論文 / 著書情報
Article / Book Information

題目(和文)	
Title(English)	Linear Time Dense Direct Solver Without Trailing Sub-matrix Dependencies
著者(和文)	MAQianxiang
Author(English)	Qianxiang Ma
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:甲第12880号, 授与年月日:2024年9月20日, 学位の種別:課程博士, 審査員:横田 理央,吉瀬 謙二,宮﨑 純,DEFAGO XAVIER,小野 峻佑
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第12880号, Conferred date:2024/9/20, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Linear Time Dense Direct Solver Without Trailing Sub-matrix Dependencies

Qianxiang Ma
Department of Computer Science
School of Computing
Tokyo Institute of Technology

A thesis submitted for the degree of
Doctor of Engineering

September 2024
Under the supervision of Prof. Rio Yokota

Tokyo Tech

Contents

List of Figures

List of Algorithms

Abstract

Obtaining the dense matrix factorization and direct solutions is used in various fields in scientific applications, such as the frontal matrix in multi-frontal sparse solvers and preconditioners of the integral equation solvers based on the boundary element methods. This thesis explores the hierarchical low-rank approximation of dense matrices, which can reduce the complexity of the arithmetic from $\mathcal{O}(N^3)$ to close to $\mathcal{O}(N)$, including matrix-vector multiplication, matrix-matrix-multiplication, the factorization, and the forward and backward substitution. Inspired by the fast multipole method, the hierarchical low-rank methods originated as an algebraic interpretation of the linear time matrix-vector multiplication in FMM. By constructing explicitly the matrix being approximated into different structures, we have a chance of directly inverting the hierarchically low-rank approximated matrix.

The structured low-rank approximated matrices differ in the following aspects: the number of levels, the admissibility condition, and the use of shared or independent bases. The three aspects influence the implementation complexity, the algorithmic complexity, the different algorithms to adopt, and the ranks needed to meet the desired compression or factorization accuracy. Existing approaches have extremely parallel implementations in the matrix-vector and matrix-matrix multiplications, but the factorization is hard to run in parallel due to the data dependencies. For BLR^2 and HSS matrices, a possibility has been found to remove the data dependency amongst the diagonals from the Cholesky/LU factorization via a ULV factorization. However, only using the weak admissibility configurations in the BLR^2 and HSS matrices poses great limits on the number of dimensions in the problem geometry, primarily due to the growing ranks of off-diagonal blocks in 3-D problems, and the method is no longer made $\mathcal{O}(N)$.

To address this problem, this thesis discusses the extension to the ULV factorization and substitution, in particular, made for strongly admissible $\mathcal{H}^2$-matrices. The contribution discusses the algorithm, which removes the need to wait for the basis updates via a pre-compression phase, that brings the re-compression computations happening inside the ULV factorization before all factorization operations. By having this means of integrating the pre-compressed results into the shared basis, we also made the data dependencies in the diagonal trailing blocks disappear. The work presented featured the first inherently

parallel factorization and substitution on distributed memory for a strongly admissible hierarchical low-rank matrix in $\mathcal{O}(N)$ complexity, without any assistance or the overhead from the parallel runtime tools. From the algorithm improvement alone, we achieved a maximum speed up of 4,700x for a 3-D problem with complex geometry, compared with a block low-rank factorization code LORAPO. In the many-GPU implementation of the same algorithm, we utilize the batched interfaces of BLAS and LAPACK and unify the different block sizes. The batching of the kernels compensates for the low arithmetic intensity in the low-rank computations, and we eventually extracted over 800 TFLOPS of performance on 512 NVIDIA V100 GPUs. Our implementation is also the first implementation for obtaining the direct solution of a $\mathcal{H}^2$-matrix running on multiple GPUs.

Acknowledgements

It has been wonderful years spent in the Tokyo Institute of Technology and Professor Rio Yokota's lab. Completing this thesis has been a pleasant yet challenging journey, and has marked an important event in my life. Without the support and guidance of many surrounding individuals, this would not been made possible.

First and foremost, I would like to express my deepest gratitude to my supervisor and mentor, Professor Rio Yokota. Your guidance in the research expertise fosters the growth of many students including me. Thank you for having faith in our developments, and providing us the precious opportunity to conduct research and present our work to the rest of the world.

I would also like to express my appreciation for my colleagues and lab-mates in the Yokota lab, Tokyo Institute of Technology. The technical discussions made and the leisure time spent together have been enjoyable, and have greatly impacted the quality of this presented work. I am particularly grateful to Dr. Sameer Deshmukh, Dr. Muhammad Ridwan Apriansyah, Mr. Peter Spalthoff, and Mr. Thomas Spendlhofer, for their support and collaboration in the various aspects of this research.

On a personal note, I am grateful to my parents, Dr. Jie Wang. and Zhizhong Ma M.D., for their unwavering support and encouragement especially during the period of COVID-19. Thank you for understanding, in this particularly challenging time.

Lastly, to my prospective partner, Ms. Xiaoshan Qiu. Thank you for your patience, company, belief, and support. The time spent together and our adventures in Eorzea and Final Fantasy XIV are my unforgettable memories.

This thesis is a dedicated work to all of you mentioned above.

Chapter 1

Introduction

Dense linear algebra libraries BLAS and LAPACK have provided application interfaces for matrix and vector operations such as multiplication, matrix factorization, and eigenvalue solvers. Extracting good floating point operation performance on linear algebra problems has been a fundamental block and the primary motivation for building faster and larger-scale supercomputers.

More recently, driven by the popularity of deep learning applications, the trend in hardware architecture has shifted towards incorporating low-precision arithmetic. With such low-precision hardware, alternative data representation or approximation methods are available, and they are more efficient in computing and storage costs compared with the exact linear algebra operations used. In this thesis, I focus mainly on the methods relating to the numerical low-rank approximations, that the storage format is universal for all elements in the matrix and vectors. Among the low-rank matrix approximation methods, the hierarchical low-rank approximation is a blocked and level-wise approximation method that efficiently compresses the data storage and the number of floating point operations. When a matrix block can have a reduced-data representation, the matrix is decomposed into the product of a series of lower-rank matrices.

Hierarchical low-rank approximations can reduce the complexity of these dense linear algebra operations to close to $\mathcal{O}(N)$, a reduction from the $\mathcal{O}(N^3)$ floating point operations used by the exact dense arithmetics while retaining the necessary accuracy. Many properties of the dense matrix, such as the problem conditioning, the eigenvalues, and the orthogonality, have been kept from the direct approximation of the matrix entries.

The critical observation is that most matrices used in practice have some structure (unless they have entirely random entries). Exploiting this structure by partitioning the matrix hierarchically into blocks allows us to approximate the off-diagonal blocks with low rank. Such hierarchically low-rank matrices have been used to compute matrix multiplication and factorization. However, factorization of these hierarchical matrices is a highly challenging problem.

Not all dense matrix shows hierarchical low-rank compressibility, and we are more likely to find compressible dense matrices arise in applications computed from an all-to-all distance-based

interaction force. Boundary element methods and Nyström methods are used in many scientific computing applications, where the discretization can be performed on a low-dimensional manifold in a higher dimensional space. These methods significantly reduce the number of unknowns but turn the otherwise sparse matrix into a dense matrix. In these applications, the resulting dense matrix from the discrete partial differential equation constructed from Green's function is directly factorized and solved.

Another application where dense matrices are used is the convolution of kernel functions, which frequently arise in image processing, statistics, and machine learning. The kernel functions can consist of Gaussian distributions and other radial basis functions. Due to the likely very severe problem conditioning, these matrices require an explicit factorization as the preconditioner of the Krylov subspace iterative solvers such as GMRES in most cases.

This thesis proposes an efficient implementation of one of the approximate linear algebra operations that directly solve dense linear systems from specific problems with close to $\mathcal{O}(N)$ complexity while achieving controllable error.

1.1 Background and related work

The work covered throughout this thesis was built upon previous knowledge of two significant aspects: running linear algebra applications on high-performance environments using tools and hardware such as OpenMP, MPI, CUDA, and numerical algorithms made explicitly for structurally low-rank approximated matrix formats, such as BLR, HSS, and $\mathcal{H}^2$-matrices, etc. This section organizes and lists only the most fundamental background research as an introductory section to promote a better understanding of the thesis's contents. In the later chapters, where the contribution of this work is described in more detail, the research of more specific relevance is also provided.

1.1.1 Background in linear algebra packages and HPC environments

The factorization of dense matrices has $\mathcal{O}(N^3)$ algorithmic complexity, and the factorization complexity is greatly influenced by the matrix multiplication algorithm that it is using internally[27], which typically is $\mathcal{O}(N^3)$ for dense matrices as well, unless adopting different matrix multiplication methods, such as using the Strassen matrix multiplication with the complexity of $\mathcal{O}(N^{2.80})$[22]. In practice, having an algorithmic complexity close to $\mathcal{O}(N^3)$ does not apply to any problem sizes, especially on the previously mentioned force fields and fluid dynamics simulations.

Optimizations for dense solves have a long history of development, starting on the block LU factorization[24], all the way to the modern linear algebra libraries that utilize GPUs, such as CUTLASS[43], cuBLAS[44] and MAGMA[25], already have very efficient methods doing many dense matrix calculations, but they have minimal support on $\mathcal{H}$-matrix formats.

One of the more recent trends in HPC development is the introduction of performance portability tools. Starting with Kokkos and Raja and later SYCL, these tools require the developer to write the code only once and can run on different hardware platforms, irrespective of the CPU or GPU configurations.

1.1.2 Background in structured low-rank approximation algorithms

The dense matrices used as the input are generated from dedicated kernel functions for many simulations and similar applications. The Fast Multipole Method, often abbreviated as FMM, has used approximation methods on the input clusters with far enough distances between them to reduce the overall complexity of operating on such large and dense matrices to reduce the complexity of matrix-vector multiplications to $\mathcal{O}(N)$[54, 53]. Broadening up into more generalized matrix forms, as long as the matrices are generated from a kernel function, such as the Green's function in the FMM case, these matrices then can be sliced into parts where the blocks can become numerically low-rank if it has far distances between the two input clusters, which makes the large dense matrix capable of being compressed into a hierarchically low-rank format.

The hierarchical low-rank matrix, or $\mathcal{H}$-matrix, is a useful representation of compressed large matrices that combines dense and low-rank blocks to store matrices according to their ranks with controllable numerical accuracy.[27] By operating directly on the low-rank blocks located on the hierarchy, the algorithmic complexity reduces drastically where the rank k is significantly less than the matrix dimension N. Multiplying a low-rank matrix with a dense matrix scales by $\mathcal{O}(N^2k)$, and the product between two low-rank matrices the problem scales by $\mathcal{O}(Nk^2)$ comparing to the $\mathcal{O}(N^3)$ complexity in dense matrix multiplications. However, factorizing the matrix in the hierarchically low-rank compressed format without converting it back and doing it fast and efficiently is not simple.

Highly paralleled hierarchical matrix factorization is more challenging than hierarchical matrix multiplications due to the data dependencies existing between the operations among the blocks. One reason is that matrix factorization includes different routines that use other parts of the matrix blocks internally, causing the data dependency structure to become complicated, and ignoring data dependencies causes incorrect factorization results. Another challenge arises from the hierarchy levels, which are different-level tasks with different-sized blocks, resulting in the problem sizes becoming different. Launching many processes from the dense libraries solving small-sized tasks introduces large overheads. It makes load balancing on different processors extremely difficult, which could eventually build up into a performance-costly move.

$\mathcal{H}$-matrix libraries, on the other hand, utilize CPUs primarily, such as H2LIB and HLIBpro[12]. There are many optimization efforts[36] done to make the hierarchical solver faster and more efficient, as well as more salable and parallizable. GPU-based $\mathcal{H}$-matrix library, hmglib[29, 56], used

batched operations at different levels compresses dense matrices generated from the boundary element method (BEM) kernel and used it for general multiplications of matrices.

$\mathcal{H}^2$-matrix[11], as an extended format of the $\mathcal{H}$-matrix, utilizes the nested basis structure to introduce further algorithmic complexity and storage cost reduction. Direct dense solver[39] that uses $\mathcal{H}^2$-matrices can already handle largely-sized problems in electromagnetic analysis, using partial LU factorization multiple times on different levels of hierarchy.

1.2 Key research question

The key question of this research is how to build an efficient dense linear system solver based on the knowledge of the hierarchical low-rank approximations. To answer this question, we can break it down into smaller sub-questions.

Firstly, the realm of hierarchical low-rank approximation contains many pieces of work that feature different matrix structures and algorithms. We need to have a thorough understanding of this background work in order to make a decision on which to choose to fit our needs. Secondly, we investigate the inhibiting factors of the current approaches that prevent them from solving linear systems efficiently and what we can do to resolve these. We expect the answer to this question to lie in the research beyond the scopes of hierarchical low-rank approximations or even linear algebra in HPC environments, and it is also possible exists a need for developing a brand-new approach. Lastly, we shall answer the following question: What details should we pay attention to to implement the solver efficiently? To elaborate on this, what tools and computing hardware we should focus on primarily, how we interface with our target numerical applications, and what types of problems other mature implementations are capable of solving, are the questions of our interests.

1.3 Objectives

In this work, we are focusing on the implementation details of the different structured low-rank matrices. The nested basis structure $\mathcal{H}^2$-matrices has been proven to be a helpful matrix format in replacing the cost of dense matrix-vector multiplication from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$. Although providing a drastic decrease in time and memory complexity of the matrix-vector operations, we still pursue the capability of direct matrix factorization. Established work on the matrix factorization on nested basis structure matrices has focused only on the weakly admissible structures, such as the hierarchical-semi-separable (HSS) matrix and the BLR^2 matrix, utilizing a differently formulated ULV factorization instead of the ordinary LU or Cholesky factorization. Despite the limitation in the applications that the weakly admissible structures can handle, the inherently parallel nature and its ideal $\mathcal{O}(N)$ of the direct factorization has attracted many researchers to dig deeper into this field. In this thesis, We focus on the $\mathcal{H}^2$-matrices, the strong admissibility variant of the nested basis

structured matrices, while introducing the other related alternative matrix structures to compare with our approach.

We list the objective of this thesis as follows:

- The development of a variant to the ULV factorization algorithm for $\mathcal{H}^2$-matrices that has optimal complexity of $\mathcal{O}(N)$ and inherently parallel nature.
- The comparison between the newly developed factorization algorithm with the existing approaches and other matrix formats, such as the ULV factorization for HSS matrices.
- Providing sounding mathematical proof on the correctness of the factorization and clearly showing the numerical low-rank properties as well as the influence of the problem conditioning.
- Experiments on largely parallel supercomputers to support the analysis of parallel scalability and computational complexity.

While comparing the similarities and differences between the two formats, we assimilate the useful techniques on performing basic arithmetic functions, to reuse and reorganize them into a complete form of LU factorization, which is a well-known matrix factorization form we have chosen. While implementing the features, we keep the efficiency of the routines we use as a priority, as well as coming up with accurate results that is comparable to dense matrix routines. Lastly, we will explore extending into a larger problem and a larger scale of parallelization and optimizations.

1.4 Contributions

From this work, we have explored the direction of performing a LU factorization directly on the hierarchical formats and have achieved satisfactory results. The contribution of this thesis in all of the results provided is summarized in follows:

- Proposed and implemented a dense direct solver based on the existing inherently parallel HSS-ULV factorization algorithm with optimal $\mathcal{O}(N)$ complexity for kernel matrices on 3-D geometry.
- Along with the factorization algorithm, we have developed the first inherently parallel forward and backward substitution algorithm for the triangular factorized matrix.
- Running only on CPUs, our inherently parallel factorization algorithm has achieved over 10,000 times speed up on supercomputer ABCI compared to an efficient implementation of the BLR-Cholesky algorithm on a linear system with 950,000 unknowns.
- We provided the only implementation of dense direct solver for kernel matrices in the market that can run on multiple NVIDIA GPUs, and achieved 0.8 PFLOPS performance using 512 nodes of ABCI for matrix factorization.

1.5 Outline of the thesis

We have organized the thesis into the following chapters.

- Chapter 2 includes the basics of the related topics of low-rank approximation and matrix factorization, including the LU factorization of a dense matrix and its block-partitioned variant, two different low-rank approximation methods singular value decomposition with truncation and interpolative decomposition, and the formation of a hierarchically low-rank approximated matrix.

- Chapter 3 introduces the concept of using the shared basis and the hierarchical low-rank approximation. A shared basis is a concept derived from the method of Fast Multipole Method (FMM), that the same basis matrix can be used for the low-rank representations of all matrix blocks located on the same row or column. The shared basis representation of the hierarchical low-rank matrices develops a specialized matrix factorization technique, the ULV factorization, which features an optimal $\mathcal{O}(N)$ algorithmic complexity and an embarrassing parallelism for weak admissibility.

- Chapter 4 presents the first half of my contribution to the society of high-performance matrix factorization of hierarchical low-rank approximated matrices. In this work, I used a two-step approach to strongly admissible matrix factorization, where the prior step removes the dominating data dependencies of the second step, to reduce the length of the critical path from $\mathcal{O}(N)$ to $\mathcal{O}(\log N)$. Although at the cost of using more FP64 operations, the new algorithm efficiently utilizes many processors and shows better parallelism than the best existing practices for complex 3-D geometry.

- Chapter 5 presents the latter half of my contribution, that moving beyond the work done on CPUs and a distributed memory environment, the newly developed algorithm is also very suitable for running on many GPUs.

- Chapter 6 concludes the contents included in this thesis and introduces visions of futuristic work.

Chapter 2

Background on hierarchical low-rank approximation

The method introduced in this thesis lies in the larger categories of dense direct linear system solvers. In particular, any linear system solver is either a direct solver or an iterative solver, solving either a dense or sparse linear system. The standard approach for inverting a dense linear system is by performing LU factorization so that $A = LU$, where A is the dense linear system and L and U are its lower and upper triangular factors correspondingly. After the factorization, the user applies the forward and backward substitution $Ly = b$ and $Ux = y$ to find the solution to the equation $Ax = b$.

The standard approach of the dense LU factorization features a memory complexity of $\mathcal{O}(N^2)$ and a compute complexity of $\mathcal{O}(N^3)$. Although bounded under polynomial time, the exponent of N made such large dense factorization techniques impracticable for problems with extreme sizes. As the usage of deep learning inspires the recent trend of computer hardware development, numerical computations can also benefit from using low-precision arithmetics. In this thesis, instead of targeting any dense matrix with arbitrary entries, we focus on the distance-related kernel matrices for solving discretized partial differential equations, which expose numerically low-rank sub-matrix structures when the distance is far away.

In this chapter, we describe matrices and the kernels our method suits and move on to the different structure low-rank matrix formats. Over the years of development of differently structured low-rank matrices, people have made both the factorization and the substitution cost of the matrices approaching $\mathcal{O}(N)$ from multi-leveled low-rank approximation technique and the usage of nested basis.

Unless otherwise specified, throughout this thesis, we use the methods defined in the Eigen3[51] for matrix slicing and blocking, a linear algebra library written in C++. The basic means for creating a matrix slice in Eigen3 is via the `.block(y,x,M,N)` method, which corresponds to a $M \times N$ matrix slice with the (y, x) indexed element located at the top left corner and the $(y + M - 1, x + N - 1)$ element on the bottom right.

In the descriptions of algorithms, we use the 1-based indexing system instead of the 0-based, the default configuration for Eigen3. The reason for using the 1-based indexing is to be consistent with the conventions for BLAS and LAPACK, which are written in the Fortran programming language initially. Many applications and published literature in the field of numerical linear algebra computations follow this convention for better readability to the users compared to the 0-based indexing.

The 0-based indexing system is advantageous in implementations since it directly translates into pointer address displacements, such as in assembly, C, and C++. However, it is unnatural for most mathematicians and people who are not working in computer science. In the later chapters, as we discuss the specific implementations of the algorithm instead of looking only at the math, we switch back to the 0-based system for our readers to have a better sense of grasping the details of the implementations when looking at the source code, especially in the looping ranges.

In addition to the `.block` method, we also use other methods including shortcuts in the blocking or the different calls to the different matrix decomposition, having the assumption that the input matrix has dimensions $M \times N$. The listed methods have parameters in the 1-based indexing, that are not the exact function calls in Eigen3.

- `.Row(y)` and `.Col(x)`: equivalents are `.block(y,1,1,N)` and `.block(1,x,M,1)`. The y-th row and the x-th column as vectors.
- `.topLeftCorner(K,L)`, `.topRightCorner(K,L)`, and more: equivalents are `.block(1,1,K,L)` and `.block(1,N-L+1,K,L)`. The corner blocks as the names suggest.
- `.lu()`, `.qr()`, and more: The LU factorization, QR factorization, and other commonly used dense factorization of the input matrix.

2.1 Methods of Low-rank Approximation

The rank of the matrix corresponds to the number of independent rows and columns the linear system contains. However, the matrix rank number is not obvious to identify just by looking at the entries. For example, a 3×3 matrix can have a rank 2 decomposition.

$$\begin{pmatrix} 4 & 8 & 1 \\ 2 & 7 & 2 \\ 2 & 9 & 3 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 2 & 1 \\ 3 & 1 \end{pmatrix} \begin{pmatrix} 0 & 2 & 1 \\ 2 & 3 & 0 \end{pmatrix} \tag{2.1}$$

By decomposing the original matrix into a product of two or more matrices of smaller dimensions, we have exposed the matrix rank number as one of the dimensions, which is evident to us just by looking at its composition. Using the low-rank decomposed format introduces many advantages in reducing storage and algorithmic complexity. For a $M \times N$ matrix with rank k, where k is significantly smaller, instead of storing all $M \times N$ entries in the matrix, the low-rank approximated

factors have only $k(M+N)$ entries in total. For a low-rank matrix provided as a dense matrix or a function, the compute cost of the low-rank approximation can vary from linear $\mathcal{O}(N)$ to cubic $\mathcal{O}(N^3)$ despite the resulting storage cost being $\mathcal{O}(N)$. The later sections are ordered to list the different methods for computing a low-rank representation from the classical and more generalized means to the problem-specific approaches.

2.1.1 Classic singular value decomposition

The most costly but widely used method, the singular value decomposition, decomposes the matrix into a series of matrix products $A = USV^H$, that U and V^H both are orthogonal, and unitary in the cases of complex entries. The S matrix in the middle is a diagonal matrix with all positive real entries, which is usually referred to as the coupling matrix. Each element in the matrix A can be recovered from the following equation

$$A(i,j) = \sum_{n}^{N} U(i,n)S(n,n)V^H(n,j) \approx \sum_{n}^{k} U(i,n)S(n,n)V^H(n,j) \tag{2.2}$$

Since the two basis matrices U and V^H are unitary matrices where the 1-norm and ∞-norm are all exactly 1, they are matrices with full ranks we can know in advance. The truncation threshold, either by the number of wanted singular values (fixed matrix rank), or the magnitude of the singular value entries (fixed approximation accuracy), is applied to the decomposed singular values inside the matrix S. We can throw away all of the redundant entries in the bottom-right corner of the diagonal matrix S along with the columns inside the orthogonal bases U and V^H, to recover the original entry in the matrix A from only the largest k entries of singular value, where k is usually referred to as the numerical rank of A. The low-rank approximation is completed from this truncation, which results in the reduced size of the matrix factors.

The singular value decomposition method is fundamental in many linear algebra methods, and its implementation can be found in any numerical libraries implementing LAPACK functions, such as cuSOLVER and MKL. The implementations of SVD usually sort the singular values in descending order in these numerical libraries so that the truncation does not require additional sorting and reordering.

2.1.2 Randomized Singular Value Decomposition

The ordinary singular value decomposition method requires a highly compute-intensive cost of $\mathcal{O}(N^3)$, that despite its robustness and accuracy, it cannot be fully utilized for all problems due to having approximation to become even more costly than the matrix factorization. In practice, a randomized variant of the singular value decomposition has been introduced, that reduces this cost to $\mathcal{O}(N^2)$ specifically for low-rank matrices' approximation.

The randomized singular value decomposition algorithm takes advantage of the low-rank nature of the input matrix. It uses a random matrix multiplied from the side to retain and capture the basis on the sides. The key concept is to use two separate QR decomposition on the row side and column side of input matrix, in order to greatly shrink down the size of the matrix that needs to be decomposed by singular values. When the actual rank of A is small enough, multiplying a tall and slim random matrix to the right side does not introduce accuracy loss, which means the tall and slim QR performed on the product can still reflect the true row basis of A. This process is referred to as the randomized range finder algorithm of A, and [42] specifically discusses the use and the variants of the randomized range-finding methods. Written in equations, the range finder does the following

$$(Q, R) = (A * \text{Matrix.Random}(N, k)).\text{qr}() \implies QQ^H A \approx A \tag{2.3}$$

From the random matrix multiplication and the tall and slim QR factorization, the complexity of the singular value decomposition in the middle reduces to $\mathcal{O}(k^3)$ where k is the matrix rank. When the matrix rank is much smaller than the input matrix dimension N and not linearly associated, the overall algorithm complexity is bounded by the random matrix multiplication $\mathcal{O}(N^2k)$. The complete algorithm for the randomized singular value decomposition is as follows.

Algorithm 1: Randomized Singular Value Decomposition

Input: Matrix A (size $M \times N$), rank k
Output: Matrix U, S, V^H

1 (Q, R)=$(A$*Matrix.Random$(N, k))$.qr();
2 Matrix $B = Q^H A$;
3 $(U, S, V^H) = B$.svd();
4 $U = QU$;
5 Return U, S, V^H;

2.1.3 Interpolative Decomposition

Although, in most cases, the singular value decomposition and its randomized variant give the best low-rank approximation, the singular value decomposition uses the ranks of the smallest of all other methods for the same desired accuracy. The interpolative decomposition still has its unique advantage in that the left or right side matrix contains the original entries of the rows or columns, after the low-rank approximation. In practice, an interpolative decomposition can be formed from the rank-revealing QR decomposition, from the column-pivoting QR decomposition `geqp3` routine inside LAPACK, or the `.colPivHouseholderQR()` method from the Eigen3 library. From the column-pivoting QR decomposition, it only takes another triangular matrix solve, `trsm` from BLAS, and a reordering of the columns, to form the interpolative decomposition.

The column-pivoted QR decomposition takes in the matrix A and produces a decomposition of $AP = QR$, where Q is an orthogonal/unitary matrix, R is upper triangular, and P is a series of

integers indicating the change of columns. The main difference in the outputted matrices between the column-pivoting QR and the classic QR decomposition is that the R matrix has entries in the diagonal being sorted in the descending order in the magnitude, granting it a rank-revealing characteristic. Written in equations gives that

$$AP \approx Q \begin{pmatrix} R_1 & R_2 \\ 0 & 0 \end{pmatrix} \tag{2.4}$$

The interpolative decomposition of the matrix A is formed via a selection of k rows or columns A' directly from the matrix A, multiplied by another matrix Y with k by N or M by k in sizes. This section focuses on the description of the column-side interpolative decomposition, where $A \approx A'Y$ since it can be constructed directly from the result of the column-pivoting QR decomposition. From the column-pivoting QR decomposition, the matrix $\begin{pmatrix} R1 & R2 \end{pmatrix}$ is an under-determined linear system with rank k and can be reconstructed from the matrix multiplications $R1 \begin{pmatrix} I & R_1^{-1}R_2 \end{pmatrix}$ without losing any accuracy. The previous equation can be rewritten as follows

$$AP \approx Q_k R_1 \begin{pmatrix} I & R_1^{-1}R_2 \end{pmatrix} \tag{2.5}$$

In the above equation, Q_k is the selection of the first k columns of the unitary matrix Q. The remaining $M - k$ columns in the previous equation are safely dropped, from being multiplied with the zero matrix blocks in the decomposed R. Since the matrix $\begin{pmatrix} I & R_1^{-1}R_2 \end{pmatrix}$ has an internal identity matrix block, it is implied that the matrix product $Q_k R_1$ in the front forms some columns from the original matrix A. From this, we can finally connect the column-pivoting QR decomposition to the column-side interpolative decomposition from $A' = Q_k R_1$ and $Y = \begin{pmatrix} I & R_1^{-1}R_2 \end{pmatrix} P^{-1}$, to make $A \approx A'Y$. There is usually no need to explicitly compute the matrix A' since it is a selection of the k columns from the original, a more efficient representation is to record the first k permutations that determine the k columns, stored in the permutation matrix P. The eventual final picture of the column-side interpolative decomposition algorithm from the column-pivoting QR decomposition is presented in the following algorithm 2.

2.1.4 Linear time approximations and adaptive cross approximation

The adaptive cross approximation[57], usually abbreviated as ACA, is a low-rank approximation method with a notable advantage in the algorithmic complexity to generate a low-rank approximation. In the previously introduced SVD-based and ID-based methods, all entries of the matrix are accessed, which implies a minimal computational cost of $\mathcal{O}(MN)$ reads where M and N stands for the number of rows and columns correspondingly for of the matrix being approximated. In contrast, the ACA method builds a rank 1 skeleton of the low-rank approximation at each iteration, accesses, and computes exactly one row and one column of the original matrix. As a result, ACA utilizes only $\mathcal{O}(k^2(M + N))$ floating point operations to generate a low-rank approximation. The algorithmic flow of the ACA method is listed in the algorithm 3 below.

Algorithm 2: Interpolative Decomposition

Input: Matrix A (size M * N), tolerance *tol*
Output: Matrix Y, matrix rank k and a list of k integers

$(Q, R, P) = A$.colPivHouseholderQR();
$k = \min(M, N)$;
for $R_{i,i}$ *in the diagonals of* R **do**
  if $R_{i,i}/R_{1,1}$ *is smaller than tol* **then**
    $k = \min(k,i)$;
Matrix $R_1 = R$.topLeftCorner(k, k);
Matrix $R_2 = R$.topRightCorner(k, $N - k$);
Matrix $Y = \begin{pmatrix} I & R_1^{-1}R_2 \end{pmatrix}$;
$Y = Y * P[1:k]^{-1}$;
Return Y, k, $P[1:k]$;

Algorithm 3: Adaptive Cross Approximation

Input: Matrix A (size M * N), rank k or tolerance *tol*
Output: Matrix U and V to have $A \approx UV$

$j = 0$;
for *iters from* 1 *up to* k **do**
  U.Col($iters$) = A.Col(j) - $\sum_{x=1}^{iters-1} U$.Col(x)$V(x, j)$;
  i = Index of largest magnitude element in U.Col($iters$);
  U.Col($iters$) = U.Col($iters$) / U(i, j);
  V.Row($iters$) = A.Row(i) - $\sum_{y=1}^{iters-1} U(i, y)V$.Row($y$);
  j = Index of largest magnitude element in V.Row($iters$);
  if $(||UV||^{iters} - ||UV||^{iters-1})/||UV||^{iters}$ *is smaller than tol* **then**
    Return U and V;
Return U and V;

ACA was also the only algebraic and geometry-free low-rank approximation method with a complexity strictly below $\mathcal{O}(MN)$ when this thesis was drafted. Other low-rank approximation methods that can achieve the $\mathcal{O}(k(M+N))$ complexity require either the knowledge of geometry or the kernel function. There are notable approaches for producing low-rank approximations that utilize the geometry information to achieve close to linear complexity, such as the kernel-independent FMM and the proxy point method. Analytical methods, such as FMM, require knowledge of the differentiable kernel function and also have drawbacks in not utilizing the ranks as efficiently as algebraic methods. There are also attempts to mix the algebraic and analytical approaches to take advantage of both, such as the hybrid approximation method. Given that this thesis mainly addresses algorithms applied to the structured low-rank approximated matrices instead of the different constructing methods, we will not further explore the geometry-assisted or analytical ways of producing a low-rank approximation.

In the actual use cases, ACA has been shown to have vulnerabilities in several types of matrices that it cannot compress to a given accuracy. The ACA approach is still practical in compressing the matrices generated from a smooth kernel function such as the BEM method of the Laplace or the Helmholtz equation. The original publication in [57], used ACA to approximate parts of the Methods of moments solutions of integral equations.

Given the simplicity of making an implementation and its unique advantage in numerical complexity, ACA is still a method worth trying to compress an unknown matrix in the first place. The ACA method is an entry point for all structured low-rank matrix structures in the later applications of this thesis. The construction of any low-rank matrix format could not have a numerical cost below the quadratic of the matrix dimension, or $\mathcal{O}(N^2)$, without using a method that accesses only parts of the original matrix.

2.2 Structured low-rank matrices and their assembly process

In many applications, irrespective of the problem conditioning, we assumed that the input matrix for factorization and inversion has full numerical rank. The low-rank approximation methods mentioned in the previous section cannot be directly applied. However, depending on the problem formulation, many sub-blocks of the matrix can be numerically low-rank approximated. As a result, the structured low-rank matrices arise in these problem settings.

In the rest of the paper, we focus on the primary application of the structured low-rank matrices, the discretized solution of partial differential equations, such as the Laplace/Poisson and Helmholtz equations. The Green's function solutions to these PDEs have in common that the interaction strength decays as a function of the separation of the particle placement. In the boundary element method for solving PDEs, the particles are placed on the boundary of the structures to form a

relatively smooth surface. From a proper clustering and sorting method, we extract some cluster-cluster pairs that are well separated from each other to be approximated. The compressibility of far interacting particles is a fundamental condition for applying the fast multipole method, an $\mathcal{O}(N)$ algorithm for computing an approximate dense matrix-vector product.

2.2.1 Admissiblity condition

To determine which blocks are low-rank approximated, we use a concept named admissibility condition[27], usually set as a positive real number. The admissibility condition, *admis*, is the ratio from the center distance between two clusters of points divided by the average bounding box dimensions. This concept is used to determine the relative separation between the clusters, as in Green's functions of the PDEs above, an important characteristic is that the numerical rank is lowered concerning the weaker interactions from the greater separation distances.

The admissibility condition defines a condition for any two clusters of points that if the separation is enough for the interaction between them to be low-rank approximated. When the distance between clusters divided by the sum of the boundary box sizes surpasses this defined condition, the two clusters are "admissible" or, in other words, well-separated from each other. Written in the equation gives the following condition, where the `.center()` and `.diag()` methods correspond to the vectors to the cluster centers and diagonal vectors of the cluster bounding boxes.

$$admis < \frac{||x.\text{center}() - y.\text{center}()||_2}{||x.\text{diag}()||_2 + ||y.\text{diag}()||_2} \tag{2.6}$$

In the later sections of this thesis, we refer to the concept of admissibility conditions sometimes via the term "weak" and "strong". By these abbreviations, matrices that are "weakly admissible" have $admis = 0$, which makes any cluster of points $x \neq y$ satisfy the condition defined in equation 2.6. Comparatively, "strongly admissible" matrices have a constant $admis > 0$ but $admis < \infty$. This setting of the admissibility condition tightens the number of interactions where the low-rank approximation can be applied, that more dense matrix blocks will show up after the construction of the structured low-rank matrix. Examples of the weakly admissible matrices include the HODLR matrix and the HSS matrix, and the strongly admissible matrices are the $\mathcal{H}$-matrix and the $\mathcal{H}^2$-matrix.

When the clusters are partitioned in a non-overlapping fashion and contain a constant amount of particles inside, given a constant admissibility condition, each cluster only has a constant number of neighboring or non-admissible clusters. In contrast, the number of admissible clusters grows with problem size $\mathcal{O}(N)$. For this reason, even though we may not directly apply the low-rank approximation method to the full-rank input matrix, we can still obtain a low-rank representation of the matrix using close to $\mathcal{O}(N)$ matrix entries by selectively approximating different matrix blocks.

2.2.2 The top-down dual-tree traversal method

The formation of the single-level structured matrices, BLR and BLR^2, is an all-to-all interaction between the finest partitioned clusters on the source and target sides. However, for multi-level structured hierarchical low-rank matrices, constructing from the finest level all-to-all interactions leads to a $\mathcal{O}(N^2)$ complexity construction, which is unacceptable for this type of matrices where the multiplication and factorization complexity is already close $\mathcal{O}(N)$. Therefore, this section discusses the dual-tree traversal method, originally adopted from the Fast Multipole Method, being a generic method that can be configured to adapt to a $\mathcal{O}(N)$ construction of the hierarchical structure, as well as a single-level structure all-to-all interactions.

The prerequisites and the parameters of the dual-tree traversal include the points (sources and targets) partitioned/sorted in a tree, and the admissibility condition, $admis$. The proposed partition method is via a recursive orthogonal dissection, which outputs a binary tree for problems set in any number of dimensions and produces relatively square-shaped boxes with an equal amount of particles placed in each. The admissibility condition determines the well-separateness of the clusters and the applicability of the low-rank approximations, introduced in the previous section.

The dual-tree traversal method outputs the lists of interactions, both near and far. Since the cluster of points is already multi-leveled, we now use the notation of $x_{l:i}$ to address the partitioned cluster, that the 0th level represents the not-partitioned original cluster, and the lth level has clusters that have been dissected l times. The i index indicates that the serial number of the partitioned cluster appears on that level, ranging from 1 to 2^l.

Algorithm 4: The recursive dual-tree traversal

Input: Cluster-Tree $y_{l:i}$ and $x_{l:j}$, admissibility condition $admis$
Output: list of near-cluster pairs $Near$ and far-cluster pairs Far

if *admissible*($y_{l:i}$, $x_{l:j}$, $admis$) **then**
  Far.append($y_{l:i}$, $x_{l:j}$);
else
  Near.append($y_{l:i}$, $x_{l:j}$);
  if $l < levels$ **then**
    $Near_{1,1}, Far_{1,1}$ = `dual_tree`($y_{l+1:2i}$, $x_{l+1:2j}$, $admis$);
    $Near_{1,2}, Far_{1,2}$ = `dual_tree`($y_{l+1:2i}$, $x_{l+1:2j+1}$, $admis$);
    $Near_{2,1}, Far_{2,1}$ = `dual_tree`($y_{l+1:2i+1}$, $x_{l+1:2j}$, $admis$);
    $Near_{2,2}, Far_{2,2}$ = `dual_tree`($y_{l+1:2i+1}$, $x_{l+1:2j+1}$, $admis$);
    $Near$.append($Near_{1,1}$, $Near_{1,2}$, $Near_{2,1}$, $Near_{2,2}$);
    Far.append($Far_{1,1}$, $Far_{1,2}$, $Far_{2,1}$, $Far_{2,2}$);
Return $Near$ and Far;

The algorithm 4 is a recursive function to itself, for simplicity, the provided algorithm is based on a binary tree partition of the clusters of the points. The algorithm can be easily extended to a tree of any number of children, as well as a single-level partition that the BLR matrix uses. One can feed the points inside each partitioned cluster into the kernel function, to build a $\mathcal{H}$-matrix

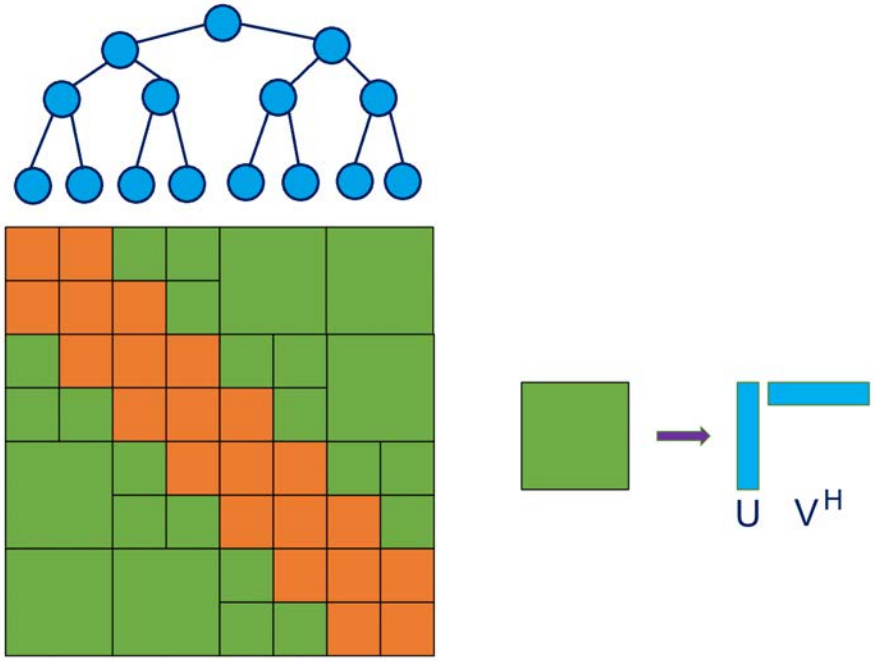


Figure 2.1: A constructed $\mathcal{H}$-matrix

from the traversed $Near$ and Far pairs. For the matrix blocks generated from the Far interaction pairs, the ideal low-rank approximation method is ACA, introduced in 2.1.4. The dual-tree method coming from FMM has a strict $\mathcal{O}(N)$ complexity for admissibility condition $admis < \infty$, and the $\mathcal{H}$-matrix construction that utilizes a linear time approximation, say the complexity is bounded by $\mathcal{O}(M+N)$ instead of $\mathcal{O}(MN)$ for approximating a matrix of size $M \times N$, will have $\mathcal{O}(N \log N)$ cost for construction. The discussion on whether or not to use other approximation methods such as the classical SVD and the randomized SVD can come after confirming that the ACA method fails for the researched problem. Figure 2.1 includes a constructed $\mathcal{H}$-matrix from the dual-tree method with a strong admissibility condition in a simple 1-D geometry.

2.2.3 Conversion of $\mathcal{H}$-matrix to $\mathcal{H}^2$-matrix

The greatest difference between a $\mathcal{H}$-matrix and $\mathcal{H}^2$-matrix is that the $\mathcal{H}^2$-matrix uses nested basis for the low-rank approximations. Shared basis and nested basis in simple words are concepts derived from the singular value decomposition, where the unitary matrices U and V^H are shared amongst different low-rank blocks located on the same row or column. The concept of nested basis is used at structured low-rank matrices with multiple levels, where the basis matrices for the upper levels are also constructed via the concatenation and translation of the lower-level bases.

The introduction of the shared basis and the nested basis originated from the Fast Multipole

Method, where the leaf-level bases correspond to the multipole-to-particle (M2P) and particle-to-local (P2L) operators, and the translations between the bases from different levels correspond to the M2M and L2L operators. For an FMM-able problem setting, the low-rank approximation of the divided matrix blocks can be directly constructed from the different types of FMM operators from the analytical expansions. This thesis focuses on the algebraic context of the structured low-rank approximated matrices, where the $\mathcal{H}^2$-matrix can also be constructed without any knowledge of calculus.

Similar to the construction of $\mathcal{H}$-matrix, $\mathcal{H}^2$-matrix can also be constructed from the low-rank approximation of different blocks of a dense matrix. In an algebraic construction, the shared or nested basis needs to include the low-rank contents of the entire row or column to build an accuracy-controllable approximation. For this reason, in addition to the level notation introduced previously, we added another symbol "+" to indicate the concatenation of all far-interacting points. For example for a 2-level partitioned cluster $y_{2:4}$, if we suppose that the admissibility condition specifies that the cluster $x_{2:3}$ and $x_{2:4}$ are close interactions, then the joint matrix for constructing the shared basis for cluster $y_{2:4}$, $A_{2:4,+}$, is made of the following

$$A_{2:4,+} = \begin{pmatrix} A_{2:4,1} & A_{2:4,2} \end{pmatrix} \implies U_{2:4}, S, V^H = A_{2:4,+}.\text{svd}() \tag{2.7}$$

Since the goal is to apply another low-rank approximation on the concatenation of a series of low-rank matrices, using the $\mathcal{H}$-matrix representation before constructing the $\mathcal{H}^2$-matrix gives an advantage in the SVD appearing in equation 2.7. By defining another matrix of $A'_{2:4+}$, which excludes the V^H bases content from the low-rank approximation in $A_{2:4+}$, the size of the SVD shrinks from being applied on a $B \times N$ matrix into a matrix of size $B \times k \log N$, where B is the leaf level block size and k is the numerical rank. Written in the equation gives the following

$$A_{2:4,+} = A'_{2:4,+} \begin{pmatrix} V^H_{2:4,1} & V^H_{2:4,2} \end{pmatrix} \implies U_{2:4}, S, V^H = A'_{2:4,+}.\text{svd}() \tag{2.8}$$

Since this procedure is applied $2^{levels} = \mathcal{O}(N)$ times to obtain each leaf level basis, using $A'_{2:4,+}$ will make the construction have an algorithmic complexity of $\mathcal{O}(N \log N)$, where the naive construction from the dense matrix using $A_{2:4,+}$ will be $\mathcal{O}(N^2)$.

The construction of the intermediate-level bases poses a slight difference from the leaf-level. Instead of storing the full-sized basis, a translation matrix is used to construct the full-sized basis from the bases in the level below. To distinguish between the full-sized basis and the translation of the basis, we denote the full-sized basis as $U^{big}_{l:i}$, and the regular leaf level basis and translation matrices as $U_{l:i}$. The translation is performed via a matrix multiplication in the equation listed below

$$U^{big}_{1:2} = \begin{pmatrix} U_{2:3} & \\ & U_{2:4} \end{pmatrix} U_{1:2} \tag{2.9}$$

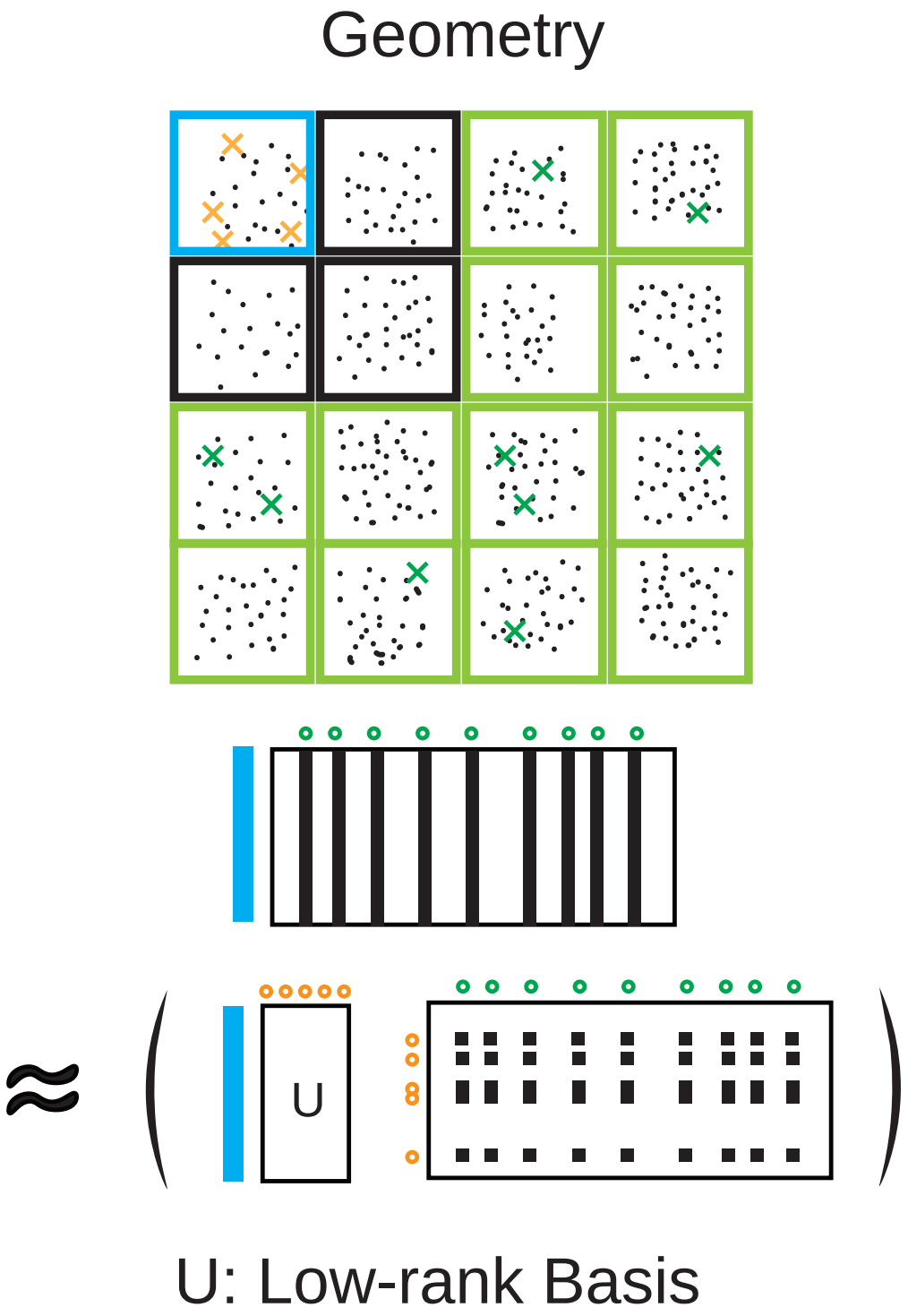


Figure 2.2: Constructing the basis U from the interpolation of the far-field particles

We could follow the same ways of constructing the leaf-level bases to form U^{big}, from the intermediate-level approximation matrices. Optimization can also apply to the procedure in forming the intermediate level bases by using interpolative decomposition introduced in section 2.1.3. The interpolative decomposition can apply the permutation matrix P returned from ID to the row-side clusters that form the approximation matrix $A'_{2:4,+}$, to construct the translation matrix U directly, and this process is depicted in the figure 2.2. However, it is not as impactful as using the low-rank approximated $\mathcal{H}$-matrix versus the dense matrix in the complexity, as using ID in constructing the intermediate level bases can only reduce one $\log N$ factor from the overall complexity, for not forming the U^{big} matrix explicitly.

After forming the shared/nested basis, the originally low-rank approximated matrix ditches the original low-rank approximation, to represent the original contents from the shared basis in addition to a local coupling matrix S. As we temporarily add the superscript "shared" to indicate the shared basis U and V different from the independently low-rank approximated ones, this procedure in equations can be written as follows

$$A_{2:4,1} \approx U_{2:4,1} V_{2:4,1}^H \approx U_{2:4}^{shared} S_{2:4,1} V_{2:1}^{shared} \tag{2.10}$$

2.3 Linear-time matrix-vector multiplication for $\mathcal{H}^2$-matrix

The introduction of the $\mathcal{H}^2$-matrix to the matrix-vector multiplication becomes one of the most fundamental but important arithmetic that can be completed in linear time complexity. As early as introduced in [11], the $\mathcal{H}^2$-matrix vector multiplication becomes an algebraic variant of the fast multipole method.

The key concept for the $\mathcal{H}^2$-matrix vector multiplication is reusing the intermediate low-rank matrix products. $\mathcal{H}^2$-matrix is sometimes viewed as an algebraic interpretation of the fast multipole method, as the connections exist in treating the multipole expansions as the low-rank approximation of the far-field forces where the matrix rank is the expansion order. The connection in the FMM, $\mathcal{H}^2$-matrix, and more algebraic $\mathcal{H}$-matrix is depicted in the figure 2.3. The FMM is the primary inspiration for doing the algebraic $\mathcal{H}^2$-matrix vector multiplication, and the $\mathcal{H}^2$-matrix refers back to FMM in constructing the algorithm for this matrix-vector multiplication. The three stages in FMM include the upward pass, the horizontal pass, and the downward pass, corresponding to the collection of all P2M and M2M, P2P and M2L, L2L and L2P operations. In the context of $\mathcal{H}^2$-matrix, we define the intermediate multiplication results using the same concepts of multipoles P

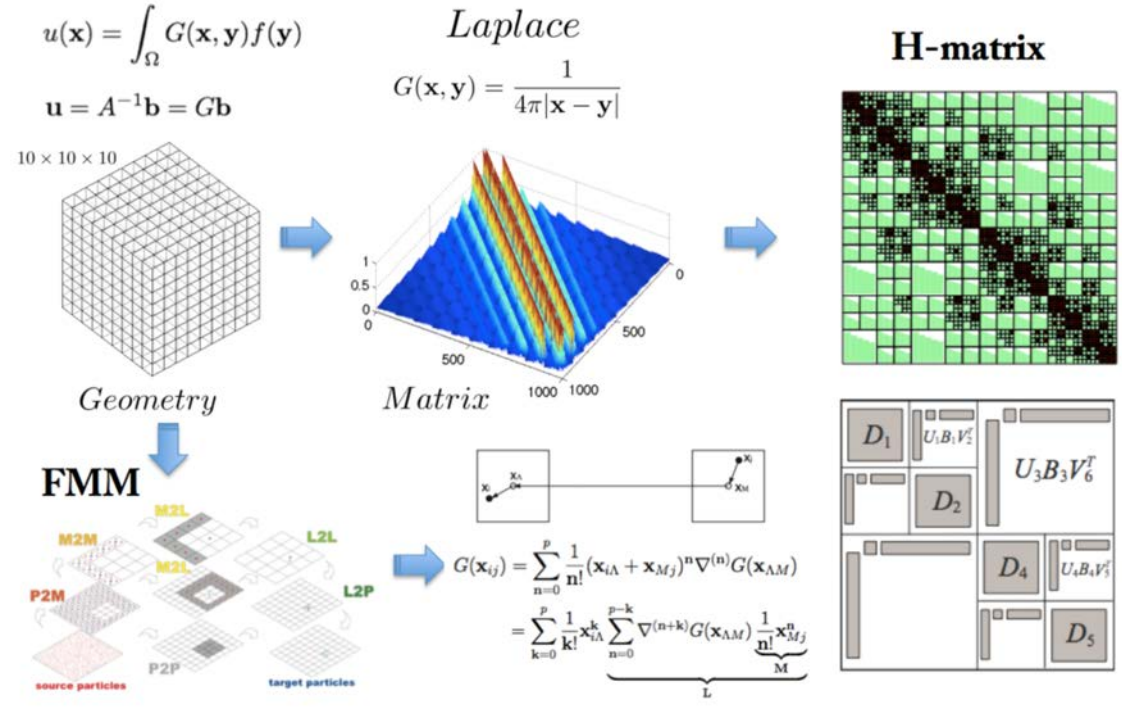


Figure 2.3: The connection of FMM, $\mathcal{H}$-matrix, to $\mathcal{H}^2$-matrix

and locals C, in the following equations

$$P_{l:j} = \begin{cases} x_j, l = levels \\ \begin{pmatrix} V_{l+1:2j} P_{l+1:2j} \\ V_{l+1:2j+1} P_{l+1:2j+1} \end{pmatrix}, l < levels \end{cases} \tag{2.11}$$

$$\begin{pmatrix} C_{l:2i} \\ C_{l:2i+1} \end{pmatrix} = \begin{pmatrix} \sum_j S_{l:2i,j} P_{l:j} \\ \sum_j S_{l:2i+1,j} P_{l:j} \end{pmatrix} + U_{l-1:i} C_{l-1:i} \tag{2.12}$$

The definitions themself already included the formation of different passes in FMM, from the formation of the upper level P using the lower level P and V (upward pass), the summation over S that passes the same level P to C (horizontal pass), and the upper level C is decomposed and added to the lower level Cs (downward pass). Lastly, the matrix-vector multiplication can be reduced to the following equations involving only the dense blocks in A, locals C, and right-hand sides x and b

$$b_i = \sum_j A_{l:i,j} x_j + U_{l:i} C_{l:i} \tag{2.13}$$

In the equations above, the linearity of the algorithmic complexity is evident, given the summation over the j variable in S and A is bounded by a constant determined by the admissibility condition, in any $l : i$ combination. The total number of intermediate Ps and Cs is also bounded by the geometric series $\mathcal{O}(N)$, and computing out each P and C requires only constant complexity in upward and downward translation and summation, making the whole algorithm $\mathcal{O}(N)$.

The linear time matrix-vector multiplication has been the most widely used application after the concept of $\mathcal{H}^2$-matrix arose in this field. The matrix-vector multiplication also has usage in obtaining the solution to linear systems from methods such as GMRES, BiCG, and power iterations.

2.4 LU Factorization and direct solution to linear system of equations

The direct approach for obtaining the solution to the linear system $Ax = b$ is via a LU factorization of the matrix A. LU factorization decomposes the matrix A into the products of two matrices L and U, each corresponding to a lower and an upper triangular matrix. The LU factorization can be obtained naturally from the Gaussian elimination process. The L matrix is a lower triangular matrix with all diagonal values having entry 1, and the U matrix is an upper triangular matrix. This decomposition is also one of the many entry points for obtaining the inverse of a matrix, due to the easiness of obtaining the inverses of two triangular matrices when compared to the impossibility of calculating the matrix inverse from a general matrix directly. In this thesis, we primarily adopt the factorization methods, including the LU factorization without pivoting and the Cholesky factorization, which factorize a symmetric and semi-positive definite matrix into a $A = LL^T$ format.

To obtain the LU factorization of a matrix, the conventional way is via the process of Gaussian elimination, named `getrf` in the LAPACK library. The Gaussian elimination visits equations one by one, and joints two equations to eliminate one unknown in each iteration. Seen in the matrix perspective, this algorithm loops from the top-left of the diagonal to the bottom-right corner of the matrix. As the elimination loops through a large matrix, it features a strong serialization and has many scattered reads in both the row and column directions, having a relatively poor performance.

The primary optimization technique introduced to the LU factorization process is matrix blocking. Blocking is a frequently seen technique in linear algebra libraries' implementations, known for its ability to improve memory efficiencies and hardware cache hit rates. The block LU factorization does not have any reduction in floating-point operations nor memory reads or writes, but it switches the order in the operations to cluster the operations to read out and write to one of the matrix blocks at a time. Aside from the `getrf` calls applied to the blocks, the block LU factorization also uses the `trsm` and `gemm` from BLAS, correspondingly stands for triangular solve for matrix and general matrix-matrix multiplication.

The block LU factorization has the following structure in algorithm 5. In the algorithm description, we use the more straightforward alternative representations for `trsm` from the multiplication of a triangular matrix inverse, and for `gemm`, the multiplication operator for matrix blocks.

Despite being an optimization technique, the block LU factorization becomes a fundamental algorithm in the realm of structured low-rank matrices. The structured low-rank matrices construct

Algorithm 5: Block LU Factorization

Input: Matrix A (size $N \times N$), block-size B
Output: Triangular matrices L and U, $A = LU$

for *i in [1, N] with increments in B* **do**
  $(L.\text{block}(i, i, B, B), U.\text{block}(i, i, B, B)) = A.\text{block}(i, i, B, B).\text{getrf}()$;
  for *j in [i+B:N] with increments in B* **do**
    $U.\text{block}(i, j, B, B) = L.\text{block}(i, i, B, B).\text{inverse}()$ * $A.\text{block}(i, j, B, B)$;
  for *j in [i+B:N] with increments in B* **do**
    $L.\text{block}(j, i, B, B) = A.\text{block}(j, i, B, B)$ * $U.\text{block}(i, i, B, B).\text{inverse}()$;
    for *k in [i+B:N] with increments in B* **do**
      $A.\text{block}(j, k, B, B) = A.\text{block}(j, k, B, B)$ - $L.\text{block}(j, i, B, B)$ * $U.\text{block}(i, k, B, B)$;

the matrix block by block, naturally using the blocking algorithms of LU factorization and matrix multiplications.

Extending the triangular matrix solves and matrix-matrix multiplications to take low-rank approximated blocks as inputs, the block LU factorization can be directly applied to the Block Low-rank matrix. In the cases of $\mathcal{H}$-matrix, where hierarchies are present, the algorithm recurses into lower-level representations by creating sub-matrix partitions. As a result, the LU factorization of the single-level BLR structure can have an algorithmic complexity of $\mathcal{O}(N^2)$, and $\mathcal{H}$-matrix has a complexity of $\mathcal{O}(Nlog^2N)$[28].

2.4.1 Limited parallel capability and the existence of a diagonal critical path

BLR-matrix, $\mathcal{H}$-matrix, and even matrices with shared basis such as HSS matrix can be directly LU factorized from the block or the recursive algorithms, but their parallel scalability has been greatly limited. The limitation in scalability comes from a chain of data dependencies of the diagonal entries from the top-left to the bottom-right corner in the Gaussian elimination process. Low-rank approximation on the off-diagonal matrix entries reduces the complexity from the reduced number of updates, but the length of the critical path cannot be further reduced. In the sense of parallel execution, with a close-to-linear compute complexity, the performance on parallel hardware is even degraded, as the length of the critical path is strictly N.

In the past research, people have made attempts to make such structured low-rank matrices run efficiently on parallel hardware, but the reported scalability has all been very limited if based on the direct LU factorization approach. Among the implementations, the simplest form of all structured low-rank matrices, the BLR matrix, has reported the best parallel scalability due to its still relatively high numeric complexity of $\mathcal{O}(N^2)$, which enables the use of tasked based runtime systems.

For more complicated formats such as the $\mathcal{H}$-matrix, the factorization complexity $\mathcal{O}(N \log_2^2 N)$ is almost identical to the critical path length, being $\mathcal{O}(N)$. In other words, the existence of the critical

path has made the compute time scalability almost impossible once the number of computing CPUs is beyond a certain point, where the off-diagonal contents of the matrix have been completely off-loaded and the diagonal computation dominates. As a result, even using the simplest 1-D row or column process mapping will not show a major disadvantage in compute time compared to more advanced approaches.

$\mathcal{H}$ACApk [32, 35] made a distributed memory parallelization attempt on the LU factorization of independent basis low-rank matrix. The authors used a format name lattice $\mathcal{H}$-matrix, that completely flattens the partitioned cluster tree, to create an intermediate BLR-matrix with dense blocks represented as $\mathcal{H}$-matrices. With the overall complexity still bounded by $\mathcal{H}$-matrix characteristics that are close to $\mathcal{O}(N \log_2^2 N)$, the lattice $\mathcal{H}$-matrix shows significantly better parallel capability that benefits from the flattening of the upper levels, that comes with the cost of using finer grain partitioning and doing practically more computation to distribute out.

Other implementations of $\mathcal{H}$-matrices other than $\mathcal{H}$ACApk mostly limited the use of parallelization hardware and libraries. Hlibpro[36] implements $\mathcal{H}$-matrix's matrix multiplications and factorization algorithms based on using the StarPU runtime system[8] and only running on CPUs and in a shared memory environment. The hmglib library[29, 56] utilized NVIDIA GPUs and used the CUDA programming language to implement the $\mathcal{H}$-matrix, but its functionality is only limited to the matrix-vector multiplications.

A recent parallelization attempt to obtain the direct solution of the $\mathcal{H}$-matrices is made by [21] to break the curse of the $\mathcal{O}(N)$ critical path. The novel factorization takes the approach of extended sparsification, which is the underlying idea for the method in the Inverse FMM[4, 23]. Only in the weakly-admissible configured version of the $\mathcal{H}$-matrix, the HODLR matrix, the authors successfully parallelize the factorization and forward/backward substitution to remove the data dependencies existing in the diagonals.

In the later chapters, this thesis focuses on the BLR^2, HSS, and $\mathcal{H}^2$-matrix, the variants of the structured low-rank approximated matrices that are constructed from the shared and nested bases. Even without discussing the removal of the $\log N$ factor existing in the multiplication and the factorization complexities, these matrices have a unique advantage in parallelization from the ULV factorization, which is going to be discussed in more detail in the next chapter.

Chapter 3

The use of shared basis and the existing best practices

The use of shared basis in structured low-rank approximated matrices originated from a matrix interpretation of the fast multipole method. Understand from a $\mathcal{H}^2$-matrix perspective, the inverse FMM[4, 23] is one of the first approaches to produce a direct factor of the matrix in $\mathcal{H}^2$-approximated format with strong admissibility. The inverse FMM uses two important concepts to achieve this goal, the extended sparsification and the real-time re-compression of the dense fill-ins to dissolve into the low-rank approximated interactions.

In this chapter, we take a different ULV factorization approach for factorizing the structured low-rank approximated matrices that use shared bases. Drastically different from the independent basis approaches based on LU or Cholesky factorizations, the parallel implementations, specifically for weakly admissible shared-basis structured low-rank matrices, use ULV factorization to achieve an inherently parallel factorization[20]. The ULV factorization in simple words, utilizes the complementary of the constructed shared bases, to project the original $\mathcal{H}^2$-matrix into a block sparse matrix in the middle, and then apply the block LU or Cholesky factorization to resolve the sparse matrix factorization. In the cases of strong admissibility configurations, the concept of real-time re-compression is introduced again in [39, 40].

3.1 Notation

We first define the common notations we use throughout this paper in Fig. 3.1. A sub-block of a hierarchically sub-divided matrix A is indexed using $A_{level;row,column}$. The shared column basis and row basis are indexed by $U_{level;row}$ and $V_{level;column}$, respectively. At the leaf level this notation is used for the shared basis and at other levels it is used to denote the transfer matrix. For example, matrix $A_{1;0,1}$ can be written in the following hierarchical low-rank form

$$A_{1;0,1} = \begin{bmatrix} U_{2;0} & 0 \\ 0 & U_{2;1} \end{bmatrix} U_{1;0} S_{1;0,1} V_{1;1} \begin{bmatrix} V_{2;2} & 0 \\ 0 & V_{2;3} \end{bmatrix}. \tag{3.1}$$

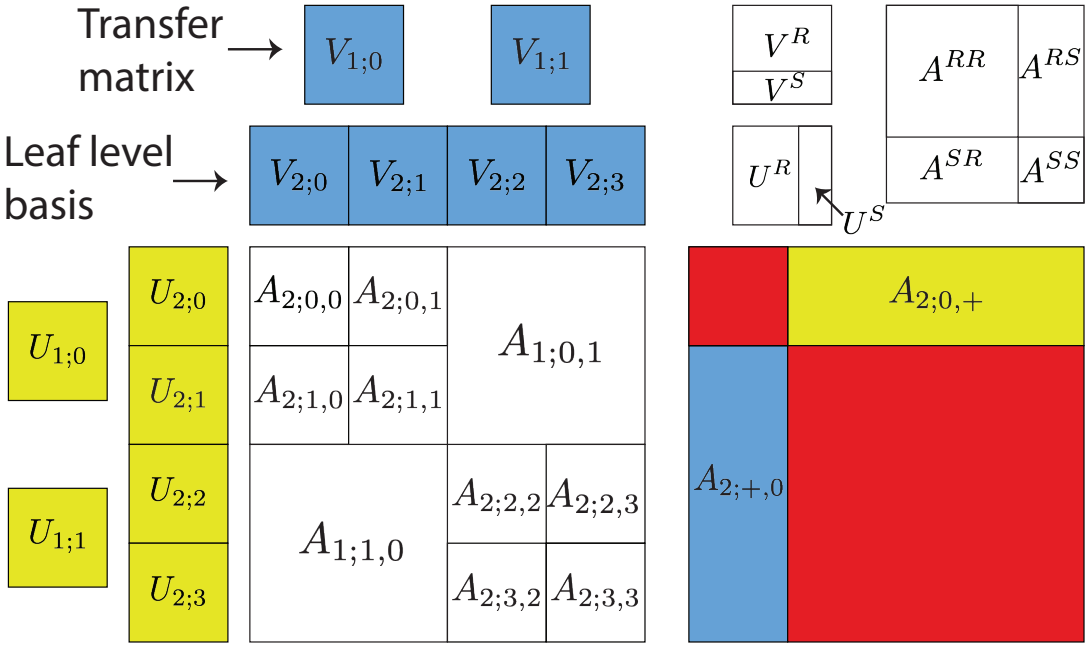


Figure 3.1: Index notation for hierarchical matrices

where $S_{1;0,1}$ is the skeleton matrix shown inside the corresponding large off-diagonal block in Fig. 4.1. We also introduce a convenient notation for expressing the concatenation of all low-rank blocks in a given row/column as shown on the right in Fig. 3.1. This is used in the first step of the algorithm to compute the shared bases

$$[U^S_{2;0}\ U^R_{2;0}], R = \mathrm{QR}(A_{2;0,+}) \tag{3.2}$$

$$[V^{S\top}_{2;0}\ V^{R\top}_{2;0}], R = \mathrm{QR}(A_{2;+,0}{}^{\top}), \tag{3.3}$$

where the S and R superscripts represent the skeleton part and redundant part of the basis, respectively. The QR() represents a column pivoted or rank revealing QR, where the R matrix is not used. This can be replaced by an interpolative decomposition [42] if that is preferred, but the loss of orthogonality will lead to other complications in this case. In order for the ULV factorization to work, we need to permute the skeleton part and redundant part to look like the shapes shown in Figs. 4.1 and 3.1. Using this split in the column and row basis, any dense or low-rank matrix can be subdivided into A^{RR}, A^{RS}, A^{SR}, and A^{SS}, as shown in the upper right corner of Fig. 3.1. Using these notations, the dense matrix A can be decomposed as

$$A = [U^R\ U^S] \begin{bmatrix} S^{RR} & S^{RS} \\ S^{SR} & S^{SS} \end{bmatrix} \begin{bmatrix} V^R \\ V^S \end{bmatrix}. \tag{3.4}$$

This is what is shown in the top-right of Fig. 3.2. Similarly, a low-rank matrix A can be

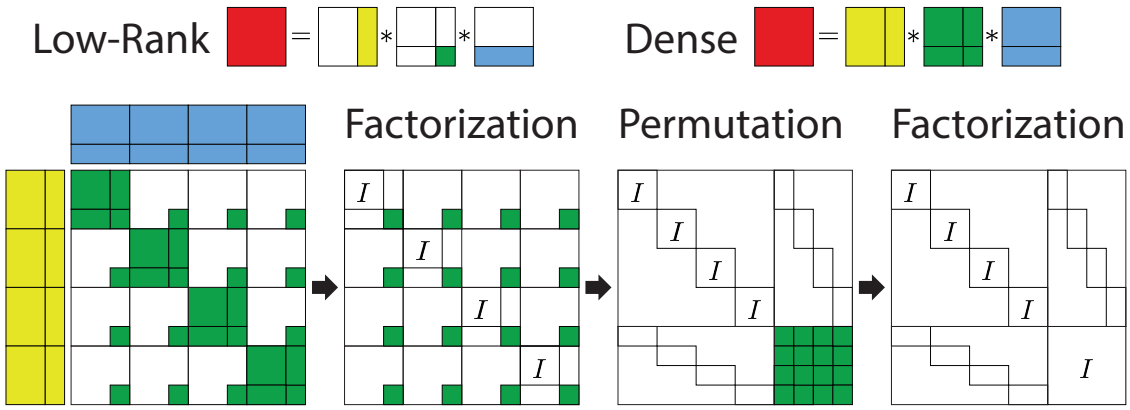


Figure 3.2: Flow of the BLR^2-ULV factorization.

decomposed as

$$A = [U^R \; U^S] \begin{bmatrix} 0 & 0 \\ 0 & S^{SS} \end{bmatrix} \begin{bmatrix} V^R \\ V^S \end{bmatrix}. \tag{3.5}$$

The bases $[U^R \; U^S]$ and $[V^R \; V^S]$ are shared among both the dense and low-rank blocks in that entire row and column, respectively. This allows us to perform a USV decomposition of the entire matrix, which is what is shown in the leftmost figure in Figs. 4.1 and 3.2. With these notations, we can proceed to describe our proposed method along with some existing ones.

3.2 BLR^2-ULV factorization

We will start out with a description of a ULV factorization for the BLR^2 structure, which a non-hierarchical version of the HSS. In the following sections, we will add hierarchy to form the HSS-ULV factorization, and then extend that to strong admissibility to obtain the $\mathcal{H}^2$-ULV factorization that we use in our current work.

Similar to the HSS-ULV factorization in Fig. 4.1 the BLR^2-ULV factorization is described in Fig. 3.2. We first obtain the shared bases by performing

$$[U^S_{2;i} \; U^R_{2;i}], R = \text{QR}(A_{2;i,+}) \tag{3.6}$$

$$[V^{S^\top}_{2;j} \; V^{R^\top}_{2;j}], R = \text{QR}({A_{2;+,j}}^\top), \tag{3.7}$$

for all rows of i and all columns of j. We then compute all the S matrices of the dense diagonal blocks as

$$\begin{bmatrix} S^{RR}_{2;i,i} & S^{RS}_{2;i,i} \\ S^{SR}_{2;i,i} & S^{SS}_{2;i,i} \end{bmatrix} = [U^R_{2;i} \; U^S_{2;i}]^\top A_{2;i,i} \begin{bmatrix} V^R_{2;i} \\ V^S_{2;i} \end{bmatrix}^\top, \tag{3.8}$$

for all the diagonal blocks $A_{2;,i,i}$. For the low-rank blocks computing the S matrices is simply

$$S^{SS}_{2;i,j} = U^{S^\top}_{2;i} A_{2;i,j} V^{S^\top}_{2;j}, \tag{3.9}$$

For all the off-diagonal blocks $A_{2;,i,j}$, where $i \neq j$. This allows us to decompose for example, the sub-matrix $A_{1;0,0}$ into

$$A_{1;0,0} = \begin{bmatrix} U_{2;0} & 0 \\ 0 & U_{2;1} \end{bmatrix} \begin{bmatrix} S_{2;0,0} & S_{2;0,1} \\ S_{2;1,0} & S_{2;1,1} \end{bmatrix} \begin{bmatrix} V_{2;0} & 0 \\ 0 & V_{2;1} \end{bmatrix}. \tag{3.10}$$

Note that although the column basis and row basis seem like tall skinny matrices in Figs. 4.1 and 3.2, they are actually block diagonal matrices as shown in Eq. (3.10). The column and row basis remain constant throughout the factorization. When a Cholesky factorization is performed only on the S matrix of the USV decomposition it is called the ULV factorization. Actually, calling it a $\mathrm{ULL}^\top\mathrm{V}$ factorization might be more descriptive. Although the original ULV factorization uses a Cholesky factorization, we may extend this to the LU factorization as well, which is what we will show in the following description.

The LU factorization on the S matrix is done by first eliminating the $S^{RR}_{2;i,i}$ blocks on the diagonal

$$\hat{L}^{RR}_{2;i,i}, \hat{U}^{RR}_{2;i,i} = \mathrm{LU}(S^{RR}_{2;i,i}). \tag{3.11}$$

We use $\hat{U}$ to distinguish the upper triangular matrix from the column basis U. This is followed by an elimination of the $S^{RS}_{2;i,i}$ and $S^{SR}_{2;i,i}$ blocks.

$$\hat{U}^{RS}_{2;i,i} = (\hat{L}^{RR}_{2;i,i})^{-1} S^{RS}_{2;i,i} \tag{3.12}$$

$$\hat{L}^{SR}_{2;i,i} = S^{SR}_{2;i,i} (\hat{U}^{RR}_{2;i,i})^{-1}. \tag{3.13}$$

Finally, the elimination for the block is completed by computing

$$S^{SS}_{2;i,i} = S^{SS}_{2;i,i} - \hat{L}^{SR}_{2;i,i} \hat{U}^{RS}_{2;i,i}. \tag{3.14}$$

The elimination of these blocks for different i can be done in parallel since there are no dependencies among them. Following the factorization phase, the $S^{SS}_{2;i,j}$ blocks that remain to be factorized are clustered in the bottom-left corner, as shown in the permutation phase in Fig. 3.2. For the BLR^2-ULV factorization, this entire remaining block is eliminated as a single dense matrix, which completes the LU factorization.

$$\hat{L}^{SS}_{2;1:4,1:4}, \hat{U}^{SS}_{2;1:4,1:4} = LU \left(\begin{bmatrix} S^{SS}_{2;0,0} & S^{SS}_{2;0,1} & S^{SS}_{2;0,2} & S^{SS}_{2;0,3} \\ S^{SS}_{2;1,0} & S^{SS}_{2;1,1} & S^{SS}_{2;1,2} & S^{SS}_{2;1,3} \\ S^{SS}_{2;2,0} & S^{SS}_{2;2,1} & S^{SS}_{2;2,2} & S^{SS}_{2;2,3} \\ S^{SS}_{2;3,0} & S^{SS}_{2;3,1} & S^{SS}_{2;3,2} & S^{SS}_{2;3,3} \end{bmatrix} \right). \tag{3.15}$$

The forward elimination and backward substitution of the resulting

$$U\hat{L}\hat{U}Vx = b \tag{3.16}$$

can be performed in the following three steps

$$\hat{L}z = U^\top b \tag{3.17}$$

$$\hat{U}y = z \tag{3.18}$$

$$x = V^\top y. \tag{3.19}$$

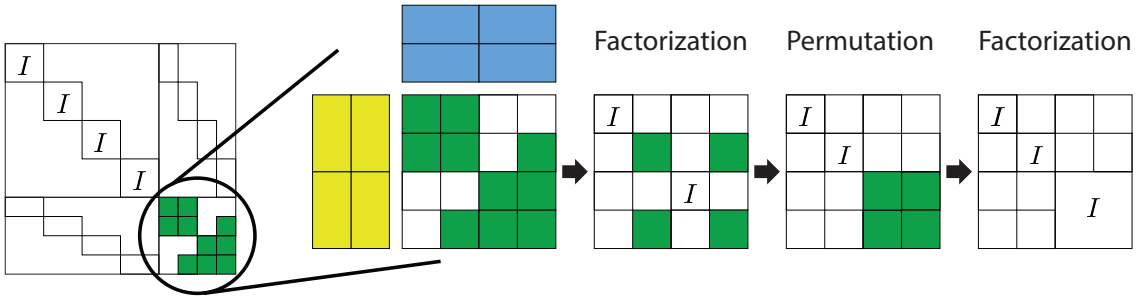


Figure 3.3: Upper levels of the HSS-ULV factorization.

3.3 HSS-ULV factorization

The HSS structure is a multi-level version of the BLR^2 structure with weak admissibility. The flow of the HSS-ULV factorization is identical to that of the BLR^2-ULV factorization until the step in Eq. (3.15). As shown in Fig. 3.3, instead of treating the leftover block as a dense block, the HSS-ULV recursively applies the same procedure to the remaining part. The column basis and row basis shown in Fig. 3.3 are actually the transfer matrices $U_{1;0}$ and $V_{1;1}$ shown in Eq. (3.1). They can be computed from

$$[U^S_{1;0}\ U^R_{1;0}], R = \mathrm{QR}\left(\begin{bmatrix} U_{2;0} & 0 \\ 0 & U_{2;1} \end{bmatrix}^\top A_{1;0,1}\right) \tag{3.20}$$

$$[V^{S\top}_{1;1}\ V^{R\top}_{1;1}], R = \mathrm{QR}\left(\begin{bmatrix} V_{2;2} & 0 \\ 0 & V_{2;3} \end{bmatrix} A^\top_{1;0,1}\right). \tag{3.21}$$

Before we compute the skeleton/redundant decomposition of the S block as in Eq. (3.8), we first need to merge the S blocks in Eq. (3.15) as follows.

$$\begin{bmatrix} S_{1;0,0} & S_{1;0,1} \\ S_{1;1,0} & S_{1;1,1} \end{bmatrix} = \begin{bmatrix} S^{SS}_{2;0,0} S^{SS}_{2;0,1} S^{SS}_{2;0,2} S^{SS}_{2;0,3} \\ S^{SS}_{2;1,0} S^{SS}_{2;1,1} S^{SS}_{2;1,2} S^{SS}_{2;1,3} \\ S^{SS}_{2;2,0} S^{SS}_{2;2,1} S^{SS}_{2;2,2} S^{SS}_{2;2,3} \\ S^{SS}_{2;3,0} S^{SS}_{2;3,1} S^{SS}_{2;3,2} S^{SS}_{2;3,3} \end{bmatrix} \tag{3.22}$$

Then we can decompose all the S matrices of the dense diagonal blocks as

$$\begin{bmatrix} S^{RR}_{1;i,i} & S^{RS}_{1;i,i} \\ S^{SR}_{1;i,i} & S^{SS}_{1;i,i} \end{bmatrix} = [U^R_{1;i}\ U^S_{1;i}]^\top S_{1;i,i} \begin{bmatrix} V^R_{1;i} \\ V^S_{1;i} \end{bmatrix}^\top. \tag{3.23}$$

For the low-rank blocks we have

$$S^{SS}_{1;i,j} = U^{S\top}_{1;i} S_{1;i,j} V^{S\top}_{1;j}, \tag{3.24}$$

Once all the S blocks at level 1 are computed, the same procedure shown in Eqs. (3.11) to (3.15) is repeated. We have shown the HSS matrix with a 4×4 subdivision as an example, but the same recursive algorithm can be extended to any number of subdivision levels and any matrix size.

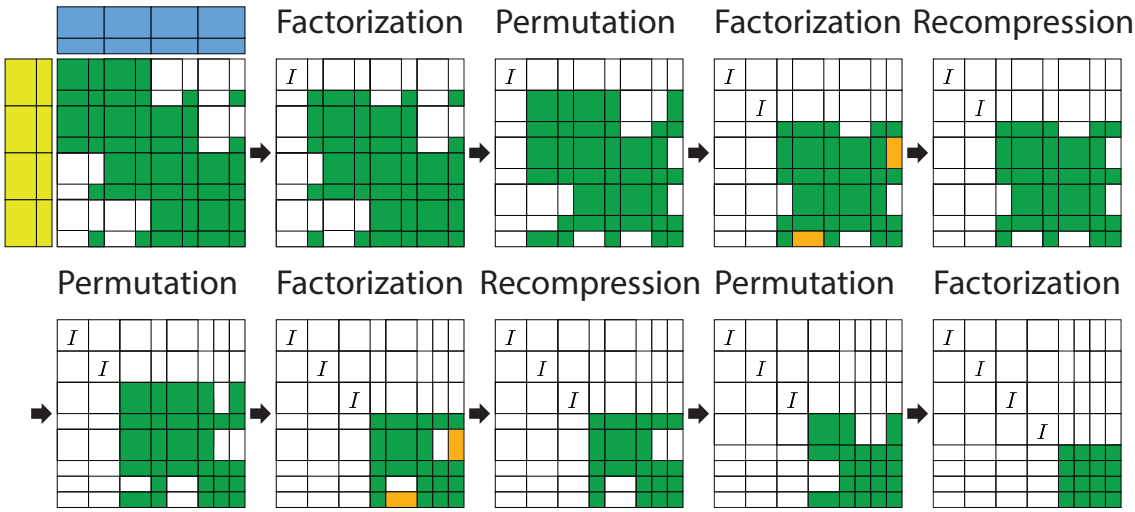


Figure 3.4: Flow of the $\mathcal{H}^2$-ULV factorization with dependencies.

3.4 $\mathcal{H}^2$-ULV factorization with dependencies

The flow of the $\mathcal{H}^2$-ULV factorization is shown in Fig. 3.4. The strong admissibility of $\mathcal{H}^2$-matrices gives dense off-diagonal blocks. The example shown in Fig. 3.4 has a block tri-diagonal structure, but $\mathcal{H}^2$-matrices can handle any pattern of off-diagonal dense blocks. The factorization cost will be $\mathcal{O}(N)$ as long as the number of dense blocks is $\mathcal{O}(N)$. However, these off-diagonal dense blocks result in trailing sub-matrix dependencies, so the factorization at each level cannot be done in parallel like the HSS-ULV factorization. This is one of the main reasons why HSS-ULV has been preferred over $\mathcal{H}^2$-ULV despite its suboptimal performance in 3-D problems. We will describe in the next section how these trailing sub-matrix dependencies can be removed, but here we will first describe the one with dependencies. For $\mathcal{H}^2$-ULV, the stages for constructing the USV decomposition given by Eqs. (3.6) to (3.9), are identical to the BLR^2-ULV factorization. The $A_{2;0,+}$ matrix in Eq. (3.6) looks like the concatenation of all *off-diagonal* blocks in Fig. 3.1, but for $\mathcal{H}^2$-matrices, it is the concatenation of all *low-rank* blocks. Dense blocks are never used for the construction of the shared basis. Eq. (3.8) is applied to all the dense blocks, not only the diagonal blocks. Otherwise, the USV decomposition is identical to that of the BLR^2-ULV factorization.

The factorization stage is quite different from that of the HSS-ULV, since the off-diagonal blocks create a trailing sub-matrix dependency. This has two consequences, 1) the serialization of the factorization and permutation steps, and 2) the existence of fill-ins that result in an extra recompression step. While the HSS-ULV shown in Fig. 4.1 has only a single factorization step and permutation step per level, the $\mathcal{H}^2$-ULV performs a series of factorization, recompression, and permutation steps for each block row/column. The fill-in blocks are shown in orange in Fig. 3.4. These fill-in blocks can be eliminated by merging them back into the row/column basis. For example for the fourth

matrix from the left on the top row of Fig. 3.4, there are two fill-in blocks. The recompression of these blocks involve the following updates to the column basis and row basis as

$$[U^S_{2;2}\ U^R_{2;2}], R = \mathrm{QR}\left(\left[U^S_{2;2}S^{SS}_{2;2,0} \quad [U^R_{2;2}\ U^S_{2;2}]\begin{bmatrix}S^{RS}_{2;2,0}\\ S^{SS}_{2;2,0}\end{bmatrix}\right]\right) \tag{3.25}$$

$$[V^{S\top}_{2;2}\ V^{R\top}_{2;2}], R = \mathrm{QR}\left(\begin{bmatrix}S^{SS}_{2;0,2}V^S_{2;2}\\ [S^{SR}_{2;0,2}\ S^{SS}_{2;0,2}]\begin{bmatrix}V^R_{2;2}\\ V^S_{2;2}\end{bmatrix}\end{bmatrix}^\top\right), \tag{3.26}$$

where $U^S_{2;2}S_{2;2,0}$ and $S_{2;0,2}V^S_{2;2}$ are products of the basis and the skeleton matrix. For this particular case, there is only one low-rank block in this row/column, but in the general case it is necessary to concatenate this product for all low-rank blocks in the row/column that is being recompressed. The operations in Eqs. (3.25) and (3.26) allow the fill-in to be incorporated into the U and V bases, so that it can be eliminated from the S matrix. All fill-in blocks are eliminated as soon as they fill-in, so the sparsity of the skeleton matrix is retained. By comparing Fig. 4.1 and 3.4, it is clear that the $\mathcal{H}^2$-ULV is much more complicated and much less parallel.

Chapter 4

Data-parallel ULV factorization for $\mathcal{H}^2$-matrix on many CPUs

Factorization of dense matrices has been at the heart of high performance computing where the high performance LINPACK (HPL) benchmark has been used to track the performance of the Top500 supercomputers for nearly 30 years. Even though HPL is often criticized for not being representative of the actual workloads in HPC, there exist many applications that require the factorization of large dense matrices. For example, preconditioners for iterative boundary integral solvers [48], frontal matrices in sparse multifrontal solvers [38, 6], and computing the determinant of covariance matrices in statistics [37]. The dense matrices that arise in these problems have a low-rank structure that can be exploited to perform approximate factorization in linear time. The accuracy of the approximation is controllable by adjusting the rank or the admissibility. The admissibility condition determines how far to subdivide the blocks, and which ones to consider as dense blocks. The hierarchical semi-separable (HSS) matrix [20] has weak admissibility, where only the diagonal blocks are subdivided recursively, and off-diagonal blocks are approximated with low-rank matrices. Since the numerical rank of each block is determined by the proximity of the sub-domains in the underlying geometry, for 3-D problems this large off-diagonal block in HSS will contain many sub-blocks that have large rank. This causes the rank of off-diagonal blocks to grow with the problems size, and the method would no longer have linear complexity. The $\mathcal{H}^2$-matrix [28] on the other hand has strong admissibility, where the off-diagonal blocks can be subdivided further so that each block can maintain a somewhat constant rank that is independent of the problem size. For the off-diagonal sub-blocks that have close proximity in the geometry will be subdivided until the leaf-level is reached, at which point it will be treated as a dense block. There are only a constant number of such off-diagonal dense blocks per row/column, due to the number of neighboring boxes in the geometry being constant and independent of the problem size. This allows the $\mathcal{H}^2$-matrix to achieve linear complexity even for 3-D problems. Both, HSS and $\mathcal{H}^2$-matrices share the basis among the low-rank blocks in the same block row/column. These shared bases are also nested between the levels so that the bases

of large low-rank blocks do not need to be stored explicitly. Variants that use independent bases for each low-rank block also exist, *e.g.* HODLR (weak admissibility) [3] and $\mathcal{H}$-matrix (strong admissibility) [26]. There are also non-hierarchical variants such as BLR (independent basis) [5] and BLR^2 (shared basis) [7]. Table 4.1 summarizes the various types of structures categorized by their basis and admissibility, along with the factorization complexity.

The different structures each have their pros and cons. The non-hierarchical structures such as BLR and BLR^2 are obviously much simpler to implement, but cannot achieve the near-linear complexity that hierarchical structures enjoy. It is worth noting that frontal matrices in multifrontal solvers have a size of $\mathcal{O}(N^{\frac{2}{3}})$ for 3-D problems, so reducing the complexity of factorizing this part to $\mathcal{O}(N^2)$ with BLR is enough to achieve an overall complexity of $\mathcal{O}(N^{\frac{4}{3}})$ for the multifrontal solver. Sharing the basis among the block rows/columns reduces the memory consumption, but requires the basis for the whole block row/column to be updated every time a trailing sub-block is updated. The ULV factorization for HSS matrices [20] avoids this problem by factorizing only the redundant part of the dense blocks at each level.

As shown at the top of Fig. 4.1, low-rank blocks can be decomposed into the shared column basis, skeleton matrix, and shared row basis. We can then add back the redundant part of the basis to the shared column basis and shared row basis to form a square matrix. These bases are used to compute the skeleton matrix for the dense block, which are partitioned into four parts. This allows us to decompose a dense matrix into a sparsifed structure shown on the leftmost side of Fig. 4.1. This block sparse matrix can be factorized without trailing sub-matrix dependencies, if we leave the skeleton matrices to be factorized at the next level. This has huge implications with regards to both the complexity and parallelism of the algorithm. The sparsification results in $\mathcal{O}(N)$ compleixity, and the removal of dependencies allows the diagonal blocks to be factorized in parallel. Concepts such as right-looking and left-looking become irrelevant, since there are no GEMM operations nor trailing sub-matrix dependencies required in this method. Runtime systems such as StarPU [8] and PaRSEC [14] and coloring schemes to extract parallelism are also unnecessary since the method has no dependencies.

However, the procedure illustrated in Fig. 4.1 only works for HSS matrices, which has suboptimal complexity for problems with 3-D geometry, as mentioned earlier. On the other hand, $\mathcal{H}^2$-matrices

Low-rank structure	Basis	Admissibility	Complexity
BLR [5]	Independent	Strong or weak	$O(N^2)$
BLR^2 [7]	Shared	Strong or weak	$O(N^{1.8})$
HODLR [3]	Independent	Weak	$O(N \log^2 N)$
$\mathcal{H}$-matrix [26]	Independent	Strong	$O(N \log^2 N)$
HSS [20]	Shared	Weak	$O(N)$
$\mathcal{H}^2$-matrix [28]	Shared	Strong	$O(N)$

Table 4.1: List of Different Low-rank Structures

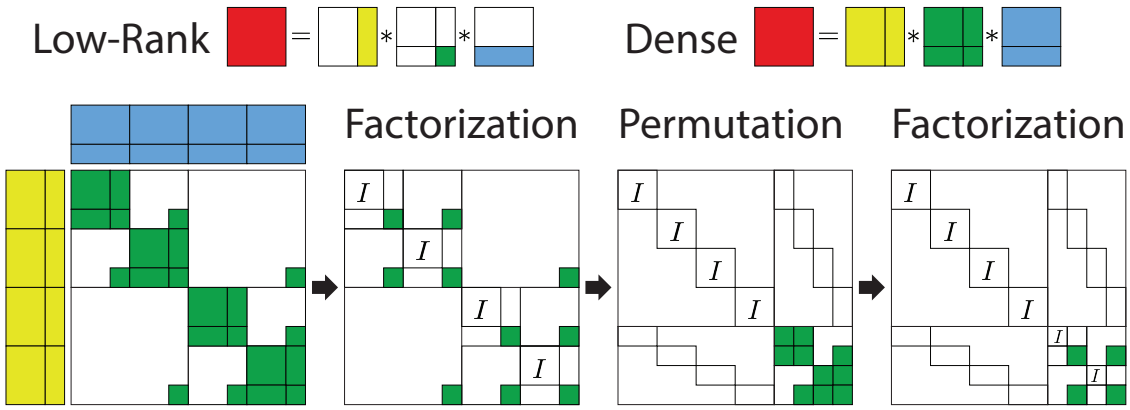


Figure 4.1: $\mathcal{O}(N)$ Dense Factorization Without Trailing Sub-Matrix Dependencies

can still achieve linear complexity even for 3-D problems. However, having dense blocks in the off-diagonal will result in fill-in during the factorization, which introduces the dependency on the trailing sub-matrices to this otherwise highly parallel factorization method. In the present work, we pre-compute the fill-ins and integrate them into the shared basis, in order to factorize an $\mathcal{H}^2$-matrix without any dependency on the trailing sub-matrices. Unlike existing methods that rely on coloring to extract parallelism from an $\mathcal{H}^2$-matrix factorization, our method is inherently parallel.

We make the following contributions in the present work:

- We extend the HSS-ULV factorization to $\mathcal{H}^2$-matrices, which has $\mathcal{O}(N)$ complexity even for 3-D problems.
- We do this while retaining the parallelism of HSS-ULV by pre-computing the fill-ins and including them in the shared basis.
- This is the first method that can factorize dense matrices arising from 3-D problems in linear time without trailing sub-matrix dependencies.

4.1 $\mathcal{H}^2$-ULV without dependencies

Existing methods without trailing sub-matrix dependencies such as GOFMM [55] and STRUMPACK [47] are limited to weak admissibility *e.g.* HSS matrices. For 3-D problems, the rank of the off-diagonal blocks in an HSS matrix grows as a function of N, so the $\mathcal{O}(N)$ complexity of the algorithm is lost. On the other hand, $\mathcal{H}^2$-matrices [40, 13] with strong admissibility do not have this problem, and are able to achieve $\mathcal{O}(N)$ even for problems with 3-D geometry. The $\mathcal{H}^2$-matrices having dense off-diagonal blocks is both a blessing and a curse, since this is what allows the low-rank off-diagonal blocks to have $\mathcal{O}(1)$ ranks, while it is also the source of fill-ins that ultimately serialize the factorization process. Previous implementations of $\mathcal{H}^2$-matrix factorization such as IFMM [48] depend

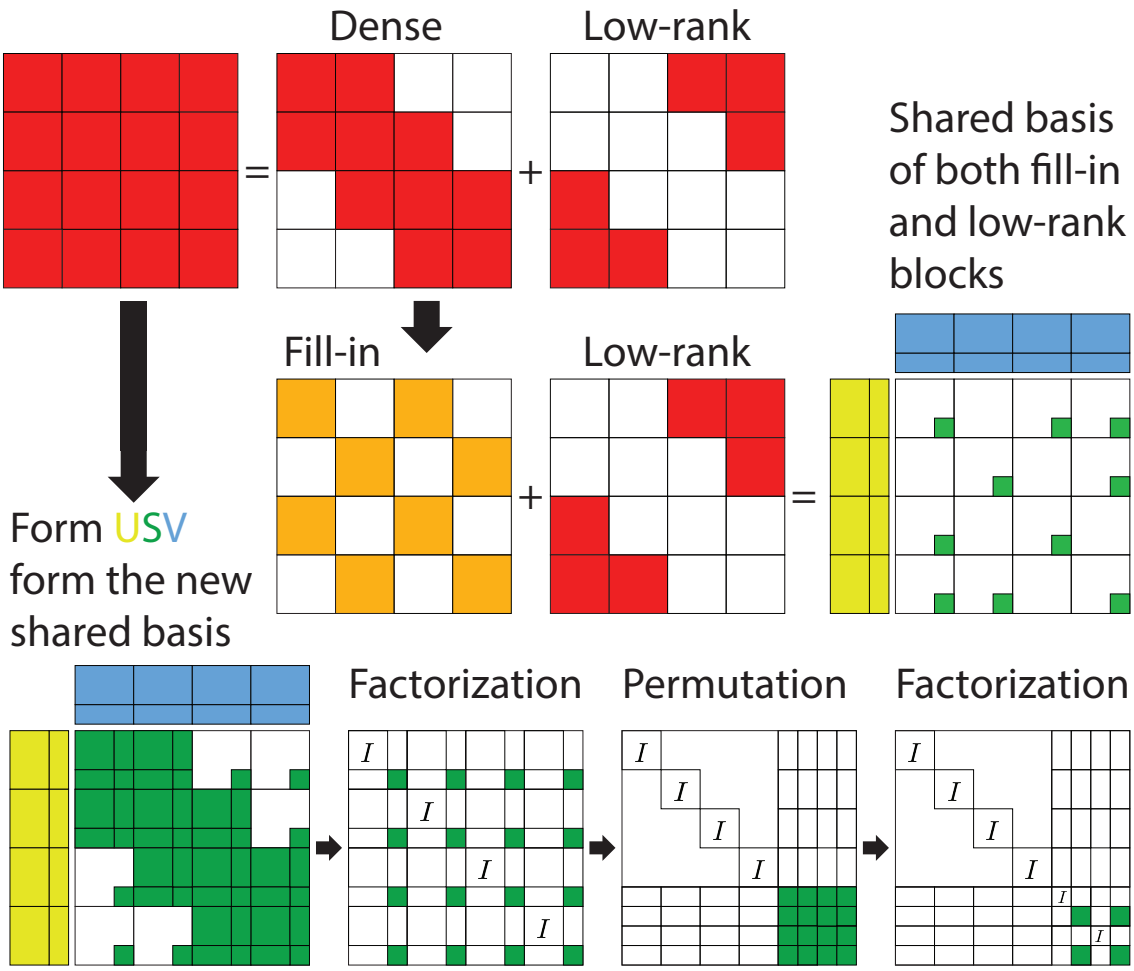


Figure 4.2: Flow of the $\mathcal{H}^2$-ULV factorization without dependencies.

on coloring schemes to extract the parallelism from an inherently serial algorithm. In the present work, we develop a method that has the best of both worlds, where we use $\mathcal{H}^2$-matrices in order to achieve linear complexity for 3-D problems, while also removing the dependencies on the trailing sub-matrices during the factorization.

4.1.1 Fill-in blocks are low-rank

The nullity theorem [52] states that the LU factors of a matrix have the same low-rank structure as the matrix before it is factorized. This is only possible if all the fill-ins can be compressed back to low-rank blocks. Existing $\mathcal{O}(N)$ dense direct solvers such as IFMM [4] and HIF [30] are also based on this property of the fill-ins being low-rank. In the present work, we also exploit this property of structured low-rank matrices to recompress the fill-in blocks to low-rank. Furthermore, under the USV formulation, compressing a block to low-rank is equivalent to eliminating that block for that level, as we have shown in Section 2.2. Therefore, none of the blocks remain filled-in during

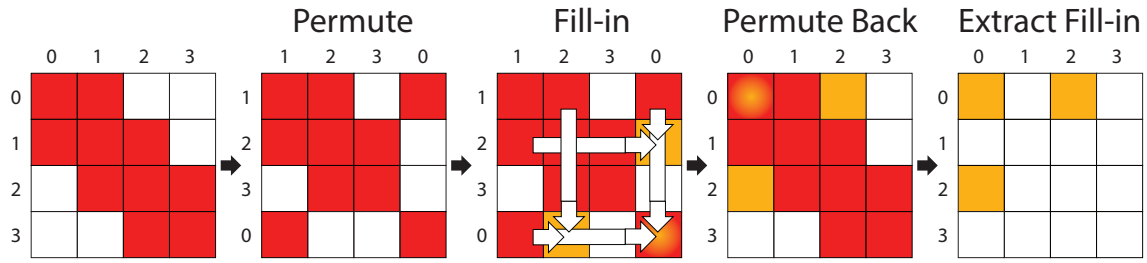


Figure 4.3: Fill-in of the first row and column during the pre-factorization.

the $\mathcal{H}^2$-ULV factorization. They may temporarily fill-in, but are immediately eliminated through recompression to low-rank blocks. However, it can be seen by comparing Figs. 4.1 and 3.4 that the $\mathcal{H}^2$-ULV needs to sequentially factorize the S matrix, whereas the HSS-ULV can factorize each level in parallel. This is due to the fact that the fill-in blocks still need to be recompressed to low-rank blocks, and the recompression requires an update to the shared basis. Since the next block row/column cannot proceed until the shared basis is updated, this is what actually causes the serialization. Therefore, this update to the shared basis is what prevents us from removing the dependencies on the trailing sub-matrices. If we can somehow form a shared basis that contains all the fill-in blocks (and not just the low-rank blocks), there would be no need to update this shared basis during the factorization. This would lead to an $\mathcal{H}^2$-ULV factorization without trailing sub-matrix dependencies. Unlike the fill-reducing ordering in sparse direct solvers, the objective here is not to reduce the fill-ins. It is rather to avoid the need to update the shared basis, even when some of the blocks temporarily fill-in and need to be recompressed to low-rank so that they do not remain filled-in.

4.1.2 Pre-computing the fill-ins per block row/column

Our method consists of two separate factorization phases – one for dense blocks in the original matrix A to compute the fill-ins, and another for the S matrix after the shared basis has been constructed. Both phases can be done without dependency on trailing sub-matrices. The flow of computation is shown in Fig. 4.2. We first decompose the matrix into the dense blocks and low-rank blocks. Then, the fill-ins that occur during the factorization of the dense blocks is computed using the method shown in Fig. 4.3. In order to compute all potential fill-in blocks in a given block row/column, we permute that block row to the bottom and block column to the far right. We then compute all possible fill-ins from all other rows/columns into that block row/column. In this particular case, the $A_{2;0,2}$, $A_{2;2,0}$, and $A_{2;0,0}$ blocks will fill-in. We do not accumulate the fill-in blocks into the dense blocks during this process, but store them separately. We then permute the block row/column back to their original position along with the shared basis. Actually, the permutation is done only for the sake of explaining the fill-in process in an intuitive way, and this process can be implemented

without the permutation if one can identify which blocks will fill-in without permuting them. This process can be executed in parallel for all block rows/columns, since they do not depend on each other. Note that the factorization of the diagonal blocks and the triangular solves of the dense off-diagonal blocks are not redundantly computed for the same block more than once during this process. The resulting fill-in matrix is shown at the center of Fig. 4.2.

4.1.3 A shared basis for both fill-in and low-rank blocks

The next step is to form a shared basis between the fill-in and low-rank blocks for each block row/column. This shared basis will contain all the information necessary to compress the low-rank blocks initially, and any of the fill-in blocks that arise during the factorization. All fill-in and low-rank matrices for a given row/column are concatenated to form the shared row/column bases

$$[U^S_{2;i}\ U^R_{2;i}], R = \text{QR}([F_{2;i,+}\ A_{2;i,+}]) \tag{4.1}$$

$$[V^{S^\top}_{2;j}\ V^{R^\top}_{2;j}], R = \text{QR}([F_{2;+,j}{}^\top\ A_{2;+,j}{}^\top]), \tag{4.2}$$

where $F_{2;i,+}$ and $A_{2;i,+}$ are the concatenation of all fill-in blocks and low-rank blocks in the ith row, respectively. Similarly, $F_{2;+,j}$ and $A_{2;+,j}$ are the concatenation of all fill-in blocks and low-rank blocks in the jth column, respectively. The center row in Fig. 4.2 corresponds to the operation in Eqs. (4.1) and (4.2). The procedure is identical to the construction of the shared basis for the low-rank blocks besides the inclusion of the fill-in blocks. These bases are then used to compute the S matrices for the dense blocks using Eq. (3.8) and the low-rank blocks using Eq. (3.9). Once the USV decomposition is formed for the shared basis that incorporates both the fill-in and low-rank blocks, the factorization of the $\mathcal{H}^2$-ULV can proceed just as the HSS-ULV, as shown in Fig. 4.2. The same procedure shown in Eqs. (3.11) to (3.15) can be applied here as well. After the leaf level has been processed, the matrix can be permuted to cluster the remaining skeleton parts, which can be solved recursively by applying the same procedure at each level.

In summary, the key ideas that make it possible to remove the trailing sub-matrix dependency even for strong admissibility are:

- Fill-in blocks are always low-rank. This has been demonstrated in previous studies [4, 30], and is not a unique claim of this paper.
- The fill-in blocks can be pre-computed and shared bases can be formed to incorporate both the fill-in and low-rank blocks. To the extent of our knowledge, this has not been done before.
- With this new shared basis that incorporates the fill-in blocks, there is no need to update the shared basis during the factorization of $\mathcal{H}^2$-matrices, which allows us to remove the trailing sub-matrix dependency.

4.1.4 Parallel implementation on distributed memory

Previous attempts to parallelize $\mathcal{H}^2$-ULV required coloring [48] or the use of task-based runtime systems [16] in order to extract parallelism from an inherently serial algorithm. The algorithm is inherently serial because there are trailing sub-matrix dependencies, where the blocks in the lower-right need to wait for the blocks in the upper-left to finish computing the GETRF (LU factorization), TRSM (traiangular solve), and GEMM (matrix multiplication for the Schur complements). Our proposed method removes this dependency by pre-computing a shared basis that does not need to be updated during the factorization. The only dependency that remains is the one between the levels. (By "level", we mean the level of granularity in the hierarchically sub-divided matrix structure.) We can somewhat relax this level-wise dependency as well, by computing the upper levels redundantly on multiple processes.

The resulting partitioning scheme is shown in Fig. 4.4, where the different colors represent the different partitions. In this example, each partition has only one block row/column, but there could be many block rows/columns in each partition. The cross shape of the partitions comes from partitioning the underlying geometry, where a certain block row and block column correspond to a sub-domain in the geometry. At the upper levels in the matrix the colors are mixed, which represents the redundant storage and computation of these blocks. This redundancy is the key to achieving good scalability at the upper levels, because it utilizes the processes that will be idle otherwise, while eliminating the need to communicate the results to each other.

Since it is always possible to split the range of processes in half (for odd numbers roughly half), the process tree shown in Fig. 4.4 is always a full binary tree, regardless of the underlying geometry or the type of matrix. The rows and columns of the $\mathcal{H}^2$-matrix also form a full binary tree, which is usually deeper than the process tree. This means that the lower levels of the row/column tree are grafted to the leaves of the process tree as shown in Fig. 4.4. When the factorization reaches the level where more than one process owns the block, the two child blocks exchange their information through an the `Allgather` collective with a split communicator, which exchanges information between the pair of child blocks that are being merged. At even higher levels in the process tree we can also exchange the necessary information between more processes through an `Allgather` collective with a communicator that is split accordingly. Note that for these `Allgather` collectives with split communicators, each process only communicates with one other process at any given level. We are essentially performing a tree `Allgather` through a hierarchy of communicators that are split according to the process tree. However, the size of the blocks that are gathered become smaller as the level increases, since 3/4 of the blocks get eliminated at each level for the upper levels of the $\mathcal{H}^2$-matrix.

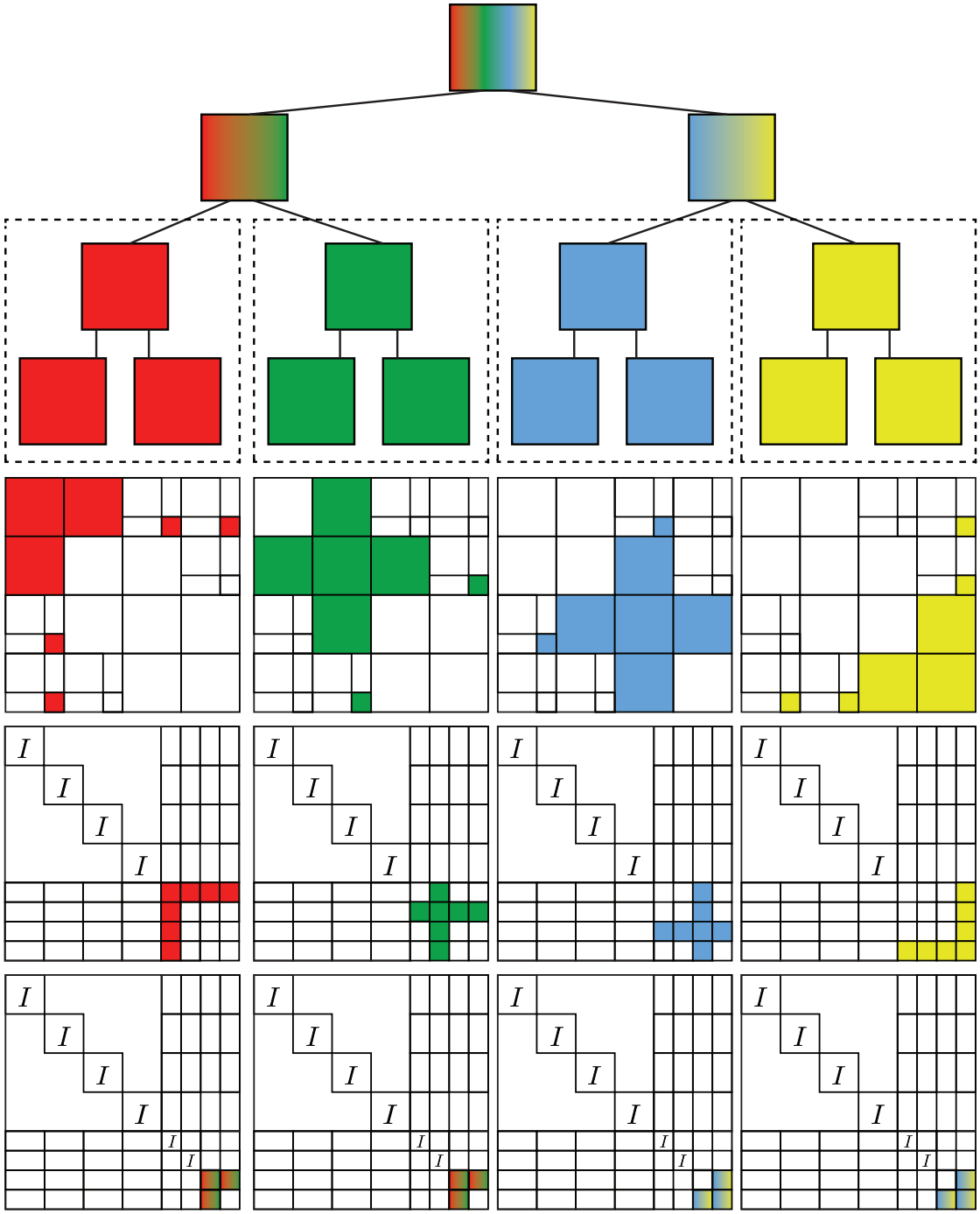


Figure 4.4: Partitioning of the $\mathcal{H}^2$-matrix. Upper levels are computed redundantly by multiple processes.

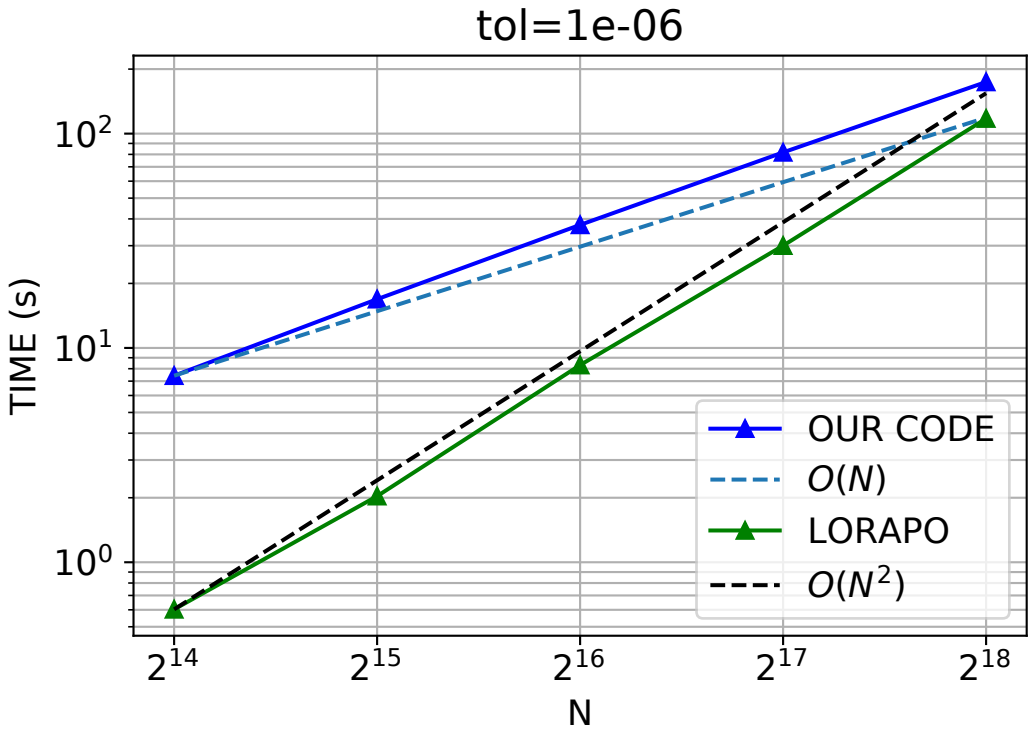


Figure 4.5: Comparison of LORAPO vs. our code on a single core for different problem sizes and relative error of 10^{-6}.

4.2 Tests on a simple geometry with Laplace potential

We first test our algorithm on a simple geometry with particles uniformly distributed inside a 3-D unit cube. We assume unit charges on each particle. We use the Green's function solution of the Laplace equation as our kernel.

$$\Phi_i = \frac{q_j}{4\pi r_{ij}}, \tag{4.3}$$

where r_{ij} is the Euclidean distance between the points $\mathbf{x}_i$ and $\mathbf{x}_j$, and q_j is the charge. The Green's function matrix $G_{ij} = \frac{1}{4\pi r_{ij}}$ results in a dense but rank-structured matrix.

4.2.1 Experimental setup

We perform tests on a single node with 2xAMD EPYC 7742 CPUs, each with 64 physical cores running at 3.3 GHz. The total available memory of the node is 1000GB, evenly split between both CPUs. We use GCC 9.3.0 as our compiler and openMPI 4.0.3 for the MPI processes, and link to Intel MKL 2020.1.

We compare our implementation with LORAPO [45], an adaptive-rank BLR Cholesky factorization using the PaRSEC PTG [17] runtime system for achieving asynchronous parallelism. Although LORAPO uses mixed precision for the low rank parts [1], we switch it off for these tests and use only

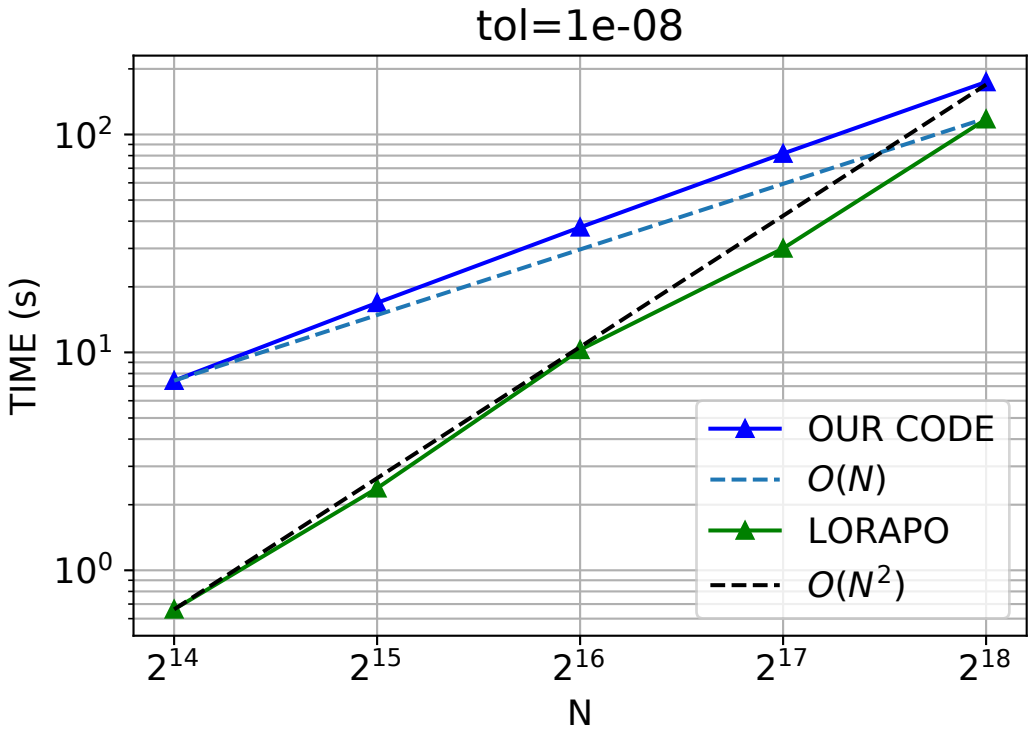


Figure 4.6: Comparison of LORAPO vs. our code on a single core for different problem sizes and relative error of 10^{-8}.

double precision for all our experiments. We found the most optimal way to run our code was by attaching a single process to each physical core, whereas LORAPO is run by spawning one process per node and attaching threads to each physical core. We report our strong scaling experiments by only reporting the number of cores being utilized. In all of our results, the relative L2 error is calculated by comparing the accuracy of the solution obtained using our method to the one obtained using a dense LU factorization from LAPACK.

Fig. 4.5 and fig. 4.6 show time taken for factorization for our code vs. LORAPO on a single core with varying problem sizes. Our algorithm is able to scale linearly (ideal $O(N)$ scaling is shown by the blue dotted line) for all problem sizes. It can be seen that although our code scales almost linearly, LORAPO has a better time to solution for most problem cases in spite of having a time complexity of $O(N^2)$, which is denoted by the black dotted line.

Proof that the faster performance of LORAPO is purely due to the algorithmic computations and not due to some anomaly in our code can be seen in Fig. 4.7, where we compare the number of floating point operations performed for the same single core experiment shown in Fig. 4.6. ULV based factorization requires more flops compared to a normal factorization. Applying the U and V basis to the dense blocks of the matrix has a cost similar to that of factorizing these dense blocks, which results in a large extra cost for ULV. In addition, sharing and nesting bases generally results

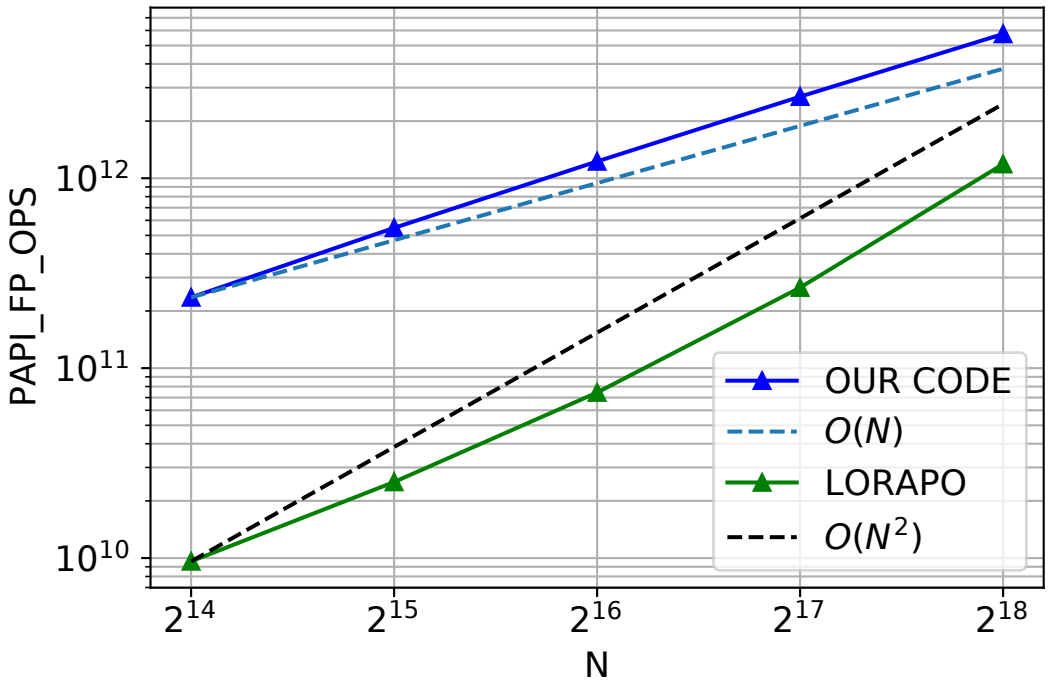


Figure 4.7: Comparison of PAPI_FP_OPS between our code and LORAPO for an accuracy of 10^{-8} and the same tile sizes as used in Fig. 4.6

in a larger rank than what the individual low-rank blocks have. BLR takes advantage of being able to independently compress each low-rank block, so that their rank can be minimized to save flops. In upper levels of the $\mathcal{H}^2$-matrix, we have reported seeing a rank that is as high as 180, whereas BLR uses a maximum of rank 50 at the leaf. A combination of these factors is what leads to the greater number of flops for our method under serial and small problem size settings.

4.2.2 Strong scalability of shared memory parallelism

The above figures show the strong scaling of our code vs. LORAPO for a constant problem size of $N = 131072$ in Fig. 4.8 and for $N = 262144$ in Fig. 4.9. The accuracy for all experiments is constant at 10^{-8}. The advantage of the inherent parallelism of our algorithm described in the above sections can be seen here. Our algorithm is able to beat LORAPO when using a large number of cores, even though LORAPO uses a runtime system for asynchronous parallelism, and uses fewer floating-point operations.

The reason behind poor scaling of LORAPO in Fig. 4.8 can be seen from the trace of the computation taken for 64 cores on a problem size of $N = 131072$ with a leaf size of 1024. The red tasks in the trace are overhead introduced by the run time system PaRSEC, and the green tasks are actual useful computation, i.e. the time actually spent in executing the tasks. The chief reason

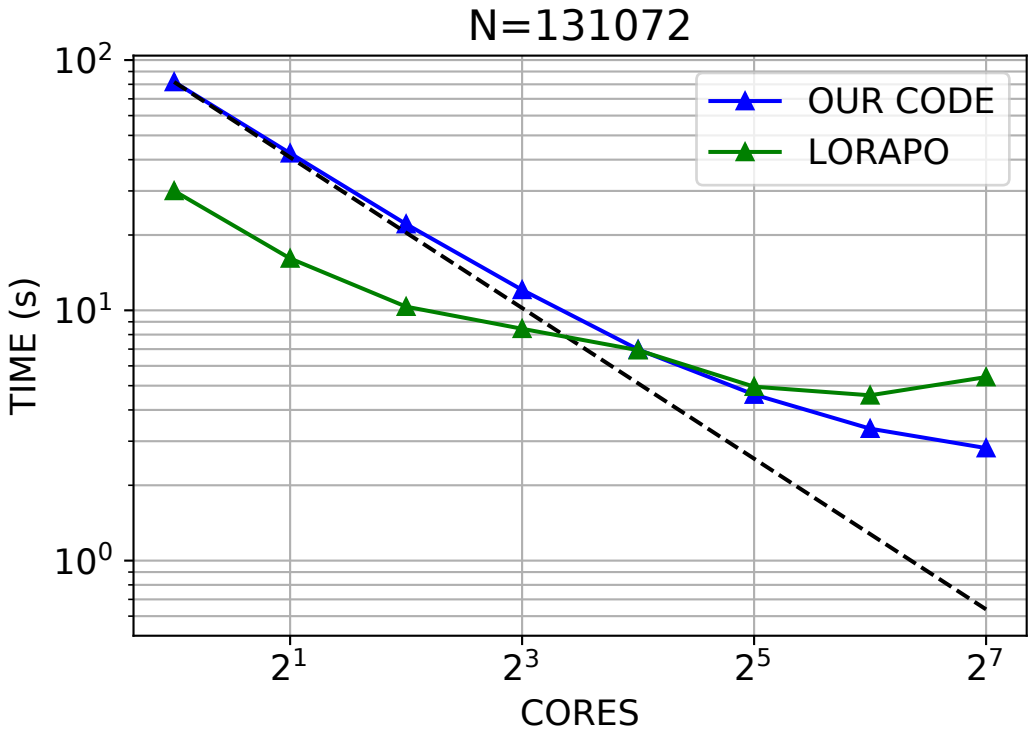


Figure 4.8: Strong scaling experiments for N=131072 on a single node utilizing upto 128 cores. The dotted black line shows perfect scaling.

behind the poor strong scaling is that the sizes of the tasks are too tiny to overcome the overhead of PaRSEC. The sizes of the red tasks are almost similar to the sizes of the useful computation. As a result of the dependencies introduced by the Cholesky factorization algorithm, new tasks do not become available fast enough, thus leading to poor performance.

4.2.3 Variation of the leaf size

The leaf size for LORAPO is the size of each block in the block low rank matrix. For our code it is the number of particles present in the leaf node of the tree. Varying these parameters, while keepin the number of cores constant at 32 and the problem size constant at $N = 131072$ leads to changes in the time to solution as shown in Fig. 4.10. It can be seen that LORAPO reaches an optimal execution time as the leaf size is increased until 2048, whereas our code is most optimal when the leaf size is 256.

Our algorithm uses a tree structure for representing the $\mathcal{H}^2$-matrix, and increasing the leaf size decreases the height of the tree, which increases the amount of computation that has to be performed for the factorization. The increased leaf size also increases the amount of work performed per process and reduces the available parallelism, thus leading to an increase in run time as the leaf size is increased. The impact of leaf size is exactly the reverse for LORAPO due to considerations

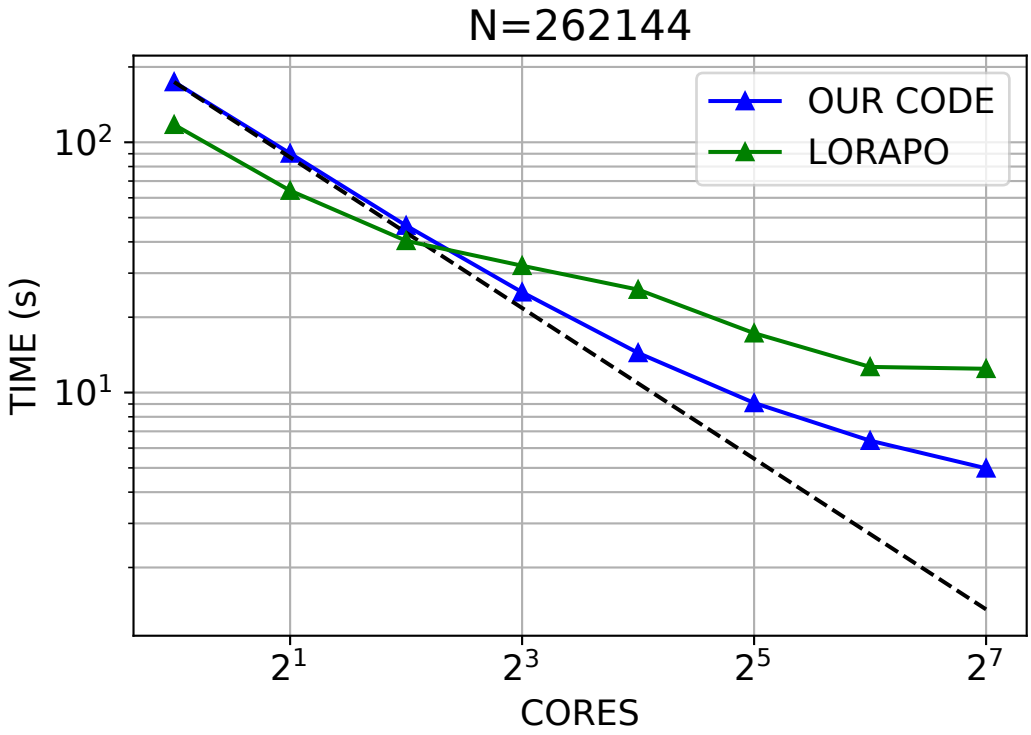


Figure 4.9: Strong scaling experiments for N=262144 on a single node utilizing upto 128 cores. The dotted black line shows perfect scaling.

of optimal computation for the block low rank matrix structure, adaptive rank capability of the factorization, run-time system overhead and available parallelism. The run time increases slightly as LORAPO approaches tile size 4096 due to reduction of available parallelism.

4.3 Tests on complex geometry with Yukawa potential

We now conduct experiments using an actual boundary element application in implicit solvent bio-molecular electrostatics. This is different from classical molecular dynamics where the water molecules are computed explicitly, adding millions more degrees of freedom to the simulation. In implicit solvent methods, the molecules and the solvent are treated as continuous dielectric media with different dielectric constants. This jump in the dielectric constant across the surface of the molecule is accounted for by placing a boundary element mesh over the surface. An example of the boundary element mesh we use in the current computations is shown in Fig. 4.12. For even larger problem sizes, we use a crowded environment of many hemoglobin as shown in Fig. 4.13. We use a collocation boundary element method, which essentially turns this mesh into a cloud of points. We use a 3-D k-means clustering to partition those cloud of points to form the leaf blocks of the $\mathcal{H}^2$-matrix. The flexibility of k-means clustering allows us to enforce the number of clusters

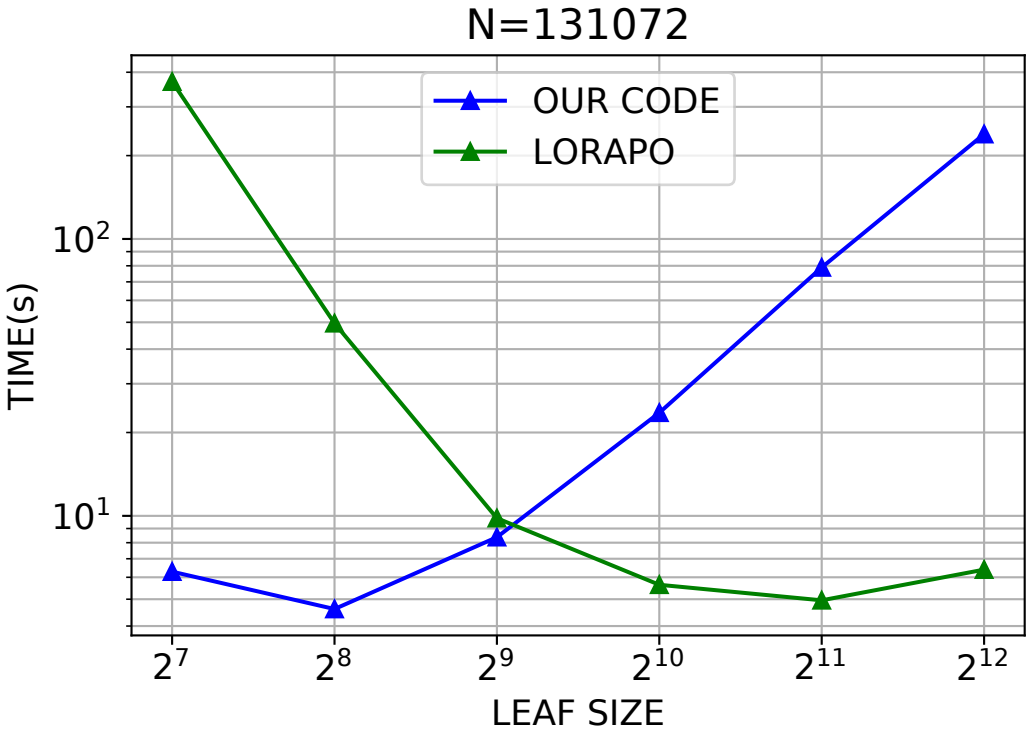


Figure 4.10: Impact of change the leaf size for LORAPO and our code for the same problem size ($N = 131072$ and available resources (32 cores).

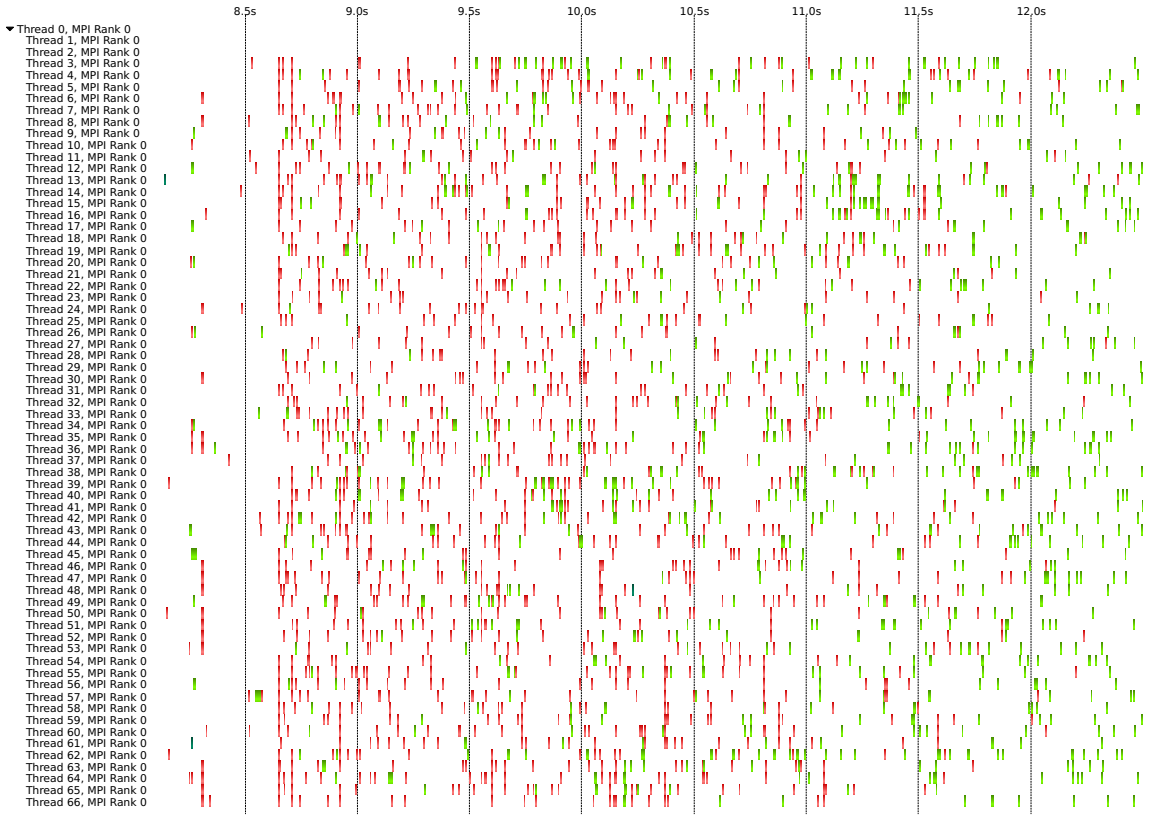


Figure 4.11: Trace visualization of LORAPO for a problem size of 131072 using 64 physical cores. The first two threads are reserved by PaRSEC for monitoring, task submission and communication purposes leading to a total of 66 threads showing in the trace. The red tasks are run time system overhead and the green tasks are useful computation.

to always be a power of two. We found that this works much better than space-filling curves for partitioning points on the surface of a complex geometry.

The potential we use here is the Yukawa potential, also known as the screened Coulomb potential, which takes the following form.

$$\Phi_i = \frac{q_i q_j}{4\pi\varepsilon_0 r_{ij}} \exp(-\alpha m r_{ij}), \tag{4.4}$$

where r_{ij} is the Euclidean distance between the points $\mathbf{x}_i$ and $\mathbf{x}_j$, and q_i, q_j are the charges at those points, α is a scaling constant, m is the mass of the particle, and ε_0 is the permittivity.

4.3.1 Experimental setup

The distributed memory tests are performed on the ABCI machine at AIST, with a total of 1088 compute nodes. Each node is configured with 2xIntel Xeon Gold 6148 CPUs, each with 20 physical cores running at 2.2 GHz each. The total available memory per node is 384GB, which is evenly split between the two CPUs. We use GCC 11.2.0 compiler with Intel MPI 2021.5 and link to Intel MKL 2022.0.

4.3.2 Strong scalability of the distributed memory parallelism

The results of our strong scalability tests on up to 10,240 cores are shown in Fig. 4.14. The coverage of data points is limited for various reasons, where our code ran out of memory for some cases, while LORAPO crashed for others. The $N = 119,264$ case has a size similar to the previous experiments on the simple geometry, and on 160 cores our code is already faster. This agrees with our results in the previous section. When the problem size is increased to $N = 954,112$, the difference between our code and LORAPO becomes much larger. This is due to the difference between the linear complexity of our code vs. the quadratic complexity of LORAPO. N is increased roughly an order of magnitude between these two experiments, so the factorization time of our code increases and order of magnitude, while the time of LORAPO increases two orders of magnitude. One can see that our code scales better than LORAPO as well, where the multi-node scalability of LORAPO is rather poor, while our code continues to scale well up to 10,240 cores. If we compare the factorization time for the $N = 954,112$ case on 10,240 cores between LORAPO and our code, we see that our code is approximately 4,700 times faster.

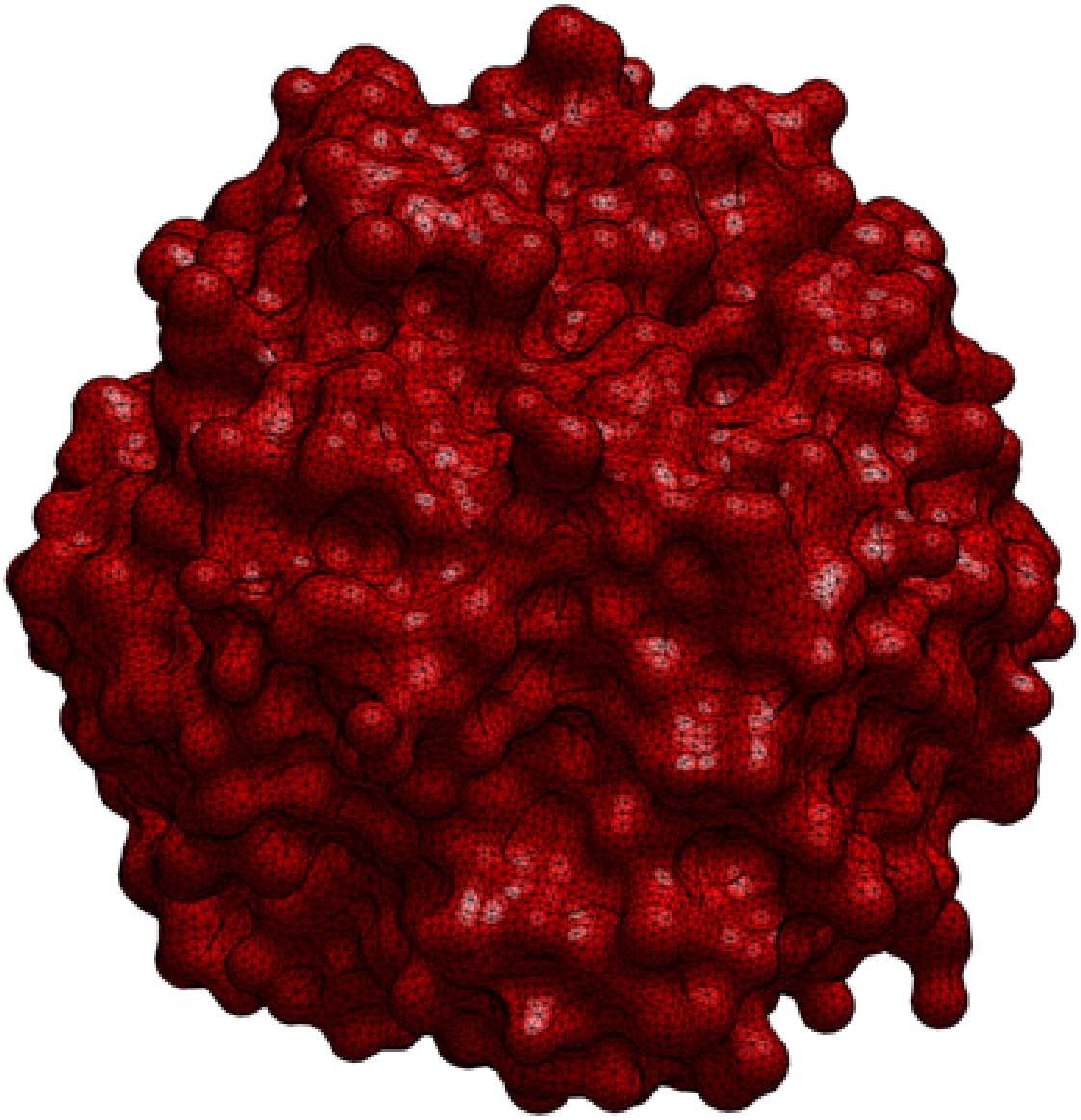


Figure 4.12: Boundary element mesh on a single hemoglobin.

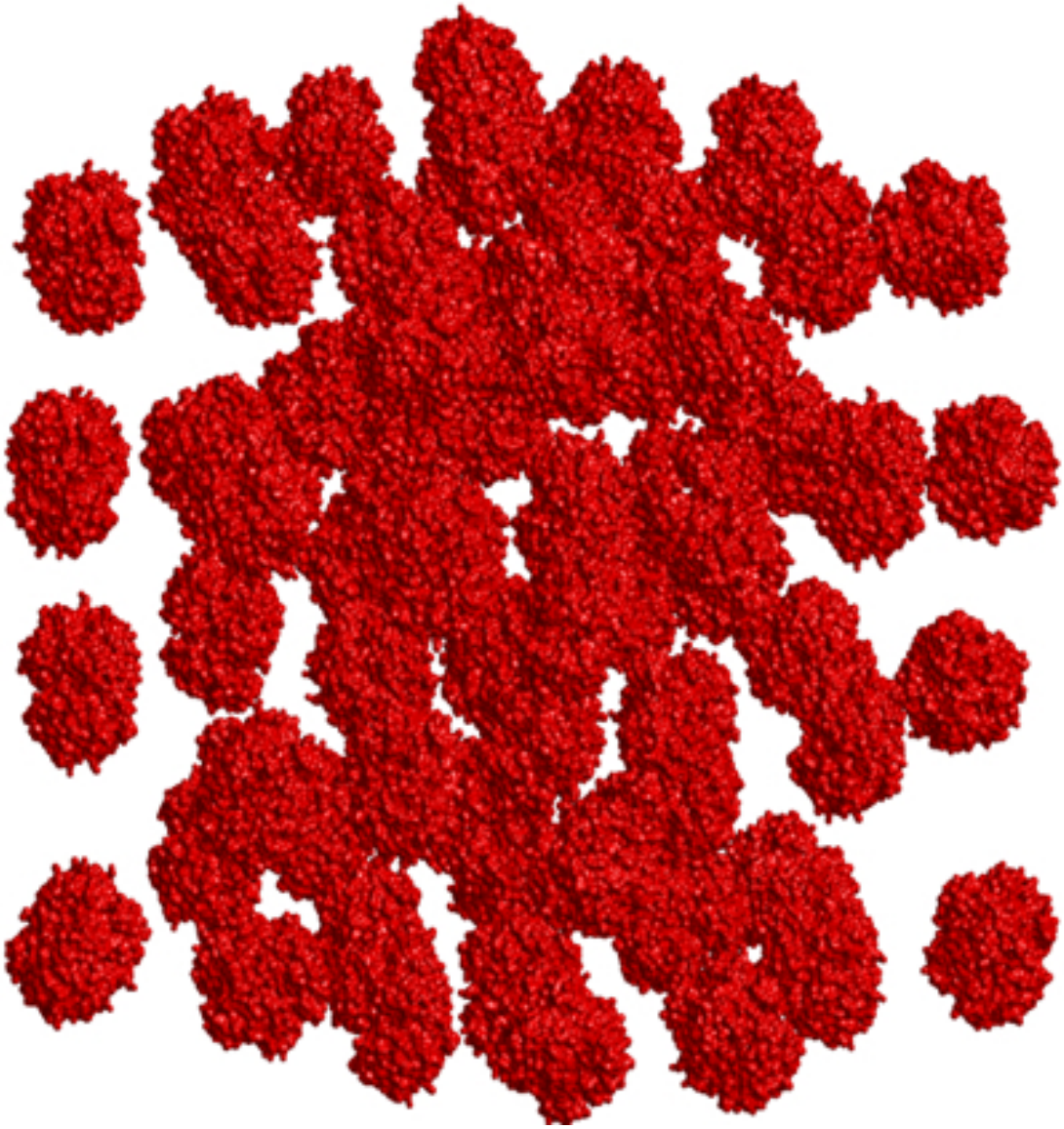


Figure 4.13: A crowded environment of 64 hemoglobin.

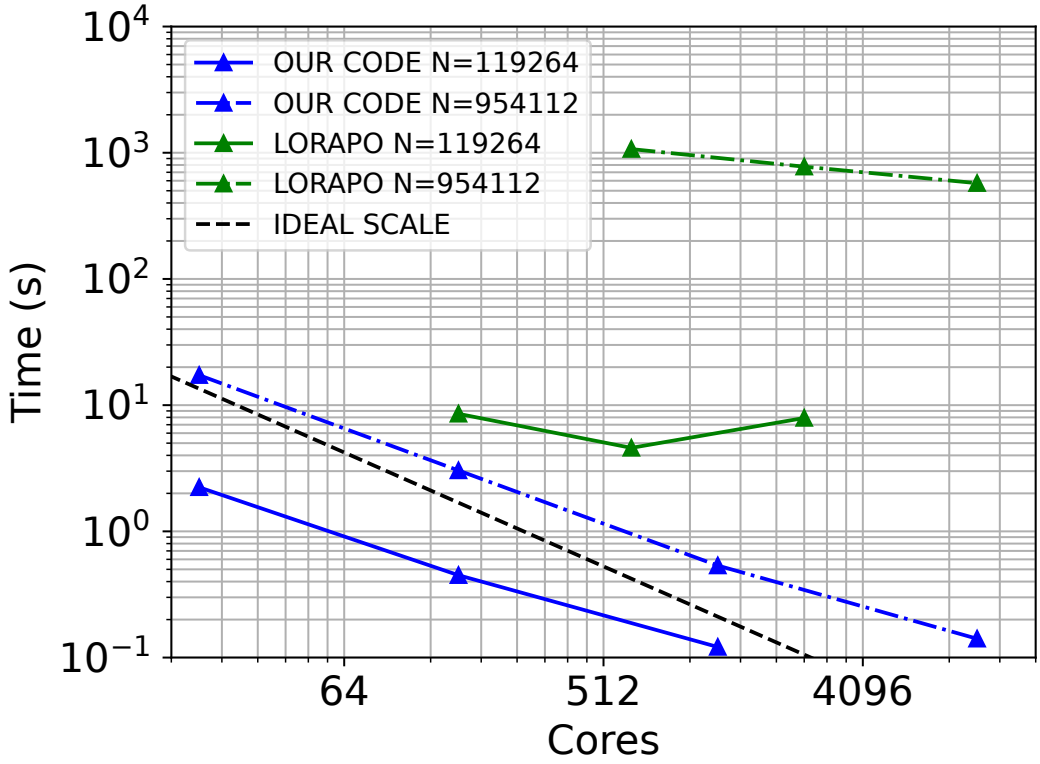


Figure 4.14: Strong scaling experiments on multiple nodes for different problem sizes of the hemoglobin boundary element problem using our code and LORAPO.

Chapter 5

Data-parallel ULV factorization for $\mathcal{H}^2$-matrix on many GPUs

Green's function matrices arising from boundary integral equations, covariance matrices in statistics, and kernel matrices in machine learning are large dense matrices, but their sub-blocks exhibit a low-rank structure. This structure can be exploited to reduce the complexity of multiplication and factorization of these matrices from $\mathcal{O}(N^3)$ to $\mathcal{O}(N)$. The accuracy can be controlled through a truncation threshold when compressing the low-rank blocks. Unlike sparse approximation of matrices, increasing the accuracy does not change the complexity, since the rank can become large but does not increase with N. Various structured low-rank matrix formats have been proposed such as the BLR matrix format from [5], the BLR^2 matrix from [7], the HODLR format from [3], the HSS format from [20], and lastly $\mathcal{H}$-matrix and $\mathcal{H}^2$-matrices in [26] and [28] correspondingly. BLR and BLR^2 are non-hierarchical structures, where the off-diagonal blocks of a block dense matrix are compressed to low-rank blocks. BLR^2 shares the low-rank basis between the blocks in each row and column, while BLR does not. HODLR and HSS are hierarchical structures, where the diagonal blocks are recursively subdivided. HSS shares the bases of the low-rank blocks among the rows and columns, while HODLR does not. $\mathcal{H}$ and $\mathcal{H}^2$ matrices are also hierarchical but subdivide not only the diagonal blocks but also the off-diagonal blocks. How far the blocks are subdivided is determined by the admissibility condition, which is based on a distance metric of the underlying geometry of the problem that forms the dense matrix. $\mathcal{H}^2$-matrices share the bases while $\mathcal{H}$-matrices do not. A subset of these structured low-rank matrix formats has been applied to matrix-vector and matrix-matrix products from [11], Cholesky and LU from [27], QR factorizations in [34], and eigenvalue decomposition [9] and [33]. Libraries that implement such algorithms can be used instead of BLAS and LAPACK when the matrix is large and has a low-rank structure.

The recent trend in hardware architecture is driven by machine learning, where low-precision matrix engines are becoming increasingly popular. Sparse matrices have difficulty extracting the full potential of such hardware since these matrix engines are designed for dense matrices. Unlike

sparse matrices, structured low-rank matrices consist of many small dense matrices, which can extract the potential of matrix engines through batched dense matrix operations. In addition, when using low-precision arithmetic, it does not make sense to perform the matrix operations exactly. Structured low-rank matrices can adjust their accuracy to match the precision of the matrix engines, and can reduce the complexity of matrix multiplication and factorization to $\mathcal{O}(N)$ while doing so. Therefore, we anticipate that there will be many cases where structured low-rank matrices can be used instead of BLAS/cuBLAS and LAPACK/cuSOLVER on future processor architectures with low-precision matrix engines. However, structured low-rank matrices are mainly studied by applied mathematicians and many implementations are written in Matlab. Even the few implementations on GPUs are for simple variants of structured low-rank matrices with suboptimal computational complexity and parallelism. It is true that structured low-rank matrices have a highly complex data structure and flow of computation, which makes them difficult to implement on GPUs. In the present work, we adopt the ULV factorization introduced from [20] to remove the dependency on trailing sub-matrices. This allows us to perform the factorization of an originally dense matrix in a highly parallel fashion, which allows us to extract the true potential of GPUs. [41] extended the original ULV factorization for HSS matrices to $\mathcal{H}^2$-matrices, while preserving the highly parallel nature of the algorithm. However, their implementation only used CPUs, and the inherent parallelism of the ULV factorization has never been used to develop a highly parallel GPU implementation. Furthermore, Ma *et al.* only focus on the factorization, and the forward and backward substitution still have limited parallelism even when using their method. In the present work, we address these shortcomings of existing work and develop a highly scalable dense LU factorization method on GPUs. We also develop a novel algorithm that allows the forward and backward substitution to be performed in a highly parallel manner. Our main contributions can be summarized as follows.

- We develop for the first time a highly parallel GPU implementation of a dense LU factorization with linear complexity based on the $\mathcal{H}^2$-ULV factorization.
- We propose the very first inherently parallel forward and backward substitution algorithm for structured low-rank matrices.
- We provide an efficient implementation of our proposed algorithms of factorization and substitution, that extracts the full potential of large GPU-accelerated systems.

We organize the rest of the paper as follows. Section "Background" contains the related work on other structured low-rank approximated matrix formats and existing attempts to make them run on parallel machines. In Section "Method", we describe our method as an augmentation to the existing ULV factorization for $\mathcal{H}^2$-matrices. We cover aspects including the mathematical justification of pre-compressing Schur complements, and the modifications done to the construction, factorization, and substitution to make all of these phases inherently parallel. Section "Method" concludes with a

complexity analysis of our proposed method. Section "Design Considerations for GPUs" describes the implementation and optimization details, along with the reasons why we have chosen to use specific libraries for GPUs and distributed memory implementation. We also provide details of our implementation and experiments for the sake of reproducibility. Section "Numerical Results" contains all the experiments we performed, including time complexity, FLOPS performance, relation between the rank and solution accuracy, strong scaling, and weak scaling results. Finally, we add closing remarks in the Section "Conclusion".

5.1 Background

Our method belongs to a larger family of direct solvers for dense matrices that arise from kernel matrices generated in complex 3-D geometry problems. In this section, we compare our approach to other groups' notable work, that falls into the same or similar categories. We organize this section by the internal format of the structured low-rank approximation being used by different software packages and review them based on the resulting algorithmic complexity and the reported parallel capabilities.

5.1.1 Block (recursive block) factorization on matrices with independently low-rank approximated blocks

Independently approximated low-rank blocks applied to the dense matrices are closely related to the block partitioned dense matrix. Therefore, many routines and methods designed for the dense matrices, can be directly used on the approximated formats with slight modifications for low-rank representations. The Block low-rank (BLR) matrix is the simplest of all structured low-rank approximated matrix formats and leads to the most simple implementations and parallelization. Although having a suboptimal $\mathcal{O}(N^2)$ factorization complexity and $\mathcal{O}(N^{1.5})$ in memory consumption, it is still much better than the dense $\mathcal{O}(N^3)$ complexity. HiCMA in [2, 19] and LORAPO in [45, 18] provide a highly optimized implementation in the BLR format that delivers excellent performance on CPUs and distributed memory environment. Many of the BLR implementations including the ones mentioned above, use the 2-D block-cyclic process distribution used by ScaLAPACK: [10]. The benefits come from 1) the workload is still balanced from the cyclic distribution after a row and a column have been eliminated, and 2) both the communication volume and the neighbors in the process grid of each processor are $\sqrt{P}$, where P is the total number of processors.

The multi-level approach of independently approximated blocks is named HODLR for weak admissibility and $\mathcal{H}$-matrix for strong admissibility. Despite having an even further reduction in factorization and memory complexity of $\mathcal{O}(N \log N)$, the $\mathcal{H}$-matrix format possesses a very complicated recursive structure of factorization data-dependencies that is one of the most notorious problems to

parallelize. A novel method has been presented by [21] for HODLR matrices, that is capable of factorizing and applying forward/backward substitution in an entirely parallel fashion. Although their method is limited to the weak admissibility configuration, which shares a common weakness with the HSS matrices, they provide the first inherently parallel factorization to the families of the structured low-rank matrices with independently compressed blocks. More traditional approaches of recursive factorization algorithms for strongly admissible configurations are extremely difficult to parallelize, and the scalability of factorization using the $\mathcal{H}$-matrix format is quite poor in the existing literature. One of the most notable distributed memory implementations of the $\mathcal{H}$-matrix $\mathcal{H}$ACApK in [32] and [35] uses a modified structure named lattice $\mathcal{H}$-matrix, which embeds an $\mathcal{H}$-matrix structure within the dense blocks of a BLR matrix. The lattice $\mathcal{H}$-matrix combines the parallelism of BLR matrix with the efficiency of the $\mathcal{H}$-matrix. Although good speed up on factorization has been reported by comparing with the BLR matrix formats, the algorithm struggles to scale for a relatively small number of cores used ($< 10^3$) due to the highly complex data-dependency structures. Unlike the HSS-ULV method, this approach suffers from the data dependency of trailing submatrices.

Our method differs significantly from approaches covered in this section of matrices compressed with independent bases. Even the simple flat structure of the BLR matrix possesses the factorization data dependency from the top-left corner to the bottom-right corner, and the reduction in algorithmic intensity via low-rank approximation has only reduced the parallel regions and exposed the data dependency even more. The methods described in the following section exploit the nature of the shared basis in order to avoid the data dependencies in the block and recursive block factorization algorithms.

5.1.2 Structured low-rank matrices with shared basis and ULV Factorization

Hierarchically Semi-separable (HSS) matrices and the corresponding ULV factorization algorithms have first been introduced by [20] and had led to multiple parallel implementations (STRUMPACK: [46], H2pack: [31]) that produced results that scale on extremely parallel machines on very large matrices (degrees of $10^8 \times 10^8$). Among them, STRUMPACK is a library that provides excellent support for running on NVIDIA GPUs and distributed memory configuration, and has made several significant contributions. H2pack is implemented on CPU only but uses a highly advanced construction method of HiDR introduced in [15] that reports good accuracy and robustness for the low-rank approximation.

The ULV factorization method used on weak admissibility structures, such as BLR^2 and HSS, are inherently parallel by nature and have relatively simple structure. Before applying the internal LU or Cholesky factorization, ULV factorization methods first separate each block into the skeleton part and redundant part. When the shared row and column basis are factored out, this creates

a sparsified matrix similar to that of the sparse multi-frontal LU solvers with nested dissection ordering. Therefore, the data-dependency structure of the HSS-ULV factorization looks exactly like a multi-frontal LU solver with a constant dissector size (low-rank approximation rank), and has an entirely parallel region amongst each level with size $\mathcal{O}(2^l)$. The complexity of the method is also the same for a constant dissector size problem, which is strictly $\mathcal{O}(N)$. However, one weakness of the HSS-ULV factorization is its weak admissibility condition. When solving for kernel matrices in 3-D geometry, the off-diagonal low-rank blocks no longer compresses into a constant rank, but an $\mathcal{O}(N)$ rank that grows with the matrix dimension. The non-constant rank of the off-diagonal eventually makes the algorithm lose its linear complexity for the factorization and solution.

One of the notable attempts of running the ULV factorization for strongly admissible $\mathcal{H}^2$-matrices is made by [39]. They followed the same algorithms of the HSS-ULV factorization for the full basis transformations and block-LU factorization internally. With the addition of an innovative re-compression technique, that has been widely used in factorization for independent basis formats, they can eliminate the fill-ins and maintain the linear complexity even for 3-D geometry. The inverse FMM and LoRaSp from [4] and [23] reported the same capability of factorizing strongly admissible $\mathcal{H}^2$-matrices also under linear complexity. LoRaSp uses extended sparsification to exclude the low-rank blocks from the dense operations, as well as extending the factorization algorithm to an initially sparse matrix and uses low-rank approximation hierarchically to compress the fill-ins. Both ULV factorization and LoRaSp rely on the creation of an intermediate block sparse representation while recompressing the fill-ins to maintain the $\mathcal{O}(N)$ algorithmic complexity. However, even though the approaches by [39], [4], and [23] are able to extend the HSS-ULV factorization to 3-D problems while maintaining the linear complexity, they are not able to inherit the highly parallel nature of the HSS-ULV factorization because the recompression brings back the dependency for the trailing submatrices. Distributed memory parallelization attempts for the strongly admissible $\mathcal{H}^2$-matrices have been made by LoRaSp, but the reported parallel scalability is far from that of the HSS-matrices.

The work by [41] has been the first attempt to unite the advantages of parallel capabilities of the ULV factorization to the ideal linear complexity from the strong admissibility configurations of $\mathcal{H}^2$-matrices. The results presented include a highly parallel CPU implementation both in shared memory as well as distributed memory configuration, that shows good strong scaling up to $10,000$ CPU cores. Although their method can potentially be extended to GPUs, such an effort has been left for future work. Also left for future work was the forward and backward substitution phase, which was not implemented in parallel in their work. With traditional factorization methods for structured low-rank matrices, it has been difficult to extract the full potential of GPUs. This is because the arithmetic intensity of the low-rank blocks is quite low, and the dependency between the blocks prevents them from being processed in batches. The present work exploits the lack of dependency in the $\mathcal{H}^2$-ULV method by [41], and shows that a scalable batched GPU implementation is indeed

possible for this algorithm. Further, we supplement the approach taken by [41] with a more accurate and sound mathematical proof. We also extend their attempt of parallelization into an inherently parallel forward and backward substitution that accompanies the $\mathcal{H}^2$-ULV factorization. To the best of our knowledge, this is the first work that proposes an inherently parallel forward and backward substitution for dense matrices.

5.2 Method

The main difference of our work from the other $\mathcal{H}^2$-matrix factorization methods is the introduction of another set of bases, which we called the factorization basis. The factorization basis, in short, is a low-rank shared basis that approximates all of the Schur complements computation that will arise during the factorization, even if the Schur complement has been computed from the closely interacting matrix (in other words, dense matrix) blocks. The introduction of the factorization basis is to augment the ULV factorization method that is initially used for factorizing and inverting shared-basis structured low-rank matrices, such as BLR^2, HSS, and $\mathcal{H}^2$ matrices. Merging the factorization basis with the shared basis for approximating the low-rank matrix blocks removes the need to update the basis during the recompression of the fill-in blocks. This results in an inherently parallel LU factorization even for dense matrices arising from 3-D geometry.

We first clarify what we mean by "Compression of the Schur complement". where we are addressing the compression of a particular matrix block $C_{jk} = A_{ji}A_{ii}^{-1}A_{ik}$, which is, in fact, the difference matrix between the Schur complement and the original matrix block. The proper definition of Schur complement is given in the following, which is a full-rank matrix block located on the diagonal:

$$M/A_{jj} = A_{jj} - A_{ji}A_{ii}^{-1}A_{ij} \tag{5.1}$$

Compressing only the Schur complement update uses significantly lower numeric rank even for very closely interacting matrix blocks, including completely overlapping geometry such as for updating matrix locating on the diagonal. The low-rank approximation rank correlates very weakly with the size of the input matrix, behaving more similarly to the compression of the far-field interactions, instead of having a rank that grows with $\mathcal{O}(N)$ for 3-D problems. We provide a comparison of the rank and difference in the accuracy of our method versus the traditional means of compressing the closely interacting off-diagonal blocks that the HODLR and HSS matrices use, later in the results section.

5.2.1 Characteristics of the Factorization Basis

The primary functionality of introducing the factorization basis is to approximate the difference matrix in Schur complement with the original matrix, given as:

$$A_{ji}A_{ii}^{-1}A_{ik} \approx U_j \Sigma V_k^T, \forall i \neq j \;\; \& \;\; i \neq k \tag{5.2}$$

$\mathcal{H}^2$-matrix factorization methods proposed by [39], [4], and [23] also use a factorization basis to recompress the fill-in blocks. The significance of the method proposed by [41] is that the factorization basis is pre-computed and merged into the shared basis before the factorization. This way, the shared basis once constructed in the beginning is not modified later during the recompression of the blocks that fill-in (low-rank blocks that become dense) during the factorization. Even with trailing updates introduced, the approximation with the same basis still holds as the Woodbury matrix identity of matrix inversion suggests that

$$(A_{ii} - A_{il}A_{ll}^{-1}A_{li})^{-1} = A_{ii}^{-1} + A_{ii}^{-1}A_{il}...A_{li}A_{ii}^{-1} \tag{5.3}$$

Multiplying with A_{ji} and A_{ik} from the sides, Eq. (5.3) becomes

$$A_{ji}(A_{ii}^{-1} + A_{ii}^{-1}A_{il}...A_{li}A_{ii}^{-1})A_{ik} \approx U_j(\Sigma + \Sigma_1...\Sigma_2)V_k^T \tag{5.4}$$

suggesting that the Schur complement update produced from the updated diagonal is still compressible using the same basis.

The updates introduced to the off-diagonal blocks can also be compressed using the same logic

$$(A_{ji} - A_{jl}A_{ll}^{-1}A_{li})A_{ii}^{-1}A_{ik} \approx U_j(\Sigma - \Sigma_1...\Sigma_2)V_k^T \tag{5.5}$$

The purpose of constructing the factorization basis is to prevent the need to update the shared basis during the ULV factorization. Therefore, we want all possible fill-ins that might occur during the ULV factorization to occur during the construction of the factorization basis. Contrary to sparse matrix factorization where we would like to minimize the amount of fill-in, here we want to maximize the fill-in so that the factorization basis includes all possible fill-ins. Therefore, as we construct the factorization basis, we not only consider the updates from the ordinary elimination order from top-left to bottom-right, but also the updates that come from the latter block eliminations. This is also the key to making this phase inherently parallel. Because we are considering the elimination of arbitrary order, we are free to perform permutations in the factorization ordering during the computation of the factorization basis. This means that we can compute the fill-ins for each block row/column independently, so this operation is inherently parallel.

5.2.2 Trailing submatrix update for single-level

This subsection describes the trailing submatrix updates for a single-level factorization. For all structured low-rank matrices with shared bases (HSS, $\mathcal{H}^2$-matrices, etc.), the ULV-based factorization methods progress on a level-by-level order that allows us to deal with a simpler single-level variant of the problem before recursing on to the next.

We show how the factorization basis takes advantage of the low-rankness of the Schur complement updates during the factorization, to enable an inherently parallel factorization algorithm even when the matrix contains many dense off-diagonal blocks.

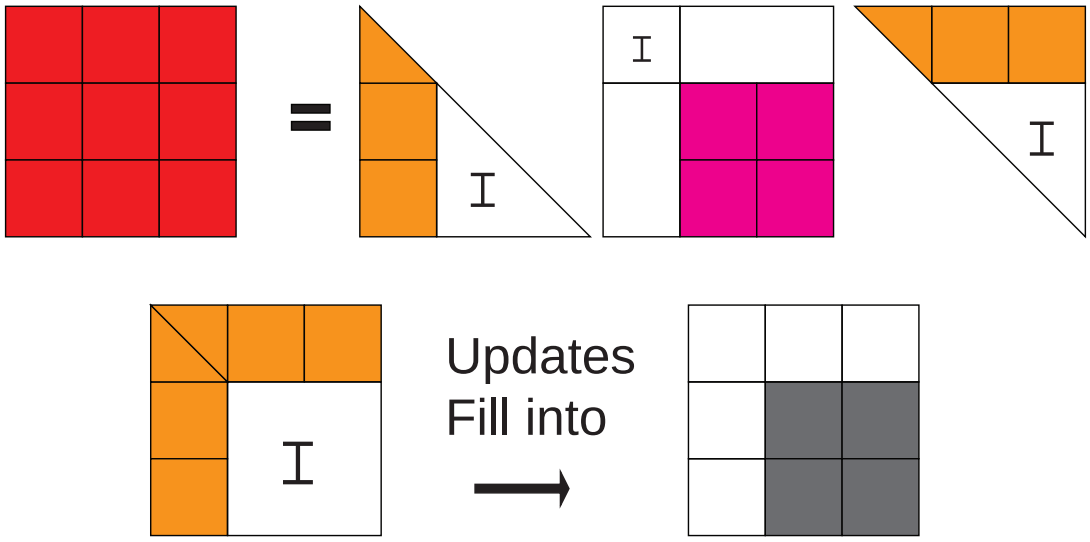


Figure 5.1: Step 1 of LU factorization on a 3×3 dense matrix with trailing Schur complement updates

By the term "skeleton", we address the matrix contents that correspond to the compressible or projected region to the basis as the U and V transformations are applied from the sides of the $\mathcal{H}^2$-matrix. In contrast, the regions that are dropped from the basis projection are named "redundant". During the ULV factorization of the $\mathcal{H}^2$-matrix, the factorization basis redirects trailing updates to only update the matrix blocks corresponding to the skeleton so that the redundant matrix blocks are never updated until they are eliminated from the factorization. This absence of trailing updates leads to an inherently parallel factorization. In the remainder of the paper, the dense blocks that correspond to the different regions of the basis, are named A^{SS}, A^{SR}, A^{SR}, and A^{RR}, to indicate the contents have been obtained from either the skeleton or the redundant regions of the left or right basis, as shown in Fig. 5.2. The concept of a factorization basis is independent of the traditional low-rank shared basis and can be used on matrix structures that do not have initial low-rank approximations, such as a fully block-dense matrix or a sparse matrix. [23] use this concept to form a sparse direct solver that recompresses the fill-ins using the factorization basis.

In figure 5.1, we show the first step of an LU factorization of a symmetric block-dense matrix, where the first block row/column has been factorized. For simplicity, we will use a 3×3 block dense matrix to introduce the basic concepts, before moving to the $\mathcal{H}^2$-matrix. In our method, the Schur complement update is presumed to be numerically low-rank, but Fig. 5.1 shows the updates to be dense. In this case, it is obvious that the factorization of trailing sub-matrices (shown in grey) cannot proceed until the Schur complement updates are computed.

We now introduce the concept of low-rank approximation, where the SVD of a submatrix A_{jk}

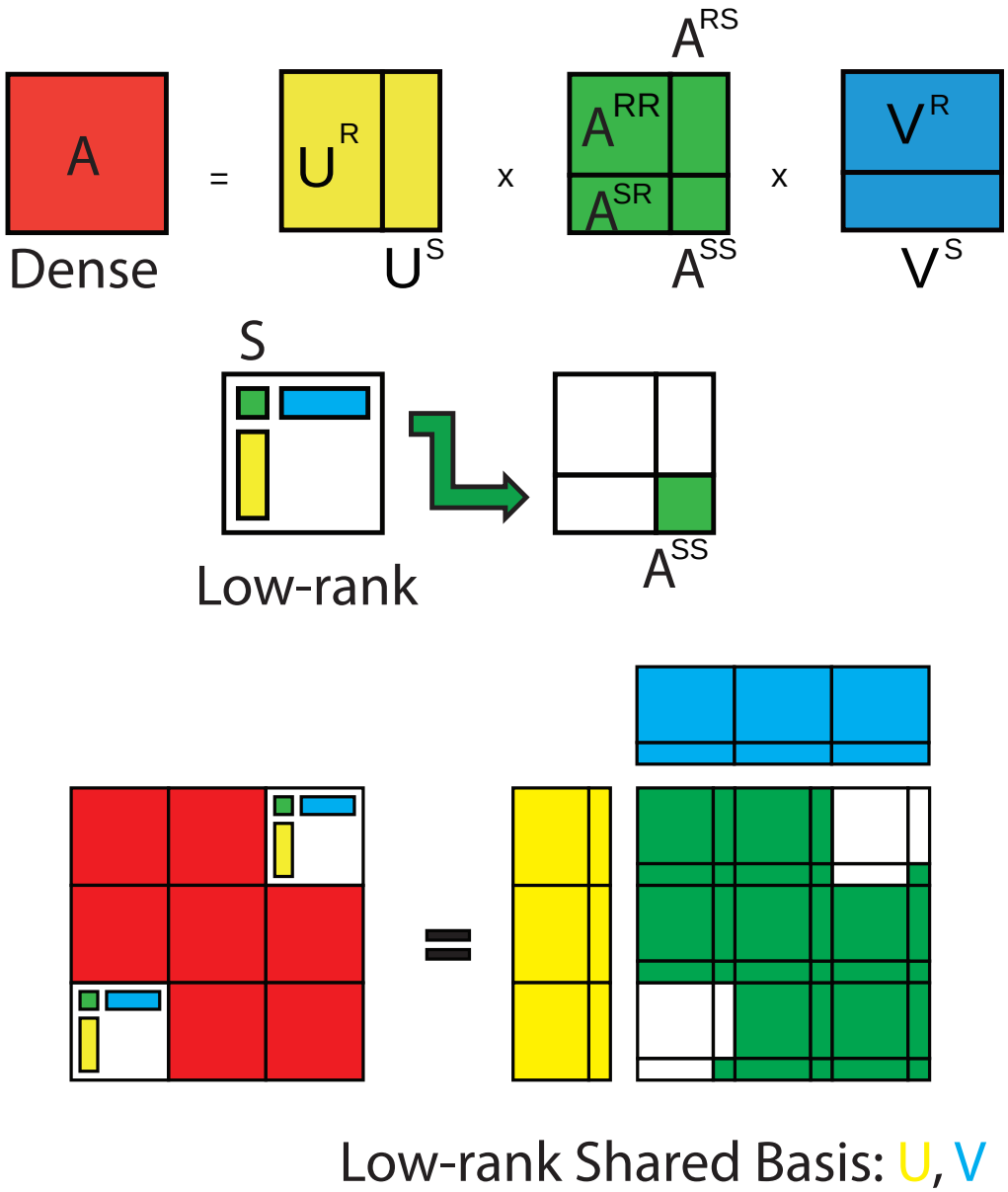


Figure 5.2: Matrix sparsification from full-size shared basis

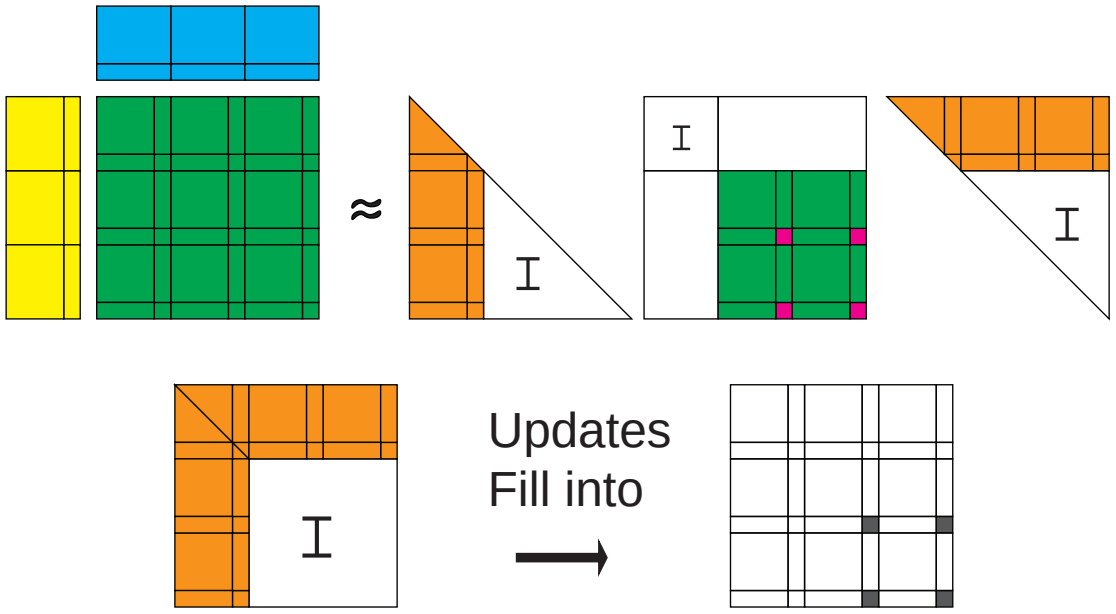


Figure 5.3: Step 1 of the internal block-LU factorization on a 3×3 dense matrix during ULV factorization, and corresponding update locations

can be written as

$$A_{jk} = \begin{pmatrix} U_j^S & U_j^R \end{pmatrix} \begin{pmatrix} A_{jk}^{SS} & A_{jk}^{SR} \\ A_{jk}^{RS} & A_{jk}^{RR} \end{pmatrix} \begin{pmatrix} V_k^S & V_k^R \end{pmatrix}^T \tag{5.6}$$

As mentioned earlier, we introduce the concept of the "skeleton" part (with superscript S) and the "redundant" part (with superscript R). For a low-rank block U^R, V^R, A^{RS}, A^{SR}, and A^{RR} are zero, but for a dense matrix they are non-zero. This allows us to partition both the dense and low-rank blocks using the same split between the skeleton and redundant parts. We then permute the skeleton and redundant parts for all blocks

$$A_{jk} = \begin{pmatrix} U_j^R & U_j^S \end{pmatrix} \begin{pmatrix} A_{jk}^{RR} & A_{jk}^{RS} \\ A_{jk}^{SR} & A_{jk}^{SS} \end{pmatrix} \begin{pmatrix} V_k^R & V_k^S \end{pmatrix}^T \tag{5.7}$$

This results in a conversion from a standard $\mathcal{H}^2$-matrix to an $\mathcal{H}^2$-ULV matrix as shown in Fig. 5.2. The process of operating on each dense or low-rank block is included in figure 5.2, as well as the global view where the original matrix is a mixture of dense and low-rank blocks.

In figure 5.3, we show how the elimination of the first block row/column results in fill-ins for the A^{SS} part only. Note that this holds even when no low-rank blocks exist, and only requires the Schur complement updates to be low-rank. This is more clear when expressed in the form of an equation

$$(U_j^T A_{ji} U_i)(U_i^T A_{ii}^{-1} U_i)(U_i^T A_{ik} U_k) \approx \begin{pmatrix} 0 & 0 \\ 0 & A^{SS} \end{pmatrix} \tag{5.8}$$

Other trailing block partitions A^{RR}, A^{RS}, and A^{SR} are not being updated from the defined behavior in Eq. (5.8). Transforming this equation by separating it into different parts of R and S, we can

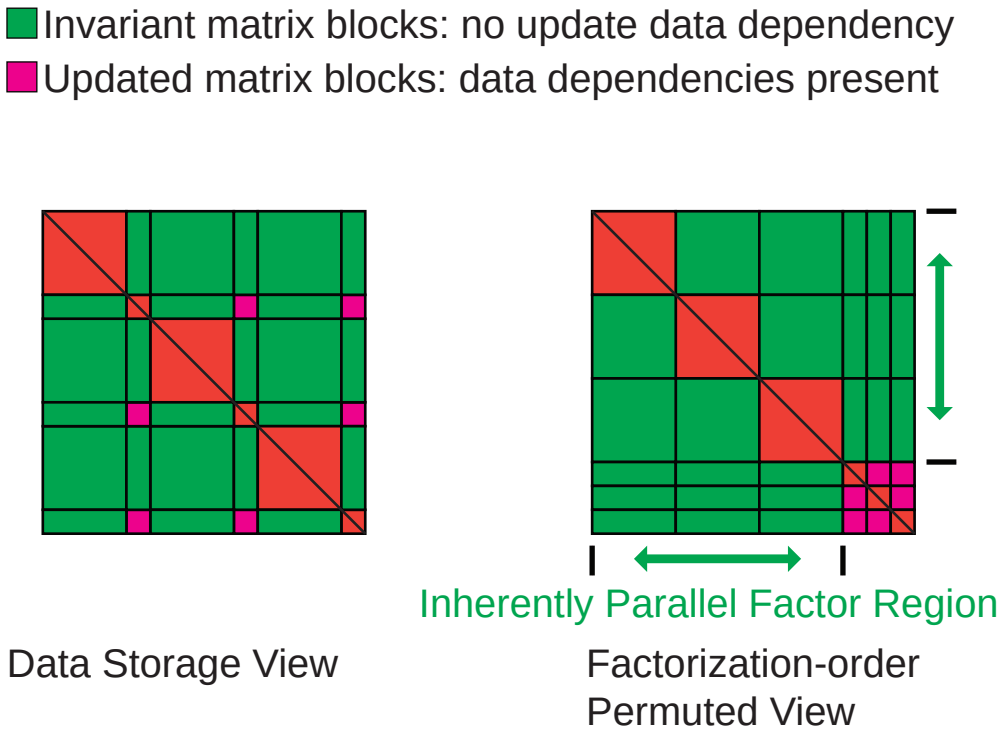


Figure 5.4: Any symmetric permutation of elimination order, invariant matrix blocks are not updated until elimination → Permute to eliminate all A^{RR} before A^{SS} to form an inherently parallel factoring region

obtain the following list of equations that are easier to comprehend

$$\begin{cases} A_{ji}^{RX}(A_{ii}^{XX})^{-1}A_{ik}^{XR} = 0 \\ A_{ji}^{RX}(A_{ii}^{XX})^{-1}A_{ik}^{XS} = 0 \\ A_{ji}^{SX}(A_{ii}^{XX})^{-1}A_{ik}^{XR} = 0 \\ A_{ji}^{SX}(A_{ii}^{XX})^{-1}A_{ik}^{XS} \neq 0 \\ X := R \parallel S \end{cases} \tag{5.9}$$

Fig. 5.4 shows a complete view of the entire factorization, where the invariant matrix blocks with no update data dependency are shown in green and the updated matrix blocks with data dependency are shown in pink. The left figure shows the original view where each block is partitioned into the skeleton and redundant parts. The right figure shows the permuted view where the skeleton part is clustered to the upper-left. From this view, we can see that all of the possible locations of updates are clustered to the bottom-right corner of the A^{SS} region and have removed the trailing diagonal update dependencies inside the A^{RR} region, enabling a parallel elimination of all blocks in different rows and columns. Compared with the traditional approach for a block LU factorization on dense matrices, we have exposed a much larger parallel region by taking advantage of the low-rank compressible Schur complement updates. This concept only requires the Schur complement updates to be low-rank, and remains inherently parallel regardless of the number of dense off-diagonal blocks.

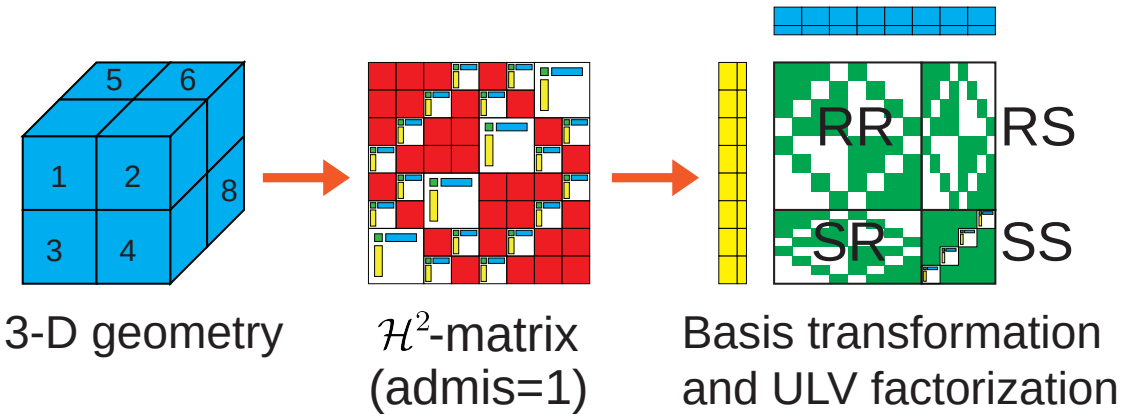


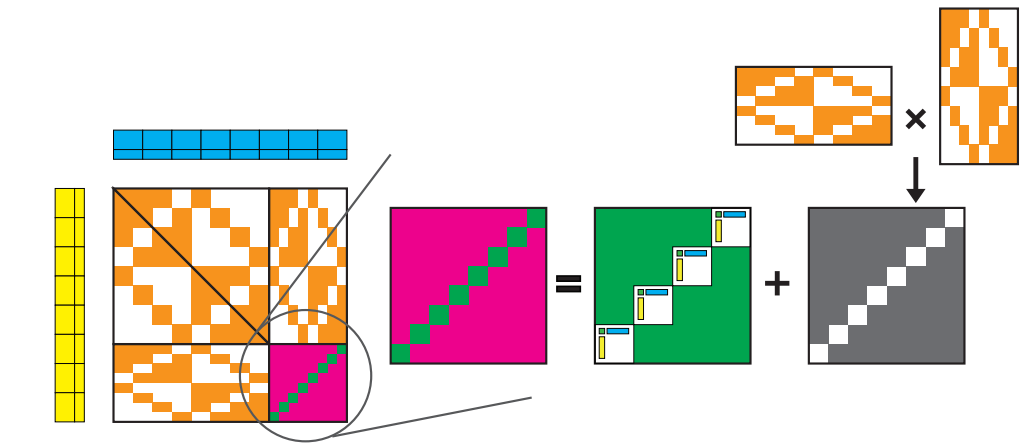
Figure 5.5: A $\mathcal{H}^2$-matrix compression to a simple 3-D geometry

5.2.3 Trailing submatrix update for multi-level

For multi-level ULV factorization of strongly admissible $\mathcal{H}^2$-matrices, we must deal with the trailing submatrix update patterns for the A^{SS} blocks across levels. Filling into the blocks, which are originally low-rank, leads to extra dense matrix blocks created for the next level and eventually having super linear factorization complexity. The chain of fill-ins is a critical problem reported by the original $\mathcal{H}^2$-ULV factorization and LoRaSp. This is the primary reason for introducing the re-compression of the fill-in blocks. In this section, we focus exclusively on the updates that go to the next level of factorization by looking at a simple 3-D cubic geometry which requires a strong admissibility $\mathcal{H}^2$-matrix as shown in Fig. 5.5. In this example, the interaction between cubes that share the same surface are considered as dense blocks. Clustering the redundant part to the upper-left, results in a sparsity pattern shown in the rightmost figure. To consider the sparsity pattern at the next level, we zoom in to take a closer look at the updates created for the A^{SS} region.

On the top picture in Fig. 5.6, we show how directly multiplying the factorized A^{SR} with A^{RS} creates a substantial amount of updates. Some dense matrix blocks are filled into the upper-level low-rank matrices and converted to dense matrix blocks based on the sparsity pattern alone. The naively computed fill-ins do not exploit the low-rankness of the factorization basis to its full extent. In the following analysis, we show that the actual pattern of updates can be limited to only the diagonal blocks of the A^{SS}, as shown in the bottom picture in Fig. 5.6.

To be consistent with the notation used, we again use the letter combination C_{ij} to denote the Schur complement update being introduced to the superscript SS region of the (i, j) matrix block. Similar to the updates in the regions correlating to the redundant basis, we utilize the low-rank approximation results and use the factorization basis for redirecting the updated locations, to expose sparsity (zero entries) in the update matrix. This property of the factorization basis



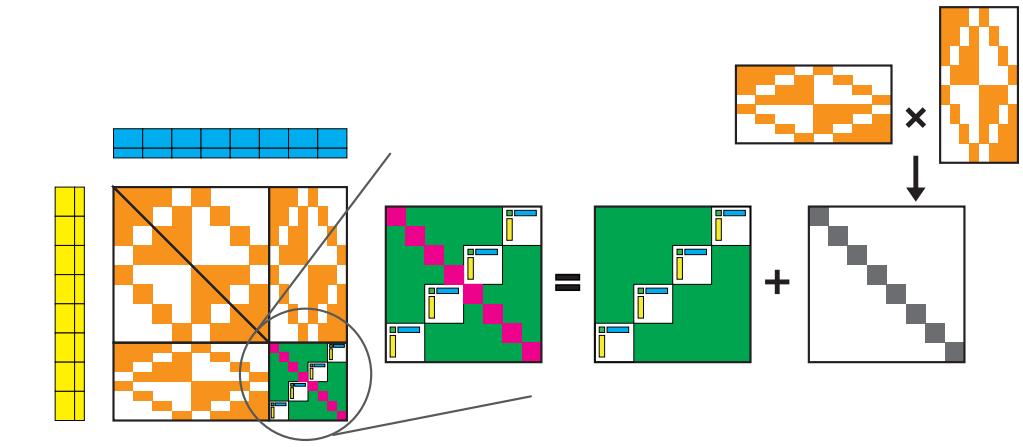


Figure 5.6: Acutal updated locations for ULV factorization when eliminating A^{RR}, symmetric to A^{SS} elimination

removes the need for re-compression during the ULV factorization and greatly simplifies the overall implementation.

We divide the proof into three incremental steps that correlates to the different means for the trailing panel updates produced. Firstly, we show that for all updates introduced to the diagonal blocks

$$C_{jj} = A_{ji}^{SR}(A_{ii}^{RR})^{-1}A_{ij}^{RS} = 0, \forall i \neq j \tag{5.10}$$

Secondly, we show that for all updates introduced to the off-diagonal blocks

$$C_{jk} = A_{ji}^{SR}(A_{ii}^{RR})^{-1}A_{ik}^{RS} = 0, \forall i \neq j \neq k \tag{5.11}$$

Lastly, we show that for a single A^{RS} or A^{SR} that lies in the same block as the eliminated A^{RR}

$$\begin{cases} C_{ji} = A_{ji}^{SR}(A_{ii}^{RR})^{-1}A_{ii}^{RS} = 0, \forall i \neq j \\ C_{ij} = A_{ii}^{SR}(A_{ii}^{RR})^{-1}A_{ij}^{RS} = 0, \forall i \neq j \end{cases} \tag{5.12}$$

We can prove Eq. (5.10) by looking at the following inversion of the 2×2 block dense linear system, taken from portions the $\mathcal{H}^2$-matrix after the transformations of U and V:

$$\begin{pmatrix} A_{jj}^{SS} & A_{ji}^{SR} \\ A_{ij}^{RS} & A_{ii}^{RR} \end{pmatrix}^{-1} = \begin{pmatrix} p & r \\ q & s \end{pmatrix} \tag{5.13}$$

We assume that the diagonal blocks A_{jj}^{SS} and A_{ii}^{RR} are invertible, and that A_{ji}^{SR} and A_{ij}^{RS} are non-zero matrices (otherwise $C_{jj} = 0$ is trivial). We know that the elimination of block A_{jj} produces zero updates to A_{ii}^{RR}, and A_{jj}^{SS} being a subsystem contained inside A_{jj}. The updates in A_{ii}^{RR} from the elimination of A_{jj}^{SS} is also a zero matrix as $A_{ij}^{RS}(A_{jj}^{SS})^{-1}A_{ji}^{SR} = 0$, and the inverted system can be written as

$$\begin{pmatrix} (A_{jj}^{SS} - C_{jj})^{-1} & -(A_{jj}^{SS})^{-1}A_{ji}^{SR}(A_{ii}^{RR})^{-1} \\ -(A_{ii}^{RR})^{-1}A_{ij}^{RS}(A_{jj}^{SS})^{-1} & (A_{ii}^{RR})^{-1} \end{pmatrix} \tag{5.14}$$

Thus, multiplying the original system with the inverted form should result in an identity matrix

$$\begin{pmatrix} A_{jj}^{SS}p + A_{ji}^{SR}q & A_{jj}^{SS}r + A_{ji}^{SR}s \\ A_{ij}^{RS}p + A_{ii}^{RR}q & A_{ij}^{RS}r + A_{ii}^{RR}s \end{pmatrix} = \begin{pmatrix} I & 0 \\ 0 & I \end{pmatrix} \tag{5.15}$$

Expanding the lower-left block on both sides, we have:

$$A_{ij}^{RS}(A_{jj}^{SS} - C_{jj})^{-1} - A_{ii}^{RR}(A_{ii}^{RR})^{-1}A_{ij}^{RS}(A_{jj}^{SS})^{-1} = 0 \tag{5.16}$$

Given that all blocks are non-zero matrices, we complete the proof of $C_{jj} = 0$ in Eq. (5.10).

The proof for Eqs. (5.11) and (5.12) can be arranged together by looking at a 3×3 system and extending the proof of Eq. (5.10)

$$A' = \begin{pmatrix} A_{ii}^{RR} & A_{ij}^{RS} & A_{ik}^{RS} \\ A_{ji}^{SR} & A_{jj}^{SS} & A_{jk}^{SS} \\ A_{ki}^{SR} & A_{kj}^{SS} & A_{kk}^{SS} \end{pmatrix} \tag{5.17}$$

The proof for the two cases focuses on the updates introduced to block A_{jk}^{SS} and A_{kj}^{SS}, from the elimination of A_{ii}^{RR}. The sole difference lies in whether $k = i$ or $k \neq i$, affecting whether the updates

introduced to the last block of A_{kk}^{SS} are zero matrices or not. We can proceed with the proof since this difference in the trailing update characteristic does not play a significant role. Considering another similar system to the previous A' system by applying permutations to the 1st and 2nd block rows and columns we have

$$A'' = \begin{pmatrix} & I & \\ I & & \\ & & I \end{pmatrix} A' P^T = \begin{pmatrix} A_{jj}^{SS} & A_{ji}^{SR} & A_{jk}^{SS} \\ A_{ij}^{RS} & A_{ii}^{RR} & A_{ik}^{RS} \\ A_{kj}^{SS} & A_{ki}^{SR} & A_{kk}^{SS} \end{pmatrix} \tag{5.18}$$

Since the applied permutation keeps the bottom right block $A'_{33} = A''_{33}$, this equality also holds when creating the inverse of both systems. Writing out the equations for both

$$\begin{cases} C'_{kk} = (A_{kj}^{SS} + C_{kj})(A_{jj}^{SS})^{-1}(A_{jk}^{SS} + C_{jk}) \\ C''_{kk} = A_{kj}^{SS}(A_{jj}^{SS})^{-1}A_{jk}^{SS} \\ (A'_{33})^{-1} = (A_{kk}^{SS} - A_{ki}^{SR}(A_{ii}^{RR})^{-1}A_{ik}^{RS} - C'_{kk})^{-1} \\ (A''_{33})^{-1} = (A_{kk}^{SS} - C''_{kk} - A_{ki}^{SR}(A_{ii}^{RR})^{-1}A_{ik}^{RS})^{-1} \end{cases} \tag{5.19}$$

For obtaining these equations, we consider the block LU factorization applied to the 1st and 2nd rows correspondingly and let them update the bottom right block. For the A' system, we assumed that the elimination of A_{ii}^{RR} introduces updates to all trailing blocks except A_{jj}^{SS}, which is already proven by Eq. (5.10). The elimination of A_{jj}^{SS} in the A' system accommodates the updates introduced and carries it over to the bottom right block. For the A'' system, the elimination of A_{jj}^{SS} updates only the A_{kk}^{SS}, as the other blocks lie in the regions of redundancy and by using the factorization basis they do not require any updates. The following elimination of the block A_{ii}^{RR} operates on its original contents to produce the trailing update. By comparing these relations, we can identify that $(A'_{33})^{-1} = (A''_{33})^{-1}$ implies $C'_{kk} = C''_{kk}$. We can formulate the result of $C'_{kk} - C''_{kk}$ as the following:

$$C'_{kk} - C''_{kk} = \begin{pmatrix} A_{kj}^{SS} \\ C_{kj} \end{pmatrix}^T \begin{pmatrix} 0 & (A_{jj}^{SS})^{-1} \\ (A_{jj}^{SS})^{-1} & (A_{jj}^{SS})^{-1} \end{pmatrix} \begin{pmatrix} A_{jk}^{SS} \\ C_{jk} \end{pmatrix} \tag{5.20}$$

No other C_{kj} and C_{jk} combinations other than $C_{kj} = C_{jk} = 0$ can make $C'_{kk} - C''_{kk} = 0$ hold for arbitrary non-zero entries. Thus, we complete the proof for both Eqs. (5.11) and (5.12), by setting $k \neq i$ and $k = i$ correspondingly.

For the remaining case where $i = j = k$, we made a generic assumption that the matrix is non-singular. The factorization of the redundant matrix will produce a non-zero update to the skeleton matrix of the same origin matrix before the orthogonal transformations are applied. In other words, $C_{ii} = A_{ii}^{SR}(A_{ii}^{RR})^{-1}A_{ii}^{RS} \neq 0$. Combining the proof for Eqs. (5.10), (5.11), and (5.12) on the updates introduced to regions of A^{SS} gives

$$A_{ji}^{SR}(A_{ii}^{RR})^{-1}A_{ik}^{RS} \begin{cases} \neq 0, \forall i = j = k \\ = 0, \forall i \neq j \parallel i \neq k \end{cases} \tag{5.21}$$

Eq. (5.21) guarantees the number of dense matrix blocks to be bounded to the same structure as it is being constructed. Therefore, the only computed updates lie on the diagonal blocks.

Briefly summarizing this section, unlike the original ULV factorization or the inverse FMM based on extended sparsification, our method of using factorization basis by design does not require the usage of re-compression during factorization. Using the factorization basis and pre-compressing the trailing updates can be seen as a means to both introduce parallelism by eliminating data dependencies, and also maintain linear algorithmic complexity. These properties play a major role in the implementation of the parallel ULV factorization and substitution, which are described in the later sections.

5.2.4 $\mathcal{H}^2$-matrix construction

The construction of the $\mathcal{H}^2$-matrix includes steps to build the low-rank shared bases, and forming the low-rank coupling matrix for each individual low-rank block. The low-rank approximations applied to each row or column, are independent of each other and can be executed in an embarrassingly parallel fashion. Since the focus of this paper is on the parallelization of the matrix factorization and substitution, we do not go into the details of the $\mathcal{H}^2$-matrix construction nor provide results corresponding to this section.

In this work, we adopt a conventional method for constructing the shared basis known as interpolative decomposition (ID) as shown in Fig. 5.7. The key difference between our method and conventional $\mathcal{H}^2$-matrix construction methods is that we include a factorization basis along with the low-rank approximation basis, which introduces some extra work and requires a pre-factorization step. As shown in Fig. 5.7, ID is based on the sampling of points in the underlying geometry. More complicated and better-performing sampling methods such as HiDR used by H2pack or GOFMM can approximate to the same degree or better accuracy using even fewer number of sampled bodies, but their method is beyond the scope of this paper so we do not discuss it further. Disabling the sampling and always using the entire domain (which is $\mathcal{O}(N)$ size) of well-separated points will lead to a $\mathcal{O}(N^2)$ construction cost but having the best construction accuracy, while any constant sample size reduces this complexity to $\mathcal{O}(N)$.

In order to include the factorization basis of compressed Schur complement updates in Eq. (5.2), we combine the contents of the factorization basis of the near-field with the low-rank basis of the far-field, as shown in Fig. 5.8. Instead of forming all of the possible combinations i, j, k in $A_{ji}A_{ii}^{-1}A_{ik}$ of the Schur complement that arise during the factorization, a simpler alternative is to approximate the term $A_{ji}A_{ii}^{-1}$ instead for approximating U_j. The expression $A_{ji}A_{ii}^{-1}$ uses the same row basis and poses an upper bound in rank for all possible k in different Schur complement $A_{ji}A_{ii}^{-1}A_{ik}$

$$A_{ji}A_{ii}^{-1}A_{ik} \approx \begin{pmatrix} U_j^R & U_j^S \end{pmatrix} \begin{pmatrix} 0 & 0 \\ 0 & A^{SS} \end{pmatrix} \begin{pmatrix} U_k^R & U_k^S \end{pmatrix}^T \tag{5.22}$$

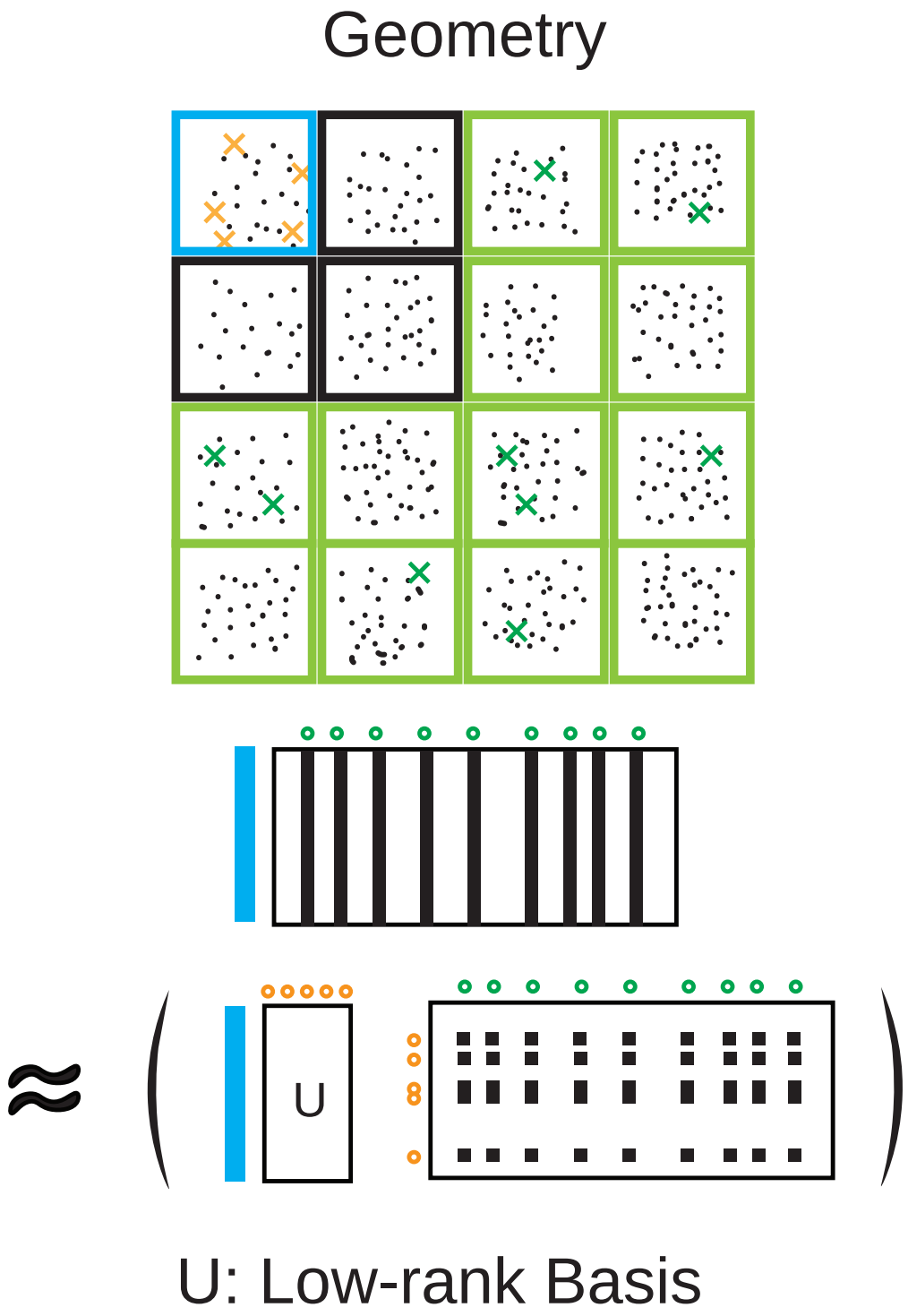


Figure 5.7: Interpolative decomposition on well-separated boxes

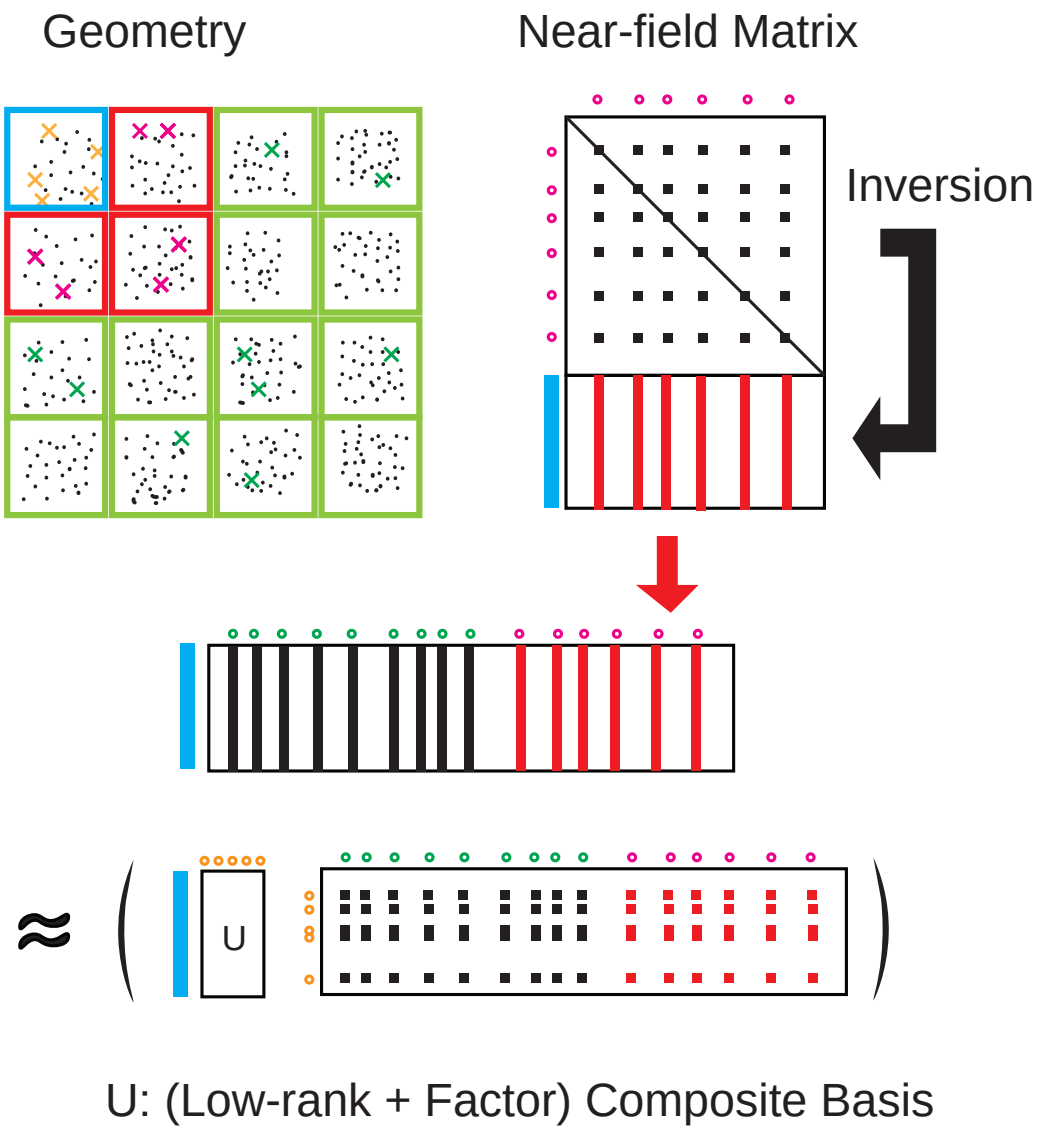


Figure 5.8: Interpolative decomposition on both well-separated and close boxes

In order to produce the internal zero matrix blocks on the first row of the A^{SS} matrix, we need the first two terms on the left hand side to satisfy

$$A_{ji}A_{ii}^{-1} \approx \begin{pmatrix} U_j^R & U_j^S \end{pmatrix} \begin{pmatrix} 0 & 0 \\ p & q \end{pmatrix} \tag{5.23}$$

Working backwards, $A_{ji}A_{ii}^{-1}$ needs to be compressible when using the U_j factorization basis, if U_j can approximate the Schur complement $A_{ji}A_{ii}^{-1}A_{ik}$. By combining the contents of the far field with the $A_{ji}A_{ii}^{-1}$ block and applying a low-rank approximation, we can obtain a composite basis that serves as both the factorization basis and low-rank shared basis.

5.2.5 Pre-factorization

In this section, we focus more on the process of obtaining the compressible Schur complement instead of the means of applying a low-rank approximation on it, for the reason that the compressible contents can be concatenated together with the originally low-rank components of the $\mathcal{H}^2$-matrix to form a composite basis in the construction phase. The difference in the different methods of low-rank approximation can have a large difference in accuracy and performance according to different pieces of literature [11, 15].

In order to include the factorization basis into the shared low-rank basis, we essentially need to factorize the matrix twice; one for the pre-factorization to compute the factorization basis, and another to perform the actual ULV factorization using this shared basis. The fact that both factorizations are inherently parallel still gives us an advantage over existing methods, but we would like to reduce this overhead if possible. In order to alleviate the cost of factorizing the matrix twice, we pre-process and approximate the joint term of $A_{ji}A_{ii}^{-1}$ before the actual low-rank approximation for constructing the basis. One approach is to also adopt sampling for the near-field boxes, as shown in Fig. 5.8. Sampling of the near-field does not reduce the complexity like the sampling for the far-field since the leaf block size is bounded by a constant anyway, but it does reduce the overhead of factorizing twice by a constant. This constant can actually be quite large, since factorization of a dense matrix has $\mathcal{O}(N^3)$ complexity.

Furthermore, we discovered that without factorizing the close field interaction A_{ii}, a good approximation on the content and the rank of $A_{ji}A_{ii}^{-1}$ can be found via iterative solvers. In our implementation which internally uses Cholesky factorization for the dense diagonals, we assumed the kernel function defined generates an input matrix with semi-positive definiteness. Therefore, we used the Gauss-Seidel iterative method for approximating the contents of $A_{ji}A_{ii}^{-1}$ without explicitly factorizing it. For the kernel functions that we have tested, one or two Gauss-Seidel iterations produce a sufficiently accurate approximation of $A_{ji}A_{ii}^{-1}$. We close the description of the construction phase by providing the complete algorithm of $\mathcal{H}^2$-matrix construction using interpolative decomposition as an example while including the pre-factorization, in Algorithm 6. The Gauss-Seidel optimization is specific to the step where $A_{c,c}^{-1}$ is computed in 7 in the provided algorithm.

Algorithm 6: $\mathcal{H}^2$-matrix construction including pre-factorization

Data: Geometry info with boxes $B_0, B_1, B_2...$
Data: Green's Function $G(x, y)$
Output: $\mathcal{H}^2$-matrix A with shared basis U
Output: Skeletons SK_i for each box B_i
Output: Coupling matrices S

for $l \leftarrow levels$ **to** 1 **do**
  for *all* U_i *in level* l **do**
    $S_F \leftarrow$Sample($\forall B_j$ well-separated with B_i);
    $S_C \leftarrow$Sample($\forall B_j$ close to B_i);
    $A_{c,c} \leftarrow G(S_C, S_C)$;
    $A_{Far} \leftarrow G(B_i, S_F)$;
    $A_{Close} \leftarrow G(B_i, S_C)A_{c,c}^{-1}$;
    $(U_i, SK_i) \leftarrow \text{ID}([A_{Far}, A_{Close}])$;
  for *all* A_{ij} *in level* l *where* B_i *and* B_j *are close* **do**
    $A_{ij} \leftarrow G(B_i, B_j)$;
  for *all* S_{ij} *in level* l *where* B_i *and* B_j *are well-separated* **do**
    $S_{ij} \leftarrow G(SK_i, SK_j)$;
  for *all* B_i^{l-1} *in level* $l-1$ **do**
    $B_i^{l-1} \leftarrow \begin{pmatrix} SK_{2i} & SK_{2i+1} \end{pmatrix}$;

5.2.6 Parallel $\mathcal{H}^2$-ULV Factorization

In this section, we cover the $\mathcal{H}^2$-ULV factorization phase. The factorization algorithm is formulated as a level-wise algorithm, where the factorization within each level is completely parallel. However, it does require synchronized merging and reduction operations between levels. By making the factorization inherently parallel within each level, we expose a large parallel region that can take advantage of batched execution on GPUs. Even when comparing with an implementation of the inherently parallel HSS-ULV factorization, we also expect better parallel scalability due to higher numeric operation intensity in the dense off-diagonal blocks for our $\mathcal{H}^2$-ULV factorization.

The details of the factorization phase are shown in algorithm 7. At this phase we assume that the shared basis that includes both the factorization basis and the low-rank basis has been computed. We start the factorization from the bottom most level and treat each level as a BLR^2 matrix and apply the ULV factorization to it. The ULV factorization of each level consists of a sparsification step and factorization step. In the sparsification step, the transpositions (inverses) of the shared basis are multiplied to the dense matrix to obtain a block-sparse matrix, as shown in Fig. 5.2. In the factorization step, a block-Cholesky factorization is performed on the block-sparse matrix. As we have covered in the sections relating to the factorization basis, the individual sections of A^{RR}, A^{SR}, A^{RS}, and A^{SS} does not require an explicit permutation to align with the block factorization order. We apply the factorization only to the corresponding locations in the partitioned matrix and do the same when applying the forward and backward substitution.

Algorithm 7: $\mathcal{H}^2$-matrix factorization

Data: Symmetric $\mathcal{H}^2$-matrix A with shared basis U
Data: Coupling matrices S
1 **for** $l \leftarrow levels$ **to** 1 **do**
2 **for** *all* A_{ij} *in level* l **do**
3 A_{ij}^{SS},A_{ij}^{RS},A_{ij}^{SR},$A_{ij}^{RR} \leftarrow U_i^{-1} A_{ij} U_j^{T-1}$;
4 **for** *all* S_{ij} *in level* l **do**
5 $A_{ij}^{SS} \leftarrow S_{ij}$;
6 **for** *all* A_{ii}^{RR} *in level* l **do**
7 $L(r)_{ii}$,$L(r)_{ii}^T \leftarrow cholesky(A_{ii}^{RR})$;
8 **for** *all* A_{ji}^{RR} *where* $j > i$ **do**
9 $L(r)_{ji} \leftarrow A_{ji}^{RR} L(r)_{ii}^{T-1}$;
10 **for** *all* A_{ji}^{SR} **do**
11 $L(s)_{ji} \leftarrow A_{ji}^{SR} L(r)_{ii}^{T-1}$;
12 $A_{ii}^{SS} \leftarrow A_{ii}^{SS} - L(s)_{ii} L(s)_{ii}^T$;
13 **for** *all* A_{ij}^{l-1} *in level* $l-1$ **do**
14 $A_{ij}^{l-1} \leftarrow \begin{pmatrix} A_{2i,2j}^{SS} & A_{2i,2j+1}^{SS} \\ A_{2i+1,2j}^{SS} & A_{2i+1,2j+1}^{SS} \end{pmatrix}$;
15 L_{00},$L_{00}^T \leftarrow cholesky(A_{00})$;

In our parallel ULV factorization, we intentionally skip all Schur complements calculations except for the self-to-self redundant to skeleton Schur complement (on line 12). As we have covered in Eq. (5.21), this is the only trailing update, and no other trailing updates are computed nor recompressed. The same algorithmic complexity analysis for other $\mathcal{H}^2$-ULV factorizations also applies to our method, except we can do it in an inherently parallel manner.

5.2.7 Parallel Forward and Backward Substitution

The forward and backward substitution applies the operations $x = (L^T)^{-1} L^{-1} b$ to obtain the solution to the linear system. Substitution is an essential part of the direct solver or preconditioner, but has been difficult to parallelize due to its inherently serial nature. Although, the HSS-ULV factorization by [20] and the $\mathcal{H}^2 - ULV$ factorization by [41] can process the factorization phase in an inherently parallel manner, these methods do not provide any means to parallelize the substitution phase. In the present work, we show for the first time that the substitution phase can also be parallelized. We only provide the algorithm for performing the forward substitution, since the backward substitution can be applied in a similar manner.

We first describe the naïve method for forward substitution of an $\mathcal{H}^2$-matrix based on the block TRSV algorithm in Algorithm 8. The loop on line number 7 introduces a write after write data dependency and also causes a read after write dependency for the trailing b_i^R in line number 5. When using the block version of the TRSV/TRSM algorithm, the forward and backward substitution will result in an inherently serial algorithm. However, closer examination of the substitution algorithm

Algorithm 8: A naïve $\mathcal{H}^2$-matrix forward substitution.

Data: ULV-factorized $\mathcal{H}^2$-matrix $A = ULL^TU^T$
Data: Vector b
Output: Vector $y = L^{-1}U^{-1}b$ overwritten in b

for $l \leftarrow$ *levels* **to** 1 **do**
  for *all* b_i *in level* l **do**
    $\begin{pmatrix} b_i^R \\ b_i^S \end{pmatrix} \leftarrow U_i^{-1} b_i$;
  for *all* L_{ii}^{RR} *in level* l **do**
    $b_i^R \leftarrow (L_{ii}^{RR})^{-1} b_i^R$;
    for *all* L_{ji}^{RR} *where* $j > i$ **do**
      $b_j^R \leftarrow b_j^R - L_{ji}^{RR} b_i^R$;
    for *all* L_{ji}^{SR} **do**
      $b_j^S \leftarrow b_j^S - L_{ji}^{SR} b_i^R$;
  for *all* b_i^{l-1} *in level* $l-1$ **do**
    $b_i^{l-1} \leftarrow \begin{pmatrix} b_{2i}^S \\ b_{2i+1}^{SS} \end{pmatrix}$;

reveals that, the inverse computation of the triangular factors is a block matrix with low-rank structures, which we can further apply the Schur complements computations to. In other words, we can use a low-rank approximation of L^{-1} and $(L^T)^{-1}$, to avoid the update after update data dependency. In order to parallelize the substitution phase, we take advantage of the following two properties:

- The write after write and read after write dependency affect only the RR regions of the matrix
- The existence of the off-diagonal dense matrix is the source of such data dependencies. (For example, a diagonal block RR structure in HSS matrices does not have this data dependency)

Because the portions of the vector b corresponding to the skeleton are not being solved until the next level, the updates can happen later than the redundant portion and be processed in parallel. We demonstrate this process using a 4×4 lower-triangular block matrix with all of the off-diagonal blocks filled with dense matrix blocks that correspond with the RR portions of LL^T. This demonstrates the triangular matrix inversion for the most challenging case to parallelize. Even for such a case, we show that our algorithm still has a large degree of parallelism.

$$L = \begin{pmatrix} L_{00} & & & \\ A_{10}(L_{00}^T)^{-1} & L_{11} & & \\ A_{20}(L_{00}^T)^{-1} & A_{21}(L_{11}^T)^{-1} & L_{22} & \\ A_{30}(L_{00}^T)^{-1} & A_{31}(L_{11}^T)^{-1} & A_{32}(L_{22}^T)^{-1} & L_{33} \end{pmatrix} \tag{5.24}$$

The L factor shown above is constructed with the zeroed redundant trailing fill-ins described in Eq. (5.21). The L_{ii} blocks are the actual factors of $A_{ii} = L_{ii}L_{ii}^T$ due to the absence of the trailing updates. To invert the L factor, we decompose the block matrix into a product of lower-triangular matrices,

where each multiplicand contains one block column of the original L, as $L = L_0L_1L_2L_3$. The inversion of the complete L factor consists of the separate inversion of the 4 multiplicand matrices and multiplied in reverse order, as $L^{-1} = (L_0L_1L_2L_3)^{-1} = L_3^{-1}L_2^{-1}L_1^{-1}L_0^{-1}$. For simplicity, we first consider only the contents of L_0 and L_1. The inversion of L_2 and L_3 factors can be derived following similar rules. The contents of L_0 and L_1 are shown below.

$$L_0 = \begin{pmatrix} L_{00} & & & \\ A_{10}(L_{00}^T)^{-1} & I & & \\ A_{20}(L_{00}^T)^{-1} & & I & \\ A_{30}(L_{00}^T)^{-1} & & & I \end{pmatrix} \tag{5.25}$$

$$L_1 = \begin{pmatrix} I & & & \\ & L_{11} & & \\ & A_{21}(L_{11}^T)^{-1} & I & \\ & A_{31}(L_{11}^T)^{-1} & & I \end{pmatrix} \tag{5.26}$$

Inverting the L_0 and L_1 separately each gives

$$L_0^{-1} = \begin{pmatrix} L_{00}^{-1} & & & \\ -A_{10}A_{00}^{-1} & I & & \\ -A_{20}A_{00}^{-1} & & I & \\ -A_{30}A_{00}^{-1} & & & I \end{pmatrix} \tag{5.27}$$

$$L_1^{-1} = \begin{pmatrix} I & & & \\ & L_{11}^{-1} & & \\ & -A_{21}A_{11}^{-1} & I & \\ & -A_{31}A_{11}^{-1} & & I \end{pmatrix} \tag{5.28}$$

In these inverted triangular factors, we merge the product of $(L_{ii}^T)^{-1}L_{ii}^{-1} = A_{ii}^{-1}$ for $i = 0, 1$. Multiplying $L_1^{-1}L_0^{-1}$ gives

$$L_1^{-1}L_0^{-1} = \begin{pmatrix} L_{00}^{-1} & & & \\ -L_{11}^{-1}A_{10}A_{00}^{-1} & L_{11}^{-1} & & \\ -A_{20}A_{00}^{-1} + C_{20}A_{00}^{-1} & -A_{21}A_{11}^{-1} & I & \\ -A_{30}A_{00}^{-1} + C_{30}A_{00}^{-1} & -A_{31}A_{11}^{-1} & & I \end{pmatrix} \tag{5.29}$$

where $C_{20} = A_{21}A_{11}^{-1}A_{10}$ and $C_{30} = A_{31}A_{11}^{-1}A_{10}$, which are the Schur complement update term from the elimination of block A_{11}. As the matrix notation A corresponds to RR regions of the matrix factor, the intermediate results C_{20} and C_{30} are both zero matrix blocks as can be seen from Eq. (5.21). Rewriting the intermediate product of $L_1^{-1}L_0^{-1}$ using the zeroed trailing redundant fill-ins, and transforming $L_{ij} = A_{ij}(L_{jj}^T)^{-1}$ gives

$$L_1^{-1}L_0^{-1} = \begin{pmatrix} L_{00}^{-1} & & & \\ -L_{11}^{-1}L_{10}L_{00}^{-1} & L_{11}^{-1} & & \\ -L_{20}L_{00}^{-1} & -L_{21}L_{11}^{-1} & I & \\ -L_{30}L_{00}^{-1} & -L_{31}L_{11}^{-1} & & I \end{pmatrix} \tag{5.30}$$

Performing a similar operation on L_2^{-1} and L_3^{-1} gives us the final form of L^{-1}

$$\begin{pmatrix} L_{00}^{-1} & & & \\ -L_{11}^{-1}L_{10}L_{00}^{-1} & L_{11}^{-1} & & \\ -L_{22}^{-1}L_{20}L_{00}^{-1} & -L_{22}^{-1}L_{21}L_{11}^{-1} & L_{22}^{-1} & \\ -L_{33}^{-1}L_{30}L_{00}^{-1} & -L_{33}^{-1}L_{31}L_{11}^{-1} & -L_{33}^{-1}L_{32}L_{22}^{-1} & L_{33}^{-1} \end{pmatrix} \tag{5.31}$$

By applying the forward and backward substitution using this reformed L^{-1}, the triangular solves are transformed into matrix-vector multiplications, which removes the data dependency during the substitution. Implementation-wise, we have the option to explicitly compute L^{-1} as in Eq. (5.9). We have decided not to do this, because of the number of floating point operations to operate on a dense matrix block. The cost of matrix-matrix operations on dense blocks, compared with operating with vectors, is significantly higher, with no apparent benefit in other aspects, such as in communication patterns for distributed memory or implementation difficulties.

5.3 Design Considerations for GPUs

Our hierarchical low-rank factorization code has been developed as a higher-level set of algorithms that internally operates on dense matrix structures using BLAS/LAPACK routines. We have covered in the previous section the algorithms for construction, factorization, and substitution. The smallest unit of computation in these algorithms are single matrix blocks, and the operations applied are defined in BLAS and LAPACK. For this reason, rather than building a custom set of GPU/CPU kernels, we decided to use the highly optimized libraries written for GPUs, such as cuBLAS, MAGMA: [49, 50, 25], and OneAPI MKL for CPUs. cuBLAS and cuSOLVER are proprietary libraries that are included with each release of CUDA. They are capable of delivering the best performance on a wide range of basic matrix operations. MAGMA, on the other hand, provides a wider range of interfaces and algorithms, such as batched data copying and batching of variable-sized matrices. OneAPI MKL is part of Intel's implementation of SYCL standard and DPC++ and has extended the original MKL, an x86-64 BLAS and LAPACK implementation, to heterogeneous computing platforms that run with OpenCL. The programming model of SYCL and OneAPI DPC++ promises a bright future for performance portability across platforms and different types of hardware. Fugaku at RIKEN which has A64FX CPUs, Frontier at ORNL which has AMD GPUs, and Aurora at ANL has Intel GPUs. These are all good candidates for cross-platform tools. As our primary interest lies in delivering the best performance with the hardware we currently have, we view the usage of such performance portable programming models as a future step to take. Experiences and optimizations written for CUDA and NVIDIA libraries are also helpful when switching to more complex tools like SYCL for deciding different design considerations, such as choosing from the internal data storage layout or padding and alignment options. In this section, we will discuss the various design decisions we made for the highly optimized implementation of our algorithm on GPUs.

5.3.1 Variable-size batch vs. constant-size batch

We have introduced the ULV factorization as an inherently parallel algorithm, which can be implemented using a series of inherently parallel loops. To extract the best performance, these loops need to be executed as batched calls due to the relatively small size of the matrix blocks. The

NVIDIA libraries cuBLAS and cuSOLVER provide batched interfaces for most functions that we use, such as Cholesky factorization (POTRF), triangular matrix solve (TRSM), and matrix-matrix multiplications (GEMM). Although delivering very high performance even for relatively small matrix sizes, these batched functions require the batch of matrices to have matching dimensions and strides. On the other hand, MAGMA and the group call in OneAPI MKL provide both constant-size batch interfaces and variably-sized batch interfaces. In our case, batched calls for varying matrix sizes enable us to use adaptive matrix ranks when constructing the $\mathcal{H}^2$-matrix, which results in a significant improvement in both performance and memory efficiency.

We have run benchmarks on these different libraries on our computational workload. Shortly summarizing the performance, we discovered a significant performance degradation when using variable-sized matrix dimensions and strides. Even if running on the same dimension but with a variable-size batching interface, the performance can be roughly 50% slower than the constant-sized version. For this reason, we decided that for most use cases where memory is sufficient, zero-padding the matrix with lower ranks to the maximum in the batch gives the best performance instead of using the variable-sized batch provided by MAGMA, although some computational FLOPs are being wasted from padding.

For processing matrices with different sizes, padding zeros to the unmatched dimensions is a useful technique to take advantage of the batched routines. We align the dimensions of the redundant and the skeleton to the maximum of the level, as well as making this dimension multiples of 4, as the cuBLAS and cuSOLVER libraries have suggested. In our case however, the primary challenge lies in simply padding zeros to the deficient dimensions can only work with the matrix-matrix multiply operations that exist in matrix sparsification as well as Schur complements computations. Completely zeroed-out rows and columns halt the Cholesky factorization computations as it encounters diagonal zero entries and produces division by zero errors when later applied in the triangular matrix solves. During the factorization phase, we used a batched AXPY call that writes entries to the diagonal fill up the zero rows and columns that will appear in the factorization. The cuBLAS and cuSOLVER libraries do not provide a native AXPY interface, so we used the batched GEMM interface instead by setting the input M and K to 1 so that LDB and LDC correspond to the functions of INCX and INCY parameters accordingly.

5.3.2 Optimizations for ULV Factorization

In the previous section, we introduced a ULV factorization based on an internal block Cholesky factorization in Algorithm 7. The naïve implementation already exposed large regions of parallelism, where each diagonal block could be factorized independently. Compared to simple multi-threading with "parallel for", the batched API requires additional considerations as we will describe in the following subsection.

5.3.2.1 Temporary storage for matrix sparsification

The primary concern for an effective GPU implementation is the amount of temporary storage that is needed for the factorization, which must not exceed the total amount of the total available on board memory. The composite basis U needs to be stored explicitly in full dense matrix format, for accessing the skeleton and the redundant parts of the basis. The matrix sparsification process requires the same amount of temporary storage to store the intermediate result of basis transformation as the storage required by the original form of the matrix. In the ideal case where we can always allocate an abundant amount of temporary storage, we can launch a single batched GEMM call for all basis transformations. A more practical approach that is adopted by our implementation is to allocate the amount needed for the diagonals. We believe that having the amount of temporary storage matching the number of diagonal blocks is a good call for two reasons. Firstly, the diagonal blocks are factorized in parallel and can happen concurrently. If we want to launch a single batched POTRF call for all diagonal blocks, the sparsification process needs to have the U and V transformations of all of the diagonal blocks in the GPU memory. Secondly, the total number of blocks including the off-diagonals for each level is linearly dependent on the number of diagonal blocks. Therefore, even if we cannot launch a single batched GEMM call for all blocks, the number of launched batched GEMMs is bounded by a constant, which is the average number of blocks per column. We give a more detailed and quantitative analysis of this optimization, in the "Numerical Results" section with NVIDIA profiler results.

5.3.2.2 Reusing intermediate results of TRSM

For entries in the lower triangular factor, two transformations are being applied from the right side to the original entry.

$$\begin{cases} L(r)_{ij} = (U_i^R)^T A_{ij} U_j^R (L(r)_{jj}^T)^{-1}, i > j \\ L(s)_{ij} = (U_i^S)^T A_{ij} U_j^R (L(r)_{jj}^T)^{-1} \end{cases} \tag{5.32}$$

For the blocks on the same columns of the lower triangular factor, the term $U_j^R (L(r)_{jj}^T)^{-1}$ can be reused to reduce the number of TRSM operations. The factorization algorithm is described in Algorithm 9, featuring a new temporary storage V to store the intermediate result of $U_j^R (L(r)_{jj}^T)^{-1}$.

5.4 Distributed-memory Implementation

Our algorithm for parallel $\mathcal{H}^2$-ULV factorization and substitution features decoupled diagonal blocks, which allows us to use the simple 1-D row/column partitioning that has been used for fast multipole method (FMM) and HSS implementations for distributed memory. This is different from distributed memory implementations of dense direct solvers like ScaLAPACK, which use block cyclic partitioning. The primary reason for using block cyclic partitioning is to maximize load balance when there are dependencies on the trailing updates. Our $\mathcal{H}^2$-ULV factorization removes such dependencies, so

Algorithm 9: Improved $\mathcal{H}^2$-matrix factorization

Data: symmetric $\mathcal{H}^2$-matrix A with shared basis U
Data: coupling matrices S
for $l \leftarrow levels$ **to** 1 **do**
  for *all* A_{ii} *in level* l **do**
    A_{ii}^{SS},A_{ii}^{RS},A_{ii}^{SR},$A_{ii}^{RR} \leftarrow U_i^{-1} A_{ii} U_i^{T-1}$;
    $L(r)_{ii}$,$L(r)_{ii}^T \leftarrow cholesky(A_{ii}^{RR})$;
    $L(s)_{ii} \leftarrow A_{ii}^{SR} L(r)_{ii}^{T-1}$;
    $A_{ii}^{SS} \leftarrow A_{ii}^{SS} - L(s)_{ii} L(s)_{ii}^T$;
    $V_i \leftarrow U_i^{T-1} L(r)_{ii}^{T-1}$;
  for *all* A_{ij} *in level* l *where* $i \neq j$ **do**
    A_{ij}^{SS},A_{ij}^{RS},A_{ij}^{SR},$A_{ij}^{RR} \leftarrow U_i^{-1} A_{ij} V_j$;
    if $i > j$ **then**
      $L(r)_{ij} \leftarrow A_{ij}^{RR}$;
    $L(s)_{ij} \leftarrow A_{ij}^{SR}$;
  for *all* S_{ij} *in level* l **do**
    $A_{ij}^{SS} \leftarrow S_{ij}$;
  for *all* A_{ij}^{l-1} *in level* $l-1$ **do**
    $A_{ij}^{l-1} \leftarrow \begin{pmatrix} A_{2i,2j}^{SS} & A_{2i,2j+1}^{SS} \\ A_{2i+1,2j}^{SS} & A_{2i+1,2j+1}^{SS} \end{pmatrix}$;
L_{00},$L_{00}^T \leftarrow cholesky(A_{00})$;

there is no need to use a block cyclic partitioning. Using the simple row/column partitioning scheme has dramatically simplified our implementation, and allows us to use the same structure as a shared memory implementation for most of our distributed implementation.

For the libraries chosen, limited support for MPI and direct means for device-to-device communications in the SYCL standards and Intel's implementation of DPC++ has also been a factor for us to consider in the long term. For native CUDA applications, although we are fully aware of the existence of CUDA-aware MPI implementations such as OpenMPI and MVAPICH, we choose to use NCCL for communicating among GPUs. The primary reason for using NCCL over MPI is the better integration with the CUDA programming model of less host-device synchronization overhead before initiating any collective communications. NCCL communicators are fully integrated with cudaStreams and launch explicit kernels for communication and peer access, which automatically resolves the compute-communication data dependency within the stream. One drawback of using NCCL is that it does not have all of the collectives that the MPI standard defines. Another issue is that it limits the portability to other distributed memory platforms that do not have NVIDIA GPUs. NCCL also does not support AllGatherv, but our choice to use a constant rank during the batched matrix operations allows us to get away with AllGather.

In order to minimize communication, it is necessary to map the proximity in the underlying geometry to the proximity of the process distribution. Though for higher dimensional problems,

such a mapping is not possible, and there will always be a set of points that are close in the geometry but placed far apart in the process distribution. The use of space-filling curves has dramatically reduced the number of neighbor communications compared to using any cyclic process distributions. In our implementation, we used a 1-D column process distribution for better data-locality in the LL^T factorization, where the diagonal $(L^T)^{-1}$ needs to be applied to all of the off-diagonal blocks located on the same column. If the internal operations are factored into the upper triangular form $U_*^T U_*$, using the 1-D row process distribution will become more advantageous, as they share a significant commonality.

5.4.1 Communications in Factorization

In the $\mathcal{H}^2$-ULV factorization, the upper levels have relatively small amount of computation, but could result in a large amount of communication if not implemented carefully. We use a common technique for hierarchical algorithms such as FMM and multigrid, where a hierarchical AllGather is used to merge the contents while computing the upper levels redundantly. The hierarchical AllGather forms a binary tree with split communicators at each level, as described in Fig. 5.9 along with the process distribution pattern. Although some contents in each node might be duplicated, the tree representation of the $\mathcal{H}^2$-matrix representation remains distributed across all available processing units. More precisely, the AllGather is required when performing the following operation

$$A_{ij}^{l-1} : P_{0,1} \leftarrow \begin{pmatrix} A_{2i,2j}^{SS} : P_0 & A_{2i,2j+1}^{SS} : P_1 \\ A_{2i+1,2j}^{SS} : P_0 & A_{2i+1,2j+1}^{SS} : P_1 \end{pmatrix} \tag{5.33}$$

which is the last step of the $\mathcal{H}^2$-ULV factorization algorithm inside each level. Column $2j$ and column $2j+1$ are computed and stored on different processor ranks. To minimize the data movement, we directly perform memory writes, and also store the low-rank coupling matrices directly to the merged locations. An alternative to the AllGather collective is to use AllReduce to the merged block on the next level. Using AllReduce greatly simplifies the implementation as shown in Eq. (5.34).

$$A_{ij}^{l-1} \leftarrow \begin{pmatrix} A_{2i,2j}^{SS} & 0 \\ A_{2i+1,2j}^{SS} & 0 \end{pmatrix} : P_0 + \begin{pmatrix} 0 & A_{2i,2j+1}^{SS} \\ 0 & A_{2i+1,2j+1}^{SS} \end{pmatrix} : P_1 \tag{5.34}$$

Two subjects in this approach for distributed-memory communication are worth noting that show significance to the results. Firstly, for factorization only, both the number of collective communication function calls and the message sizes are independent of the problem size N. This is true because the matrix factorization for levels under $\log_2 P$ levels below the root is computed completely locally inside each processing unit. In a more extreme comparison, the factorization of a $\mathcal{H}^2$-matrix with 2 levels using 4 processing units, has the same communication complexity as factorizing another $\mathcal{H}^2$-matrix with 20 or even 200 levels with the same leaf size and admissibility condition, despite having 10x and 100x difference in memory and algorithmic complexities.

Secondly, the processes holding the gathered or reduced content perform redundant computation on the replicated data. The amount of replicated data and redundant computation is dependent

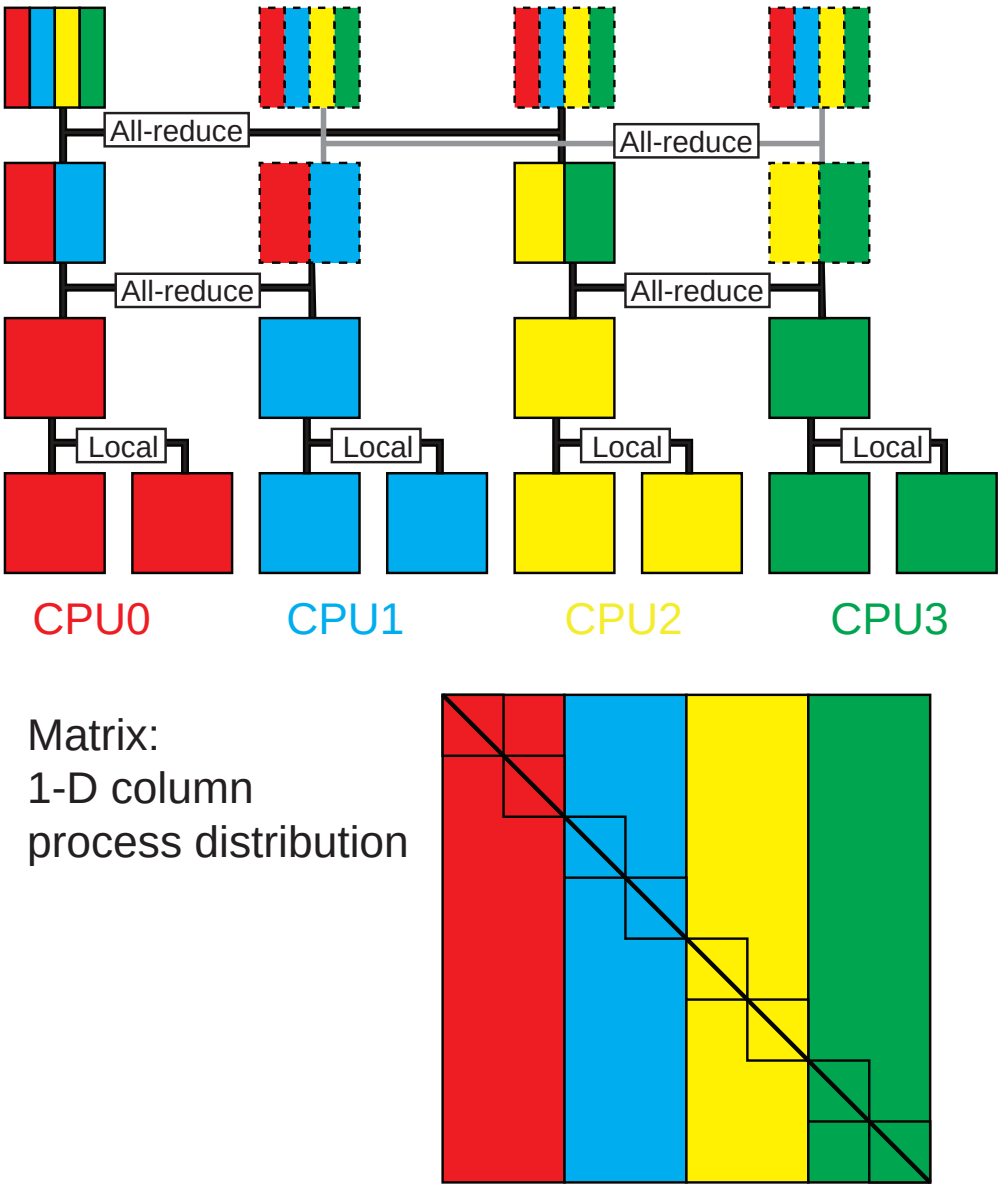


Figure 5.9: Merging pattern for leftover A^{SS} blocks from ULV-factorization

on the number of levels, as well as the number of available processes. The total amount of work computed beyond level $l = \log_2 P$ is $\mathcal{O}(P \log_2 P)$, where P is the number of processors. Among this work, only $\mathcal{O}(2^{\log_2 P}) = \mathcal{O}(P)$ is distinctive. Although for a fixed number of computing units, this number becomes a constant, with an increasing number of processes, this factor of $\mathcal{O}(P \log_2 P)$ becomes significant.

The redundant computation and replicated data during the factorization become useful during the forward and backward substitution. Holding the same content in different processes saves the effort of broadcasting the split partitions to the children when moving from the root to the leaf during the substitution. The gradual reduction in parallel regions for levels beyond level=$\log_2 P$ also makes many processors idle if the redundant work is not performed. In other words, the redundant computation utilizes the otherwise idle compute resources in order to reduce communication.

5.4.2 Communications in Substitution

Forward substitution features the same merging strategy as that of the factorization. The substitution algorithm also requires neighbor-to-neighbor communications, which is similar to that of FMM and $\mathcal{H}^2$ matrix-vector multiplication. Compared to the factorization, which only performs merge operations between levels and has very high intensity of floating point operations, the substitution is significantly more memory and communication bound even for our inherently parallel version. In addition to the previously used merging communication when moving between levels of the $\mathcal{H}^2$-matrix, two different patterns of communication are present during the substitution; 1) summing the updated contents among neighbors, and 2) broadcasting the finalized results to neighbors, as shown in Fig. 5.10. Despite the difference in the type of collective, we can reuse the same communicator for these collectives. In fact, the same neighbor communicator can be reused even for $\mathcal{H}^2$ matrix-vector multiplication as well as in $\mathcal{H}^2$-matrix construction.

As the number of closely interacting neighbors in a geometry configuration with a fixed admissibility condition is constant, the cost of neighbor communications can be conditionally bounded by a constant. Both the summation (reduction) and broadcast message sizes are dependent on the size of the locally computed content. The size of the substituted local vector has a length of $\frac{N}{P}$, being the same as the overall memory complexity $\mathcal{O}(\frac{N}{P})$ of the $\mathcal{H}^2$-matrix. In the weak scaling experiments where the amount of work distributed to each processing unit is the same, $\mathcal{O}(\frac{N}{P})$ becomes $\mathcal{O}(1)$. As a result of communicating on all levels and interleaving with the substitution computations, the neighboring communications appear to be more costly latency-wise compared to the simple reduce-broadcast communication pattern when merging between levels. This is also a contributing factor to the eventual loss of scalability in the forward and backward substitution algorithms when compared with the factorization.

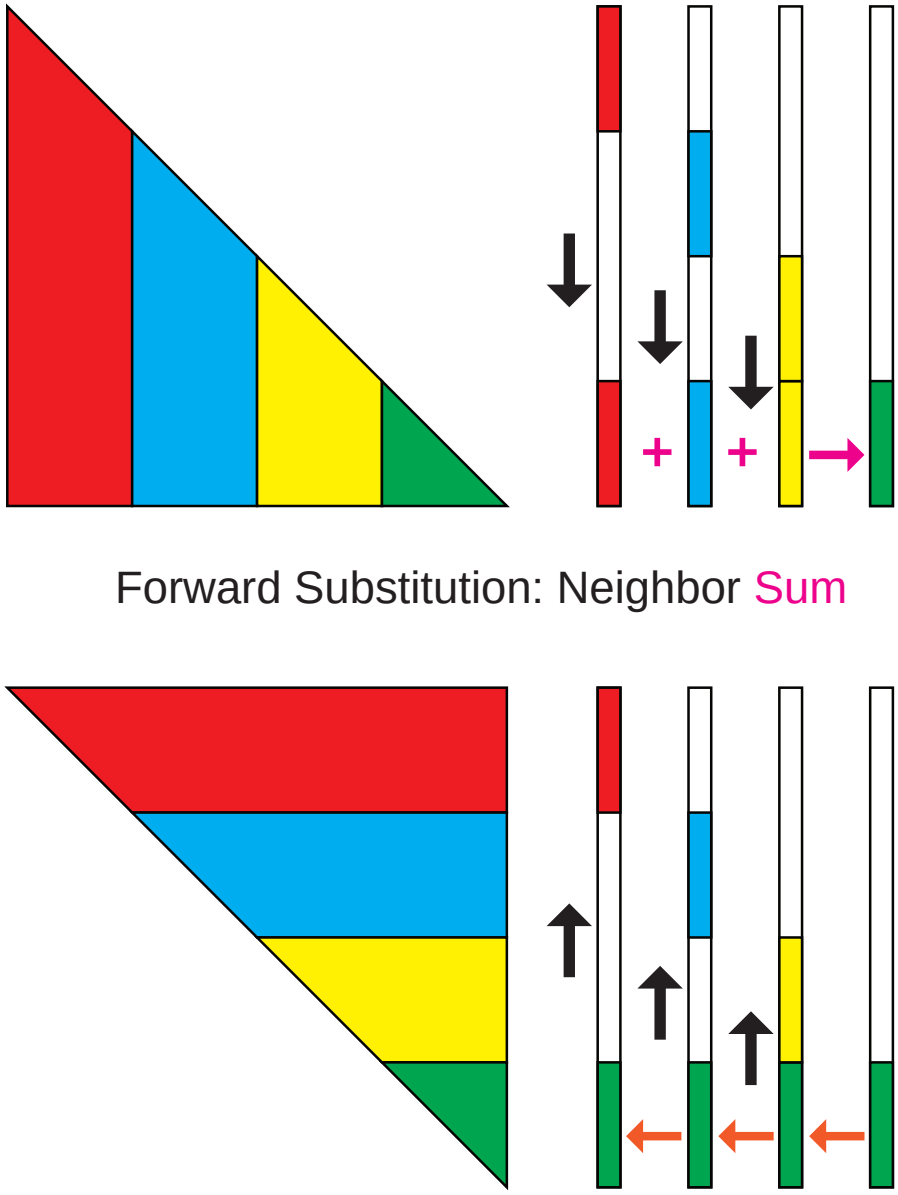


Figure 5.10: Neighbor communication pattern for forward and backward substitution

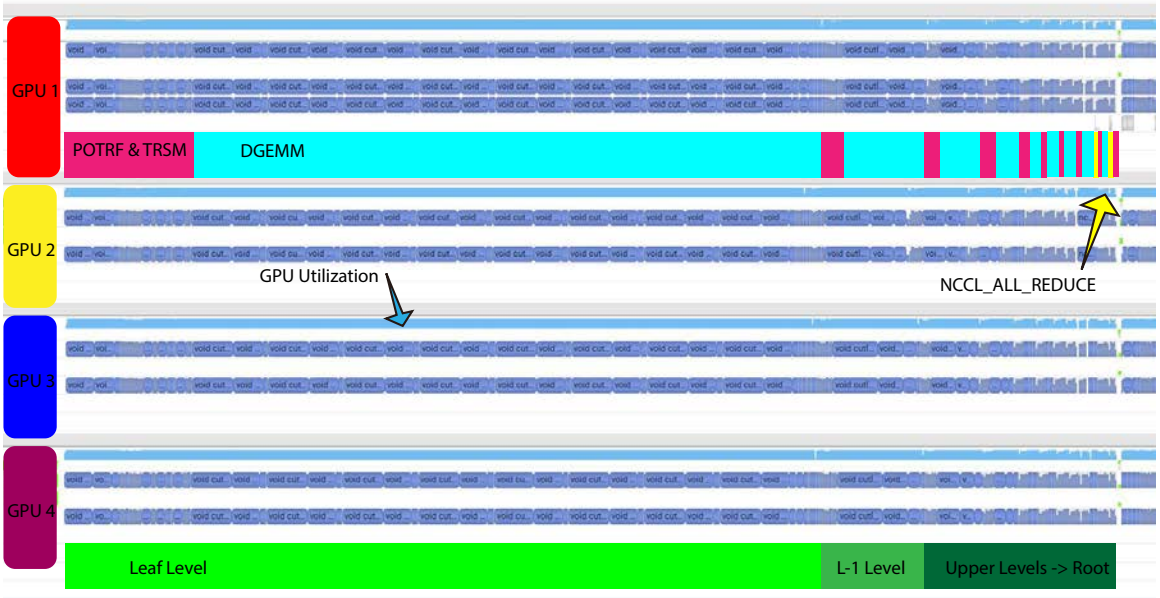


Figure 5.11: Nsys profile screenshot for factorization on 4x A100

In the previous section, we claimed that explicitly forming L^{-1} does not simplify the communications, as the neighboring communication is still needed due to the diagonal contents not being present in all processes for computing the explicit inversion of the off-diagonal blocks in Eq. (5.31). Our approach to the forward and backward substitution showed a significant impact on the neighboring communication patterns. For the naïve block-TRSV-based algorithm, both the reduction as well as the broadcasting of the solved results are serialized due to the data dependencies from the previous blocks. Our parallel approach uses only TRSV on the diagonal blocks, and the off-diagonal computations have been modified into a triangular matrix-vector multiplication. This transition from a triangular solve to a matrix-vector multiplication removes the data dependency during the substitution. This greatly reduces the idling of processes and also enables separate communication kernel launching to achieve even lower latency.

5.5 Numerical Results

In the section, we present the numerical results on two different test cases; 1) 3-D Laplace equation on a uniformly distributed spherical surface geometry, 2) 3-D Yukawa potential on Hemoglobin molecule (Fig. 5.12). As there have been very few attempts to run any hierarchical low-rank matrix factorization on GPUs in distributed memory environments, the only competitive libary we could find was LORAPO - a library that performs Cholesky factorization on block low-rank (BLR) matrices. Our code was implemented in C++ and CUDA, with only dependencies to dense linear algebra and MPI libraries. Experiments are carried out on the ABCI system at AIST in Japan, and

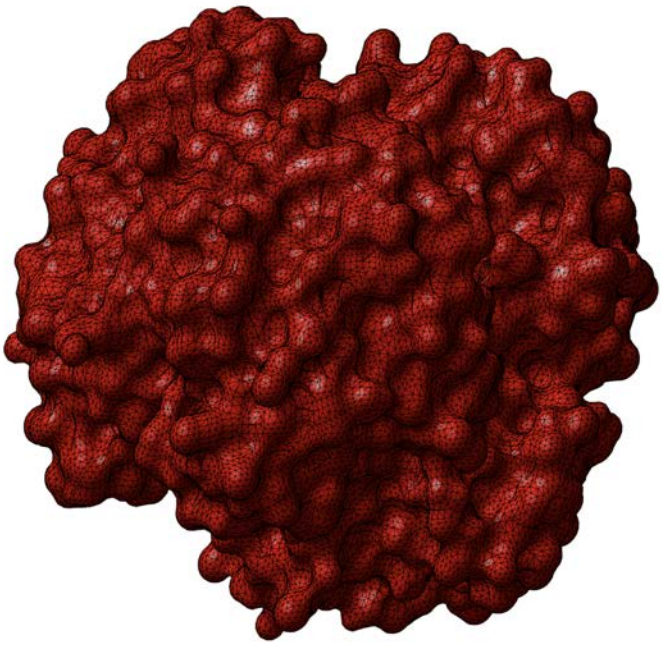


Figure 5.12: A single hemoglobin molecule

both our code and LORAPO have been compiled with GCC 9.3.0, CUDA 11.3, Open-MPI 4.1.3, Intel MKL 2022, and NCCL 2.9.

We organize the following sections into 2 parts; 1) the verification of the computational complexity and numerical accuracy, and 2) the parallel scalability. For the larger scale experiments, we used up to 128 of the compute V nodes on ABCI, featuring 2x Intel Xeon Gold 6148 Processor running at 2.4 GHz, 20 Cores (40 Threads), and 4x NVIDIA V100 16GiB HBM2 interconnected with NVLink within each node. The single-node experiments were performed on privately owned NVIDIA A100 GPUs and AMD EPYC 7502 CPUs.

5.5.1 Profiling Results

Before analyzing the numerical results and actual timings of the experiments, we present the output from a profiler, which shows the behavior of our proposed algorithm and what performance to expect. We used the profiler tools provided by NVIDIA to check the behavior as well as GPU utilization. Fig. 5.11 shows an annotated screenshot of only the factorization part running on 4 NVIDIA A100 GPUs for a matrix size of $N = 262,144$. The profiler results show that the 4 GPUs show high concurrency and utilization. The GPU utilization starts to fluctuate at higher levels of the tree, but remains high throughout the entire execution. The batched calls to POTRF, TRSM, and GEMM are shown using a colored bar only for GPU1, but the other GPUs are also calling the same libraries at similar timings.

5.5.2 Single Node Performance and Algorithmic Complexity

The computation time for the factorization and substitution phase of our $\mathcal{H}^2$-ULV method is shown in Fig. 5.13. The blue lines indicate the results on an AMD EPYC 7502 CPU and the red lines indicate the results on an NVIDIA A100 GPU. For these experiments, we used a dense matrix generated from a 3-D Laplace equation with points distributed on a spherical surface. This setup places the mesh points evenly on the spherical surface with roughly equal spacing for simplicity. The discrete solution to the 3-D Laplace equation is obtained via the Green's function of the Laplace operator, and the individual matrix entry can be obtained through the following kernel, where r_{ij} stands for the 3-D Euclidean distance between the i-th and j-th particle in the sorted list

$$A_{ij} = \begin{cases} 1.E^3, \text{if } i = j \\ \frac{1.}{r_{ij}}, \text{if } i \neq j \end{cases} \tag{5.35}$$

Our method shows the expected $\mathcal{O}(N)$ algorithmic complexity with respect to the matrix dimension N. For the CPU, we utilize the multiple cores by launching as many processes as there are cores, while for the GPU we launch a single process and call the batched libraries in cuBLAS/cuSOLVER. The substitution phase is considerably less compute-intensive and is memory-bound. Further, the substitution on the CPU does not use our inherently parallel algorithm, so the computational time

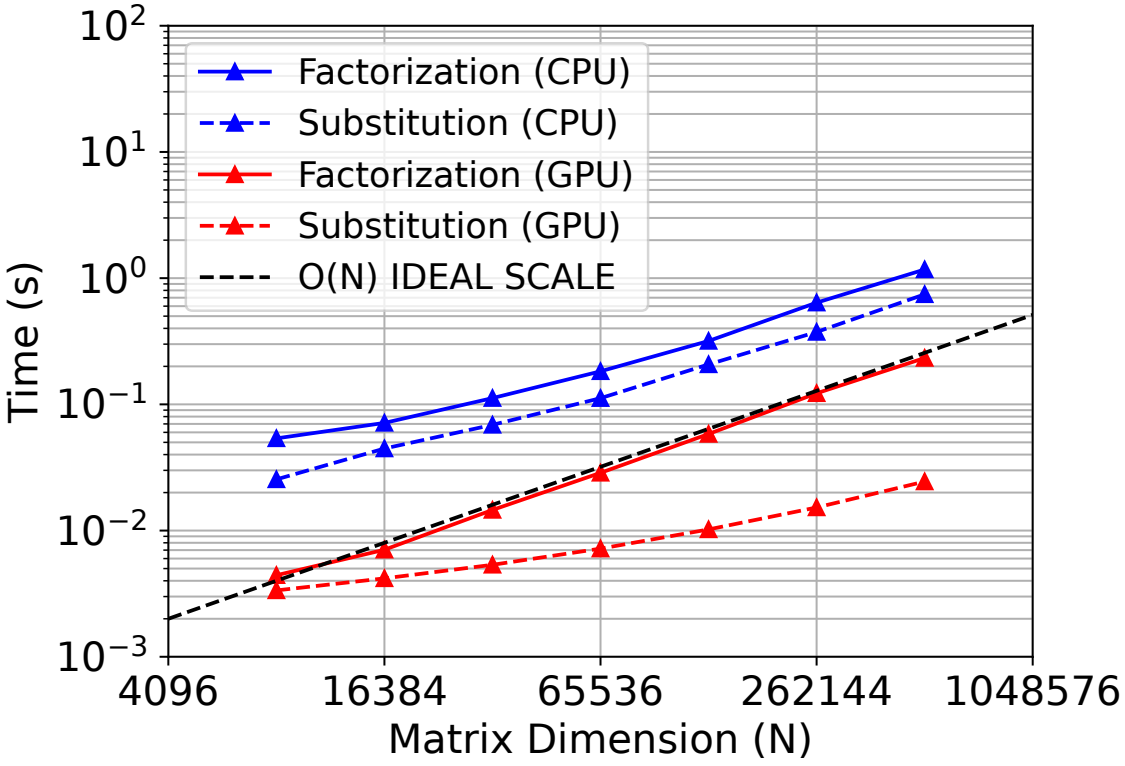


Figure 5.13: $\mathcal{O}(N)$ Factorization Time

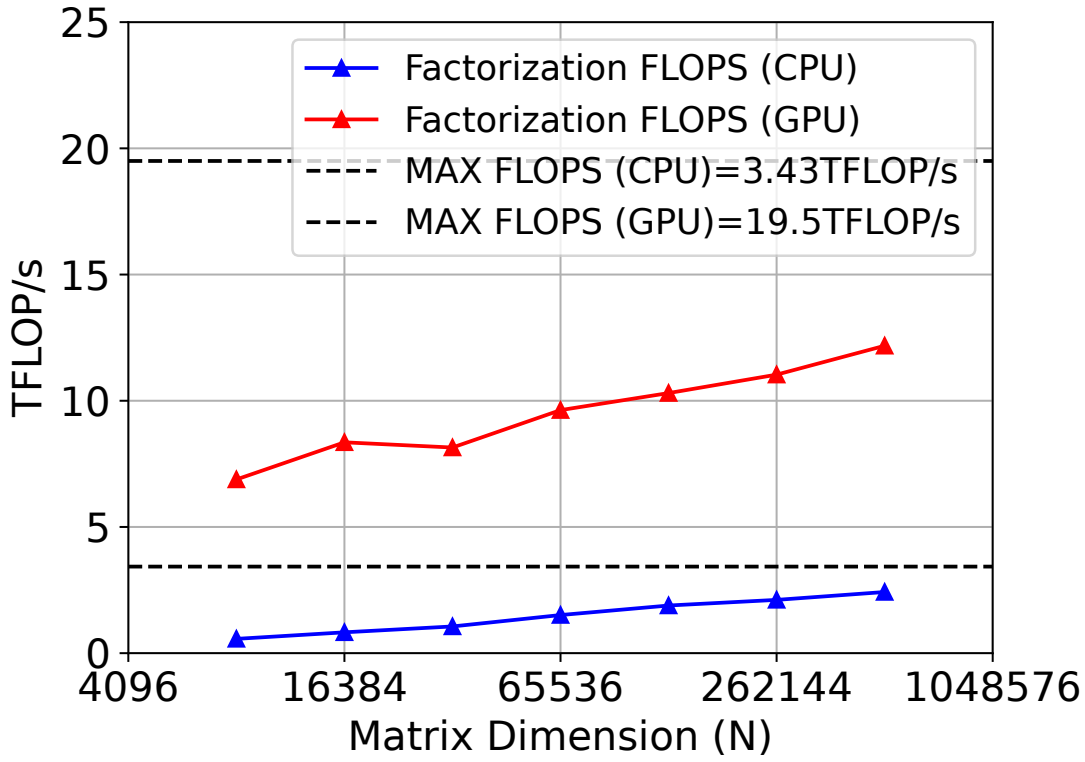


Figure 5.14: Matrix dimension vs. Factorization FLOPS performance

is close to that of the factorization despite the significantly smaller workload. The substitution on the GPU uses our inherently parallel algorithm and utilizes the batched kernels. The substitution time scales sub-linearly with respect to the increasing problem size, which shows that the algorithm is capable of utilizing more and more computing capabilities from the GPU as the parallel workload intensifies. For a large enough matrix size, the substitution on the GPU is an order of magnitude faster than the factorization. This would not have been possible without our inherently parallel substitution algorithm.

As a more precise indicator of the hardware utilization, we show the arithmetic throughput [TFLOP/s] for the factorization phase in Fig. 5.14. The highest achieved performance from the single node factorization experiments shows a maximum of 2.42TFLOP/s on CPU and 12.18TFLOP/s on GPU. These numbers correspond to 70.6% and 62.5% of the maximum achievable performance on this hardware, which is 3.43 TFLOP/s (on AVX2) and 19.5 TFLOP/s (on FP64 Tensor Cores) respectively.

As a supplement to the arithmetic complexity, we also counted the number of floating point operations that were performed in these experiments in Fig. 5.15. We plotted both the ideal $\mathcal{O}(N)$ and $\mathcal{O}(N \log_2 N)$ reference lines in the figure. The slope of our $\mathcal{H}^2$-ULV factorization seems to be

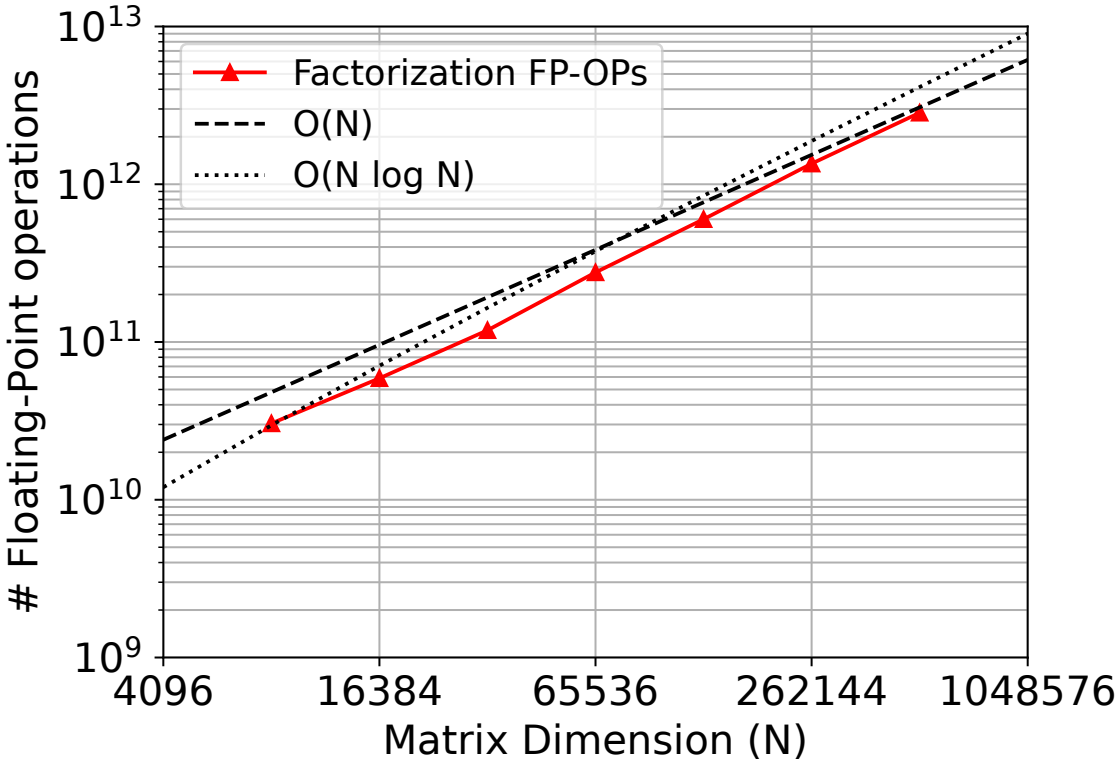


Figure 5.15: Matrix dimension vs. Factorization FLOPS count

somewhere in between $\mathcal{O}(N)$ and $\mathcal{O}(N \log_2 N)$, and not strictly $\mathcal{O}(N)$. The timing results in Fig. 5.13 obfuscate this fact, due to the under-utilization of hardware for smaller matrix sizes. In order to investigate this deviation from the ideal $\mathcal{O}(N)$ complexity, we looked at the number of dense matrix blocks N_{NZB} as a function of the number of blocks at the leaf level. The number of dense matrix blocks is equal to the number of neighbor interactions, which is what we show in Fig. 5.16. When both leaf size and the admissibility condition are fixed, $N_{NZB} = \mathcal{O}(N)$ that scales linearly with respect to the matrix size, and has a theoretical upper bound. When the number of leaf-level boxes is small, or in other words, the level of the tree is shallow, the actual number of neighboring interactions is much lower than the theoretical upper bound. This is the reason why the number of arithmetic operations in Fig. 5.15 shows a trend that is closer to $\mathcal{O}(N \log_2 N)$ instead of $\mathcal{O}(N)$. By extending the problem size and levels of the partitioning, we can observe that the complexity becomes closer to $\mathcal{O}(N)$ eventually.

In the last figure of this section Fig. 5.17, we compared the cost of the two separate factorization steps. In the experiments, we used the matrix size where $N = 262,144$ and adjusted the admissibility condition number from 0.0 (HSS admissibility) to 3.0 to produce $\mathcal{H}^2$-matrices with different numbers of off-diagonal dense blocks. The admissibility condition number is defined as the acceptable ratio of the maximum radius and the center distances of the two different boxes of particles. The comparison is made in the number of FP64 operations instead of providing the exact timings of either the pre-factorization or the $\mathcal{H}^2$-matrix construction. The pre-factorization does not require a specific selection of the low-rank approximation method, and such difference in $\mathcal{H}^2$-matrix approximation can create a significant difference in the achievable accuracy and performance. It is also worth noting that the factorization costs, both the pre-factorization and the actual factorization, do not scale linearly with respect to the admissibility condition. In other words, the linear complexity experiments need to keep the admissibility condition number constant. However, this factorization cost comparison result suggests that as the $\mathcal{H}^2$-matrices can have more off-diagonal dense blocks, the pre-factorization does not cost more than 46% of the total cost, as well as scaling linearly just as the actual factorization.

5.5.3 Matrix Rank and Solution Accuracy

There are existing parallel implementations of the HSS-ULV factorization, such as STRUMPACK and H2pack. As a justification for extending this to $\mathcal{H}^2$-matrix format with stronger admissibility conditions at the cost of increased code complexity, we compare the solution accuracy from the different admissibility conditions, as well as the compute time and the number of floating point operations being used to obtain the answers. In these experiments, we used the same code but different runtime parameters for the admissibility condition for computing in the HSS and $\mathcal{H}^2$ matrix formats. The HSS matrix is a subset of the more general $\mathcal{H}^2$ matrix, and can be configured as an

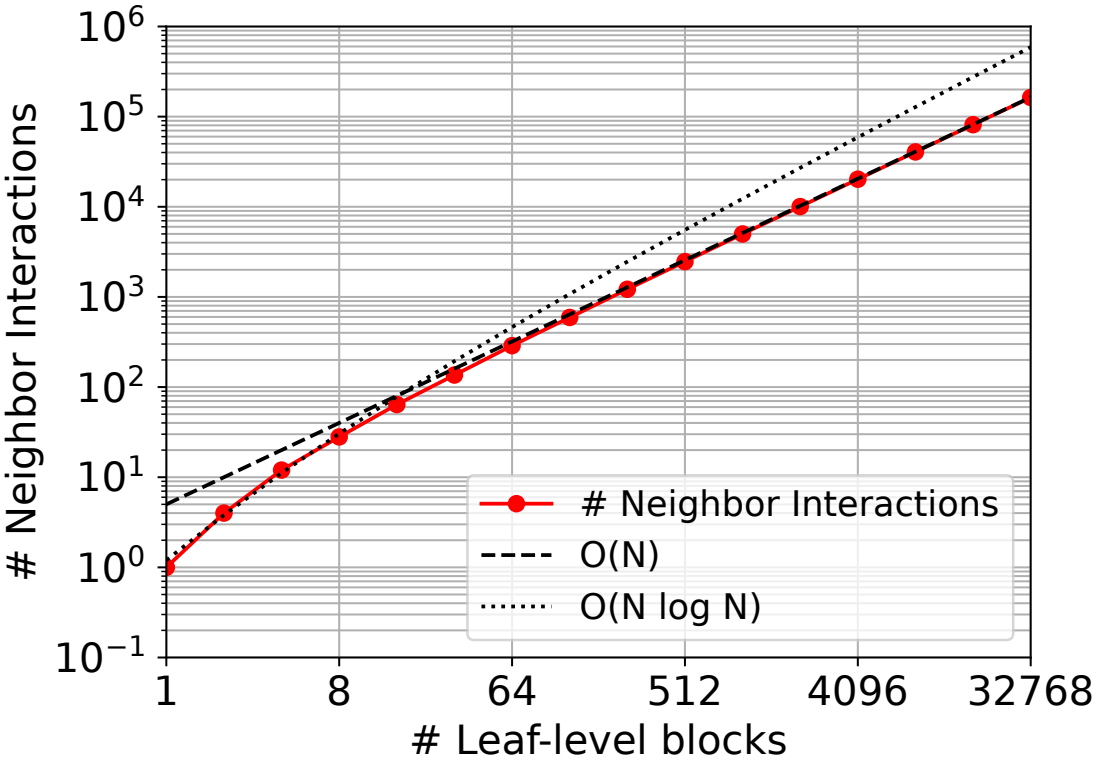


Figure 5.16: Number of neighboring interactions

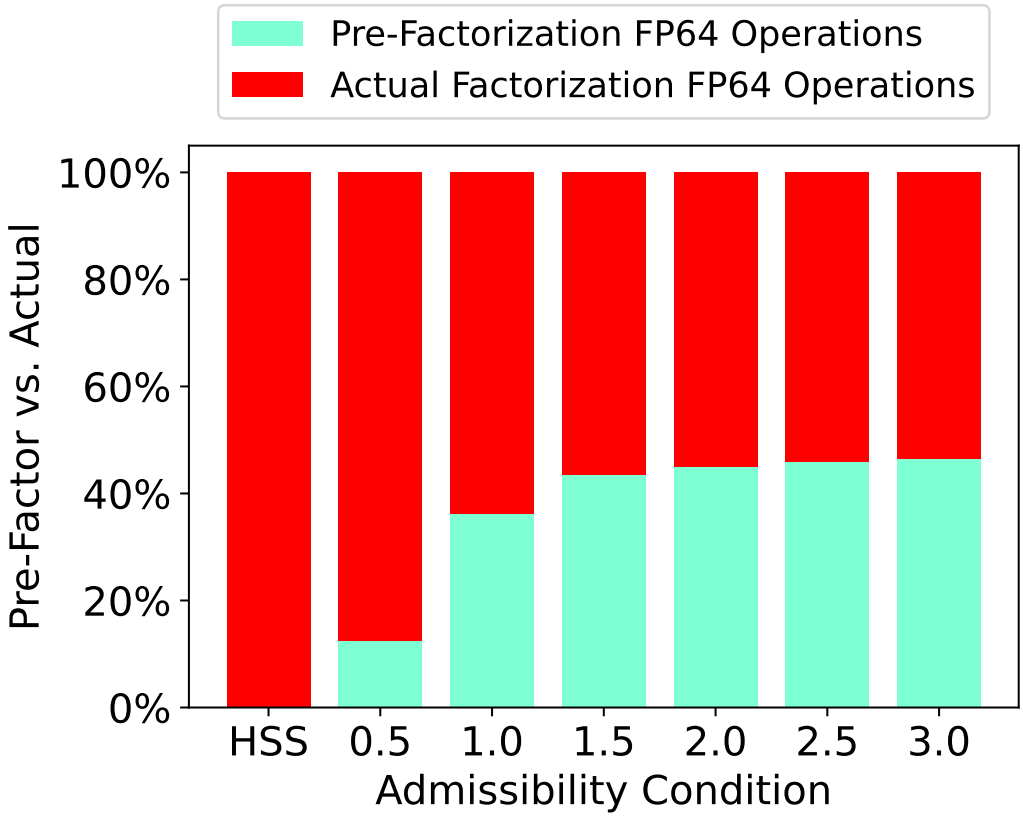


Figure 5.17: Factorization FLOPS distribution vs. Admissibility condition

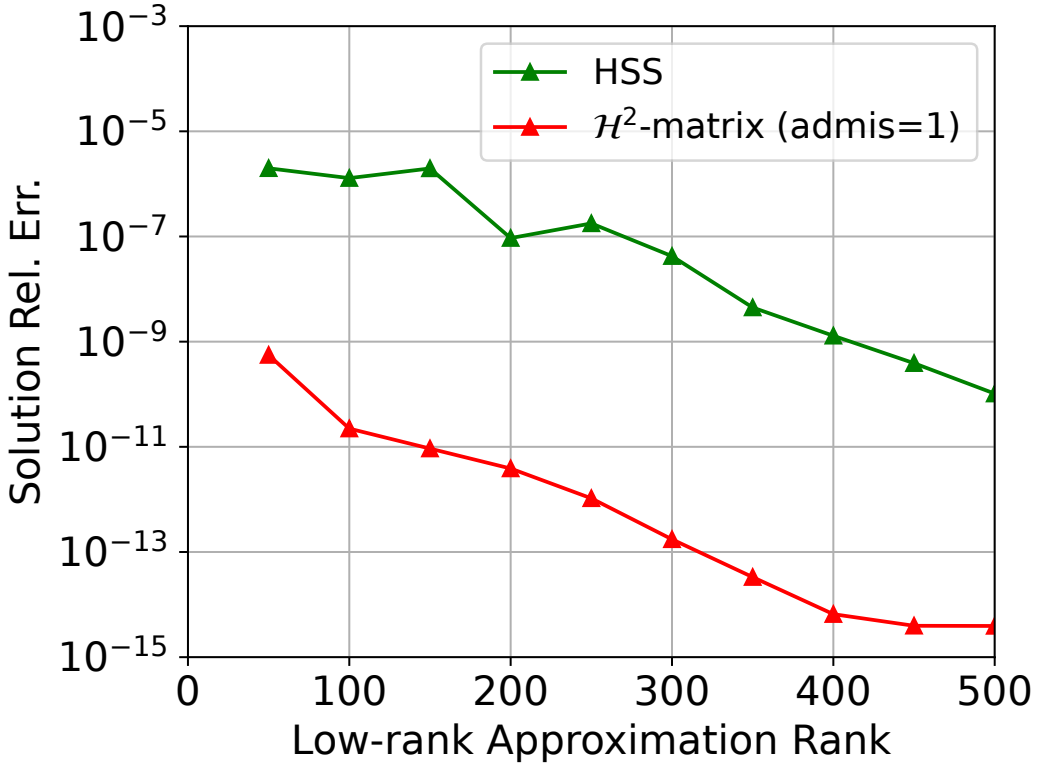


Figure 5.18: Low-rank Approximation rank vs. Solution Accuracy

$\mathcal{H}^2$-matrix where all of the off-diagonal blocks are low-rank matrices. Although we are fully aware of other existing HSS-matrix libraries that can run on GPUs, we used our implementation for this comparison to avoid contaminating the results with other differences such as the method of low-rank approximation and the level of optimization.

We used a matrix of size $N = 8192$ and $Leaf = 512$ and configured the matrix construction to use a fixed-rank truncation on all of the available far-field particles (far-field sampling disabled, leading to $\mathcal{O}(N^2)$ cost to construct), to give the best low-rank approximation results. Fig. 5.18 shows the result from our experiments, which suggests that in order to achieve the same solution accuracy that of the $\mathcal{H}^2$-matrix at rank 50, the HSS matrix will require a rank of more than 400. Despite that our method of constructing the $\mathcal{H}^2$-matrix includes both the content from the far-field (low-rank basis) and the neighboring boxes (factorization basis), our method gives considerably better approximation results with small matrix ranks. As the HSS uses very high ranks to achieve the same factorization accuracy, it quickly exhausts the available memory and also results in much slower time-to-solution as can be seen from Fig. 5.19.

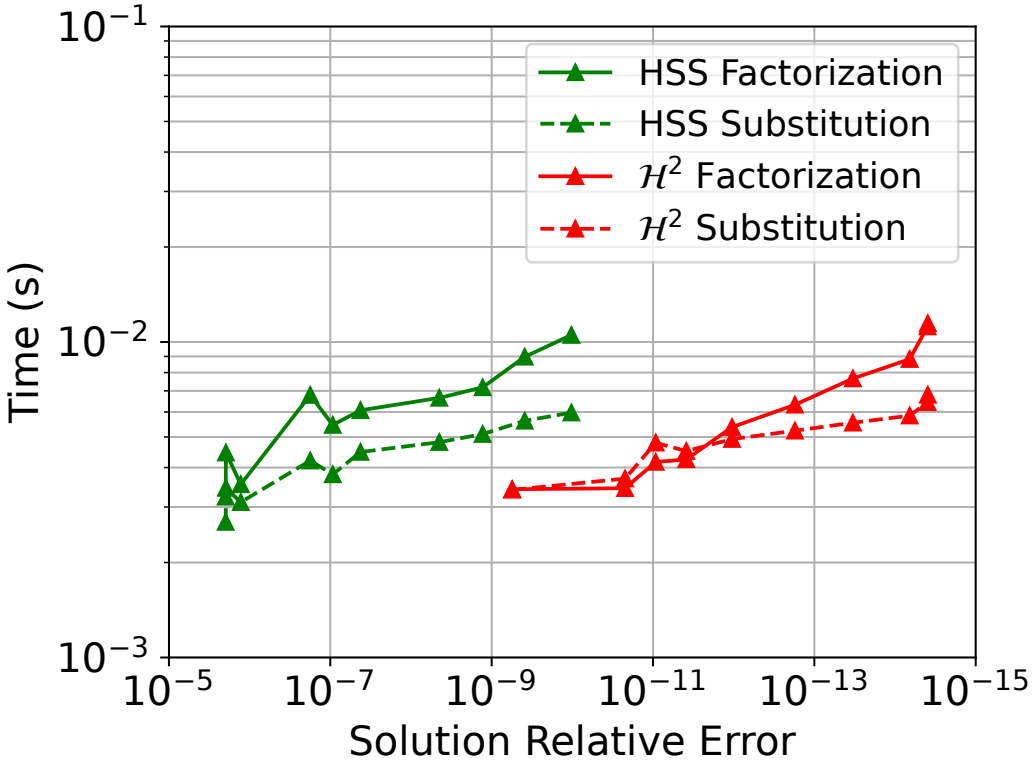


Figure 5.19: Solution Accuracy vs. Time to Solution

5.5.4 Parallel Scalability

We now show the parallel scalability results of our $\mathcal{H}^2$-ULV factorization on up to 512 CPU/GPUs on the ABCI system with a maximum problem size of $N = 29,242,368$. In this experiment setup, we compute the Yukawa potential for the boundary mesh points placed on the surface of hemoglobin molecules. The Yukawa potential is similar to Green's function of the Laplace equation, in that the potential at $r = 0$ is a singular point of infinite potential. A simplified Yukawa potential can be computed from the following kernel function by setting all of the constants to 1

$$A_{ij} = \begin{cases} 1.E^3, \text{if } i = j \\ \frac{e^{-r_{ij}}}{r_{ij}}, \text{if } i \neq j \end{cases} \tag{5.36}$$

The molecule geometries data contains 14908 and 57114 mesh points for two differently shaped hemoglobin molecules, and at most 512 duplicates of the same molecule are placed in the same domain. By reading the portions of the geometry of the molecules, we create variations in the problem sizes.

The strong scaling result is presented in Fig. 5.20, where we compare with a BLR-Cholesky factorization implementation in LORAPO. Since the BLR matrix format has a factorization complexity of $\mathcal{O}(N^2)$, we are not able to run LORAPO up to the same problem sizes that we are capable of solving with our method. Our code is executed on the GPU while LORAPO is executed on the CPU. Our code exhibits good strong scaling up to 64 GPUs, but struggled to get close to the ideal scaling past 128 GPUs. This is because the local problem size becomes too small to saturate the performance of the GPUs in such strong scaling experiments. The best performance in the strong scaling experiments showed a speedup of over 13,300x in factorization time over LORAPO using the same number of 128 sockets of CPUs/GPUs.

We also further conducted weak scaling experiments to demonstrate our GPU implementation's parallel scalability. Instead of having an ideal constant computing time, our code is expected to have a $\mathcal{O}(\log_2 P)$ compute complexity with a varying problem size where $N = \mathcal{O}(P)$, and P stands for the number of computing processes (resources). This $\mathcal{O}(\log_2 P)$ complexity results from redundant computation at the levels that are close to the root. From levels $\log_2 P$ to level 1, each computing level requires $\mathcal{O}(1)$ work. We present the weak scaling result for the factorization phase in Fig. 5.21, and for the substitution phase in Fig. 5.22. For a maximum problem size of $N = 29,242,368$ and using 512 GPUs, we can complete the factorization using only 0.25s and the substitution using 0.83s, utilizing 0.808 PFLOP/s (1.579TFLOP/s per GPU) for factorization. The factorization has high arithmetic intensity and does not need communication until levels that are close to the root. Therefore, it shows almost ideal scaling with respect to the theoretical complexity of $\mathcal{O}(\log_2 P)$. The substitution is memory and communication bound and requires a significant amount of neighbor-to-neighbor communication even at the leaf level. In Fig. 5.22, we plotted both the $\mathcal{O}(\log_2 P)$ and

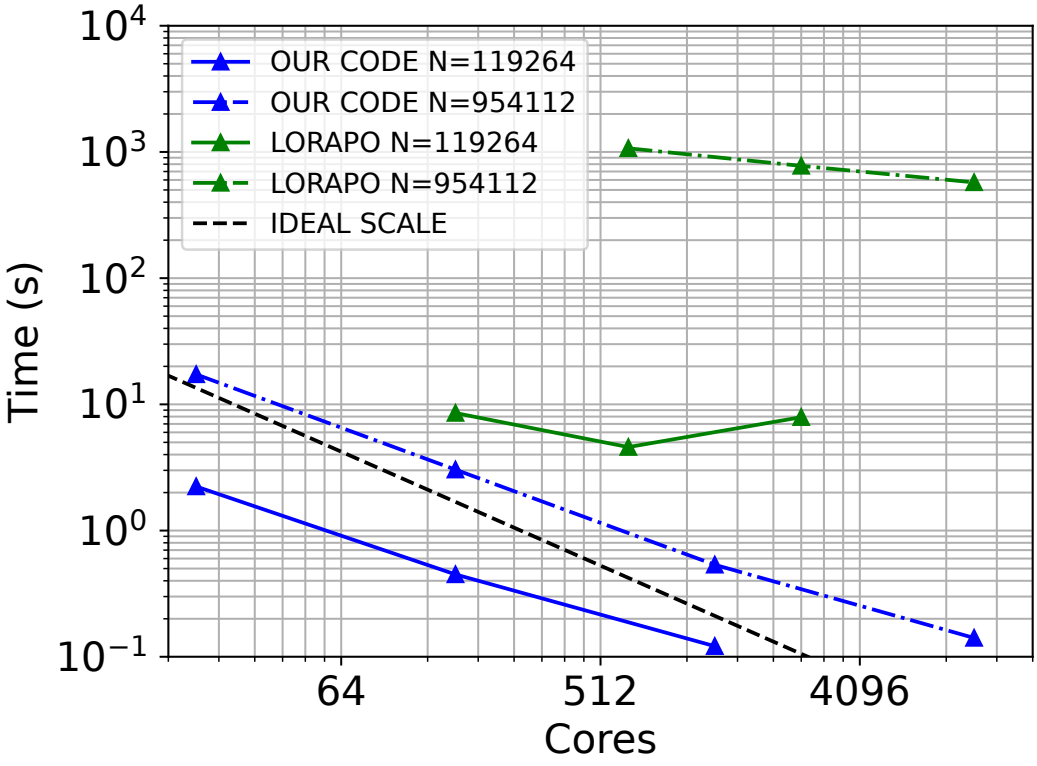


Figure 5.20: Factorization Time vs. Number of CPU/GPUs (Strong Scaling

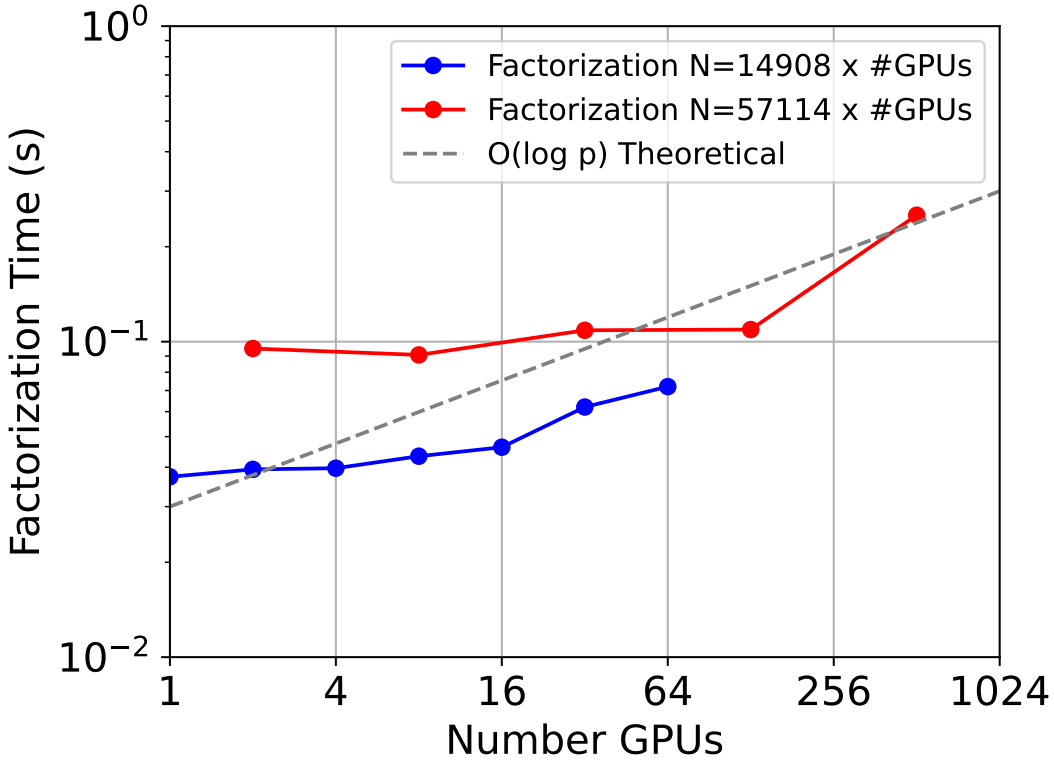


Figure 5.21: Factorization Time with increasing Computing Resource and Problem Size(Weak Scaling)

$\mathcal{O}(P)$ scaling lines for reference. As the neighbor-to-neighbor communication intensifies between the nodes, the substitution time grows as $\mathcal{O}(P)$, which is expected. By using a large number of GPUs, the substitution time shows the ideal scaling of $\mathcal{O}(\log_2 P)$.

In the weak scaling experiments, the detailed timing breakdown is provided in Fig. 5.23 for experiments ranging from $N = 114,228$ to $N = 29,242,368$. Throughout the different experiment configurations, the factorization kernel time maintained high occupancy (more than 60%) due to its high numeric intensity. For the forward and backward substitution, due to operating on a $width = 1$ vector, the total numbers of FP64 operations are significantly lower compared to the factorization, and the communication costs become dominating as the number of processing units increases. However, since the communication cost has the same complexity as $\mathcal{O}(\log_2 P)$, which is the same as both the factorization and the substitution cost, we can observe that at the eventual point where using 512 GPUs the substitution kernel time ratio does not drop lower comparing to the previous data point of 128 GPUs. From this we conclude, that despite having a high cost for communication, the proposed method maintained its scalability and is expected to scale beyond the parallel resources available to us.

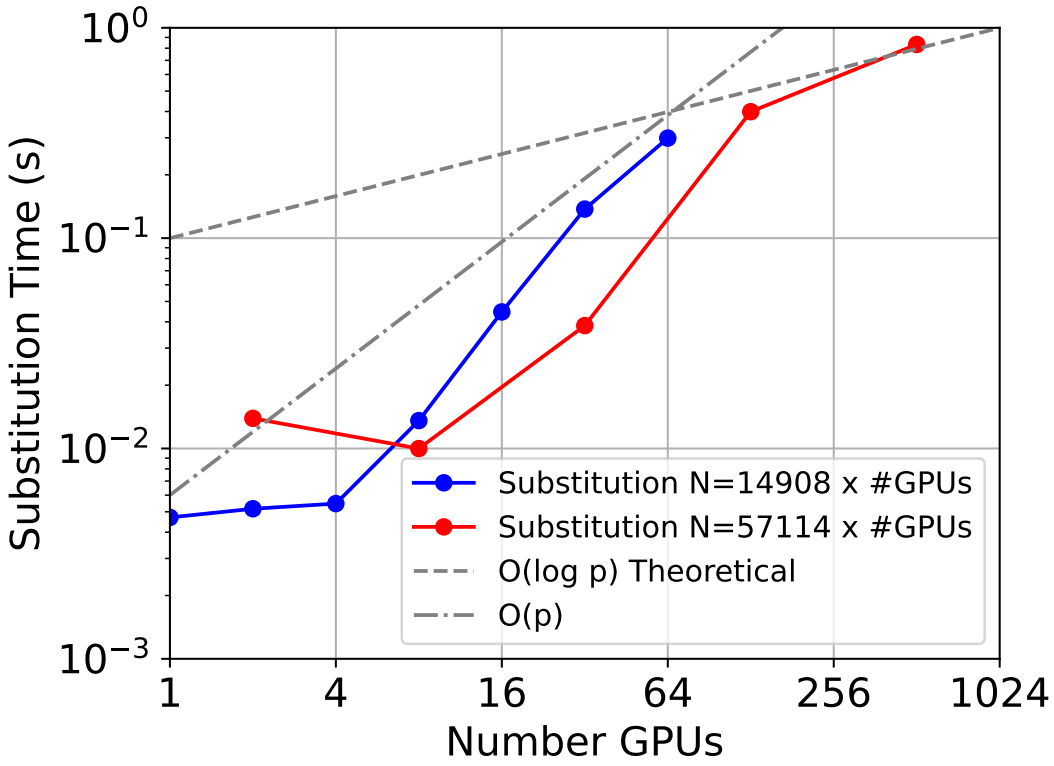


Figure 5.22: Substitution Time with increasing Computing Resource and Problem Size(Weak Scaling)

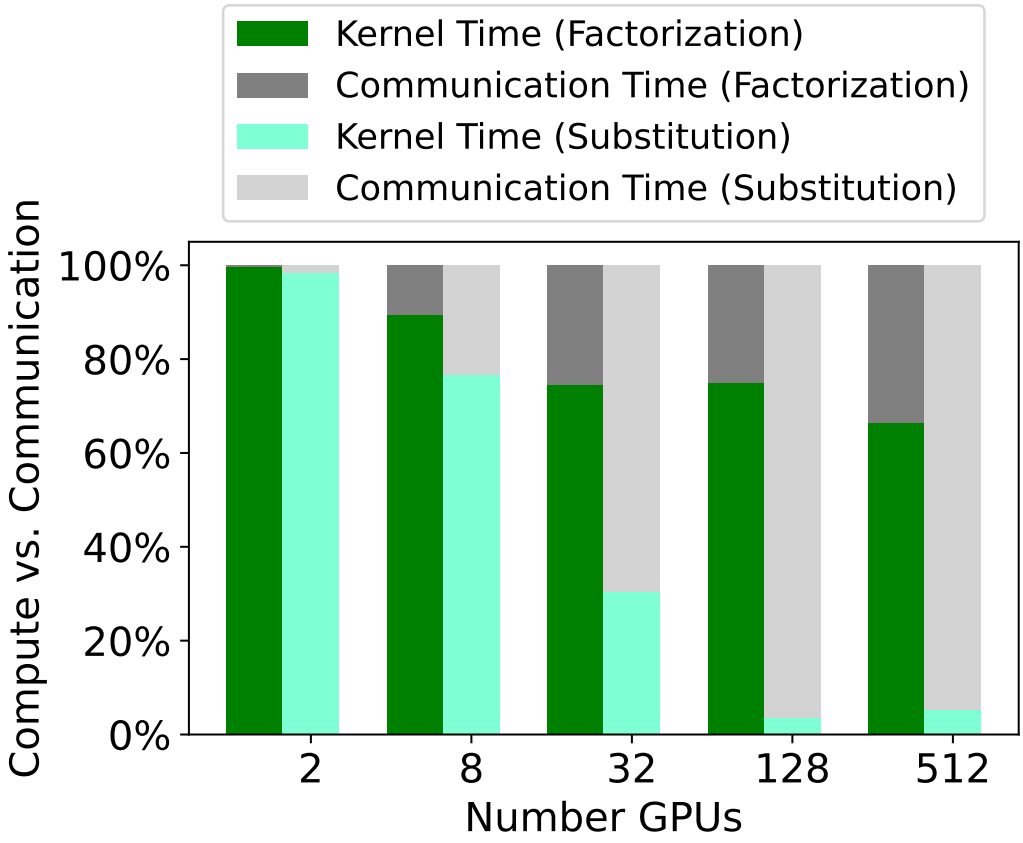


Figure 5.23: Timings percentage breakdown for Compute versus Communication in Weak Scaling

Chapter 6

Conclusion

This thesis has visited the basics of the low-rank approximation and the formulation of the structured low-rank matrices. Moving from the simpler independent basis and single-leveled structures, this thesis shifted the focus towards the use of shared-basis and the ULV factorization on the optimal $\mathcal{H}^2$-matrices to build a rivaling approach that can be as parallel as the direct factorization of HSS matrix.

As a summary of the thesis, we conclude the work we have commenced in the two aspects, linear algebra algorithms and HPC environments. Firstly, we have extended the existing practices of the ULV factorization algorithm for $\mathcal{H}^2$-matrix to decouple the re-compression computations with the actual factorization computations. By adding to the compressed blocks to consider all Schur's complement computations, we have made the actual factorization inherently parallel to the ULV factorization applied to the HSS matrices. In the decoupled re-compression phase, as we moved it to happen before the factorization, we renamed it to pre-compression. The pre-compression includes another separate factorization step where cost is analyzed to maintain $\mathcal{O}(N)$ and made to be parallelizable. Using the improved algorithm alone, we have shown a speedup of over 4,700x over the simpler structured BLR matrix.

In the HPC aspects, we have advanced the usage of structured low-rank matrices to run on the latest supercomputers. Before our work, the parallelization in the direct factorization implementations in this area is limited to the single-level BLR matrices and HSS matrices to run on largely distributed machines. Our work extended the $\mathcal{H}^2$-matrix to the distributed memory environment without assistance from the runtime systems, as we did not see a need after breaking the chaining data dependencies. Our GPU implementation included parallel ULV factorization and forward and backward substitutions on a distributed memory environment. Using the batched interfaces from cuBLAS and cuSOLVER, we have extracted more than 1.5 TFLOPS per GPU and have over 800 TFLOPS in performance on 512 NVIDIA V100s.

6.1 Future work

The exploration of the structured low-rank matrices and low-precision arithmetics in linear algebra is constantly advancing. The work presented in the thesis despite having advantages in the parallelism introduced, also has its own weaknesses such as the vulnerability to ill problem conditioning, and the utilization of two explicit factorizations, which is still costly in floating point operations. Therefore, we also listed some promising and prospective fields of the futuristic areas that could be based on this work.

- The combination of the current work and the iterative methods to obtain the solution to linear systems. The matrix factorization cost and timings are strongly connected to the desired accuracy, which relates to the corresponding ranks, admissibility conditions, and the internal floating point number representations. Although not presented in the current work, we conducted preliminary experiments suggesting that combining a low-precision factorization as a preconditioner in the iterative methods for obtaining solutions to the linear systems, such as iterative refinement or Krylov subspace methods, can be faster than using a high-precision factorization alone.

- Extending on the reliability of the work, as well as supporting the sparse matrix methods. As the publications have suggested, as well as the LoRaSp project[23] that employs the method of inverse FMM, the ULV factorization of the $\mathcal{H}^2$-matrix can naturally support a sparse matrix input. We did not compare with sparse matrix solvers in this work, as we have not yet carefully considered directly constructing a structured low-rank matrix from a sparse grid or geometry. We expect challenges by extending into the area of finding solutions for sparse matrices, as there are already well-developed and implemented approaches such as multigrid, incomplete factorization pre-conditioned iterative methods, and the fill-reduction reorderings.

- The search for more parallelism for structured low-rank matrices with independent basis. The independent basis $\mathcal{H}$-matrices have advantages, such as a pure algebraic compression and no prior knowledge required from FMM or calculus, but have fallen behind in the methods to find direct solutions in parallel. There have been many efforts for the $\mathcal{H}$-matrix LU factorization via different ways of optimizations, but not until recently, did the method that adopts the extended sparsification make the weakly admissible configured HODLR matrix have more parallel capabilities. We also believe there could be more parallelism to the structured low-rank matrices that use an independent basis, which might not be a direct mean or accurate factorization. We are still in active searches for such methods.

Appendices

Appendix A

Publication list

A.1 Included in this thesis

- Qianxiang Ma and Rio Yokota. An inherently parallel $\mathcal{H}^2$-ULV factorization for solving dense linear systems on GPUs. *The International Journal of High Performance Computing Applications*. 2024;0(0). doi:10.1177/10943420241242021

- Qianxiang Ma, Sameer Deshmukh, and Rio 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*, pages 1198–1209, Dallas, TX, USA, November 2022. IEEE Computer Society. ISBN 978-1-66545-444-5. URL https://www.computer.org/csdl/proceedings-article/sc/2022/544400b198/1I0bTeDr9fy. ISSN: 2167-4337.

A.2 Not included in this thesis

- Qianxiang Ma, Rio Yokota. Poster 48: Runtime System for GPU-based Hierarchical LU factorization. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC19, November 17–22, 2019, Colorado Convention Center, Denver, Colorado, USA. https://sc19.supercomputing.org/proceedings/tech_poster/tech_poster_pages/rpost128.html

Bibliography

[1] Sameh Abdulah, Qinglei Cao, Yu Pei, George Bosilca, Jack Dongarra, Marc M. Genton, David Keyes, Hatem Ltaief, and Ying Sun. Accelerating Geostatistical Modeling and Prediction With Mixed-Precision Computations: A High-Productivity Approach with PaRSEC. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2021. ISSN 1045-9219, 1558-2183, 2161-9883. doi: 10.1109/TPDS.2021.3084071. URL `https://ieeexplore.ieee.org/document/9442267/`.

[2] K. Akbudak, H. Ltaief, A. Mikhalev, and D. Keyes. Tile Low Rank Cholesky Factorization for Climate/Weather Modeling Applications on Manycore Architectures. In *ISC High Performance 2017, LNCS*, volume 10266, pages 22–40, Frankfurt, Germany, 2017. Springer, Cham. ISBN 978-3-319-58666-3. doi: 10.1007/978-3-319-58667-02.

[3] S. Ambikasaran. *Fast Algorithms for Dense Numerical Linear Algebra and Applications.* PhD Thesis, Stanford University, 2013.

[4] S. Ambikasaran and E. Darve. The Inverse Fast Multipole Method. *arXiv:1407.1572v1*, 2014.

[5] P. Amestoy, C. Ashcraft, O. Boiteau, A. Buttari, J.-Y. L'Excellent, and C. Weisbecker. Improving Multifrontal Methods by Means of Block Low-Rank Representations. *SIAM Journal on Scientific Computing*, 37(3):A1451–A1474, 2015. Number: 3.

[6] 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*, 45(1):1–26, March 2019. ISSN 0098-3500, 1557-7295. doi: 10.1145/3242094. URL `https://dl.acm.org/doi/10.1145/3242094`. Number: 1.

[7] Cleve Ashcraft, Alfredo Buttari, and Theo Mary. Block Low-Rank Matrices with Shared Bases: Potential and Limitations of the BLR$ˆ2$ Format. *SIAM Journal on Matrix Analysis and Applications*, 42(2):990–1010, January 2021. ISSN 0895-4798, 1095-7162. doi: 10.1137/20M1386451. URL `https://epubs.siam.org/doi/10.1137/20M1386451`.

[8] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: a Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23:187–198, 2011. doi: 10.1002/cpe.1631.

[9] Peter Benner and Thomas Mach. Computing All or Some Eigenvalues of Symmetric Hl-Matrices. *SIAM Journal on Scientific Computing*, 34(1):A485–A496, 2012. doi: 10.1137/100815323. URL https://doi.org/10.1137/100815323.

[10] L. Blackford, J. Choi, A. Cleary, J. Demmel, Inderjit S. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[11] S. Borm. H^2-Matrix Arithmetics in Linear Complexity. *Computing*, 77:1–28, 2006. URL http://search.proquest.com/openview/7652d5a0ccaaccd1ac5bc10f7db0c0f5/1?pq-origsite=gscholar.

[12] S. Borm. Hierarchical Matrix Arithmetic with Accumulated Updates. *Computing and Visualization in Science*, pages 1–14, 2019. doi: 10.1007/s00791-019-00311-3.

[13] Steffen Borm, Maria Lopez-Fernandez, and Stefan Sauter. Variable Order, Directional $\mathcal{H}^2$-Matrices for Helmholtz Problems with Complex Frequency. *arXiv:1903.02803 [math]*, March 2019. URL http://arxiv.org/abs/1903.02803. arXiv: 1903.02803.

[14] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed DAG engine for High Performance Computing. *Parallel Computing*, 38(1-2):37–51, January 2012. ISSN 01678191. doi: 10.1016/j.parco.2011.10.003. URL https://linkinghub.elsevier.com/retrieve/pii/S0167819111001347.

[15] Difeng Cai, Hua Huang, Edmond Chow, and Yuanzhe Xi. Data-driven construction of hierarchical matrices with nested bases. *SIAM Journal on Scientific Computing*, 46(2):S24–S50, 2024. doi: 10.1137/22M1500848. URL https://doi.org/10.1137/22M1500848.

[16] Léopold Cambier and Eric Darve. A task-based distributed parallel sparsified nested dissection algorithm. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, pages 1–11, Geneva Switzerland, July 2021. ACM. ISBN 978-1-4503-8563-3. doi: 10.1145/3468267.3470619. URL https://dl.acm.org/doi/10.1145/3468267.3470619. cambierTaskBasedDistributed2021.

[17] Qinglei Cao, Yu Pei, Kadir Akbudak, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. Leveraging PaRSEC Runtime Support to Tackle Challenging 3D Data-Sparse Matrix Problems. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*,

pages 79–89, Portland, OR, USA, May 2021. IEEE. ISBN 978-1-66544-066-0. doi: 10.1109/IPDPS49936.2021.00017. URL https://ieeexplore.ieee.org/document/9460493/.

[18] Qinglei Cao, Sameh Abdulah, Rabab Alomairy, Yu Pei, Pratik Nag, George Bosilca, Jack Dongarra, Marc G. Genton, David E. Keyes, Hatem Ltaief, and Ying 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*, SC '22, pages 1–12, Dallas, Texas, November 2022. IEEE Press.

[19] Quinglei Cao, Yu Pei, Thomas Herauldt, Kadir Akbudak, Aleksandr Mikhalev, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. Performance Analysis of Tile Low-Rank Cholesky Factorization Using PaRSEC Instrumentation Tools. In *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*, pages 25–32, November 2019. doi: 10.1109/ProTools49597.2019.00009. URL https://ieeexplore.ieee.org/document/8955679. ISSN: null.

[20] S. Chandrasekaran, P. Dewilde, M. Gu, W. Lyons, and T. Pals. A Fast Solver for HSS Representations via Sparse Matrices etc. *SIAM Journal on Matrix Analysis and Applications*, 29(1): 67–81, May 2006. doi: 10.1137/050639028. Number: 1.

[21] Chao Chen and Per-Gunnar Martinsson. Solving linear systems on a gpu with hierarchically off-diagonal low-rank approximations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022. ISBN 9784665454445.

[22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844.

[23] 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*, 39(3):A761–A796, 2017. doi: 10.1137/15M1034477. Number: 3.

[24] James Demmel, Nicholas Higham, and Robert Schreiber. Block lu factorization. *Numerical Linear Algebra with Applications*, 1995.

[25] Jack Dongarra, Mark Gates, Azzam Haidar, Jakub Kurzak, Piotr Luszczek, Stanimire Tomov, and Ichitaro Yamazaki. Accelerating numerical dense linear algebra calculations with gpus. *Numerical Computations with GPUs*, pages 1–26, 2014.

[26] W. Hackbusch. A Sparse Matrix Arithmetic Based on H-Matrices, Part I: Introduction to H-Matrices. *Computing*, 62:89–108, 1999. doi: 10.1007/s006070050015.

[27] W. Hackbusch. *Hierarchical Matrices: Algorithms and Analysis*, volume 49 of *Springer Series in Computational Mathematics*. Springer Berlin Heidelberg, 2015. doi: 10.1007/978-3-662-47324-5.

[28] W. Hackbusch, B. Khoromskij, and S. A. Sauter. On H^2-Matrices. In H. Bungartz, R. Hoppe, and C. Zenger, editors, *Lectures on Applied Mathematics*. Springer Berlin Heidelberg, 2000. doi: 10.1007/978-3-642-59709-1_2.

[29] Helmut Harbrecht and Peter Zaspel. A Scalable {H}-Matrix Approach for the Solution of Boundary Integral Equations on Multi-{GPU} Clusters. *arXiv:1806.11558 [cs, math]*, June 2018. URL `http://arxiv.org/abs/1806.11558`. arXiv: 1806.11558.

[30] K. L. Ho and L. Ying. Hierarchical Interpolative Factorization for Elliptic Operators: Differential Equations. *Communications on Pure and Applied Mathematics*, 69(8):1415–1451, 2016. Number: 8.

[31] Hua Huang, Xin Xing, and Edmond Chow. H2Pack: High-performance H2 Matrix Package for Kernel Matrices Using the Proxy Point Method. *ACM Transactions on Mathematical Software*, 47(1):1–29, January 2021. ISSN 0098-3500, 1557-7295. doi: 10.1145/3412850. URL `https://dl.acm.org/doi/10.1145/3412850`.

[32] Akihiro Ida. Lattice H-Matrices on Distributed-Memory Systems. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 389–398, Vancouver, BC, May 2018. IEEE. ISBN 978-1-5386-4368-6. doi: 10.1109/IPDPS.2018.00049. URL `https://ieeexplore.ieee.org/document/8425193/`.

[33] Akihiro Ida. Solving block low-rank matrix eigenvalue problems. *Journal of Information Processing*, 30:538–551, 2022. doi: 10.2197/ipsjjip.30.538.

[34] Akihiro Ida, Hiroshi Nakashima, Tasuku Hiraishi, Ichitaro Yamazaki, Rio Yokota, and Takeshi Iwashita. QR Factorization of Block Low-rank Matrices with Weak Admissibility Condition. *Journal of Information Processing*, 27:831–839, 2019. doi: 10.2197/ipsjjip.27.831.

[35] Akihiro Ida, Tadashi Ataka, and Atsushi Furuya. Lattice H-Matrices for Massively Parallel Micromagnetic Simulations of Current-Induced Domain Wall Motion. *IEEE Transactions on Magnetics*, 56(4):1–4, April 2020. ISSN 1941-0069. doi: 10.1109/TMAG.2019.2959349. Conference Name: IEEE Transactions on Magnetics.

[36] R. Kriemann. H-LU Factorization on Many-Core Systems. In *Computing and Visualization in Science*. Springer Berlin Heidelberg, 2014.

[37] A. Litvinenko, Y. Sun, M. G. Genton, and D. E. Keyes. Likelihood Approximation with Hierarchical Matrices for Large Spatial Datasets. *Computational Statistics and Data Analysis*, 137: 115–132, 2019. doi: 10.1016/j.csda.2019.02.002.

[38] Yang Liu, Pieter Ghysels, Lisa Claus, and Xiaoye Sherry Li. Sparse Approximate Multifrontal Factorization with Butterfly Compression for High Frequency Wave Equations. *SIAM Journal on Scientific Computing*, 43(5):S367–S391, 2021. doi: https://doi.org/10.1137/20M1349667. URL `http://arxiv.org/abs/2007.00202`. arXiv: 2007.00202.

[39] M. Ma and D. Jiao. Accuracy Directly Controlled Fast Direct Solution of General Hˆ2-matrices and Its Application to Solving Electrodynamics Volume Integral Equations. *IEEE Transactions on Microwave Theory and Techniques*, 66(1):35–48, 2018. doi: 10.1109/TMTT.2017.2734090. tex.ids= ma2018a number: 1.

[40] Miaomiao Ma and Dan Jiao. Direct Solution of General H2 -Matrices With Controlled Accuracy and Concurrent Change of Cluster Bases for Electromagnetic Analysis. *IEEE Transactions on Microwave Theory and Techniques*, 67(6):2114–2127, June 2019. ISSN 1557-9670. doi: 10.1109/TMTT.2019.2914391. Conference Name: IEEE Transactions on Microwave Theory and Techniques.

[41] Qianxiang Ma, Sameer Deshmukh, and Rio 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*, pages 1198–1209, Dallas, TX, USA, November 2022. IEEE Computer Society. ISBN 978-1-66545-444-5. URL `https://www.computer.org/csdl/proceedings-article/sc/2022/544400b198/1I0bTeDr9fy`. ISSN: 2167-4337.

[42] P. G. Martinsson, V. Rokhlin, and M. Tygert. A Randomized Algorithm for the Decomposition of Matrices. *Applied and Computational Harmonic Analysis*, 30:47–68, 2011.

[43] NVIDIA. Nvidia cutlass, May 2017. URL `https://github.com/NVIDIA/cutlass/releases`.

[44] NVIDIA. Nvidia cuda toolkit: cublas, July 2020. URL `https://developer.nvidia.com/cublas`.

[45] Qinglei Cao, Yu Pei, Kadir Akbudak, Aleksandr Mikhalev, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. Extreme-Scale Task-Based Cholesky Factorization Toward Climate and Weather Prediction Applications. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '20, pages 1–11, New York, NY, USA, June 2020. Association for Computing Machinery. ISBN 978-1-4503-7993-9. doi: 10.1145/3394277.3401846. URL `https://doi.org/10.1145/3394277.3401846`.

[46] F.-H. Rouet, X. S. Li, P. Ghysels, and A. Napov. A Distributed-Memory Package for Dense Hierarchically Semi-Seperable Matrix Computations Using Randomization. *ACM Transactions on Mathematical Software*, 42(4):Article 27, 2016. Number: 4.

[47] François-Henry Rouet, Xiaoye S. Li, Pieter Ghysels, and Artem Napov. A distributed-memory package for dense Hierarchically Semi-Separable matrix computations using randomization. *arXiv:1503.05464 [cs]*, 42(4):Article No.: 27, pages 1–35, June 2015. URL `http://arxiv.org/abs/1503.05464`. arXiv: 1503.05464.

[48] Toru Takahashi, Chao Chen, and Eric Darve. Parallelization of the Inverse Fast Multipole Method with an Application to Boundary Element Method. *Computer Physics Communications*, 247:106975, February 2020. ISSN 00104655. doi: 10.1016/j.cpc.2019.106975. URL `https://linkinghub.elsevier.com/retrieve/pii/S0010465519303194`.

[49] Stanimire Tomov, Jack Dongarra, and Marc Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, June 2010. ISSN 0167-8191. doi: 10.1016/j.parco.2009.12.005.

[50] Stanimire Tomov, Rajib Nath, Hatem Ltaief, and Jack Dongarra. Dense linear algebra solvers for multicore with GPU accelerators. In *Proc. of the IEEE IPDPS'10*, pages 1–8, Atlanta, GA, April 19-23 2010. IEEE Computer Society. DOI: 10.1109/IPDPSW.2010.5470941.

[51] tuxfamily.org. Eigen is a c++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms, August 2021. URL `https://eigen.tuxfamily.org/`.

[52] R. Vandebril and M. V. Barel. A Note on the Nullity Theorem. *Journal of Computational and Applied Mathematics*, 189:179–190, 2006. doi: 10.1016/j.cam.2005.02.007.

[53] R. Yokota. An FMM Based on Dual Tree Traversal for Many-Core Architectures. *Journal of Algorithms and Computational Technology*, 7(3):301–324, 2013. Number: 3.

[54] R. Yokota, T. Narumi, R. Sakamaki, S. Kameoka, S. Obi, and K. Yasuoka. Fast Multipole Methods on a Cluster of GPUs for the Meshless Simulation of Turbulence. *Computer Physics Communications*, 180:2066–2078, 2009.

[55] C. D. Yu, S. Reiz, and G. Biros. Distributed-Memory Hierarchical Compression of Dense SPD Matrices. In *Proceedings of the 2018 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018.

[56] P. Zaspel. Algorithmic Patterns for H-Matrices on Many-Core Processors. *Journal of Scientific Computing*, 78:1174–1206, 2019. doi: 10.1007/s10915-018-0809-4.

[57] Kezhong Zhao, M.N. Vouvakis, and Jin-Fa Lee. The adaptive cross approximation algorithm for accelerated method of moments computations of emc problems. *Electromagnetic Compatibility, IEEE Transactions on*, 47:763 – 773, 12 2005. doi: 10.1109/TEMC.2005.857898.