2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3503-8711-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/IPDPS57955.2024.00046

An O(N) distributed-memory parallel direct solver for planar integral equations

Tianyu Liang
University of California, Berkeley, USA
tianyul@berkeley.edu

Chao Chen
North Carolina State University, USA
cchen49@ncsu.edu

Per-Gunnar Martinsson
The University of Texas at Austin, USA
pgm@oden.utexas.edu

George Biros
The University of Texas at Austin, USA
biros@oden.utexas.edu

***Abstract*—Boundary value problems involving elliptic PDEs such as the Laplace and the Helmholtz equations are ubiquitous in mathematical physics and engineering. Many such problems can be alternatively formulated as integral equations that are mathematically more tractable. However, an integral-equation formulation poses a significant computational challenge: solving large dense linear systems that arise upon discretization. In cases where iterative methods converge rapidly, existing methods that draw on fast summation schemes such as the Fast Multipole Method are highly efficient and well-established. More recently, linear complexity direct solvers that sidestep convergence issues by directly computing an invertible factorization have been developed. However, storage and computation costs are high, which limits their ability to solve large-scale problems in practice. In this work, we introduce a distributed-memory parallel algorithm based on an existing direct solver named "strong recursive skeletonization factorization [1]." Specifically, we apply low-rank compression to certain off-diagonal matrix blocks in a way that minimizes computation and data movement. Compared to iterative algorithms, our method is particularly suitable for problems involving ill-conditioned matrices or multiple right-hand sides. Large-scale numerical experiments are presented to show the performance of our `Julia` implementation.**

I. Introduction

Boundary value problems of classical potential theory appear frequently in scientific and engineering domains. While it may be challenging to solve these problems directly through their typical formulations involving elliptic partial differential equations (e.g., Laplace's and the Helmholtz equations), alternative formulations of these problems via integral equations (IEs) can be solved more efficiently in many environments [2, Chapter 10]. An integral equation takes the form

$$a(\boldsymbol{x})u(\boldsymbol{x}) + \int_{\Omega} K(\boldsymbol{x},\boldsymbol{y})u(\boldsymbol{y})d\boldsymbol{y} = f(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega \tag{1}$$

where $u(\boldsymbol{x})$ is the unknown to be determined, $K(\boldsymbol{x},\boldsymbol{y})$ is derived from the free-space fundamental solution associated with the underlying elliptic operator, $a(\boldsymbol{x})$ and $f(\boldsymbol{x})$ are given functions, and Ω is the problem domain.

We are interested in solving the associated *dense* linear system

$$A\,x = b, \quad A \in \mathbb{C}^{N\times N}, \quad x \text{ and } b \in \mathbb{C}^{N} \tag{2}$$

that arises from the discretization of Eq. (1) using approaches such as collocation, the Nyström method, or the Galerkin method. To construct a fast solver, we exploit the fact that A is a kernel matrix, that is

$$A_{i,j} = K(\boldsymbol{x}_i, \boldsymbol{x}_j), \quad \forall i \neq j, \tag{3}$$

where $\{\boldsymbol{x}_i\}_{i=1}^{N} \subset \Omega$ are points related to the discretization of Eq. (1). Here, the kernel function K comes from Eq. (1) and is smooth except when $\boldsymbol{x}_i = \boldsymbol{x}_j$. In this paper, we restrict ourselves to the case that K is not too oscillatory and the problem domain Ω is planar.

Since kernel function K is smooth in Eq. (3), certain off-diagonal blocks in A can be compressed using low-rank approximations. In particular, the required numerical rank is constant (independent of N) for a prescribed accuracy when an off-diagonal block satisfies the so-called *strong admissibility*. This observation is crucial to the success of the fast multipole method (FMM) [3], [4], which requires only $\mathcal{O}(N)$ operations for applying matrix A to a vector. It may be somewhat surprising that under mild assumptions motivated by numerical experiments, a factorization of A can also be computed using $\mathcal{O}(N)$ operations [1], [5]–[8]. Although the asymptotic complexity is appealing, such factorization algorithms still require a significant amount of computation and memory footprint. A natural solution is to distribute the required computation and storage across multiple compute nodes in order to solve large-scale problems. The challenge here comes from the fact that the matrix A is dense, which seems to imply that the cost of communication among compute nodes may be excessive.

A. Contributions

We introduce a distributed-memory parallel direct solver based on the (sequential) strong recursive skeletonization (RS-S) [1], one of a few methods that require $\mathcal{O}(N)$ storage and computation for solving *dense* linear systems Eq. (2) arising from the discretization of planar integral equations. To distribute the storage and the computation, we partition the problem domain Ω among a grid of p processes, and we show that in our distributed-memory algorithm every process needs to communicate with only its neighbors in the process grid.

1530-2075/24/$31.00 ©2024 IEEE
DOI 10.1109/IPDPS57955.2024.00046

As a result, every process sends a total of $\mathcal{O}(\log N + \log p)$ messages with a total of $\mathcal{O}(\sqrt{N/p} + \log p)$ words (under the assumptions in Section IV). We also describe an implementation using the `Julia` programming language [9], [10] that can tackle dense matrices with 1 *billion* rows and columns.

As a direct solver, our algorithm has two stages: (1) a factorization stage that constructs a compressed form of the inverse A^{-1} subject to a prescribed accuracy and (2) a solution stage that applies the compressed inverse to a right-hand side (vector) b to obtain an approximate solution of Eq. (2). Once the first phase finishes, the second phase is extremely efficient. Therefore, our algorithm is particularly suitable for problems where

1) the condition number of A is large. This typically occurs when Eq. (1) is a first-kind Fredholm integral equation for Laplace's equation or the Stokes equation, which has applications in magnetostatics, electrostatics and fluid dynamics. For example, the condition number of A scales empirically as $\mathcal{O}(N)$ for Laplace's equation.
2) multiple right-hand sides need to be solved. This typically occurs when Eq. (1) is the Lippmann-Schwinger equation that models the scattering of acoustic waves in media with variable speed. Solving the Lippmann-Schwinger equation has applications in seismology, ultrasound tomography, and sonar, where incident waves from multiple angles need to be solved.

As in the FMM, our method starts by partitioning the set of row/column indices $\{1, 2, \ldots, N\}$ of matrix A into clusters $\mathcal{B}_1, \mathcal{B}_2, \ldots$ such that indices in every cluster correspond to points spatially close to each other. The mathematical property we exploit in our method is the following. Given a cluster $\mathcal{B}$ and the union of non-adjacent clusters $\mathcal{F}$, the off-diagonal submatrices $A_{\mathcal{B},\mathcal{F}}$ and $A_{\mathcal{F},\mathcal{B}}$ are numerically low-rank and can be compressed efficiently without explicitly forming the entire matrices. We show that applying Gaussian elimination to a subset of indices in $\mathcal{B}$ after the compression (a.k.a, skeletonization) leads to a Schur-complement update that affects only adjacent clusters.

Based on these observations, we propose a domain decomposition strategy where the computational domain Ω is partitioned among all processes and show that all processes can work on their interior clusters in parallel. To process the remaining boundary clusters, we color all processes so any pair of processes have two different colors if they own adjacent clusters. Then, we loop over all colors, and processes with the same color can work in parallel at each iteration.

We implemented our parallel algorithm using `Julia` because of its ease of usage and support for various numerical linear algebra routines. In particular, our implementation explores the distributed computing capability of `Julia`[1].

B. Relation to existing work

The only existing distributed-memory parallel direct solver with $\mathcal{O}(N)$ complexity for solving Eq. (2) is Ma et al. [13]. They introduced an $\mathcal{H}^2$-ULV factorization in two passes: the first pass pre-computes all possible fill-ins and includes them in low-rank approximations, and thus the second pass computing the actual factorization is fully parallel. The disadvantage is that the overhead of an extra pass can be significant.

The inverse FMM [5], [8] is another $\mathcal{O}(N)$ (sequential) method and shares a lot of similarities with the RS-S [1], which our distributed-memory algorithm is built upon. Takahashi et al. [14] analyzed the parallelism of the inverse FMM and proposed a parallel algorithm for *shared-memory* machines. The proposed algorithm by Takahashi et al. [14] is similar to a parallel strategy briefly discussed at the end of [1], which colors all clusters of indices (our distributed-memory algorithm colors processes).

Both the inverse FMM and the RS-S implicitly make use of the $\mathcal{H}^2$-matrices introduced by Hackbusch and collaborators. Their seminal work [15]–[17] establishes the algebra of $\mathcal{H}$- and $\mathcal{H}^2$-matrices, which was implemented by several software packages [18]–[20] attaining the theoretical linear or quasilinear complexity for solving Eq. (2). The algorithms are recursive and rely on expensive hierarchical matrix-matrix multiplication, so the hidden constants in the asymptotic scalings are quite large.

Other forms of hierarchical low-rank approximations include the hierarchical semi-separable (HSS) matrices [21]–[23], hierarchical off-diagonal low-rank (HODLR) matrices [24], [25], among others [26], [27]. These methods have $\mathcal{O}(N)$ complexity in 1D (e.g., boundary IEs on curves), but their costs deteriorate to super linear in 2D and 3D if the a fixed target accuracy is desired. High-performance and parallel implementations can be found in [28]–[35]. Corona et al. [6] and Ho et al. [7] improve upon earlier recursive skeletonization ideas [36], [37] and attain $\mathcal{O}(N)$ complexity by incorporating extra compression steps on intermediate skeletons. However, parallelizing such algorithms is challenging.

Besides approximations in hierarchical formats, flat formats such as block low-rank [38] or tile low-rank [39] have also been proposed recently. Although algorithms leveraging these flat formats have led to significant speedups compared to classical algorithms for general dense matrices in practical applications [40]–[43], they do not attain $\mathcal{O}(N)$ complexity.

II. Sequential algorithm and data dependency

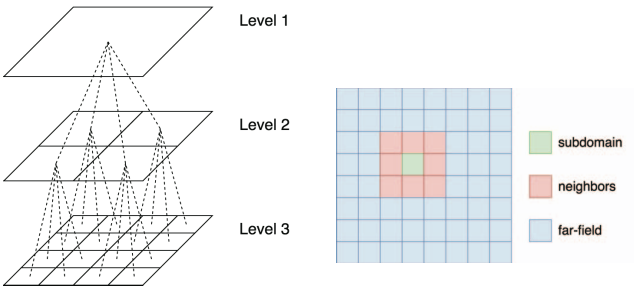
In this section, we review the sequential algorithm for factorizing and solving Eq. (2) approximately. The focus here is analyzing data dependency and parallelism. While there exist a few methods that are closely related such as [5], [8], [11], [12], we mostly follow [1], [44].

Suppose we are given a set of points $\mathcal{X} = \{\boldsymbol{x}_i\}_{i=1}^N$ in $\mathbb{R}^2$ and a kernel function K that defines matrix A as in Eq. (3). Our approach for solving Eq. (2) employs approximate Gaussian elimination in a multi-level fashion.

A. Hierarchical domain decomposition

The algorithm relies on a hierarchical decomposition of the points $\mathcal{X} = \{\boldsymbol{x}_i\}_{i=1}^N$ as in the FMM. The hierarchical

[1]https://docs.julialang.org/en/v1/manual/distributed-computing/



(a) Hierarchical domain decomposition and the quad-tree.

(b) A subdomain, its near-field (neighbors), and its far-field.

Fig. 1: Assume the points are uniformly distributed in a square domain. (a) a 3-level hierarchical domain decomposition and the corresponding quad-tree. (b) a subdomain at the fourth level in a hierarchical domain decomposition, its near-field (neighbors), and its far-field.

decomposition is as follows. Suppose all points lie in a square domain. We divide the domain into four equal-sized subdomains. We call the entire domain the *parent* of the four subdomains, which are the *children* of the original domain. We continue subdividing the four subdomains recursively until every subdomain contains $\mathcal{O}(1)$ points. This hierarchical decomposition of the problem domain naturally maps to a quad tree $\mathcal{T}$, where the root is the entire domain and the leaves are all subdomains that are not subdivided. See a pictorial illustration in Figure 1a.

For ease of presentation, we further make the assumption that the points are uniformly distributed, so the hierarchical domain decomposition corresponds to a *perfect* quad tree (all internal nodes have 4 children and all the leaf nodes are at the same level). Extensions to a non-uniform distribution of points are straightforward but quite tedious; see details in [1], [44].

We refer to a subdomain (or a node in tree $\mathcal{T}$) as a *box* for the rest of this paper. Given a box $\mathcal{B}$ at level ℓ in the tree $\mathcal{T}$, its *parent* $\mathcal{P}(\mathcal{B})$ and its *children* $\mathcal{C}(\mathcal{B})$ are naturally defined (if they exist). We also define the *near-field* (a.k.a., *neighbors*) $\mathcal{N}(\mathcal{B})$ and the *far-field* $\mathcal{F}(\mathcal{B})$ as follows:

- $\mathcal{N}(\mathcal{B})$: boxes that are physically adjacent to $\mathcal{B}$ (excluding $\mathcal{B}$) at level ℓ.
- $\mathcal{F}(\mathcal{B})$: boxes excluding $\mathcal{B} \cup \mathcal{N}(\mathcal{B})$ at level ℓ.

See an example in Figure 1b. Assuming $\mathcal{T}$ is a perfect quad-tree, no box has more than eight neighbors, i.e., $|\mathcal{N}(\mathcal{B})| \leq 8$. With some abuse in notation, we use $\mathcal{B}, \mathcal{N}, \mathcal{F}$ to also denote indices of points in $\mathcal{X}$ that are contained in box $\mathcal{B}$, its near-field/neighbors $\mathcal{N}(\mathcal{B})$, and its far-field $\mathcal{F}(\mathcal{B})$, respectively.

Given a leaf node $\mathcal{B}$, there exists a permutation matrix P such that

$$P^\top A P = \begin{pmatrix} A_{\mathcal{B},\mathcal{B}} & A_{\mathcal{B},\mathcal{N}} & A_{\mathcal{B},\mathcal{F}} \\ A_{\mathcal{N},\mathcal{B}} & A_{\mathcal{N},\mathcal{N}} & A_{\mathcal{N},\mathcal{F}} \\ A_{\mathcal{F},\mathcal{B}} & A_{\mathcal{F},\mathcal{N}} & A_{\mathcal{F},\mathcal{F}} \end{pmatrix}. \tag{4}$$

Here, $A_{\mathcal{I},\mathcal{J}}$ for two index sets $\mathcal{I}$ and $\mathcal{J}$ stands for the corresponding submatrix in A.

B. Low-rank property

Assumption 1: For any box $\mathcal{B}$, it holds that $A_{i,j} = K(x_i, x_j)$ for (1) $i \in \mathcal{B}$ and $j \in \mathcal{F}$; or (2) $i \in \mathcal{F}$ and $j \in \mathcal{B}$. In other words, the submatrices $A_{\mathcal{B},\mathcal{F}}$ and $A_{\mathcal{F},\mathcal{B}}$ come from kernel evaluation.

The observation is that the off-diagonal block $A_{\mathcal{B},\mathcal{F}}$ (or $A_{\mathcal{F},\mathcal{B}}$) for an arbitrary box $\mathcal{B}$ can be approximated efficiently by a low-rank approximation for a prescribed accuracy ε. In particular, the numerical rank is $\mathcal{O}(1)$, independent of the problem size N, for a smooth kernel that is not highly oscillatory.

Next, we explain a specific type of low-rank approximation named the interpolative decomposition (ID) [45], which is applied to the submatrix $A_{\mathcal{B},\mathcal{F}}$. Other low-rank approximations such as the truncated singular value decomposition can also be used; see the resulting solvers for Eq. (2) in, e.g., [11], [46].

Definition 1: Let $\mathcal{I} = \{1, 2, \ldots, m\}$ and $\mathcal{J} = \{1, 2, \ldots, n\}$ be the row and column indices of a matrix $A_{\mathcal{I},\mathcal{J}} \in \mathbb{C}^{m \times n}$. 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* $T \in \mathbb{C}^{|\mathcal{S}| \times |\mathcal{R}|}$ such that

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

While the strong rank-revealing QR factorization of Gu and Eisenstat [47] is the most robust method for computing an ID, we employ the column-pivoting QR factorization as a greedy approach [45], which has better computational efficiency and behave well in practice. In particular, we adopted the implementation from the `LowRankApprox.jl`[2] package. 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 [48].

C. Fast compression

As stated earlier, we want to compress the two off-diagonal blocks $A_{\mathcal{B},\mathcal{F}}$ and $A_{\mathcal{F},\mathcal{B}}$ using their IDs. In practice, we conduct one (column) ID compression of the concatenation

$$\begin{pmatrix} A_{\mathcal{F},\mathcal{B}} \\ A^*_{\mathcal{B},\mathcal{F}} \end{pmatrix} \tag{5}$$

(that has $\mathcal{O}(N)$ rows and $\mathcal{O}(1)$ columns) and obtain a single interpolation matrix T that satisfies both

$$A_{\mathcal{F},\mathcal{R}} \approx A_{\mathcal{F},\mathcal{S}} T \quad \text{and} \quad A_{\mathcal{R},\mathcal{F}} \approx T^* A_{\mathcal{S},\mathcal{F}}. \tag{6}$$

This approach leads to a slightly larger set of skeleton indices but makes the algorithm/implementation easier.

Notice that *the computational cost would be $\mathcal{O}(N)$ if the full matrix in Eq.* (5) *is formed, which turns out to be unnecessary*. As in the FMM, there are a few techniques that require only

[2]https://github.com/JuliaLinearAlgebra/LowRankApprox.jl

$\mathcal{O}(1)$ operations such as the (analytical) multipole expansion (see, e.g., [3], [4], [49]), the Chebyshev interpolation (see, e.g., [50], [51]), and the proxy method (see, e.g., [36], [52]).

Below, we focus on illustrating the proxy method for compressing the submatrix $A_{\mathcal{F},\mathcal{B}}$. Specifically, we form and compress the following matrix

$$\begin{pmatrix} A_{\mathcal{M},\mathcal{B}} \\ K_{\text{proxy},\mathcal{B}} \end{pmatrix}, \tag{7}$$

which has $\mathcal{O}(1)$ *rows* (and columns). In Eq. (7), $\mathcal{M} = \mathcal{M}(\mathcal{B})$ is a *small* subset of $\mathcal{F}(\mathcal{B})$ surrounding $\mathcal{N}(\mathcal{B})$ defined as follows (see Figure 2):

Definition 2: The distance-2 neighbors of a box $\mathcal{B}$ is defined as

$$\mathcal{M}(\mathcal{B}) = \mathcal{N}(\mathcal{N}(\mathcal{B}))\backslash\left(\mathcal{N}(\mathcal{B}) \cup \mathcal{B}\right).$$

Note that Assumption 1 does not hold in our algorithm, where the submatrix $A_{\mathcal{M},\mathcal{B}}$ is updated and is *not* the kernel evaluation any more; see Section II-D. However, we can show the following to be true:

Theorem 1: For any leaf box $\mathcal{B}$, it holds that $A_{i,j} = K(x_i, x_j)$ for (1) $i \in \mathcal{B}$ and $j \in \mathcal{F}\backslash\mathcal{M}$; or (2) $i \in \mathcal{F}\backslash\mathcal{M}$ and $j \in \mathcal{B}$. In other words, the submatrices $A_{\mathcal{B},\mathcal{F}\backslash\mathcal{M}}$ and $A_{\mathcal{F}\backslash\mathcal{M},\mathcal{B}}$ come from kernel evaluation.

This relaxation means that the established rank estimation in Section II-B no longer holds. However, we empirically verify that the resulting rank from compressing Eq. (7) still follows the previous estimation; see numerical results in Section V.

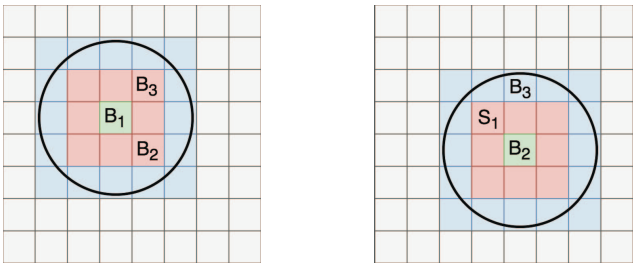
In Eq. (7), $K_{\text{proxy},\mathcal{B}}$ stands for a matrix where the (i,j)-th entry is given by $K(y_i, x_j)$, i.e., a kernel evaluation between a discretization point y_i on the *proxy circle*[3] and a point $x_j \in \mathcal{B}$ lying inside the box $\mathcal{B}$. The "interaction" $K_{\text{proxy},\mathcal{B}}$ accounts for the "interaction" between points lying in $\mathcal{F}(\mathcal{B})\backslash\mathcal{M}(\mathcal{B})$ and points inside $\mathcal{B}$. To that end, the proxy circle must lie inside $\mathcal{M}(\mathcal{B})$. See a pictorial illustration in Figure 2. In this paper, we choose the radius of the proxy circle to be $2.5L$, where L is the side length of boxes at this level.

Remark 1: The computation of skeleton indices $\mathcal{S}$, redundant indices $\mathcal{R}$, and the interpolation matrix T in Eq. (6) requires (reads) submatrices $A_{\mathcal{M},\mathcal{B}}$ and $A_{\mathcal{B},\mathcal{M}}$ (*not* the entire submatrices $A_{\mathcal{F},\mathcal{B}}$ or $A_{\mathcal{B},\mathcal{F}}$) for a leaf box $\mathcal{B}$.

D. Approximate matrix factorization

The algorithm starts at the leaf level of the quad-tree $\mathcal{T}$. Let us apply an ID compression for a box $\mathcal{B}$ to obtain skeleton indices $\mathcal{S}$, redundant indices $\mathcal{R} = \mathcal{B}\backslash\mathcal{S}$, and an interpolation matrix T such that Eq. (6) holds. (As explained in the previous section, this step requires only $\mathcal{O}(1)$ operations.)

[3]A circle is chosen for ease of implementation but not necessary mathematically.



(a) (b)

Fig. 2: Illustration of proxy circle and distance-2 neighbors $\mathcal{M}$. (a) For a box $\mathcal{B}_1$, its distance-2 neighbors $\mathcal{M}(\mathcal{B}_1)$ (in blue) is the small subset (16 boxes at most) of its far-field $\mathcal{F}(\mathcal{B}_1)$ that surrounds its neighbors (in red). The proxy circle for $\mathcal{B}_1$ lies in $\mathcal{M}(\mathcal{B})$. (b) An example showing that the "interactions" between $\mathcal{B}_3$ and $\mathcal{B}_2$ (submatrices $A_{\mathcal{B}_2,\mathcal{B}_3}$ and $A_{\mathcal{B}_3,\mathcal{B}_2}$) have been modified after the redundant indidces $\mathcal{R}_1$ is eliminated (the skeleton indices $\mathcal{S}_1$ remain). See Remark 2 below.

Rewrite Eq. (4) as

$$\tilde{P}^\top A \tilde{P} = \left(\begin{array}{cc|cc} A_{\mathcal{R},\mathcal{R}} & A_{\mathcal{R},\mathcal{S}} & A_{\mathcal{R},\mathcal{N}} & A_{\mathcal{R},\mathcal{F}} \\ A_{\mathcal{S},\mathcal{R}} & A_{\mathcal{S},\mathcal{S}} & A_{\mathcal{S},\mathcal{N}} & A_{\mathcal{S},\mathcal{F}} \\ \hline A_{\mathcal{N},\mathcal{R}} & A_{\mathcal{N},\mathcal{S}} & A_{\mathcal{N},\mathcal{N}} & A_{\mathcal{N},\mathcal{F}} \\ A_{\mathcal{F},\mathcal{R}} & A_{\mathcal{F},\mathcal{S}} & A_{\mathcal{F},\mathcal{N}} & A_{\mathcal{F},\mathcal{F}} \end{array}\right)$$
$$\overset{Eq.\ (6)}{\approx} \left(\begin{array}{cc|cc} A_{\mathcal{R},\mathcal{R}} & A_{\mathcal{R},\mathcal{S}} & A_{\mathcal{R},\mathcal{N}} & T^* A_{\mathcal{S},\mathcal{F}} \\ A_{\mathcal{S},\mathcal{R}} & A_{\mathcal{S},\mathcal{S}} & A_{\mathcal{S},\mathcal{N}} & A_{\mathcal{S},\mathcal{F}} \\ \hline A_{\mathcal{N},\mathcal{R}} & A_{\mathcal{N},\mathcal{S}} & A_{\mathcal{N},\mathcal{N}} & A_{\mathcal{N},\mathcal{F}} \\ A_{\mathcal{F},\mathcal{S}}\, T & A_{\mathcal{F},\mathcal{S}} & A_{\mathcal{F},\mathcal{N}} & A_{\mathcal{F},\mathcal{F}} \end{array}\right),$$

where $\tilde{P}$ is an appropriate permutation matrix. Define the *sparsification* matrix

$$S = \begin{pmatrix} I & & & \\ -T & I & & \\ & & I & \\ & & & I \end{pmatrix}, \tag{8}$$

where the partitioning of row and column indices is the same as that of $\tilde{P}^\top A \tilde{P}$ and the diagonal blocks are identity matrices of appropriate sizes. Applying the sparsification matrix, we have that

$$S^* \left(\tilde{P}^\top A \tilde{P}\right) S \approx \left(\begin{array}{ccc|c} X_{\mathcal{R},\mathcal{R}} & X_{\mathcal{R},\mathcal{S}} & X_{\mathcal{R},\mathcal{N}} & \\ X_{\mathcal{S},\mathcal{R}} & A_{\mathcal{S},\mathcal{S}} & A_{\mathcal{S},\mathcal{N}} & A_{\mathcal{S},\mathcal{F}} \\ \hline X_{\mathcal{N},\mathcal{R}} & A_{\mathcal{N},\mathcal{S}} & A_{\mathcal{N},\mathcal{N}} & A_{\mathcal{N},\mathcal{F}} \\ & A_{\mathcal{F},\mathcal{S}} & A_{\mathcal{F},\mathcal{N}} & A_{\mathcal{F},\mathcal{F}} \end{array}\right),$$

where the coupling (submatrices) between $\mathcal{R}$ and $\mathcal{F}$ disappears. Here, the notation X denotes modified blocks that can be easily derived and be calculated using $\mathcal{O}(1)$ operations. *Notice that we never need to explicitly form $A_{\mathcal{R},\mathcal{F}}$ or $A_{\mathcal{S},\mathcal{F}}$.*

Suppose that the diagonal block $X_{\mathcal{R},\mathcal{R}}$ is non-singular and that $X_{\mathcal{R},\mathcal{R}} = L_{\mathcal{R}} U_{\mathcal{R}}$ is an LU factorization. We apply (block)

Gaussian elimination to obtain

$$L\,(S^*\tilde{P}^\top A\tilde{P}S)\,U \approx \left(\begin{array}{c|cc} I & & \\ \hline & X_{\mathcal{S},\mathcal{S}} & A_{\mathcal{S},\mathcal{F}} \\ & X_{\mathcal{N},\mathcal{S}} & A_{\mathcal{N},\mathcal{F}} \\ & A_{\mathcal{F},\mathcal{S}} & A_{\mathcal{F},\mathcal{F}} \end{array}\right), \quad (9)$$

where

$$L = \begin{pmatrix} I & & & \\ -X_{\mathcal{S},\mathcal{R}}U_{\mathcal{R}}^{-1} & I & & \\ -X_{\mathcal{N},\mathcal{R}}U_{\mathcal{R}}^{-1} & & I & \\ & & & I \end{pmatrix} \begin{pmatrix} L_{\mathcal{R}}^{-1} & & & \\ & I & & \\ & & I & \\ & & & I \end{pmatrix} \quad \text{and}$$

$$U = \begin{pmatrix} U_{\mathcal{R}}^{-1} & & & \\ & I & & \\ & & I & \\ & & & I \end{pmatrix} \begin{pmatrix} I & -L_{\mathcal{R}}^{-1}X_{\mathcal{R},\mathcal{S}} & -L_{\mathcal{R}}^{-1}X_{\mathcal{R},\mathcal{N}} & \\ & I & & \\ & & I & \\ & & & I \end{pmatrix}$$

Notice that *the original "interaction" among $\mathcal{B}$'s neighbors, i.e., submatrix $A_{\mathcal{N},\mathcal{N}}$, has been updated.*

Define

$$V = LS^*\tilde{P}^\top, \qquad W = \tilde{P}SU, \quad \text{and} \quad Z(A;\mathcal{B}) = VAW. \quad (10)$$

Here, $Z(A;\mathcal{B})$ is called the *strong skeletonization operator* in [1]. Note that V^{-1} and W^{-1} can be applied efficiently without explicitly forming their inverses.

Remark 2: Given the interpolation matrix T in Eq. (6), applying the strong skeletonization operator $Z(A;\mathcal{B})$ requires (reads) matrices $X_{\mathcal{N},\mathcal{R}}$ and $X_{\mathcal{R},\mathcal{N}}$, and it updates (read & write) submatrix $A_{\mathcal{N},\mathcal{N}}$ for a leaf box $\mathcal{B}$. (No far-field information is required.)

E. Multi-level algorithm

Let n_ℓ denote the number of boxes at level ℓ in the quad-tree $\mathcal{T}$ ($\ell = 1, 2, \ldots, L$). Given an ordering of all boxes $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_{n_L}$ at the leaf level, our algorithm applies the strong skeletonization operator Eq. (10) to all boxes one after another:

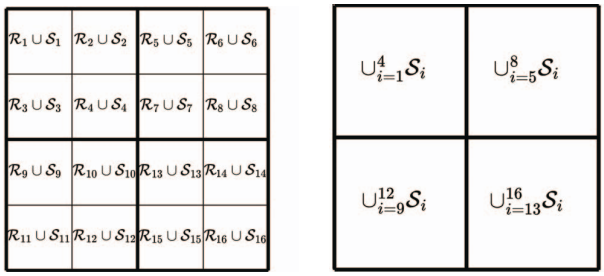
$$Z(A;\mathcal{B}_1,\mathcal{B}_2) \triangleq Z(Z(A;\mathcal{B}_1);\mathcal{B}_2)$$
$$Z(A;\mathcal{B}_1,\mathcal{B}_2,\mathcal{B}_3) \triangleq Z(Z(A;\mathcal{B}_1,\mathcal{B}_2);\mathcal{B}_3)$$
$$\ldots$$

The resulting (approximate) factorization becomes

$$Z(A;\mathcal{B}_1,\mathcal{B}_2,\ldots,\mathcal{B}_{n_L}) \approx \begin{pmatrix} I_{\mathcal{R}_1} & & & & \\ & I_{\mathcal{R}_2} & & & \\ & & \ddots & & \\ & & & I_{\mathcal{R}_{n_L}} & \\ & & & & \tilde{A} \end{pmatrix}, \quad (11)$$

where the leading diagonal blocks are identity matrices of sizes $|\mathcal{R}_i|$, which correspond to the redundant indices in box $\mathcal{B}_i$ for $i = 1, 2, \ldots, n_L$. The (approximate) Schur complement $\tilde{A}$ has $\sum_i \mathcal{S}_i$ rows and columns.

Next, we demonstrate that the single-level algorithm Eq. (11) can be applied to $\tilde{A}$ recursively. In order to factorize $\tilde{A}$ (approximately), we consider the remaining skeleton points at the coarse level $L-1$. Let every box $\mathcal{B}_i$ own the skeleton points of its children, i.e., $\cup_{k\in\mathcal{C}(\mathcal{B}_i)}\mathcal{S}_k$. See Figure 3 for a pictorial illustration. We obtain a partitioning of $\tilde{A}$ as we did for A (a leaf box $\mathcal{B}_i$ owns points physically lying inside it).



(a) The leaf level. (b) The parent level.

Fig. 3: Illustration of the merge process. Suppose a square domain is divided into 16 boxes $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_{16}$. (a) At the leaf level, a box $\mathcal{B}_i$ owns points/indices that physically lie inside it. These indices are divided into skeleton indices $\mathcal{S}_i$ and redundant indices $\mathcal{R}_i$ after an ID compression. (b) At the coarse/parent level, a parent box owns the skeleton indices of its children. (Again, they will be divided into skeleton and redundant indices after an ID compression.)

In order to continue applying the strong skeletonization operator to every box at level $L-1$, we need to first verify Assumption 1 still holds. For the skeletons $\mathcal{S}_i$ in box $\mathcal{B}_i$, the modified interactions (submatrices) only exist between $\mathcal{S}_i$ and $\cup_{k\in\mathcal{M}(\mathcal{B}_i)}\mathcal{S}_k$. At level $L-1$, it can be shown that the parent of $\mathcal{B}_i$ is the neighbor of the parent of any box in $\mathcal{M}(\mathcal{B}_i)$. As a result, Assumption 1 holds, and thus we can repeat the previous approach to apply the strong skeletonization operator to every box at level $L-1$. By induction, we know that

Theorem 2: Assumption 1 holds at all levels $\ell = L, L-1, \ldots, 2, 1$[4].

We refer to [1, Section 3.3.2] for a formal proof of the above theorem. The implication of the theorem is that Remark 2 and Remark 1 both hold at all levels. Therefore, we obtain a multi-level algorithm:

$$Z(A;\underbrace{\mathcal{B}_1,\mathcal{B}_2,\ldots,\mathcal{B}_{n_L}}_{\text{level } L},\ \ldots,\ \underbrace{\mathcal{B}_1,\mathcal{B}_2,\ldots,\mathcal{B}_{n_1}}_{\text{level 1}}) \approx I. \quad (12)$$

As explained in Section II-D, the inverse of a strong skeletonization operator can be applied efficiently (without forming the inverse operator explicitly). We summarize the complete algorithm in Algorithm 1.

As mentioned earlier, other low-rank approximation methods can also be used to construct similar factorizations [5], [8]. The same framework can be applied to solving large *sparse* linear systems with minor modifications [11], [12]. While the described framework leverages the so-called strong admissibility (compression of far-field), a few methods based on the weak admissibility (compression of both far-field and near-field) can also construct efficient factorizations [6], [7].

[4]There is no far-field for any box at level 1, so the theorem holds trivially.

Algorithm 1 `sequential factorization`

Input: Kernel function K, points $\mathcal{X} = \{x_i\}_{i=1}^N \in \mathbb{R}^2$.
Output: Factorization in eq. (12).

1: Compute a hierarchical partition of $\mathcal{X}$. For every box $\mathcal{B}$, we obtain its neighbors $\mathcal{N}$ and distance-2 neighbors $\mathcal{M}$ (definition 2).
2: **for** $\ell = L, L-1, \ldots, 1$ **do** // bottom-up sweep
3: **for** $i = 1, 2, \ldots, n_\ell$ **do** // a given ordering of boxes
4: STRONG_SKELETONIZATION(box $\mathcal{B}_i$ at level ℓ)

1: **function** STRONG_SKELETONIZATION($\mathcal{B}$)
2: Read matrices $A_{\mathcal{M},\mathcal{B}}$ and $A_{\mathcal{B},\mathcal{M}}$, and compute the ID compression. // section II-C
3: Read matrices $A_{\mathcal{R},\mathcal{R}}$, $A_{\mathcal{N},\mathcal{R}}$, and $A_{\mathcal{R},\mathcal{N}}$; and update matrix $A_{\mathcal{N},\mathcal{N}}$. // section II-D

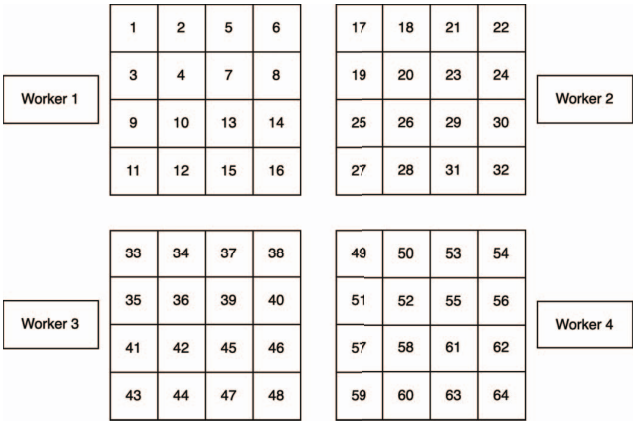


Fig. 4: Distribution of our data structure associated with boxes among all worker processes. The data structure is stored using distributed arrays.

F. Solution/applying the inverse of factorization

The solution phase follows the standard forward and backward substitution procedures besides applying the inverse of the sparsification operator Eq. (8). The algorithm consists of an upward and a downward level-by-level traversal of the tree $\mathcal{T}$. During the upward pass, all boxes at the same level are visited according to the order of factorization. For every box $\mathcal{B}$, we first update a subset of the right-hand-side (RHS) $b(\mathcal{B})$ and then update $b(\mathcal{N})$ associated with neighbor boxes of $\mathcal{B}$ in our in-place algorithm. The operations are "reversed" during the downward pass, where we read neighbor data $b(\mathcal{N})$ and update local data $b(\mathcal{B})$ for every box $\mathcal{B}$.

III. Distributed-memory parallel algorithm

In the following, we analyze data dependency at the leaf level based on previous results summarized in Remarks 1 and 2, but the analysis also holds at other levels in the tree (see Section II-E). In `Julia`, a master process (process 1) provides the interactive prompt and coordinates worker processes to conduct parallel operations. To store our data structures such as a list of required submatrices for every box, we used distributed arrays from `DistributedArrays.jl`. A distributed array has the property that a process can make a fast local access but has only read permission for a remote access. Our parallel algorithm starts by constructing local data structures on each worker and assembling them into distributed arrays.

At the leaf level, all boxes are distributed evenly among all worker processes. See Figure 4 for an example. On every worker $\mathcal{W}_i$, the boxes are classified into two groups: (1) "interior boxes", whose neighbors are on the same worker, and (2) "boundary boxes", whose neighbors are on $\mathcal{W}_i$ and at least one other worker $\mathcal{W}_j$. In Figure 4, boundary boxes on worker 1 include 6, 8, 14, 16, 11, 12, and 15; and the rest are all interior boxes. Notice that if the total number of boxes is large, the number of interior boxes dominates.

Consider a pair of boxes $\mathcal{B}_i$ and $\mathcal{B}_j$ at the leaf level. Suppose the two boxes are on two different workers $\mathcal{W}_i$ and $\mathcal{W}_j$, respectively. (Otherwise, the two boxes will be processed sequentially.) Consider the distance $d(\mathcal{B}_i, \mathcal{B}_j)$ between the two boxes,[5] where

1) $\text{dist}(\mathcal{B}_i, \mathcal{B}_j) = 1$ if $\mathcal{B}_i$ and $\mathcal{B}_j$ are neighbors/adjacent (e.g., boxes 6 and 17 in Figure 4). In fact, both boxes must be boundary boxes. We discuss the parallel algorithm for handling all boundary boxes in Section III-B.
2) $\text{dist}(\mathcal{B}_i, \mathcal{B}_j) = 2$ if $\mathcal{B}_i$ and $\mathcal{B}_j$ are not neighbors but share a common neighbor, i.e., they are distance-2 neighbors as defined in Definition 2 (e.g., boxes 5 and 17 in Figure 4). In fact, one of the two boxes must be a boundary box and the other one must be an interior box. If one of them, say $\mathcal{B}_i$, is processed first, it requires accessing the corresponding submatrices $A_{\mathcal{B}_i,\mathcal{B}_j}$ and $A_{\mathcal{B}_j,\mathcal{B}_i}$ (see Remark 1) but will not update submatrices in the rows/columns of $\mathcal{B}_j$ (see Remark 2).
3) $\text{dist}(\mathcal{B}_i, \mathcal{B}_j) > 2$ if $\mathcal{B}_i$ and $\mathcal{B}_j$ do not share any common neighbors (e.g., boxes 5 and 18 in Figure 4). In this case, the two boxes can be processed in parallel according to Remarks 1 and 2.

Our parallel algorithm for the described matrix factorization employs a level-by-level traversal of the quad-tree $\mathcal{T}$. At every level, we have three steps: (1) processing the massive interior boxes in parallel, (2) processing the remaining boundary boxes, and (3) creating the next/coaser level. The pseudocode is shown in Algorithm 2, and we explain implementation details in the following sections.

A. Interior boxes

For the first step in our parallel algorithm at every level, notice that if a pair of interior boxes $\mathcal{B}_i$ and $\mathcal{B}_j$ are on two different workers, then $\text{dist}(\mathcal{B}_i, \mathcal{B}_j) > 2$ (e.g., boxes 5 and 18 in Figure 4). Therefore, all workers can apply the strong

[5] $d(\mathcal{B}_i, \mathcal{B}_j) = \max(|x_i - x_j|, |y_i - y_j|)/\ell$, where (x_i, y_i) and (x_j, y_j) are the centers of $\mathcal{B}_i$ and $\mathcal{B}_j$, respectively, and ℓ is the side length of boxes at the same level as $\mathcal{B}_i$ and $\mathcal{B}_j$.

Algorithm 2 `parallel factorization`

1: **for** tree level $\ell = L, L-1, \ldots, 1$ **do**
2: All workers factorize interior boxes at level ℓ.
3: **for** color $i = 1, 2, 3, 4$ **do**
4: Workers with color i factorize boundary boxes at level ℓ.
5: Workers with color i send data to neighbors.
6: All workers construct level $\ell - 1$.

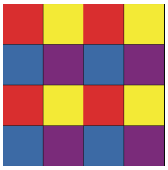


Fig. 5: Coloring of a 4×4 process grid. For a 2D grid, we need 4 colors so that every process has a different color from its adjacent processes.

skeletonization operator (see Algorithm 1) to factorize their respective interior boxes concurrently. In our implementation, every worker stores all necessary submatrices for this step. For example, in Figure 4 both worker 1 and worker 2 store copies of the submatrices $A_{5,17}$ and $A_{17,5}$. Recall Remark 1 that both box 5 and box 17 need the two submatrices for computing their IDs.

The distributed programming model in Julia (Distributed.jl) is similar to task-based programming in, e.g., OpenMP [53], where the main process creates new processes (workers) and launches tasks to all workers through remote procedure calls. An example is the following:

```
@sync for i in workerIDs
  @async remotecall_wait(factor, i, boxes)
end
```

where `remotecall_wait` tells worker i to factorize a set of boxes, the `async` macro means the task launch is asynchronous, and the `sync` macro on the loop makes sure that the master process waits for all workers to finish their tasks. (If `remotecall` is used instead, then the master process only waits for all workers to receive their tasks.)

B. Boundary boxes

After the first step, all workers are left with their boundary boxes to be processed. We assign colors to all workers such that any pair of adjacent processes have two different colors. Although mature software packages exist for coloring an arbitrary graph, we assume the boxes at every level form a grid (a full quad-tree $\mathcal{T}$), and thus a coloring of the process grid is straightforward as shown in Figure 5. Notice that 4 colors are needed for a 2D process grid regardless of the number of boxes on every worker.

Once the coloring is available, our parallel algorithm loops over all colors. For each color, the associated processors need to fetch submatrices required for subsequent computation and then process their boundary boxes in parallel. For the example in Figure 5, our algorithm will schedule 4 boxes, say with red color; wait for their completion; then schedule the next 4 boxes, say with yellow color; wait for their completion; and so on. Notice that if a pair of boundary boxes $\mathcal{B}_i$ and $\mathcal{B}_j$ are on two different processors with the same color, then $\text{dist}(\mathcal{B}_i, \mathcal{B}_j) > 2$ as long as every processor has at least $2 \times 2 = 4$ boxes.

After all workers with a specific color α finish factorizing their boundary boxes, they need to send submatrices from Schur complement updates to their neighbors. In `Julia`, communication is "one-sided," which means that only one process needs to be explicitly managed by the programmer. According to our best knowledge, remote direct memory access [54] is not available in `Julia` for distributed computing. So we implemented explicit communication through remote procedural calls. In particular, the master process launches `receive_data` tasks on the neighbor workers as follows:

```
@sync for i in neighborIDs
  @async
  remotecall_wait(receive_data, i, α)
end
```

Here, `receive_data` is a function defined on all workers through the use of the `everywhere` macro as follows:

```
@everywhere function receive_data(α)
  @sync for k in neighborWithColor[α]
    @async
    data=remotecall_wait(send_data, k, myID)
  end
end
@everywhere function send_data(id)
  return UpdatesForWorker[id]
end
```

where the worker that needs data launches tasks on its neighbors who have color α, and data will be returned in the form of a `future` object. Finally, the neighbor with color α returns/sends the requested data back, which is stored in a distributed array.

C. Level transition

Since the algorithm traverses the quad-tree in a level-by-level fashion starting from the leaf level, it is important to keep track of the data structure when the algorithm transitions from one level to the next. The approach we take is to explicitly store the modified interactions for every box. As a result, whenever the algorithm changes level, it will need to reconstruct the data structure by regrouping interactions among accumulated points. An additional thing that we need to keep track of is where the coordinates of the given points are stored. At the start of the algorithm, the points are stored evenly distributed across all processes using a distributed array. However, as the algorithm progresses from one level to the next, two things will change. First, since each new level are constructed from the skeletonized points of the previous level,

the points that are involved in the factorization of the new level are a subset of the previous points. Second, the number of processes involved in the new level may also decrease. Hence, this suggests that during the transition stage, we also need to reconstruct a distributed array by using only the relevant skeletonized points.

IV. Complexity analysis

For simplicity, we make the following three assumptions. First, the discretization points $\{\boldsymbol{x}_i\}_{i=1}^N$ in Eq. (3) are uniformly distributed in a square domain, so a hierarchical partitioning of the points correspond to a perfect quad-tree $\mathcal{T}$ (see Figure 1b). Second, the numerical ranks of the two submatrices $A_{\mathcal{B},\mathcal{F}}$ and $A_{\mathcal{F},\mathcal{B}}$ are constant for every cluster $\mathcal{B}$ (see Section II-B), which is denoted by a scalar r. We further assume r does not depend on the problem size N (see numerical results in Figure 9). Third, the number of points per box at the leaf level is $\mathcal{O}(r)$, so the number of levels in the quad-tree $\mathcal{T}$ is $L = \mathcal{O}(\log(N/r)) = \mathcal{O}(\log N)$.

A. Computational and memory complexity

The costs of the strong RS algorithm were derived in [1, Section 3.3.4]: the factorization cost $t_f = \mathcal{O}(N)$, the solve cost $t_s = \mathcal{O}(N)$, and the memory footprint also scales as $\mathcal{O}(N)$, where N is the problem size. Similar results for the inverse FMM algorithm can be found in [8, Section 2.5]. In the parallel algorithm, we uniformly partition the problem, the required work, and the memory footprint among all processors.

B. Communication cost

Notice the quad-tree $\mathcal{T}$ has $L = \mathcal{O}(\log N)$ levels, and all processors are organized as another quad-tree with $\mathcal{O}(\log p)$ levels. So every processor owns a subtree of $\mathcal{O}(\log(N/p))$ levels. In our parallel factorization algorithm, every processor sends a constant number of messages with a constant number of words for the first $\mathcal{O}(\log(N/p))$ levels. Then, the remaining algorithm behaves as a parallel reduction among p processors. As a result, the number of messages sent by every processor is $\mathcal{O}(\log N + \log p)$, and the number of words moved is

$$\mathcal{O}(\sqrt{N/rp} + \log p) = \mathcal{O}(\sqrt{N/p} + \log p) \tag{13}$$

where $\sqrt{N/rp}$ is the number of boundary boxes on every processor. For strong scaling experiments, i.e., $N = \mathcal{O}(1)$, the number of messages required is $\mathcal{O}(1)$, and the number of words is $\mathcal{O}(1/\sqrt{p} + \log p)$. For weak scaling experiments, i.e., $N/p = \mathcal{O}(1)$, the number of messages required is $\mathcal{O}(\log p)$, and the number of words is $\mathcal{O}(\log p)$.

V. Numerical results

In this section, we show benchmark results for solving two types of dense linear systems $Ax = b$ that are associated with the free-space Green's function for the Laplace equation in 2D and that for the Helmholtz equation in 2D, respectively. They represent kernel functions that are non-oscillatory and mildly oscillatory. For ease of setting up experiments, we discretized the integral equation Eq. (1) on uniform grids in the unit square. As a result, the matrix-vector product with dense matrix A can be performed efficiently via the fast Fourier transform. (Otherwise, a fast summation algorithm such as the distributed-memory FMM is required.) Table I summaries our notations for reporting numerical results.

TABLE I: *Notations used to report results of solving $Ax = b$, where A is from discretizing the integral equation Eq.* (1)*, and b is a standard uniform random vector. Timings are in seconds.*

N	size of matrix A, i.e., $A \in \mathbb{C}^{N\times N}$.
ε	tolerance for low-rank compression; see Definition 1. Unless otherwise noted, $\varepsilon = 10^{-6}$ is used.
p	number of processes. In strong scaling tests (N fixed), we started with the minimum number of compute nodes (processes) and increased the number of processes to at most 64 per node.
t_{fact}	wall time of constructing the factorization Eq. (12) in parallel.
t_{solve}	wall time of applying the inverse of the factorization in parallel.
t_{comp}	fraction of t_{fact} or t_{solve} spent on computation.
t_{other}	fraction of t_{fact} or t_{solve} spent on communication and overhead.
relres	**rel**ative **res**idual, i.e., $\|A\tilde{x} - b\|/\|b\|$, where $\tilde{x}$ is the output of our solver.
n_{it}	number of preconditioned CG or GMRES iterations to reach 10^{-12} tolerance, where our solver is used as a preconditioner.

All experiments were performed with `Julia`, version 1.9.4, on the CPU nodes of Perlmutter[6], an HPE (Hewlett Packard Enterprise) Cray EX supercomputer. Each compute node has two AMD EPYC 7763 (Milan) CPUs, 64 cores per CPU, and 512 GB of memory. Since `Julia` employs a just-in-time compiler that compiles a code before its first execution, we always ran a small problem size before timing our numerical experiments.

A. Laplace kernel

Consider the following problem involving the free-space Green's function for the 2D Laplace equation: solving the first-kind volume integral equation

$$\int_\Omega K(\|\boldsymbol{x} - \boldsymbol{y}\|)u(\boldsymbol{y})d\boldsymbol{y} = f(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega = [0,1]^2 \tag{14}$$

with the kernel function

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = -\frac{1}{2\pi}\log\left(\|\boldsymbol{x}_i - \boldsymbol{x}_j\|\right). \tag{15}$$

We discretized Eq. (14) using piecewise-constant collocation on a $\sqrt{N}\times\sqrt{N}$ uniform grid $\mathcal{X}$. The resulting linear system involves a dense matrix $A \in \mathbb{R}^{N\times N}$: the off-diagonal entries are

$$A_{i,j} = -\frac{h^2}{2\pi}\log\left(\|\boldsymbol{x}_i - \boldsymbol{x}_j\|\right), \quad \forall i \neq j, \tag{16}$$

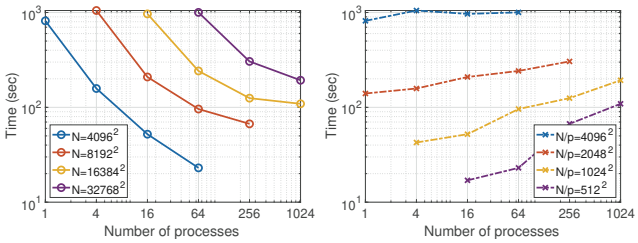
where $\boldsymbol{x}_i, \boldsymbol{x}_j \in \mathcal{X}$ and $h = 1/\sqrt{N}$ is the grid spacing; and the diagonal entries are given by

$$A_{i,i} = \int_{-h/2}^{h/2}\int_{-h/2}^{h/2} -\frac{1}{2\pi}\log\left(\|\boldsymbol{x}\|\right) dx_1\, dx_2, \quad \forall i, \tag{17}$$

[6]https://docs.nersc.gov/systems/perlmutter/architecture/

TABLE II: *Runtime for solving dense linear systems associated with the 2D Laplace kernel Eq.* (15). *The tolerance for low-rank compression is* $\varepsilon = 10^{-6}$, *and the accuracy of our solver is shown in Table III.*

N	p	factorization time			solve time (one iteration)		
		$t_{\text{fact}} =$	$t_{\text{comp}}+$	t_{other}	$t_{\text{solve}} =$	$t_{\text{comp}}+$	t_{other}
2048^2	1	140	126	14	4.12	3.82	0.30
	4	42.6	36.9	5.7	1.50	1.25	0.25
	16	17.0	13.2	3.80	0.77	0.55	0.22
	64	10.6	7.6	3.0	0.64	0.37	0.27
4096^2	1	817	719	98	36.36	32.30	4.06
	4	158	141	17	6.19	5.47	0.72
	16	52.1	42.2	9.9	2.06	1.63	0.43
	64	23.0	17.1	5.9	1.41	1.00	0.41
8192^2	4	1050	847	203	54.11	48.25	5.86
	16	209	147	62	7.19	6.12	1.07
	64	96	46	50	3.69	2.81	0.88
	256	67	21	46	2.96	1.63	1.33
16384^2	16	967	749	218	30.63	27.20	3.43
	64	242	158	84	10.99	8.18	2.81
	256	125	51	74	5.39	4.03	1.36
	1024	109	26	83	6.18	2.85	3.33
32768^2	64	1003	771	232	35.83	29.23	6.60
	256	305	168	137	9.34	7.23	2.11
	1024	193	59	134	11.29	5.81	5.48



(a) Strong scaling (N fixed) (b) Weak scaling (N/p fixed)

Fig. 6: Scalability of the factorization time t_{fact} in Table II.

where $\boldsymbol{x} = (x_1, x_2) \in \mathbb{R}^2$. We evaluated Eq. (17) numerically using an adaptive quadrature (`dblquad` from `MultiQuad.jl`).

Tables II and III show the relevant numerical results, and Figure 6 shows the parallel scalability of the factorization time. We make the following observations:

1) We were able to solve a large problem size of $N = 32\,768^2 \approx 1$ billion using 1024 processes in less than five minutes (200 seconds for factorization and around $11 \times 6 = 66$ seconds for six PCG iterations).
2) The approximate factorizations constructed in parallel required almost constant number of PCG iterations to arrive at a tolerance of 10^{-12}. By contrast, the number of CG iterations is approximately $5\sqrt{N}$ without any preconditioners.
3) The solve time for a single RHS is much smaller than the factorization time.

TABLE III: *Accuracies for solving dense linear systems associated with the 2D Laplace kernel Eq.* (15). ε *denotes the tolerance for low-rank compression.*

ϵ	N	p	t_{fact}	t_{solve}	relres	n_{it}
10^{-6}	2048^2	1	139	4.12	1.11e-4	4
	4096^2	4	158	6.61	2.63e-4	5
	8192^2	16	209	9.85	4.48e-4	5
	16384^2	64	242	12.01	6.57e-4	6
10^{-9}	2048^2	1	225	4.69	1.31e-7	2
	4096^2	4	262	6.09	2.14e-7	2
	8192^2	16	374	10.73	2.48e-7	3
	16384^2	64	452	14.17	5.39e-7	3
10^{-12}	2048^2	1	446	5.85	1.44e-10	2
	4096^2	4	642	7.42	2.58e-10	2
	8192^2	16	867	9.25	4.47e-10	2
	16384^2	64	1048	16.45	5.58e-10	2

B. Helmholtz Kernel

Consider the next problem involving the free-space Green's function for the 2D Helmholtz equation: solving the second-kind volume integral equation

$$\sigma(\boldsymbol{x}) + \kappa^2 b(\boldsymbol{x}) \int_\Omega K(\|\boldsymbol{x}-\boldsymbol{y}\|)\sigma(\boldsymbol{y})d\boldsymbol{y} = -\kappa^2 b(\boldsymbol{x})\, u_{\text{in}}(\boldsymbol{x}) \tag{18}$$

where $\boldsymbol{x} \in \Omega = [0,1]^2$, known as the Lippmann-Schwinger equation, a reformulation of the variable coefficient Helmholtz equation (see, e.g., [2, Section 11.2]) that models, e.g., acoustic wave propagation in a medium with a variable wave speed. Here $u_{\text{in}}(\boldsymbol{x})$ is the incoming wave with frequency κ; the "scattering potential" $0 < b(\boldsymbol{x}) \le 1$ is a known smooth function compactly supported inside Ω; and the kernel function

$$K(\boldsymbol{x}_i, \boldsymbol{x}_j) = \frac{i}{4} H_0^{(1)}(\kappa\|\boldsymbol{x}_i - \boldsymbol{x}_j\|), \tag{19}$$

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

We symmetrized Eq. (18) via change of variable $\mu(\boldsymbol{x}) = \sigma(\boldsymbol{x})/\sqrt{b(\boldsymbol{x})}$ and applied piecewise-constant collocation on a $\sqrt{N} \times \sqrt{N}$ uniform grid $\mathcal{X}$. The resulting *indefinite complex* linear system involves a dense matrix $A \in \mathbb{C}^{N\times N}$: the off-diagonal entries are

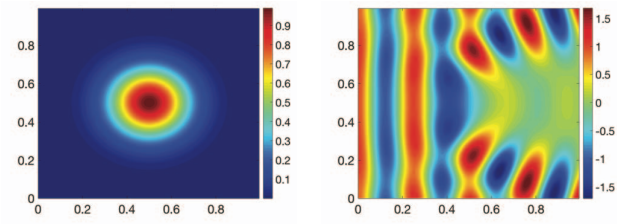
$$A_{i,j} = h^2\kappa^2\sqrt{b(\boldsymbol{x}_i)b(\boldsymbol{x}_j)}\left(\frac{i}{4}H_0^{(1)}(\kappa\|\boldsymbol{x}_i - \boldsymbol{x}_j\|)\right) \tag{20}$$

where $\boldsymbol{x}_i, \boldsymbol{x}_j \in \mathcal{X}$, and $h = 1/\sqrt{N}$ is the grid spacing; and the diagonal entries are given by

$$A_{i,i} = 1 + \kappa^2 b(\boldsymbol{x}_i) \int_{-h/2}^{h/2}\int_{-h/2}^{h/2} \frac{i}{4} H_0^{(1)}(\kappa\|\boldsymbol{x}\|)\, dx_1\, dx_2 \tag{21}$$

where $\boldsymbol{x} = (x_1, x_2) \in \mathbb{R}^2$. We evaluated Eq. (21) numerically using an adaptive quadrature (`dblquad` from `MultiQuad.jl`).

For the following experiments, we use a Gaussian bump scattering potential $b(\boldsymbol{x}) = e^{-32\|\boldsymbol{x}-\boldsymbol{c}\|^2}$ with the center $\boldsymbol{c} = (\frac{1}{2}, \frac{1}{2})$, as shown in Figure 7a. For an incoming plan wave $u_{\text{in}}(\boldsymbol{x})$ pointing to the right, the total field $u(\boldsymbol{x}) = u_{\text{in}}(\boldsymbol{x}) + \int_\Omega K(\|\boldsymbol{x}-\boldsymbol{y}\|)\sigma(\boldsymbol{y})d\boldsymbol{y}$ is shown in Figure 7b.

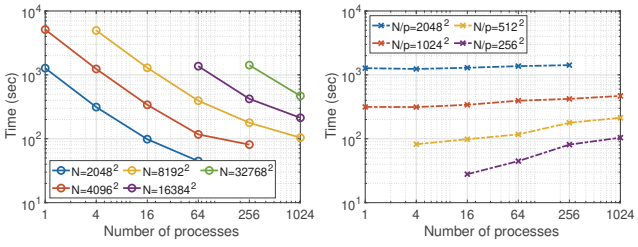


(a) Scattering potential (b) Total field

Fig. 7: Gaussian bump scattering potential and the total field for an incoming plan wave traveling from left to right.

TABLE IV: *Results for solving dense linear systems associated with the 2D Helmholtz kernel Eq.* (19) *with* fixed *frequency* $\kappa = 25$. *The tolerance for low-rank compression is* $\varepsilon = 10^{-6}$, *and the accuracy of our solver is shown in Table VI.*

N	p	factorization time			solve time (one iteration)		
		$t_{\text{fact}} =$	$t_{\text{comp}}+$	t_{other}	$t_{\text{solve}} =$	$t_{\text{comp}}+$	t_{other}
1024^2	1	315	300	15	1.77	1.60	0.17
	4	82	76	6	0.64	0.50	0.14
	16	27	23	4	0.47	0.24	0.23
	64	11	8	3	0.56	0.20	0.36
2048^2	1	1273	1212	61	6.74	6.43	0.31
	4	313	294	19	2.00	1.74	0.26
	16	98	88	10	1.08	0.85	0.23
	64	44	38	6	0.89	0.54	0.35
4096^2	1	5116	4873	243	30.69	29.37	1.32
	4	1237	1174	63	8.05	7.31	0.74
	16	340	314	26	3.59	2.79	0.80
	64	117	103	14	2.02	1.60	0.42
	256	81	44	37	1.78	0.97	0.81
8192^2	4	4958	4636	322	36.08	33.26	2.82
	16	1292	1178	114	12.65	9.88	2.77
	64	394	318	76	5.28	3.85	1.43
	256	178	109	69	4.62	3.21	1.41
	1024	104	50	54	4.47	2.32	2.15
16384^2	64	1369	1187	182	16.07	13.35	2.72
	256	419	315	104	7.72	5.36	2.36
	1024	213	110	103	7.57	3.78	3.79
32768^2	256	1420	1206	215	22.44	15.33	7.12
	1024	468	329	139	17.13	8.69	8.44
	4096	289	131	158	28.28	9.88	18.40



(a) Strong scaling (N fixed) (b) Weak scaling (N/p fixed)

Fig. 8: Scalability of the factorization time t_{fact} in Table IV.

TABLE V: *Results for solving dense linear systems associated with the 2D Helmholtz kernel Eq.* (19) *with* increasing *frequencies* ($\kappa = \pi\sqrt{N}/16$, *i.e., 32 points per wavelength). The last column* $\tilde{n}_{it}$ *shows the number of GMRES iterations (restart = 20) without any preconditioners.*

N	p	$\kappa/(2\pi)$	t_{fact}	t_{solve}	n_{it}	$\tilde{n}_{\text{it}}$
1024^2	1	32	365	1.83	3	452
2048^2	4	64	495	14.56	4	1 752
4096^2	16	128	684	22.26	5	8 198
8192^2	64	256	1138	29.74	12	$> 10\,000$

1) Fixed frequency ($\kappa = 25$): Table IV shows the relevant numerical results for a fixed frequency $\kappa = 25$, and Figure 8 shows the parallel scalability of the factorization time.

- The factorization time is much longer than that for the Laplace kernel. The reason is that an evaluation of the (complex) Helmholtz kernel in Eq. (19) takes longer.
- The parallel factorization algorithm achieves greater speedups compared to those for the Laplace kernel. The advantage of constructing a highly accurate approximation is clear: the solve time is much faster, suitable for situations where multiple RHSs need to be addressed.
- Again, the approximate factorizations constructed in parallel required a consistent number (three) of GMRES iterations to arrive at a tolerance of 10^{-12}.

2) Increasing frequency ($\kappa = \pi\sqrt{N}/16$, i.e., 32 points per wavelength): Table V shows the relevant numerical results for incoming waves with increasing frequencies.

- The factorization time is increasingly longer than that in Table IV, where the frequency was fixed. Recall that the numerical rank for the Helmholtz kernel is $\mathcal{O}(\kappa D)$; see Figure 9. As a result, the factorization requires $\mathcal{O}(\kappa^3) = \mathcal{O}(N^{3/2})$ operations.
- As κ increases in Eq. (18), the integral equation becomes increasingly more ill-conditioned. So is the discretized linear system, which requires more and more preconditioned GMRES iterations to converge. However, the savings from employing the preconditioner is clear: the number of GMRES iterations without any preconditioner is orders-of-magnitudes larger and grows rapidly.

C. Comparison and 1-process-per-node run

We present two experiments on solving dense linear systems $Ax = b$ associated with the 2D Helmholtz kernel Eq. (19) with *fixed* frequency $\kappa = 25$, where the matrix A is defined in Eqns. (20) and (21). The first one is a comparison between our distributed-memory solver implemented in `Julia` and a shared-memory solver implemented in `C++` with `OpenMP` [53]. Both parallel solvers are based on (sequential) RS-S [1]. The shared-memory solver follows ideas from Takahashi et al. [14] and the parallel strategy briefly mentioned at the end of [1]: it colors all boxes at the same level, and every pair of neighbors has different colors. By contrast, our distributed-memory solver colors only the boundary boxes

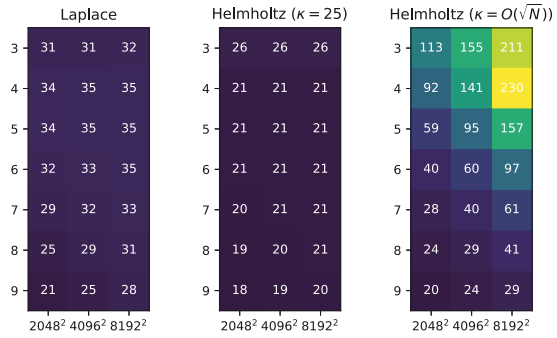


Fig. 9: Numerical ranks of the 2D Laplace kernel Eq. (15) and the 2D Helmholtz kernel Eq. (19). The y-axis stands for the tree level (see Figure 1a), where low-rank compression occurs starting from level 3. Every entry stands for the average rank of all boxes at a certain level.

TABLE VI: *Comparison between a shared-memory solver in* `C++` *with* `OpenMP` *and our distributed-memory solver in* `Julia`*. Both codes ran on one compute node with one process or one thread per core. The matrix size* $N = 2048^2$.

ϵ	p	C++ (reference) t_{fact}	t_{solve}	relres	Julia (this paper) t_{fact}	t_{solve}	relres	n_{it}
10^{-3}	1	641	5.47	8.5e-4	866	5.63	1.9e-3	5
	4	168	1.50		225	1.92	1.8e-3	6
	16	60	0.46		64	0.76	1.8e-3	5
	64	37	0.25		23	0.58	2.1e-3	5
10^{-6}	1	1215	8.11	4.1e-7	1272	6.74	3.2e-6	3
	4	310	2.17		313	2.00	4.2e-6	3
	16	101	0.82		98	1.08	2.5e-6	3
	64	59	0.39		45	0.89	3.5e-6	3
10^{-9}	1	3621	13.18	9.1e-10	1730	7.11	3.3e-9	2
	4	845	3.22		485	3.06	1.3e-9	2
	16	260	1.06		166	1.24	1.5e-9	2
	64	117	0.76		92	1.03	2.1e-9	2
10^{-12}	1	6010	18.22	7.9e-13	2591	8.33	4.2e-12	2
	4	1531	5.18		728	3.59	6.3e-12	2
	16	437	1.54		283	1.34	6.1e-12	2
	64	204	1.02		182	1.30	8.7e-12	2

on every process, and the boundary boxes on each process have the same color. The results of comparison are shown in Table VI and fig. 10, where we observe that our `Julia` code performed similarly with the reference `C++` code when running on 64 cores of one compute node. The factorization of the Julia solver is faster but the solve time is slower with slightly worse accuracy.

The second experiment investigates the extra costs for launching one process per compute node rather than launching multiple processes per compute node in previous sections. In particular, we reran a subset of experiments in Table IV using as many compute nodes as the number of processes. Intuitively, we expect the new experiments to take longer due to more communication through a network. The results in Table VII show that the extra wall-clock time is, however,

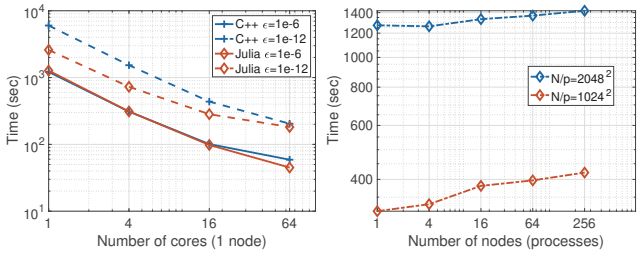


Fig. 10: Left: Comparison of the factorization time (t_{fact} in Table VI) between a shared-memory solver in `C++` with `OpenMP` (blue) and our distributed-memory solver in `Julia` (red). Both codes ran on one compute node with 1 process or 1 thread per core. Right: Weak scaling of the factorization time t_{fact} in Table VII. One process was launched on every compute node.

TABLE VII: *Results for launching 1 process per compute node for a subset of experiments in Table IV. Here, the number of processes* p *equals the number of compute nodes.*

N	nodes	factorization time t_{fact} =	t_{comp}+	t_{other}	solve time (one iteration) t_{solve} =	t_{comp}+	t_{other}
1024^2	1	315	300	15	1.77	1.60	0.17
2048^2	1	1273	1212	61	6.74	6.43	0.31
	4	332	292	40	2.09	1.74	0.35
4096^2	1	5116	4873	243	30.69	29.37	1.32
	4	1263	1160	103	8.86	7.95	0.91
	16	381	311	70	3.24	2.68	0.56
8192^2	4	4958	4636	322	36.08	33.26	2.82
	16	1333	1172	161	11.97	10.69	1.28
	64	397	315	82	4.38	3.35	1.03
16384^2	64	1369	1187	182	16.07	13.35	2.72
	256	421	314	107	6.97	4.67	2.30
32768^2	256	1420	1206	215	22.45	15.33	7.12

negligible for these large-scale problems on the Perlmutter supercomputer.

ACKNOWLEDGMENTS

T. Liang acknowledges support from the National Science Foundation Graduate Research Fellowship Program (No. 2146752), the Department of Energy ASCR (DE-AC02-05CH11231), and computing time on the Perlmutter supercomputer provided by an allocation from DOE. C. Chen acknowledges support from startup funds of North Carolina State University. P.G. Martinsson acknowledges support from the Office of Naval Research (N00014-18-1-2354), the National Science Foundation (DMS-2313434 and DMS-1952735), and the Department of Energy ASCR (DE-SC0022251). G. Biros acknowledges support from the National Science Foundation (OAC-2204226), the Department of Energy ASCR (DE-SC0023171), and computing time on the Texas Advanced Computing Centers (TACC) Stampede system provided by an allocation from TACC and the NSF.

REFERENCES

[1] V. Minden, K. L. Ho, A. Damle, and L. Ying, "A recursive skeletonization factorization based on strong admissibility," *Multiscale Modeling & Simulation*, vol. 15, no. 2, pp. 768–796, 2017.

[2] P.-G. Martinsson, *Fast direct solvers for elliptic PDEs*. SIAM, 2019.

[3] L. Greengard and V. Rokhlin, "A fast algorithm for particle simulations," *Journal of computational physics*, vol. 73, no. 2, pp. 325–348, 1987.

[4] ——, "A new version of the fast multipole method for the Laplace equation in three dimensions." YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE, Tech. Rep., 1996.

[5] S. Ambikasaran and E. Darve, "The inverse fast multipole method," *arXiv preprint arXiv:1407.1572*, 2014.

[6] E. Corona, P.-G. Martinsson, and D. Zorin, "An O(N) direct solver for integral equations on the plane," *Applied and Computational Harmonic Analysis*, vol. 38, no. 2, pp. 284–317, 2015.

[7] K. L. Ho and L. Ying, "Hierarchical interpolative factorization for elliptic operators: integral equations," *Comm. Pure Appl. Math*, vol. 69, no. 7, pp. 1314–1353, 2016.

[8] P. Coulier, H. Pouransari, and E. Darve, "The inverse fast multipole method: using a fast approximate direct solver as a preconditioner for dense linear systems," *SIAM Journal on Scientific Computing*, vol. 39, no. 3, pp. A761–A796, 2017.

[9] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, 2012.

[10] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.

[11] H. Pouransari, P. Coulier, and E. Darve, "Fast hierarchical solvers for sparse matrices using extended sparsification and low-rank approximation," *SIAM Journal on Scientific Computing*, vol. 39, no. 3, pp. A797–A830, 2017.

[12] D. A. Sushnikova and I. V. Oseledets, ""compress and eliminate" solver for symmetric positive definite sparse matrices," *SIAM Journal on Scientific Computing*, vol. 40, no. 3, pp. A1742–A1762, 2018.

[13] Q. Ma, S. Deshmukh, and R. Yokota, "Scalable linear time dense direct solver for 3-d problems without trailing sub-matrix dependencies," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '22. IEEE Press, 2022.

[14] T. Takahashi, C. Chen, and E. Darve, "Parallelization of the inverse fast multipole method with an application to boundary element method," *Computer Physics Communications*, vol. 247, p. 106975, 2020.

[15] W. Hackbusch, "A sparse matrix arithmetic based on H-matrices. part I: Introduction to H-matrices," *Computing*, vol. 62, no. 2, pp. 89–108, 1999.

[16] W. Hackbusch and S. Börm, "Data-sparse approximation by adaptive H^2-matrices," *Computing*, vol. 69, no. 1, pp. 1–35, 2002.

[17] W. Hackbusch and B. N. Khoromskij, "A sparse H-matrix arithmetic. part ii: Application to multi-dimensional problems," *Computing*, vol. 64, no. 1, p. 21–47, Jan. 2000.

[18] L. Grasedyck, R. Kriemann, and S. Le Borne, "Parallel black box-lu preconditioning for elliptic boundary value problems," *Computing and visualization in science*, vol. 11, no. 4-6, pp. 273–291, 2008.

[19] R. Kriemann, "H-lu factorization on many-core systems," *Computing and Visualization in Science*, vol. 16, no. 3, pp. 105–117, 2013.

[20] ——, "Parallel-matrix arithmetics on shared memory systems," *Computing*, vol. 74, pp. 273–297, 2005.

[21] S. Chandrasekaran, M. Gu, and T. Pals, "A fast ULV decomposition solver for hierarchically semiseparable representations," *SIAM Journal on Matrix Analysis and Applications*, vol. 28, no. 3, pp. 603–622, 2006.

[22] S. Chandrasekaran, P. Dewilde, M. Gu, W. Lyons, and T. Pals, "A fast solver for HSS representations via sparse matrices," *SIAM Journal on Matrix Analysis and Applications*, vol. 29, no. 1, pp. 67–81, 2007.

[23] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li, "Superfast multifrontal method for large structured linear systems of equations," *SIAM Journal on Matrix Analysis and Applications*, vol. 31, no. 3, pp. 1382–1411, 2010.

[24] S. Ambikasaran and E. Darve, "An\mathcal o (n\log n) o (n log n) fast direct solver for partial hierarchically semi-separable matrices: With application to radial basis function interpolation," *Journal of Scientific Computing*, vol. 57, pp. 477–501, 2013.

[25] A. Aminfar, S. Ambikasaran, and E. Darve, "A fast block low-rank dense solver with applications to finite-element matrices," *Journal of Computational Physics*, vol. 304, pp. 170–188, 2016.

[26] Y. Chen, "A fast, direct algorithm for the lippmann–schwinger integral equation in two dimensions," *Advances in Computational Mathematics*, vol. 16, pp. 175–190, 2002.

[27] J. Bremer, "A fast direct solver for the integral equations of scattering theory on planar curves with corners," *Journal of Computational Physics*, vol. 231, no. 4, pp. 1879–1899, 2012.

[28] F.-H. Rouet, X. S. Li, P. Ghysels, and A. Napov, "A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization," *ACM Transactions on Mathematical Software (TOMS)*, vol. 42, no. 4, pp. 1–35, 2016.

[29] P. Ghysels, X. S. Li, F.-H. Rouet, S. Williams, and A. Napov, "An efficient multicore implementation of a novel hss-structured multifrontal solver using randomized sampling," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S358–S384, 2016.

[30] X. Liu, J. Xia, and M. V. De Hoop, "Parallel randomized and matrix-free direct solvers for large structured dense linear systems," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S508–S538, 2016.

[31] D. Cai, E. Chow, L. Erlandson, Y. Saad, and Y. Xi, "Smash: Structured matrix approximation by separation and hierarchy," *Numerical Linear Algebra with Applications*, vol. 25, no. 6, p. e2204, 2018.

[32] S. Ambikasaran, K. R. Singh, and S. S. Sankaran, "Hodlrlib: A library for hierarchical matrices," *Journal of Open Source Software*, vol. 4, no. 34, p. 1167, 2019.

[33] D. Y. Chenhan, S. Reiz, and G. Biros, "Distributed o (n) linear solver for dense symmetric hierarchical semi-separable matrices," in *2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE, 2019, pp. 1–8.

[34] C. Chen and P.-G. Martinsson, "Solving linear systems on a gpu with hierarchically off-diagonal low-rank approximations," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 1–15.

[35] S. Deshmukh, R. Yokota, G. Bosilca, and Q. Ma, "O (n) distributed direct factorization of structured dense matrices using runtime systems." in *Proceedings of the 52nd International Conference on Parallel Processing*, 2023, pp. 1–10.

[36] P.-G. Martinsson and V. Rokhlin, "A fast direct solver for boundary integral equations in two dimensions," *Journal of Computational Physics*, vol. 205, no. 1, pp. 1–23, 2005.

[37] L. Greengard, D. Gueyffier, P.-G. Martinsson, and V. Rokhlin, "Fast direct solvers for integral equations in complex three-dimensional domains," *Acta Numerica*, vol. 18, pp. 243–275, 2009.

[38] T. Mary, "Block low-rank multifrontal solvers: complexity, performance, and scalability," Ph.D. dissertation, Université Paul Sabatier-Toulouse III, 2017.

[39] K. Akbudak, H. Ltaief, A. Mikhalev, and D. Keyes, "Tile low rank cholesky factorization for climate/weather modeling applications on manycore architectures," in *International Conference on High Performance Computing*. Springer, 2017, pp. 22–40.

[40] P. R. Amestoy, A. Buttari, J.-Y. L'excellent, and T. Mary, "Performance and scalability of the block low-rank multifrontal factorization on multicore architectures," *ACM Transactions on Mathematical Software (TOMS)*, vol. 45, no. 1, pp. 1–26, 2019.

[41] W. Boukaram, S. Zampini, G. Turkiyyah, and D. Keyes, "H2opus-tlr: High performance tile low rank symmetric factorizations using adaptive randomized approximation," *arXiv preprint arXiv:2108.11932*, 2021.

[42] Q. Cao, R. Alomairy, Y. Pei, G. Bosilca, H. Ltaief, D. Keyes, and J. Dongarra, "A framework to exploit data sparsity in tile low-rank cholesky factorization," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 414–424.

[43] Q. Cao, S. Abdulah, R. Alomairy, Y. Pei, P. Nag, G. Bosilca, J. Dongarra, M. G. Genton, D. E. Keyes, H. Ltaief, and Y. Sun, "Reshaping geostatistical modeling and prediction for extreme-scale environmental applications," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '22. IEEE Press, 2022.

[44] D. Sushnikova, L. Greengard, M. O'Neil, and M. Rachh, "Fmm-lu: A fast direct solver for multiscale boundary integral equations in three dimensions," *arXiv preprint arXiv:2201.07325*, 2022.

[45] H. Cheng, Z. Gimbutas, P.-G. Martinsson, and V. Rokhlin, "On the compression of low rank matrices," *SIAM Journal on Scientific Computing*, vol. 26, no. 4, pp. 1389–1404, 2005.

[46] L. Cambier, C. Chen, E. G. Boman, S. Rajamanickam, R. S. Tuminaro, and E. Darve, "An algebraic sparsified nested dissection algorithm using low-rank approximations," *SIAM Journal on Matrix Analysis and Applications*, vol. 41, no. 2, pp. 715–746, 2020.
[47] M. Gu and S. C. Eisenstat, "Efficient algorithms for computing a strong rank-revealing qr factorization," *SIAM Journal on Scientific Computing*, vol. 17, no. 4, pp. 848–869, 1996.
[48] Y. Dong and P.-G. Martinsson, "Simpler is better: a comparative study of randomized algorithms for computing the cur decomposition," *arXiv preprint arXiv:2104.05877*, 2021.
[49] H. Cheng, W. Y. Crutchfield, Z. Gimbutas, L. F. Greengard, J. F. Ethridge, J. Huang, V. Rokhlin, N. Yarvin, and J. Zhao, "A wideband fast multipole method for the helmholtz equation in three dimensions," *Journal of Computational Physics*, vol. 216, no. 1, pp. 300–325, 2006.
[50] W. Fong and E. Darve, "The black-box fast multipole method," *Journal of Computational Physics*, vol. 228, no. 23, pp. 8712–8725, 2009.
[51] R. Wang, C. Chen, J. Lee, and E. Darve, "PBBFMM3D: a parallel black-box algorithm for kernel matrix-vector multiplication," *Journal of Parallel and Distributed Computing*, vol. 154, pp. 64–73, 2021.
[52] L. Ying, G. Biros, and D. Zorin, "A kernel-independent adaptive fast multipole algorithm in two and three dimensions," *Journal of Computational Physics*, vol. 196, no. 2, pp. 591–626, 2004.
[53] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks," *IEEE Transactions on Parallel and Distributed systems*, vol. 20, no. 3, pp. 404–418, 2008.
[54] W. Lu, L. E. Peña, P. Shamis, V. Churavy, B. Chapman, and S. Poole, "Bring the bitcode – moving compute and data in distributed heterogeneous systems," 2022. [Online]. Available: https://arxiv.org/abs/2208.01154