario Bebendorf and Sergej Rjasanow. Adaptive low-rank approximation of collocation matrices. *Computing*, 70(1):1–24, 2003.

[19] Marsha J. Berger and Shahid H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Trans. Comput*, C-36(5):570–580, 1987.

[20] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Rev.*, 59(1):65–98, 2017.

[21] OpenMP Architecture Review Board. OpenMP Fortran application program interface. Available at `https://www.openmp.org/wp-content/uploads/fspec10.pdf`, 1997.

[22] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring. *Scientific Programming*, 20(2):129–150, 2012.

[23] Dan Bonachea and P. Hargrove. GASNet specification, v1.8.1. Technical report, Lawrence Berkeley National Lab. Berkeley, CA, USA, 2017.

[24] Steffen Börm and Lars Grasedyck. Low-rank approximation of integral operators by interpolation. *Computing*, 72(3):325–332, 2004.

[25] Steffen Börm and Lars Grasedyck. Hybrid cross approximation of integral operators. *Numer. Math.*, 101(2):221–249, 2005.

[26] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra. PaRSEC: Exploiting heterogeneity to enhance scalability. *Computing in Science Engineering*, 15(6):36–45, 2013.

[27] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, et al. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1432–1441. IEEE, 2011.

[28] James H. Bramble. *Multigrid methods*, volume 294. CRC Press, 1993.

[29] A. Brandt, S. McCormick, and J. Ruge. Algebraic multigrid (AMG) for automatic multigrid solutions with application to geodetic computations. Technical report, Inst. for computational Studies, Fort Collins, Colorado, 1982.

[30] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In *18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 180–186. IEEE, 2010.

[31] David Callahan, Bradford L. Chamberlain, and Hans P. Zima. The cascade high productivity language. In *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.*, pages 52–60. IEEE, 2004.

[32] Léopold Cambier, Chao Chen, Erik G. Boman, Sivasankaran Rajamanickam, Raymond S. Tuminaro, and Eric Darve. An algebraic sparsified nested dissection algorithm using low-rank approximations. *SIAM J. Matrix Anal. Appl.*, 41(2):715–746, 2020.

[33] Léopold Cambier and Eric Darve. Fast low-rank kernel matrix factorization using skeletonized interpolation. *SIAM J. Sci. Comput.*, 41(3):A1652–A1680, 2019.

[34] Léopold Cambier, Yizhou Qian, and Eric Darve. TaskTorrent: a lightweight distributed task-based runtime system in C++. *To appear in proceedings of the 2020 Parallel Application Workshop, Alternatives to MPI+X, IEEE. arXiv preprint arXiv:2009.10697*, 2020.

[35] Léopold Cambier and Rahul Sarkar. The index of invariance and its implications for a parameterized least squares problem. *arXiv preprint arXiv:2008.11154*, 2020.

[36] Tony F. Chan. Rank revealing QR factorizations. *Linear Algebra Appl.*, 88:67–82, 1987.

[37] S. Chandrasekaran, M. Gu, and T. Pals. Fast and stable algorithms for hierarchically semi-separable representations, 2004.

[38] Shiv Chandrasekaran, Patrick Dewilde, Ming Gu, T. Pals, Xiaorui Sun, Alle-Jan van der Veen, and Daniel White. Some fast algorithms for sequentially semiseparable representations. *SIAM J. Matrix Anal. Appl.*, 27(2):341–364, 2005.

[39] Shiv Chandrasekaran, Patrick Dewilde, Ming Gu, and Naveen Somasunderam. On the numerical rank of the off-diagonal blocks of schur complements of discretized elliptic PDEs. *SIAM J. Matrix Anal. Appl.*, 31(5):2261–2290, 2010.

[40] Shiv Chandrasekaran, Ming Gu, and Timothy Pals. A fast ULV decomposition solver for hierarchically semiseparable representations. *SIAM J. Matrix Anal. Appl.*, 28(3):603–622, 2006.

[41] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph Von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Notices*, 40(10):519–538, 2005.

[42] Chao Chen, Léopold Cambier, Erik G Boman, Sivasankaran Rajamanickam, Raymond S Tuminaro, and Eric Darve. A robust hierarchical solver for ill-conditioned systems with applications to ice sheet modeling. *J. Comput. Phys.*, 396:819–836, 2019.

[43] Chao Chen, Hadi Pouransari, Sivasankaran Rajamanickam, Erik G Boman, and Eric Darve. A distributed-memory hierarchical solver for general sparse linear systems. *Parallel Comput.*, 74:49–64, 2018.

[44] Yanqing Chen, Timothy A. Davis, William W. Hager, and Sivasankaran Rajamanickam. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Trans. Math. Software*, 35(3):22, 2008.

[45] Hongwei Cheng, Zydrunas Gimbutas, Per-Gunnar Martinsson, and Vladimir Rokhlin. On the compression of low rank matrices. *SIAM J. Sci. Comput.*, 26(4):1389–1404, 2005.

[46] Cédric Chevalier and François Pellegrini. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Comput.*, 34(6-8):318–331, 2008.

[47] MA. Christie, MJ. Blunt, et al. Tenth SPE comparative solution project: A comparison of upscaling techniques. In *SPE reservoir simulation symposium*. Society of Petroleum Engineers, 2001.

[48] Fan RK. Chung and Fan Chung Graham. *Spectral graph theory*. Number 92 in Regional Conference Series in Mathematics. American Mathematical Soc., 1997.

[49] Robert Cimrman, Vladimír Lukeš, and Eduard Rohan. Multiscale finite element calculations in Python using SfePy. *Adv. Comput. Math.*, 2019.

[50] Eduardo Corona, Abtin Rahimian, and Denis Zorin. A Tensor-Train accelerated solver for integral equations in complex geometries. *J. Comput. Phys.*, 334:145–169, 2017.

[51] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[52] Timothy A. Davis. *Direct methods for sparse linear systems*. SIAM, 2006.

[53] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Software*, 38(1):1, 2011.

[54] Emmanuel Detournay and Alexander H.-D. Cheng. Fundamentals of poroelasticity. In *Analysis and design methods*, pages 113–171. Elsevier, 1993.

[55] Guido KE Dietl. *Linear estimation and detection in Krylov subspaces*, volume 1. Springer Science & Business Media, 2007.

[56] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Process. Lett.*, 21(02):173–193, 2011.

[57] Thomas D. Economon, Francisco Palacios, Sean R. Copeland, Trent W. Lukaczyk, and Juan J. Alonso. SU2: An open-source suite for multiphysics simulation and design. *AIAA Journal*, 54(3):828–846, 2016.

[58] Tarek El-Ghazawi and Lauren Smith. UPC: unified parallel C. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pages 27–es, 2006.

[59] Robert D. Falgout and Ulrike Meier Yang. Hypre: A library of high performance preconditioners. In *International Conference on Computational Science*, pages 632–641. Springer, 2002.

[60] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J Dally,

et al. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pages 83–es, 2006.

[61] Mathieu Faverge, Grégoire Pichon, Pierre Ramet, and Jean Roman. On the use of $\mathcal{H}$-matrix arithmetic in PaStiX: a preliminary study. In *Workshop on Fast Direct Solvers*, Toulouse, France, June 2015.

[62] Radii Petrovich Fedorenko. A relaxation method for solving elliptic difference equations. *Comput. Math. Math. Phys.*, 1(4):1092–1096, 1962.

[63] Jordi Feliu-Fabà, Kenneth L. Ho, and Lexing Ying. Recursively preconditioned hierarchical interpolative factorization for elliptic partial differential equations. *arXiv preprint arXiv:1808.01364*, 2018.

[64] David Chin-Lung Fong and Michael Saunders. LSMR: An iterative algorithm for sparse least-squares problems. *SIAM J. Sci. Comput.*, 33(5):2950–2971, 2011.

[65] William Fong and Eric Darve. The black-box fast multipole method. *J. Comput. Phys.*, 228(23):8712–8725, 2009.

[66] Message Passing Interface Forum. MPI: A message-passing interface standard, 1994.

[67] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 212–223, 1998.

[68] Michael Garland, Manjunath Kudlur, and Yili Zheng. Designing a unified programming model for heterogeneous machines. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[69] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. SLATE: design of a modern distributed and accelerated linear algebra library. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–18, 2019.

[70] Thierry Gautier, Joao VF. Lima, Nicolas Maillard, and Bruno Raffin. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In

*27th International Symposium on Parallel and Distributed Processing*, pages 1299–1308. IEEE, 2013.

[71] Alan George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10(2):345–363, 1973.

[72] Alan George, Kh Ikramov, and Andrey B. Kucherov. Some properties of symmetric quasi-definite matrices. *SIAM J. Matrix Anal. Appl.*, 21(4):1318–1323, 2000.

[73] Pieter Ghysels, Xiaoye S. Li, François-Henry Rouet, Samuel Williams, and Artem Napov. An efficient multicore implementation of a novel HSS-structured multifrontal solver using randomized sampling. *SIAM J. Sci. Comput.*, 38(5):S358–S384, 2016.

[74] John R. Gilbert and Robert Endre Tarjan. The analysis of a nested dissection algorithm. *Numer. Math.*, 50(4):377–404, 1986.

[75] Gene H. Golub and Charles F. Van Loan. *Matrix computations*, volume 4. JHU Press, 2013.

[76] Ananth Grama, Vipin Kumar, Anshul Gupta, and George Karypis. *Introduction to parallel computing*. Pearson Education, 2003.

[77] Lars Grasedyck, Ronald Kriemann, and Sabine Le Borne. Domain decomposition based $\mathcal{H}$-LU preconditioning. *Numer. Math.*, 112(4):565–600, 2009.

[78] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73(2):325–348, 1987.

[79] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73(2):325–348, 1987.

[80] Ming Gu and Stanley C. Eisenstat. Efficient algorithms for computing a strong rank-revealing QR factorization. *SIAM J. Sci. Comput.*, 17(4):848–869, 1996.

[81] Ayse Guven. *Quantitative Perturbation Theory for Compact Operators on a Hilbert Space*. PhD thesis, Queen Mary University of London, 2016.

[82] Khoromskij Hackbusch. A sparse $\mathcal{H}$-matrix arithmetic. part II: application to multi-dimensional problems. *Computing*, 64(1):21–47, 2000.

[83] Wolfgang Hackbusch. A sparse matrix arithmetic based on $\mathcal{H}$-matrices. part I: Introduction to $\mathcal{H}$-Matrices. *Computing*, 62(2):89–108, 1999.

[84] Wolfgang Hackbusch. *Multi-grid methods and applications*, volume 4. Springer Science & Business Media, 2013.

[85] Wolfgang Hackbusch and Steffen Börm. Data-sparse approximation by adaptive $\mathcal{H}$ 2-matrices. *Computing*, 69(1):1–35, 2002.

[86] Wolfgang Hackbusch and Steffen Börm. $\mathcal{H}^2$-matrix approximation of integral operators by interpolation. *Appl. Numer. Math.*, 43(1-2):129–143, 2002.

[87] Wolfgang Hackbusch and Zenon Paul Nowak. On the fast matrix multiplication in the boundary element method by panel clustering. *Numer. Math.*, 54(4):463–491, 1989.

[88] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Rev.*, 53(2):217–288, 2011.

[89] Eric Hallman and Ming Gu. LSMB: Minimizing the backward error for least-squares problems. *SIAM J. Matrix Anal. Appl.*, 39(3):1295–1317, 2018.

[90] Pascal Hénon, Pierre Ramet, and Jean Roman. PASTIX: a high-performance parallel direct solver for sparse symmetric positive definite systems. *Parallel Comput.*, 28(2):301–321, 2002.

[91] Magnus Rudolph Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6), 1952.

[92] Nicholas J. Higham. *Accuracy and stability of numerical algorithms*, volume 80. Siam, 2002.

[93] Kenneth L. Ho and Sheehan Olver. LowRankApprox.jl: Fast low-rank matrix approximation in Julia, May 2018.

[94] Kenneth L. Ho and Lexing Ying. Hierarchical interpolative factorization for elliptic operators: integral equations. *Comm. Pure Appl. Math.*, 2015.

[95] Kenneth L. Ho and Lexing Ying. Hierarchical interpolative factorization for elliptic operators: differential equations. *Comm. Pure Appl. Math.*, 69(8):1415–1451, 2016.

[96] Bayram Ali Ibrahimoglu. Lebesgue functions and Lebesgue constants in polynomial interpolation. *J. Inequal. Appl*, 2016(1):93, 2016.

[97] Francisco D. Igual, Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, Robert A. Van De Geijn, and Field G. Van Zee. The FLAME approach: From dense linear algebra algorithms to high-performance multi-accelerator implementations. *Journal of Parallel and Distributed Computing*, 72(9):1134–1143, 2012.

[98] Christopher Frank Joerg. *The Cilk system for parallel multithreaded computing.* PhD thesis, Massachusetts Institute of Technology, 1996.

[99] Laxmikant V Kale and Sanjeev Krishnan. Charm++ a portable concurrent object oriented system based on C++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, 1993.

[100] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[101] George Karypis, Kirk Schloegel, and Vipin Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library. Technical report, University of Minnesota, 1997.

[102] Kyungjoo Kim and Victor Eijkhout. A parallel sparse direct solver via hierarchical DAG scheduling. *ACM Trans. Math. Software*, 41(1):1–27, 2014.

[103] Xavier Lacoste, Mathieu Faverge, George Bosilca, Pierre Ramet, and Samuel Thibault. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In *International Parallel & Distributed Processing Symposium Workshops*, pages 29–38. IEEE, 2014.

[104] Pierre L'Eplattenier, Grant Cook, Cleve Ashcraft, Mike Burger, Jose Imbert, and Michael Worswick. Introduction of an electromagnetism module in LS-DYNA for coupled mechanical-thermal-electromagnetic simulations. *Steel research international*, 80(5):351–358, 2009.

[105] Randall Leveque. *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems (Classics in Applied Mathematics)*. Classics in Applied Mathematics. SIAM, Society for Industrial and Applied Mathematics, 2007.

[106] Yingzhou Li and Lexing Ying. Distributed-memory hierarchical interpolative factorization. *Res. Math. Sci.*, 4(1):12, 2017.

[107] Edo Liberty, Franco Woolfe, Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert. Randomized algorithms for the low-rank approximation of matrices. *Proc. Natl. Acad. Sci.*, 104(51):20167–20172, 2007.

[108] Jonathan Josiah Lifflander and Philippe Pierre Pebay. DARMA/vt FY20 mid-year status report. Technical report, Sandia National Lab.(SNL-CA), Livermore, CA, USA, 2020.

[109] Richard J. Lipton, Donald J. Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM J. Numer. Anal.*, 16(2):346–358, 1979.

[110] Tzon-Tzer Lu and Sheng-Hua Shiou. Inverses of 2× 2 block matrices. *Computers & Mathematics with Applications*, 43(1-2):119–129, 2002.

[111] Michael W. Mahoney and Petros Drineas. CUR matrix decompositions for improved data analysis. *Proc. Natl. Acad. Sci. USA*, 106(3):697–702, 2009.

[112] L. Miranian and Ming Gu. Strong rank revealing LU factorizations. *Linear Algebra Appl.*, 367:1–16, 2003.

[113] Eliakim H. Moore. On the reciprocal of the general algebraic matrix. *Bull. Am. Math. Soc.*, 26:394–395, 1920.

[114] Gordon E. Moore et al. Cramming more components onto integrated circuits, 1965.

[115] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[116] Robert W. Numrich and John Reid. Co-Array Fortran for parallel programming. In *SIGPLAN Fortran Forum*, volume 17, pages 1–31. ACM New York, NY, USA, 1998.

[117] Dianne P. O'Leary. The block conjugate gradient algorithm and related methods. *Linear Algebra Appl.*, 29:293–322, 1980.

[118] Christopher C. Paige and Michael A. Saunders. Solution of sparse indefinite systems of linear equations. *SIAM J. Numer. Anal.*, 12(4):617–629, 1975.

[119] Christopher C. Paige and Michael A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Trans. Math. Software*, 8(1):43–71, 1982.

[120] Roger Penrose. A generalized inverse for matrices. *Math. Proc. Cambridge Philos. Soc.*, 51(3):406–413, 1955.

[121] Josep M. Perez, Rosa M. Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *International Conference on Cluster Computing*, pages 142–151. IEEE, 2008.

[122] Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Handling task dependencies under strided and aliased references. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 263–274, 2010.

[123] Kaare Brandt Petersen, Michael Syskind Pedersen, et al. The matrix cookbook.

[124] Grégoire Pichon, Eric Darve, Mathieu Faverge, Pierre Ramet, and Jean Roman. Sparse supernodal solver using block low-rank compression: Design, performance and analysis. *J. Comput. Sci.*, 27:255–270, 2018.

[125] Hadi Pouransari, Pieter Coulier, and Eric Darve. Fast hierarchical solvers for sparse matrices using extended sparsification and low-rank approximation. *SIAM J. Sci. Comput.*, 39(3):A797–A830, 2017.

[126] Andrey Prokopenko, Christopher M. Siefert, Jonathan J. Hu, Mark Hoemmen, and Alicia Klinvex. Ifpack2 User's Guide 1.0. Technical Report SAND2016-5338, Sandia National Labs, 2016.

[127] Michael Reed and Barry Simon. *Methods of modern mathematical physics. vol. 1. Functional analysis*. Academic, 1980.

[128] James Reinders. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. " O'Reilly Media, Inc.", 2007.

[129] Michael Renardy and Robert C. Rogers. *An introduction to partial differential equations*, volume 13. Springer Science & Business Media, 2006.

[130] Arch D. Robison. Composable parallel patterns with Intel Cilk Plus. *Computing in Science & Engineering*, 15(2):66–71, 2013.

[131] Vladimir Rokhlin. Rapid solution of integral equations of classical potential theory. *J. Comput. Phys.*, 60(2):187–207, 1985.

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

[133] Yousef Saad. ILUT: A dual threshold incomplete lu factorization. *Numer. Linear Algebra Appl.*, 1(4):387–402, 1994.

[134] Yousef Saad. *Iterative methods for sparse linear systems*, volume 82. SIAM, 2003.

[135] Yousef Saad and Jun Zhang. BILUM: block versions of multielimination and multilevel ILU preconditioner for general sparse linear systems. *SIAM J. Sci. Comput.*, 20(6):2103–2121, 1999.

[136] Miloud Sadkane. Block-Arnoldi and Davidson methods for unsymmetric large eigenvalue problems. *Numer. Math.*, 64(1):195–211, 1993.

[137] Miloud Sadkane. A block Arnoldi-Chebyshev method for computing the leading eigenpairs of large sparse unsymmetric matrices. *Numer. Math.*, 64(1):181–193, 1993.

[138] Phillip G. Schmitz and Lexing Ying. A fast direct solver for elliptic problems on general meshes in 2D. *J. Comput. Phys.*, 231(4):1314–1338, 2012.

[139] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: a high-productivity programming language for HPC with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[140] Fengguang Song, Asim YarKhan, and Jack Dongarra. Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In *Proceedings of*

*the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11. IEEE, 2009.

[141] Weglein P.A.B. Stolt D.R.H. *Seismic Imaging and Inversion: Volume 1: Application of Linear Inverse Theory*. Cambridge University Press, 2012.

[142] Gilbert Strang and Tri Nguyen. The interplay of ranks of submatrices. *SIAM Rev.*, 46(4):637–646, 2004.

[143] Klaus Stüben. A review of algebraic multigrid. In *Partial Differential Equations*, pages 281–309. Elsevier, 2001.

[144] Daria A Sushnikova and Ivan V Oseledets. "Compress and Eliminate" solver for symmetric positive definite sparse matrices. *SIAM J. Sci. Comput.*, 40(3):A1742–A1762, 2018.

[145] Irina K. Tezaur, Mauro Perego, Andrew G. Salinger, Raymond S. Tuminaro, and Stephen F. Price. Albany/FELIX: a parallel, scalable and robust, finite element, first-order Stokes approximation ice sheet solver built for advanced analysis. *Geo. Model Dev. (Online)*, 8(4), 2015.

[146] Martin Tillenius. Superglue: A shared memory framework using data versioning for dependency-aware task-based parallelization. *SIAM J. Sci. Comput.*, 37(6):C617–C642, 2015.

[147] S. Tomov, J. Dongarra, V. Volkov, and J. Demmel. MAGMA library. *Univ. of Tennessee and Univ. of California, Knoxville, TN, and Berkeley, CA*, 2009.

[148] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010.

[149] Eugene Tyrtyshnikov. Incomplete cross approximation in the mosaic-skeleton method. *Computing*, 64(4):367–380, 2000.

[150] Henk A. Van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13(2):631–644, 1992.

[151] Thorsten Von Eicken, David E Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. *SIGARCH Computer Architecture News*, 20(2):256–266, 1992.

[152] Luiz C. Wrobel. *The Boundary Element Method, Volume 1: Applications in Thermo-Fluids and Acoustics*, volume 1. John Wiley & Sons, 2002.

[153] Zedong Wu and Tariq Alkhalifah. The optimized expansion based low-rank method for wavefield extrapolation. *Geophysics*, 79(2):T51–T60, 2014.

[154] Jianlin Xia. Efficient structured multifrontal factorization for general large sparse matrices. *SIAM J. Sci. Comput.*, 35(2):A832–A860, 2013.

[155] Jianlin Xia. Randomized sparse direct solvers. *SIAM J. Matrix Anal. Appl.*, 34(1):197–227, 2013.

[156] Jianlin Xia, Shivkumar Chandrasekaran, Ming Gu, and Xiaoye S. Li. Superfast multifrontal method for large structured linear systems of equations. *SIAM J. Matrix Anal. Appl.*, 31(3):1382–1411, 2009.

[157] Jianlin Xia, Shivkumar Chandrasekaran, Ming Gu, and Xiaoye S. Li. Fast algorithms for hierarchically semiseparable matrices. *Numer. Linear Algebra Appl.*, 17(6):953–976, 2010.

[158] Jianlin Xia and Ming Gu. Robust approximate Cholesky factorization of rank-structured symmetric positive definite matrices. *SIAM J. Matrix Anal. Appl.*, 31(5):2899–2920, 2010.

[159] Jianlin Zixing Xia. Effective and robust preconditioning of general SPD matrices via structured incomplete factorization. *SIAM J. Matrix Anal. Appl.*, 38(4):1298–1322, 2017.

[160] Kai Yang, Hadi Pouransari, and Eric Darve. Sparse hierarchical solvers with guaranteed convergence. *arXiv preprint arXiv:1611.03189*, 2016.

[161] Mihalis Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. Alg. Disc. Meth.*, 2(1):77–79, 1981.

[162] Asim YarKhan, Jakub Kurzak, and Jack Dongarra. Quark users' guide: Queueing and runtime for kernels. Technical Report ICL-UT-11-02, University of Tennessee, Innovative Computing Laboratory, 2011.

[163] Norman Yarvin and Vladimir Rokhlin. Generalized Gaussian quadratures and singular value decompositions of integral operators. *SIAM J. Sci. Comput.*, 20(2):699–718, 1998.

[164] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. UPC++: a PGAS extension for C++. In *28th International Parallel and Distributed Processing Symposium*, pages 1105–1114. IEEE, 2014.