The Dissertation Committee for Anna Yesypenko
certifies that this is the approved version of the following dissertation:

Randomized Algorithms for the Efficient Solution of Elliptic PDEs on Modern Architectures

Committee:

Per-Gunnar Martinsson, Supervisor

George Biros, Co-supervisor

Bjorn Engquist

Omar Ghattas

Karen Willcox

Randomized Algorithms for the Efficient Solution of Elliptic PDEs on Modern Architectures

by

Anna Yesypenko

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2024

Dedicated to my grandparents,

Mr. Stanislav Ivanovich and Mrs. Nadezhda Pavlovna.

Acknowledgments

I am grateful to my advisor Per-Gunnar Martinsson and my co-advisor George Biros for their invaluable guidance through this PhD. Their mentorship has played a pivotal role in shaping my research endeavors, and for that, I extend my heartfelt appreciation. I would also like to express my gratitude to my committee members, whose insights and feedback have been instrumental in shaping the direction of my research. The collective wisdom and support provided by my committee have enriched the quality of my work.

In particular, I want to express thanks to Gunnar for his unwavering support in encouraging me to explore and tackle complex, thought-provoking problems that align with my passions and intellectual curiosities. His guidance has not only fostered my intellectual growth but has also inspired me to push the boundaries of my capabilities. In addition, I am deeply grateful to Chao Chen for his mentorship and support throughout various stages of my PhD. His guidance has been a source of wisdom and encouragement.

A special and extended mention is due to Tom O'Leary-Roseberry. His unwavering encouragement and understanding have been the cornerstone of my support system. He has provided me with both intellectual insights and the stability needed to navigate the challenges of academia. Tom's belief in my capabilities and his unwavering dedication have been indispensible.

The combined efforts of these remarkable mentors, colleagues, family, and Tom, along with the nurturing environment at the Oden Institute, have been fundamental to completing this PhD. I am truly fortunate to have been surrounded by such dedicated individuals who have helped shape my academic and personal development.

Abstract

Randomized Algorithms for the Efficient Solution of Elliptic PDEs on Modern Architectures

Anna Yesypenko, PhD
The University of Texas at Austin, 2024

SUPERVISORS: Per-Gunnar Martinsson, George Biros

This thesis describes techniques for efficiently and robustly computing approximate solutions to elliptic partial differential equations (PDEs). It presents novel algorithms for solving linear systems that exploit randomized methods in linear algebra to attain high computational efficiency and scalability. These algorithms are designed to leverage a variety of compute kernels, such as vectorization and specialized hardware acceleration features found on modern architectures. The ideas in this thesis demonstrate viable paths to re-envisioning classical linear solvers in a changing hardware landscape.

The unifying theme of the work is the use of hierarchical matrices ($\mathcal{H}$-matrices) to accelerate key compute kernels. The term $\mathcal{H}$-matrix refers to a matrix that can hierarchically be tessellated into submatrices in such a way that each submatrix is either of low numerical rank, or small enough that it can be stored densely. Using $\mathcal{H}$-matrix representation for a matrix of size $N \times N$ allows the complexity of matrix-vector products and matrix inversion to have linear, or close to linear, complexity when dense linear algebra would require $\mathcal{O}(N^2)$ and $\mathcal{O}(N^3)$ operations, respectively. In this thesis, $\mathcal{H}$-matrix structure is used to represent discretized integral equations, as well as to represent large dense matrices that arise in sparse direct solvers for

discretized PDEs.

The thesis contains three key contributions. First, a novel fast multipole method (FMM) is presented. This is a fast algorithm for applying an $\mathcal{H}$-matrix to a vector, with applications in the numerical solution of integral equation formulations of PDEs using iterative methods. The method is based on linear algebraic tools such as randomized low rank approximation, and "skeleton representations" of far-field interactions. Its key feature is significantly simplified data structure compared to the original FMM.

Second, a sparse direct solver for discretized PDEs is presented, which is compatible with a variety of PDE discretizations. This work addresses a critical limitation of sparse direct solvers, which is that techniques such as LU and Cholesky lead to factors that are far less sparse than the original matrix; we ameliorate this "fill-in" effect by exploiting $\mathcal{H}$-matrix structure in the dense blocks that arise. This solver distinguishes itself from related work by using a decomposition of the computational domain into slabs. This approach leads to simplified data structures, which facilitate parallelization and also makes the scheme more amenable to using GPU acceleration for improved performance.

Third, a novel algorithm for the simultaneous compression and LU factorization of a particular class of $\mathcal{H}$-matrices is presented. This process is achieved in linear time from black-box matrix-vector products of an operator with $\mathcal{H}$-matrix structure. The work builds upon a previously proposed algorithm for "strong recursive skeletonization" but provides significant simplifications and accelerations. The work has applications in the direct solution of boundary integral equations and in sparse direct solvers for discretized PDEs.

Table of Contents

Chapter 1: Introduction

1.1 Problem formulation

The dissertation describes numerical methods for solving boundary value problem of the form

$$\begin{cases} \mathcal{A}u(x) = f(x), & x \in \Omega, \\ u(x) = g(x), & x \in \Gamma, \end{cases} \tag{1.1}$$

where $\mathcal{A}$ is a second order elliptic differential operator, and Ω is a rectangular domain in two or three dimensions with boundary Γ. For the sake of concreteness, we will for the most part focus on the case where $\mathcal{A}$ is a variable coefficient Helmholtz operator with appropriate boundary conditions

$$\mathcal{A}u(x) = -\Delta u(x) - \kappa^2 b(x) u(x), \tag{1.2}$$

where κ is a reference ("typical") wavenumber and $b(x)$ is a smooth non-negative function. This equation governs the phenomena of acoustic and electromagnetic scattering and has a wide variety of applications (e.g. medical imaging, sonar, seismology). There are a number of challenges to numerically solving (1.2); we highlight a few of them. First, the solutions u are highly oscillatory, and the number of discretization points needed grows as $N \sim \kappa^d$, where $d = \{2, 3\}$ is the spatial dimension of the domain Ω. Second, the discretized system (1.2) is indefinite, which causes issues for many numerical methods which are effective for coercive elliptic PDEs. Because the discretization of (1.2) is sparse, iterative methods are a natural choice; however, iterative methods often struggle to converge, especially in the presence of challenging wave phenomena (e.g. multiple reflections, trapped rays). This thesis focuses on direct solvers for computing solutions to (1.1).

Although a differential operator is local, the solution operator which maps given data f, g to the solution u is inherently global. A small perturbation in the given data causes the solution to change at every point in Ω. Solution operators (or

direct solvers) are often dense; however, they often have structure that allow them to be computed and applied to given boundary data efficiently. Direct solvers are particularly compelling for applications where the same PDE needs to be solved for many boundary data. The methods in this dissertation work for a broad range of elliptic PDEs, but are particularly competitive for (1.2), for which the solution is classically challenging to compute using iterative methods.

We describe fast methods for computing solvers for elliptic PDEs in two settings. In the first setting, we formulate (1.1) as an integral equation where the solution u appears implicitly. In the second setting, we discretize the PDE using a sparse discretization (e.g. finite differences) and factorize the resulting system to form a sparse direct solver.

1.1.1 Integral equation formulation

In this section, we describe how to formulate the elliptic BVP (1.1) using an integral equation formulation. For simplicity, consider Ω a simply connected set and Γ a smooth boundary. Consider as well that the body load $f = 0$ and that $b \equiv 1$ in Ω. We start with an ansatz that the solution u could be expressed as convolution of the following form

$$u(x) = \int_\Gamma K(x-y)\sigma(y)ds(y), \qquad x \in \Omega, \tag{1.3}$$

where K is the known analytic Green's function for the PDE and σ is an unknown function on the boundary Γ. The solution $u(x)$ is continuous on Ω, including up to the boundary. In order for the given boundary data $g(x)$ to be satisfied, the following boundary integral equation (BIE) must be satisfied

$$g(x) = \int_\Gamma K(x-y)\sigma(y)ds(y) \qquad x \in \Gamma. \tag{1.4}$$

The equations (1.4) and (1.3) together provide a means of solving the elliptic BVP. First, solve the BIE in (1.4) for the unknown $\sigma(x)$ on the boundary Γ. Then, the solution $u(x)$ can be computed by evaluating the ansatz (1.3).

This formulation significantly reduces the dimensionality of the problem because we need to solve the BIE (1.4) on the boundary Γ instead solving the PDE (1.1) in the domain Ω. There are, however, a number of challenges to integral equation formulations. First, the Green's function K is often singular at the origin, and special quadrature rules are required to discretize (1.4) accurately. Second (and importantly, a focus of this dissertation), the discretized equations (1.3) and (1.4) are dense. Luckily, there are rank-deficiencies in the discretized system which allow the fast solution of (1.4); these approaches are discussed further in Section 1.2.1.

Among the benefits of boundary integral equations, is that the formulation can lead to better conditioned systems than discretizations of PDEs. For example, discretizing (1.1) using finite differences for mesh size h leads to a system with condition number $\mathcal{O}(h^{-2})$. Poor conditioning leads to a number of issues, among them, that it may cause a loss of accuracy when solving the discretized system. The formulation of the boundary integral equation in (1.4) leads to a discretized system with condition number $\mathcal{O}(h^{-1})$, which is already an improvement. Using what is called a 'second-kind Fredholm' integral equation formulation can lead to a condition number which is independent of the discretization parameter h [68, Ch.10], allowing the discretized system to be solved to very high accuracy.

Remark 1.1. Note that we have made a number of assumptions that there is no body load and that the elliptic PDE has constant coefficients to formulate the BIE in (1.4). Solving PDEs using integral equations is possible without these assumptions, but the formulation in (1.4) will instead involve solving an integral equation over Ω, leading to a dense discretized system to solve on a volume. When BIE formulations are possible, they provide an compelling means for efficiently solving elliptic PDEs because the dimensionality of the problem is reduced (e.g. from a 3D volume to a 2D surface).

1.1.2 Sparse Direct Solvers for PDEs

Discretizing (1.1) using a local discretization such as finite differences, finite elements, finite volume, etc., results in a sparse linear system

$$\mathbf{Au} = \mathbf{f}. \tag{1.5}$$

The Helmholtz equation (1.2) discretized with low-order elements can suffer from the effects of pollution [7] for large κ, leading to large errors. This, however, can be overcome with the use of high order discretization, which we incorporate as a part of this thesis. There are a number of challenges in solving (1.5) using iterative methods [34, 39], especially when challenging scattering phenomena are present. The system (1.5) can instead be solved by first computing a sparse factorization of $\mathbf{A}$ of the following form

$$\mathbf{A} = \mathbf{LU} \tag{1.6}$$

where $\mathbf{L}$ is lower triangular and $\mathbf{U}$ is upper triangular, and both factors are sparse. Once (1.6) has been computed, the solution $\mathbf{u}$ can be computed using forward and backward substitution $\mathbf{u} = \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{f}$, and the computed factorization can be reused for multiple right hand sides $\mathbf{f}$.

A challenge of sparse direct solvers is the cost of computing and storing the factorization (1.6). The factorization is a representation of a dense solution operator, and as a result, the factors $\mathbf{L}$ and $\mathbf{U}$ are often much less sparse than the original system $\mathbf{A}$. Luckily, the dense "fill-in" of the factorization (1.6) has structure that allows for the factorization to be computed and stored efficiently.

1.2 $\mathcal{H}$-matrices in the context of PDE solvers

The unifying theme of this thesis is the use of $\mathcal{H}$-matrices, both in the context of Section 1.1.1 and in the context of Section 1.1.2, to accelerate key compute kernels. The term $\mathcal{H}$-matrix refers to a matrix that can hierarchically be tessellated into submatrices in such a way that each submatrix is either of low numerical rank, or

small enough that it can be stored densely. Using this representation for a matrix of size $N \times N$ allows the complexity of matrix-vector products and matrix inversion to have linear, or close to linear, complexity when dense linear algebra would require $\mathcal{O}(N^2)$ and $\mathcal{O}(N^3)$ operations, respectively. For a simple example of an $\mathcal{H}$-matrix, see Figure 1.1.

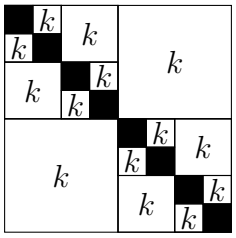


Figure 1.1: Hierarchically Off-Diagonal Low-Rank (HODLR) matrix [8] of size $n \times n$ with off-diagonal rank k. The matrix can be applied in $\mathcal{O}(n \log n)$ time and factorized in $\mathcal{O}\left(n(\log n)^2\right)$ time.

In this section, we describe how $\mathcal{H}$-matrices arise in the context of integral equations, as well as in the context of sparse direct solvers, providing a brief literature review as well as contextualizing the contributions of this thesis.

1.2.1 Fast Solvers for Discretized Integral Equations

For a constant coefficient elliptic PDE in the absence of a body load, the elliptic BVP (1.1) on a domain Ω can be formulated as an integral equation (1.4) on the boundary Γ. As discussed in Section 1.1.1, integral equations can be formulated to be very well conditioned, so the condition number is independent of the number of discretization points. This leads to a number of benefits, including that the discretized equation could be solved using iterative methods. The challenge, however, is that discretizing (1.4) leads to a dense system of equations.

Suppose that (1.4) is discretized using N collocation points on the boundary. In order to solve (1.4) using an iterative method, we need to evaluate an N-body sum

of the following form

$$u_i = \sum_{j=1, i \neq j}^{N} K(x_i - x_j)\, q_j, \qquad i = 1, \ldots N, \tag{1.7}$$

where $\{x_i\}_{i=1}^N$ is a set of points on the boundary Γ at which the functions u and q are collocated. The function K is a given kernel function that is associated with a standard elliptic PDE of mathematical physics. For instance, K may be the fundamental solution of Helmholtz equation,

$$K(r) = \begin{cases} \frac{i}{4} H_0^{(1)}(\kappa \|r\|), & x_i \neq x_j \in \mathbb{R}^2 \\ -\frac{1}{4} \frac{e^{i\kappa\|r\|}}{\|r\|}, & x_i \neq x_j \in \mathbb{R}^3, \end{cases} \tag{1.8}$$

where $H_0^{(1)}$ is the Hankel function of the first kind of order zero. It is often convenient to formulate the summation problem (1.7) as the matrix-vector product

$$\mathbf{u} = \mathbf{A}\,\mathbf{q} \tag{1.9}$$

where $\mathbf{A}$ is the $N \times N$ matrix with off-diagonal entries $\mathbf{A}_{ij} = K(x_i, x_j)$ and zeros on the diagonal, and where $\mathbf{u} = [u_1, \ldots, u_N]$ and $\mathbf{q} = [q_1, \ldots, q_N]$.

The fast multipole method (FMM) is a general framework for computing (1.9) approximately within a given tolerance ε using only $\mathcal{O}(N)$ operations. The method partitions the problem domain hierarchically into subdomains of different scales and exploits the multi-scale decomposition in the evaluation of (1.7). In particular, the method evaluates exactly the calculation associated with adjacent subdomains at the finest scale, and approximately evaluates the calculation between non-adjacent subdomains at every scale. Overall, the procedure requires $\mathcal{O}(N)$ operations to compute an approximation within a given tolerance ε.

The FMM is an analytic method has been derived for specific kernels that appear frequently in computational physics [37, 38, 48, 51, 102], however the framework is general and analytic function expansions are not necessarily needed to evaluate (1.9) quickly. There is a vast literature of similar methods developed using linear

algebraic tools [64, 70, 101, 103], including the method of Chapter 2 which we summarize in Section 1.4.1. The FMM led to the development of a more general algebraic framework for $\mathcal{H}$-matrices [12, 53, 94], which are applicable to a wide set of problems in computational science.

1.2.2 Fast Sparse Direct Solvers for Discretized PDEs

In this section, we summarize some key facts about direct solvers for a linear system such as (1.5), resulting from the discretization of (1.1) using finite differences, finite elements, etc. The objective is often to calculate an LU factorization

$$\mathbf{PAQ}^T = \mathbf{LU}, \tag{1.10}$$

where $\mathbf{P}, \mathbf{Q}$ are a permutation matrices, $\mathbf{L}$ is lower triangular, and $\mathbf{U}$ is upper triangular. For dense LU factorizations, the rows are pivoted to avoid instability in the factorization. For sparse LU, permutations are selected to balance numerical stability on the one hand, with the need to maintain sparsity in $\mathbf{L}$ and $\mathbf{U}$ on the other.

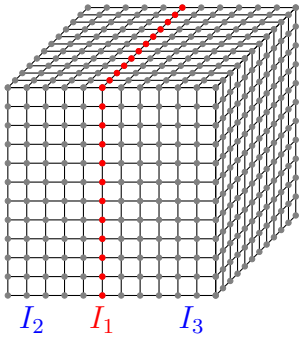


Figure 1.2: Mesh partitioning for a 3D cube of size $N = n^3$.

Suppose that a mesh of $I = (1 : N)$ points is partitioned into index sets $I = [I_1, I_2, I_3]$, as shown in Figure 1.2. For an appropriate permutation of $\mathbf{A}$, we can compute a sparse factorization of $\mathbf{A}$ of the following form

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{33} & & \mathbf{A}_{31} \\ & \mathbf{A}_{22} & \mathbf{A}_{21} \\ \mathbf{A}_{13} & \mathbf{A}_{12} & \mathbf{A}_{11} \end{bmatrix} = \begin{bmatrix} \mathbf{L}_{33} & & \\ & \mathbf{L}_{22} & \\ \mathbf{L}_{13} & \mathbf{L}_{12} & \mathbf{L}_{11} \end{bmatrix} \begin{bmatrix} \mathbf{U}_{33} & & \mathbf{U}_{31} \\ & \mathbf{U}_{22} & \mathbf{U}_{21} \\ & & \mathbf{U}_{11} \end{bmatrix}, \tag{1.11}$$

where $\mathbf{L}_{11}\mathbf{U}_{11}$ is an LU factorization of Schur complement

$$\mathbf{L}_{11}\mathbf{U}_{11} = \underset{n^2 \times n^2}{\mathbf{S}_{11}} := \mathbf{A}_{11} - \mathbf{A}_{12}\mathbf{A}_{22}^{-1}\mathbf{A}_{21} - \mathbf{A}_{13}\mathbf{A}_{33}^{-1}\mathbf{A}_{31}. \tag{1.12}$$

Though the factors in (1.12) are sparse, their composition is dense. The cost to factorize $\mathbf{S}_{11}$ is $\mathcal{O}(n^6) = \mathcal{O}(N^2)$ flops, and the memory requirements to store the factors $\mathbf{L}_{11}, \mathbf{U}_{11}$ are $\mathcal{O}(n^4) = \mathcal{O}(N^{4/3})$. In order to efficiently compute $\mathbf{A}_{22}$ and $\mathbf{A}_{33}$, the mesh can be partitioned recursively in a "nested dissection" ordering [31, 40]; however, the dominant costs are factoring and storing $\mathbf{S}_{11}$. The nested dissection ordering is discussed further in Section 1.4.2 to contextualize the work of Chapter 3.

The compute and memory requirements of sparse LU limit the scalability of the method, especially for 3D problems. The dense fill-in is rich in rank-structure, which allows us to factorize dense matrices efficiently. Leveraging rank-structured matrix algebra in the context of sparse LU gives rise to a class of "accelerated sparse direct methods" [5, 15, 95], which can scale to larger problem sizes than sparse direct solvers. To give physical intuition on the rank structures, we give an interpretation of the Schur complement $\mathbf{S}_{11}$. Consider the following matrix

$$\mathbf{A}_{12}\ \mathbf{A}_{22}^{-1}\mathbf{A}_{21}, \tag{1.13}$$

where $|I_1| = n^2$ and $|I_2| = n^3/2$. Consider that Dirichlet data $\mathbf{g}$ is on interface I_1. Then the operator

$$\mathbf{u} := \mathbf{A}_{22}^{-1}\mathbf{A}_{21}\mathbf{g}$$

maps Dirichlet data to the solution of the PDE on the interior. Applying $\mathbf{h} := \mathbf{A}_{12}\mathbf{u}$ to the solution on the interior differentiates the solution and produces Neumann data $\mathbf{h}$ that is consistent with the given Dirichlet data $\mathbf{g}$. We call (1.13) a Dirichlet-to-Neumann operator. Surface-to-surface operators of this type are dense but often have $\mathcal{H}$-matrix structure that allow them to be applied and inverted quickly.

1.3 Randomized compression for $\mathcal{H}$-matrices

Dense matrices with $\mathcal{H}$-matrix structure arise in solution of elliptic PDEs in many contexts. In the setting of integral equations, there are often efficient means to compress $\mathcal{H}$-matrix structure because the matrix $\mathbf{A}$ of equation (1.9) arises from the evaluation of a known Green's function. In the setting of sparse direct solvers, however, compressing $\mathcal{H}$-matrix structure is not as straightforward because accessing individual matrix entries of (1.12) is computationally expensive.

In this section, we provide an overview of methods to recover structure of a matrix $\mathbf{A}$ via the action of the matrix and its adjoint on vectors, e.g. $\mathbf{v} \mapsto \mathbf{A}\mathbf{v}$ and $\mathbf{v} \mapsto \mathbf{A}^*\mathbf{v}$. In situations where a fast matrix-vector product is available, $\mathbf{A}$ can often be recovered using a small number of samples. These algorithms are particularly useful in the setting of direct solvers because the Schur complement (1.12) is a composition of sparse matrices. In Section 1.3.1, we discuss fairly well known algorithms for low rank approximation. These techniques can be generalized to $\mathcal{H}$-matrix structure, as we discuss in Section 1.3.2.

1.3.1 Low rank

Consider a matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ which has exact rank k. We can compute a low rank approximation using random sketching. First, generate the sketch

$$\underset{m\times k}{\mathbf{Y}} := \underset{m\times n}{\mathbf{A}}\ \underset{n\times k}{\mathbf{\Omega}} \tag{1.14}$$

where $\mathbf{\Omega}$ is a Gaussian random matrix. With high probability, the sketch $\mathbf{Y}$ spans the column space of $\mathbf{A}$. We then orthonormalize the sample $\mathbf{Q} = \text{orth}(\mathbf{Y})$ to form an orthonormal basis. The remaining steps use classical techniques, by calculating the QR of the smaller matrix $\mathbf{B} := \mathbf{A}^*\mathbf{Q}$ as

$$\hat{\mathbf{Q}}\mathbf{R} = \text{qr}(\mathbf{B}).$$

The resulting QR factorization of $\mathbf{A}$ is

$$\mathbf{A} = \left(\mathbf{Q}\hat{\mathbf{Q}}\right)\mathbf{R}.$$

The complexity is $T_{\text{mult}}k + O(k^2 n)$, where T_{mult} is the complexity of multiplying $\mathbf{A}$ by a vector. In the case that the rank is not exact but decays quickly, we can recover the factorization to high accuracy by sampling with a Gaussian random matrix with $k+p$ columns, where p is a small oversampling parameter. In practice, setting $p = 10$ often yields good numerical results.

1.3.2 Randomized Black-Box Algorithms for $\mathcal{H}$-matrices

If $\mathbf{A}$ is compressible as a rank-structured matrix, there are efficient algorithms to recover the factorization using similar operations to those described above, by post-processing random samples

$$\underset{N\times p}{\mathbf{Y}} = \underset{N\times N}{\mathbf{A}}\ \underset{N\times p}{\mathbf{\Omega}}, \qquad \underset{N\times p}{\mathbf{Z}} = \underset{N\times N}{\mathbf{A}^*}\ \underset{N\times p}{\mathbf{\Psi}} \tag{1.15}$$

where $\mathbf{\Omega}, \mathbf{\Psi}$ are random matrices and $p \ll N$. Different rank-structures require different choices of $\mathbf{\Omega}, \mathbf{\Psi}$ and post-processing steps [63, 73]. Some algorithms may also require direct access to a small number of matrix entries, which may be prohibitively expensive or inconvenient to compute efficiently, e.g. for $\mathbf{A}$ which is represented as a composition of sparse matrices. In this thesis, we focus on compression algorithms which are truly 'black-box' in that they only access the matrix through matrix-vector products.

To demonstrate how these methods work, we will briefly describe one of the techniques for black-box $\mathcal{H}$-matrix sampling from [59, 60]. Consider a 'hierarchically block-separable' matrix (HBS) $\mathbf{A}$, which in spirit, is similar to the HODLR matrix of Figure 1.1. Tessellate the $N \times N$ matrix into $b \times b$ blocks of size $m \times m$, where $mb = N$, so that

$$\mathbf{A} = \begin{bmatrix} \mathbf{D}_1 & \mathbf{A}_{12} & \dots & \mathbf{A}_{1b} \\ \mathbf{A}_{21} & \mathbf{D}_2 & \dots & \mathbf{A}_{2b} \\ \vdots & \vdots & & \vdots \\ \mathbf{A}_{b1} & \mathbf{A}_{b2} & \dots & \mathbf{D}_b \end{bmatrix} \tag{1.16}$$

and each off-diagonal block admits the factorization

$$\underset{m\times m}{\mathbf{A}_{ij}} = \underset{m\times k}{\mathbf{U}_i}\ \underset{k\times k}{\widetilde{\mathbf{A}}_{ij}}\ \underset{k\times m}{\mathbf{V}_j}^*, \qquad i,j\in\{1,\ldots,b\},\ i\neq j. \tag{1.17}$$

When (1.17) holds, the matrix admits a block factorization

$$\underset{N\times N}{\mathbf{A}} = \underset{N\times bk}{\mathbf{U}}\ \underset{bk\times bk}{\widetilde{\mathbf{A}}}\ \underset{bk\times N}{\mathbf{V}^*} + \underset{N\times N}{\mathbf{D}},$$

where

$$\begin{aligned} \mathbf{U} &= \operatorname{diag}(\mathbf{U}_1,\ldots,\mathbf{U}_b), \\ \mathbf{V} &= \operatorname{diag}(\mathbf{V}_1,\ldots,\mathbf{V}_b), \\ \mathbf{D} &= \operatorname{diag}(\mathbf{D}_1,\ldots,\mathbf{D}_b) \end{aligned} \qquad \widetilde{\mathbf{A}} = \begin{bmatrix} \mathbf{0} & \widetilde{\mathbf{A}}_{12} & \ldots & \widetilde{\mathbf{A}}_{1b} \\ \widetilde{\mathbf{A}}_{21} & \mathbf{0} & \ldots & \widetilde{\mathbf{A}}_{2b} \\ \vdots & \vdots & & \vdots \\ \widetilde{\mathbf{A}}_{b1} & \widetilde{\mathbf{A}}_{b2} & \ldots & \mathbf{0} \end{bmatrix}. \tag{1.18}$$

To achieve linear storage costs for this representation of $\mathbf{A}$, the matrix $\widetilde{\mathbf{A}}$ can be tessellated into a coarser block partitioning and compressed as (1.16). Once a hierarchical decomposition of $\mathbf{A}$ is available, linear-time algorithms for computing $\mathbf{A}^{-1}$ are also available.

Consider that we would like to recover the structure (1.16) in the black-box setting of (1.15). In particular, our aim is to recover the basis $\mathbf{U}_i$ for the column space for each block $i = 1,\ldots,b$. Ideally, the test matrix $\mathbf{\Omega}$ would have structure that corresponds to the block we would like to sample. As an example, let us design structured test matrix $\mathbf{\Omega}'$ for block $i = 2$ of the form

$$\underset{N\times k}{\mathbf{\Omega}'} = \begin{bmatrix} \underset{m\times k}{\mathbf{\Omega}'_1} \\ \mathbf{0} \\ \mathbf{\Omega}'_3 \\ \vdots \\ \mathbf{\Omega}_b \end{bmatrix} \tag{1.19}$$

applying the test matrix to $\mathbf{A}$ and extracting the I_2 block would give a basis for $\mathbf{U}_2$:

$$\mathbf{U}_2 = \texttt{orth}(\mathbf{Y}'_2), \qquad \text{where } \underset{N\times k}{\mathbf{Y}'} = \underset{N\times N}{\mathbf{A}}\ \underset{N\times k}{\mathbf{\Omega}'}. \tag{1.20}$$

Designing structured test matrices of the form in equation (1.19) would lead to high sampling costs. Instead, we use another approach that uses far fewer samples and

slightly higher post-processing costs. Consider that we draw instead $p = (m + k)$ samples with a fully dense Gaussian random matrix $\mathbf{\Omega} \in \mathbb{R}^{N \times p}$ and post-process $\mathbf{\Omega}$ to compute a structured test matrix of the form in (1.19). Let

$$\underset{p \times k}{\mathbf{N}} = \mathtt{null}(\underset{m \times p}{\mathbf{\Omega}_2}) \tag{1.21}$$

be an orthogonal basis for the null-space of $\mathbf{\Omega}_2$. Then applying $\mathbf{N}$ on the right of $\mathbf{\Omega}$ gives

$$\underset{N \times p}{\mathbf{\Omega}} \underset{p \times k}{\mathbf{N}} = \begin{bmatrix} \underset{m \times p}{\mathbf{\Omega}_1} \underset{p \times k}{\mathbf{N}} \\ \mathbf{0} \\ \mathbf{\Omega}_3 \, \mathbf{N} \\ \vdots \\ \mathbf{\Omega}_b \, \mathbf{N} \end{bmatrix} := \underset{N \times k}{\mathbf{\Omega}'} \tag{1.22}$$

This observation gives an efficient means of computing bases for *all* blocks $i = 1, \ldots, b$ using a sketch of the form

$$\underset{N \times p}{\mathbf{Y}} = \underset{N \times N}{\mathbf{A}} \underset{N \times p}{\mathbf{\Omega}}, \qquad \text{where } \mathbf{\Omega} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \text{ and } p = m + k. \tag{1.23}$$

Note that to compute a basis for $\mathbf{U}_2$ in equation (1.20), only the $\mathbf{Y}_2'$ block needs to be used. Then the desired bases for all blocks $i = 1, \ldots, b$ can be computed as

$$\mathbf{U}_i = \mathtt{orth}(\mathbf{Y}_i'), \qquad \text{where } \underset{m \times k}{\mathbf{Y}_i'} = \underset{m \times p}{\mathbf{Y}_i} \underset{p \times k}{\mathtt{null}(\mathbf{\Omega}_i)}. \tag{1.24}$$

This technique is called *block nullification*. This technique, in conjunction with others, allows $\mathcal{H}$-matrices can be efficiently recovered in the fully black-box setting. The algorithm presented in [60] is applied to represent dense Schur complements with much less storage for a 2D sparse direct solver. This solver is the subject of Chapter 3; a brief overview is provided in Section 1.4.2.

The $\mathcal{H}$-matrix structure presented in (1.16), however, not expressive enough to efficiently represent dense structures which arise on 3D surfaces. The representation needed instead, is called an $\mathcal{H}^2$ matrix with strong admissibility, and can be seen as a linear-algebraic formalism for the FMM. A substantial challenge for this

representation is in computing a sparse representation of the inverse $\mathbf{A}^{-1}$. Section 1.4.3 provides an overview for a black-box algorithm which simultaneous compresses *and factorizes* an $\mathcal{H}^2$ matrix using black-box samples in the setting (1.15); this work is the subject of Chapter 4.

1.4 Overview of Chapters

1.4.1 SkelFMM: A Simplified Fast Multipole Method

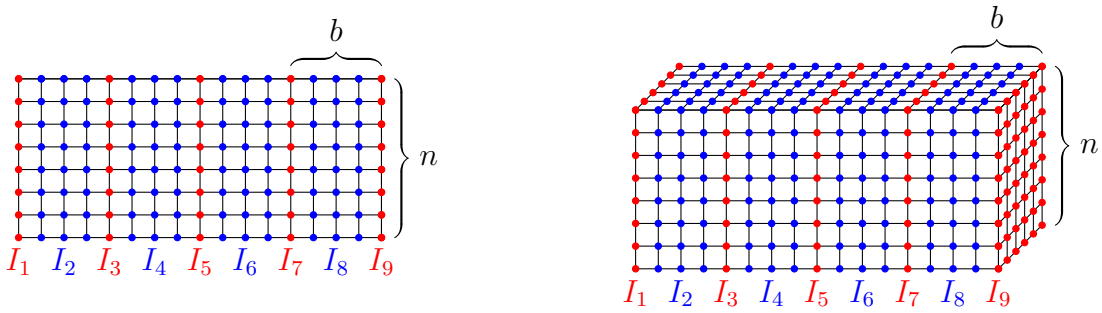
The fast multipole method (FMM) is a framework for the efficient computation of N-body sums of the form (1.7). The FMM can be used to solve discretized boundary integral equation (1.4) using an iterative method, which is a particularly effective approach when a well-conditioned formulation is used. Though the FMM has been derived for a variety of kernels using analytic techniques, the derivation may be tedious and difficult for an arbitrary kernel function. Therefore, algebraic approaches have been developed, which require only the evaluation of a given kernel function [64, 70, 101, 103]. SkelFMM is based on linear algebraic tools such as randomized low rank approximation and "skeleton representations" of far-field interactions.

The work is related to previously proposed linear algebraic reformulations of the FMM, but is distinguished by relying on simpler data structures. In particular, skelFMM does not require an "interaction list", as it relies instead on algebraically-modified kernel interactions between near-neighbors at every level. (The "W-list" and the "X-list" of classical FMMs [49] are not needed either.) The simplicity of the algorithm makes it particularly amenable to parallel implementation on heterogeneous hardware architectures. After a precomputation stage which builds a tailored representation for a given set of points, the matrix-vector product achieves high speed on the GPU, leveraging highly-tuned batched BLAS primitives.

1.4.2 SlabLU: A Two-Level Sparse Direct Solver

Sparse direct solvers, for which we provided an overview in Section 1.2.2, often rely on a nested dissection ordering to achieve competitive complexity for the cost of factorizing and storing (1.10). The nested dissection order is often multi-level and defined on a quad-tree or oct-tree data structure. Our approach is based on decomposing the domain into thin "slabs" of width b, as illustrated in Figure 1.3, instead yielding a two-level scheme.

Slab-based solvers are an alternative to multi-level nested dissection schemes [31, 40]. Multi-level nested dissection schemes attain better asymptotic complexity, while two-level slab based schemes can often attain higher practical performance since they are simple to parallelize and optimize for high performance on modern CPU/GPU architectures. A key finding of our work is that even basic optimizations of a two-level scheme attain practical speed that compares favorably to state-of-the-art nested dissection methods. Our slab-based scheme is also closely related to domain decomposition (DD) methods [17, 85, 88].



(a) Slab decomposition for 2D mesh.

(b) Slab decomposition for 3D mesh.

Figure 1.3

Slab-based schemes have a two-stage structure. In the first stage, we "eliminate" the even-numbered index sets by constructing sparse direct solvers that allow us to implicitly apply the operators

$$\mathbf{A}_{22}^{-1},\ \mathbf{A}_{44}^{-1},\ \ldots,\ \mathbf{A}_{2n/b}^{-1}.$$

We exploit the sparsity of these blocks, and also that this step is trivially parallelizable.

The elimination of the even-numbered nodes are eliminated results in a linear system with a "reduced matrix" $\mathbf{T}$ that encodes the interactions on the remaining slab interfaces. In the second stage of the solver, we compute an LU factorization of the block tridiagonal matrix $\mathbf{T}$. Observe that the blocks of $\mathbf{T}$ are dense. However, they typically have internal structure that can be exploited to accelerate the factorization process. In particular, the sub-blocks of $\mathbf{T}$ are compressible as a 'hierarchically block separable' matrices of the form (1.16) for the 2D mesh in Figure 1.3a and are recovered using the black-box algorithm of [60]. This $\mathcal{H}$-matrix structure, however, is not expressive enough to efficiently store dense structures for the 3D mesh in Figure 1.3b, motivating the work in Section 1.4.3.

1.4.3 Randomized Strong Recursive Skeletonization: Simultaneous Compression and Factorization of $\mathcal{H}^2$ Matrices

As described in Sections 1.2.1 and 1.2.2, dense structured matrices often arise in the context of PDE solvers. Techniques such as the FMM exploit mathematical properties of the kernel function directly to enable the fast application of dense matrices to vectors. The $\mathcal{H}^2$-matrix methodology is a reinterpretation of the FMM in linear algebraic terms, where a matrix is tessellated into blocks in such as way that each block is either small or of numerically low rank. This reinterpretation opened the door to linear complexity algorithms for a wider range of algebraic operations, including the matrix-matrix multiplication, and the construction of invertible factorizations.

Randomized Strong Recursive Skeletonization (RSRS) is an algorithm for simultaneously compressing and inverting an $\mathcal{H}^2$-matrix, given a means of applying the matrix and its adjoint to vectors. The precise problem formulation is this: Suppose that $\mathbf{A}$ is an $\mathcal{H}^2$-matrix, and that you are given a fast method for applying $\mathbf{A}$ and its adjoint $\mathbf{A}^*$ to vectors. We then seek to build two "test-matrices" $\mathbf{\Omega}$ and $\mathbf{\Psi}$ with the property that the $\mathcal{H}^2$ representation of $\mathbf{A}^{-1}$ can be constructed from the set

$\{\mathbf{Y}, \mathbf{Z}, \mathbf{\Omega}, \mathbf{\Psi}\}$, where

$$\mathbf{Y} = \mathbf{A}\mathbf{\Omega} \qquad \text{and} \qquad \mathbf{Z} = \mathbf{A}^*\mathbf{\Psi}.$$

RSRS is immediately applicable in a range of important environments. First, it can be used to derive a rank-structured representation of any integral operator for which a fast matrix-vector multiplication algorithm, such as the Fast Multipole Method [48, 50], is available. In this context, RSRS will directly output an invertible factorization, which is useful for the direct solver for boundary integral equations, cf. 1.2.1. Second, it can greatly simplify algebraic operations involving products of rank-structured or sparse matrices, as they arise in the context of sparse direct solvers, cf. 1.2.2.

1.5 Contributions by Area

Area A

- The work reported in Chapter 2 contains a complexity analysis that draws on results of mathematical physics to bound the (numerical) rank of matrices that represent interactions between different parts of the computational domain. It also relies on results from potential theory to execute the low rank compression step efficiently.

- We analyzed rank structure of sparse Schur complements of the form

  $$\mathbf{T}_{11} = \mathbf{A}_{11} - \mathbf{A}_{11}\mathbf{A}_{22}^{-1}\mathbf{A}_{21}$$

  arising from the slab decomposition of Figure 1.3. This allows us to justify the complexity of the algorithm in Section 3; see Appendix A.1 for this analysis.

- Randomized linear algebra plays a large role in the black-box algorithm of Chapter 4. Compression schemes apply statistical and linear algebraic techniques to estimate the error at various stages of the compression and adaptively change

the compression rank to meet the desired accuracy of the computed factorization. Properties of Gaussian matrices are exploited to reconstruct matrices that were probed through randomized sketching.

Area B

- SkelFMM, cf. Chapter 2, is a linear algebraic framework for the N-body problem which builds on previously proposed linear algebraic Fast multipole methods (FMMs). The novelty of skelFMM is that it operates on much simpler data structures. In particular, it does not involve the "interaction list" of the classical FMM and operates only on the "neighbor list" at every level. The simplicity of the new algorithm allows for the straightforward implementation on modern hardware.

- SlabLU, cf. Chapter 3, presents an effective sparse direct solver for a variety of PDE discretizations. The method uses a domain decomposition into slabs, as opposed to a multi-level nested dissection approach. The resulting algorithm has slightly higher flop count but is far simpler to parallelize and implement on modern architectures, as the numerical results demonstrate.

- The work of Chapter 4 is an algorithm for compressing and factorizing an important class of $\mathcal{H}$-matrices, called $\mathcal{H}^2$ matrices with strong admissibility. These matrices can be seen as a linear algebraic formulation of the FMM. This algorithm is completely black-box in that it only relies on accessing the operator through its action and the action of its adjoint. The algorithm is immediately applicable to direct solvers for boundary integral equations and to sparse direct solvers.

- Experimental results demonstrate the accuracy and scalability of each algorithm. The work of Chapters 2 and 3 incorporate parallelism and implementation on GPUs with batched linear algebra.

Area C

- SkelFMM, cf. Chapter 2, can be used as an efficient iterative solver for exterior scattering problems for boundary integral discretizations. We have conducted numerical experiments for the low-frequency Helmholtz equation which demonstrate that the method is particularly effective on 3D surfaces.

- We have conducted numerical experiments for variable-coefficient Helmholtz problems of physical relevance to electromagnetic and acoustic scattering; this is a large portion of the work of Chapter 3 which includes high order discretizations. We have demonstrated that the solver can attain high accuracy for applications with strong back-scattering. In particular, we have investigated applications to photonic crystals and resonant cavities.

- Randomized strong recursive skeletonization, cf. Chapter 4, is a black-box algorithm that enables the efficient solution of boundary integral equations and the acceleration of sparse direct solvers. We demonstrate in our numerical experiments that it is effective at factorizing sparse Schur complements arising from 3D PDE discretizations.

Chapter 2: SkelFMM: A Simplified Fast Multipole Method Based on Recursive Skeletonization [1]

This work introduces the kernel-independent multi-level algorithm "skelFMM" for evaluating all pairwise interactions between N points connected through a kernel such as the fundamental solution of the Laplace or the Helmholtz equations. The method is based on linear algebraic tools such as randomized low rank approximation and "skeleton representations" of far-field interactions. The work is related to previously proposed linear algebraic reformulations of the fast multipole method (FMM), but is distinguished by relying on simpler data structures. In particular, skelFMM does not require an "interaction list", as it relies instead on algebraically-modified kernel interactions between near-neighbors at every level. Like other kernel independent algorithms, it only requires evaluation of the kernel function, allowing the methodology to easily be extended to a range of different kernels in 2D and 3D. The simplicity of the algorithm makes it particularly amenable to parallel implementation on heterogeneous hardware architectures.

The performance of the algorithm is demonstrated through numerical experiments conducted on uniform and non-uniform point distributions in 2D and 3D, involving Laplace and (low frequency) Helmholtz kernels. The algorithm relies on a precomputation stage that constructs a tailored representation for a given geometry of points. Once the precomputation has completed, the matrix-vector multiplication attains high speed through GPU acceleration that leverages batched linear algebra.

[1] This work was completed in collaboration with Per-Gunnar Martinsson and Chao Chen. It has appeared in preprint [99].

2.1 Introduction

We present an algorithm for evaluating a sum of the form

$$u_i = \sum_{j=1, i \neq j}^{N} K(x_i, x_j)\ q_j, \qquad i = 1, \ldots N \tag{2.1}$$

where $\mathcal{X} = \{x_i\}_{i=1}^N$ is a given set of points in $\mathbb{R}^2$ or $\mathbb{R}^3$, and where K is a given kernel function that is associated with a standard elliptic PDE of mathematical physics. For instance, K may be the fundamental solution of Laplace's equation,

$$K(x_i, x_j) = \begin{cases} -\frac{1}{2\pi} \log\left(\|x_i - x_j\|\right) & x_i \neq x_j \in \mathbb{R}^2 \\ -\frac{1}{4\pi \|x_i - x_j\|} & x_i \neq x_j \in \mathbb{R}^3. \end{cases} \tag{2.2}$$

The task of evaluating a sum such as (2.1) arises frequently in particle simulations, in computational chemistry, in solving boundary integral equations [48, 50, 84] using iterative methods, and in many other contexts. It is often convenient to formulate the summation problem (2.1) as the matrix-vector product

$$\mathbf{u} = \mathbf{A}\ \mathbf{q}, \tag{2.3}$$

where $\mathbf{A}$ is the $N \times N$ matrix with off-diagonal entries $\mathbf{A}_{ij} = K(x_i, x_j)$ and zeros on the diagonal, and where $\mathbf{u} = [u_1, \ldots, u_N]$ and $\mathbf{q} = [q_1, \ldots, q_N]$.

The fast multipole method (FMM) is a general framework for computing (2.3) approximately within a given tolerance ε using only $\mathcal{O}(N)$ operations. The method partitions the problem domain hierarchically into subdomains of different scales and exploit the multi-scale decomposition in the evaluation of (2.1). In particular, the method evaluates exactly the calculation associated with adjacent subdomains at the finest scale, and approximately evaluates the calculation between non-adjacent subdomains at every scale. Overall, the procedure requires $\mathcal{O}(N)$ operations to compute an approximation within a given tolerance ε. The FMM has been derived for specific kernels that appear frequently in computational physics [37, 38, 48, 51, 102], including the Green's function for the Helmholtz equation [22, 81, 82] which is parameterized

by wavenumber $\kappa > 0$ and governs the phenomena of electromagnetic and acoustic scattering

$$K(x_i, x_j) = \begin{cases} \frac{i}{4} H_0^{(1)}(\kappa \|x_i - x_j\|), & x_i \neq x_j \in \mathbb{R}^2 \\ -\frac{1}{4} \frac{e^{i\kappa \|x_i - x_j\|}}{\|x_i - x_j\|}, & x_i \neq x_j \in \mathbb{R}^3, \end{cases} \tag{2.4}$$

where $H_0^{(1)}$ is the Hankel function of the first kind of order zero.

Though the FMM has been derived for a variety of kernels using analytic techniques, the derivation may be tedious and difficult for an arbitrary kernel function. Therefore, black-box algorithms have been developed, which require only the evaluation of a given kernel function. Such methods can be classified into two groups. The first consists of methods that approximate the kernel function (away from the origin) with polynomials, such as Legendre polynomials or Chebyshev polynomials [20, 32, 35, 47]. The other group consists of methods that compute the so-called equivalent densities or the so-called skeletons for every subdomain to efficiently represent the contained source points and their weights [64, 70, 101, 103]. Theoretically, this approach is justified by the potential theory for kernel functions that are associated with fundamental solutions of elliptic PDEs.

The contributions of the work are the following:

1. The algorithm is kernel-independent and can be extended to a variety of constant-coefficient kernels. Like other kernel-independent approaches, it replaces analytic expansions by using an equivalent set of source points ('skeleton points') that replicates the effect of the original source points in the far field. This representation is particularly effective for highly non-uniform point distributions (as results, for instance, when discretizing a boundary integral equation on a surface).

2. A novel description of a kernel-independent FMM that relies on simpler data structures, compared to traditional FMM approaches. The algorithm is multilevel, compatible with adaptive trees, and does not require the interaction list,

c.f. Remark 2.2. (The "W-list" and the "X-list" of classical FMMs [49] are not needed either.) Instead, the algorithm only involves calculations between near-neighbors at every level of the algorithm.

3. A parallel implementation of the proposed algorithm on modern architectures. The preprocessing stage is optimized for CPU computations and parallelized through OpenMP; the matrix-vector product is implemented using the GPU, leveraging highly-tuned batched BLAS primitives.

2.2 Methodology

In this section, the summation problem and general methodology used for the FMM are discussed. As a simple example, consider the points $\mathcal{X}$ in $\mathbb{R}^2$ shown in Figure 2.1. The points are partitioned in a uniform tree $\mathcal{T}$, and the geometry is described by some simple terminology. Boxes on the same level are called *colleagues*. Colleagues that share a corner or edge (e.g. are adjacent) are called *neighbors*. Colleagues which are not adjacent are called *well-separated*. For a box $\mathcal{B}$, the set of neighbor boxes is called $\mathcal{N}(\mathcal{B})$, and the *far-field* $\mathcal{F}(\mathcal{B})$ is the set of all well-separated boxes. The *parent* $\mathcal{P}(\mathcal{B})$ of a box $\mathcal{B}$ is the box on the next coarsest level which contains $\mathcal{B}$. Likewise, the *children* $\mathcal{C}(\mathcal{B})$ of a box $\mathcal{B}$ are the set of boxes whose parent is $\mathcal{B}$. A box $\mathcal{B}$ without children is called a *leaf*.

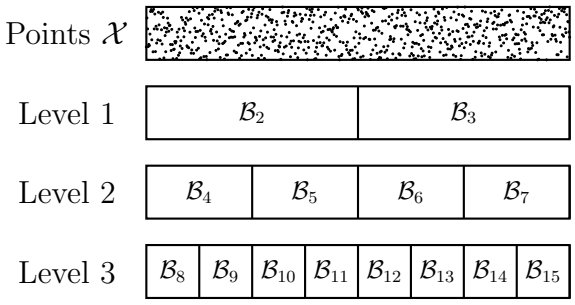


Figure 2.1: Points $\mathcal{X}$ partitioned into multi-level tree $\mathcal{T}$. For a box $\mathcal{B}$, the set of neighbors is called $\mathcal{N}(\mathcal{B})$, and the set of well-separated boxes is called the far field $\mathcal{F}(\mathcal{B})$. As an example $\mathcal{N}(\mathcal{B}_9) = \{\mathcal{B}_8, \mathcal{B}_{10}\}$ and $\mathcal{F}(\mathcal{B}_9) = \{\mathcal{B}_{10}, \ldots, \mathcal{B}_{15}\}$.

Consider the partitioning of points on level 3 into $n_3 = 8$ boxes of $m := N/n_3$ points. The kernel matrix $\mathbf{A}$ has the following property for index sets $I_{\mathcal{B}_i}, I_{\mathcal{B}_j}$ corresponding to well-separated boxes

$$\underset{m\times m}{\mathbf{A}_{\mathcal{B}_i,\mathcal{B}_j}} = \underset{m\times k}{\mathbf{L}_{\mathcal{B}_i}} \; \underset{k\times k}{\tilde{\mathbf{A}}_{\mathcal{B}_i,\mathcal{B}_j}} \; \underset{k\times m}{\mathbf{R}_{\mathcal{B}_j}} + O(\epsilon), \qquad \mathcal{B}_i, \mathcal{B}_j \text{ well-separated.} \tag{2.5}$$

That is, the interaction between a box $\mathcal{B}_i$ and its far field $\mathcal{F}(\mathcal{B}_i)$ is low rank to accuracy ϵ. In particular, the numerical rank $k_{\max}$ is $k \sim \mathcal{O}(\log(1/\epsilon))$, independent of the problem size N, for a smooth non-oscillatory kernel such as the the free-space Green's function for the Laplace equation (2.2) in 2D (see, e.g., [48]). For non-oscillatory kernels on 3D point distributions, the numerical rank $k_{\max}$ is also independent of N but has slower decay with ϵ, in particular, $k \sim \mathcal{O}(\log(1/\epsilon)^2)$. The low-rank property may deteriorate for oscillatory kernels. In particular, for a box of diameter D, the numerical rank is $\mathcal{O}(\kappa D)$; however, for fixed κ and increasing N, the rank is still independent of N, and the techniques described in this work apply.

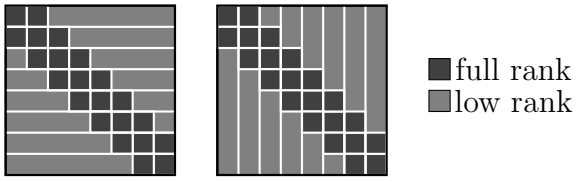


Figure 2.2: The structure of $\mathbf{A}$ corresponding to geometry in Figure 2.1 for the boxes on level 3. Interactions between neighbors are full rank. Interactions between a box and its far field are approximately rank $k_{\max}$.

The next subsections describe how the structure (2.5) can be used to construct a sparse factorization of $\mathbf{A}$. First, Section 2.2.1 describes a process known as *skeletonization*, where a set of source points in box $\mathcal{B}$ is replaced with an equivalent set of source points that replicates the effect in the far field of $\mathcal{B}$. Skeletonization can be applied to all boxes on a single level, leading to a sparse factorization of $\mathbf{A}$, which is discussed in Section 2.2.2. Finally, Section 2.2.3 discuss how this process can be applied recursively on a tree $\mathcal{T}$ to produce a multi-level algorithm that involves only near-neighbor calculations. The key differences between our approach and traditional FMM algorithms are highlighted as well.

2.2.1 Skeletonization of a single box

Consider a box $\mathcal{B}$ with near neighbors $\mathcal{N}$ and far-field $\mathcal{F}$. For an appropriate permutation matrix $\mathbf{P}$, we write $\mathbf{A}$ as

$$\mathbf{P}^\top \mathbf{A} \mathbf{P} = \begin{pmatrix} \mathbf{A}_{\mathcal{B}\mathcal{B}} & \mathbf{A}_{\mathcal{B}\mathcal{N}} & \mathbf{A}_{\mathcal{B}\mathcal{F}} \\ \mathbf{A}_{\mathcal{N}\mathcal{B}} & \mathbf{A}_{\mathcal{N}\mathcal{N}} & \mathbf{A}_{\mathcal{N}\mathcal{F}} \\ \mathbf{A}_{\mathcal{F}\mathcal{B}} & \mathbf{A}_{\mathcal{F}\mathcal{N}} & \mathbf{A}_{\mathcal{F}\mathcal{F}} \end{pmatrix}. \tag{2.6}$$

The observation is that the off-diagonal block $\mathbf{A}_{\mathcal{B}\mathcal{F}}$ (or $\mathbf{A}_{\mathcal{F}\mathcal{B}}$) for an arbitrary box $\mathcal{B}$ can be approximated efficiently by a low-rank approximation for a prescribed accuracy ε. Next, a specific type of low-rank approximation named the interpolative decomposition (ID) [21] is explained. This technique is applied to the submatrix $\mathbf{A}_{\mathcal{B}\mathcal{F}}$.

Definition 2.1. Let $\mathcal{I} = \{1, 2, \ldots, m\}$ and $\mathcal{J} = \{1, 2, \ldots, n\}$ be the row and column indices of a matrix $\mathbf{A}_{\mathcal{I}\mathcal{J}} \in \mathbb{C}^{m \times n}$. A (column) interpolative decomposition (ID) for a prescribed accuracy ε finds the so-called *skeleton* indices $\mathcal{S} \subset \mathcal{J}$, the *redundant* indices $\mathcal{R} = \mathcal{J} \backslash \mathcal{S}$, and an *interpolation matrix* $\mathbf{T} \in \mathbb{C}^{|\mathcal{S}| \times |\mathcal{R}|}$ such that

$$\|\mathbf{A}_{\mathcal{I}\mathcal{R}} - \mathbf{A}_{\mathcal{I}\mathcal{S}} \mathbf{T}\| \leq \varepsilon \, \|\mathbf{A}_{\mathcal{I}\mathcal{J}}\|.$$

While the strong rank-revealing QR factorization of Gu and Eisenstat [52] is the most robust method for computing an ID, this work employs the column-pivoting QR factorization as a greedy approach [21], which has better computational efficiency and behaves well in practice. The cost to compute an ID using the aforementioned deterministic methods is $\mathcal{O}(mn\,|\mathcal{S}|)$, which can be further reduced to $\mathcal{O}(mn\,\log(|\mathcal{S}|) + |\mathcal{S}|^2 n)$ using randomized algorithms that may incur some loss of accuracy [27].

As stated earlier, the aim is to compress the two off-diagonal blocks $\mathbf{A}_{\mathcal{B}\mathcal{F}}$ and $\mathbf{A}_{\mathcal{F}\mathcal{B}}$ using their IDs. Instead of compressing each block independently, it is more convenient to conduct column ID compression of the concatenation,

$$\begin{pmatrix} \mathbf{A}_{\mathcal{F}\mathcal{B}} \\ \mathbf{A}^*_{\mathcal{B}\mathcal{F}} \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{\mathcal{F}\mathcal{R}} & \mathbf{A}_{\mathcal{F}\mathcal{S}} \\ \mathbf{A}^*_{\mathcal{R}\mathcal{F}} & \mathbf{A}^*_{\mathcal{S}\mathcal{F}} \end{pmatrix} \approx \begin{pmatrix} \mathbf{A}_{\mathcal{F}\mathcal{S}} \\ \mathbf{A}^*_{\mathcal{S}\mathcal{F}} \end{pmatrix} \begin{pmatrix} \mathbf{T} & \mathbf{I} \end{pmatrix} \tag{2.7}$$

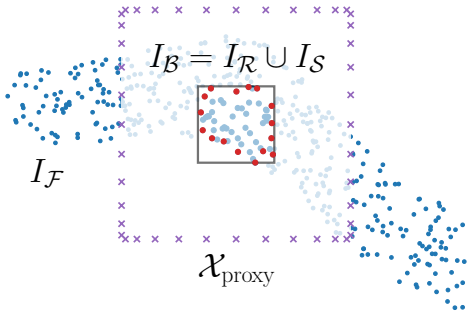


Figure 2.3: For a box $\mathcal{B}$, skeleton indices $I_{\mathcal{S}} \subseteq I_{\mathcal{B}}$ (shown in red) can be computed efficiently using a proxy surface $\mathcal{X}_{\text{proxy}}$, which are fictitious points replicating the effect of the far-field points $I_{\mathcal{F}}$. In our implementation of proxy surfaces, we use Chebyshev points on a box of length $3a$ centered at c, where a and c are the length and center of box $\mathcal{B}$, respectively.

which leads to a slightly larger set of skeleton indices but makes the implementation easier. Notice that *the computational cost would be $\mathcal{O}(N)$ if the full matrix in (2.7) is formed, which turns out to be unnecessary.* In the literature, there are a few techniques that require only $\mathcal{O}(1)$ operations such as the (analytical) multipole expansion (see, e.g., [22, 48, 50]), the Chebyshev interpolation (see, e.g., [35, 92]), and the proxy method (see, e.g., [57, 69, 100, 101]). This work uses on the proxy method, which introduces points $\mathcal{X}_{\text{proxy}}$ which replicate the effect of far-field points $I_{\mathcal{F}}$ [68, Sec. 17.1]. Instead of using the matrix in equation (2.7), smaller matrix with $\mathcal{O}(1)$ rows and columns is formed and compressed

$$\begin{pmatrix} \mathbf{A}_{\text{proxy},\mathcal{B}} \\ \mathbf{A}^{*}_{\mathcal{B},\text{proxy}} \end{pmatrix}, \qquad \text{where} \quad (\mathbf{A}_{\text{proxy},\mathcal{B}})_{i,j} = K\left((\mathcal{X}_{\text{proxy}})_i \, , (\mathcal{X}_{\mathcal{B}})_j \right). \tag{2.8}$$

With the appropriate choice of proxy surface, the matrix $\mathbf{A}_{\text{proxy},\mathcal{B}}$ spans the row space of $\mathbf{A}_{\mathcal{F},\mathcal{B}}$, and likewise, $\mathbf{A}_{\mathcal{B},\text{proxy}}$ spans the column space of $\mathbf{A}_{\mathcal{B},\mathcal{F}}$ for *any* point distribution in the far-field. Therefore, the indices $I_{\mathcal{B}} = I_{\mathcal{R}} \cup I_{\mathcal{S}}$ and interpolative matrix $\mathbf{T}$ computed by forming and factorizing the matrix (2.8) will satisfy (2.7). Figure 2.3 shows a proxy surface $\mathcal{X}_{\text{proxy}}$ for a box $\mathcal{B}$ as well as the chosen skeleton indices $I_{\mathcal{S}} \subseteq I_{\mathcal{B}}$.

2.2.2 Matrix Sparsification

In this subsection, the discussion centers on sparse algebraic operators that, when applied to the dense kernel matrix $\mathbf{A}$, lead to a sparser system. For a box $\mathcal{B}$, the ID compression of 2.8 gives skeleton indices $I_{\mathcal{S}}$, redundant indices $I_{\mathcal{R}} = I_{\mathcal{B}} \setminus I_{\mathcal{S}}$, and an interpolation matrix $\mathbf{T}$ such that (2.7) holds. For an appropriate permutation matrix $\mathbf{P}$, rewrite (2.6) as

$$\mathbf{P}^\top \mathbf{A} \mathbf{P} = \left(\begin{array}{cc|cc} \mathbf{A}_{\mathcal{R}\mathcal{R}} & \mathbf{A}_{\mathcal{R}\mathcal{S}} & \mathbf{A}_{\mathcal{R}\mathcal{N}} & \mathbf{A}_{\mathcal{R}\mathcal{F}} \\ \mathbf{A}_{\mathcal{S}\mathcal{R}} & \mathbf{A}_{\mathcal{S}\mathcal{S}} & \mathbf{A}_{\mathcal{S}\mathcal{N}} & \mathbf{A}_{\mathcal{S}\mathcal{F}} \\ \hline \mathbf{A}_{\mathcal{N}\mathcal{R}} & \mathbf{A}_{\mathcal{N}\mathcal{S}} & \mathbf{A}_{\mathcal{N}\mathcal{N}} & \mathbf{A}_{\mathcal{N}\mathcal{F}} \\ \mathbf{A}_{\mathcal{F}\mathcal{R}} & \mathbf{A}_{\mathcal{F}\mathcal{S}} & \mathbf{A}_{\mathcal{F}\mathcal{N}} & \mathbf{A}_{\mathcal{F}\mathcal{F}} \end{array}\right) \tag{2.9}$$

$$\overset{(2.7)}{\approx} \left(\begin{array}{cc|cc} \mathbf{A}_{\mathcal{R}\mathcal{R}} & \mathbf{A}_{\mathcal{R}\mathcal{S}} & \mathbf{A}_{\mathcal{R}\mathcal{N}} & \mathbf{T}^* \mathbf{A}_{\mathcal{S}\mathcal{F}} \\ \mathbf{A}_{\mathcal{S}\mathcal{R}} & \mathbf{A}_{\mathcal{S}\mathcal{S}} & \mathbf{A}_{\mathcal{S}\mathcal{N}} & \mathbf{A}_{\mathcal{S}\mathcal{F}} \\ \hline \mathbf{A}_{\mathcal{N}\mathcal{R}} & \mathbf{A}_{\mathcal{N}\mathcal{S}} & \mathbf{A}_{\mathcal{N}\mathcal{N}} & \mathbf{A}_{\mathcal{N}\mathcal{F}} \\ \mathbf{A}_{\mathcal{F}\mathcal{S}}\mathbf{T} & \mathbf{A}_{\mathcal{F}\mathcal{S}} & \mathbf{A}_{\mathcal{F}\mathcal{N}} & \mathbf{A}_{\mathcal{F}\mathcal{F}} \end{array}\right), \tag{2.10}$$

Define the *sparsification* operators

$$\mathbb{R}^{N\times N} \ni \mathbf{S}_{\mathcal{B}} \triangleq \left(\begin{array}{cc|cc} \mathbf{I} & & & \\ \mathbf{T} & \mathbf{I} & & \\ \hline & & \mathbf{I} & \\ & & & \mathbf{I} \end{array}\right) \tag{2.11}$$

where the partitioning of row and column indices is the same as that of $\mathbf{P}^\top \mathbf{A} \mathbf{P}$ and the diagonal blocks are identity matrices of appropriate sizes.

Remark 2.1. The inverse of a sparsification operator $\mathbf{S}_{\mathcal{B}}$ is simple to compute.

$$\mathbb{R}^{N\times N} \ni \mathbf{S}_{\mathcal{B}}^{-1} = \left(\begin{array}{cc|cc} \mathbf{I} & & & \\ -\mathbf{T} & \mathbf{I} & & \\ \hline & & \mathbf{I} & \\ & & & \mathbf{I} \end{array}\right) \tag{2.12}$$

Applying sparsification operators to the right and left of $\mathbf{A}$, the modified system is

$$\mathbf{S}_{\mathcal{B}}^{-*} \left(\mathbf{P}^\top \mathbf{A} \mathbf{P}\right) \mathbf{S}_{\mathcal{B}}^{-1} \approx \left(\begin{array}{cc|cc} \mathbf{X}_{\mathcal{R}\mathcal{R}} & \mathbf{X}_{\mathcal{R}\mathcal{S}} & \mathbf{X}_{\mathcal{R}\mathcal{N}} & \\ \mathbf{X}_{\mathcal{S}\mathcal{R}} & \mathbf{A}_{\mathcal{S}\mathcal{S}} & \mathbf{A}_{\mathcal{S}\mathcal{N}} & \mathbf{A}_{\mathcal{S}\mathcal{F}} \\ \hline \mathbf{X}_{\mathcal{N}\mathcal{R}} & \mathbf{A}_{\mathcal{N}\mathcal{S}} & \mathbf{A}_{\mathcal{N}\mathcal{N}} & \mathbf{A}_{\mathcal{N}\mathcal{F}} \\ & \mathbf{A}_{\mathcal{F}\mathcal{S}} & \mathbf{A}_{\mathcal{F}\mathcal{N}} & \mathbf{A}_{\mathcal{F}\mathcal{F}} \end{array}\right), \tag{2.13}$$

where the coupling (submatrices) between $\mathcal{R}$ and $\mathcal{F}$ disappears. Here, the notation $\mathbf{X}$ denotes a modified block. Notice that the process of skeletonizing box $\mathcal{B}$ only modifies rows and columns of indices $I_{\mathcal{R}} \subset I_{\mathcal{B}}$. This means all boxes (at the same level) can be skeletonized in parallel.

Consider two boxes $\mathcal{B}_i$ and $\mathcal{B}_j$ on the same level of tree $\mathcal{T}$. Then applying sparsification operators on the left and right of $\mathbf{A}$, the resulting matrix $\mathbf{X}$ is much sparser

$$\mathbf{X} := \mathbf{S}_{\mathcal{B}_j}^{-*}\ \mathbf{S}_{\mathcal{B}_i}^{-*}\ \left(\mathbf{P}^\top \mathbf{A} \mathbf{P}\right)\ \mathbf{S}_{\mathcal{B}_i}^{-1}\ \mathbf{S}_{\mathcal{B}_j}^{-1}. \tag{2.14}$$

The following is true about $\mathbf{X}$:

1. The interaction between $\mathcal{R}_{\mathcal{B}_i}$ and the far field of box $\mathcal{B}_i$, denoted as $\mathcal{F}_{\mathcal{B}_i}$, is approximately zero. The same is true of the interaction between $\mathcal{R}_{\mathcal{B}_j}$ and the far field of box $\mathcal{B}_j$.

2. If the two boxes $\mathcal{B}_i$ and $\mathcal{B}_j$ are not neighbors, applying sparsification operators does not modify the original interaction, i.e. $\mathbf{X}_{\mathcal{B}_i,\mathcal{B}_j} = \mathbf{A}_{\mathcal{B}_i,\mathcal{B}_j}$.

3. If the two boxes $\mathcal{B}_i$ and $\mathcal{B}_j$ are neighbors or if $\mathcal{B}_i = \mathcal{B}_j$, $\mathbf{X}_{\mathcal{B}_i,\mathcal{B}_j}$ will have modified interactions, defined as follows:

$$\begin{aligned} \mathbf{X}_{\mathcal{R}_i\,\mathcal{S}_j} &= \mathbf{A}_{\mathcal{R}_i\,\mathcal{S}_j} - \mathbf{T}_i^* \mathbf{A}_{\mathcal{S}_i\,\mathcal{S}_j} & \mathbf{X}_{\mathcal{S}_i\,\mathcal{R}_j} &= \mathbf{A}_{\mathcal{S}_i\,\mathcal{R}_j} - \mathbf{A}_{\mathcal{S}_i\,\mathcal{S}_j} \mathbf{T}_j \\ \mathbf{X}_{\mathcal{R}_i\,\mathcal{R}_j} &= \mathbf{A}_{\mathcal{R}_i\,\mathcal{R}_j} - \mathbf{T}_i^* \mathbf{A}_{\mathcal{S}_i\,\mathcal{R}_j} - \mathbf{A}_{\mathcal{R}_i\,\mathcal{S}_j} \mathbf{T}_j + \mathbf{T}_i^* \mathbf{A}_{\mathcal{S}_i\,\mathcal{S}_j} \mathbf{T}_j \end{aligned} \tag{2.15}$$

4. Unconditionally, the interactions between skeleton nodes of boxes $\mathcal{S}_i, \mathcal{S}_j$ are not modified, i.e. $\mathbf{X}_{\mathcal{S}_i,\mathcal{S}_j} = \mathbf{A}_{\mathcal{S}_i,\mathcal{S}_j}$.

As mentioned, the sparsification operators can be computed and applied for a single level in parallel. It is straightforward to verify that the composition of operators $S_{\mathcal{B}}$

for boxes $\mathcal{B}$ on leaf level ℓ of tree $\mathcal{T}$ has the following form

$$\mathbb{R}^{N\times N} \ni \mathbf{S}_{(\ell)} \triangleq \mathbf{S}_{\mathcal{B}_1}\,\mathbf{S}_{\mathcal{B}_2}\,\ldots\mathbf{S}_{\mathcal{B}_{n_\ell}} = \left(\begin{array}{c|c|c|c} \begin{array}{cc}\mathbf{I} & \\ \mathbf{T}_1 & \mathbf{I}\end{array} & & & \\ \hline & \begin{array}{cc}\mathbf{I} & \\ \mathbf{T}_2 & \mathbf{I}\end{array} & & \\ \hline & & \ddots & \\ \hline & & & \begin{array}{cc}\mathbf{I} & \\ \mathbf{T}_{n_\ell} & \mathbf{I}\end{array} \end{array}\right), \tag{2.16}$$

where $\mathbf{T}_i$ is the interpolation matrix, $\mathbf{S}_i$ defined in (2.11) is the sparsification operator associated with every box $\mathcal{B}$, and n_ℓ is the number of boxes on level ℓ. Define the index set $I_{(\ell)}$ as the set of of all skeleton indices from boxes B on level ℓ

$$I_{(\ell)} \triangleq \bigcup_{i=1}^{n_\ell} \mathcal{S}_{\mathcal{B}_i}, \tag{2.17}$$

and $\mathbf{I}_{(\ell)}$ as the sparse matrix which selects the index set $I_{(\ell)}$

$$\mathbb{R}^{n_l k\times N} \ni \mathbf{I}_{(\ell)} \triangleq \left(\begin{array}{c|c|c|c} \mathbf{I} & & & \\ \hline & \mathbf{I} & & \\ \hline & & \ldots & \\ \hline & & & \mathbf{I} \end{array}\right). \tag{2.18}$$

Following the claims made earlier in this section, applying sparsification operator $\mathbf{S}_{(\ell)}$ to $\mathbf{A}$ does not modify the submatrix $\mathbf{A}_{(\ell),(\ell)}$. Then applying $\mathbf{S}_{(\ell)}$ to the left and right of $\mathbf{A}$, the resulting sparse system is

$$\underset{N\times N}{\mathbf{S}^{-*}_{(\ell)}} \Big(\underset{N\times N}{\mathbf{P}^\top\,\mathbf{A}\,\mathbf{P}}\Big) \underset{N\times N}{\mathbf{S}^{-1}_{(\ell)}} \quad \triangleq \quad \underset{N\times N}{\mathbf{X}_{(\ell)}} \tag{2.19}$$

$$\triangleq \underset{N\times N}{[\mathbf{X}_{(\ell)}]_{\text{mod}}} + \underset{N\times n_\ell k}{\mathbf{I}^*_{(\ell)}} \quad \underset{n_\ell k\times n_\ell k}{\mathbf{A}_{(\ell),(\ell)}} \quad \underset{n_\ell k\times N}{\mathbf{I}_{(\ell)}}, \tag{2.20}$$

where $[\mathbf{X}_{(\ell)}]_{\text{mod}}$ has *algebraically modified* kernel interactions. The interactions between near neighbors are modified, as described in equation (2.15). Crucially, interactions between $\mathcal{S}_{(l)}$ are not modified and appear as the second term in the sum (2.20). Applying the inverse of $\mathbf{S}^{(\ell)}$ to the left and right of equation (2.20), the resulting sparse factorization of $\mathbf{A}$ has the following form

$$\mathbf{P}^\top\,\mathbf{A}\,\mathbf{P} \;=\; \mathbf{S}^*_{(\ell)}\left([\mathbf{X}_{(\ell)}]_{\text{mod}} \;+\; \mathbf{I}^*_{(\ell)}\,\mathbf{A}_{(\ell),(\ell)}\,\mathbf{I}_{(\ell)}\right)\mathbf{S}_{(\ell)}. \tag{2.21}$$

2.2.3 Multi-Level Algorithm for a Simple Geometry

Using the operators defined in previous sections, this section generalizes to a multi-level algorithm to compute the matrix-vector product (2.3). Consider the simple geometry in Figure 2.1 on a uniform tree $\mathcal{T}$ with $n_3 = 8$ boxes on Level 3. As discussed previously, the low-rank property in equation (2.5) holds for well-separated boxes on the same level. Using the skeletonization process described in Section 2.2.1,

$$\mathbf{A}_{\mathcal{B}_i,\mathcal{B}_j} = \begin{pmatrix} \mathbf{A}_{\mathcal{R}_i,\mathcal{R}_j} & \mathbf{A}_{\mathcal{R}_i,\mathcal{S}_j} \\ \mathbf{A}_{\mathcal{S}_i,\mathcal{R}_j} & \mathbf{A}_{\mathcal{S}_i,\mathcal{S}_j} \end{pmatrix} \approx \begin{pmatrix} \mathbf{T}_i^* \\ \mathbf{I} \end{pmatrix} \mathbf{A}_{\mathcal{S}_i,\mathcal{S}_j} \begin{pmatrix} \mathbf{T}_j & \mathbf{I} \end{pmatrix} \tag{2.22}$$

where $\mathcal{B}_i, \mathcal{B}_j$ are well-separated colleagues. Equivalently,

$$\mathbf{A}_{\mathcal{B}_i,\mathcal{B}_j} \approx \begin{pmatrix} \mathbf{I} & \mathbf{T}_i^* \\ & \mathbf{I} \end{pmatrix} \begin{pmatrix} & \\ & \mathbf{A}_{\mathcal{S}_i,\mathcal{S}_j} \end{pmatrix} \begin{pmatrix} \mathbf{I} & \\ \mathbf{T}_j & \mathbf{I} \end{pmatrix} \tag{2.23}$$

For colleagues $\mathcal{B}_i, \mathcal{B}_j$ which are neighbors, one can write a similar form of decomposition with *modified* interactions

$$\mathbf{A}_{\mathcal{B}_i,\mathcal{B}_j} = \begin{pmatrix} \mathbf{I} & \mathbf{T}_i^* \\ & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{X}_{\mathcal{R}_i,\mathcal{R}_j} & \mathbf{X}_{\mathcal{R}_i,\mathcal{S}_j} \\ \mathbf{X}_{\mathcal{S}_i,\mathcal{R}_j} & \mathbf{A}_{\mathcal{S}_i,\mathcal{S}_j} \end{pmatrix} \begin{pmatrix} \mathbf{I} & \\ \mathbf{T}_j & \mathbf{I} \end{pmatrix} \tag{2.24}$$

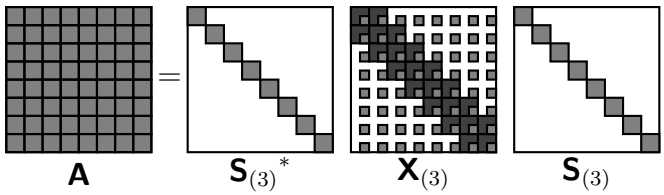
where the modified interactions have the form described in equation (2.15). Note that (2.24) is exact equality. Using the sparsification operator $\mathbf{S}_{(3)}$ defined in (2.16), $\mathbf{A}$ has the sparse factorization

$$\mathbf{P}^\top \mathbf{A} \mathbf{P} = \mathbf{S}_{(3)}^* \left([\mathbf{X}_{(3)}]_{\text{mod}} + \mathbf{I}_{(3)}^* \mathbf{A}_{(3),(3)} \mathbf{I}_{(3)} \right) \mathbf{S}_{(3)} \tag{2.25}$$

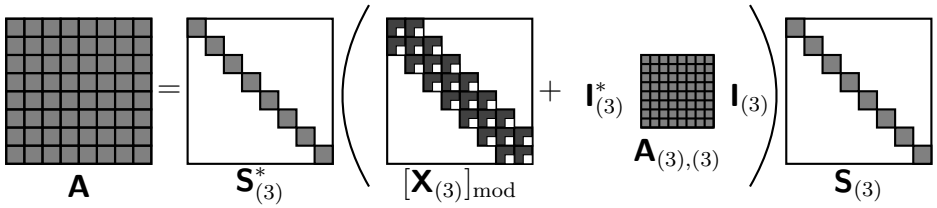
Figure 2.4 depicts the sparsity pattern of each matrix. Importantly, the matrix $A_{(3),(3)}$ is a dense kernel matrix of interactions between a smaller set of points $\mathcal{X}_{(3)}$. These points can be partitioned according the tree decomposition on level 2, and a recursive approach can be applied to calculate a sparse factorization of $\mathbf{A}_{(3),(3)}$.

Remark 2.2. This brief remark highlights the difference between the approach in this work and previous algorithms [58, 70]. Consider again the simple geometry of Figure 2.1. For the boxes on level 3, the matrix $\mathbf{A}$ written as a sum of near and far interactions on level 3 as

$$\mathbf{A} = \mathbf{A}_{\text{near}} + \mathbf{A}_{\text{far}}, \tag{2.26}$$



(a) A sparse factorization of $\mathbf{A}$ using sparsification operators $\mathbf{S}_{(3)}$.



(b) The matrix $\mathbf{X}_{(3)}$ is a sum of modified near neighbor interactions $[\mathbf{X}_{(3)}]_{\text{mod}}$ and unmodified interactions between the skeleton indices $I_{(3)}$.

Figure 2.4: In our approach, we apply operators to all of the matrix $\mathbf{A}$, as shown in Figure 2.4a. The operators $\mathbf{S}_{(3)}$ *algebraically modify* the kernel interactions between near neighbors, shown in dark gray, but do not modify kernel interactions between skeleton indices $I_{(3)}$. Figure 2.4*b* shows that the matrix $\mathbf{X}_{(3)}$ can be separated into a sparse matrix of modified interactions and a dense matrix of unmodified kernel interactions $\mathbf{A}_{(3),(3)}$. A multi-level algorithm is derived by coarsening the matrix $\mathbf{A}_{(3),(3)}$ into 4 blocks, as shown in level 2 of tree $\mathcal{T}$, of Figure 2.1. This leads to a multi-level algorithm involving only *modified* near neighbor interactions at every level.

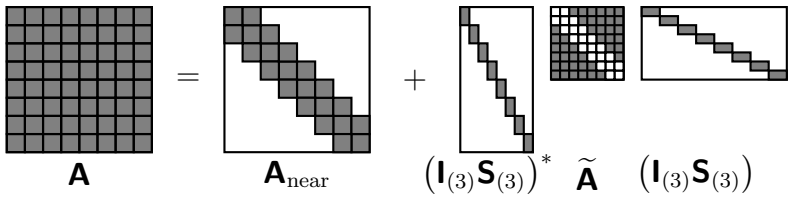
where $\mathbf{A}_{\text{near}} = \text{tridiag}(\mathbf{A})$. Using (2.22), $\mathbf{A}_{\text{far}}$ can be factorized using a sparsification operator of the following form

$$\mathbb{R}^{n_3 k \times N} \ni \begin{pmatrix} \mathbf{T}_1 & \mathbf{I} & & & & & \\ & & \mathbf{T}_2 & \mathbf{I} & & & \\ & & & & \cdots & & \\ & & & & & \mathbf{T}_{n_3} & \mathbf{I} \end{pmatrix} = \left(\mathbf{I}_{(3)}\mathbf{S}_{(3)}\right) \tag{2.27}$$

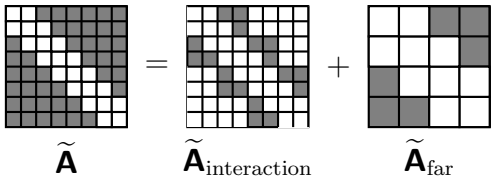
The resulting factorization of $\mathbf{A}_{\text{far}}$ has the following form

$$\underset{N\times N}{\mathbf{A}_{\text{far}}} = \underset{N\times n_3 k}{\left(\mathbf{I}_{(3)}\mathbf{S}_{(3)}\right)^*} \underset{n_3 k\times n_3 k}{\widetilde{\mathbf{A}}} \underset{n_3 k\times N}{\left(\mathbf{I}_{(3)}\mathbf{S}_{(3)}\right)} \tag{2.28}$$

where $\widetilde{\mathbf{A}} = \mathbf{A}_{(3),(3)} - \text{tridiag}\left(\mathbf{A}_{(3),(3)}\right)$ consists of sub-blocks of original kernel interactions of $\mathbf{A}$.



(a) Sparsity pattern of $\mathbf{A} = \mathbf{A}_{\text{near}} + \mathbf{A}_{\text{far}}$, where $\mathbf{A}_{\text{far}}$ is factorized using sparsification operator $\left(\mathbf{I}_{(3)}\mathbf{S}_{(3)}\right)$. Note that $\widetilde{\mathbf{A}}$ consists of sub-blocks of original kernel interactions of $\mathbf{A}$.



(b) Sparsity pattern of $\widetilde{\mathbf{A}} = \widetilde{\mathbf{A}}_{\text{interaction}} + \widetilde{\mathbf{A}}_{\text{far}}$.

Figure 2.5: Previous approaches for the FMM separate the matrix $\mathbf{A}$ into near and far interactions and factorize only the far-field interactions, as shown in Figure 2.5a. The approach can be applied recursively to $\widetilde{\mathbf{A}}$, leading to the interaction list in $\widetilde{\mathbf{A}}_{\text{interaction}}$, shown in Figure 2.5b. For the simple geometry of Figure 2.1, the interaction list for a box $\mathcal{B}$ is of size at most 3; generally, interaction list is much larger for uniform point distributions in a square (at most size 27) or in a cube (at most size 189).

See Figure 2.5a for the sparsity pattern of $\widetilde{\mathbf{A}}$. In order to use a similar approach recursively for $\widetilde{\mathbf{A}}$, the matrix is coarsened into 4×4 blocks according to the tree decomposition on level 2. However, the structure of $\widetilde{\mathbf{A}}$ is significantly different from the fully-dense matrix $\mathbf{A}$. Instead, $\widetilde{\mathbf{A}}$ can be expressed as a sum

$$\widetilde{\mathbf{A}} = \widetilde{\mathbf{A}}_{\text{interaction}} + \widetilde{\mathbf{A}}_{\text{far}}, \tag{2.29}$$

where the sparsity pattern is shown in Figure 2.5b.

The matrix $\widetilde{\mathbf{A}}_{\text{interaction}}$ has the so-called interaction list, which involves non-zero blocks between the skeleton indices $\mathbf{A}_{\mathcal{S}_i,\mathcal{S}_j}$ of level 3 colleagues $\mathcal{B}_i, \mathcal{B}_j$ where the following conditions are met (1) the parents of $\mathcal{B}_i$ and $\mathcal{B}_j$ are neighbors and (2) boxes $\mathcal{B}_i$ and $\mathcal{B}_j$ are not neighbors. In 2D, the interaction list for a box $\mathcal{B}$ has size 27 for a fully populated tree, and in 3D, it has size 189. The interaction list for $\mathcal{B}$ contains only well-separated boxes and may be challenging to manipulate. For traditional

FMMs, additional data structures (e.g. W-list and X-list) are needed for adaptive trees. These lists, like the interaction list, require additional bookkeeping and make the FMM difficult to implement.

The decomposition presented in this work instead operates on much simpler data structures, requiring only the list of neighbor boxes. Additionally, the extension to adaptive trees is simple and does not involve much additional machinery beyond what has already been described.

2.3 Algorithm

This section describes the algorithm for adaptive quad-trees and oct-trees, which extend the proposed methodology to non-uniform point distributions. While Section 2.2 describes a sparse factorization of $\mathbf{A}$, this section instead uses matrix-free notation which is more amenable to parallel implementation. Additionally, the implementation of the pre-computation stage (on the CPU with OpenMP [23]) and the FMM apply (on the GPU with batched linear algebra [1, 28]) are discussed.

2.3.1 Adaptive tree data structure

The FMM relies on a hierarchical decomposition of the points $\mathcal{X} = \{x_i\}_{i=1}^N$ into an adaptive quad-tree or oct-tree. Much of the terminology is the same as discussed earlier for the simple geometry in Figure 2.1 which is defined on a uniform tree. An adaptive tree can be formed using a recursive approach. First, the root box $\mathcal{B}_0$ which contains all the points is defined. A parameter b is also defined, for the maximum number of points a box may have. Then $\mathcal{B}_0$ is split into 4 (or 8 for 3D) subdomains. Any box which is empty is pruned from the tree, and only boxes containing more than b points are subdivided recursively. The top-down construction produces a tree which is unbalanced. There may be adjacent leaf boxes that vary tremendously in their size, leading to a possibly unbounded number of near neighbors for a large box adjacent to very refined box. To limit the number of interactions, adaptive trees are

constrained to satisfy a 2:1 balance constraint, that is, adjacent leaf boxes must be within one level of each other. Given an unbalanced tree, additional leaves may be added in a sequential procedure that produces a balanced tree; as described in [64], this requires $\mathcal{O}(n_{\text{boxes}} \log n_{\text{boxes}})$, where n_{boxes} are the number of total boxes in the tree.

The *near-field* of box $\mathcal{B}$ of length a and center c contains points within an area defined by a box of length $3a$ centered at c. For uniform trees, the near-field $I_{\mathcal{N}(\mathcal{B})}$ only has adjacent colleagues of $\mathcal{B}$, that is boxes on the same level as $\mathcal{B}$ which share a point or an edge. For adaptive trees, the near-field $I_{\mathcal{N}(\mathcal{B})}$ may contain a *coarse neighbor* on a level above. For box $\mathcal{B}$ to have a coarse neighbor $\mathcal{B}'$, the parent $\mathcal{P}(\mathcal{B})$ must be adjacent colleagues with $\mathcal{B}'$, and $\mathcal{B}'$ must be a leaf. Figure 2.6 shows a box $\mathcal{B}$ whose near-field $I_{\mathcal{N}}$ includes a coarse neighbor. For $\mathcal{T}$ a perfect quad-tree, it is obvious that no box has more than 9 neighbor boxes (including itself). For $\mathcal{T}$ an adaptive quad-tree, the number of boxes in the near-field is bounded as well by 9. For 3D point distributions, $\mathcal{T}$ is a uniform or adaptive oct-tree, and the number of neighbor boxes is bounded by 27.

2.3.2 Build stage

As described in Section 2.2.1, each leaf box $\mathcal{B}$ can be skeletonized into redundant and skeleton indices $I_{\mathcal{B}} = I_{\mathcal{R}} \cup I_{\mathcal{S}}$ so that equation (2.7) holds. The skeleton nodes $I_{\mathcal{S}}$ and interpolation matrix $\mathbf{T}_{\mathcal{B}}$ can be efficiently computed using proxy surfaces by performing the column ID on the matrix in equation (2.8). After the skeleton nodes are computed for leaf boxes, skeleton indices are computed for the remaining tree boxes (i.e. boxes with children) by traversing the tree upwards. For tree boxes $\mathcal{B}$, the nodes $I_{\mathcal{B}}$ are set to be the union of skeleton indices of $\mathcal{C}(\mathcal{B})$.

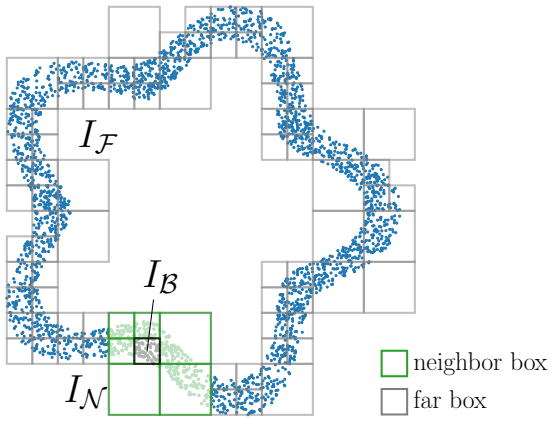


Figure 2.6: The figure shows the leaf boxes of an adaptive quadtree for a non-uniform distribution of points. The tree is adaptive, with reasonable restrictions on the adaptivity (e.g. adjacent leaf boxes are within one level of each other). For box $\mathcal{B}$ with corresponding index set $I_{\mathcal{B}}$, we also show neighbor boxes in green and far boxes in gray. The corresponding index sets for points in the near field and far-field are $I_{\mathcal{N}}$ and $I_{\mathcal{F}}$, respectively. For adaptive trees, a neighbor box of $\mathcal{B}$ may be on the level above; we call this a *coarse neighbor*.

Algorithm 1 `Recursive skeletonization`

Input: Kernel function K and tree decomposition $\mathcal{T}$ of points $\mathcal{X}$.
Output: Indices $I_{\mathcal{B}} = I_{\mathcal{S}} \cup I_{\mathcal{R}}$ and interpolation matrix $\mathbf{T}_{\mathcal{B}}$ for each box $\mathcal{B}$.

1: **for** $\ell = L, L-1, \ldots, 1$ **do**
2: **for all** boxes $\mathcal{B}$ at level ℓ **do** // in parallel
3: **if** box $\mathcal{B}$ is a leaf **then**
4: Set $I_{\mathcal{B}}$ according to tree decomposition $\mathcal{T}$.
5: **else**
6: Accumulate $I_{\mathcal{B}}$ from children as

$$I_{\mathcal{B}} := \bigcup_{\mathcal{B}' \in \mathcal{C}(\mathcal{B})} I_{\mathcal{S}_{\mathcal{B}'}}.$$

7: Compute $I_{\mathcal{B}} = I_{\mathcal{R}} \cup I_{\mathcal{S}}$ and $\mathbf{T}_{\mathcal{B}}$ using column ID of matrix (2.8).

2.3.3 FMM apply

Sections 2.2.2 and 2.2.3 describe a sparse factorization for the finest level of a uniform tree and also described how the algorithm may be generalized for a multi-level approach. This section defines a matrix-free algorithm and discuss modifications needed for adaptive trees. To describe the algorithm succinctly, some notation from potential theory is introduced. Consider that charges $\mathbf{q}_{\mathcal{B}}$ are given at sources for box $\mathcal{B}$ and that the aim is to compute potentials $\mathbf{u}_{\mathcal{B}}$ at targets for box $\mathcal{B}$, through equation (2.3).

Terminology for the outgoing representation $\hat{\mathbf{q}}_{\mathcal{B}}$ and the incoming representation $\hat{\mathbf{u}}_{\mathcal{B}}$ are also introduced. The skeleton nodes of an outgoing representation $\hat{\mathbf{q}}_{\mathcal{B}}$ replicate, to accuracy ε, the effect of $\mathbf{q}_{\mathcal{B}}$ at targets in the far field. Similarly, the skeleton nodes of an incoming representation of potentials $\hat{\mathbf{u}}_{\mathcal{B}}$ are the [approximate] potential at skeleton points $\mathcal{X}_{\mathcal{S}_{\mathcal{B}}}$ due to all sources $\mathbf{q}_{\mathcal{F}}$ in the far field. Translation operators are used to convert one representation to another. For example, the translation operator $\mathbf{X}^{(\mathrm{ifo})}_{\mathcal{B},\mathcal{B}'}$ converts an outgoing representation $\hat{\mathbf{q}}_{\mathcal{B}'}$ in box $\mathcal{B}'$ to an incoming representation $\hat{\mathbf{u}}_{\mathcal{B}}$ in box $\mathcal{B}$. Table 2.1 describes translation operators $\mathbf{X}^{(\mathrm{ifo})}_{\mathcal{B},\mathcal{B}'}$, $\mathbf{X}^{(\mathrm{tfo})}_{\mathcal{B},\mathcal{L}}$, and $\mathbf{X}^{(\mathrm{ofs})}_{\mathcal{L},\mathcal{B}}$ and summarizes the terminology used to describe the algorithm.

$\mathbf{q}_{\mathcal{B}}$	charges at sources $\mathcal{X}_{\mathcal{B}}$ (given)
$\mathbf{u}_{\mathcal{B}}$	potentials at targets $\mathcal{X}_{B}$
$\hat{\mathbf{q}}_{\mathcal{B}}$	outgoing representation for box $\mathcal{B}$
$\hat{\mathbf{u}}_{\mathcal{B}}$	incoming representation for box $\mathcal{B}$
$\mathbf{X}^{(\mathrm{ifo})}_{\mathcal{B},\mathcal{B}'}$	translation operator $\hat{\mathbf{u}}_{\mathcal{B}} \leftarrow \hat{\mathbf{q}}_{\mathcal{B}'}$
$\mathbf{X}^{(\mathrm{ifs})}_{\mathcal{B},\mathcal{L}}$	translation operator $\hat{\mathbf{u}}_{\mathcal{B}} \leftarrow \mathbf{q}_{\mathcal{L}}$
$\mathbf{X}^{(\mathrm{tfo})}_{\mathcal{L},\mathcal{B}}$	translation operator $\mathbf{u}_{\mathcal{L}} \leftarrow \hat{\mathbf{q}}_{\mathcal{B}}$

Table 2.1: Notation used to describe the FMM apply. $\mathcal{B}$ is a box in the tree (may be a leaf or not), and $\mathcal{L}$ is specifically a leaf box.

The translation operator $\mathbf{X}^{(\mathrm{ifo})}_{\mathcal{B},\mathcal{B}'}$ has the form

$$\mathbf{X}^{(\mathrm{ifo})}_{\mathcal{B},\mathcal{B}'} \triangleq \begin{pmatrix} \mathbf{X}_{\mathcal{R}_{\mathcal{B}},\mathcal{R}_{\mathcal{B}'}} & \mathbf{X}_{\mathcal{R}_{\mathcal{B}},\mathcal{S}_{\mathcal{B}'}} \\ \mathbf{X}_{\mathcal{S}_{\mathcal{B}},\mathcal{R}_{\mathcal{B}'}} & \mathbf{0} \end{pmatrix}, \tag{2.30}$$

where

$$\begin{aligned}
\mathbf{X}_{\mathcal{R}_\mathcal{B}\,\mathcal{R}_{\mathcal{B}'}} &= \mathbf{A}_{\mathcal{R}_\mathcal{B}\,\mathcal{R}_{\mathcal{B}'}} - \mathbf{T}^*_\mathcal{B}\mathbf{A}_{\mathcal{S}_\mathcal{B}\,\mathcal{R}_{\mathcal{B}'}} - \mathbf{A}_{\mathcal{R}_\mathcal{B}\,\mathcal{S}_{\mathcal{B}'}}\mathbf{T}_{\mathcal{B}'} + \mathbf{T}^*_\mathcal{B}\mathbf{A}_{\mathcal{S}_\mathcal{B}\,\mathcal{S}_{\mathcal{B}'}}\mathbf{T}_{\mathcal{B}'} \\
\mathbf{X}_{\mathcal{R}_\mathcal{B}\,\mathcal{S}_{\mathcal{B}'}} &= \mathbf{A}_{\mathcal{R}_\mathcal{B}\,\mathcal{S}_{\mathcal{B}'}} - \mathbf{T}^*_\mathcal{B}\mathbf{A}_{\mathcal{S}_\mathcal{B}\,\mathcal{S}_{\mathcal{B}'}} \qquad\qquad \mathbf{X}_{\mathcal{S}_B\,\mathcal{R}_{\mathcal{B}'}} = \mathbf{A}_{\mathcal{S}_\mathcal{B}\,\mathcal{R}_{\mathcal{B}'}} - \mathbf{A}_{\mathcal{S}_\mathcal{B}\,\mathcal{S}_{\mathcal{B}'}}\mathbf{T}_{\mathcal{B}'}
\end{aligned} \tag{2.31}$$

The translation operator $\mathbf{X}^{(\mathrm{ifs})}_{\mathcal{B},\mathcal{L}}$ has the form

$$\mathbf{X}^{(\mathrm{ifs})}_{\mathcal{B},\mathcal{L}} \triangleq \begin{pmatrix} \mathbf{X}_{\mathcal{R}_\mathcal{B},\mathcal{L}} \\ \mathbf{0} \end{pmatrix}, \quad \text{where } \mathbf{X}_{\mathcal{R}_\mathcal{B},\mathcal{L}} = \mathbf{A}_{\mathcal{R}_\mathcal{B},\mathcal{L}} - \mathbf{T}^*_\mathcal{B}\mathbf{A}_{\mathcal{S}_B,\mathcal{L}} \tag{2.32}$$

Finally, $\mathbf{X}^{(\mathrm{tfo})}_{\mathcal{L},\mathcal{B}}$ has the form

$$\mathbf{X}^{(\mathrm{tfo})}_{\mathcal{L},\mathcal{B}} \triangleq \begin{pmatrix} \mathbf{X}_{\mathcal{L},\mathcal{R}_\mathcal{B}} & \mathbf{0} \end{pmatrix}, \quad \text{where } \mathbf{X}_{\mathcal{L},\mathcal{R}_\mathcal{B}} = \mathbf{A}_{\mathcal{L},\mathcal{R}_\mathcal{B}} - \mathbf{A}_{\mathcal{L},\mathcal{S}_\mathcal{B}}\mathbf{T}_\mathcal{B}. \tag{2.33}$$

Briefly, the algorithm for the simplified FMM is presented. The procedure is summarized in Algorithm 2. Note that the operators $\mathbf{T}_\mathcal{B}$ are computed using the precomputation phase in Algorithm 1 and that translation operators are computed during runtime. First, outgoing representations for the leaf boxes are computed

$$\begin{pmatrix} \hat{\mathbf{q}}_\mathcal{R} \\ \hat{\mathbf{q}}_\mathcal{S} \end{pmatrix} := \begin{pmatrix} \mathbf{I} & \\ \mathbf{T}_\mathcal{L} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{q}_\mathcal{R} \\ \mathbf{q}_\mathcal{S} \end{pmatrix}, \qquad \text{for leaf boxes } \mathcal{L}. \tag{2.34}$$

In an upwards pass through the tree, outgoing representations $\hat{\mathbf{q}}_\mathcal{B}$ are computed for all boxes of the tree. For a box with children, this procedure requires accumulating values from the skeleton indices $\hat{\mathbf{q}}_{\mathcal{S}_\mathcal{C}}$ for $\mathcal{C} \in \mathcal{C}(\mathcal{B})$ and applying a operator of the same form as in (2.34).

Then, the incoming expansion $\hat{\mathbf{u}}_\mathcal{B}$ for every box $\mathcal{B}$ is computed by accumulating values from the neighbors of $\mathcal{B}$ using translation operators. In particular,

$$\hat{\mathbf{u}}_\mathcal{B} = \sum_{\substack{\text{colleague} \\ \text{neighbor } \mathcal{B}'}} \mathbf{X}^{(\mathrm{ifo})}_{\mathcal{B},\mathcal{B}'}\hat{\mathbf{q}}_{\mathcal{B}'} + \sum_{\substack{\text{coarse} \\ \text{neighbor } \mathcal{L}}} \mathbf{X}^{(\mathrm{ifs})}_{\mathcal{B},\mathcal{L}}\mathbf{q}_\mathcal{L}, \qquad \text{for all boxes } \mathcal{B}, \tag{2.35}$$

where $\mathcal{B}$ is included as a colleague neighbor to itself. Recall that a coarse neighbor must necessarily be a leaf box, so it is called $\mathcal{L}$. For the boxes on level 2 of the tree,

additional values are accumulated; the skeleton indices $\mathcal{S}_{(2)}$ interact densely through kernel interactions:

$$\hat{\mathbf{u}}_{\mathcal{S}_\mathcal{B}} \mathrel{+}= \sum_{\substack{\text{box } \mathcal{B}' \\ \text{on level 2}}} \mathbf{A}_{\mathcal{S}_\mathcal{B},\mathcal{S}'_\mathcal{B}} \hat{\mathbf{q}}_{\mathcal{S}'_\mathcal{B}}, \qquad \text{for boxes } \mathcal{B} \text{ on level 2.} \tag{2.36}$$

In a downward pass through the tree, incoming representations $\hat{\mathbf{u}}_\mathcal{B}$ are computed for every box $\mathcal{B}$. For box with children, this procedure requires doing linear algebraic operations with $\mathbf{T}^*_\mathcal{B}$ and copying values $\hat{\mathbf{u}}_\mathcal{B}$ appropriately to children $\mathcal{C} \in \mathcal{C}(\mathcal{B})$. For a leaf boxes $\mathcal{L}$, $\mathbf{u}_\mathcal{L}$ is computed using

$$\mathbf{u}_\mathcal{L} = \begin{pmatrix} \mathbf{u}_\mathcal{R} \\ \mathbf{u}_\mathcal{S} \end{pmatrix} := \begin{pmatrix} \mathbf{I} & \mathbf{T}^*_\mathcal{L} \\ & \mathbf{I} \end{pmatrix} \begin{pmatrix} \hat{\mathbf{u}}_\mathcal{R} \\ \hat{\mathbf{u}}_\mathcal{S} \end{pmatrix}, \qquad \text{for leaf boxes } \mathcal{L}. \tag{2.37}$$

For boxes $\mathcal{B}$ with coarse neighbor $\mathcal{L}$, contributions are added to $\mathbf{u}_\mathcal{L}$ as

$$\mathbf{u}_\mathcal{L} \mathrel{+}= \sum_{\substack{\text{boxes } \mathcal{B} \text{ with} \\ \text{coarse neighbor } \mathcal{L}}} \mathbf{X}^{(\text{tfo})}_{\mathcal{L},\mathcal{B}} \hat{\mathbf{u}}_\mathcal{B}, \qquad \text{for leaf boxes } \mathcal{L}. \tag{2.38}$$

2.3.4 Parallel Implementation

In this section, the efficient parallel implementations of Algorithm 1 for the precomputation and Algorithm 2 for evaluating (2.3) are discussed. In the precomputation phase, the calculation of the skeleton indices and interpolation matrix $\mathbf{T}_\mathcal{B}$ can be done in parallel for all boxes on a level ℓ. The rank $k_{\max}$ is chosen adaptively for each box according to a prescribed user tolerance. Because of the adaptivity, this calculation is most fittingly done on the CPU with OpenMP.

In the apply stage, various parallel loops are indicated in Algorithm 2. The numerical results demonstrate the performance of a GPU implementation. To facilitate this implementation, we made a small number of modifications to the pseudocode in the algorithm.

First, each short vector $\mathbf{q}_\mathcal{B}, \hat{\mathbf{q}}_\mathcal{B}, \ldots$ is padded with zeros so they are all of the same length. The matrices $\mathbf{T}_\mathcal{B}$ are stored as tensor of size $n_{\text{boxes}} \times k_{\max} \times b$, where $k_{\max}$ is

the maximum skeleton rank and b is the leaf size of the tree decomposition. Then the computation becomes fairly regular in nature, despite operating on an adaptive tree data structure. Second, we noticed that much of the computation can be expressed as matrix-matrix multiplication. For instance,

$$\mathbf{X}^{(\mathrm{ifo})}_{\mathcal{B},\mathcal{B}'} = \mathbf{A}_{\mathcal{B},\mathcal{B}'} - \begin{pmatrix} \mathbf{0} & \mathbf{T}^{*}_{\mathcal{B}} \end{pmatrix} \mathbf{A}_{\mathcal{S}_{\mathcal{B}},\mathcal{B}'} - \mathbf{A}_{\mathcal{B},\mathcal{S}'_{\mathcal{B}}} \begin{pmatrix} \mathbf{0} \\ \mathbf{T}_{\mathcal{B}'} \end{pmatrix} - \begin{pmatrix} \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{\mathcal{S}_{\mathcal{B}},\mathcal{S}_{\mathcal{B}'}} \end{pmatrix} \tag{2.39}$$

Additionally, the pairwise distance calculation needed for evaluating sub-blocks of $\mathbf{A}$ can be expressed using matrix-matrix multiplication, and vectorized CUDA [78] implementations of many mathematical functions are available in `Python` packages. The simplicity of the algorithm, as well as the ability to express the algorithm using BLAS primitives [30], lends itself well to the implementation of Algorithm 2 on the GPU. To achieve good parallel performance, the implementation uses batched linear algebra libraries (e.g. batched BLAS). Batched BLAS performs linear algebraic operations on many small matrices by grouping them together and processing them in a single routine, to achieve effective parallelism with minimal overhead. The library is highly tuned for a variety of sub-task sizes, and the parallel load balancing is handled automatically.

The skeleton indices and interpolation matrices are loaded into the memory of the GPU prior to executing Algorithm 2. Depending on the amount of memory remaining on the GPU, a number of translation operators are formed, leveraging batched matrix-matrix multiplication as in (2.39). Then, these operators are multiplied by relevant short vectors (e.g. $\hat{\mathbf{q}}_{\mathcal{B}}$ for translation operator $\mathbf{X}^{(\mathrm{ifo})}$), and the result is accumulated into short vectors (i.e. $\hat{\mathbf{u}}_{\mathcal{B}}$ corresponding to the previous example). Then, the translation operators are discarded. They are fast to generate again as needed for subsequent evaluations of Algorithm 2 because of the efficiency of batched BLAS and hardware acceleration features. This approach works well for problems of modest size up to $N = 8\mathrm{M}$, but modified approaches will be required to scale to larger problem sizes, especially for high accuracy in 3D. The interpolation matrices $\mathbf{T}_{\mathcal{B}}$ require more memory because the rank is higher in 3D for fine user tolerances ϵ.

Algorithm 2 `FMM apply`

Input: $\mathbf{q} \in \mathbb{C}^N$. For each box $\mathcal{B}$, indices $I_{\mathcal{B}} = I_{\mathcal{S}} \cup I_R$, and matrix $\mathbf{T}_{\mathcal{B}}$.
Output: $\mathbf{u} = \mathbf{A}\,\mathbf{q}$.

for all boxes $\mathcal{B}$ **do**
  Set $\hat{\mathbf{q}}_{\mathcal{B}} := \mathbf{0}$ and $\hat{\mathbf{u}}_{\mathcal{B}} := \mathbf{0}$.
for $\ell = L, L-1, \ldots, 2$ **do** // upward pass
  for all box $\mathcal{B}$ at level ℓ **do** // in parallel
    if box $\mathcal{B}$ is a leaf **then**
      Set $\hat{\mathbf{q}}_{\mathcal{B}} = \mathbf{q}_{\mathcal{B}}$.
    else
      Accumulate $\hat{\mathbf{q}}_{\mathcal{B}}$ from skeleton indices of children $\hat{\mathbf{q}}_{\mathcal{S}_{\mathcal{C}}}$ for $\mathcal{C} \in \mathcal{C}(\mathcal{B})$.
    Set

$$\begin{pmatrix} \hat{\mathbf{q}}_{\mathcal{R}} \\ \hat{\mathbf{q}}_{\mathcal{S}} \end{pmatrix} := \begin{pmatrix} \mathbf{I} & \\ \mathbf{T}_{\mathcal{B}} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \hat{\mathbf{q}}_{\mathcal{R}} \\ \hat{\mathbf{q}}_{\mathcal{S}} \end{pmatrix}$$

for all box $\mathcal{B}$ with colleague neighbor $\mathcal{B}'$ (including $\mathcal{B}$) **do** // in parallel
  Evaluate

$$\hat{\mathbf{u}}_{\mathcal{B}} \mathrel{+}= \mathbf{X}^{(\mathrm{ifo})}_{\mathcal{B},\mathcal{B}'} \hat{\mathbf{q}}_{\mathcal{B}'}$$

for all box $\mathcal{B}$ with coarse neighbor $\mathcal{L}$ **do** // in parallel
  Evaluate

$$\hat{\mathbf{u}}_{\mathcal{B}} \mathrel{+}= \mathbf{X}^{(\mathrm{ifs})}_{\mathcal{B},\mathcal{L}} \mathbf{q}_{\mathcal{L}}$$

for all box $\mathcal{B}$ on level 2 **do**
  Evaluate

$$\hat{\mathbf{u}}_{\mathcal{S}_{\mathcal{B}}} \mathrel{+}= \sum_{\mathcal{B}' \text{ on level 2}} \mathbf{A}_{\mathcal{S}_{\mathcal{B}} \mathcal{S}'_{\mathcal{B}}} \hat{\mathbf{q}}_{\mathcal{S}'_{\mathcal{B}}}.$$

for $\ell = 2, \ldots, L$ **do** // downward pass
  for all box $\mathcal{B}$ at level ℓ **do** // in parallel
    Set

$$\begin{pmatrix} \hat{\mathbf{u}}_{\mathcal{R}} \\ \hat{\mathbf{u}}_{\mathcal{S}} \end{pmatrix} := \begin{pmatrix} \mathbf{I} & \mathbf{T}^*_{\mathcal{B}} \\ & \mathbf{I} \end{pmatrix} \begin{pmatrix} \hat{\mathbf{u}}_{\mathcal{R}} \\ \hat{\mathbf{u}}_{\mathcal{S}} \end{pmatrix}$$

    if box $\mathcal{B}$ is a leaf **then**
      Set $\mathbf{u}_{\mathcal{B}} = \hat{\mathbf{u}}_{\mathcal{B}}$.
    else
      Copy values $\hat{\mathbf{u}}_{\mathcal{B}}$ to skeleton indices of children $\hat{\mathbf{u}}_{\mathcal{S}_{\mathcal{C}}}$ for $\mathcal{C} \in \mathcal{C}(\mathcal{B})$.
for all box $\mathcal{B}$ with coarse neighbor $\mathcal{L}$ **do** // in parallel
  Evaluate

$$\mathbf{u}_{\mathcal{L}} \mathrel{+}= \mathbf{X}^{(\mathrm{tfo})}_{\mathcal{L},\mathcal{B}} \hat{\mathbf{q}}_{\mathcal{B}}$$

2.4 Complexity analysis

In this section, the computational complexity and storage costs of Algorithms 1 and 2 are analyzed. Assume that the point distribution of N particles is partitioned into a tree with at most b points in the leaf boxes and that the numerical rank from ID compression is a constant $k_{\max}$. Let n_{boxes} be the total number of boxes in the tree and n_{neigh} be the maximum number of near-neighbor boxes. The quantity n_{boxes} is a function of other problem parameters and depends on the dimension of points in $\mathcal{X}$:

$$n_{\text{boxes}} = c_{\text{tree}} \frac{N}{b}, \tag{2.40}$$

where $c_{\text{tree}} = \frac{4}{3}$ for a uniform quad-tree and $c_{\text{tree}} = \frac{8}{7}$ for a uniform oct-tree. Likewise, N_{neigh} also depends on the geometry of points in $\mathcal{X}$; for instance, $N_{\text{neigh}} = 9$ for a uniform quad-tree and $N_{\text{neigh}} = 27$ for a uniform oct-tree. Let t_{kernel} and t_{flop} be constants for the time needed for one kernel evaluation and one linear algebraic flop, respectively.

Precomputation Costs: For each box, we evaluate the kernel interactions between indices I_B and proxy surface $\mathcal{X}_{\text{proxy}}$ then compute the column ID. Assuming the accumulated skeleton indices have size at most b and the proxy surface has at most b points, then the total cost is

$$T_{\text{skel}} = n_{\text{boxes}} \left(b^2 \, t_{\text{kernel}} + b^2 \, k_{\max} \, t_{\text{flop}} \right) = c_{\text{tree}} \left(b \, t_{\text{kernel}} + b \, k_{\max} \, t_{\text{flop}} \right) N \tag{2.41}$$

The memory cost of storing the interpolation operator $\mathbf{T}_{\mathcal{B}}$ for each box is

$$M_{\text{proj}} = n_{\text{boxes}} \, b \, k_{\max} = c_{\text{tree}} \, k_{\max} \, N \tag{2.42}$$

Algorithm Apply: For each box, the translation operators are evaluated for at most n_{neigh} neighbors. Each translation operator requires evaluating a kernel matrix $\mathbf{A}_{\mathcal{B},\mathcal{B}'}$ for a sub-matrix of size $b \times b$, then applying interpolation matrices on the left and

right, requiring at most $2\ k_{\max}^2\ b$ flops. Finally, evaluating the matrix vector product requires b^2 flops. Then the total cost is

$$\begin{aligned} T_{\text{apply}} &= n_{\text{boxes}}\ n_{\text{neigh}}\ \left(b^2\ t_{\text{kernel}} + 2k_{\max}^2\ b\ t_{\text{flop}} + b^2 t_{\text{flop}}\right) \\ &= c_{\text{tree}}\ n_{\text{neigh}}\ \left(b\ t_{\text{kernel}} + 2\ k_{\max}^2\ b\ t_{\text{flop}} + b\ t_{\text{flop}}\right)\ N \\ &:= T_{\text{kernel}}\ +\ T_{\text{mod}}\ +\ T_{\text{mv}}, \end{aligned} \tag{2.43}$$

where the terms in the last line break down the total apply time T_{apply} in terms of various operations; T_{kernel} is the total time for kernel evaluation, T_{mod} for the total time for modifying near-neighbor interactions, T_{mv} is the total time to multiply modified operators by vectors.

Note that T_{mod} has a potentially large pre-factor of $2\ k_{\max}^2\ b$. Despite this cost, leveraging BLAS primitives in the implementation makes the wall-clock time fairly small. In Table 2.2, we report a break-down of the wall-clock time for a non-uniform distribution of points on the unit sphere for Laplace and Helmholtz kernels for various leaf sizes b and maximum skeleton rank $k_{\max}$.

N	b	$k_{\max}$	T_{kernel}	T_{mod}	T_{mv}	T_{apply}
	120	22	0.61 s	0.23 s	0.03 s	0.88 s
1 M	160	48	1.20 s	0.58 s	0.06 s	2.80 s
	400	110	2.44 s	1.41 s	0.11 s	3.97 s

(a) 3D Laplace kernel (2.2)

N	b	$k_{\max}$	T_{kernel}	T_{mod}	T_{mv}	T_{apply}
	120	34	1.27 s	0.59 s	0.06 s	1.93 s
1 M	220	70	3.07 s	1.66 s	0.14 s	4.90 s
	750	138	7.48 s	5.04 s	0.30 s	12.84 s

(b) 3D Helmholtz kernel (2.4) with $\kappa = 20$

Table 2.2: Tables 2.2a and 2.2b report the timing breakdown for T_{apply} for a random distribution of points on the unit sphere for kernel functions (2.2) and (2.4), using a variety of parameters for the leaf size b and the maximum skeleton rank $k_{\max}$. Although T_{mod} has a large constant pre-factor, the modifications are implemented using matrix-matrix multiply, which is particularly efficient on the GPU. In all the cases reported, T_{kernel} is the dominant cost of the total apply time T_{apply}.

2.5 Numerical results

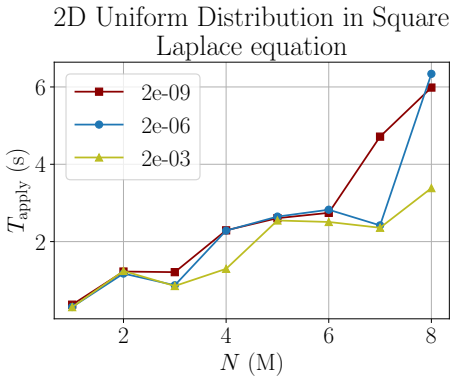
In this section, the performance of a parallel implementation is demonstrated for a variety of uniform and non-uniform point distributions in 2D and 3D. The FMM is described on an adaptive tree $\mathcal{T}$ which satisfies the 2:1 balance constraint, as described in Section 2.3.1. To produce $\mathcal{T}$, an unbalanced adaptive tree is created with at most b points in the leaf boxes. Then, the tree is balanced by adding additional leaf nodes in a sequential algorithm; geometric look-ups are efficiently implemented using Morton codes. The time to partition and balance the tree is reported as T_{tree}.

Before the FMM is applied, some precomputation is necessary to compute the skeleton indices for each box $\mathcal{B}$ and the interpolation matrix $\mathbf{T}_{\mathcal{B}}$ which satisfies equation (2.7). The procedure is described in Algorithm 1. At each level of the tree, the relevant information is computed for each box embarrassingly in parallel on the CPU using OpenMP. The skeleton rank k for each box is computed adaptively according to provided user tolerance. In the numerical results, we report the maximum skeleton rank $k_{\max}$. The time required for Algorithm 1 is reported as T_{skel}. The memory required to store $\mathbf{T}_{\mathcal{B}}$ for all boxes is reported as M_{proj}.

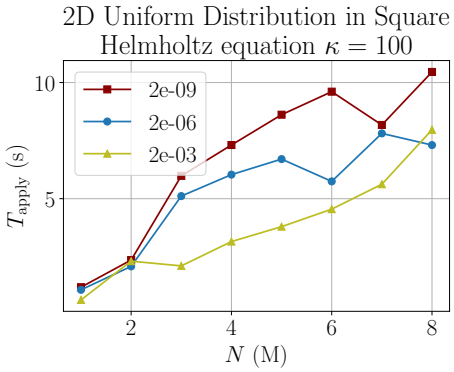
N	number of points
b	leaf size of tree
$k_{\max}$	maximum skeleton rank
T_{tree}	time to partition and balance tree
T_{skel}	time for Algorithm 1
M_{proj}	memory needed for $\mathbf{T}_{\mathcal{B}}$ for all boxes
T_{apply}	time for Algorithm 2
relerr	relative maximum error (on a subset of points)

Table 2.3: Notation for reported numerical results.

To apply the FMM with GPU acceleration using Algorithm 2, the matrices $\mathbf{T}_{\mathcal{B}}$ are first moved to the GPU for storage. The modifications needed for efficient GPU implementation using batched linear algebra are described in Section 2.3.4. Because the translation operators would require significant memory to store, we instead



(a) 2D Laplace kernel (2.2)



(b) 2D Helmholtz kernel (2.4) with $\kappa = 100$

N	b	$k_{\max}$	T_{tree}	T_{skel}	M_{proj}	T_{apply}	relerr
1 M	120	8	1.9 s	3.4 s	0.1 GB	0.30 s	3.23e-03
		16	1.8 s	3.5 s	0.3 GB	0.31 s	1.91e-06
		29	1.8 s	3.6 s	0.6 GB	0.37 s	1.82e-09
8 M	120	8	27.4 s	33.3 s	1.8 GB	3.38 s	3.05e-03
		16	27.4 s	31.8 s	3.6 GB	6.34 s	1.51e-06
		29	27.4 s	33.1 s	6.5 GB	5.98 s	1.79e-09

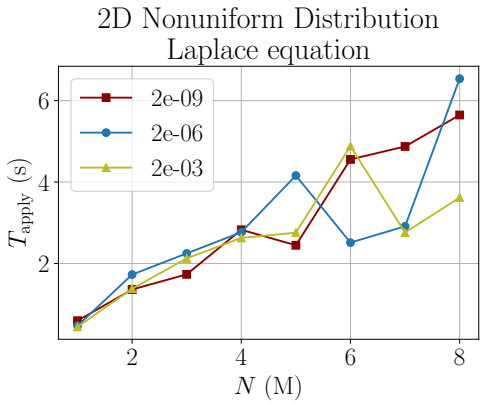
(c) 2D Laplace kernel (2.2)

N	b	$k_{\max}$	T_{tree}	T_{skel}	M_{proj}	T_{apply}	relerr
1 M	120	49	1.8 s	7.0 s	2.1 GB	0.64 s	1.45e-03
	140	59	1.8 s	10.4 s	3.2 GB	1.08 s	1.23e-06
	180	68	1.8 s	14.4 s	4.6 GB	1.19 s	1.40e-09
8 M	120	49	27.4 s	47.6 s	22.2 GB	7.96 s	1.50e-03
	140	60	24.9 s	51.5 s	15.2 GB	7.31 s	3.33e-06
	180	69	20.5 s	71.1 s	18.5 GB	10.46 s	1.35e-09

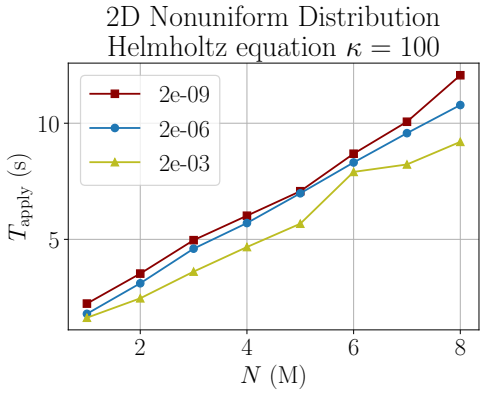
(d) 2D Helmholtz kernel (2.4) with $\kappa = 100$

Figure 2.7: The performance on a random distribution of points in the unit square is demonstrated. Figure 2.7a reports T_{apply} for kernel (2.2) with additional data reported in Table 2.7c. Similarly, Figure 2.7b and Table 2.7d report results for kernel (2.4) with $\kappa = 100$. In the pre-computation stage, skeleton indices and operators $\mathbf{T}_{\mathcal{B}}$ are computed adaptively to provided user tolerance; this is done in parallel on the CPU with OpenMP. Then, the relevant operators $\mathbf{T}_{\mathcal{B}}$ are stored on the GPU, and the FMM is evaluated rapidly using batched BLAS primitives and specialized hardware accelerations on the GPU.

generate the operators at run-time, evaluate them, store the accumulated results in vectors $\hat{\mathbf{q}}_{\mathcal{B}}, \hat{\mathbf{u}}_{\mathcal{B}}, \mathbf{u}_{\mathcal{B}}$, then discard the translation operators. Though this procedure



(a) 2D Laplace kernel (2.2)



(b) 2D Helmholtz kernel (2.4) with $\kappa = 100$

N	b	$k_{\max}$	T_{tree}	T_{skel}	M_{proj}	T_{apply}	relerr
1 M	120	13	4.3 s	3.4 s	0.3 GB	0.44 s	8.58e-04
		25	4.4 s	3.7 s	0.5 GB	0.47 s	1.69e-06
		36	4.3 s	4.6 s	0.8 GB	0.59 s	1.43e-09
8 M	120	14	50.2 s	34.7 s	3.3 GB	3.62 s	2.61e-03
		25	50.0 s	33.3 s	5.9 GB	6.54 s	9.14e-06
		37	50.1 s	35.3 s	10.5 GB	5.65 s	3.17e-09

(c) 2D Laplace kernel (2.2)

N	b	$k_{\max}$	T_{tree}	T_{skel}	M_{proj}	T_{apply}	relerr
1 M	150	89	3.4 s	7.8 s	4.7 GB	1.62 s	1.07e-03
	190	103	3.4 s	11.1 s	6.6 GB	1.79 s	1.92e-06
	200	115	3.6 s	15.7 s	8.6 GB	2.23 s	1.66e-09
8 M	150	90	47.8 s	41.9 s	32.9 GB	9.20 s	2.02e-03
	190	103	46.3 s	53.6 s	28.2 GB	10.78 s	2.80e-06
	200	115	46.3 s	77.3 s	36.1 GB	12.07 s	6.71e-09

(d) 2D Helmholtz kernel (2.4) with $\kappa = 100$

Figure 2.8: The performance of the method for the non-uniform 2D geometry shown in Figure 2.6 is demonstrated. Figure 2.8a reports T_{apply} for kernel (2.2) with additional data reported in Table 2.8c. Similarly, Figure 2.8b and Table 2.8d report results for kernel (2.4) with $\kappa = 100$. Though the algorithm operates on an adaptive tree, the computation is made to be fairly uniform and executed efficiently on the GPU.

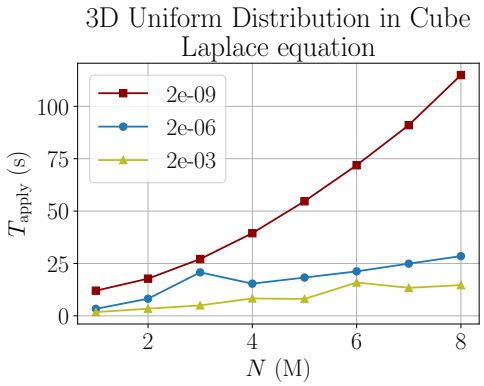
seems wasteful, it is instead very efficient. The translation operators are very small $b \times b$ matrices, and it is more efficient to generate them on the fly (using batched BLAS primitives), then to retrieve them from memory. Our implementation also

leverages matrix-matrix multiplication when possible as described in Section 2.3.4; this operation is highly efficient on the GPU. In the numerical results, the time to apply Algorithm 2 on the GPU is reported as T_{apply}. The maximum relative error of $\mathbf{u}$ is also reported as relerr; this quantity is approximated by evaluating the true potential at a small number of points.

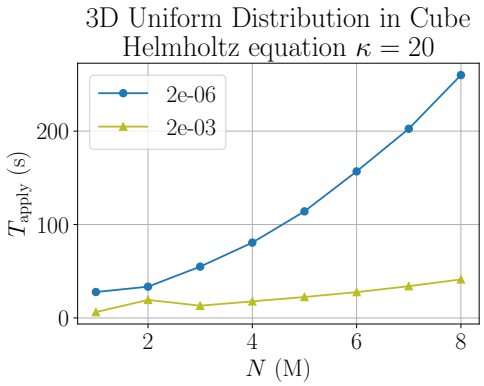
The code is implemented in Python. The pre-processing stage described in Algorithm 1 uses packages `numpy` [56], `scipy` [90], and `multiprocessing`. The GPU implementation of the FMM apply in Algorithm 2 uses the package `PyTorch` [79], which has an interface to highly-efficient batched BLAS libraries as well as many mathematical functions implemented in CUDA. For the Hankel function $H_0^{(1)}$ in kernel (2.4), a CUDA implementation is available in the package `Cupy`. Though `PyTorch` is a library designed for deep learning, gradient calculations are disabled, and it is used only for its linear algebra functionality. The numerical results were run on a workstation with an Intel Xeon Gold 6326 CPU (running at 2.9 GHz) with 16 cores and access to 250GB of RAM; the workstation also has an NVIDIA A100 GPU with 80GB of RAM.

2.5.1 2D Experiments

Figure 2.7 reports results for a random distribution of points in the unit square, and Figure 2.8 reports results for a random non-uniform distribution of points, shown in Figure 2.6. Both tables show the performance of the method for the Laplace and Helmholtz kernel for $\kappa = 100$ for a variety of user tolerances. For 2D distributions, the skeleton rank scales roughly as $k_{\max} \sim \log(1/\epsilon)$ for the Laplace equation. The method performs very well on the GPU, and making the user tolerance ϵ finer does not leads only to minor increases to the runtime T_{apply}.



(a) 3D Laplace kernel (2.2)



(b) 3D Helmholtz kernel (2.4) with $\kappa = 20$

N	b	$k_{\max}$	T_{tree}	T_{skel}	M_{proj}	T_{apply}	relerr
1 M	180	40	2.0 s	6.7 s	3.6 GB	1.76 s	2.70e-04
	750	100	1.2 s	6.7 s	2.9 GB	3.34 s	1.90e-06
	2100	277	0.9 s	55.0 s	2.8 GB	12.00 s	1.78e-09
8 M	180	45	24.3 s	51.8 s	36.9 GB	14.65 s	5.06e-04
	750	110	14.4 s	34.9 s	28.2 GB	28.50 s	2.10e-06
	2100	289	13.8 s	213.4 s	24.9 GB	114.93 s	4.45e-09

(c) 3D Laplace kernel (2.2)

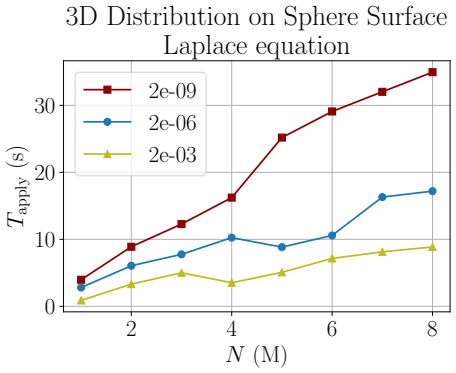
N	b	$k_{\max}$	T_{tree}	T_{skel}	M_{proj}	T_{apply}	relerr
1 M	500	91	1.2 s	7.4 s	3.6 GB	6.22 s	1.60e-03
	2100	262	0.9 s	87.3 s	5.1 GB	27.77 s	1.41e-06
8 M	500	91	14.5 s	41.1 s	30.8 GB	41.26 s	2.50e-03
	2100	268	13.8 s	339.9 s	42.2 GB	260.01 s	1.20e-06

(d) 3D Helmholtz kernel (2.4) with $\kappa = 20$

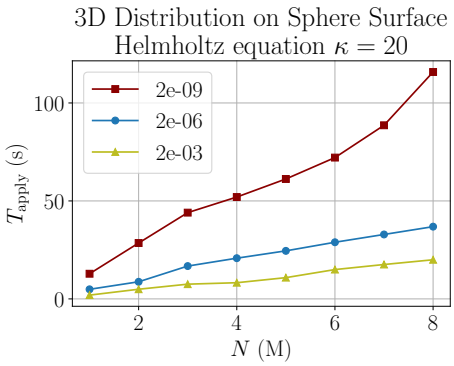
Figure 2.9: The performance of the method for a random distribution of points in the unit cube is demonstrated. Figure 2.9a reports T_{apply} for kernel (2.2) with additional data reported in Table 2.9c. Similarly, Figure 2.9b and Table 2.9d report results for kernel (2.4) with $\kappa = 20$. The skeleton rank $k_{\max}$ is large for fine tolerances ϵ, leading to large storage costs for $\mathbf{T}_{\mathcal{B}}$ and worse performance for batched linear algebra.

2.5.2 3D Experiments

Figure 2.9 reports results for a random distribution of points in the unit cube, and Figure 2.10 reports results for a random distribution of points on the surface of a unit sphere. Both tables show the performance of the method for the Laplace and Helmholtz kernel for $\kappa = 20$ for a variety of user tolerances.



(a) 3D Laplace kernel (2.2)



(b) 3D Helmholtz kernel (2.4) with $\kappa = 20$

N	b	$k_{\max}$	T_{tree}	T_{skel}	M_{proj}	T_{apply}	relerr
1 M	120	22	4.3 s	3.9 s	0.5 GB	0.88 s	1.03e-03
	160	48	3.2 s	4.8 s	1.9 GB	2.80 s	1.82e-06
	400	110	3.0 s	9.1 s	3.2 GB	3.97 s	2.53e-09
8 M	120	25	49.1 s	40.7 s	7.4 GB	8.86 s	7.77e-04
	160	55	48.6 s	39.7 s	25.3 GB	17.19 s	1.75e-06
	400	121	34.8 s	53.2 s	47.1 GB	34.95 s	1.84e-09

(c) 3D Laplace kernel (2.2)

N	b	$k_{\max}$	T_{tree}	T_{skel}	M_{proj}	T_{apply}	relerr
1 M	120	34	4.2 s	5.5 s	1.5 GB	1.93 s	1.69e-03
	220	70	3.2 s	10.7 s	6.4 GB	4.90 s	1.97e-06
	750	138	2.2 s	31.6 s	8.4 GB	12.84 s	2.00e-09
8 M	120	34	49.1 s	53.0 s	24.3 GB	19.97 s	2.06e-03
	220	70	43.1 s	65.0 s	34.1 GB	36.87 s	2.82e-06
	750	138	34.0 s	186.4 s	58.1 GB	115.80 s	2.48e-09

(d) 3D Helmholtz kernel (2.4) with $\kappa = 20$

Figure 2.10: The performance of the method for a non-uniform distribution of random points on the surface of a sphere is demonstrated. Figure 2.10a reports T_{apply} for kernel (2.2) with additional data reported in Table 2.10c. Similarly, Figure 2.10b and Table 2.10d report results for kernel (2.4) with $\kappa = 20$. The skeleton basis is highly effective in capturing the local geometry of the surface, leading to skeleton rank that does not grow as substantially with desired user tolerance ϵ.

For 3D point distributions, the skeleton ranks $k_{\max}$ are considerably higher, leading to poor performance of the method for uniform 3D distributions. This phenomenon is well known for 3D FMMs. In particular for analytic multipole expansions

the rank grows as $k \sim (\log(1/\epsilon))^2$ for the Laplace equation. Much work has been done in creating diagonal translation operators [50, 81] using analytic techniques, or in using FFT acceleration for kernel-independent algebraic methods [64, 101]. Given that the present work included no such machinery, the poor performance for volume distributions is not very surprising. The method, however, performs particularly well for point distributions on surfaces, as shown in Figure 2.10. The skeleton basis can capture the local geometry with a much smaller skeleton rank $k_{\max}$, even to high accuracy.

2.6 Conclusions

This work introduces a novel kernel-independent algorithm for the fast multipole method. Because the algorithm is kernel-independent, it can readily be extended to a variety of constant-coefficient kernels. The numerical results demonstrate the effectiveness of the approach for Laplace and low-frequency Helmholtz problems. Similar to other kernel-independent approaches, analytical expansions are replaced by a chosen set of 'skeleton' points which replicate the effect of the original source points in the far-field. The use of skeleton basis is particularly effective for representing incoming and outgoing representations for point distributions on surfaces.

The major contribution of this work is a simplified description of the FMM that does not involve the interaction list. The interaction list contains only well-separated boxes and requires additional book-keeping to be traversed efficiently. Instead, our algorithm requires algebraically modified kernel interaction between near neighbors. There is an additional flop cost to calculate the modifications, but the numerical results demonstrate that this can be done efficiently by leveraging BLAS primitives.

Finally, the work includes an implementation of the proposed algorithm on modern hardware architectures, using OpenMP on the CPU for the precomputation stage and batched BLAS primitives for the GPU implementation of the FMM apply. The simplicity of the proposed algorithm make it particularly amenable to implemen-

tation on a variety of hardware architectures and to extension to a broad range of kernels.

Acknowledgments

The work reported was supported by the Office of Naval Research (N00014-18-1-2354), by the National Science Foundation (DMS-1952735, DMS-2012606, and DMS-2313434), and by the Department of Energy ASCR (DE-SC0022251). We thank Umberto Villa for access to computing resources.

Chapter 3: SlabLU: A Two-Level Sparse Direct for Elliptic PDEs [2]

The work describes a sparse direct solver for the linear systems that arise from the discretization of an elliptic PDE on a two dimensional domain. The solver is designed to reduce communication costs and perform well on GPUs; it uses a two-level framework, which is easier to implement and optimize than traditional multi-frontal schemes based on hierarchical nested dissection orderings. The scheme decomposes the domain into thin subdomains, or "slabs". Within each slab, a local factorization is executed that exploits the geometry of the local domain. A global factorization is then obtained through the LU factorization of a block-tridiagonal reduced coefficient matrix. The solver has complexity $O(N^{5/3})$ for the factorization step, and $O(N^{7/6})$ for each solve once the factorization is completed.

The solver described is compatible with a range of different local discretizations, and numerical experiments demonstrate its performance for regular discretizations of rectangular and curved geometries. The technique becomes particularly efficient when combined with very high-order convergent multi-domain spectral collocation schemes. With this discretization, a Helmholtz problem on a domain of size $1000\lambda \times 1000\lambda$ (for which $N = 100\text{M}$) is solved in 15 minutes to 6 correct digits on a high-powered desktop with GPU acceleration.

[2] This work was completed in collaboration with Per-Gunnar Martinsson and has appeared in preprints [96, 97].

3.1 Introduction

3.1.1 Problem setup

We present a direct solver for boundary value problem of the form

$$\begin{cases} \mathcal{A}u(x) = f(x), & x \in \Omega, \\ u(x) = g(x), & x \in \partial\Omega, \end{cases} \tag{3.1}$$

where $\mathcal{A}$ is a second order elliptic differential operator, and Ω is a domain in two dimensions with boundary $\partial\Omega$. The method works for a broad range of constant and variable coefficient differential operators, but is particularly competitive for problems with highly oscillatory solutions that are difficult to pre-condition. For the sake of concreteness, we will focus on the case where $\mathcal{A}$ is a variable coefficient Helmholtz operator

$$\mathcal{A}u(x) = -\Delta u(x) - \kappa^2 b(x) u(x), \tag{3.2}$$

where κ is a reference ("typical") wavenumber, and where $b(x)$ is a smooth non-negative function. Upon discretizing (3.1), one obtains a linear system

$$\mathbf{Au} = \mathbf{f}, \tag{3.3}$$

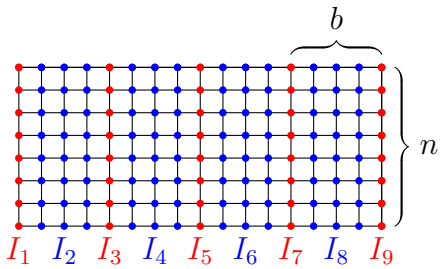
involving a coefficient matrix $\mathbf{A}$ that is typically sparse. Our focus is on efficient algorithms for directly building an invertible factorization of the matrix $\mathbf{A}$. We specifically consider two different discretization schemes, first a basic finite difference scheme with second order convergence, and then a high (say $p = 20$) order multidomain spectral collocation scheme [68, Ch. 25]. However, the techniques presented can easily be used with other (local) discretization schemes such as finite element methods.

3.1.2 Overview of proposed solver

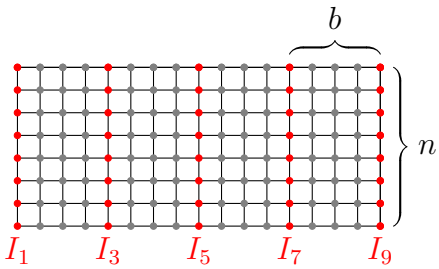
The solver presented is based on a decomposition of the computational domain into thin "slabs", as illustrated in Figure 3.1a. Unlike previously proposed sweeping schemes [33, 39, 89] designed for preconditioning, our objective is to directly factorize

the coefficient matrix, or at least compute a factorization that is sufficiently accurate that it can handle problems involving strong backscattering.

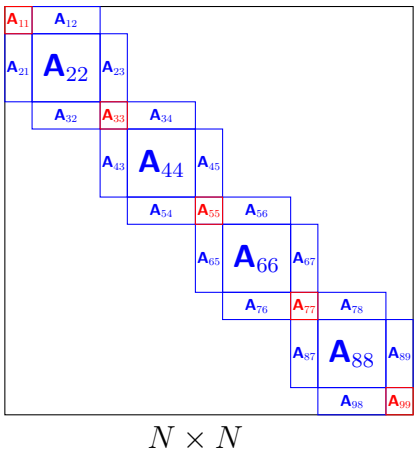
To describe how the solver works, let us consider a simple model problem where the PDE is discretized using a standard five-point finite difference stencil on a uniform grid such as the one shown in Figure 3.1a. The nodes in the grid are arranged into slabs of width b, and are ordered as shown in Figure 3.1a, resulting in a coefficient matrix $\mathbf{A}$ with the block diagonal sparsity pattern shown in Figure 3.1c. The factorization of $\mathbf{A}$ then proceeds through two stages.



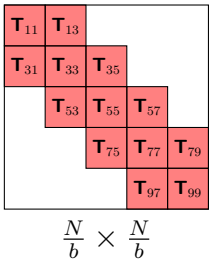
(a) Original grid, partitioned into slabs of width $b + 2$, where $b = 3$.



(b) Reduced grid, after eliminating blue nodes. Only the red nodes are "active".



(c) Sparsity structure of $\mathbf{A}$ corresponding to the original grid. Each block of $\mathbf{A}$ is sparse.



(d) Sparsity structure of $\mathbf{T}$ corresponding to the reduced grid. Each block of $\mathbf{T}$ is dense.

Figure 3.1: Illustration of the elimination order used in SlabLU.

In the first stage, the nodes that are internal to each slab (identified by the index vectors $I_2, I_4, \ldots$ and shown as blue in Figure 3.1a) are eliminated from the linear system, resulting in the reduced problem shown in Figure 3.1b, with the associated coefficient matrix shown in Figure 3.1d. In this elimination step, we exploit that each subdomain is thin, which means that classical sparse direct solvers are particularly fast. To further accelerate this step, we use that the Schur complements that arise upon the elimination of the interior nodes are rank structured. Specifically, they are "HBS/HSS matrices" [45, 93, 94] with *exact* HBS/HSS rank at most $2b$. This allows us to accelerate this reduction step using a recently proposed randomized algorithm for compressing rank structured matrices [60].

The second stage is to factorize the remaining coefficient matrix $\mathbf{T}$ shown in Figure 3.1d. This matrix is much smaller than the original matrix $\mathbf{A}$, but its blocks are dense. These blocks are rank structured as well, which could in principle be exploited to attain a linear complexity solver (for the case where the PDE is kept fixed as N increases). However, we have found that for 2D problems, the dense operations are fast enough that exploiting rank structure for the second step is not worthwhile when $N \leq 10^8$.

A particular advantage of our framework is that it is simple to optimize and accelerate on GPUs, compared to traditional multi-frontal schemes. The numerical results demonstrate compelling speed and memory scaling, when compared to SuperLU [61]. For meshes with 100 million points, the factorization can be computed in 20 minutes on a desktop with an Intel i9-12900k CPU with 16 cores and an RTX 3090 GPU. Once the factorization is available, subsequent solves take about a minute. The numerical results feature timing results on a variety of architectures to demonstrate that the scheme is portable to many hardware settings.

The scheme also interacts very well with high order discretization schemes such as those described in [66] and [68, Ch. 25], which makes it a particularly powerful tool for solving problems with highly oscillatory solutions. The numerical results feature

constant and variable coefficient Helmholtz problems on rectangular and curved domains. Using high order discretizations, we are able to discretize the PDE to 10 points per wavelength and accurately resolve solutions on domains of size $1000\lambda \times 1000\lambda$, where λ is the wavelength, to 6 digits of relative accuracy, compared to the true solution of the PDE.

3.1.3 Context and related work

Methods to solve (3.3) can be characterized into two groups – direct and iterative. The linear systems involved are typically ill-conditioned, which necessitates the use of pre-conditioners or multigrid solvers. However, preconditioners typically struggle for problems with oscillatory solutions that involve phenomena such as multiple reflections, back-scattering, and waves trapped in cavities [34, 39]. Sparse direct methods, which factorize matrix $\mathbf{A}$ exactly, offer a robust means of solving challenging PDEs. Direct solvers are also particularly advantageous in situations involving multiple right-hand sides or low-rank updates to the matrix $\mathbf{A}$.

The solver we describe in this work is related to multi-frontal LU solvers [25] which often use a hierarchical nested dissection ordering of grid nodes [4, 40]. For a 2D grid with N nodes, the resulting techniques have complexity $O(N^{3/2})$ to build and $O(N \log N)$ complexity to solve, which is known to be work optimal among solvers that exploit only sparsity in the system [24, 31].

By replacing the nested dissection ordering with a partition into thin slabs, SlabLU improves communication efficiency, and allows the method to better take advantage of hardware accelerators. Nevertheless, its asymptotic flop count is only slightly sub-optimal. Specifically, by choosing $b \sim O(n^{2/3})$, sparsity alone results in complexity $O(N^{5/3})$ for the factorization stage, and $O(N^{7/6})$ for the solve stage, when applied to 2D problems.

The use of rank structured matrix algebra to accelerate sparse direct solvers is inspired by prior work including [3, 18, 41, 44, 80, 95]. A key feature of SlabLU

is that unlike prior work, the rank structures that we exploit are *exact*, relying only on the sparsity pattern of the original coefficient matrix (cf. Section 3.3.2). Importantly, these rank-deficiencies are present in both the non-oscillatory and oscillatory regimes. This makes the randomized compression particularly efficient, achieving very high computational efficiency with no loss of accuracy beyond floating point errors. Another novelty is the usage of a recently developed black box randomized algorithm for compressing rank structured matrices [60] (which in turn draws on insights from [63, 67, 73]) when eliminating the interior nodes in each slab. The use of black-box randomized algorithms makes the solver relatively simple to implement, and easier to port between different local discretization.

SlabLU is also related to Schur complement methods, which are two-level methods based on a non-overlapping domain decomposition with implicit treatment of the interface conditions. The interior of each subdomain is factorized in parallel; the reduced coefficient matrix $\mathbf{T}$ is typically not formed but instead applied matrix-free in an iterative method [16, 83]. These methods have similar advantages as SlabLU in that they are easy to parallelize. They implicitly solve a reduced system that is better conditioned than the original system, but is often difficult to precondition for oscillatory problems.

3.1.4 Extensions and limitations

The solver presented is purely algebraic and can be applied to a range of different discretization schemes, including finite element and finite volume methods. In this manuscript, we restrict attention to uniform grids and domains that are either rectangular themselves, or can easily be mapped to a union of rectangles. It is in principle possible to adapt the method to more general discretizations, including ones that involve local refinement. However, some of the accelerations (e.g. batched linear algebra) would in this case lose efficiency to some degree.

While we in this manuscript restrict attention to the two dimensional case, the

method is designed to handle three dimensional problems as well. All ideas presented carry over directly, but additional complications do arise. The key challenge is that in three dimensions, it is no longer feasible to use dense linear algebra when factorizing the block tridiagonal reduced coefficient matrix $\mathbf{T}$. Instead, rank structure must be exploited in both stages of the algorithm, and rank deficiencies will no longer be exact. However, the 3D version of SlabLU is also very easy to parallelize, and the idea of using randomized compression combined with efficient sparse direct solvers to eliminate the nodes interior to each slab still applies, cf. Section 3.7.

3.2 Discretization and node ordering

We introduce two different discretization techniques for (3.1). The first is simply the standard second order convergent five point finite difference stencil. Since this discretization is very well known, it allows us to describe how the solver works without the need to introduce cumbersome background material. To demonstrate that the solver works for a broader class of discretization schemes, the numerical experiments reported in Section 3.6 also include results that rely on the high order convergent *Hierarchical Poincaré-Steklov (HPS)* scheme, which we briefly describe in Section 3.2.3.

3.2.1 A model problem based on the five point stencil

For purposes of describing the factorization scheme, let us introduce a very simple discretization of the boundary value problem (3.1). We work with a rectangular domain $\Omega = [0, L_1] \times [0, L_2]$ and the second order linear elliptic operator $\mathcal{A}$ defined by (3.2). We assume that $L_1 \geq L_2$, and that $L_1 = hn_1$ and $L_2 = hn_2$ for some grid spacing h and some positive integers n_1 and n_2. We then discretize $\mathcal{A}$ with a standard second-order finite difference scheme, to obtain the linear system

$$\frac{1}{h^2}\big(\mathbf{u}(n_{\rm w}) + \mathbf{u}(n_{\rm e}) + \mathbf{u}(n_{\rm n}) + \mathbf{u}(n_{\rm s}) - 4\mathbf{u}(n)\big) - \kappa^2\mathbf{b}(n)\mathbf{u}(n) = \mathbf{f}(n). \qquad (3.4)$$

The vector $\mathbf{f}$ holds values of the body load at the discretization nodes, and the vector u holds approximations to the solution u. See Figure 3.2 for a visualization of the 5 point stencil. We write the system (3.4) compactly as $\mathbf{Au} = \mathbf{f}$.

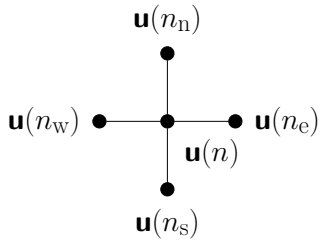


Figure 3.2: Five point stencil in 2D.

3.2.2 Clustering of the nodes

We next subdivide the computational domain into thin "slabs", as shown in Figure 3.1a. We let b denote the number of grid points in each slab ($b = 4$ in Figure 3.1a), and then introduce index vectors I_1, I_2, $I_3, \ldots$ that keep track of which slabs each grid point belongs to. The odd numbered index vectors I_1, I_3, $I_5, \ldots$ indicate nodes on the interfaces between slabs (red in Figure 3.1a), while the even numbered ones indicate nodes that are interior to each slab (blue in Figure 3.1a). With this ordering of the grid points, the stiffness matrix associated with the discretization (3.4) has the sparsity pattern shown in Figure 3.1c.

3.2.3 High order discretizations

To accurately resolve oscillatory wave phenomena, we rely on a high order accurate multi-domain spectral collocation discretization known as the Hierarchical Poincaré-Steklov scheme (HPS). This discretization scheme is designed to allow for high choices of the local discretization order p without degrading the performance of direct solvers.

In HPS, the computational domain is subdivided into small rectangles, and a $p \times p$ tensor product grid of Chebyshev nodes is placed on each rectangle, cf. Figure

3.3. The vector of unknowns $\mathbf{u}$ simply holds approximations to the solution u at the discretization nodes. We then discretize (3.1) through collocation of the spectral differentiation operator for each internal node. For the nodes on cell boundaries, we enforce continuity of the normal derivatives, again through spectral differentiation.

To improve efficiency when HPS is combined with sparse direct solvers, we "eliminate" the dense interactions of nodes interior to each subdomain through a static condensation step. Due to the domain decomposition used in HPS, the leaf operations required to produce the equivalent system are independent and can be performed in parallel. This leaves us with a reduced linear system that involves the approximately $n_1 n_2/p$ points that sit on the edges between cells. We denote the reduced stiffness matrix by $\tilde{\mathbf{A}}$, and the reduced load vector by $\tilde{\mathbf{f}}$, cf. Figure 3.3, to obtain the equivalent system

$$\tilde{\mathbf{A}}\tilde{\mathbf{u}} = \tilde{\mathbf{f}}. \tag{3.5}$$

For details, see [68, Ch. 25], as well as [6, 46, 55, 66].

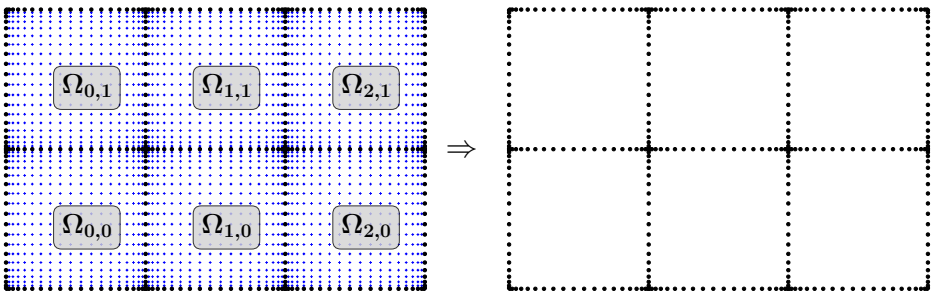


Figure 3.3: HPS is a multi-domain spectral collocation scheme where the PDE is enforced on each subdomain interior using dense spectral differentiation. Prior to interfacing with SlabLU, we "eliminate" the interior blue nodes in parallel and produce an equivalent system to solve on the boundaries. The original grid has $n_1 \times n_2$ points, and remaining grid has $\approx n_1 n_2/p$ points.

Remark 3.1. A key point of the present work is that the solver has only two levels, which makes the "H" in HPS a slight misnomer, as it refers to "hierarchical". We nevertheless stick with the HPS acronym to conform with the prior literature.

3.3 Stage one: Elimination of nodes interior to each slab

This section describes the process that we use to eliminate the nodes interior to each slab that we sketched out in Section 3.1.2. The objective is to reduce the sparse stiffness matrix $\mathbf{A}$ (illustrated in Figure 3.1c) into the smaller block tridiagonal matrix $\mathbf{T}$ (illustrated in Figure 3.1d). The techniques described form the core algorithmic innovation of the manuscript.

3.3.1 Schur complements

With the ordering introduced in Section 3.2.2, the coefficient matrix $\mathbf{A}$ has the block form

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{A}_{23} & \mathbf{0} & \mathbf{0} & \dots \\ \mathbf{0} & \mathbf{A}_{32} & \mathbf{A}_{33} & \mathbf{A}_{34} & \mathbf{0} & \dots \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_{43} & \mathbf{A}_{44} & \mathbf{A}_{45} & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{u}_3 \\ \mathbf{u}_4 \\ \vdots \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \mathbf{f}_3 \\ \mathbf{f}_4 \\ \vdots \end{bmatrix}. \tag{3.6}$$

We eliminate the vectors $\mathbf{u}_2$, $\mathbf{u}_4$, $\mathbf{u}_6$, ... that represent unknown variables in the interior of each slab through a step of block Gaussian elimination. To be precise, we insert the relation

$$\mathbf{u}_i = \mathbf{A}_{ii}^{-1}\big(\mathbf{f}_i - \mathbf{A}_{i,i-1}\mathbf{u}_{i-1} - \mathbf{A}_{i,i+1}\mathbf{u}_{i+1}\big), \qquad i = 2,\, 4,\, 6,\, \dots \tag{3.7}$$

into the odd-numbered rows in (3.6) to obtain the reduced system

$$\begin{bmatrix} \mathbf{T}_{11} & \mathbf{T}_{13} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots \\ \mathbf{T}_{31} & \mathbf{T}_{33} & \mathbf{T}_{35} & \mathbf{0} & \mathbf{0} & \dots \\ \mathbf{0} & \mathbf{T}_{53} & \mathbf{T}_{55} & \mathbf{T}_{57} & \mathbf{0} & \dots \\ \mathbf{0} & \mathbf{0} & \mathbf{T}_{57} & \mathbf{T}_{77} & \mathbf{T}_{79} & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_3 \\ \mathbf{u}_5 \\ \mathbf{u}_7 \\ \vdots \end{bmatrix} = \begin{bmatrix} \tilde{\mathbf{f}}_1 \\ \tilde{\mathbf{f}}_3 \\ \tilde{\mathbf{f}}_5 \\ \tilde{\mathbf{f}}_7 \\ \vdots \end{bmatrix}, \tag{3.8}$$

where the sub-blocks of $\mathbf{T}$ are defined as

$$\mathbf{T}_{11} = \mathbf{A}_{11} - \mathbf{A}_{12}\ \mathbf{A}_{22}^{-1}\ \mathbf{A}_{21}, \tag{3.9}$$

$$\mathbf{T}_{13} = \mathbf{A}_{13} - \mathbf{A}_{12}\ \mathbf{A}_{22}^{-1}\ \mathbf{A}_{23}, \tag{3.10}$$

$$\mathbf{T}_{31} = \mathbf{A}_{31} - \mathbf{A}_{32}\ \mathbf{A}_{22}^{-1}\ \mathbf{A}_{23}, \tag{3.11}$$

$$\mathbf{T}_{33} = \mathbf{A}_{33} - \mathbf{A}_{32}\ \mathbf{A}_{22}^{-1}\ \mathbf{A}_{23} - \mathbf{A}_{34}\ \mathbf{A}_{44}^{-1}\ \mathbf{A}_{43}, \tag{3.12}$$

$$\mathbf{T}_{35} = \mathbf{A}_{35} - \mathbf{A}_{34}\ \mathbf{A}_{44}^{-1}\ \mathbf{A}_{23}, \tag{3.13}$$

and so on. The reduced right-hand sides $\tilde{\mathbf{f}}$ are defined as

$$\tilde{\mathbf{f}}_1 = \mathbf{f}_1 - \mathbf{A}_{12}\ \mathbf{A}_{22}^{-1}\ \mathbf{f}_2, \tag{3.14}$$

$$\tilde{\mathbf{f}}_3 = \mathbf{f}_3 - \mathbf{A}_{32}\ \mathbf{A}_{22}^{-1}\ \mathbf{f}_2 - \mathbf{A}_{34}\ \mathbf{A}_{44}^{-1}\ \mathbf{f}_4, \tag{3.15}$$

$$\tilde{\mathbf{f}}_5 = \mathbf{f}_5 - \mathbf{A}_{54}\ \mathbf{A}_{44}^{-1}\ \mathbf{f}_4 - \mathbf{A}_{56}\ \mathbf{A}_{66}^{-1}\ \mathbf{f}_6, \tag{3.16}$$

etc.

3.3.2 Rank structure in the reduced blocks

We next discuss algebraic properties of the blocks of the reduced coefficient matrix $\mathbf{T}_{ij}$ that allow for $\mathbf{T}$ to be formed efficiently. The sub-blocks of $\mathbf{T}$ are Schur-complements of sparse matrices. For instance, block $\mathbf{T}_{11}$ has the formula

$$\underset{n_2\times n_2}{\mathbf{T}_{11}} = \underset{n_2\times n_2}{\mathbf{A}_{11}} - \underset{n_2\times n_2b}{\mathbf{A}_{12}}\ \underset{n_2b\times n_2b}{\mathbf{A}_{22}^{-1}}\ \underset{n_2b\times n_2}{\mathbf{A}_{21}}\ , \tag{3.17}$$

where $\mathbf{A}_{11}, \mathbf{A}_{12}, \mathbf{A}_{21}$ are sparse with $O(n_2)$ non-zero entries and $\mathbf{A}_{22}$ is a sparse banded matrix. Factorizing $\mathbf{A}_{22}$ can be done efficiently using a sparse direct solver, but forming $\mathbf{A}_{22}^{-1}\mathbf{A}_{21}$ naively may be slow and memory-intensive, especially considering that $\mathbf{A}_{21}$ is sparse and would need to be converted to dense to interface with solve routines.

We use an alternate approach for efficiently forming $\mathbf{T}_{11}$ that achieves high arithmetic intensity while maintaining a very low memory footprint. First, we prove

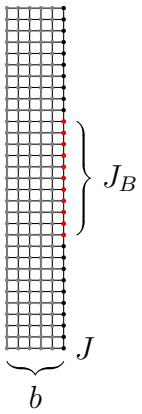


Figure 3.4: Contiguous set of points defined as $J_B \subset J$.

algebraic properties about $\mathbf{T}_{11}$, which is a dense but structured matrix that only needs $O(n_2 b)$ storage in exact precision. The matrix $\mathbf{T}_{11}$ is compressible in a format called Hierarchically Block-Separable (HBS) or Hierarchically Semi-Separable (HSS) with exact HBS rank at most $2b$. HBS matrices are a type of hierarchical matrix ($\mathcal{H}$-matrix), which allow dense matrices to be stored efficiently by exploiting low-rank structure in sub-blocks at different levels of granularity [9, 53, 68]. The matrices $\mathbf{T}_{jk}$ are compressible in several $\mathcal{H}$-matrix formats, e.g. Hierarchical Off-Diagonal Low Rank (HODLR). The rank property of $\mathbf{T}_{11}$ is formally stated in Proposition 3.3.2.

After establishing the HBS structure of $\mathbf{T}_{11}$, we then describe how this structure can be recovered with only $O(b)$ matrix vector products of $\mathbf{T}_{11}$ and $\mathbf{T}_{11}^*$. Vectors can efficiently be applied because $\mathbf{T}_{11}$ and its transpose are compositions of sparse matrices. Since $\mathbf{T}_{11}$ is admissible as a HODLR matrix, the structure can be efficiently recovered from matrix-vector products in this format as well [63, 67], though more vectors are required for reconstruction in this format.

[Rank Property] Let J_B be a contiguous set of points on the slab interface J, and let J_F be the rest of the points $J_F = J_1 \setminus J_B$. The sub-matrices $(\mathbf{T}_{11})_{BF}$, $(\mathbf{T}_{11})_{FB}$ have exact rank at most $2b$. See Figure 3.4 for an illustration of I_B. The proof is in Appendix A.1.

3.3.3 Recovering $\mathcal{H}$-matrix structure from matrix-vector products

We next describe how to extract an $\mathcal{H}$-matrix representation of the reduced blocks purely from matrix-vector products. For concreteness, we are trying to recover $\mathbf{T}_{11} \in \mathbb{R}^{n_2 \times n_2}$ as an HBS matrix with HBS rank at most $2b$. Typically, $\mathcal{H}$-matrices are used when the cost of forming or factorizing these matrices densely, is prohibitively large. In the context of SlabLU, $\mathbf{T}_{11} \in \mathbb{R}^{n_2 \times n_2}$ can be stored densely for the problem sizes of interest, but traditional methods for forming $\mathbf{T}_{11}$ densely may be inefficient, as described in Section 3.3.2. Instead, we recover $\mathbf{T}_{11}$ as an HBS matrix with HBS rank at most $2b$ from matrix-matrix products

$$\underset{n_2 \times s}{\mathbf{Y}} = \underset{n_2 \times n_2}{\mathbf{T}_{11}} \underset{n_2 \times s}{\mathbf{\Omega}}, \qquad \underset{n_2 \times s}{\mathbf{Z}} = \underset{n_2 \times n_2}{\mathbf{T}_{11}^*} \underset{n_2 \times s}{\mathbf{\Psi}}, \tag{3.18}$$

where $\mathbf{\Omega}, \mathbf{\Psi}$ are Gaussian random matrices and $s = 6b$ using the algorithm presented in [60]. This is theoretically possible because an HBS matrix of size $n_2 \times n_2$ with HBS rank at most $2b$ can be encoded in $O(n_2 b)$ storage. The HBS structure can be recovered from samples $\mathbf{Y}, \mathbf{Z}$ after post-processing in $O(n_2 b^2)$ flops. The algorithm presented in [60] can be seen as an extension of algorithms for recovering low-rank factors from random sketches [71]. A particular advantage of these algorithms is that they scale linearly and are truly black-box. The matrix-matrix products (3.18) are simple to evaluate using the matrix-free formula (3.17) of $\mathbf{T}_{11}$ and its transpose. Applying $\mathbf{T}_{11}$ involves two applications of sparse matrices and two triangular solves using pre-computed sparse triangular factors.

3.4 Stage Two: Factorizing the reduced block tridiagonal coefficient matrix

The elimination of nodes interior to each slab that we described in Section 3.3 results in a reduced linear system

$$\mathbf{T}\tilde{\mathbf{u}} = \tilde{\mathbf{f}}, \tag{3.19}$$

where $\tilde{\mathbf{u}}$ is the reduced solution vector and where $\mathbf{f}$ is the equivalent body load on slab interfaces $I_1, I_3, \ldots$ We recall that the matrix $\mathbf{T}$ is of size roughly $N/b \times N/b$, and is block tridiagonal with blocks of size $n_2 \times n_2$. Solving a system involving a block tridiagonal matrix is straight-forward using a blocked version of Gaussian elimination. In Algorithms 3 and 4 we summarize a basic scheme for solving (3.19) where we separate the process of Gaussian elimination into a "build stage" where we explicitly form and invert the Schur complements that arise in the Gaussian elimination, and a "solve stage" where the computed inverses are used to solve (3.19) for a given right hand side.

Algorithm 3 Sweeping build stage.
Given a block-tridiagonal matrix $\mathbf{T}$, Algorithm 3 builds a direct solver for $\mathbf{T}$, with the result stored in the matrices $\mathbf{S}_1, \mathbf{S}_3, \ldots$

$\mathbf{S}_1 \leftarrow \mathbf{T}_{11}$.
for $j = 2, \ldots, n_1/b$ **do**
    $\mathbf{S}_{2j+1} \leftarrow \mathbf{T}_{2j+1,2j+1} - \mathbf{T}_{2j+1,2j-1}\mathbf{S}_{2j-1}^{-1}\mathbf{T}_{2j-1,2j+1}$.
    Store $\mathbf{S}_{2j-1}^{-1}$.

Algorithm 4 Sweeping solve.
Given a body load $\tilde{\mathbf{f}}$ and precomputed inverses of $\mathbf{S}_1, \mathbf{S}_3, \ldots$, Algorithm 4 computes the solution vector $\tilde{\mathbf{u}}$.

$\mathbf{y}_1 \leftarrow \mathbf{S}_1^{-1}\tilde{\mathbf{f}}_1$.
for $j = 1, \ldots, n_1/b$ **do**
    $\mathbf{y}_{2j+1} \leftarrow \mathbf{S}_{2j+1}^{-1}\left(\tilde{\mathbf{f}}_{2j+1} - \mathbf{T}_{2j+1,2j-1}\mathbf{y}_{2j-1}\right)$.
$\tilde{\mathbf{u}}_{2n_1/b+1} \leftarrow \mathbf{S}_{2n_1/b+1}^{-1}\mathbf{y}_{2n_1/b+1}$.
for $j = n_1/b, n_1/b-1, \ldots, 1$ **do**
    $\tilde{\mathbf{u}}_{2j-1} \leftarrow \mathbf{S}_{2j-1}^{-1}\left(\mathbf{y}_{2j-1} - \mathbf{T}_{2j-1,2j+1}\tilde{\mathbf{u}}_{2j+1}\right)$.

We observe that the elimination process described in Section 3.3 automatically results in blocks that are represented in the HBS format. The most elegant way to solve (3.19) is to maintain the HBS format throughout Algorithms 3 and 4, as this leads to a solver with linear complexity in the case where the PDE is kept fixed as N increases. However, we have found that for two dimensional problems, dense linear

algebra is fast enough that the most efficient way to solve (3.19) in practice is to simply convert the HBS matrix representation to dense matrices, and carry out all computations by brute force. (In the 3D version of SlabLU, rank structure must be exploited, however.)

Remark 3.2. In the case where (3.1) has oscillatory solutions, and the number of points per wavelength is kept fixed as N increases, it is still possible to exploit rank structure in the blocks of $\mathbf{T}$. However, the ranks will grow during the execution of Algorithm 3. To minimize rank-growth, an odd-even elimination order can be used instead of the sequential one in Algorithm 3, but the method would still not attain linear complexity We observe that our conversion to dense linear algebra allows us to side-step this complication, as our method relies on the sparsity pattern of the original matrix only.

3.5 Algorithm and complexity costs

In this section, we provide a summary of the proposed algorithm, and discuss its efficient parallelization using batched linear algebra on a GPU. We also analyze the complexity costs, and choose the buffer size b as a function of the number of grid points $N = n_1 n_2$, in order to balance costs and achieve competitive complexities for the build and solve times. While our discussion on choosing b focuses on the 5-point stencil model problem, the conclusions are generally applicable to other discretizations.

We briefly summarize the algorithm for SlabLU. In Stage One, we compute factorizations of the form

$$\underset{n_2 b \times n_2 b}{\mathbf{A}_{22}^{-1}}, \ \underset{n_2 b \times n_2 b}{\mathbf{A}_{44}^{-1}}, \ldots \tag{3.20}$$

for n_1/b sparse matrices. The reduced matrix $\mathbf{T}$ is constructed using efficient black-box algorithms that recover $\mathbf{T}_{jk}$ in HBS format through a random sampling method, as discussed in Section 3.3.3. It is important to note that Stage One can be trivially parallelized for each slab. In Stage Two, dense linear algebra is used to factorize $\mathbf{T}$ as discussed in Section 3.4.

Algorithm 5 outlines how to use the computed factorization of $\mathbf{A}$ to solve systems (3.3). This requires storing the sparse factorizations (3.20) and the factorization of $\mathbf{T}$.

Algorithm 5 Solving $\mathbf{Au} = \mathbf{f}$ using computed direct solver.

1: Calculate the equivalent body $\tilde{\mathbf{f}}$ on $I_1, I_3, \ldots$ using (3.14-3.16) in parallel.
2: Solve $\mathbf{T}\tilde{\mathbf{u}} = \tilde{\mathbf{f}}$ for $\tilde{\mathbf{u}}$ on $I_1, I_3, \ldots$ using serial Algorithm 4.
3: Solve for $\mathbf{u}$ on $I_2, I_4, \ldots$ using $\tilde{\mathbf{u}}$ on $I_1, I_3, \ldots$ with (3.7) in parallel.

3.5.1 Ease of Parallelism and Acceleration with Batched Linear Algebra

We highlight the features of the SlabLU scheme that make it particularly suitable for implementation on GPUs and compare it to multi-level nested dissection schemes. Although nested dissection is work-optimal for minimizing the fill-in of the computed factorization, parallelizing it for high performance on GPUs remains an active area of research. Recent results include the GPU-accelerated multifrontal algorithm proposed in [42], which splits the problem into independent sub-problems that can be factorized entirely in GPU memory. The factorization is then performed using a level-by-level traversal with custom hand-coded kernels for small matrices and vendor-optimized BLAS libraries for larger matrices.

However, there are significant challenges and avenues for improvement. Efficient nested dissection codes require parallel operations on a wide range of matrix sizes, and fully exploiting the potential of modern GPUs would require careful load balancing and memory access patterns [91]. Sparse direct solvers generally do not make use of batched linear algebra, a highly optimized framework for parallelized BLAS operations on many small matrices, because the framework is limited to matrices of the same size, and variable-sized batching is often needed in the context of a sparse direct solver. Although an extension to variable sizes is currently under development [2], it is far less mature than its constant-sized counterpart.

In contrast, the SlabLU scheme takes advantage of the two-level structure to

simplify the optimization process significantly. The trade-off is a small asymptotic increase in the flop and memory complexity of the algorithm, but there are large benefits in terms of practical performance on GPUs. SlabLU only requires factorizing two front sizes: b in Stage One and n_2 in Stage Two. Since the scheme has only two levels, it is relatively easy to optimize the data movement and coordination between the GPU, which has limited memory, and the CPU.

In Stage One, parallel operations are performed on many small matrices, while in Stage Two, serial operations are performed on a relatively small number of larger matrices. Our implementation makes use of highly optimized libraries (e.g., MAGMA for batched linear algebra on small matrices [28, 29] and vendor-optimized BLAS for large matrices [78]), and the high practical performance required little, if any, hand tuning. Stage One also provides an opportunity to incorporate highly optimized batched linear algebra schemes by using matrix-free randomized algorithms to recover $\mathbf{T}_{jk}$ in HBS format, as we do in our implementation. $\mathcal{H}$-matrix algebras can be efficiently parallelized using batched linear algebra because the off-diagonal rank is small and uniform on each level of the $\mathcal{H}$-matrix structure [13, 19].

Overall, the SlabLU scheme offers a more straightforward path to optimizing sparse direct solvers on modern GPUs than traditional nested dissection codes. While parallelizing nested dissection codes remains an active area of research, the simplicity of SlabLU makes it an attractive option for efficient parallelization of sparse direct solvers.

3.5.2 Choosing the buffer size b.

The buffer size is chosen to balance the costs of Stage One and Stage Two. Stage One involves computing sparse factorizations (3.20), where each sparse matrix is banded with bandwidth b. This process requires $\frac{n_1}{b} \times O\left(b^3 n_2\right)$ flops. The cost of constructing $\mathbf{T}_{jk}$ in HBS format is dominated by the cost of constructing the random samples (3.18), which requires $O\left(n_2 b^3\right)$ flops using matrix-free formulas for

applying $\mathbf{T}_{jk}$. The cost of post-processing random samples is asymptotically small. To summarize, Stage One is dominated by the costs of the factorizations of sparse matrices (3.20) and constructing random samples (3.18), which both have costs $\frac{n_1}{b} \times O\left(n_2 b^4\right) = O\left(b^3 N\right)$. Importantly, Stage One can be done in parallel for n_1/b subdomains.

Stage Two involves a simple sweeping factorization of $\mathbf{T}$ using Algorithm 3. The reduced matrix $\mathbf{T}$ has $\approx n_1/b$ blocks, each of size $n_2 \times n_2$. The total cost of constructing the factorization is

$$T_{\text{build}} = \underset{\text{Stage One}}{O\left(b^2 N\right)} + \underset{\text{Stage Two}}{O\left(\frac{1}{b} n_1 n_2^3\right)}. \tag{3.21}$$

To balance the costs of Stage One and Stage Two, we choose $b = O\left(n_2^{2/3}\right)$. The total cost of the build stage is $T_{\text{build}} = O\left(n_1 n_2^{7/3}\right)$.

To apply the direct solver for $\mathbf{A}$, one needs to store the sparse factorizations (3.20) and the direct solver for $\mathbf{T}$. The sparse factorizations (3.20) require $O\left(\frac{n_1}{b} \times b^2 n_2\right)$ storage, and storing the direct solver for $\mathbf{T}$ requires $O\left(\frac{n_1}{b} \times n_2^2\right)$ storage. With the choice of $b = O\left(n_2^{2/3}\right)$, the sparse factorizations require asymptotically more space to store. We can rebalance the memory costs of Stage One and Stage Two by using a nested dissection ordering for the sparse factorization within each slab or by "discarding" some of the factorization in Stage One (e.g. corresponding to small subdomains) and re-factorizing as needed to apply Algorithm 5. Then the memory costs are

$$T_{\text{solve}} = \frac{n_1}{b} \times O\left(n_2^2\right) = O\left(n_1 n_2^{4/3}\right). \tag{3.22}$$

Our implementation also exploits rank-structures (e.g. in the off-diagonal sub-blocks of $\mathbf{T}$) to efficiently store the computed direct solver.

The algorithm scales particularly well for domains with high aspect ratio when $n_1 > n_2$. A square domain is the adversarial worst case in terms of the algorithm

complexity. When $n_1 = n_2$, then

$$T_{\text{build}} = O\left(N^{5/3}\right), \qquad T_{\text{solve}} = O\left(N^{7/6}\right). \tag{3.23}$$

3.5.3 Complexity Analysis for SlabLU with HPS discretization

The numerical results feature a high order discretization scheme that interfaces particularly well with sparse direct solvers which we use to resolve high frequency scattering problems to high accuracy. As discussed in Section 3.2.3, HPS is a spectral collocation discretization that employs multiple subdomains to enforce the PDE using spectral differentiation. The subdomains are coupled together by ensuring continuity of the solution and its derivative across subdomains. A natural approach to factorizing the sparse coefficient matrix is to first factorize each leaf subdomain in parallel and then solve the remaining sparse matrix $\tilde{\mathbf{A}}$. The leaf operations require independent dense linear algebraic operations (e.g., LU factorization, matrix-matrix multiply) on N/p^2 systems, each of size $p^2 \times p^2$, resulting in an overall cost of $O(p^4 N)$.

The total cost of factorizing a sparse matrix arising from the HPS discretization can be expressed as

$$T_{\text{build}} = O\left(\underset{\text{leaf operations}}{p^4 N} + \underset{\text{direct solver}}{N^{5/3}}\right), \tag{3.24}$$

where N is the total number of unknowns and p is the polynomial degree used in the spectral collocation method. After the static condensation step, it is important to note that the cost of factorizing $\tilde{\mathbf{A}}$ has no pre-factor dependence on p.

The leaf operations have a large pre-factor cost with p, and have long been viewed as prohibitively expensive for large p. We make these operations efficient using batched linear algebra and GPU acceleration, demonstrating compelling results for p up to 42 in Section 3.6. The numerical results also demonstrate that the choice of p does not have substantial effects on the build time for the direct factorization stage, allowing p to be chosen based on physical considerations instead of practical concerns.

The technique we present is most readily applicable to the case where the same discretization order p is used on every discretization patch. However, it would not be too difficult to allow p to be chosen from a fixed set of values (say $p \in \{6, 10, 18, 36\}$ or something similar). This would enable many of the advantages of hp-adaptivity, while still enabling batching to accelerate computations. For larger p, methods that produce a sparser equivalent system may be more appropriate [14, 36].

Remark 3.3. The leaf operations are so efficient that we can save on storage costs by not explicitly storing the factorizations of the local spectral differentiation matrices in each HPS subdomain. Instead, we can reform and refactor these matrices as needed, and these costs are included in the solve time reported in Section 3.6.3.

3.6 Numerical experiments

In this section, we demonstrate the effectiveness of our solver through a series of numerical experiments. We report the build time, solve time, and accuracy of solving constant and variable-coefficient elliptic PDEs using two collocation-based discretization schemes on both rectangular and curved geometries. Our experiments were conducted on various hardware architectures to showcase the portability and ease of performance tuning of our framework.

Our experiments compare SlabLU to SuperLU in the factorization of a PDE discretized with 2nd order finite differences. The results demonstrate that SlabLU achieves a significant speedup compared to traditional nested dissection schemes, even without GPU acceleration.

Additionally, we utilize a high-order multidomain spectral collocation scheme, briefly introduced in Section 3.2.3, to solve challenging scattering phenomena. The high-order discretization scheme allows us to accurately discretize the PDE, while the flexibility of the multidomain scheme enables us to solve PDEs on curved domains using SlabLU. The combination of SlabLU and high-order discretization provides a powerful tool for simulating electromagnetic and acoustic scattering. Our focus is

on physical phenomena that may be beyond the reach of preconditioned iterative methods.

3.6.1 Description of Benchmark PDEs and Accuracies Reported

We briefly describe the PDEs with manufactured solutions used as benchmarks in our numerical experiments and how we calculate accuracy. The first PDE is the Laplace equation

$$\begin{cases} -\Delta u(x) = 0, & x \in \Omega, \\ u(x) = u_{\text{true}}(x), & x \in \Gamma, \end{cases} \tag{3.25}$$

The Dirichlet data is the restriction of the true analytic solution to the boundary

$$u_{\text{true}}(x) = \log\left(\|x - (-0.1, 0.5)\|\right). \tag{3.26}$$

The second PDE is a constant coefficient Helmholtz problem

$$\begin{cases} -\Delta u(x) - \kappa^2 u(x) = 0, & x \in \Omega, \\ u(x) = u_{\text{true}}(x), & x \in \Gamma, \end{cases} \tag{3.27}$$

where the true solution u_{true} is given by

$$u_{\text{true}} = J_0\left(\kappa\|x - (-0.1, 0.5)\|\right), \tag{3.28}$$

where $x \mapsto J_0(\kappa|x|)$ is the zeroth Bessel function of the first kind.

After applying the direct solver, we obtain the calculated solution $\mathbf{u}_{\text{calc}}$ at discretization points within the domain. We report the relative error with respect to the residual of the discretized system (3.3) and with respect to the true solution $\mathbf{u}_{\text{true}}$ evaluated at the collocation points as follows:

$$\text{relerr}_{\text{res}} = \frac{\|\mathbf{A}\mathbf{u}_{\text{calc}} - \mathbf{f}\|_2}{\|\mathbf{f}\|_2}, \qquad \text{relerr}_{\text{true}} = \frac{\|\mathbf{u}_{\text{calc}} - \mathbf{u}_{\text{true}}\|_2}{\|\mathbf{u}_{\text{true}}\|_2}. \tag{3.29}$$

We also report T_{factor}, which is the wall-clock time needed to factorize the coefficient matrix $\mathbf{A}$, and M_{factor}, which is the memory required to store the computed direct solver. Additionally, we report T_{solve}, which is the time needed to apply the direct solver to solve systems (3.3), as described in Algorithm 5. Unless otherwise stated, the solve is done on the entirely on the CPU for a single right hand side vector.

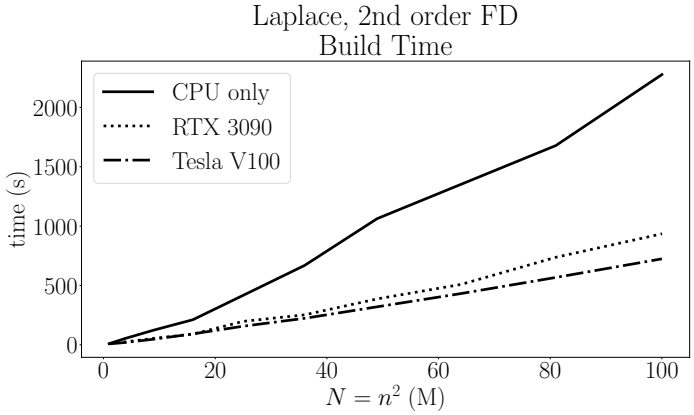
3.6.2 Benchmark Experiments using Low-Order Discretization

In this section, we demonstrate how SlabLU performs on sparse coefficient matrices arising from PDEs discretized with 2nd order finite differences. We also compare SlabLU to SuperLU, an existing multi-level nested dissection code. We conducted the experiments on three different architectures: (1) a 16-core Intel i9-12900k CPU with 128 GB of RAM, (2) an NVIDIA RTX 3090 GPU with 24 GB of memory and access to 128 GB of RAM, and (3) an NVIDIA V100 with 32 GB of memory and access to 768 GB of RAM. We chose to run experiments on architectures (1) and (2) to demonstrate that the memory volume required to run SlabLU is reasonable.

All experiments used double precision. While the RTX 3090 supports double precision calculations, its peak performance using fp64 is significantly less than that of fp32. In comparison, the V100 is designed for high performance with fp64. Surprisingly, we found that the timings on the two systems did not differ significantly. We observed significant speedups for Stage Two on the V100 compared to the RTX 3090, but little change for Stage One, leading us to conclude that the factorization of $\mathbf{T}$ is compute-bound, whereas the sparse factorizations in Stage One are latency-bound.

We demonstrate competitive scaling for the build time of the factorization and for the memory footprint. See Figure 3.5 for the Poisson equation (3.25) and Figure 3.6 for the constant-coefficient Helmholtz equation (3.27). Despite the super-linear complexity scaling, the scaling appears to be linear for grids of size up to $N = 100$M. However, for coercive PDEs, there is rarely a reason to scale the discretizations to such large grids. For the Helmholtz equation, we discretize to at least 10 points per wavelength to resolve the oscillatory solutions, leading to large grids. However, due to the effect of pollution when using low-order discretization, we need to discretize the Helmholtz equation to 250 points per wavelength to attain 3 digits of accuracy with respect to the known analytic solution.

Sparse direct solvers use sparsity in the discretized operator in order to factorize the sparse coefficient matrix exactly. SlabLU uses sparsity in the traditional



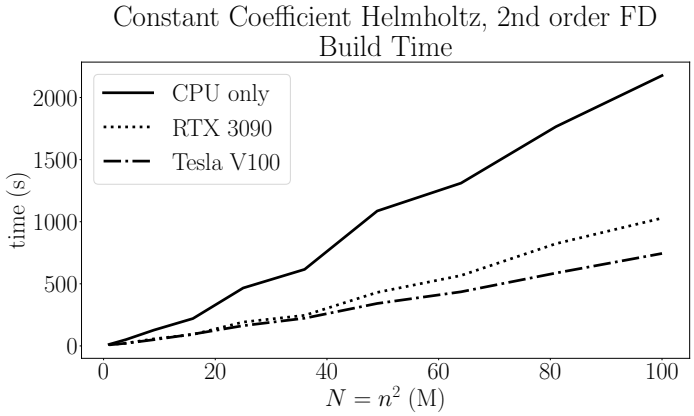
(a) Build time for SlabLU on various architectures.

N	b	M_{factor}	T_{solve}	$\text{relerr}_{\text{res}}$	$\text{relerr}_{\text{true}}$
1.0 M	50	0.5 GB	0.5 s	1.99e-12	5.22e-07
4.0 M	100	2.5 GB	1.6 s	3.41e-12	1.31e-07
9.0 M	125	6.5 GB	3.6 s	4.78e-12	5.81e-08
16.0 M	160	12.9 GB	7.3 s	4.15e-12	3.27e-08
25.0 M	200	22.5 GB	12.2 s	4.66e-12	2.09e-08
36.0 M	200	32.7 GB	18.2 s	5.90e-12	1.46e-08
49.0 M	200	46.8 GB	27.4 s	1.10e-11	1.04e-08
64.0 M	250	63.9 GB	37.7 s	6.08e-12	8.10e-09
81.0 M	250	83.3 GB	55.3 s	6.16e-12	6.32e-09
100.0 M	250	105.6 GB	65.4 s	7.96e-12	4.92e-09

(b) Memory footprint, solve time, and solution accuracy using SlabLU.

Figure 3.5: The plot in Figure 3.5a and corresponding table in Table 3.5b for solving the Poisson equation with a body load (3.25) using a 2nd order finite difference discretization. SlabLU is portable across a variety of architectures, which is illustrated by reporting results using the CPU only, as well as two different GPU architectures. Though the build time of SlabLU scales as $O(N^{5/3})$, the observed asymptotic behavior has linear scaling with N; using GPU acceleration substantially accelerates the build time.

sense for the sparse factorizations (3.20) in Stage One. In order to construct the reduced coefficient matrix $\mathbf{T}$, we prove a rank property in Proposition (3.3.2) that allows us to represent sub-blocks $\mathbf{T}_{jk} \in \mathbb{R}^{n_2 \times n_2}$ *exactly* in $O(n_2 b)$ storage and recover $\mathbf{T}_{jk}$ in HBS format using random sampling; see Section 3.3.3. Crucially, Proposition (3.3.2) is a purely algebraic property that holds regardless of the PDE. Often, the HBS rank of $\mathbf{T}_{jk}$ is considerably less than $2b$; SlabLU uses adaptive rank sampling in the HBS construction to considerably save on build time and storage costs. For a thin slab with λ wavelengths in the thin dimension, we have observed that the ranks



(a) Build time for SlabLU on various architectures.

N	κ	M_{factor}	T_{solve}	$\text{relerr}_{\text{res}}$	$\text{relerr}_{\text{true}}$
1.0 M	27.12	0.5 GB	0.5 s	1.00e-11	1.79e-03
4.0 M	52.25	2.5 GB	1.4 s	1.58e-11	2.59e-03
9.0 M	77.39	6.6 GB	3.5 s	2.64e-11	3.95e-03
16.0 M	102.52	12.9 GB	7.2 s	2.27e-11	1.01e-02
25.0 M	127.65	22.5 GB	10.5 s	2.41e-11	1.01e-02
36.0 M	152.78	32.3 GB	17.8 s	2.86e-11	4.84e-03
49.0 M	177.92	46.9 GB	28.3 s	4.12e-11	6.50e-03
64.0 M	203.05	64.0 GB	37.6 s	1.89e-10	3.15e-02
81.0 M	228.18	83.8 GB	61.0 s	1.46e-10	1.09e-02
100.0 M	253.32	105.7 GB	70.8 s	1.04e-10	8.70e-03

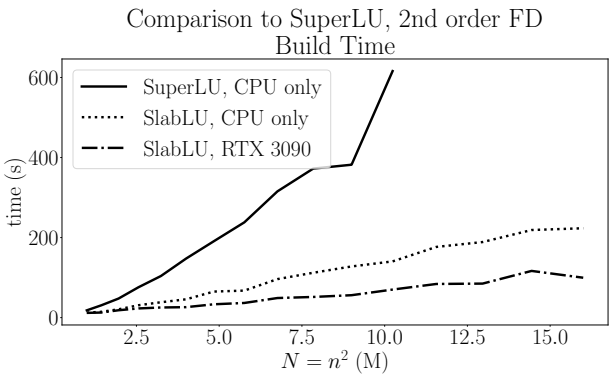
(b) Memory footprint, solve time, and solution accuracy using SlabLU.

Figure 3.6: The plot in Figure 3.6a and corresponding table in Table 3.6b report results for solving the constant coefficient Helmholtz equation (3.27) using a 2nd order finite difference discretization. The wavenumber κ is increased with the problem size to maintain 250 points per wavelength. Although we use some approximations for SlabLU, the factorization is able to resolve the solution is resolved to at least 10 digits in the residual. Because of the effect of pollution, the relative error compared to the true solution is only 3 digits.

are roughly twice λ plus a small constant factor. For the Poisson equation, HBS rank about 50 approximates $\mathbf{T}_{jk}$ to high accuracy.

We compare the performance of SlabLU and SuperLU in solving the 2nd order finite difference discretization of the constant coefficient Helmholtz equation (3.27), which results in very ill-conditioned sparse matrices (3.3) that need to be solved.

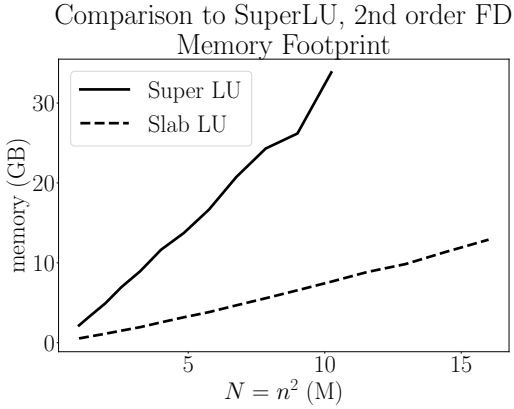
SuperLU is a generic sparse direct solver that computes a sparse LU factorization of any given sparse matrix to high accuracy. It uses groupings of nodes into "supernodes" to leverage BLAS3 operations on dense sub-blocks [11, 61] and



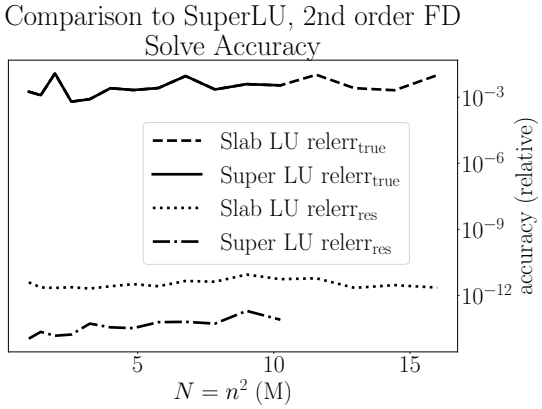
(a) Comparison of build time for SuperLU and SlabLU.

		T_{solve} (CPU only)	
N	κ	SlabLU	SuperLU
1.0 M	27.12	0.21	0.46
1.4 M	27.12	0.31	0.51
2.0 M	33.40	0.43	0.69
2.6 M	39.69	0.58	1.03
3.2 M	45.97	0.74	1.27
4.0 M	52.25	0.69	1.38
4.8 M	52.25	1.16	1.92
5.8 M	58.54	1.10	2.25
6.8 M	64.82	1.69	2.48
7.8 M	71.10	1.96	3.21
9.0 M	77.39	1.63	3.47
10.2 M	77.39	1.97	3.84
11.6 M	83.67	—	4.78

(b) Comparison of solve time for SuperLU and SlabLU.



(c) Comparison of memory footprint for SuperLU and SlabLU.



(d) Comparison of computed solution accuracy for SuperLU and SlabLU.

Figure 3.7: Figures 3.7a to 3.7d report a comparison between Super LU and SlabLU for equation 3.27 discretized with a constant number of points per wavelength using 2d order finite differences. SlabLU groups nodes into supernodes to leverage batched BLAS3 operations in Stage One and uses GPU acceleration for the large fronts in Stage Two. For the CPU-comparison, SlabLU is faster by a factor of 4 for N=10.2M. Using GPU acceleration makes the method faster by a factor of 8. Additionally, SlabLU is more memory efficient by a factor of 4 for $N = 10.2$ M.

is computed with the default permutation specification of COLAMD ordering [4] to minimize fill-in. We use the Scipy interface (version 1.8.1) to call SuperLU. During the factorization, SuperLU may pivot between sub-blocks to achieve stability in the computed factorization, resulting in superior accuracy in the residual. Despite limita-

tions in the pivoting scheme of SlabLU, both schemes are able to resolve the solution up to the discretization error (see Figure 3.7). However, our comparison shows that SlabLU outperforms SuperLU in terms of build times and memory costs up to about 10M points. It is somewhat surprising that SlabLU outperforms SuperLU in terms of memory costs; we believe this is because SuperLU stores its computed factorization in sparse CSR format, leading to large storage costs for dense sub-blocks. For problems larger than 10.2M points, SuperLU does not compute the factorization, likely because the memory requirements exceed some pre-prescribed limit.

3.6.3 Benchmark Experiments using High-Order Discretization

High-order discretization is crucial for resolving variable-coefficient scattering phenomena due to the pollution effect, which requires increasing the number of points per wavelength as the domain size increases [10, 26]. The HPS discretization (cf. Section 3.2.3) is less sensitive to pollution because it allows for a high choice of local polynomial order [44, 66].

In this subsection, we demonstrate the ability of HPS to resolve oscillatory problems without the effect of pollution by reporting results for the constant coefficient Helmholtz equation (3.27). For these experiments, the wavenumber κ increases with N to maintain 10 points per wavelength. We also demonstrate performance of SlabLU in factorizing sparse systems arising from the HPS discretization with local polynomial order p. The leaf operations are handled efficiently as discussed in 3.5.3, and factorizing the reduced system $\tilde{\mathbf{A}}$ has no pre-factor dependence on p. In many plots (e.g. Figures 3.8, 3.9) results are reported for a range of local polynomial order p. To keep N fixed for various p, we choose the number of local HPS subdomains appropriately (e.g. more subdomains for $p = 17$ and fewer for $p = 42$).

First, we report performance of the leaf operations on their own. These are handled efficiently using batched linear algebra, and the local leaf factorizations are discarded and re-factorized as needed during the solve stage to save on memory costs

for the direct solver. The efficiency of the leaf operations for fixed N and varying p is shown in Figure 3.8. Parallel leaf operations are particularly efficient when using GPU acceleration.

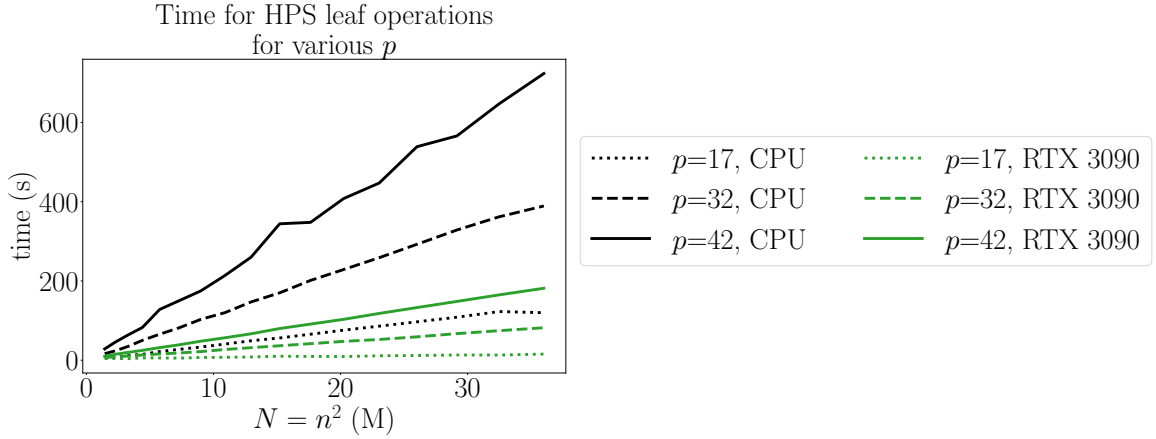


Figure 3.8: The plot shows the time required for leaf operations for HPS, for various local polynomial orders p and fixed total degrees of freedom N. Leaf operations for HPS require $O(p^4N)$ operations, though the practical scaling for parallel operations has a small constant prefactor for p up to 32. Parallel HPS leaf operations are further accelerated on the GPU. The speedup factor is at least 7.7x, 4.7x, and 3.9x for $p = 17, 32$, and 42, respectively.

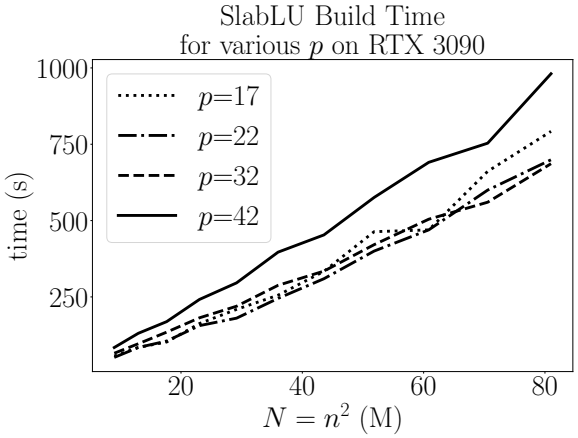
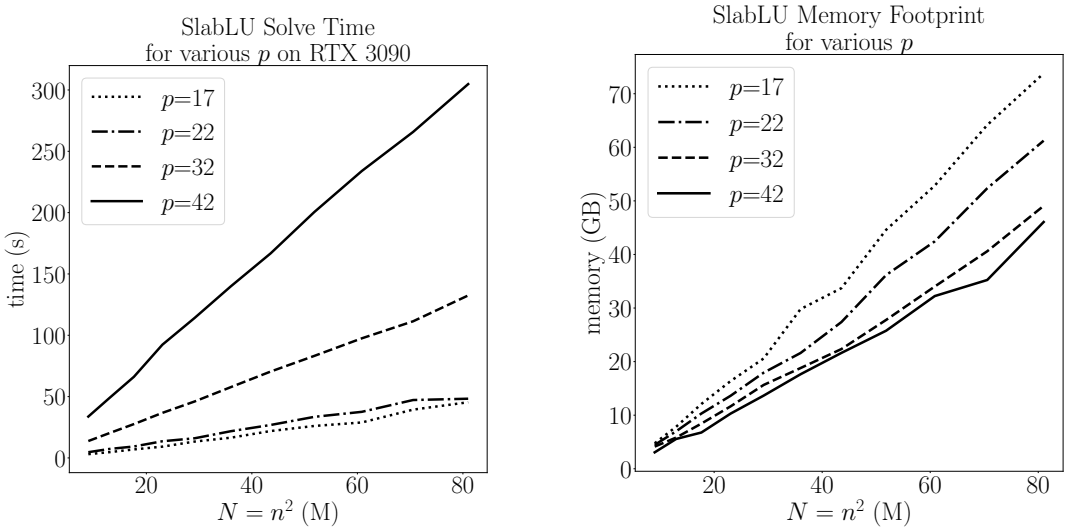


Figure 3.9: We discretize equation (3.27) with HPS for various local polynomial order p. Refinement in the local polynomial order p allows the user to attain faster convergence in the relative error, compared to the true solution. For many discretizations, increasing p may lead to substantially slower factorization time. Because HPS interfaces well with sparse direct solvers, the choice of p does not substantially affect the build time for SlabLU.



(a) Solve time for various p.

(b) Memory footprint for various p.

Figure 3.10: Figures 3.10a and 3.10b show the solve time and memory footprint, respectively for equation (3.27) discretized with HPS for various local polynomial order p. Recall in Remark 3.3, we save on space by discarding local leaf differentiation operators and reforming them as needed for the solve stage. Because the leaf operations are efficient on the GPU, the solve stage is fairly fast, even for large p.

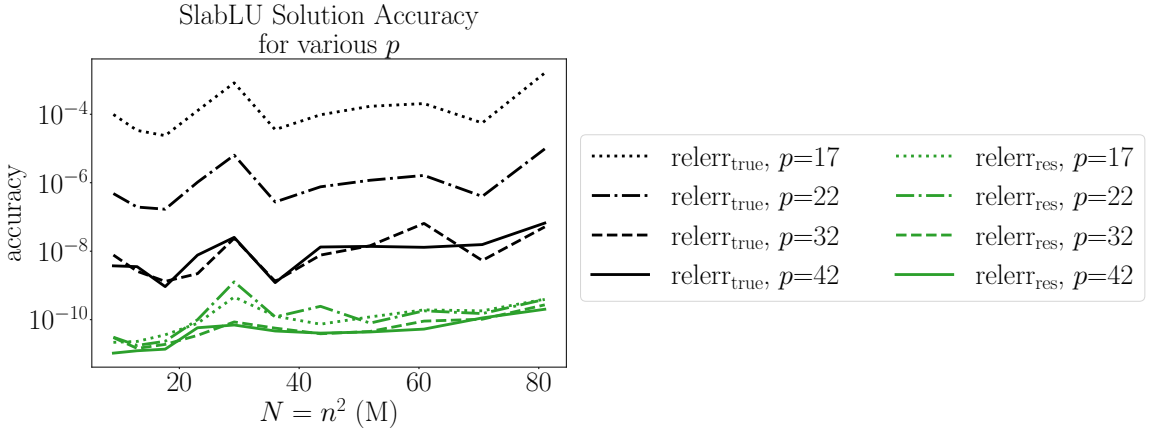
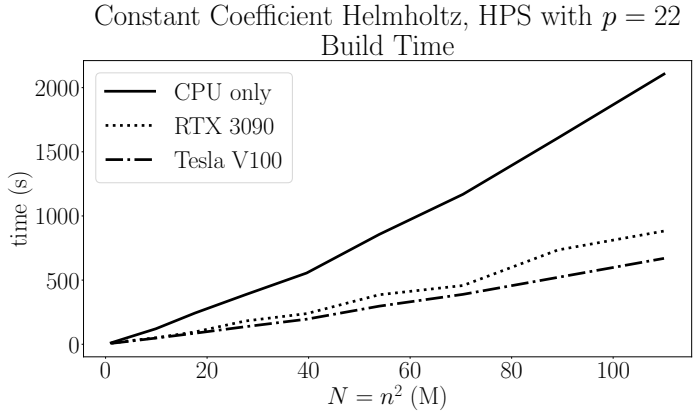


Figure 3.11: We discretize equation (3.27) with HPS for various local polynomial order p. Recall that the wavenumber κ is increased with N to maintain 10 points per wavelength. Regardless of the choice of p, SlabLU resolves the solution to at least 10 digits of relative accuracy in the residual (i.e. in the quantity relerr$_{\text{res}}$). With increasing p, one can calculate solutions with higher relative accuracy, compared to the true solution of the PDE, which we refer to as relerr$_{\text{true}}$.

Next, we report the performance of SlabLU in factorizing systems discretized with HPS for various polynomial orders p. The flexibility of the HPS discretization allows for p to be changed easily, depending on the user's preference. Because the leaf operations are efficient and HPS interfaces well with SlabLU, the build and solve time are not substantially affected by changing p, c.f. Figures 3.9 and 3.10. Higher choices of p lead to better resolution of the computed solution, compared to the true known solution, as reported in Figure 3.11.



(a) Build time for SlabLU on various architectures.

N	κ	M_{build}	T_{solve}	$\text{relerr}_{\text{res}}$	$\text{relerr}_{\text{true}}$
1.1 M	630.31	0.4 GB	0.8 s	3.54e-12	2.37e-08
4.4 M	1258.63	1.8 GB	2.3 s	6.29e-12	9.57e-08
9.9 M	1886.94	4.4 GB	4.7 s	1.14e-11	2.16e-07
17.6 M	2515.26	8.9 GB	8.1 s	2.39e-11	4.03e-07
27.5 M	3143.58	15.6 GB	13.9 s	6.44e-11	9.66e-07
39.6 M	3771.90	21.5 GB	21.6 s	3.87e-10	7.11e-07
53.9 M	4400.22	32.4 GB	29.7 s	6.23e-11	7.92e-07
70.4 M	5028.54	45.9 GB	39.0 s	1.24e-09	4.56e-07
89.1 M	5656.86	61.0 GB	52.5 s	4.33e-10	1.19e-06
110.0 M	6285.17	80.0 GB	68.1 s	2.33e-10	7.49e-07

(b) Memory footprint, solve time, and solution accuracy using SlabLU.

Figure 3.12: The plot in Figure 3.12a and corresponding table in Table 3.12b report results for solving the constant coefficient Helmholtz equation (3.27) using an HPS discretization with $p = 22$, where the wavenumber κ is increased with the problem size to maintain 10 points per wavelength. Using a high order discretization scheme allows us to scale to physically large domains up to $1000\lambda \times 1000\lambda$ without the effect of pollution. The leaf operations for HPS and SlabLU are portable across a variety of hardware architectures and perform especially well on GPUs.

Finally, we show results for the Helmholtz equation discretized with HPS for

$p = 22$; the wavenumber κ is increased with N to maintain 10 points per wavelength. We scale up to $N = 110$M points on a variety of hardware architectures, c.f. Figure 3.12. The largest domain is of size $1000\lambda \times 1000\lambda$ and is resolved 7 digits of relative accuracy, compared to the known solution of the PDE. Meanwhile, 2nd order FD requires 100-250 points per wavelength to achieve low accuracy, c.f. 3.6.

3.6.4 Solving Challenging Scattering Problems with High Order Discretization

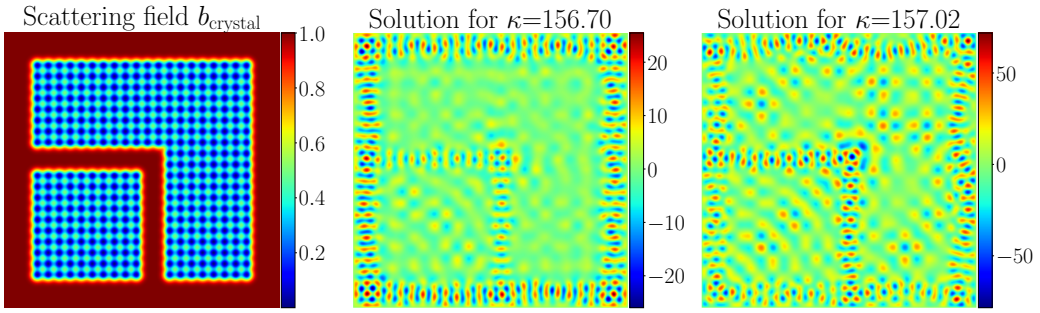
We will now demonstrate the ability of HPS, combined with SlabLU as a sparse direct solver, to solve complex scattering phenomena on various 2D domains. Combining HPS with SlabLU provides a powerful tool for resolving challenging scattering phenomena to high accuracy, especially for situations where efficient preconditioners are not available [34].

For the presented PDEs, we will show how the accuracy of the calculated solution converges to a reference solution depending on the choice of p in the discretization. Specifically, we will solve the BVP (3.1) with the variable-coefficient Helmholtz operator (3.2) for Dirichlet data on curved and rectangular domains. We fix the PDE and refine the mesh to compare calculated solutions to a reference solution obtained on a fine mesh with high p, as the exact solution is unknown. The relative error is calculated by comparing $\mathbf{u}_{\text{calc}}$ to the reference solution $\mathbf{u}_{\text{ref}}$ at a small number of collocation points $\{x_j\}_{j=1}^{M}$ using the l_2 norm

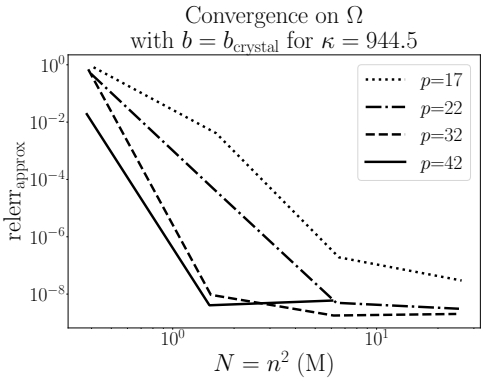
$$\text{relerr}_{\text{approx}} = \frac{\|\mathbf{u}_{\text{calc}} - \mathbf{u}_{\text{ref}}\|_2}{\|\mathbf{u}_{\text{ref}}\|_2}. \tag{3.30}$$

We demonstrate the convergence on a unit square domain $\Omega = [0, 1]^2$ with a variable coefficient field b_{crystal} corresponding to a photonic crystal, shown in Figure 3.13a. The solutions and convergence plot are presented in Figure 3.13.

Next, we show the convergence on a curved domain Φ with a constant-coefficient field $b \equiv 1$, where Φ is given by an analytic parametrization over a reference square



(a) Computed solutions on Ω.



(b) Convergence of computed solutions on Ω.

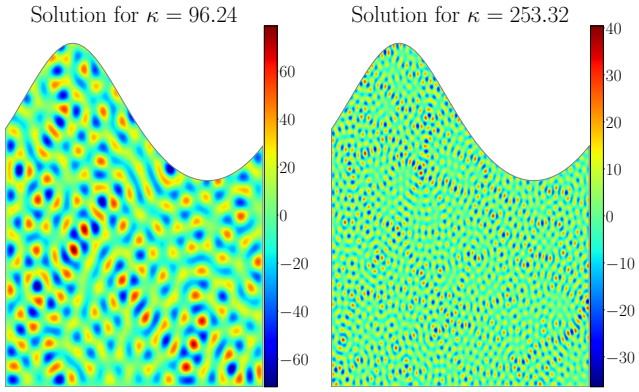
Figure 3.13: Figure 3.13a shows solutions of variable-coefficient Helmholtz problem on square domain Ω with Dirichlet data given by $u \equiv 1$ on $\partial\Omega$ for various wavenumbers κ. The scattering field is b_{crystal}, which is a photonic crystal with an extruded corner waveguide. The crystal is represented as a series of narrow Gaussian bumps with separation $s = 0.04$ and is designed to filter wave frequencies that are roughly $1/s$. Figure 3.13b shows the convergence of computed solutions, for reference solution $\mathbf{u}_{\text{ref}}$ on HPS discretization for N=36M with $p = 42$.

$\Omega = [0,1]^2$. The domain Φ is parameterized as

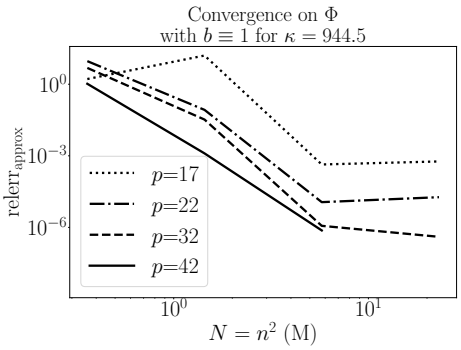
$$\Psi = \left\{ \left(x_1, \frac{x_2}{\psi(x_1)} \right) \text{ for } (x_1, x_2) \in \Omega = [0,1]^2 \right\}, \text{ where } \psi(z) = 1 - \frac{1}{4}\sin(z). \quad (3.31)$$

Using the chain rule, (3.2) on Φ takes the following form on Ω

$$\begin{aligned} -\frac{\partial^2 u}{\partial x_1^2} - 2\frac{\psi'(x_1)x_2}{\psi(x_1)}\frac{\partial^2 u}{\partial x_1 \partial x_2} - \left(\left(\frac{\psi'(x_1)x_2}{\psi(x_1)} \right)^2 + \psi(x_1)^2 \right) \frac{\partial^2 u}{\partial x_2^2} \\ - \frac{\psi''(x_1)x_2}{\psi(x_1)}\frac{\partial u}{\partial x_2} - \kappa^2 u = 0, \qquad (x_1, x_2) \in \Omega. \end{aligned} \quad (3.32)$$

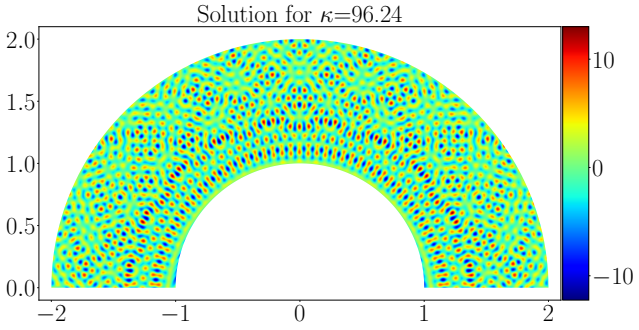


(a) Computed solution on Ψ.

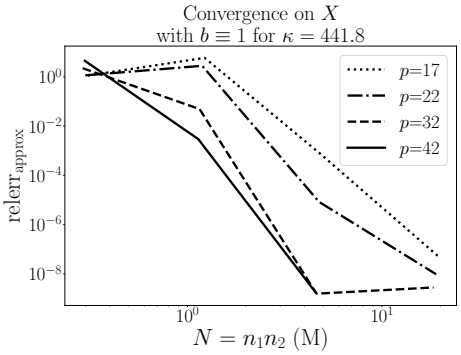


(b) Convergence of computed solutions on Ψ.

Figure 3.14: Figure 3.14a shows solutions of constant-coefficient Helmholtz problem on curved domain Ψ with Dirichlet data given by $u \equiv 1$ on $\partial\Psi$ for various wavenumbers κ. The solutions are calculated parameterizing Ψ in terms of a reference square domain Ω as (3.31) and solving (3.32) on Ω. Figure 3.14b shows convergence on curved domain Ψ for reference solution $\mathbf{u}_{\text{ref}}$ on HPS discretization for N=36M with $p = 42$. The solutions exhibit mildly singular behavior near the corners, and choosing high orders of p aids in the convergence.



(a) Computed solutions on X.



(b) Convergence of computed solutions on X.

Figure 3.15: Figure 3.15a shows solutions of constant-coefficient Helmholtz problem on curved domain X with Dirichlet data given by $u \equiv 1$ on $\partial\Psi$. The solutions are calculated parameterizing X in terms of a reference rectangle domain as (3.33) and solving a variable-coefficient elliptic PDE on the reference rectangle. Figure 3.15b shows the convergence on curved domain X for reference solution $\mathbf{u}_{\text{ref}}$ on HPS discretization for N=36M with $p = 42$.

The solutions on Φ and the convergence plot is presented in Figure 3.14.

Next, we show the convergence on a curved domain X with a constant-coefficient field $b \equiv 1$, where X is a half-annulus given by an analytic parametrization over a reference rectangle. The domain X is parameterized as

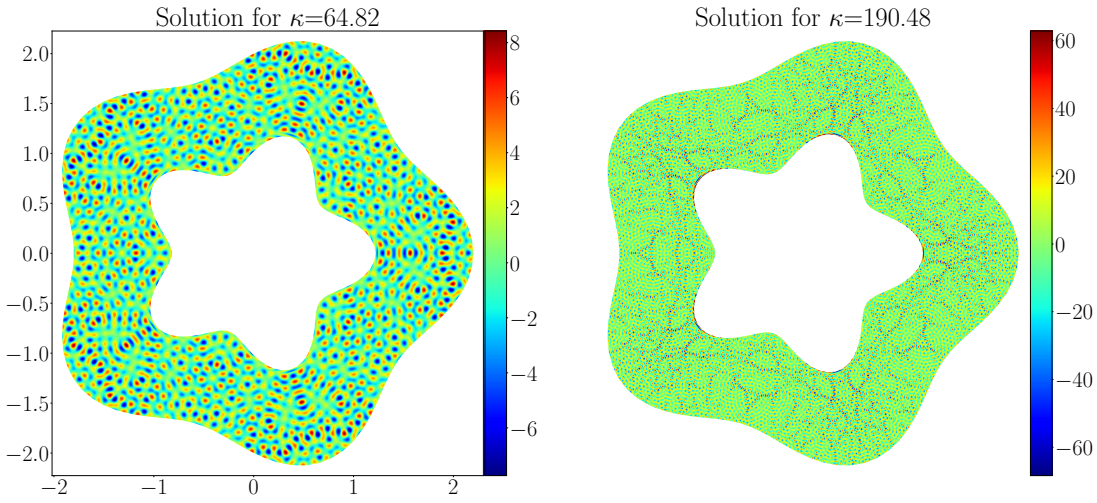
$$\Psi = \left\{ \left(\cos\left(\hat{\theta}(x_1)\right), \sin\left(\hat{\theta}(x_1)\right) \right) \text{ for } (x_1, x_2) \in [0,3] \times [0,1] \right\}, \tag{3.33}$$

where $\hat{\theta}(z) = \frac{\pi}{3} z$. Using the chain rule, (3.2) on X takes a different form of a variable-coefficient elliptic PDE on the reference rectangle. The solutions on X and the convergence plot are presented in Figure 3.15.

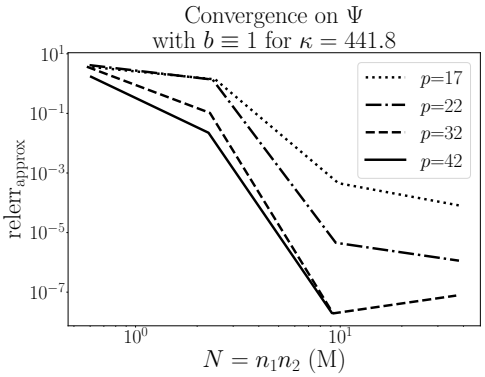
Finally, we demonstrate convergence on a curved domain Ψ with a constant coefficient field $b \equiv 1$, where we have implemented a periodic boundary condition. The domain Ψ is parameterized by the formula

$$\Phi = \left\{ \left(\hat{r}(x_1, x_2) \cos\left(\hat{\theta}(x_1)\right), \hat{r}(x_1, x_2) \sin\left(\hat{\theta}(x_1)\right) \right) \text{ for } (x_1, x_2) \in [0,6] \times [0,1] \right\}, \tag{3.34}$$

where $\hat{r}(z_1, z_2) = 1 + \frac{1}{5} \cos\left(\frac{15}{\pi} z_1 + z_2\right)$ and $\hat{\theta}(z) = \frac{\pi}{3} z$. By applying the chain rule, the Helmholtz operator (3.2) on Ψ takes a different form of a variable-coefficient elliptic PDE on the reference rectangle. The solutions on Ψ and the convergence plot is presented in Figure 3.16.



(a) Computed solutions on Ψ.



(b) Convergence of computed solutions on Ψ.

Figure 3.16: Figure 3.16a shows solutions of constant-coefficient Helmholtz problem on curved domain Ψ with Dirichlet data given by $u \equiv 1$ on $\partial\Psi$ for various wavenumbers κ. Figure 3.16b shows convergence on curved domain Ψ for reference solution $\mathbf{u}_{\text{ref}}$ on HPS discretization for N=36M with $p = 42$. Choosing high orders of p aids in the convergence.

3.7 Conclusion

This work introduces SlabLU, a sparse direct solver framework designed for solving elliptic PDEs. The approach decomposes the domain into a sequence of thin slabs. The degrees of freedom internal to each slab are eliminated in parallel, yielding a reduced matrix $\mathbf{T}$ defined on the slab interfaces. The reduced matrix $\mathbf{T}$ is then factorized directly. The general two-level framework of SlabLU is simple to

implement, easy to parallelize, and can be accelerated via batched linear algebra and GPU computations.

One key innovation of the method is the use of randomized compression with a sparse direct solver to efficiently form $\mathbf{T}$. The dense sub-blocks of $\mathbf{T}$ have exact rank deficiencies in the off-diagonal blocks present in both the non-oscillatory and oscillatory regimes. The use of randomized black-box algorithms provides a purely algebraic means of efficiently forming $\mathbf{T}$ for a variety of PDE discretizations. SlabLU requires $O(N^{5/3})$ time to build and $O(N^{7/6})$ time to apply.

The numerical experiments presented in this work demonstrate that SlabLU is highly effective when used in conjunction with high-order multi-domain spectral collocation schemes. The combination of SlabLU with high order discretization enables the rapid and accurate simulation of large-scale and challenging scattering phenomena on both rectangular and curved domains to high accuracy.

Acknowledgments Anna would like to thank her dad, Andriy, for gifting her the RTX-3090 GPU.

Funding The work reported was supported by the Office of Naval Research (N00014-18-1-2354), by the National Science Foundation (DMS-1952735 and DMS-2012606), and by the Department of Energy ASCR (DE-SC0022251).

Chapter 4: Randomized Strong Recursive Skeletonization: Simultaneous Compression and Factorization of $\mathcal{H}$-matrices in the Black-Box Setting [3]

The hierarchical matrix ($\mathcal{H}^2$-matrix) formalism provides a way to reinterpret the Fast Multipole Method and related fast summation schemes in linear algebraic terms. The idea is to tessellate a matrix into blocks in such as way that each block is either small or of numerically low rank; this enables the storage of the matrix and the application of it to a vector in linear or close to linear complexity. A key motivation for the reformulation is to extend the range of dense matrices that can be represented. Additionally, $\mathcal{H}^2$-matrices in principle also extend the range of operations that can be executed to include matrix inversion and factorization. While such algorithms can be highly efficient for certain specialized formats (such as HBS/HSS matrices based on "weak admissibility"), inversion algorithms for general $\mathcal{H}^2$-matrices tend to be based on nested recursions and recompressions, making them challenging to implement efficiently. An exception is the *strong recursive skeletonization (SRS)* algorithm by Minden, Ho, Damle, and Ying, which involves a simpler algorithmic flow. However, SRS greatly increases the number of blocks of the matrix that need to be stored explicitly, leading to high memory requirements. This manuscript presents the *randomized strong recursive skeletonization (RSRS)* algorithm, which is a reformulation of SRS that incorporates the randomized SVD (RSVD) to simultaneously compress and factorize an $\mathcal{H}^2$-matrix. RSRS is a "black box" algorithm that interacts with the matrix to be compressed only via its action on vectors; this extends the range of the SRS algorithm (which relied on the "proxy source" compression technique) to include dense matrices that arise in sparse direct solvers. Moreover, RSRS leads to

[3] This work was completed in collaboration with Per-Gunnar Martinsson and has appeared in preprint [98].

dramatically simpler data structures, faster execution time, and easier parallelization. RSRS is particularly effective when applied to geometries that have lower dimensionality than the ambient space (as when discretizing a boundary integral equation on a surface in $\mathbb{R}^3$), as it enables off-diagonal blocks to be compressed using their actual numerical ranks, rather than the rank imputed by a proxy surface.

4.1 Introduction

Dense matrices that arise in mathematical physics often have internal structure that makes it possible to solve problems involving elliptic operators in linear or close to linear time. Early techniques such as the Fast Multipole Method [48, 49, 50] exploited mathematical properties of the kernel function directly to enable the fast application of dense matrices to vectors. The $\mathcal{H}^2$-matrix methodology followed, and reinterpreted the FMM in linear algebraic terms, where a matrix is tessellated into blocks in such as way that each block is either small or of numerically low rank. This reinterpretation opened the door to linear complexity algorithms for a wider range of algebraic operations, including the matrix-matrix multiplication, and the construction of invertible factorizations.

The manuscript describes the algorithm *Randomized Strong Recursive Skeletonization (RSRS)* for simultaneously compressing and inverting an $\mathcal{H}^2$-matrix, given a means of applying the matrix and its adjoint to vectors. The precise problem formulation is this: Suppose that $\mathbf{A}$ is an $\mathcal{H}^2$-matrix, and that you are given a fast method for applying $\mathbf{A}$ and its adjoint $\mathbf{A}^*$ to vectors. We then seek to build two "test-matrices" $\mathbf{\Omega}$ and $\mathbf{\Psi}$ with the property that the $\mathcal{H}^2$ representation of $\mathbf{A}^{-1}$ can be constructed from the set $\{\mathbf{Y}, \mathbf{Z}, \mathbf{\Omega}, \mathbf{\Psi}\}$, where

$$\mathbf{Y} = \mathbf{A}\mathbf{\Omega} \qquad \text{and} \qquad \mathbf{Z} = \mathbf{A}^*\mathbf{\Psi}.$$

RSRS is immediately applicable in a range of important environments. First, it can be used to derive a rank-structured representation of any integral operator

for which a fast matrix-vector multiplication algorithm, such as the Fast Multipole Method, is available. In this context, RSRS will directly output an invertible factorization of the integral operator. Second, it can greatly simplify algebraic operations involving products of rank-structured or sparse matrices. As an illustration, the Dirichlet-to-Neumann (DtN) operator for a bounded domain can often be constructed using boundary integral equation techniques, with the overall action of the DtN operator calculated by applying an iterative solver combined with a post-processing step. RSRS would let us directly build and factorize the DtN operator. Relatedly, the perhaps key application of rank-structured matrix algebra is the acceleration of sparse direct solvers, as the dense matrices that arise during LU factorization are often rank structured [3, 65, 68, 95]. In the course of such a solver, a typical operation would be to form a Schur complement such as $\mathbf{S} = \mathbf{B}(I, J)\mathbf{B}(J, J)^{-1}\mathbf{B}(J, I)$, where I and J are two index sets. If $\mathbf{B}(J, J)$ is an $\mathcal{H}^2$-matrix, then $\mathbf{B}(J, J)^{-1}$ can easily be applied to vectors via an LU factorization. If, additionally, $\mathbf{B}(I, J)$ and $\mathbf{B}(J, I)$ are either sparse or rank structured, then $\mathbf{S}$ can easily be applied to a vector. RSRS in this environment allows us to directly compute a factorization of $\mathbf{S}$, which is precisely what is needed in order to move the overall LU factorization forwards.

The fact that RSRS simultaneously compresses and inverts the matrix is a key feature of the algorithm. Previous work on black-box randomized algorithms for compressing rank structured matrices [60, 63, 67, 73] has approached the tasks of compression and inversion as two separate computational stages, with one executed before the other. This approach is very natural for rank structured formats based on "weak admissibility" (where all off-diagonal blocks of the matrix are treated as low rank) since in this case simple and exact inversion algorithms are available [19, 45, 94]. In the "strong admissibility" case (where only interactions between well separated parts of the computational domain are compressed), LU factorization or inversion are much more challenging tasks that require repeated recompression of off-diagonal blocks as the algorithm proceeds [9, 12, 76, 87]. This complication has greatly limited the appeal of direct solvers based on rank structured matrices for

problems in involving fully three dimension geometries where strong admissibility is essential.

In terms of prior work, RSRS draws on the randomized SVD (RSVD) for low rank approximation of matrices [54, 62, 72], and on earlier methods that apply the RSVD to the problem of reconstructing rank structured matrices from matrix-vector products [60, 63, 67, 73]. It in particular draws on ideas from the recent dissertation of James Levitt [59, 60]. Finally, it draws heavily from the prior work on strong recursive skeletonization, including the original work [76], and the more recent [86].

The manuscript is structured as follows: Section 4.2 surveys the key ideas underlying the RSVD and how they can be applied to simultaneously compress all the off-diagonal blocks of a rank structured matrix. Section 4.2 also introduces the core idea that is the main contribution of the present work. Section 4.3 introduces the interpolatory decomposition and shows how it can be used to solve simple linear systems involving rank deficiencies in their coefficient matrices. Section 4.4 introduces our interpretation of the Strong Recursive Skeletonization (SRS) algorithm. Section 4.5 describes Randomized Strong Recursive Skeletonization in full, and discusses refinements to the basic scheme. Section 4.6 presents numerical experiments that demonstrate how RSRS performs in terms of speed, memory requirements, and precision.

4.2 Randomized compression and factorization of rank structured matrices

In this section, we highlight the key methodological contributions of the manuscript for a simple model problem. For points $\{\boldsymbol{x}_j\}_{j=1}^N$ shown in Figure 4.1, consider the $N \times N$ matrix $\mathbf{A}$ with entries

$$\mathbf{A}(i,j) = k(\boldsymbol{x}_i, \boldsymbol{x}_j) \tag{4.1}$$

given by a kernel function k, for instance $k(\boldsymbol{x}, \boldsymbol{y}) = \log|\boldsymbol{x} - \boldsymbol{y}|$. The singularity at $x = y$ can be handled with an appropriate quadrature rule. The matrix (4.1) and

its adjoint can be applied rapidly to vectors using the fast multipole method. The objective is to recover an *approximate* representation of $\mathbf{A}^{-1}$ using random sketches

$$\underset{N\times p}{\mathbf{Y}} = \underset{N\times N}{\mathbf{A}}\ \underset{N\times p}{\mathbf{\Omega}}, \qquad \underset{N\times p}{\mathbf{Z}} = \underset{N\times N}{\mathbf{A}^*}\ \underset{N\times p}{\mathbf{\Psi}}, \qquad \mathbf{\Omega}, \mathbf{\Psi} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{4.2}$$

using Gaussian test matrices $\mathbf{\Omega}, \mathbf{\Psi}$. We will discuss the needed number of samples later, but for now, consider that $p \ll N$. The material in Section 2.1 follows [54, 71, 77], while the techniques in Sections 2.2 and 2.3 are based on ideas in [59, 60].

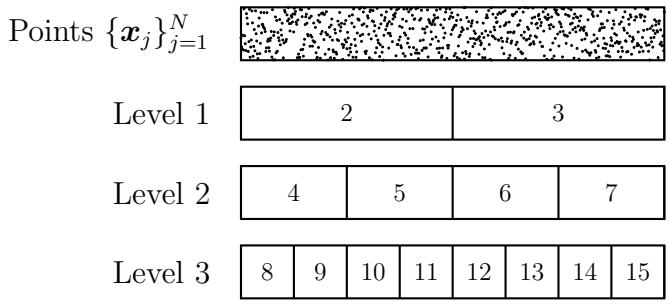


Figure 4.1: The hierarchical tessellation of boxes for a quasi one dimensional model problem. Boxes on the same level which share an edge or corner are called *neighbors*. Two boxes which are not neighbors are called *well-separated*. A box b has a *far-field* consisting of all boxes which are well-separated from b. As an example, box 9 has neighbors {8,9,10} and far-field {11,..., 15}.

The methodology uses that $\mathbf{A}$ has certain rank deficiencies to recover a sparse factorization of $\mathbf{A}^{-1}$ efficiently. Consider the tessellation of points into boxes $8, \ldots, 15$ on Level 3 of Figure 4.1, where each box has $m := N/8$ points. The matrix $\mathbf{A}$ has the following property for interactions between far-field blocks

$$\underset{m\times m}{\mathbf{A}_{i,j}} \approx \underset{m\times k}{\mathbf{Q}_i}\ \underset{k\times k}{\widetilde{\mathbf{A}}_{ij}}\ \underset{k\times m}{\mathbf{U}_j^*} \tag{4.3}$$

where index set I_i and I_j correspond to the indices of points of well-separated colleague boxes i and j, respectively. The low rank property of far-field interactions in (4.3) is depicted in Figure 4.2a. The blocks corresponding to interactions between near neighbors are stored densely.

In the next sections, we discuss methods for recovering the bases $\mathbf{Q}_i$ and $\mathbf{U}_j$, which we call *block nullification*, in the black-box setting of (4.2). We also discuss

techniques for recovering block-sparse interactions (e.g. interactions between near neighbors), which we call *block extraction*. First, to provide context, we review standard methods for recovering globally low-rank factors from matrix-vector products.

4.2.1 Review of randomized sketching for a low rank matrix

Suppose we would like to compute a low-rank approximation to the matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ where the rank k is approximate and known apriori. Randomized low rank compression provides a powerful set of techniques for accomplishing this task via the action of $\mathbf{A}$ and its adjoint on a small number of vectors.

A low rank approximation to $\mathbf{A}$ can be computed in two stages. First, we would like to compute $\mathbf{Q} \in \mathbb{R}^{m \times k}$ with orthonormal columns for which $\mathbf{A} \approx \mathbf{Q}\mathbf{Q}^*\mathbf{A}$. Once $\mathbf{Q}$ is known, then we can form the matrix $\mathbf{B} = \mathbf{A}^*\mathbf{Q}$ and the low-rank factorization of $\mathbf{A}$ takes the form

$$\underset{m \times n}{\mathbf{A}} = \underset{m \times k}{\mathbf{Q}} \; \underset{k \times n}{\mathbf{B}} \, . \tag{4.4}$$

To find an approximate basis for the range of $\mathbf{A}$ (e.g. find $\mathbf{Q}$ such that $\mathbf{A} \approx \mathbf{Q}\mathbf{Q}^*\mathbf{A}$), we generate a randomized sketch of $\mathbf{A}$

$$\underset{m \times (k+l)}{\mathbf{Y}} = \underset{m \times n}{\mathbf{A}} \; \underset{n \times (k+l)}{\mathbf{\Omega}}, \qquad \mathbf{\Omega} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \tag{4.5}$$

where l is a small oversampling parameter (e.g. $l = 5$). The sketch $\mathbf{Y}$ approximately spans the column space of $\mathbf{A}$, and desired orthonormal basis $\mathbf{Q}$ can be computed by running a procedure to orthonormalize $\mathbf{Y}$

$$\underset{m \times (k+l)}{\mathbf{Q}} = \mathtt{orth}\big(\mathbf{Y}\big), \qquad \text{where } \mathbf{Y} = \mathbf{Q}\mathbf{R} \tag{4.6}$$

When the rank k is not known apriori, there are adaptive algorithms available which build a low rank representation of $\mathbf{A}$ from successive sketches. Randomized sketching methods can be used to construct a wide range of decompositions, including the interpolative decomposition which we introduce in Section 4.3.2. These algorithms are especially useful in the black-box setting of (4.2) because the matrix $\mathbf{A}$ is only

accessed through its action on vectors. In the next section, we discuss a modification of randomized low rank matrices which can be used to recover the orthogonal bases for the row and column blocks of (4.3).

4.2.2 Block Nullification

In the black-box setting of (4.2), we would like to recover the bases $\mathbf{Q}_i, \mathbf{U}_i \in \mathbb{R}^{m\times k}$ for all blocks $i = 8, \ldots, 15$ as defined in (4.3). Consider that we do this using some modification of randomized sketching methods. Let the test and sketch matrices be tessellated according to the tree decomposition in Figure 4.1 on Level 3, so that

$$\mathbf{Y} = \begin{pmatrix} \mathbf{Y}_8 \\ \mathbf{Y}_9 \\ \vdots \\ \mathbf{Y}_{15} \end{pmatrix}, \qquad \mathbf{\Omega} = \begin{pmatrix} \mathbf{\Omega}_8 \\ \mathbf{\Omega}_9 \\ \vdots \\ \mathbf{\Omega}_{15} \end{pmatrix}, \qquad \mathbf{Y}_i \in \mathbb{R}^{m\times p},\ \mathbf{\Omega}_i \in \mathbb{R}^{m\times p} \tag{4.7}$$

and similar tessellations for $\mathbf{Z}, \mathbf{\Psi}$.

The structure of $\mathbf{A}$ has dense interactions between near neighbor blocks, c.f. Figure 4.2a, which complicate the use of randomized sketching techniques. Ideally, the structure of the test matrices $\mathbf{\Omega}$ would correspond to the sparsity pattern of the low rank blocks that we need to sample. Consider a 'structured' test matrix $\mathbf{\Omega'} \in \mathbb{R}^{N\times k}$ for sampling the far-field of block $i = 9$, where $\mathbf{\Omega}'_8, \mathbf{\Omega}'_9, \mathbf{\Omega}'_{10} = \mathbf{0}$ and the other blocks are Gaussian random matrices. Then sketching to produce $\mathbf{Y'} = \mathbf{A\Omega'}$, extracting the I_9 block, and post-processing would yield a basis for $\mathbf{Q}_9$:

$$\underset{m\times k}{\mathbf{U}_9} = \texttt{orth}(\underset{m\times k}{\mathbf{Y}'_9}), \qquad \text{where } \underset{N\times k}{\mathbf{Y}'} = \underset{N\times N}{\mathbf{A}}\ \underset{N\times k}{\mathbf{\Omega}'} \tag{4.8}$$

To recover the basis for $i = 12$, another structured test matrix $\mathbf{\Omega''} \in \mathbb{R}^{N\times k}$ can be designed with zero blocks $\mathbf{\Omega}''_{11}, \mathbf{\Omega}''_{12}, \mathbf{\Omega}''_{13} = \mathbf{0}$ and procedure in (4.8) can be repeated for the sketch $\mathbf{Y''} = \mathbf{A\Omega''}$. Using these structured test matrices, tailored for sampling the far field of every box, would require $\frac{k}{m}N$ total samples and k^2N post-processing cost.

Block nullification accomplishes the same aim with much fewer samples and slightly increased post-processing costs. Let us again consider sampling the far-field of box $i = 9$, and consider that we construct a matrix $\mathbf{N}'$ from the fully dense test matrix $\mathbf{\Omega}$ defined in (4.7)

$$\underset{p\times(p-3m)}{\mathbf{N}'} = \texttt{null}\begin{pmatrix}\underset{m\times p}{\mathbf{\Omega}_8}\\ \mathbf{\Omega}_9\\ \mathbf{\Omega}_{10}\end{pmatrix} \tag{4.9}$$

where the operation `null` gives an orthogonal basis for the nullspace of matrix. Suppose that $p = (3m + k)$, then $\mathbf{N}'$ is k-dimensional with high probability. Multiplying $\mathbf{N}'$ on the right of $\mathbf{\Omega}$ gives

$$\underset{N\times p}{\mathbf{\Omega}}\ \underset{p\times k}{\mathbf{N}'} = \begin{pmatrix}\underset{m\times p}{\mathbf{\Omega}_8}\ \underset{p\times k}{\mathbf{N}'}\\ \mathbf{0}\\ \mathbf{0}\\ \mathbf{0}\\ \mathbf{\Omega}_{11}\mathbf{N}'\\ \vdots\\ \mathbf{\Omega}_{15}\mathbf{N}'\end{pmatrix} = \begin{pmatrix}\underset{m\times k}{\mathbf{\Omega}'_8}\\ \mathbf{0}\\ \mathbf{0}\\ \mathbf{0}\\ \mathbf{\Omega}'_{11}\\ \vdots\\ \mathbf{\Omega}'_{15}\end{pmatrix} := \mathbf{\Omega}' \in \mathbb{R}^{N\times k}, \tag{4.10}$$

where $\mathbf{\Omega}'$ is the 'structured' Gaussian random matrix discussed earlier. The blocks $\mathbf{\Omega}'_9, \mathbf{\Omega}'_{10}, \mathbf{\Omega}'_{11}$ are zero matrices by construction. Likewise, the structured random matrix $\mathbf{\Omega}''$ for sampling the far field for $i = 12$ can be constructed in a similar fashion.

$$\underset{N\times p}{\mathbf{\Omega}}\ \underset{p\times k}{\mathbf{N}''} := \underset{N\times k}{\mathbf{\Omega}''}, \text{where } \underset{p\times(p-3m)}{\mathbf{N}''} = \texttt{null}\begin{pmatrix}\mathbf{\Omega}_{11}\\ \mathbf{\Omega}_{12}\\ \mathbf{\Omega}_{13}\end{pmatrix} \tag{4.11}$$

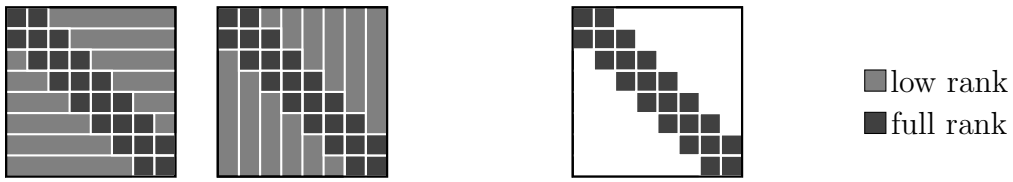
To recover $\mathbf{Q}_9$ as described in (4.8), we need the I_9 block of the sketch $\mathbf{Y}'$, which we can construct by post-processing $\mathbf{\Omega}$

$$\underset{N\times k}{\mathbf{Y}'} = \underset{N\times N}{\mathbf{A}}\ \underset{N\times k}{\mathbf{\Omega}'} = \underset{N\times N}{\mathbf{A}}\left(\underset{N\times p}{\mathbf{\Omega}}\ \underset{p\times k}{\mathbf{N}'}\right) = \underset{N\times p}{\mathbf{Y}}\ \underset{p\times k}{\mathbf{N}'}. \tag{4.12}$$

Note that because we only need the I_9 block, the full sketch $\mathbf{Y}'$ does not need to be formed explicitly. Instead, $\mathbf{Y}'_9$ can be constructed as

$$\underset{m\times k}{\mathbf{Y}'_9} = \underset{m\times p}{\mathbf{Y}_9}\ \underset{p\times k}{\mathbf{N}'}, \qquad \text{then } \mathbf{Q}_9 = \texttt{orth}\left(\mathbf{Y}'_9\right). \tag{4.13}$$

Likewise, the basis $\mathbf{Q}_{12}$ can be recovered by computing $\mathbf{N}''$ as in (4.11) and orthonormalizing the sketch $\mathbf{Y}''_{12} = \mathbf{Y}_{12}\mathbf{N}''$. Block nullification allows us to recover all bases $\mathbf{Q}_8, \ldots, \mathbf{Q}_{15}$ using only $p = (3m + k)$ samples of $\mathbf{A}$ and $\mathcal{O}\left((m^2 + m^2 k)N\right)$ post-processing cost. The big-O notation hides constants related to the geometry which are discussed in a later section. The bases $\mathbf{U}_8, \ldots, \mathbf{U}_{15}$ can be recovered using a similar procedure for post-processing $\mathbf{Z}$ and $\mathbf{\Psi}$.



(a) Setting for block nullification. (b) Setting for block extraction.

Figure 4.2: The figure depicts a rank-structured matrix $\mathbf{A}$ in two settings. Figure 4.2a shows a dense matrix $\mathbf{A}$ that uses low rank bases for each row and column block. Blocks which are well-separated, according to the geometry in Figure 4.1, are compressed using uniform bases, c.f. (4.3) for a formal definition. Figure 4.2b shows a special case of a rank-structured matrix, where the well-separated blocks have rank exactly 0.

4.2.3 Block Extraction

In addition to low-rank factors, the small dense blocks corresponding to interactions between adjacent neighbors need to be recovered as well. For this task, we describe a randomized sampling technique which is useful for block-sparse matrices. The setting for using this technique appropriately in the context of rank-structured matrices will be described in a later section. For now, we consider a special case of a rank structured matrix $\mathbf{A}$ when the off-diagonal rank is exactly 0. Then, $\mathbf{A}$ is block-sparse with the structure shown in Figure 4.2b. The technique known as *block extraction* recovers the dense blocks $\mathbf{A}_{8,8}, \mathbf{A}_{8,9}, \mathbf{A}_{9,8}$, etc. by post-processing $\{\mathbf{Y}, \mathbf{Z}, \mathbf{\Omega}, \mathbf{\Psi}\}$ from samples (4.2).

Because $\mathbf{A}$ is sparse, all the blocks of $\mathbf{A}$ can be recovered using only $3m$ samples by constructing $\mathbf{\Omega}$ with special sparse structure (e.g. carefully placed identity

matrices). For general geometries, this approach requires solving a combinatorial graph problem to find the sparsity pattern of $\mathbf{\Omega}$ to recover $\mathbf{A}$ from black-box samples. Block extraction uses a different approach, which is very similar in spirit to block nullification. Let us again design a 'structured' test matrix $\mathbf{\Omega}'$ to recover the blocks $\mathbf{A}_{9,8}, \mathbf{A}_{9,9}, \mathbf{A}_{9,10}$ of the following form

$$\mathbf{\Omega}' = \begin{pmatrix} \underset{m\times m}{\mathbf{I}} & & \\ & \underset{m\times m}{\mathbf{I}} & \\ & & \underset{m\times m}{\mathbf{I}} \\ & \underset{m\times 3m}{\mathbf{\Omega}'_{11}} & \\ & \vdots & \\ & \underset{m\times 3m}{\mathbf{\Omega}'_{15}} & \end{pmatrix} \tag{4.14}$$

Note that the entries on blocks $i = 11, \ldots, 15$ can be arbitrary because $\mathbf{A}$ is block-sparse. Then applying $\mathbf{\Omega}'$ to form the sketch $\mathbf{Y}'$ and extracting the I_9 block gives

$$\begin{pmatrix} \mathbf{A}_{98}, & \mathbf{A}_{9,9} & \mathbf{A}_{9,10} \end{pmatrix} = \underset{m\times 3m}{\mathbf{Y}'_9}, \qquad \text{where } \underset{N\times 3m}{\mathbf{Y}'} = \underset{N\times N}{\mathbf{A}}\ \underset{N\times 3m}{\mathbf{\Omega}'}. \tag{4.15}$$

Consider that we draw p samples using dense Gaussian random test matrices, as in (4.7) and that we compute a matrix $\mathbf{M}$ from the fully dense test matrix $\mathbf{\Omega}$ as

$$\underset{p\times 3m}{\mathbf{M}} = \begin{pmatrix} \underset{m\times p}{\mathbf{\Omega}_8} \\ \mathbf{\Omega}_9 \\ \mathbf{\Omega}_{10} \end{pmatrix}^{\dagger}, \tag{4.16}$$

where $\dagger$ denotes the right pseudo-inverse. With high probability, the Gaussian random matrix in (4.16) has linearly independent rows. Then applying $\mathbf{M}$ to the right of Gaussian test matrix $\mathbf{\Omega}$ gives

$$\underset{N\times p}{\mathbf{\Omega}}\ \underset{p\times 3m}{\mathbf{M}} = \underset{N\times 3m}{\mathbf{\Omega}'}, \tag{4.17}$$

where $\mathbf{\Omega}'$ has the sparsity pattern of (4.14). Note that $\mathbf{\Omega}'_{11} = \mathbf{\Omega}_{11}\mathbf{M}$ is not a Gaussian random matrix any longer (e.g. $\mathbf{\Omega}'_{11} \not\sim \mathcal{N}(\mathbf{0}, \mathbf{I})$). This, however, does not matter for

the purposes of extracting sparse blocks because $\mathbf{A}_{9,11} = \mathbf{0}$. To construct the sample $\mathbf{Y}'_9$, we simply compute $\mathbf{M}$ from $\mathbf{\Omega}$ using (4.16), then compute

$$\underset{m\times 3m}{\mathbf{Y}'_9} = \underset{m\times p}{\mathbf{Y}_9}\ \underset{p\times 3m}{\mathbf{M}}, \qquad \text{where } \underset{N\times 3m}{\mathbf{Y}'} = \underset{N\times N}{\mathbf{A}}\ \underset{N\times 3m}{\mathbf{\Omega}'} = \underset{N\times N}{\mathbf{A}}\left(\underset{N\times p}{\mathbf{\Omega}}\ \underset{p\times 3m}{\mathbf{M}}\right) \tag{4.18}$$

Note that all of $\mathbf{Y}'$ does not need to be computed, only the I_9 block $\mathbf{Y}'_9$ is needed. Block extraction allows us to recover all sparse blocks of $\mathbf{A}$ using only $p = (3m + l)$ samples, where l is a small oversampling parameter, and $\mathcal{O}(m^2 N)$ post-processing cost. Similar techniques are often useful for extracting sparse blocks by sampling the adjoint operator and post-processing $\mathbf{Z}$ and $\mathbf{\Psi}$.

4.2.4 Factorizing Rank-Structured Matrices using Randomized Sampling Techniques

The objective of this work is to compute an invertible factorization of $\mathbf{A}$. The details of this machinery are discussed in subsequent sections. For now, assume the factorization takes the following form

$$\mathbf{A} = \mathbf{V}_8 \dots \mathbf{V}_{15}\ \widetilde{\mathbf{A}}\ \mathbf{W}_{15} \dots \mathbf{W}_8, \tag{4.19}$$

where $\widetilde{\mathbf{A}}$ is block-diagonal. The matrices $\mathbf{V}_i, \mathbf{W}_i$ for $i = 8, \dots 15$ diagonalize the row interactions and the column interactions of the block i. In the subsequent sections, we describe the operators $\mathbf{V}_i$ and $\mathbf{W}_i$, but for this discussion, it is important to note that they are sparse and easy to invert.

Consider the following simpler decomposition of $\mathbf{A}$, which only includes the diagonalization operators for block $i = 8$

$$\mathbf{A} \ = \ \mathbf{V}_8\ \hat{\mathbf{A}}\ \mathbf{W}_8, \tag{4.20}$$

where $\hat{\mathbf{A}}$ is defined in terms of the factorization of (4.19). In later sections, we will describe how $\mathbf{V}_8$ and $\mathbf{W}_8$ can be recovered using only a small number of random sketches (4.2) of the operator $\mathbf{A}$ and its adjoint by post-processing $\{\mathbf{Y}, \mathbf{\Omega}, \mathbf{Z}, \mathbf{\Psi}\}$.

Suppose that $\mathbf{V}_8$ and $\mathbf{W}_8$ have been recovered and that we would like to use similar techniques to recover the operators $\mathbf{V}_9$ and $\mathbf{W}_9$. Then, it would be natural to compute random sketches of $\hat{\mathbf{A}}$

$$\hat{\mathbf{Y}} = \hat{\mathbf{A}}\hat{\mathbf{\Omega}}, \quad \hat{\mathbf{Z}} = \hat{\mathbf{A}}^*\hat{\mathbf{\Psi}}$$

and post-process $\{\hat{\mathbf{Y}}, \hat{\mathbf{\Omega}}, \hat{\mathbf{Z}}, \hat{\mathbf{\Psi}}\}$. A key observation of this manuscript, is that it is *not necessary to draw sketches* of $\hat{\mathbf{A}}$ because the matrix can be written in terms of $\mathbf{A}$ as

$$\hat{\mathbf{A}} = \mathbf{V}_8^{-1}\mathbf{A}\mathbf{W}_8^{-1}.$$

Instead, test and sketch matrices can be updated $\{\mathbf{Y}, \mathbf{\Omega}, \mathbf{Z}, \mathbf{\Psi}\} \Rightarrow \{\hat{\mathbf{Y}}, \hat{\mathbf{\Omega}}, \hat{\mathbf{Z}}, \hat{\mathbf{\Psi}}\}$ using the following formulas

$$\begin{aligned} \hat{\mathbf{Y}} &= \mathbf{V}_8\ \mathbf{Y}, \qquad & \hat{\mathbf{\Omega}} &= \mathbf{W}_8^{-1}\mathbf{\Omega} \\ \hat{\mathbf{Z}} &= \mathbf{W}_8^*\ \mathbf{Z}, \qquad & \hat{\mathbf{\Psi}} &= \mathbf{V}_8^{-*}\mathbf{\Psi} \end{aligned} \tag{4.21}$$

This observation allows us to sample the operator and its adjoint as in (4.2), then recover the factorization of $\mathbf{A}$ by reusing the random sketches drawn initially.

4.3 The interpolative decomposition and recursive skeletonization

In this section, we review preliminaries which are useful in understanding the strong recursive skeletonization algorithm. First, we discuss Gaussian elimination and block elimination matrices in Section 4.3.1. Then, we discuss interpolative decompositions (ID) for low rank compression in Section 4.3.2, which use a subset of the rows or columns of the original matrix as a basis. Section 4.3.3 describes how the interpolative decomposition can be used to represent off-diagonal blocks in a rank-structured $\mathbf{A}$ to efficiently compute a direct solver $\mathbf{A}^{-1}$.

4.3.1 Gaussian elimination and block elimination matrices

Consider a matrix of the form

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{0} \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{A}_{23} \\ \mathbf{0} & \mathbf{A}_{32} & \mathbf{A}_{33} \end{pmatrix}.$$

If $\mathbf{A}_{11}$ is non-singular, we can "decouple" it from the other blocks via one step of *block Gaussian elimination*. We express this mathematically through a factorization

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{0} \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{A}_{23} \\ \mathbf{0} & \mathbf{A}_{32} & \mathbf{A}_{33} \end{pmatrix} = \mathbf{L} \begin{pmatrix} \mathbf{A}_{11} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{S}_{22} & \mathbf{A}_{23} \\ \mathbf{0} & \mathbf{A}_{32} & \mathbf{A}_{33} \end{pmatrix} \mathbf{U} \tag{4.22}$$

where $\mathbf{L}$ and $\mathbf{U}$ are "block elimination matrices" of the form

$$\mathbf{L} = \begin{pmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{A}_{21}\mathbf{A}_{11}^{-1} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{pmatrix} \qquad \text{and} \qquad \mathbf{U} = \begin{pmatrix} \mathbf{I} & \mathbf{A}_{11}^{-1}\mathbf{A}_{12} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{pmatrix}, \tag{4.23}$$

and where $\mathbf{S}_{22}$ is the "Schur complement"

$$\mathbf{S}_{22} = \mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}.$$

For future reference, let us introduce a function `elim` that builds the elimination matrices. To be precise,

$$\texttt{elim}(\mathbf{B}, I, J, n)$$

is the $n \times n$ identity matrix, except that the matrix $\mathbf{B}$ has been inserted in the block identified by the index vector I and J. In other words, in (4.23),

$$\mathbf{L} = \texttt{elim}(\mathbf{A}_{21}\mathbf{A}_{11}^{-1}, I_2, I_1, n) \qquad \text{and} \qquad \mathbf{U} = \texttt{elim}(\mathbf{A}_{11}^{-1}\mathbf{A}_{12}, I_1, I_2, n).$$

Note that block-elimination matrices of the form `elim` are simple to invert by toggling the sign of the off-diagonal block, e.g.

$$\mathbf{L}^{-1} = \texttt{elim}(-\mathbf{A}_{21}\mathbf{A}_{11}^{-1}, I_2, I_1, n)$$

Remark 4.1. When $\mathbf{A}$ is symmetric and positive definite (spd), it is preferable to compute its Cholesky factorization. We would then first factorize the $\mathbf{A}_{11}$ block as

$$\mathbf{A}_{11} = \mathbf{C}_{11}\mathbf{C}_{11}^{*}$$

where $\mathbf{C}_{11}$ is lower triangular. Then instead of (4.22), we would form the triangular factorization

$$\begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{0} \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{A}_{23} \\ \mathbf{0} & \mathbf{A}_{32} & \mathbf{A}_{33} \end{pmatrix} = \begin{pmatrix} \mathbf{C}_{11} & \mathbf{0} & \mathbf{0} \\ \mathbf{A}_{21}\mathbf{C}_{11}^{-*} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{S}_{22} & \mathbf{A}_{23} \\ \mathbf{0} & \mathbf{A}_{32} & \mathbf{A}_{33} \end{pmatrix} \begin{pmatrix} \mathbf{C}_{11}^{*} & \mathbf{C}_{11}^{-1}\mathbf{A}_{12} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I} \end{pmatrix}.$$

When $\mathbf{A}$ is not spd, one could form analogous triangular factorizations using either the (partially pivoted) LU decomposition of $\mathbf{A}_{11}$, or potentially an LDL factorization. In this manuscript, we will focus on block diagonal factorizations, but all techniques described can easily be adapted to form triangular factorizations instead.

4.3.2 The interpolatory decomposition

Let $\mathbf{B}$ be an $m \times n$ matrix of rank k. The *interpolatory decomposition (ID)* of $\mathbf{B}$ is a low rank factorization where a subset of the columns/rows are used to span its column/row space. For instance, in the *column ID*, we pick a set of k linearly independent columns identified through the "skeleton" index vector J_{s}, and collect the remaining indices in the "residual" index vector J_{r}, and set

$$J = [J_{\mathrm{r}},\, J_{\mathrm{s}}], \qquad \mathbf{B}_{\mathrm{r}} = \mathbf{B}(:, J_{\mathrm{r}}), \qquad \mathbf{B}_{\mathrm{s}} = \mathbf{B}(:, J_{\mathrm{s}}),$$

Since $\mathbf{B}$ has rank k, there exists a matrix $\mathbf{T}_{\mathrm{sr}}$ of size $k \times (n-k)$ such that

$$\mathbf{B}_{\mathrm{r}} = \mathbf{B}_{\mathrm{s}}\mathbf{T}_{\mathrm{sr}}.$$

This allows us to factor the matrix $\mathbf{B}$ as

$$\mathbf{B}(:, J) = \big(\mathbf{B}_{\mathrm{r}},\, \mathbf{B}_{\mathrm{s}}\big) = \big(\mathbf{B}_{\mathrm{s}}\mathbf{T}_{\mathrm{sr}},\, \mathbf{B}_{\mathrm{s}}\big) = \big(\mathbf{0},\, \mathbf{B}_{\mathrm{s}}\big) \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{T}_{\mathrm{sr}} & \mathbf{I} \end{pmatrix}. \tag{4.24}$$

Interpolatory decompositions can also be computed of matrices that are only of approximate low rank. The errors induced can in theory be significantly larger than

those resulting from the optimal low rank decomposition obtained by truncating a singular value decomposition. However, in practice the error tends to be modest as long as the singular values of the input matrix decay at a decent rate. For numerical stability, we would like the matrix $\begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{T}_{\text{sr}} & \mathbf{I} \end{pmatrix}$ to be as well-conditioned as possible, but in practice tends to mean that we want the entries of $\mathbf{T}_{\text{sr}}$ to be small. It has been demonstrated that it is always possible to pick the set J_{s} so that every entry of $\mathbf{T}_{\text{sr}}$ has modulus bounded by one, and practical algorithms that ensure that the entries are of modest size are known. See [27] for a detailed discussion of different algorithms for computing the ID in practice.

For completeness, let us also describe the *row ID*, where a set of linearly independent rows of $\mathbf{B}$ would be identified by an index vector I_{s}. Collecting the remaining row indices in the index vector I_{r}, we then get the factorization

$$\mathbf{B}([I_{\text{r}}, I_{\text{s}}], :) = \begin{pmatrix} \mathbf{I} & \mathbf{T}_{\text{rs}} \\ \mathbf{0} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{0} \\ \mathbf{B}_{\text{s}} \end{pmatrix},$$

where $\mathbf{B}_{\text{rs}}$ is the $(m-k) \times k$ matrix such that $\mathbf{B}(I_{\text{r}}, :) = \mathbf{T}_{\text{rs}}\mathbf{B}(I_{\text{s}}, :)$.

4.3.3 Classical skeletonization (weak admissibility)

We in this section describe how the interpolatory decomposition that we introduced in Section 4.3.2 can be used to exploit rank deficiencies in off-diagonal blocks to map a given linear system to another "equivalent one" one with a smaller coefficient matrix. The technique described has previously been published as a fast direct solver for boundary integral equations, and is now widely known as "recursive skeletonization" [43, 45, 58, 74, 75]. It is based on "weak admissibility", and forms the basis for the more complex algorithm based on strong admissibility that is the main focus of this manuscript.

As a model problem, let us consider a matrix $\mathbf{A}$ that has been partitioned into 2×2 blocks as

$$\mathbf{A} = \left(\begin{array}{c|c} \mathbf{A}_{\text{bb}} & \mathbf{A}_{\text{fb}} \\ \hline \mathbf{A}_{\text{fb}} & \mathbf{A}_{\text{ff}} \end{array}\right), \tag{4.25}$$

where the off-diagonal blocks $\mathbf{A}_{\mathrm{fb}}$ and $\mathbf{A}_{\mathrm{bf}}$ each have low rank. Let k denote the rank of these blocks, and let m and n represent the dimensions of the blocks $\mathbf{A}_{\mathrm{bb}}$ and $\mathbf{A}_{\mathrm{ff}}$, respectively. We will construct block elimination matrices that map $\mathbf{A}$ to a block diagonal matrix where one block has size $(m-k)\times(m-k)$, and other has size $(k+n)\times(k+n)$. Let I_{b} and I_{f} denote index vectors that identify the blocks, so that

$$(1:n) = I_{\mathrm{b}} \cup I_{\mathrm{f}}.$$

(For now, our discussion is one purely of linear algebra, but we use notation that will serve us well when applying these ideas to solve potential problems in Section 4.4.) The first step of the factorization process is to form the row ID of $\mathbf{A}_{\mathrm{af}}$ and the column ID of $\mathbf{A}_{\mathrm{fa}}$ to obtain factorizations

$$\mathbf{A}_{\mathrm{bf}}([I_{\mathrm{r}}, I_{\mathrm{s}}], :) = \begin{pmatrix} \mathbf{A}_{\mathrm{rf}} \\ \mathbf{A}_{\mathrm{sf}} \end{pmatrix} = \begin{pmatrix} \mathbf{T}_{\mathrm{rs}}\mathbf{A}_{\mathrm{sf}} \\ \mathbf{A}_{\mathrm{sf}} \end{pmatrix} \tag{4.26}$$

and

$$\mathbf{A}_{\mathrm{fb}}(:, [J_{\mathrm{r}}, J_{\mathrm{s}}]) = \begin{bmatrix} \mathbf{A}_{\mathrm{fr}}, & \mathbf{A}_{\mathrm{fs}} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{\mathrm{fs}}\mathbf{T}_{\mathrm{sr}}, & \mathbf{A}_{\mathrm{fs}} \end{bmatrix}. \tag{4.27}$$

Let us reorder the indices within the current box to have the residual indices go first:

$$I = [I_{\mathrm{r}},\, I_{\mathrm{s}},\, I_{\mathrm{f}}], \qquad \text{and} \qquad J = [J_{\mathrm{r}},\, J_{\mathrm{s}},\, I_{\mathrm{f}}].$$

This results in the tessellation

$$\mathbf{A}(I,J) = \left(\begin{array}{cc|c} \mathbf{A}_{\mathrm{rr}} & \mathbf{A}_{\mathrm{rs}} & \mathbf{A}_{\mathrm{rf}} \\ \mathbf{A}_{\mathrm{sr}} & \mathbf{A}_{\mathrm{ss}} & \mathbf{A}_{\mathrm{sf}} \\ \hline \mathbf{A}_{\mathrm{fr}} & \mathbf{A}_{\mathrm{fs}} & \mathbf{A}_{\mathrm{ff}} \end{array}\right) = \left(\begin{array}{cc|c} \mathbf{A}_{\mathrm{rr}} & \mathbf{A}_{\mathrm{rs}} & \mathbf{T}_{\mathrm{rs}}\mathbf{A}_{\mathrm{sf}} \\ \mathbf{A}_{\mathrm{sr}} & \mathbf{A}_{\mathrm{ss}} & \mathbf{A}_{\mathrm{sf}} \\ \hline \mathbf{A}_{\mathrm{fs}}\mathbf{T}_{\mathrm{sr}} & \mathbf{A}_{\mathrm{fs}} & \mathbf{A}_{\mathrm{ff}} \end{array}\right). \tag{4.28}$$

Next, we factorize the matrix to introduce zero blocks in the "fr" and "rf" locations:

$$\mathbf{A}(I,J) = \left(\begin{array}{cc|c} \mathbf{I} & \mathbf{T}_{\mathrm{rs}} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} & \mathbf{I} \end{array}\right) \left(\begin{array}{cc|c} \mathbf{X}_{\mathrm{rr}} & \mathbf{X}_{\mathrm{rs}} & \mathbf{0} \\ \mathbf{X}_{\mathrm{sr}} & \mathbf{A}_{\mathrm{ss}} & \mathbf{A}_{\mathrm{sf}} \\ \hline \mathbf{0} & \mathbf{A}_{\mathrm{fs}} & \mathbf{A}_{\mathrm{ff}} \end{array}\right) \left(\begin{array}{cc|c} \mathbf{I} & \mathbf{0} & \mathbf{0} \\ \mathbf{T}_{\mathrm{sr}} & \mathbf{I} & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} & \mathbf{I} \end{array}\right). \tag{4.29}$$

Recognizing that the outside factors in (4.29) are both block elimination matrices, we define

$$\mathbf{E} := \mathtt{elim}(\mathbf{T}_{\mathrm{rs}}, I_{\mathrm{r}}, I_{\mathrm{s}}, n) \qquad \text{and} \qquad \mathbf{F} := \mathtt{elim}(\mathbf{T}_{\mathrm{sr}}, J_{\mathrm{s}}, J_{\mathrm{r}}, n).$$

so that we can rewrite (4.29) as

$$\mathbf{E}\left(\begin{array}{cc|c} \mathbf{X}_{\mathrm{rr}} & \mathbf{X}_{\mathrm{rs}} & \mathbf{0} \\ \mathbf{X}_{\mathrm{sr}} & \mathbf{A}_{\mathrm{ss}} & \mathbf{A}_{\mathrm{sf}} \\ \hline \mathbf{0} & \mathbf{A}_{\mathrm{fs}} & \mathbf{A}_{\mathrm{ff}} \end{array}\right)\mathbf{F}. \tag{4.30}$$

We next perform one step of block Gaussian elimination to decouple $\mathbf{X}_{\mathrm{rr}}$, so that

$$\mathbf{A}(I,J) = \mathbf{EL}\left(\begin{array}{cc|c} \mathbf{X}_{\mathrm{rr}} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{X}_{\mathrm{ss}} & \mathbf{A}_{\mathrm{sf}} \\ \hline \mathbf{0} & \mathbf{A}_{\mathrm{fs}} & \mathbf{A}_{\mathrm{ff}} \end{array}\right)\mathbf{UF},$$

where $\mathbf{L}$ and $\mathbf{U}$ are the block elimination matrices

$$\mathbf{L} = \mathtt{elim}(\mathbf{X}_{\mathrm{sr}}\mathbf{X}_{\mathrm{rr}}^{-1}, I_{\mathrm{s}}, I_{\mathrm{r}}, n) \qquad \text{and} \qquad \mathbf{U} = \mathtt{elim}(\mathbf{X}_{\mathrm{rr}}^{-1}\mathbf{X}_{\mathrm{rs}}, J_{\mathrm{r}}, J_{\mathrm{s}}, n).$$

4.4 Strong recursive skeletonization

In this section, we review the strong recursive skeletonization (SRS) algorithm of [76] that is used to compute an invertible factorization of a dense matrix involving a kernel matrix whose kernel represents physical interactions between a set of points in two or three dimensions. The decomposition is related to classical skeletonization as introduced in Section 4.3.3, however, the skeletonization is performed with "strong admissibility" condition, where only well-separated boxes are compressed as low-rank.

First, we discuss a hierarchical tree data structure which gives useful terminology for discussing the algorithm. In Section 4.4.2, we discuss the strong recursive skeletonization procedure for a single box of the tree, and in Section 4.4.3, we describe how the algorithm can be applied recursively in a traversal of the tree. Section 4.4.4 discusses how the factorization can be computed efficiently in circumstances where matrix entries of $\mathbf{A}$ can be accessed easily, setting the stage for computing the factorization in the black-box setting.

In the introduction of the ideas, we focus on the simple quasi-one dimensional domain of Figure 4.3, though the ideas are directly applicable to more complicated domains, as shown in Figure 4.6.

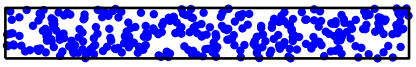
4.4.1 Hierarchical tree structure

SRS relies on a hierarchical partitioning of given points $\{\boldsymbol{x}_j\}_{j=1}^N$ into an quadtree or oct-tree. For the purposes of this discussion, we restrict our attention to uniform (e.g. fully populated) trees. Formally, we define a tree $\mathcal{T}$, in which each box b is associated with a subset of the given points. Initially, all points belong to the root node. Letting d denote the dimension, we split the root node into 2^d children nodes, and then split each child again until the size of each node is below some given threshold m. We refer to a node with children as a parent node, and a node with no children as a leaf node. The depth of a node is defined as its distance from the root node, and level ℓ of the tree is defined as the set of nodes with depth ℓ, so that level 0 consists of only the root node, level 1 consists of the 2^d children of the root node, and so on. The levels of the tree represent successively finer partitions of the points. The depth of the tree is defined as the maximum node depth, denoted by $L \approx \log_2(N/m)$. See Figure 4.6 for a hierarchical decomposition of points into a uniform quadtree.

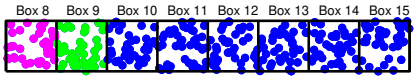
4.4.2 Strong skeletonization for a single box

Let us consider again the simple model problem on a quasi one dimensional domain in Figure 4.1 that we introduced in Section 4.2, where $\mathbf{A}$ is given by the evaluation of (4.1) on a set of points $\{\boldsymbol{x}_i\}_{i=1}^N$. Consider the tessellating the given points of Figure 4.3(a) into boxes $8, \ldots, 15$ as shown in Figure 4.3(b). The objective, in this section, is to describe the process of diagonalizing the interactions between a target box b and the rest of the points. Once this machinery is in place, the procedure can be applied successively to all boxes on a level.

Consider the target box 8, for concreteness. First, we would like to compress the interactions between box b and its far field. To this end, we split the set of points into three sets: The box itself which contains all points in the box to be compressed (box 8 in this case), the "near field set" which contains points in boxes directly adjacent to the active box (box 9 in this case), and the "far field set" which contains



(a) A rectangular domain contains the points $\{\boldsymbol{x}_j\}_{j=1}^N$.



(b) A tessellation of the points into boxes $8, \ldots 15$.

Figure 4.3: Figure 4.3a shows a rectangular domain with points $\{\boldsymbol{x}_j\}_{j=1}^N$, which is tessellated into boxes $8, \ldots 15$. The colors of Figure 4.3b indicate the index vectors defined in Section 4.4.2, with magenta for I_{a}, green for I_{n}, and blue for I_{f}.

everything else (boxes 10 through 15 in this case). These sets are identified through three index vectors:

I_{b}	The "target box". (While compressing box 8, $I_{\mathrm{b}} = I_8$.)
I_{n}	The "near field indices". (While compressing box 8, $I_{\mathrm{n}} = I_9$.)
I_{f}	The "far field indices". (While compressing box 8, $I_{\mathrm{f}} = [I_{10}, I_{11}, \ldots, I_{15}]$.)

Figure 4.3(b) illustrates these definitions. Setting $I = J = [I_{\mathrm{b}}, I_{\mathrm{n}}, I_{\mathrm{f}}]$, we now obtain a partition of $\mathbf{A}$ into the blocks

$$\mathbf{A}(I, J) = \left(\begin{array}{c|cc} \mathbf{A}_{\mathrm{bb}} & \mathbf{A}_{\mathrm{bn}} & \mathbf{A}_{\mathrm{bf}} \\ \hline \mathbf{A}_{\mathrm{nb}} & \mathbf{A}_{\mathrm{nn}} & \mathbf{A}_{\mathrm{nf}} \\ \mathbf{A}_{\mathrm{fb}} & \mathbf{A}_{\mathrm{fn}} & \mathbf{A}_{\mathrm{ff}} \end{array}\right). \tag{4.31}$$

(Comparing (4.31) to the tessellation (4.25) that is used in classical skeletonization, we see that the near field has been added. The purpose is to reduce the numerical ranks that arise.) We now invoke the well-known fact from the literature on fast summation schemes that all interactions between the active box and its far-field has low numerical rank. (To be precise, the rank is bounded by $O(\log(1/\varepsilon))$ where ε is the requested precision. This bound is independent of how many points are actually in the box.) Linear algebraically, this means that $\mathbf{A}_{\mathrm{bf}}$ and $\mathbf{A}_{\mathrm{fb}}$ are both of numerically low rank. We form the row and the column IDs, respectively, cf. (4.26) and (4.27),

$$\mathbf{A}_{\mathrm{bf}}([I_{\mathrm{r}}, I_{\mathrm{s}}], :) = \begin{pmatrix} \mathbf{A}_{\mathrm{rf}} \\ \mathbf{A}_{\mathrm{sf}} \end{pmatrix} = \begin{pmatrix} \mathbf{T}_{\mathrm{rs}} \mathbf{A}_{\mathrm{sf}} \\ \mathbf{A}_{\mathrm{sf}} \end{pmatrix} \tag{4.32}$$

and

$$\mathbf{A}_{\mathrm{fb}}(:, [J_{\mathrm{r}}, J_{\mathrm{s}}]) = \begin{bmatrix} \mathbf{A}_{\mathrm{fr}}, & \mathbf{A}_{\mathrm{fs}} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{\mathrm{fs}} \mathbf{T}_{\mathrm{sr}}, & \mathbf{A}_{\mathrm{fs}} \end{bmatrix}. \tag{4.33}$$

These factorizations partition the nodes in I_b into a set of "skeleton nodes" that represent all interactions between the box and the outside world, and a set of "residual nodes" that we will decouple from the rest of the system. To keep track of this, we update I and J to put the residual nodes first:

$$I = [I_\mathrm{r},\, I_\mathrm{s},\, I_\mathrm{n},\, I_\mathrm{f}], \qquad \text{and} \qquad J = [J_\mathrm{r},\, J_\mathrm{s},\, I_\mathrm{n},\, I_\mathrm{f}].$$

The corresponding tessellation of $\mathbf{A}$ becomes, cf. (4.28),

$$\mathbf{A}(I,J) = \left(\begin{array}{cc|cc} \mathbf{A}_{\mathrm{rr}} & \mathbf{A}_{\mathrm{rs}} & \mathbf{A}_{\mathrm{rn}} & \mathbf{A}_{\mathrm{rf}} \\ \mathbf{A}_{\mathrm{sr}} & \mathbf{A}_{\mathrm{ss}} & \mathbf{A}_{\mathrm{sn}} & \mathbf{A}_{\mathrm{sf}} \\ \hline \mathbf{A}_{\mathrm{nr}} & \mathbf{A}_{\mathrm{ns}} & \mathbf{A}_{\mathrm{nn}} & \mathbf{A}_{\mathrm{nf}} \\ \mathbf{A}_{\mathrm{fr}} & \mathbf{A}_{\mathrm{fs}} & \mathbf{A}_{\mathrm{fn}} & \mathbf{A}_{\mathrm{ff}} \end{array}\right) = \left(\begin{array}{cc|cc} \mathbf{A}_{\mathrm{rr}} & \mathbf{A}_{\mathrm{rs}} & \mathbf{A}_{\mathrm{rn}} & \mathbf{T}_{\mathrm{rs}}\mathbf{A}_{\mathrm{sf}} \\ \mathbf{A}_{\mathrm{sr}} & \mathbf{A}_{\mathrm{ss}} & \mathbf{A}_{\mathrm{sn}} & \mathbf{A}_{\mathrm{sf}} \\ \hline \mathbf{A}_{\mathrm{nr}} & \mathbf{A}_{\mathrm{ns}} & \mathbf{A}_{\mathrm{nn}} & \mathbf{A}_{\mathrm{nf}} \\ \mathbf{A}_{\mathrm{fs}}\mathbf{T}_{\mathrm{sr}} & \mathbf{A}_{\mathrm{fs}} & \mathbf{A}_{\mathrm{fn}} & \mathbf{A}_{\mathrm{ff}} \end{array}\right) \tag{4.34}$$

Eliminating the "fr" and the "rf" blocks from (4.34), we next get

$$\mathbf{A}(I,J) = \mathbf{E}\left(\begin{array}{cc|cc} \mathbf{X}_{\mathrm{rr}} & \mathbf{X}_{\mathrm{rs}} & \mathbf{X}_{\mathrm{rn}} & \mathbf{0} \\ \mathbf{X}_{\mathrm{sr}} & \mathbf{A}_{\mathrm{ss}} & \mathbf{A}_{\mathrm{sn}} & \mathbf{A}_{\mathrm{sf}} \\ \hline \mathbf{X}_{\mathrm{nr}} & \mathbf{A}_{\mathrm{ns}} & \mathbf{A}_{\mathrm{nn}} & \mathbf{A}_{\mathrm{nf}} \\ \mathbf{0} & \mathbf{A}_{\mathrm{fs}} & \mathbf{A}_{\mathrm{fn}} & \mathbf{A}_{\mathrm{ff}} \end{array}\right)\mathbf{F} \tag{4.35}$$

where

$$\mathbf{E} = \mathtt{elim}(\mathbf{T}_{\mathrm{rs}}, I_\mathrm{r}, I_\mathrm{s}, n) \qquad \text{and} \qquad \mathbf{F} = \mathtt{elim}(\mathbf{T}_{\mathrm{sr}}, J_\mathrm{s}, J_\mathrm{r}, n).$$

Then block Gaussian elimination yields the factorization

$$\mathbf{A}(I,J) = \mathbf{E}\mathbf{L}\left(\begin{array}{cc|cc} \mathbf{X}_{\mathrm{rr}} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{X}_{\mathrm{ss}} & \mathbf{X}_{\mathrm{sn}} & \mathbf{A}_{\mathrm{sf}} \\ \hline \mathbf{0} & \mathbf{X}_{\mathrm{ns}} & \mathbf{X}_{\mathrm{nn}} & \mathbf{A}_{\mathrm{nf}} \\ \mathbf{0} & \mathbf{A}_{\mathrm{fs}} & \mathbf{A}_{\mathrm{fn}} & \mathbf{A}_{\mathrm{ff}} \end{array}\right)\mathbf{U}\mathbf{F} \tag{4.36}$$

where $\mathbf{L}$ and $\mathbf{U}$ are the block elimination matrices

$$\mathbf{L} = \mathtt{elim}(\mathbf{X}_{\mathrm{sr}}\mathbf{X}_{\mathrm{rr}}^{-1}, I_\mathrm{s}, I_\mathrm{r}, n) \qquad \text{and} \qquad \mathbf{U} = \mathtt{elim}(\mathbf{X}_{\mathrm{rr}}^{-1}\mathbf{X}_{\mathrm{rs}}, J_\mathrm{r}, J_\mathrm{s}, n). \tag{4.37}$$

The equation (4.36) successfully decouples the residual nodes of target box b from the rest of the problem. Alternatively, the system can be written as

$$\widetilde{\mathbf{A}} = \mathbf{V}^{-1}\mathbf{A}(I,J)\mathbf{W}^{-1}, \qquad \mathbf{V} := \mathbf{E}\mathbf{L}, \qquad \mathbf{W} := \mathbf{U}\mathbf{F} \tag{4.38}$$

where the matrices $\mathbf{V}$ and $\mathbf{W}$ are defined as the left and right diagonalization operators, respectively, and the matrix $\widetilde{\mathbf{A}}$ is written explicitly in the equation (4.36). The redundant nodes of the target box are decoupled in equation (4.36), and the algorithm proceeds by repeating a similar procedure to the other boxes of the tree, but now for the modified system $\widetilde{\mathbf{A}}$.

4.4.3 Recursive Algorithm

In the previous section, we described the procedure for decoupling the redundant nodes of a target box from the rest of the system. Now, we describe how this procedure can be applied successively to all boxes on a level. Consider again the simple model of a quasi-one dimensional geometry in Figure 4.1. For target box 8, the left and right diagonalization operators $\mathbf{V}_8$ and $\mathbf{W}_8$ have the form described in Section 4.4.2. Applying these operators to the appropriate permutation of $\mathbf{A}$ yields the matrix $\widetilde{\mathbf{A}}$ defined in (4.38). For the sparsity pattern of $\widetilde{\mathbf{A}}$, see the leftmost panel of Figure 4.4, where the modified entries of $\widetilde{\mathbf{A}}$ are highlighted in red. The procedure can be repeated for target box 9 by computing left and right diagonalization operators $\mathbf{V}_9$ and $\mathbf{W}_9$. Note that instead of computing skeletons for the original matrix $\mathbf{A}$ as in equation (4.34), we instead need to compute skeletons which sparsify the modified matrix $\widetilde{\mathbf{A}}$. Applying these techniques to the boxes $8, \ldots, 15$ on Level 3 successively, we can compute a factorization, which (for an appropriate permutation) has the following form

$$\begin{pmatrix} \widetilde{\mathbf{A}}_{\mathrm{r}_8} & & & & \\ & \widetilde{\mathbf{A}}_{\mathrm{r}_9} & & & \\ & & \ddots & & \\ & & & \widetilde{\mathbf{A}}_{\mathrm{r}_{15}} & \\ & & & & \widetilde{\mathbf{A}}_{[\mathrm{s}_8,\ldots,\mathrm{s}_{15}]} \end{pmatrix} = \mathbf{V}_{15}^{-1} \cdots \mathbf{V}_8^{-1}\ \mathbf{A}\ \mathbf{W}_{15}^{-1} \cdots \mathbf{W}_8^{-1}, \tag{4.39}$$

where $\widetilde{\mathbf{A}}$ is a matrix of modified interactions. The redundant nodes I_{r} for every box $8, \ldots 15$ are decoupled from the rest of the system. The remaining nodes I_{s} of skeleton nodes for every box interact densely in the dense block $\widetilde{\mathbf{A}}_{[\mathrm{s}_8,\ldots,\mathrm{s}_{15}]}$. In the process

of computing the single-level factorization (4.39), the modified matrix of decoupled interactions need to be stored and updated. See Figure 4.4 for the sparsity pattern of decoupled interactions at various stages of computing the single-level factorization (4.39). Typically, strong recursive skeletonization is applied in circumstances where entries of $\mathbf{A}$ can be accessed in $\mathcal{O}(1)$ time, and the entire matrix of decoupled interactions does not need to be stored explicitly. Then, only the modified entries shown in red of Figure 4.4 need to be stored at every stage of the computation.

Since the residual degrees of freedom have been decoupled from the problem in (4.39), the task that remains is to solve the surviving linear system on the "skeleton nodes". To illustrate the structure of this problem, we show the coefficient matrix after the residual nodes have been dropped in Figure 4.5(a). For intermediate size problems, it is often possible to directly compute the LU decomposition of the linear system that connects the skeleton nodes, since the ranks tend to be small when strong admissibility it used. However, for large scale problems, there will be too many skeleton nodes surviving. In this case, we can continue the skeletonization process in a multilevel fashion. In order to reintroduce rank deficiencies in the off-diagonal blocks, we merge the boxes by twos to form larger boxes, as shown in 4.5(b). The idea is now to simply repeat the skeletonization process outlined in Section 4.4.2. This results in the further sparsified coefficient matrix shown in Figure 4.5(c).

We will shortly formalize the description of the recursive skeletonization process, but let us first briefly show what changes occur when the domain is "truly" two dimensional, as in Figure 4.6(a). It is now natural to organize the domain into a quadtree of boxes, rather than a binary tree, as shown in Figures 4.6(b) and 4.6(c). Other than that, the scheme proceeds just as it does for the quasi one dimensional example we discussed earlier. After all boxes at the finest level have been compressed, the remaining skeleton points are shown on top in Figure 4.7(a) in red, while the residual points are shown in gray. The matrix of interactions between the skeleton points is shown below. Observe that there are now many more updated (red) boxes than there were for the quasi one dimensional domain (as many as 25 in some rows). In

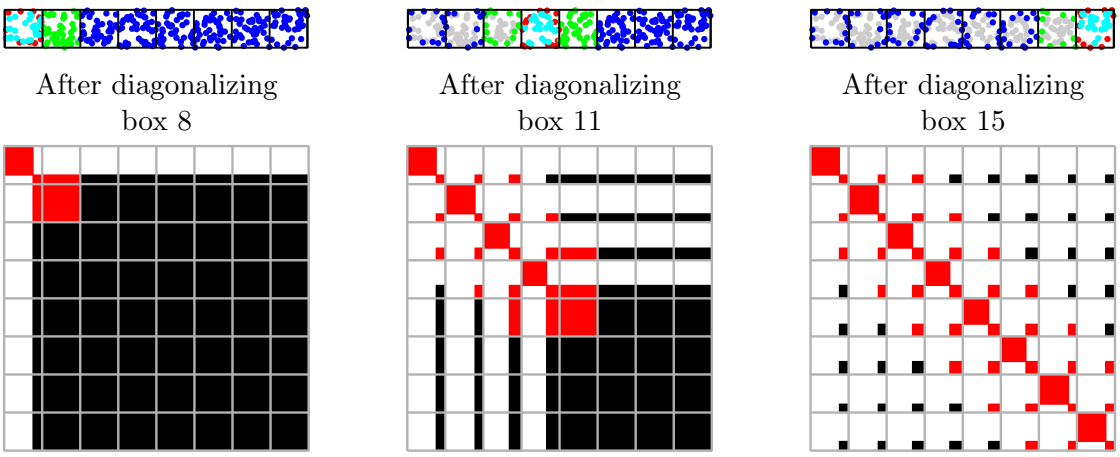


Figure 4.4: Illustration of the combined compression and factorization process described in Section 4.4.2. *Upper row:* The computational domain after steps 1, 4, and 8. The nodes are colored so that I_{s} is red, I_{r} is cyan, I_{n} is green, and I_{f} is blue. (The gray nodes mark points that were identified as "residual" nodes in the previous steps. These points have dropped out of the computation and play no active role.) *Lower row:* The sparsity pattern of the matrix $\widetilde{\mathbf{A}}$ after the residual nodes in Box 8/11/15 have been decoupled. White blocks are zeros, red blocks are entries that got modified, and black blocks are entries that have not been modified.

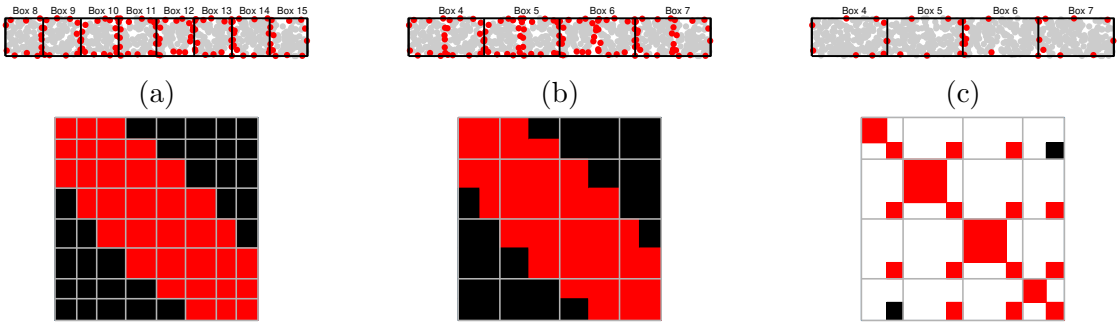
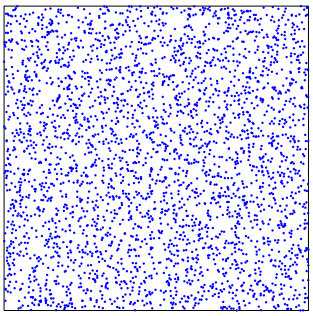


Figure 4.5: Illustration of the merging of boxes described in Section 4.4.3. (a) On top, the points remaining active after compression of the first level has completed. Below, the corresponding coefficient matrix, with entries that have been modified shown in red. (b) Boxes on the finest level have been merged by pairs to create larger boxes and reintroduce rank deficiencies in the off-diagonal blocks. (c) The geometry and the coefficient matrix after compression at the coarser level has completed.

order to reintroduce rank deficiencies in the off-diagonal blocks, we now merge boxes by sets of four to yield level 2, as illustrated in Figure 4.7(b). Once compression has

completed at level 2, very few points remain as skeleton nodes, as we see on top in Figure 4.7(c), with the very sparse coefficient matrix shown below.



(a) Computational domain.

43	44	47	48	59	60	63	64
42	45	46	49	58	61	62	65
39	40	51	52	55	56	67	68
38	41	50	53	54	57	66	69
27	28	31	32	75	76	79	80
26	29	30	33	74	77	78	81
23	24	35	36	71	72	83	84
22	25	34	37	70	73	82	85

(b) Ordering of boxes on level 3.

11	12	15	16
10	13	14	17
7	8	19	20
6	9	18	21

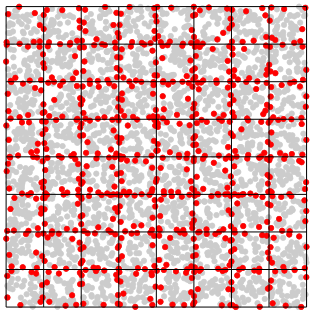
(c) Ordering of boxes on level 2.

Figure 4.6: The square computational domain described in Section 4.4.1. (a) The original domain, with the points $\{\boldsymbol{x}_i\}_{i=1}^N$ shown in blue. (b) The boxes at the finest level. (c) The merged boxes at the next coarser level.

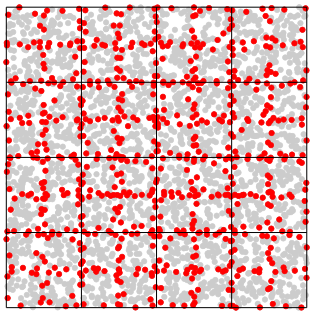
To formalize the description of SRS, consider that the boxes are diagonalized in an upward traversal through the tree in order $1, \ldots, n$. Then the decomposition takes the form

$$\widetilde{\mathbf{A}}_{\text{diag}} := \begin{pmatrix} \widetilde{\mathbf{A}}_{\text{r}_1} & & & & \\ & \widetilde{\mathbf{A}}_{\text{r}_2} & & & \\ & & \ddots & & \\ & & & \widetilde{\mathbf{A}}_{\text{r}_n} & \\ & & & & \widetilde{\mathbf{A}}_{[\text{S}]} \end{pmatrix} = \mathbf{V}_1^{-1} \cdots \mathbf{V}_n^{-1} \; \mathbf{A} \; \mathbf{W}_n^{-1} \cdots \mathbf{W}_1^{-1}, \tag{4.40}$$

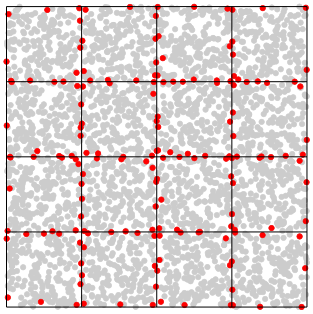
where [S] is the set of all remaining skeleton nodes at the time of terminating the algorithm (typically on level 2 of the tree). Because $\widetilde{\mathbf{A}}_{\text{diag}}$ is block-diagonal, it is easy to invert. The factorization (4.40) can also be used for a sparse factorization of $\mathbf{A}$. In the next section, we discuss techniques which allow for the efficient calculation of elimination matrices $\mathbf{E}$ and $\mathbf{F}$ of Section 4.4.2 in the setting of boundary integral equations.



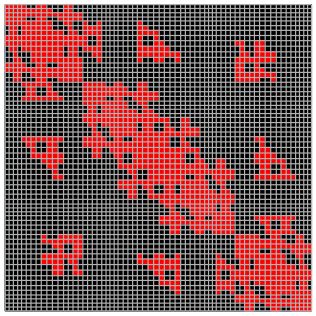
(a) Active points after level 3.



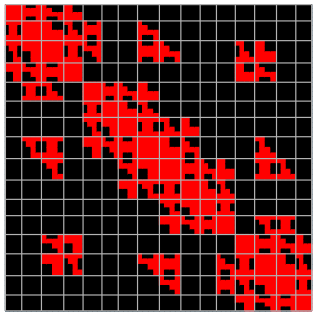
(b) Merge boxes by fours.



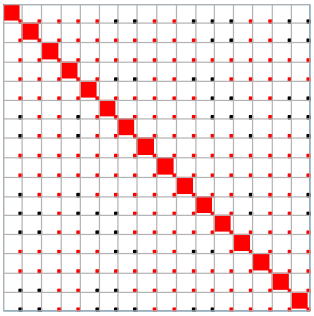
(c) Active points after level 2.



(d) Matrix after level 3.



(e) Matrix retessellated.



(f) Matrix after level 2.

Figure 4.7: The analog of Figure 4.5 when the computational domain is the box shown in Figure 4.1. Observe that many more blocks get updated for a true two-dimensional domain — up to 25 boxes per row need to be explicitly stored.

4.4.4 How to compress the far-field interactions

The aim is to compress the two off-diagonal blocks $\mathbf{A}_{\mathrm{b,f}}$ and $\mathbf{A}_{\mathrm{f,b}}$ using the column ID. Notice that the computational cost would be $\mathcal{O}(N)$ if the full matrix is formed, which turns out to be unnecessary, particularly when $\mathbf{A}$ arises from the discretization of a boundary integral equation.

The proxy method introduces points $\left\{\boldsymbol{x}_i^{(\mathrm{proxy})}\right\}_{i=1}^{n_p}$ which replicate the effect of far-field points I_{f}. Instead of using the matrix $\mathbf{A}_{\mathrm{b,f}}$, a smaller matrix with $\mathcal{O}(1)$ rows and columns is formed and compressed

$$\mathbf{A}_{\mathrm{proxy,b}} \qquad \text{where } (\mathbf{A}_{\mathrm{proxy,b}})_{i,j} = k\left(\boldsymbol{x}_i^{(\mathrm{proxy})}, \boldsymbol{x}[I_{\mathrm{b}_j}]\right). \tag{4.41}$$

With the appropriate choice of proxy surface, the matrix $\mathbf{A}_{\mathrm{proxy,b}}$ spans the row space

of $\mathbf{A}_{\mathrm{f,b}}$, and likewise, $\mathbf{A}_{\mathrm{b,proxy}}$ spans the column space of $\mathbf{A}_{\mathrm{b,f}}$ for *any* point distribution in the far-field. Therefore, the indices $I_b = I_{\mathrm{r}} \cup I_{\mathrm{s}}$ and interpolative matrix $\mathbf{T}$ computed by forming and factorizing the matrix (4.41) will satisfy (4.33). Because the matrix $\widetilde{\mathbf{A}}$ of diagonalized interactions does modify entries of $\mathbf{A}$, the proxy surface needs to be placed appropriately to compute the SRS factorization; this is discussed in further detail in [76, 86].

4.5 Randomized strong recursive skeletonization

In this section, we describe computing the SRS factorization described in Section 4.4 in the black-box setting where the matrix $\mathbf{A}$ is only accessed through its action on vectors. The algorithm relies on the randomized sampling machinery first introduced in Section 4.2. Recall that our setting is the following

$$\underset{N\times p}{\mathbf{Y}} = \underset{N\times N}{\mathbf{A}}\ \underset{N\times p}{\mathbf{\Omega}}, \qquad \underset{N\times p}{\mathbf{Z}} = \underset{N\times N}{\mathbf{A}^*}\ \underset{N\times p}{\mathbf{\Psi}}, \qquad \mathbf{\Omega}, \mathbf{\Psi} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \tag{4.42}$$

where the matrix $\mathbf{A}$ and its adjoint are sketched with Gaussian random matrices.

As in our description of SRS, we first describe the process of diagonalizing the interactions of a target box b with respect to the rest of the problem. Recall the notation used in Section 4.4.2 for a target box I_{b}, its near-field indices I_{n} and its far-field indices I_{f}.

In order to compute the skeletonization operators $\mathbf{E}$ and $\mathbf{F}$, we need to compute the column ID of the sketches

$$\underset{m\times k}{\mathbf{Y}'_{\mathrm{b}}} = \mathbf{A}_{\mathrm{bf}}\ \mathbf{\Omega}_{\mathrm{f}}, \qquad \underset{m\times k}{\mathbf{Z}'_{\mathrm{b}}} = \mathbf{A}^*_{\mathrm{fb}}\ \mathbf{\Psi}_{\mathrm{f}}, \tag{4.43}$$

respectively. The samples $\mathbf{Y}$ and $\mathbf{Z}$ in (4.42), however, include the contributions of the near field blocks I_{n}. The contribution of $\mathbf{\Omega}_{\mathrm{n}}$ and $\mathbf{\Psi}_{\mathrm{n}}$, however, can be "nullified" using block nullification as described in Section 4.2.2 and sketches (4.43) can be computed as

$$\underset{m\times k}{\mathbf{Y}'_{\mathrm{b}}} = \underset{m\times p}{\mathbf{Y}_b}\ \underset{p\times k}{\mathtt{null}\,(\mathbf{\Omega}_{\mathrm{n}})}, \qquad \underset{m\times k}{\mathbf{Z}'_{\mathrm{b}}} = \underset{m\times p}{\mathbf{Z}_{\mathrm{b}}}\ \underset{p\times k}{\mathtt{null}\,(\mathbf{\Psi}_{\mathrm{n}})}, \tag{4.44}$$

The sparsified matrix $\hat{\mathbf{A}} = \mathbf{E}^{-1}\mathbf{A}\mathbf{F}^{-1}$ has decoupled interactions between I_r and I_f, and the interactions between I_r and the rest of the system can be entirely decoupled using block-elimination matrices $\mathbf{L}, \mathbf{U}$ as defined in 4.37. Ideally, we would like to extract the modified interactions between $\mathbf{X}_{\mathrm{r,n}}$ and $\mathbf{X}_{\mathrm{n,r}}$ as well as $\mathbf{X}_{\mathrm{r,s}}$ and $\mathbf{X}_{\mathrm{s,r}}$ in equation (4.35). The samples $\{\mathbf{Y}, \mathbf{\Omega}, \mathbf{Z}, \mathbf{\Psi}\} \Rightarrow \{\hat{\mathbf{Y}}, \hat{\mathbf{\Omega}}, \hat{\mathbf{Z}}, \hat{\mathbf{\Psi}}\}$ can be updated to instead be samples of $\hat{\mathbf{A}}$.

$$\begin{aligned} \hat{\mathbf{Y}} &= \mathbf{E}\ \mathbf{Y}, \qquad & \hat{\mathbf{\Omega}} &= \mathbf{F}^{-1}\mathbf{\Omega} \\ \hat{\mathbf{Z}} &= \mathbf{F}^{*}\mathbf{Z}, \qquad & \hat{\mathbf{\Psi}} &= \mathbf{E}^{-*}\mathbf{\Psi} \end{aligned} \tag{4.45}$$

Then the modified interactions $\mathbf{X}_{\mathrm{r,n}}$ and $\mathbf{X}_{\mathrm{n,r}}$ can be extracted using block extraction, as discussed in Section 4.2.3 using the formulas

$$\mathbf{X}_{\mathrm{r,n}} \ = \ \hat{\mathbf{Y}}_\mathrm{r}\ \hat{\mathbf{\Omega}}_\mathrm{n}^{\dagger}, \qquad \mathbf{X}_{\mathrm{n,r}}^{*} \ = \ \hat{\mathbf{Z}}_\mathrm{r}\ \hat{\mathbf{\Psi}}_\mathrm{n}^{\dagger}. \tag{4.46}$$

After computing the block elimination matrices $\mathbf{L}$ and $\mathbf{U}$, the random sketches can again to updated to be instead of the matrix $\widetilde{\mathbf{A}} = \mathbf{L}^{-1}\hat{\mathbf{A}}\mathbf{U}^{-1}$ by updating the sketches $\{\hat{\mathbf{Y}}, \hat{\mathbf{\Omega}}, \hat{\mathbf{Z}}, \hat{\mathbf{\Psi}}\}$ to produce sketches $\{\widetilde{\mathbf{Y}}, \widetilde{\mathbf{\Omega}}, \widetilde{\mathbf{Z}}, \widetilde{\mathbf{\Psi}}\}$.

$$\begin{aligned} \widetilde{\mathbf{Y}} &= \mathbf{L}\ \hat{\mathbf{Y}}, \qquad & \widetilde{\mathbf{\Omega}} &= \mathbf{U}^{-1}\hat{\mathbf{\Omega}} \\ \widetilde{\mathbf{Z}} &= \mathbf{U}^{*}\hat{\mathbf{Z}}, \qquad & \widetilde{\mathbf{\Psi}} &= \mathbf{L}^{-*}\hat{\mathbf{\Psi}} \end{aligned} \tag{4.47}$$

After completing this process for a single box, the procedure can be repeated for a sequence of boxes. Over the coarse of the algorithm, the samples do not need to be redrawn, instead they are updated using the computed factorization at every stage. Because the algorithm is multi-level, compression errors do propagate from one level to the next. To handle this issue, the absolute tolerance for the compression stage is successively relaxed at every level of the computation. The matrix $\widetilde{\mathbf{A}}$ in equation (4.40) is block-diagonal to some accuracy which may deviate slightly from the desired compression tolerance. To estimate the extent to which our compression is successful for a particular level, we use randomized sampling techniques (e.g. block nullification) to estimate the extent to which $\widetilde{\mathbf{A}}_{\mathrm{r,f}}$ and $\widetilde{\mathbf{A}}_{\mathrm{f,r}}$ deviate from desired compression tolerance, where in this case $I_\mathrm{f} = (1:N) \setminus I_r$, for every box.

4.6 Numerical experiments

In this section, we illustrate the performance of the proposed algorithm. For the example in Section 4.6.1, we consider a sparse Schur complement, which arises in the context of sparse direct solvers. Sparse Schur complements are a composition of sparse matrix operations; they can be applied fast to vectors, however, matrix entries are challenging to access efficiently.

N	number of points
m	leaf size of tree
atol	absolute compression tolerance at the leaf level
n_{samples}	number of samples of $\mathbf{A}$ and $\mathbf{A}^*$
$T_{\text{factorize}}$	time for algorithm in Section 4.5
M	memory needed for $\mathbf{V}_i$ and $\mathbf{W}_i$ for all boxes
relerr errsolve	defined in equation (4.48)

Table 4.1: Notation for reported numerical results.

The RSRS algorithm is operates on a uniform quad-tree $\mathcal{T}$ which is partitioned to have at most m points in the leaf boxes. The user provides a parameter atol which dictates the absolute tolerance of compression at the leaf level; the compression rank k is chosen for each box adaptively. The success of the algorithm is measured by accessing the relative error of the computed factorization, as well as the error in the computed inverse. For the computed invertible factorization $\mathbf{K} \approx \mathbf{A}$, we report

$$\text{relerr} = \frac{\|\mathbf{A} - \mathbf{K}\|_2}{\|\mathbf{A}\|_2}, \qquad \text{errsolve} = \|\mathbf{I} - \mathbf{K}^{-1}\mathbf{A}\|_2. \tag{4.48}$$

Typical choices of user parameter atol may be 10^{-5} or even as large as 10^{-2}; the resulting errors (4.48) in the factorization are often much smaller, as we will demonstrate in the numerical results. Table 4.1 summarizes the notation used to report the numerical results.

4.6.1 3D Sparse Direct Solvers

Consider a boundary value problem of the form

$$\begin{cases} -\Delta u(x) = f(x), & x \in \Omega, \\ u(x) = g(x), & x \in \Gamma, \end{cases} \tag{4.49}$$

where $\mathcal{A}$ is a second order elliptic differential operator, and Ω is a rectangular domain in three dimensions with boundary Γ. Upon discretizing (4.49) with 2nd order finite differences, one obtains a linear system

$$\mathbf{Au} = \mathbf{f},$$

involving a sparse coefficient matrix. Sparse direct solvers produce an invertible factorization of $\mathbf{A}$ by leveraging sparsity, as well as $\mathcal{H}$-matrix structure when appropriate [68, ch. 20,21]. Often, it is useful to decompose the domain for the purposes of parallelizing the computation. The typical choice of domain decomposition is a hierarchical quad-tree or oct-tree, but a decomposition into thin slab subdomains is also possible and advantageous for ease of parallelization. This domain decomposition is employed in SlabLU, which is a simplified two-level sparse direct solver [97]. The corresponding domain decomposition is shown in Figure 4.8.

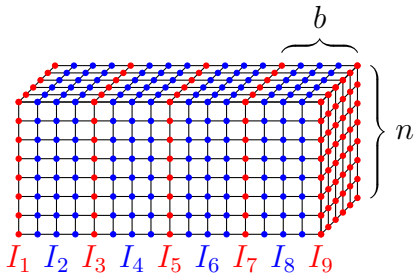


Figure 4.8: Domain decomposition used in SlabLU. The even-numbered nodes correspond to the nodes interior to each subdomain. The odd-numbered nodes correspond to interfaces between slabs. The slab partitioning is chosen so that interactions between slab interiors are zero. The slabs have width of b points.

To be precise, SlabLU uses a decomposition of the domain into elongated "slab" subdomains of dimensions $n \times n \times b$ where b is the slab thickness. The odd-numbered nodes correspond to slab interface nodes and the even-numbered nodes

correspond to slab interior nodes. With the slab decomposition, the linear system (4.6.1) has the block form

$$\begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \dots \\ \mathbf{A}_{21} & \mathbf{A}_{22} & \mathbf{A}_{23} & \mathbf{0} & \mathbf{0} & \dots \\ \mathbf{0} & \mathbf{A}_{32} & \mathbf{A}_{33} & \mathbf{A}_{34} & \mathbf{0} & \dots \\ \mathbf{0} & \mathbf{0} & \mathbf{A}_{43} & \mathbf{A}_{44} & \mathbf{A}_{45} & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} \mathbf{u}_1 \\ \mathbf{u}_2 \\ \mathbf{u}_3 \\ \mathbf{u}_4 \\ \vdots \end{bmatrix} = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \mathbf{f}_3 \\ \mathbf{f}_4 \\ \vdots \end{bmatrix}. \tag{4.50}$$

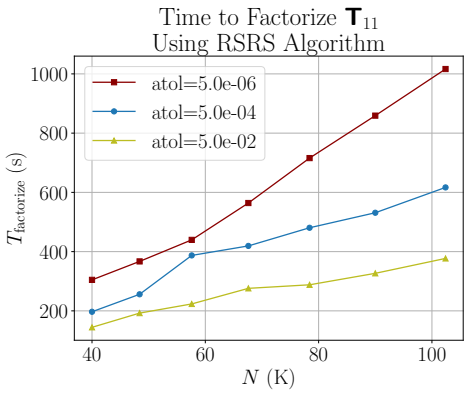
The system (4.50) can be factorized by first eliminating the nodes internal to each slab are eliminated in parallel by computing sparse direct factorizations $\mathbf{A}_{22}^{-1}, \mathbf{A}_{44}^{-1}, \dots$ in parallel. The remaining system has sparse Schur complements of the form

$$\mathbf{T}_{11} = \mathbf{A}_{11} - \mathbf{A}_{12}\ \mathbf{A}_{22}^{-1}\ \mathbf{A}_{21}. \tag{4.51}$$

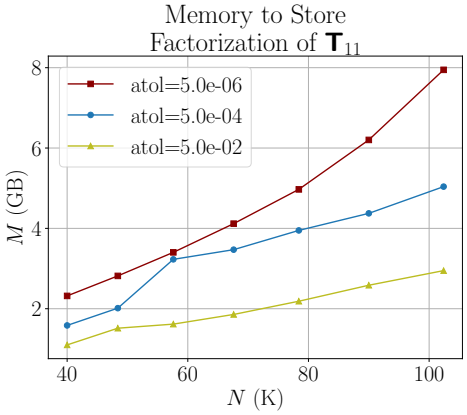
Owing to sparsity, $\mathbf{T}_{11}$ and its transpose can be rapidly applied to random vectors

$$\mathbf{Y} = \mathbf{T}_{11}\ \mathbf{\Omega}, \qquad \mathbf{Z} = \mathbf{T}_{11}^{*}\ \mathbf{\Psi} \tag{4.52}$$

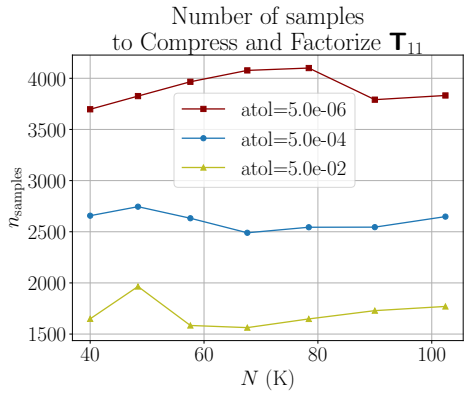
Figure 4.9 reports experiments compressing and factorizing $\mathbf{T}_{11}$ using the RSRS algorithm which only accesses the matrix through its action on vectors. The PDE is Poisson, and the slab thickness is fixed as $b = 10$ for various problem sizes $N = n^2$.



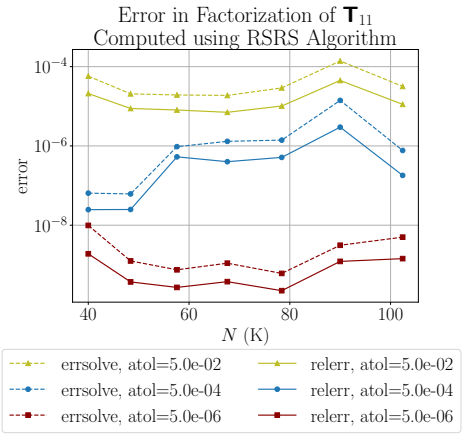
(a) Time to compute $\mathbf{T}_{11}^{-1}$ scales linearly.



(b) Memory to store $\mathbf{T}_{11}^{-1}$ scales linearly.



(c) Number of samples needed is independent of N.



(d) The computed factorization is accurate and does not deteriorate with increasing N.

Figure 4.9: The figures above show the effectiveness of using RSRS on a sparse Schur complement $\mathbf{T}_{11}$ which arises while computing sparse direct solvers for discretized PDEs. The number of samples needed to construct a factorization $\mathbf{T}_{11}^{-1}$ is constant as N increases and instead depends only on the desired user tolerance as well as the leaf size m. For atol=5e-2 and 5e-4, the leaf size m is set as $m = 200$. For atol=5e-6, the leaf size m is set as $m = 350$.

4.7 Conclusions

The manuscript introduces the algorithm Randomized Strong Recursive Skeletonization (RSRS) for simultaneously compressing and inverting an $\mathcal{H}^2$-matrix, given a means of applying the matrix and its adjoint to vectors. RSRS is immediately ap-

plicable in a range of important environments, and the numerical results demonstrate the effectiveness of the approach on Schur complements which arise while factorizing a sparse direct solver. It extends the range of the SRS algorithm using tailored approaches for randomized linear algebra applied to rank structured matrices.

Acknowledgments

The work reported was supported by the Office of Naval Research (N00014-18-1-2354), by the National Science Foundation (DMS-1952735, DMS-2012606, and DMS-2313434), and by the Department of Energy ASCR (DE-SC0022251).

Chapter 5: Conclusion

This thesis has explored innovative techniques for efficiently and robustly computing approximate solutions to elliptic partial differential equations (PDEs). The central theme throughout this work has been the use of hierarchical matrices ($\mathcal{H}$-matrices) to accelerate key computational operations. The work presents novel algorithms for solving linear systems that exploit randomized methods in linear algebra to attain high computational efficiency and scalability. The algorithms have been designed to take advantage of various compute kernels, including vectorization and specialized hardware acceleration features found in modern architectures.

This thesis has made three significant contributions. Firstly, it introduced a novel fast multipole method (FMM) based on $\mathcal{H}$-matrices, offering a simplified data structure compared to the original FMM. Secondly, a sparse direct solver for discretized PDEs was presented, addressing the issue of dense fill-in in LU and Cholesky factorizations by exploiting $\mathcal{H}$-matrix structure in dense blocks. The use of the slab decomposition of the computational domain enhances parallelization and facilitates GPU acceleration. Lastly, the thesis introduced a novel algorithm for the simultaneous compression and LU factorization of a specific class of $\mathcal{H}$-matrices, providing significant simplifications and accelerations with applications in boundary integral equations and sparse direct solvers for discretized PDEs. These contributions collectively offer valuable insights into re-envisioning classical linear solvers in the evolving hardware landscape.

Appendix A: Rank Properties

A.1 Rank Property of Thin Slabs

In this appendix, we prove Proposition 3.3.2, which makes a claim on the rank structure of $\mathbf{T}_{11}$, defined in (3.17). [Rank Property] Let J_B be a contiguous set of points on the slab interface J, and let J_F be the rest of the points $J_F = J_1 \setminus J_B$. The sub-matrices $(\mathbf{T}_{11})_{BF}$, $(\mathbf{T}_{11})_{FB}$ have exact rank at most $2b$.

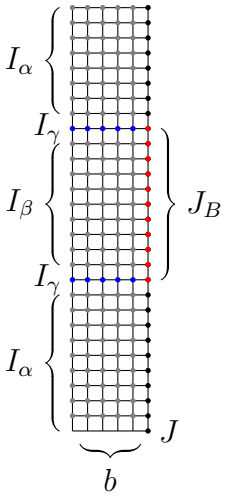


Figure A.1: To assist in the proof of Proposition 3.3.2, we define a partitioning of the slab interface $J = J_B \cup J_F$, where $J_F = J \setminus J_B$. We also partition the slab interior nodes into $I_\alpha \cup I_\beta \cup I_\gamma$.

Recall that $\mathbf{T}_{11} = \mathbf{A}_{11} - \mathbf{A}_{12}\mathbf{A}_{22}^{-1}\mathbf{A}_{21}$. The proof relies on the sparsity structure of the matrices in the Schur complement. As stated in the proposition, the slab interface I_1 is partitioned into indices I_B and I_F. The proof relies on partitioning I_2 as well, into the indices $I_\alpha, I_\beta, I_\gamma$ shown in Figure A.1, where $|I_\gamma| = 2b$.

The matrix $\mathbf{A}_{2,2}$ is sparse and can be factorized as

$$\mathbf{A}_{2,2} = \mathbf{L}_{2,2}\mathbf{U}_{2,2} := \begin{bmatrix} \mathbf{L}_{\alpha\alpha} & & \\ & \mathbf{L}_{\beta,\beta} & \\ \mathbf{L}_{\gamma,\alpha} & \mathbf{L}_{\gamma,\beta} & \mathbf{L}_{\gamma,\gamma} \end{bmatrix} \begin{bmatrix} \mathbf{U}_{\alpha,\alpha} & & \mathbf{U}_{\alpha,\gamma} \\ & \mathbf{U}_{\beta,\beta} & \mathbf{U}_{\beta,\gamma} \\ & & \mathbf{U}_{\gamma,\gamma} \end{bmatrix} \tag{A.1}$$

The formula for $(\mathbf{T}_{1,1})_{F,B}$ can be re-written as

$$(\mathbf{T}_{1,1})_{F,B} = \mathbf{A}_{F,B} - \left(\mathbf{A}_{F,2}\mathbf{U}_{2,2}^{-1}\right)\left(\mathbf{L}_{2,2}^{-1}\mathbf{A}_{2,B}\right) := \mathbf{A}_{F,B} - \mathbf{X}_{F,2}\ \mathbf{Y}_{2,B} \tag{A.2}$$

The factors $\mathbf{X}_{F,2}$ and $\mathbf{Y}_{2,B}$ have sparse structure, due the sparsity in the factorization (A.1) and the sparsity of $\mathbf{A}_{F,2}$ and $\mathbf{A}_{2,B}$.

$$\mathbf{X}_{F,2} = \begin{bmatrix}\mathbf{A}_{F,\alpha} & \mathbf{0} & \mathbf{A}_{F,\gamma}\end{bmatrix} \mathbf{U}_{22}^{-1}, \qquad \mathbf{Y}_{2,B} = \mathbf{L}_{22}^{-1} \begin{bmatrix}\mathbf{0} \\ \mathbf{A}_{\beta,B} \\ \mathbf{A}_{\gamma,B}\end{bmatrix} \tag{A.3}$$

The factors $\mathbf{X}_{F,2}$ and $\mathbf{Y}_{2,B}$ have the same sparsity pattern as $\mathbf{A}_{F,2}$ and $\mathbf{A}_{2,B}$, respectively. As a result,

$$(\mathbf{T}_{1,1})_{F,B} = \mathbf{A}_{F,B} - \begin{bmatrix}\mathbf{X}_{F,\alpha} & \mathbf{0} & \mathbf{X}_{F,\gamma}\end{bmatrix} \begin{bmatrix}\mathbf{0} \\ \mathbf{Y}_{\beta,B} \\ \mathbf{Y}_{\gamma,B}\end{bmatrix} = \underset{\text{sparse, } O(1) \text{ entries}}{\mathbf{A}_{F,B}} - \underset{\text{exact rank } 2b}{\mathbf{X}_{F,\gamma}\mathbf{Y}_{\gamma,B}}\,. \tag{A.4}$$

Similar reasoning can be used to show the result for $(\mathbf{T}_{11})_{B,F}$.

Works Cited

[1] Ahmad Abdelfattah, Timothy Costa, Jack Dongarra, Mark Gates, Azzam Haidar, Sven Hammarling, Nicholas J Higham, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, et al. A set of batched basic linear algebra subprograms and lapack routines. *ACM Transactions on Mathematical Software (TOMS)*, 47 (3):1–23, 2021.

[2] Ahmad Abdelfattah, Pieter Ghysels, Wajih Boukaram, Stanimire Tomov, Xiaoye Sherry Li, and Jack Dongarra. Addressing irregular patterns of matrix computations on GPUs and their impact on applications powered by sparse direct solvers. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2022.

[3] Patrick Amestoy, Alfredo Buttari, Jean-Yves l'Excellent, and Theo Mary. On the complexity of the block low-rank multifrontal factorization. *SIAM Journal on Scientific Computing*, 39(4):A1710–A1740, 2017.

[4] Patrick R Amestoy, Timothy A Davis, and Iain S Duff. An approximate minimum degree ordering algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[5] Patrick R Amestoy, Alfredo Buttari, Jean-Yves L'excellent, and Theo Mary. Performance and scalability of the block low-rank multifrontal factorization on multicore architectures. *ACM Transactions on Mathematical Software (TOMS)*, 45(1):1–26, 2019.

[6] Tracy Babb, Adrianna Gillman, Sijia Hao, and Per-Gunnar Martinsson. An accelerated Poisson solver based on multidomain spectral discretization. *BIT Numerical Mathematics*, 58:851–879, 2018.

[7] Ivo M Babuska and Stefan A Sauter. Is the pollution effect of the FEM avoidable for the Helmholtz equation considering high wave numbers? *SIAM Journal on numerical analysis*, 34(6):2392–2423, 1997.

[8] Jonas Ballani and Daniel Kressner. *Matrices with Hierarchical Low-Rank Structures*, pages 161–209. Springer International Publishing, Cham, 2016. ISBN 978-3-319-49887-4. doi: 10.1007/978-3-319-49887-4_3. URL `https://doi.org/10.1007/978-3-319-49887-4_3`.

[9] Mario Bebendorf. *Hierarchical matrices*, volume 63 of *Lecture Notes in Computational Science and Engineering*. Springer-Verlag, Berlin, 2008. ISBN 978-3-540-77146-3. A means to efficiently solve elliptic boundary value problems.

[10] Hadrien Bériot, Albert Prinn, and Gwénaël Gabard. Efficient implementation of high-order finite elements for Helmholtz problems. *International Journal for Numerical Methods in Engineering*, 106(3):213–240, 2016.

[11] Matthias Bollhöfer, Olaf Schenk, Radim Janalik, Steve Hamm, and Kiran Gullapalli. State-of-the-art sparse direct solvers. In *Parallel Algorithms in Computational Science and Engineering*, pages 3–33. Springer, 2020.

[12] Steffen Börm. *Efficient numerical methods for non-local operators*, volume 14 of *EMS Tracts in Mathematics*. European Mathematical Society (EMS), Zürich, 2010. ISBN 978-3-03719-091-3. $\mathcal{H}^2$-matrix compression, algorithms and analysis.

[13] Wajih Boukaram, George Turkiyyah, and David Keyes. Hierarchical matrix operations on GPUs: Matrix-vector multiplication and compression. *ACM Transactions on Mathematical Software (TOMS)*, 45(1):1–28, 2019.

[14] Pablo D Brubeck and Patrick E Farrell. A scalable and robust vertex-star relaxation for high-order FEM. *arXiv preprint arXiv:2107.14758*, 2021.

[15] 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 Journal on Matrix Analysis and Applications*, 41(2):715–746, 2020.

[16] Luiz Mariano Carvalho, Luc Giraud, and Patrick Le Tallec. Algebraic two-level preconditioners for the Schur complement method. *SIAM Journal on Scientific Computing*, 22(6):1987–2005, 2001.

[17] Tony F Chan and Tarek P Mathew. Domain decomposition algorithms. *Acta numerica*, 3:61–143, 1994.

[18] Gustavo Chávez, George Turkiyyah, Stefano Zampini, Hatem Ltaief, and David Keyes. Accelerated cyclic reduction: A distributed-memory fast solver for structured linear systems. *Parallel Computing*, 74:65–83, 2018.

[19] Chao Chen and Per-Gunnar Martinsson. Solving Linear Systems on a GPU With Hierarchically Off-Diagonal Low-Rank Approximations, 2022. URL `https://arxiv.org/abs/2208.06290`.

[20] Chao Chen, Sylvie Aubry, Tomas Oppelstrup, A Arsenlis, and Eric Darve. Fast algorithms for evaluating the stress field of dislocation lines in anisotropic elastic media. *Modelling and Simulation in Materials Science and Engineering*, 2018.

[21] Hongwei Cheng, Zydrunas Gimbutas, Per-Gunnar Martinsson, and Vladimir Rokhlin. On the compression of low rank matrices. *SIAM Journal on Scientific Computing*, 26(4):1389–1404, 2005.

[22] Hongwei Cheng, William Y Crutchfield, Zydrunas Gimbutas, Leslie F Greengard, J Frank Ethridge, Jingfang Huang, Vladimir Rokhlin, Norman Yarvin, and Junsheng Zhao. A wideband fast multipole method for the Helmholtz equation in three dimensions. *Journal of Computational Physics*, 216(1):300–325, 2006.

[23] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE computational science and engineering*, 5 (1):46–55, 1998.

[24] Timothy A Davis. *Direct methods for sparse linear systems*, volume 2. Siam, 2006.

[25] Timothy A. Davis, Sivasankaran Rajamanickam, and Wissam M. Sid-Lakhdar. A survey of direct methods for sparse linear systems. *Acta Numerica*, 25:383 – 566, 2016. doi: 10.1017/S0962492916000076.

[26] Arnaud Deraemaeker, Ivo Babuška, and Philippe Bouillard. Dispersion and pollution of the FEM solution for the Helmholtz equation in one, two and three dimensions. *International journal for numerical methods in engineering*, 46(4): 471–499, 1999.

[27] Yijun Dong and Per-Gunnar Martinsson. Simpler is better: a comparative study of randomized pivoting algorithms for CUR and interpolative decompositions. *Advances in Computational Mathematics*, 49, 08 2023. doi: 10.1007/s10444-023-10061-z.

[28] Jack Dongarra, Sven Hammarling, Nicholas J Higham, Samuel D Relton, Pedro Valero-Lara, and Mawussi Zounon. The design and performance of batched BLAS on modern high-performance computing systems. *Procedia Computer Science*, 108:495–504, 2017.

[29] Jack J Dongarra and Stanimire Tomov. Matrix algebra for GPU and multicore architectures (MAGMA) for large petascale systems. Technical report, Univ. of Tennessee, Knoxville, TN (United States), 2014.

[30] Jack J Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain S Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 16(1):1–17, 1990.

[31] I.S. Duff, A.M. Erisman, and J.K. Reid. *Direct Methods for Sparse Matrices*. Oxford, 1989.

[32] A Dutt, M Gu, and V Rokhlin. Fast algorithms for polynomial interpolation, integration, and differentiation. *SIAM Journal on Numerical Analysis*, 33(5): 1689–1711, 1996.

[33] Björn Engquist and Lexing Ying. Sweeping preconditioner for the Helmholtz equation: hierarchical matrix representation. *Communications on pure and applied mathematics*, 64(5):697–735, 2011.

[34] Oliver G Ernst and Martin J Gander. Why it is difficult to solve Helmholtz problems with classical iterative methods. *Numerical analysis of multiscale problems*, pages 325–363, 2012.

[35] William Fong and Eric Darve. The black-box fast multipole method. *Journal of Computational Physics*, 228(23):8712–8725, 2009.

[36] Daniel Fortunato, Nicholas Hale, and Alex Townsend. The ultraspherical spectral element method. *Journal of Computational Physics*, 436:110087, 2021.

[37] Yuhong Fu and Gregory J Rodin. Fast solution method for three-dimensional Stokesian many-particle problems. *International Journal for Numerical Methods in Biomedical Engineering*, 16(2):145–149, 2000.

[38] Yuhong Fu, Kenneth J Klimkowski, Gregory J Rodin, Emery Berger, James C Browne, Jürgen K Singer, Robert A Van De Geijn, and Kumar S Vemaganti. A fast solution method for three-dimensional many-particle problems of linear elasticity. *International Journal for Numerical Methods in Engineering*, 42(7): 1215–1229, 1998.

[39] Martin J Gander and Hui Zhang. Restrictions on the use of sweeping type preconditioners for Helmholtz problems. In *International Conference on Domain Decomposition Methods*, pages 321–332. Springer, 2017.

[40] A. George. Nested dissection of a regular finite element mesh. *SIAM J. on Numerical Analysis*, 10:345–363, 1973.

[41] P. Ghysels, X. Li, F. Rouet, S. Williams, and A. Napov. An Efficient Multicore Implementation of a Novel HSS-Structured Multifrontal Solver Using Randomized Sampling. *SIAM Journal on Scientific Computing*, 38(5):S358–S384, 2016. doi: 10.1137/15M1010117. URL `https://doi.org/10.1137/15M1010117`.

[42] Pieter Ghysels and Ryan Synk. High performance sparse multifrontal solvers on modern GPUs. *Parallel Computing*, 110:102897, 2022.

[43] A Gillman, S Hao, and PG Martinsson. Short note: A simplified technique for the efficient and highly accurate discretization of boundary integral equations in 2d on domains with corners. *Journal of Computational Physics*, 256:214–219, 2014.

[44] Adrianna Gillman and Per-Gunnar Martinsson. A direct solver with O(N) complexity for variable coefficient elliptic PDEs discretized via a high-order composite spectral collocation method. *SIAM Journal on Scientific Computing*, 36(4):A2023–A2046, 2014.

[45] Adrianna Gillman, Patrick Young, and Per-Gunnar Martinsson. A direct solver $o(n)$ complexity for integral equations on one-dimensional domains. *Frontiers of Mathematics in China*, 7:217–247, 2012. ISSN 1673-3452. URL `http://dx.doi.org/10.1007/s11464-012-0188-3`. 10.1007/s11464-012-0188-3.

[46] Adrianna Gillman, AlexH. Barnett, and Per-Gunnar Martinsson. A spectrally accurate direct solution technique for frequency-domain scattering problems with variable media. *BIT Numerical Mathematics*, 55(1):141–170, 2015. ISSN 0006-3835. doi: 10.1007/s10543-014-0499-8. URL `http://dx.doi.org/10.1007/s10543-014-0499-8`.

[47] Zydrunas Gimbutas and Vladimir Rokhlin. A generalized fast multipole method for nonoscillatory kernels. *SIAM Journal on Scientific Computing*, 24(3):796–817, 2003.

[48] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *J. Comput. Phys.*, 73(2):325–348, 1987. ISSN 0021-9991.

[49] Leslie Greengard. *The rapid evaluation of potential fields in particle systems*. MIT press, 1988.

[50] Leslie Greengard and Vladimir Rokhlin. A new version of the fast multipole method for the Laplace equation in three dimensions. *Acta numerica*, 6:229–269, 1997.

[51] Leslie F Greengard and Jingfang Huang. A new version of the fast multipole method for screened Coulomb interactions in three dimensions. *Journal of Computational Physics*, 180(2):642–658, 2002.

[52] Ming Gu and Stanley C Eisenstat. Efficient algorithms for computing a strong rank-revealing QR factorization. *SIAM Journal on Scientific Computing*, 17 (4):848–869, 1996.

[53] Wolfgang Hackbusch. *Hierarchical matrices: algorithms and analysis*, volume 49. Springer, 2015.

[54] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53(2):217–288, 2011.

[55] Sijia Hao and Per-Gunnar Martinsson. A direct solver for elliptic PDEs in three dimensions based on hierarchical merging of Poincaré–Steklov operators. *Journal of Computational and Applied Mathematics*, 308:419–434, 2016.

[56] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, et al. Array programming with NumPy. *Nature*, 585(7825): 357–362, 2020.

[57] Kenneth L Ho and Lexing Ying. Hierarchical interpolative factorization for elliptic operators: integral equations. *Comm. Pure Appl. Math*, 69(7):1314–1353, 2016.

[58] K.L. Ho and L. Greengard. A fast direct solver for structured linear systems by recursive skeletonization. *SIAM Journal on Scientific Computing*, 34(5): 2507–2532, 2012.

[59] James Levitt. *Building rank-revealing factorizations with randomization*. PhD thesis, University of Texas at Austin, 2022.

[60] James Levitt and Per-Gunnar Martinsson. Linear-Complexity Black-Box Randomized Compression of Hierarchically Block Separable Matrices, 2022. URL `https://arxiv.org/abs/2205.02990`.

[61] Xiaoye S Li and Meiyue Shao. A supernodal approach to incomplete LU factorization with partial pivoting. *ACM Transactions on Mathematical Software (TOMS)*, 37(4):1–20, 2011.

[62] 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. USA*, 104(51):20167–20172, 2007. ISSN 1091-6490. doi: 10.1073/pnas.0709640104. URL `http://dx.doi.org/10.1073/pnas.0709640104`.

[63] L. Lin, J. Lu, and L. Ying. Fast construction of hierarchical matrix representation from matrix-vector multiplication. *Journal of Computational Physics*, 230

(10):4071 – 4087, 2011. ISSN 0021-9991. doi: 10.1016/j.jcp.2011.02.033. URL http://www.sciencedirect.com/science/article/pii/S0021999111001227.

[64] Dhairya Malhotra and George Biros. PVFMM: A parallel kernel independent FMM for particle and volume potentials. *Communications in Computational Physics*, 18(3):808–830, 2015.

[65] Per-Gunnar Martinsson. A fast direct solver for a class of elliptic partial differential equations. *J. Sci. Comput.*, 38(3):316–330, 2009. ISSN 0885-7474. doi: 10.1007/s10915-008-9240-6. URL http://dx.doi.org/10.1007/s10915-008-9240-6.

[66] Per-Gunnar Martinsson. A direct solver for variable coefficient elliptic PDEs discretized via a composite spectral collocation method. *Journal of Computational Physics*, 242:460–479, 2013.

[67] Per-Gunnar Martinsson. Compressing rank-structured matrices via randomized sampling. *SIAM Journal on Scientific Computing*, 38(4):A1959–A1986, 2016.

[68] Per-Gunnar Martinsson. *Fast direct solvers for elliptic PDEs*. SIAM, 2019.

[69] Per-Gunnar Martinsson and Vladimir Rokhlin. A fast direct solver for boundary integral equations in two dimensions. *Journal of Computational Physics*, 205(1):1–23, 2005.

[70] Per-Gunnar Martinsson and Vladimir Rokhlin. An accelerated kernel-independent fast multipole method in one dimension. *SIAM Journal on Scientific Computing*, 29(3):1160–1178, 2007.

[71] Per-Gunnar Martinsson and Joel A. Tropp. Randomized numerical linear algebra: Foundations and algorithms. *Acta Numerica*, 29:403–572, 2020. doi: 10.1017/S0962492920000021.

[72] Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert. A randomized algorithm for the decomposition of matrices. *Appl. Comput. Harmon. Anal.*, 30(1):47–68, 2011. ISSN 1063-5203. doi: 10.1016/j.acha.2010.02.003. URL `http://dx.doi.org/10.1016/j.acha.2010.02.003`.

[73] P.G. Martinsson. A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix. *SIAM Journal on Matrix Analysis and Applications*, 32(4):1251–1274, 2011. doi: 10.1137/100786617. URL `http://link.aip.org/link/?SML/32/1251/1`.

[74] P.G. Martinsson and V. Rokhlin. A fast direct solver for boundary integral equations in two dimensions. *J. Comp. Phys.*, 205(1):1–23, 2005.

[75] E. Michielssen, A. Boag, and W. C. Chew. Scattering from elongated objects: direct solution in $O(N \log^2 N)$ operations. *IEE Proc. Microw. Antennas Propag.*, 143(4):277 – 283, 1996.

[76] Victor Minden, Kenneth L. Ho, Anil Damle, and Lexing Ying. A recursive skeletonization factorization based on strong admissibility. *Multiscale Modeling & Simulation*, 15(2):768–796, 2017. doi: 10.1137/16M1095949. URL `https://doi.org/10.1137/16M1095949`.

[77] Yuji Nakatsukasa. Fast and stable randomized low-rank matrix approximation, 2020. arxiv.org report #2009.11392.

[78] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, 2008.

[79] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[80] Grégoire Pichon, Eric Darve, Mathieu Faverge, Pierre Ramet, and Jean Roman. Sparse supernodal solver using block low-rank compression. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1138–1147. IEEE, 2017.

[81] V. Rokhlin. Diagonal forms of translation operators for the Helmholtz equation in three dimensions. *Applied and Computational Harmonic Analysis*, 1(1):82 – 93, 1993. ISSN 1063-5203. doi: 10.1006/acha.1993.1006. URL `http://www.sciencedirect.com/science/article/pii/S1063520383710067`.

[82] Vladimir Rokhlin. Rapid solution of integral equations of scattering theory in two dimensions. *Journal of Computational physics*, 86(2):414–439, 1990.

[83] Yousef Saad and Maria Sosonkina. Distributed schur complement techniques for general sparse linear systems. *SIAM Journal on Scientific Computing*, 21 (4):1337–1356, 1999.

[84] RN Simpson and Z Liu. Acceleration of isogeometric boundary element analysis through a black-box fast multipole method. *Engineering Analysis with Boundary Elements*, 66:168–182, 2016.

[85] Barry F Smith. Domain decomposition methods for partial differential equations. In *Parallel Numerical Algorithms*, pages 225–243. Springer, 1997.

[86] Daria Sushnikova, Leslie Greengard, Michael O'Neil, and Manas Rachh. FMM-LU: A fast direct solver for multiscale boundary integral equations in three dimensions, 2022. URL `https://arxiv.org/abs/2201.07325`.

[87] Toru Takahashi, Pieter Coulier, and Eric Darve. Application of the inverse fast multipole method as a preconditioner in a 3D Helmholtz boundary element method. *Journal of Computational Physics*, 341:406–428, 2017.

[88] Andrea Toselli and Olof Widlund. *Domain decomposition methods-algorithms and theory*, volume 34. Springer Science & Business Media, 2004.

[89] Alexandre Vion, R Bélanger-Rioux, L Demanet, and Christophe Geuzaine. A DDM double sweep preconditioner for the Helmholtz equation with matrix probing of the DtN map. *Mathematical and Numerical Aspects of Wave Propagation WAVES*, 2013, 2013.

[90] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods*, 17(3):261–272, 2020.

[91] Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat Guney, and Aashay Shringarpure. On the limits of GPU acceleration. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, volume 13, 2010.

[92] Ruoxi Wang, Chao Chen, Jonghyun Lee, and Eric Darve. PBBFMM3D: a parallel black-box algorithm for kernel matrix-vector multiplication. *Journal of Parallel and Distributed Computing*, 154:64–73, 2021.

[93] Shen Wang, Xiaoye S Li, Jianlin Xia, Yingchong Situ, and Maarten V De Hoop. Efficient scalable algorithms for solving dense linear systems with hierarchically semiseparable structures. *SIAM Journal on Scientific Computing*, 35(6):C519–C544, 2013.

[94] J. Xia, S. Chandrasekaran, M. Gu, and X.S. Li. Fast algorithms for hierarchically semiseparable matrices. *Numerical Linear Algebra with Applications*, 17 (6):953–976, 2010.

[95] 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, 2010. ISSN 0895-4798. doi: 10.1137/09074543X. URL `http://dx.doi.org/10.1137/09074543X`.

[96] Anna Yesypenko and Per-Gunnar Martinsson. GPU Optimizations for the Hierarchical Poincaré-Steklov Scheme. *arXiv preprint arXiv:2211.14969*, 2022.

[97] Anna Yesypenko and Per-Gunnar Martinsson. SlabLU: A Two-Level Sparse Direct Solver for Elliptic PDEs. *arXiv preprint arXiv:2211.07572*, 2022.

[98] Anna Yesypenko and Per-Gunnar Martinsson. Randomized Strong Recursive Skeletonization: Simultaneous compression and factorization of $\mathcal{H}$-matrices in the Black-Box Setting. *arXiv preprint arXiv:2311.01451*, 2023.

[99] Anna Yesypenko, Chao Chen, and Per-Gunnar Martinsson. SkelFMM: A Simplified Fast Multipole Method Based on Recursive Skeletonization. *arXiv preprint arXiv:2310.16668*, 2023.

[100] Lexing Ying. A kernel independent fast multipole algorithm for radial basis functions. *Journal of Computational Physics*, 213(2):451–457, 2006.

[101] Lexing Ying, George Biros, and Denis Zorin. A kernel-independent adaptive fast multipole algorithm in two and three dimensions. *Journal of Computational Physics*, 196(2):591–626, 2004.

[102] Ken-ichi Yoshida, Naoshi Nishimura, and Shoichi Kobayashi. Application of fast multipole Galerkin boundary integral equation method to elastostatic crack problems in 3D. *International Journal for Numerical Methods in Engineering*, 50(3):525–547, 2001.

[103] Chenhan D Yu, James Levitt, Severin Reiz, and George Biros. Geometry-oblivious FMM for compressing dense SPD matrices. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 53. ACM, 2017.

Vita

Anna Yesypenko is a PhD candidate in the Computational Science, Engineering, and Mathematics program at the Oden Institute. Prior to studying at the University of Texas at Austin, she earned a Bachelors of Science degree in Computer Science from Cornell University in 2017.

Address: annayesy@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.