International Journal of
HIGH PERFORMANCE
COMPUTING APPLICATIONS

Research Paper

The International Journal of High Performance Computing Applications
1–18
© The Author(s) 2019
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/1094342019861139
journals.sagepub.com/home/hpc
SAGE

Distributed-memory lattice $\mathscr{H}$-matrix factorization

Ichitaro Yamazaki[1], Akihiro Ida[2], Rio Yokota[3] and Jack Dongarra[4]

Abstract

We parallelize the LU factorization of a hierarchical low-rank matrix ($\mathscr{H}$-matrix) on a distributed-memory computer. This is much more difficult than the $\mathscr{H}$-matrix-vector multiplication due to the dataflow of the factorization, and it is much harder than the parallelization of a dense matrix factorization due to the irregular hierarchical block structure of the matrix. Block low-rank (BLR) format gets rid of the hierarchy and simplifies the parallelization, often increasing concurrency. However, this comes at a price of losing the near-linear complexity of the $\mathscr{H}$-matrix factorization. In this work, we propose to factorize the matrix using a "lattice $\mathscr{H}$-matrix" format that generalizes the BLR format by storing each of the blocks (both diagonals and off-diagonals) in the $\mathscr{H}$-matrix format. These blocks stored in the $\mathscr{H}$-matrix format are referred to as lattices. Thus, this lattice format aims to combine the parallel scalability of BLR factorization with the near-linear complexity of $\mathscr{H}$-matrix factorization. We first compare factorization performances using the $\mathscr{H}$-matrix, BLR, and lattice $\mathscr{H}$-matrix formats under various conditions on a shared-memory computer. Our performance results show that the lattice format has storage and computational complexities similar to those of the $\mathscr{H}$-matrix format, and hence a much lower cost of factorization than BLR. We then compare the BLR and lattice $\mathscr{H}$-matrix factorization on distributed-memory computers. Our performance results demonstrate that compared with BLR, the lattice format with the lower cost of factorization may lead to faster factorization on the distributed-memory computer.

Keywords

boundary element method, LU factorization, distributed memory, hierarchical matrix, task programming

1. Introduction

High-frequency acoustic/electromagnetic scattering problems on large-scale distributed-memory computers are a challenging problem from mathematical, algorithmic, and high-performance computing perspectives. Such problems must solve the dense, ill-conditioned, indefinite linear system of equations, arising from the oscillatory Green's function. To solve the linear system, a Krylov iterative solver, combined with a preconditioner, is commonly used. To reduce the computational and storage costs of the solver, there are several structured low-rank formats to store the coefficient matrix. These low-rank formats can significantly reduce the required storage and computation for performing the Krylov iteration (i.e. the matrix-vector multiplication) and for approximately factorizing the matrix to compute the preconditioner.

In this article, we study the parallelization of these low-rank matrix factorization on a distributed-memory computer. This is more difficult than the low-rank matrix-vector multiplication due to the dataflow of the factorization, and it is more challenging than parallelizing a dense matrix factorization due to the irregular hierarchical block structure of the matrix. We focus on the following three low-rank matrix formats:

- The $\mathscr{H}$-matrix format (Hackbusch, 1999) compresses only well-separated blocks (and has the strong admissibility structure), leading to the near-linear complexity of the factorization. However, it has an irregular hierarchical block structure that is

[1] Sandia National Laboratories, Computer Science Research Institute, Albuquerque, NM, USA
[2] Supercomputing Research Division, Information Technology Center, The University of Tokyo, Tokyo, Japan
[3] Tokyo Institute of Technology, Global Scientific Information and Computing Center, Tokyo, Japan
[4] Department of Electrical Engineering and Computer Science, The University of Tennessee, Knoxville, TN, USA

Corresponding author:
Ichitaro Yamazaki, Sandia National Laboratories, Computer Science Research Institute, Albuquerque, NM 87123, USA.
Emails: ic.yamazaki@gmail.com; iyamaza@sandia.gov

difficult to parallelize on a distributed-memory computer.
- The block low-rank (BLR) format (Amestoy et al., 2015) abandons the hierarchy but compresses the off-diagonal blocks. Although BLR could potentially improve the factorization's scalability, this comes at the price of losing the near-linear complexity of the $\mathscr{H}$-matrix factorization.
- The "lattice $\mathscr{H}$-matrix" format (Ida, 2018) generalizes the BLR format and uses the $\mathscr{H}$-matrix format to store each of the blocks (both diagonals and off-diagonals). These blocks stored in the $\mathscr{H}$-matrix format are referred to as lattices.

Although the factorization algorithm based on the $\mathscr{H}$-matrix format or the BLR format has been previously studied, factorizing the matrix using the lattice format is new. The main contributions of the article are as follows:

- The new factorization algorithm based on the lattice format combines the scalability of BLR with the near-linear complexity of the $\mathscr{H}$-matrix. To the best of our knowledge, the lattice format is the first attempt to balance the complexity and concurrency of these two structured low-rank formats. The hierarchical structure was previously embedded in a flat structure (Chavez et al., 2017; Yu et al., 2017), but the previous format did not consider the balance between complexity and concurrency. This is the first time the lattice format is used for factorizing the matrix.
- We compare the performance of factorization using the $\mathscr{H}$-matrix, BLR, and lattice formats under various conditions on a shared-memory computer, and using the BLR and lattice formats on distributed-memory computers. This helps quantify the benefit of the different formats and determines the conditions under which a given format outperforms the other formats. We hope that such studies are of interests to a wide range of audiences, including solver developers and users.
- Our performance comparison includes different combinations of message passing interface (MPI) and OpenMP thread or task configurations. For instance, dynamic task scheduling avoids the artificial synchronization and remedies the load imbalance caused by a large lattice size. In contrast, without tasking, the synchronization point exposes the load imbalance at the end of each factorization phase.

The rest of the article is organized as follows. We first provide the background and motivation for the current work in Section 2. We then survey the related work in Section 3. Finally, we describe our implementations of the factorization algorithms on the shared-memory and distributed-memory computers in Sections 4 and 5, respectively. After analyzing the complexity of the new algorithm in Section 6, we show the performance results in Section 7. The final remarks are listed in Section 8.

2. Background and motivations

Here, we first formulate the dense linear system of equations for electrostatic field simulations based on the boundary element method (BEM), which we solve for our experiments (Section 2.1). We then describe the software called HACApK (Ida et al., 2014) that we used to generate the $\mathscr{H}$-matrices in our study (Section 2.2).

2.1. Dense linear system for BEM analyses

The BEM analysis is a powerful tool for solving the boundary value problems of partial differential equations and is used in many scientific and engineering applications including the studies of acoustics, electromagnetics, and fracture and fluid mechanics.

Let H be a Hilbert space of functions on a surface domain $\Omega \subset \mathbb{R}^3$ and H' be the dual space of H. Given a function $f \in H'$ and a kernel function K for a convolution operator (i.e. $K : \mathbb{R}^3 \times \Omega \to \mathbb{R}$), we seek for the function $u \in H$ in the integral equation

$$\int_\Omega K(\mathbf{x}, \mathbf{y})u(\mathbf{y})\mathrm{d}\mathbf{y} = f \qquad (1)$$

In order to numerically solve the integral equation (1), we discretize the domain Ω into elements $\Omega^h = \{\omega_j : j \in \mathscr{J}\}$, where $\mathscr{J}$ is an index set. Using a weighted residual method like the collocation method, the function u is approximated in an n-dimensional subspace $H^h \subset H$. Then, given a basis set $(\varphi_i)_{i\in\mathscr{I}}$ for H^h with an index set $\mathscr{I} := \{1, \cdots, n\}$, the approximation $u^h \in H^h$ to u is expressed using a coefficient vector $\phi = (\phi_i)_{i\in\mathscr{I}}$ such that $u^h = \Sigma_{i\in\mathscr{I}} \phi_i\varphi_i$. Since the supports of the basis, $\Omega^h_{\varphi_i} :=$ supp φ_i, are assembled from the sets ω_j, the integral equation (1) can be transformed to the following linear system of equations:

$$A\phi = \mathbf{b} \qquad (2)$$

For instance, in the case of the Galerkin method, the entries of the coefficient matrix A are given by

$$a_{ij} = \int_\Omega \int_\Omega \phi_i(\mathbf{x})K(\mathbf{x}, \mathbf{y})\phi_j(\mathbf{y})\, \mathrm{d}\mathbf{x}\, \mathrm{d}\mathbf{y} \qquad (3)$$

for all $i, j \in \mathscr{I}$.

Given two index subsets $\mathscr{I}_i, \mathscr{I}_j \subset \mathscr{I}$ and the corresponding subdomains defined by

$$\Omega^h_{\mathscr{I}_i} := \bigcup_{i\in\mathscr{I}_i} \text{supp}\ \varphi_i, \qquad \Omega^h_{\mathscr{I}_j} := \bigcup_{i\in\mathscr{I}_j} \text{supp}\ \varphi_i \qquad (4)$$

we consider the pair $(\mathscr{I}_i, \mathscr{I}_j)$ "admissible" if the Euclidean distance between $\Omega^h_{\mathscr{I}_i}$ and $\Omega^h_{\mathscr{I}_j}$ is sufficiently large compared with their diameters, that is

$$\min\left\{\text{diam}(\Omega^h_{\mathscr{I}_i}), \text{diam}(\Omega^h_{\mathscr{I}_j})\right\} \le \eta\ \text{dist}(\Omega^h_{\mathscr{I}_i}, \Omega^h_{\mathscr{I}_j}) \qquad (5)$$

where η is a positive constant scalar whose value depends on both K and Ω^h. On the subdomain corresponding to the admissible pairs, we assume that the kernel function can be approximated to a certain accuracy by a degenerate kernel such as

$$K(\mathbf{x},\mathbf{y}) \cong \sum_{\nu=1}^{r} K_1^{\nu}(\mathbf{x})K_2^{\nu}(\mathbf{y}) \tag{6}$$

where $\mathbf{x} \in \Omega^h_{\mathscr{I}_i}$, $\mathbf{y} \in \Omega^h_{\mathscr{I}_j}$, and r is a positive number. This type of kernel function appears in many scientific and engineering applications, including in the electric field and mechanical analyses. Under this assumption, the double integral in equation (3) can be split into two separate single integrals with respect to $\mathbf{x}$ and $\mathbf{y}$. Thus, for a pair of admissible subdomains $\Omega^h_{\mathscr{I}_i}$ and $\Omega^h_{\mathscr{I}_j}$ (with a faraway interaction), the corresponding dense submatrix $A|_{\mathscr{I}_i \times \mathscr{I}_j}$ can be approximated by its low-rank factorization. Furthermore, given the admissible condition (5), it is possible to find a permutation and a partition of the index set $\mathscr{I}$ such that after the permutation and partition, many of large submatrices in A become numerically low-rank.

2.2. HACApK linear solver

HACApK (Ida et al., 2014) is a software package that uses the $\mathscr{H}$-matrix format for solving dense linear systems of equations from various applications. One such application is the ppohBEM (Iwashita et al., 2017) open-source software package that numerically solves the integral equations for the BEM analysis. The numerical solution of the integral equations requires solving the dense linear system of equation (2), for which ppohBEM relies on HACApK.

To compute the appropriate matrix permutation and partition for generating the $\mathscr{H}$-matrix, HACApK computes the cluster tree $T_{\mathscr{I}}$ based on the geometrical information associated with the index set $\mathscr{I}$. The index set is repeatedly divided into disjoint subsets $\mathscr{I}_i^d$ at each depth d of the tree such that $\mathscr{I} = \sum_i \mathscr{I}_i^d$. The clustering process continues until each cluster size $|\mathscr{I}_i^d|$ becomes sufficiently small. We then compute the direct product of the cluster tree $T_{\mathscr{I}}$ with itself to create a block cluster tree $T_{\mathscr{I}\times\mathscr{I}}$. The branches of $T_{\mathscr{I}\times\mathscr{I}}$ are truncated such that $(\mathscr{I}_i, \mathscr{I}_j) \in T_{\mathscr{I}\times\mathscr{I}}$ is an admissible cluster pair, and the corresponding off-diagonal blocks of large dimensions become low rank. The set $\mathscr{L}(T_{\mathscr{I}\times\mathscr{I}})$ contains all the block clusters that have no branch and are called leaves. These leaves represent the partition structure of the matrix A, as shown in Figure 1(a), and we refer to the user-specified minimum size of the leaves as the leaf size.

To compute the $\mathscr{H}$-matrix representation of A, HACApK generates a low-rank representation of each admissible block of A by algebraically approximating the degenerate kernel using adaptive cross approximation (ACA) or ACA+ (Bebendorf and Rjasanow, 2003; Kurtz et al., 2002). For each low-rank block given by $B \approx VY^T$,

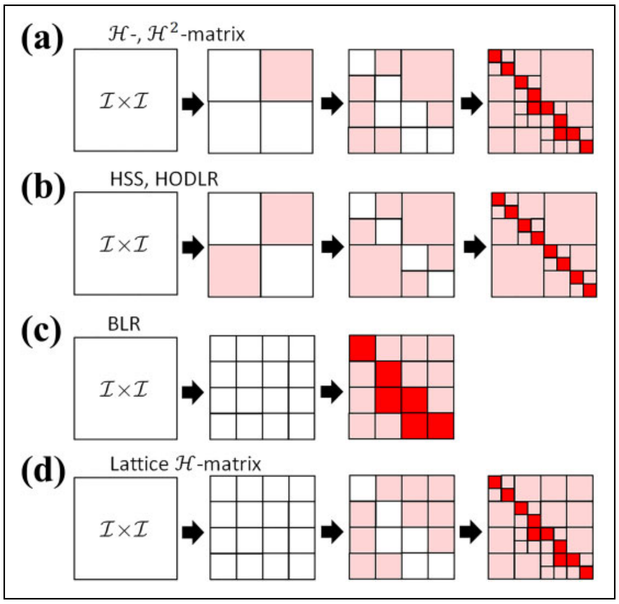

(a)
$\mathcal{H}$-, $\mathcal{H}^2$-matrix
$\mathcal{I}\times\mathcal{I}$
(b)
HSS, HODLR
$\mathcal{I}\times\mathcal{I}$
(c)
BLR
$\mathcal{I}\times\mathcal{I}$
(d)
Lattice $\mathcal{H}$-matrix
$\mathcal{I}\times\mathcal{I}$

Figure 1. Conceptual diagram of the construction of representative low-rank structured matrices using different partition structure M: (a) $\mathscr{M}_{\mathscr{H}}$ for a general $\mathscr{H}$-matrix, (b) $\mathscr{M}_W$ for an HSS, (c) $\mathscr{M}_L$ for a BLR, and (d) $\mathscr{M}_{L\mathscr{H}}$ for a lattice $\mathscr{H}$-matrix. Blank boxes show nonadmissible blocks. Blocks in light red indicate submatrices judged as low-rank, and blocks painted in deep red are remaining non-admissible blocks calculated as dense submatrices. (a) $\mathscr{H}$- and $\mathscr{H}^2$-matrix. (b) HSS, HODLR. (c) BLR. (d) Lattice $\mathscr{H}$-matrix. HSS: hierarchical semi-separable; BLR: block low-rank.

the matrices V and Y are explicitly computed and stored. Although the block cluster tree $T_{\mathscr{I}\times\mathscr{I}}$ contains all the information needed for the factorization, HACApK does not keep the block cluster tree, the individual cluster tree $T_{\mathscr{I}}$, nor any interaction list between their nodes. The only kept information is the leaves that coincide with the $\mathscr{H}$-matrix partition structure, and we use that information for factorizing the matrix.

3. Low-rank matrices

In this section, we survey existing low-rank matrix formats and software packages that utilize such formats.

3.1. Partition structures

We define a partition structure $\mathscr{M}$ as a set of disjoint subsets $\mathscr{I}_i \times \mathscr{I}_j$ of $\mathscr{I} \times \mathscr{I}$, whose union spans the whole set $\mathscr{I} \times \mathscr{I}$. These index subsets $\mathscr{I}_i \times \mathscr{I}_j$ are assumed to be appropriately chosen in $\mathscr{I} \times \mathscr{I}$ such that the corresponding submatrices $A|_{\mathscr{I}_i \times \mathscr{I}_j}$ of A can be well approximated by low-rank matrices. Figure 1 illustrates how a different low-rank structured matrix can be generated from a different partition structure $\mathscr{M}$ for a given problem. In fact, we can generate these low-rank structures by controlling the parameter η in the admissible condition (5) and the branch truncation of the block cluster tree $T_{\mathscr{I}\times\mathscr{I}}$ during the construction of the $\mathcal{H}$-matrix. Thus, these low-rank matrices can be regarded as special types of the $\mathcal{H}$-matrices.

Each partition structure has its own pros and cons. For instance, the $\mathcal{H}$-matrix has the most general low-rank matrix structure, and it is an effective way of compressing the matrix with small numerical ranks. However, the $\mathcal{H}$-matrix usually has the irregular partition structure $\mathcal{M}_{\mathcal{H}}o$, as shown in Figure 1(a), which makes it challenging to perform some matrix operations on distributed-memory computers (e.g. LU factorization).

To ease the implementation and to improve the parallel scalability of the distributed matrix operations, simpler partition structures have been proposed. For instance, by introducing the weak admissible condition—$\eta = \infty$ in the admissible condition (5)—we can construct the partition structure $\mathcal{M}_W$, which is used to generate the hierarchical semi-separable matrix or the hierarchically off-diagonal low-rank matrix, and is shown in Figure 1(b). Compared with the $\mathcal{H}$-matrix structure $\mathcal{M}_{\mathcal{H}}$, the structure $\mathcal{M}_W$ is simpler and more convenient for performing certain matrix operations on distributed-memory computers. However, this matrix structure assumes a weak admissibility condition, where all off-diagonal blocks are assumed to be low-rank. When the weak admissibility condition is applied to a 3-D or higher dimensional problem, this structure leads to a higher asymptotic complexity due to the larger ranks of off-diagonal blocks.

BLR, as shown in Figure 1(c), is another simpler matrix structure. It has the lattice partition structure $\mathcal{M}_L$, but each lattice is either low-rank or dense. Although the construction of the BLR matrix does not require the block cluster tree $T_{\mathcal{I}\times\mathcal{I}}$ used to construct the $\mathcal{H}$-matrix, it can be constructed by truncating all branches of the cluster tree $T_{\mathcal{I}}$ at a certain depth level d. The memory complexity of the BLR matrix is $\mathcal{O}(n^{1.5})$ (Amestoy et al., 2017) and is higher than $\mathcal{O}(n\log n)$ of the $\mathcal{H}$-matrix (Hackbusch, 1999). However, the BLR matrix is a simple, nonhierarchical, and effective low-rank format, especially for distributed memory. For instance, the lattice structure $\mathcal{M}_L$ observed in the BLR matrix is similar to the 2-D block layout used in many dense matrix operations (e.g. Scalable Linear Algebra PACKage (ScaLAPACK)), and it can use many of the high-performance optimization techniques developed for the dense matrix operations (Ida et al., 2018b).

Figure 1 also shows that the BLR format allows the numerical nonsymmetry (e.g., numerical ranks of the low-rank blocks, or the dense or low-rank block pattern, can be different in the transpose of the matrix). In contrast, the $\mathcal{H}$-matrix format allows both the numerical and structural nonsymmetries. This irregular structure of the $\mathcal{H}$-matrix leads to the flexibility to effectively compress the matrix but also poses the challenges when factorizing the matrix on a distributed-memory computer.

Finally, Figure 1(d) shows the hybrid partition structure $\mathcal{M}_{L\mathcal{H}}$ of the lattice $\mathcal{H}$-matrix that we use in this article. It combines the BLR's lattice structure $\mathcal{M}_L$ with the $\mathcal{H}$-matrix partition structure $\mathcal{M}_{\mathcal{H}}$ by introducing the lattice structure on top of the $\mathcal{H}$-matrix structure. In other words, the lattice $\mathcal{H}$-matrix utilizes the $\mathcal{H}$-matrix format $\mathcal{M}_{\mathcal{H}}$ for each lattice block in $\mathcal{M}_L$. These lattice blocks are then distributed among the processes in a 2-D block cyclic fashion. This structure is designed to balance the advantages of the $\mathcal{H}$-matrix and BLR formats: the high compressibility of the $\mathcal{H}$-matrix format and the parallel scalability of the BLR format. Given a large enough size of the lattice, the lattice $\mathcal{H}$-matrix can reduce the computational and memory complexities from $\mathcal{O}(n^2)$ and $\mathcal{O}(n^{1.5})$ of the BLR format to $\mathcal{O}(n\log^2 n)$ and $\mathcal{O}(n\log n)$ of the $\mathcal{H}$-matrix format, respectively. Further, the lattice matrix format has the same communication pattern as the BLR format, and it can utilize the efficient communication schemes developed for dense matrix operations. Finally, using the lattice $\mathcal{H}$-matrix format with a hybrid MPI-thread programming paradigm, the complex matrix operations originating from the $\mathcal{H}$-matrix structure are performed using the threaded computational kernels.

Table 1. Structured low-rank factorization methods.[a]

Format	Structure	Nested	Implementation
$\mathcal{H}$-matrix	Strong-$\mathcal{H}$	No	HLIBpro, AHMED
$\mathcal{H}^2$-matrix	Strong-$\mathcal{H}$	Yes	LORASP, H2lib
HSS	Weak-$\mathcal{H}$	Yes	STRUMPACK
BLR	Strong-block	No	MUMPS-BLR, HiCMA
Lattice $\mathcal{H}$-matrix	Strong-($\mathcal{H}$ + block)	No	HACApK

HSS: hierarchical semi-separable; BLR: block low-rank; HiCMA: Hierarchical Computations on Manycore Architectures; MUMPS: MUltifrontal Massively Parallel Sparse direct Solver.
[a]The $\mathcal{H}$-matrix (Hackbusch, 1999) and the $\mathcal{H}^2$-matrix (Hackbusch et al., 2000) have strong admissibility structure and compress only well-separated blocks, while the HSS matrix (Chandrasekaran et al., 2006; Liu et al., 2016; Wang et al., 2013) has weak admissibility where only the diagonal blocks remain uncompressed. The BLR (Amestoy et al., 2015) abandons the hierarchy but compresses the off-diagonal blocks. The $\mathcal{H}^2$ and HSS matrices have nested bases, whereas the others do not. The lattice $\mathcal{H}$-matrix in the last row represents our current contribution.

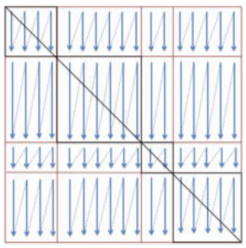
3.2. Software

Table 1 lists software packages implementing different structured low-rank factorization methods. We do not include analytical methods like the fast multipole method (FMM) (Greengard and Rokhlin, 1987) because they cannot be used for the factorization, which is the focus of this article. In the figure, "Structure" categorizes the type of admissibility condition (strong or weak) (Hackbusch, 1999) and classifies whether the blocks are subdivided recursively or not ($\mathcal{H}$ or block, respectively); "Nested" represents whether the bases are nested or not; and "Implementation" shows the representative software packages that implement these methods.

- geometry-oblivious FMM (GOFMM) (Yu et al., 2017): In applications where the matrix is manipulated (e.g. multiplied by another matrix or averaged over), it is impossible to retrieve geometry information—this is where GOFMM is useful. In our application, geometry information is accessible, and we exploit the information to achieve better performance (e.g. matrix partitioning).

- inv-ASKIT (Yu et al., 2016): In applications where the matrix comes from a kernel function in high ambient dimensions but low intrinsic dimensions, ASKIT is the state of the art. The weak admissibility of ASKIT is a significant limitation for the oscillatory kernels that we are dealing with. Our method uses strong admissibility to prevent the numerical ranks of the matrix blocks from growing with the problem size.
- LoRaSp (Pouransari et al., 2017): Although it can be used to solve the extended sparsified form of dense matrices, LoRaSp is designed primarily for solving sparse matrices. Our focus is on factorizing dense matrices.
- STRUMPACK (Rouet et al., 2016): It is built on established linear algebra libraries for distributed-memory computers. STRUMPACK uses weak admissibility and has the same limitation as ASKIT.
- Hierarchical Computations on Manycore Architectures (HiCMA) (Akbudak et al., 2017) and the MUltifrontal Massively Parallel Sparse direct Solver (MUMPS)-BLR (Amestoy et al., 2015): Using BLR instead of a hierarchical format, the library can exploit many high-performance techniques developed for dense matrix operations, including efficient parallel computation and communication schemes. It is a powerful scalable approach, though the lack of hierarchy may eventually limit their performance as the problem size increases (e.g. the nonlinear complexity). To improve the complexity, a multilevel variant of BLR has been recently proposed, where the matrix is recursively partitioned in the BLR format (Amestoy et al., 2018).
- Butterfly (Li et al., 2015): The butterfly algorithm is in general the best approach to factorize the submatrices of an oscillatory kernel matrix on a single process. However, prior work on the parallel scalability of the butterfly algorithm (Poulson et al., 2014) has shown that it has similar communication complexity as fast Fourier transform.
- Hierarchical interpolative factorization (HIF) (Li and Ying, 2016): Recently, HIF was extended to handle strong admissibility (Minden et al., 2017) and to run on a distributed-memory computer (Li and Ying, 2016). However, these works have not been combined to handle the strong admissibility on the distributed-memory computer, and it cannot address the goal of this work.
- HLIBpro (Aliaga et al., 2017; Kriemann, 2014): HLIBpro can compute the LU factorization of a $\mathscr{H}$-matrix on a distributed-memory computer. The parallel scalability of HLIBpro on a distributed-memory computer has not yet been published. This code does not offer the capability to combine the two low-rank formats for balancing complexity and concurrency like we do in the present work.

(a)



(b)

```
for $k = 1, 2, \ldots, n_t$ do
    // Factorize diagonal block
    $[P_k, L_{k,k}, U_{k,k}] := \mathrm{LU}(B_{k,k})$
    for $i = k+1, k+2, \ldots, n_t$ do
        // Compute blocks in panel column
        $L_{i,k} := B_{i,k} U_{k,k}^{-1}$
    end for
    for $j = k+1, k+2, \ldots, n_t$ do
        // Compute blocks in panel row
        $U_{k,j} := L_{k,k}^{-1} P_k B_{k,j}$
    end for
    for $i = k+1, k+2, \ldots, n_t$ do
        for $j = k+1, k+2, \ldots, n_t$ do
            // Update trailing block
            $B_{i,j} := B_{i,j} - L_{i,k} U_{k,j}$
        end for
    end for
end for
```

Figure 2. Dense block LU factorization, where n_t is the number of the blocks in the matrix row or column. (a) Dense block layout. (b) Block LU factorization algorithm.

The performance comparison with other software packages is of our interest but out of this article's scope. In the current article, we focus on comparing the performance of a new lattice format with the BLR and $\mathscr{H}$-matrix formats in the same software framework.

4. Shared-memory implementation

We now describe the low-rank factorization algorithms that we implemented for the shared-memory multicores. We first introduce the basic dense block LU factorization algorithm, on which all of our implementations are based (Section 4.1). We then describe the two low-rank factorization algorithms, that is, BLR LU (BLU) and hierarchical LU (HLU; Sections 4.2 and 4.3). The threaded computational kernels developed in this section are used by each process for our distributed-memory factorization in the next section.

The following three terms are used to describe our implementations:

- "blocks" are the blocks in either the $\mathscr{H}$-matrix or BLR formats, and can be either dense or low-rank.
- "lattices" are the lattices in the lattice $\mathscr{H}$-matrix format, and each of the lattice is stored in the $\mathscr{H}$-matrix format.
- "tiles" represent a logical partitioning of the matrix, and are conceptually generated by splitting the matrix along the diagonal blocks (i.e., in the BLR format, the tiles coincide with the blocks).

The blocks are the physical submatrices of the matrix, being stored either in the dense or low-rank format. In contrast, the tiles are the conceptual submatrices of the matrix, being used only to describe our implementations.

4.1. Dense LU

Our LU factorization algorithm for the dense matrix first splits the matrix into small blocks (typically into the square blocks of the same sizes). Hence, the matrix is partitioned into the n_t-by-n_t blocks. Each of the blocks is stored in column-major order in a contiguous memory region (see Figure 2(a)). For accessing the block using the block row and column indexes (e.g. $B_{i,j}$), we create the n_t-by-n_t array of pointers, each of which points to the beginning of the corresponding block.

The factorization algorithm is then broken into the computational tasks that operate on the blocks. At each step, the leading block column and row, referred to as the *panel* column and row, respectively, are factorized by first factorizing the diagonal block and then factorizing the off-diagonal blocks. These panels are then used to update the trailing submatrix block by block. As shown in Figure 2(b), all the operations on the blocks can be performed using the standard LAPACK or Basic Linear Algebra Subprograms (BLAS) subroutines whose vendor-optimized implementations are often available on their specific hardware. We note that in this storage format, the blocks coincide with the tiles.

To maintain the numerical stability of the LU factorization, *pivoting* is needed, for example, $PA = LU$ is the LU factorization of A where the matrices L and U are the lower and upper triangular matrices, respectively, and the matrix P pivots the rows of A. To maintain the numerical stability of the factorization for matrices arising from the kinds of applications, we are interested in, it was sufficient to seek pivots only within the diagonal blocks (line 2 of Figure 2(b)).

These blocked algorithms for dense matrix factorization have been studied to exploit the multicore architectures (Buttari et al., 2009). These blocks (stored contiguously in memory) can be loaded into the cache memory efficiently and operated on with little risk of eviction. Thus, the use of the block layout minimizes the number of cache misses and maximizes the potential for prefetching. In addition, operations on small blocks create fine-grained parallelism that provides enough independent tasks to keep a large number of threads, or processes, busy.

4.2. Block low-rank LU

In order to reduce the storage and computational cost of the factorization, the BLR format compresses some of the off-diagonal blocks into their low-rank representations (e.g. $B_{k,j} \approx V_{k,j}Y_{k,j}^T$). To exploit the low-rank structures of the off-diagonal blocks well, the diagonal blocks of the matrix could have different dimensions.

The LU factorization of the BLR matrix is then computed using the algorithm in Figure 2(b), where some of the off-diagonal blocks are now low-rank. For instance, on line 4 of Figure 2(b), if the off-diagonal block of the panel column (or row) is low rank, then the triangular solve with the upper (or lower) factor of the diagonal block is applied to its $Y_{k,j}$ (or $V_{i,k}$) matrix (i.e. $Y_{k,j} := Y_{k,j}U_{k,k}^{-1}$ or $V_{i,k} := L_{k,k}^{-1}V_{i,k}$).

Then, to update $B_{i,j}$ on line 8 (i.e. $B_{i,j} := B_{i,j} - L_{i,k}U_{k,j}$), each of the three blocks involved, $B_{i,j}$, $L_{i,k}$, and $U_{k,j}$, can be either dense or low-rank, giving eight potential configurations for the updating kernel. We update these blocks to minimize the floating-point operation (FLOP) count. For instance, to update a dense n_i-by-n_j block using two low-rank blocks, $B_{i,j} := B_{i,j} - (V_{i,k}Y_{i,k}^T)(V_{k,j}Y_{k,j}^T)$, we first compute the small $r_{i,k}$-by-$r_{k,j}$ matrix $T := Y_{i,k}^T V_{k,j}$, where $r_{i,k}$ and $r_{k,j}$ are the respective numerical ranks of the updating blocks $L_{i,k}$ and $U_{k,j}$. We then multiply T with either $V_{i,k}$ or $Y_{k,j}^T$, depending on the required FLOP counts. Finally, $B_{i,j}$ is updated with the low-rank matrix, for example, $B_{i,j} := B_{i,j} - V_{i,k}(TY_{k,j}^T)$. Compared with the dense-block update that requires $O(n_i n_j n_k)$ FLOPs, the low-rank update only requires $O(n_i n_j \min(r_{i,k}, r_{k,j}))$ FLOPs. As a result, the low-rank compression can significantly reduce the FLOP count when the blocks have small ranks (i.e. $r_{i,k}, r_{k,j} \ll n_k$). Similarly, to update a dense block using a low-rank block and a dense block, we first merge the dense block into the low-rank block, for example, $B_{i,j} := B_{i,j} - V_{i,k}(Y_{i,k}^T B_{k,j})$.

For updating a low-rank block, we first compute the low-rank representation of the update, that is, $B_{i,j} := B_{i,j} - \bar{V}_{i,j}\bar{Y}_{i,j}^T$ (similar to the dense-block update, this low-rank representation is computed by minimizing the flop count). Hence, after the update, $B_{i,j}$ can be represented as $B_{i,j} = \hat{V}_{i,j}\hat{Y}_{i,j}^T$, where $\hat{V}_{i,j} = [V_{i,j}, -\bar{V}_{i,j}]$ and $\hat{Y}_{i,j} = [Y_{i,j}, \bar{Y}_{i,j}]$, and $V_{i,j}Y_{i,j}^T$ is the original low-rank representation of $B_{i,j}$ before the update.

To avoid the increase in the numerical rank and the storage, we use ACA (Bebendorf and Rjasanow, 2003; Kurtz et al., 2002) to recompress the low-rank block after each update. As shown in Figure 3, ACA only needs to

```
1: $\Pi_{\text{row}} = \emptyset$, $\Pi_{\text{col}} := \emptyset$, $r := 0$, $\pi_1 := 1$,
2: **while** not converged **do**
3:     *// increment numerical rank*
4:     $r := r + 1$
5:     *// generate pivot row*
       $\mathbf{y}_{:,r} := \mathbf{b}_{\pi_r,:}^T - \mathbf{y}_{:,1:r-1}\mathbf{v}_{\pi_r,1:r-1}^T$
6:     *// pick pivot column*
       $\pi_r := \arg\max_j(|y_{j,r}| : j \notin \Pi_{\text{col}})$
       $\Pi_{\text{col}} := \Pi_{\text{col}} \cup \{\pi_r\}$
7:     *// generate pivot column*
       $\mathbf{v}_{:,r} := \mathbf{b}_{:,\pi_r} - \mathbf{v}_{:,1:r-1}\mathbf{y}_{\pi_r,1:r-1}^T$
8:     *// pick pivot row*
       $\pi_r := \arg\max_i(|v_{i,r}| : i \notin \Pi_{\text{row}})$
       $\Pi_{\text{row}} := \Pi_{\text{row}} \cup \{\pi_r\}$
9:     *// convergence check*
       $\|E\| := \|V_{:,1:r}\|\|Y_{:,1:r}\|$
       **if** $r == 1$ **then** $\|A\| := \|E\|$
       **if** $\|E\| \le \tau\|A\|$ **then** break;
10: **end while**
```

Figure 3. ACA algorithm. ACA: adaptive cross approximation.

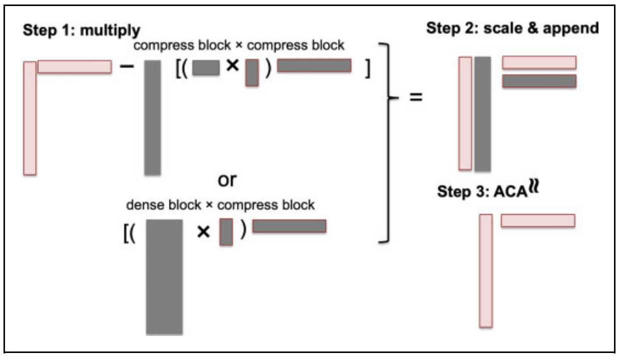


Figure 4. Illustration of BLR update of a low-rank block. BLR: block low-rank.

generate the pivot rows and columns and does not need to explicitly form the dense representation of the low-rank block. Hence, at each step of ACA, we compute the pivot row (or column) by multiplying the corresponding row of $\hat{Y}_{i,j}$ (or $\hat{V}_{i,j}$) with $\hat{V}_{i,j}$ (or $\hat{Y}^T_{i,j}$). Figure 4 illustrates these updating schemes.

To recompress the low-rank blocks, we also implemented an option that computes the QR and LQ factorization of $\hat{V}_{i,j}$ and $\hat{Y}^T_{i,j}$, respectively. The algorithm then multiplies two small resulting triangular matrices and computes the singular-value decomposition (SVD) of the resulting matrix to recompress the block before truncating the singular values to satisfy the specified accuracy. Nevertheless, this second approach has a higher computational complexity than ACA, and LAPACK's `dgesvd` can be slow, or fail, to converge for our numerically low-rank blocks.[1]

4.3. Hierarchical matrix LU (HLU)

Our next implementation directly factorizes the $\mathscr{H}$-matrix, following the same LU factorization procedure in Figure 2(b). The only difference is that unlike in the BLR format, the $\mathscr{H}$-matrix blocks are no longer coincide with the tiles (as illustrated in Figure 5). Hence, at each step of the factorization, when we are computing the off-diagonal blocks of the lower-triangular matrix, we need to apply the triangular solve to the corresponding parts of the $\mathscr{H}$-matrix blocks, and some of the blocks may span multiple tiles.

In order to access individual tiles within each of the $\mathscr{H}$-matrix blocks, we create several auxiliary integer arrays. In particular, though the boundaries of these blocks are still aligned with the diagonal blocks, and hence with the boundaries of tiles, each block may span multiple tiles. Hence, we again have the n_t-by-n_t array of pointers, but each entry of the array now points to the beginning of the block that the corresponding tile belongs to. We also have the additional array that specifies the offsets into the block for each tile. Using these offsets for the panel factorization, the triangular solve with the diagonal block can be applied to the specific part of the off-diagonal blocks. We also store the first and last (both column and row) tile indexes of the

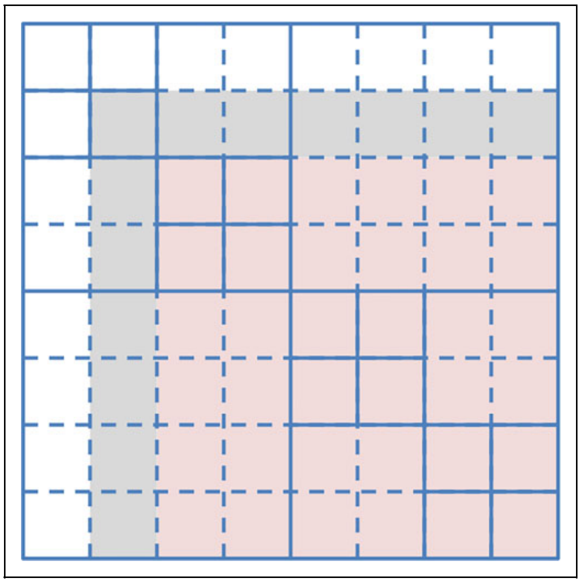


Figure 5. Data structure used for storing the $\mathscr{H}$-matrix, where solid and dashed lines show the block and tile boundaries, respectively. The matrix is stored by blocks, while the tiles are conceptual and only used by our factorization algorithms. We have a pointer for each tile such that we can access the corresponding part of the block (e.g., the part of the block colored in grey or pink). These pointers are useful because the blocks are updated by the tiles in each panel. For example, the grey tiles are used for the panel factorization and update for the second step of the factorization.

block. Hence, after applying the triangular solve to each block, we can move to the next block. Figure 6 shows the pseudocode of our subroutine that computes the off-diagonal blocks of the lower-triangular factor.

After the panel factorization, the trailing submatrix is updated block by block. The main difference between BLU and $\mathscr{H}$-matrix LU (HLU) is that, as illustrated in Figure 7, HLU may update a block using multiple blocks, each of which can be either dense or compressed. It is not difficult to update a dense block using multiple blocks, but it is a challenge to update a low-rank block, that is, $\hat{V}\hat{Y}^T := VY^T - \bar{V}\bar{Y}$. This is because explicitly generating the updating blocks $\bar{V}$ and $\bar{Y}$ as in Section 4.2 could significantly increase the memory cost (e.g. if $\bar{V}$ has a small dense block and a large low-rank block, then the low-rank block needs to be converted to a dense block).

Thus, to update the low-rank block, we compress the low-rank block $\hat{V}\hat{Y}^T$ by performing ACA without explicitly forming the blocks $\hat{V}$ or $\hat{Y}$. Namely, at each step of ACA, we first generate the row of $\hat{V}$, corresponding to the pivot row (or the pivot column of $\hat{Y}^T$). We then generate the new row of the low-rank block by multiplying this pivot row of $\hat{V}$ with the blocks in $\hat{Y}^T$ (or the column of $\hat{Y}^T$ with the blocks in $\hat{V}$). In contrast, the SVD-based recompression kernel, developed for BLU in Section 4.2, requires the explicit generation of $\bar{V}$ and $\bar{Y}$, and is difficult to be used for HLU.

```
for (int i = k+1; i < mt; i++) {
  // compute i-th off-diagonal L(i,k)
  if (isDense(i,k)) {
    // dense block
    // > L(i,k) := A(i,k)*U(k,k)^{-1}
    mb = getBlockMb(i,k);
    B = getTileB(i,k);
  } else {
    // compressed block
    // > Y(i,k) := Y(i,k)*U(k,k)^{-1}
    mb = getBlockRank(i,k);
    B = getTileY(i,k);
  }
  int nb = getTileNb(i,k);
  double *U = getBlock(k,k)
  cblas_dtrsm(CblasColMajor,
              CblasRight, CblasUpper,
              CblasNoTrans, CblasNonUnit,
              mb, nb,
              1.0, U, nb,
                   B, mb);
  // skip to the next block
  i = getEndI(i,k);
}
```

Figure 6. Subroutine to compute the off-diagonal blocks of the lower-triangular factor L at the k-th step of factorization. When the (i, k)-th tile is a part of a dense block B, `getTileB (i, k)` returns the pointer to the beginning of the tile. When the tile is a part of a low-rank block $B = VY^T$, `getTileY (i, k)` returns the pointer to the part of Y^T, corresponding to the tile. Then, `getBlockMb (i, k)` and `getBlockRank (i, k)` return the row dimension and the numerical rank of the block that the (i, k)-th tile belongs to, respectively, while `getTileNb (i, k)` returns the column dimension of the tile. Finally, `getEndI (i,k)` returns the last row tile index of the block that the (i, k)-th tile belongs to.

In addition, some of the $\mathscr{H}$-matrix blocks may be parts of both the panel and the trailing submatrix. In this case, we only update the part of the block belonging to the trailing submatrix, and then recompress only the trailing part of the block if the block is low rank. For example, in Figure 7, the low-rank block VY^T is part of the panel row and the trailing submatrix. After the update, this low-rank block is given by $\hat{V}\hat{Y}^T$ where $\hat{V} = [V, -\bar{V}]$ and $\hat{Y} = [Y, Y]$, and $-\bar{V}$ contains the trailing submatrix update. Hence, the top part of $\bar{V}$ (belonging to the panel row or the previously computed rows of the upper-triangular factor) is zero. We then update the block by accumulating the updates $-\bar{V}$ into V. Namely, the updated block is given by $\hat{V}Y^T$ where $\hat{V} := V - \bar{V}$. We also looked at applying ACA to recompress the whole low-rank block $\hat{V}\hat{Y}^T$. However, the first approach requires less computation, and it maintains the consistency of the factorization since it only updates the trailing submatrix part of the low-rank block, while the second approach updates the whole block—including the part of the block that has been already used to update the trailing submatrix. As a result, the first approach can lead to a higher accuracy of the factorization.

We emphasize that though the triangular solve and the trailing submatrix update are performed per panel, the computational tasks are performed on the blocks and not on the tiles.

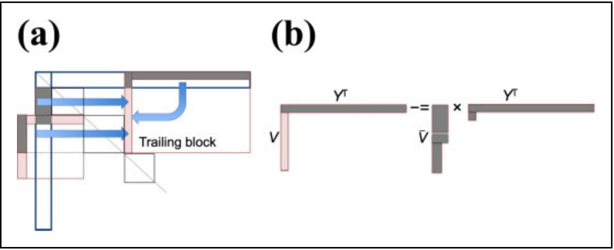


Figure 7. Updating a low-rank $\mathscr{H}$-matrix block that is part of panel and trailing submatrix (where the $\mathscr{H}$-matrix format allows the nonsymmetric partition structure). (a) Block layout. (b) Block update.

4.4. Task-based implementation

We use OpenMP task-based programming model to factorize the low-rank matrix on a shared-memory computer. OpenMP's task scheduling has also been used to develop parallel numerical software, including Parallel Linear Algebra Software for Multicore Architectures, for dense linear algebra subroutines (YarKhan et al., 2017).

To define the data dependencies among the OpenMP tasks, we typically specify the task's required data using its memory location. In contrast, we cannot use the memory pointers to keep track of the data dependencies of the low-rank factorization because we dynamically free and reallocate the compressed blocks, as their numerical ranks can change after each update. Instead, we used a separate n_t-by-n_t integer array to keep track of the task dependencies of the factorization. The task-based programming may reduce the CPU cores' idling time by removing the unnecessary synchronization points. This could be critical, especially for the low-rank factorization that performs the tasks with variable computational costs.

5. Distributed-memory implementation

We now extend our shared-memory implementations of low-rank factorization to run on the distributed-memory computers.

5.1. Block low-rank LU

To parallelize the BLU factorization on distributed-memory computers, we arrange the processes on a p-by-q 2-D grid and distribute the blocks in a 2-D cyclic fashion among the processes. Then, to factorize the matrix, each process updates and factorizes only its local blocks. This parallelization scheme is used in many popular linear algebra packages including the ScaLAPACK for factorizing dense matrices and leads to a simple but efficient communication pattern.

Namely, to communicate the blocks, we create two subcommunicators for each process: one for the processes in the same column of the process grid and the other for the processes in the same row. Then, at each factorization step, the lower and upper triangular factors of the diagonal block are broadcasted using these two sub-communicators along the row and column process grid, respectively. Once the diagonal factor is received and the off-diagonal block of the lower or upper triangular factor is computed, the computed block is broadcasted to the processes in the same row or column of the process grid, respectively. Thus, each process sends the blocks only to p and q processes (e.g. $2\sqrt{n_p}$ processes on a square grid).

The 2-D cyclic distribution of the blocks also helps maintain the processes' load balance when small enough blocks are used. However, a smaller block size also increases the total communication latency cost and may lower the performance of the BLAS or LAPACK subroutine to perform the local computation. Therefore, the block or the leave size needs to be carefully selected.

To extend our task-based implementation of the shared-memory BLU to utilize distributed-memory computers, we insert OpenMP tasks that call `MPI_Bcast` to broadcast the blocks using the sub-communicators. Once scheduled to be executed, our communication task is blocked until the corresponding communication task is scheduled on other processes, leading to the idling time of the core. In order to reduce the number of idling cores, we reduce the number of communication tasks by having each communication task send all the local off-diagonal blocks of the panel column or row.

Although the multiple broadcasts may be executed in parallel, the number of the communication tasks that may be executed at the same time is bounded by the number of threads. Hence, we create a separate communicator for each thread and use the communicators in a round-robin fashion at each step of factorization.

We note that the OpenMP runtime manages the dependency graph of only the local tasks, and avoids explicitly forming the global dependency graph of the factorization, which can be a significant overhead on the runtime system.

5.2. Lattice $\mathscr{H}$-matrix LU

As it has been discussed throughout the article, there are trade-offs between BLU and HLU. For instance, compared to BLU, HLU has lower storage and arithmetic complexities, but it is more difficult to distribute the $\mathscr{H}$-matrix blocks among the processes for parallel scalability. The distribution may lead to a load imbalance, an irregular communication pattern, or a higher communication cost than that of BLU. For instance, some blocks of the panel in the $\mathscr{H}$-matrix may need to be sent to different sets of processes, or some blocks may need to be sent to all the processes. Thus, it is a challenge to implement HLU on a distributed-memory computer. In contrast, though BLU has higher storage and arithmetic complexities, it has a more regular communication pattern that is easier to optimize for parallel scalability.

To combine the advantages of HLU and BLU, we store the matrix in the lattice $\mathscr{H}$-matrix format that generalizes the BLR format. Linke in the BLR format, the matrix is still distributed in the 2-D block cyclic fashion, but each block, referred to as *lattice*, is now stored in the $\mathscr{H}$-matrix format. This eases the parallelization because we can use the threaded computational kernels for the HLU factorization from Section 4.3 to perform the complex matrix operations originating from the $\mathscr{H}$-matrix structure, while we have the same regular communication pattern as the distributed BLU in Section 5.1. Since each of the lattices is stored as an $\mathscr{H}$-matrix, the lattice matrix LU (LLU) factorization enjoys the high compressibility of the lattices (similar to the HLU factorization), leading to the lower costs of the factorization compared to BLU.

Our implementation of the distributed-memory LLU is an extension of the shared-memory HLU (as described in Section 4.3). In other words, the trailing submatrix is updated and recompressed block by block (and not tile by tile) using the panel. Alternatively, we could factorize the matrix lattice by lattice (e.g., first factorize the whole diagonal lattice, then broadcast the diagonal lattice to compute the off-diagonal lattices, and finally broadcast the off-diagonal lattices to update the remaining lattices). It is also possible to update the matrix using blocks (broadcast the whole block only after all the updates and triangular solves are applied instead of sending a part of the block belonging to the panel at a time). Compared to updating with blocks or lattices, our approach leads to a higher communication latency costs (the same latency cost as BLU), but it increases the parallelism and simplifies the algorithm.

6. Arithmetic complexity

In order to understand the arithmetic complexity of the lattice $\mathscr{H}$-matrix factorization, in Table 2, we show the complexities for the five computational kernels needed for the factorization.[2] With the strong admissibility of the lattice $\mathscr{H}$-matrix format, the ranks of the low-rank blocks are assumed to be independent of the problem size. For the lattice farther away from the diagonal, the sizes of the blocks in the $\mathscr{H}$-matrix format typically increase and, eventually, the lattice contains only one low-rank block. With the strong admissibility, the number of lattices in the $\mathscr{H}$-matrix format along each lattice-row or lattice-column is assumed to be independent of the matrix size n.

If the size of the lattice, l, is set to be constant, the total complexity of gemm(low-rank) will become $\mathcal{O}(n^3)$. If the lattice size is set such that $l = \mathcal{O}(\sqrt{n})o$ (as in the block size for BLR) (Amestoy et al., 2017), the leading term will again be the gemm(low-rank), which in this case has a complexity of $\mathcal{O}(n^2)$. If the lattice size is set such that $l = \mathcal{O}(n)$ (i.e. the number of lattice $\frac{n}{\ell}$ is constant and independent of n), the number of calls to all functions becomes $\mathcal{O}(1)$, and the overall complexity of the method will be reduced to $\mathcal{O}(n\log^2 n)$.[3]

7. Performance results

7.1. Experimental setups

Our test matrices come from electrostatic field simulations with perfect conductors with the shape of a sphere (see

Table 2. Arithmetic complexity of the lattice $\mathcal{H}$-matrix factorization based on the five functions: (1) `getrf` ($\mathcal{H}$) for the LU factorization of the diagonal lattices; (2) and (3) `trsm` for factorizing the off-diagonal blocks in the panels by triangular solves where `trsm`(low-rank) and `trsm`($\mathcal{H}$) are `trsm` for the lattices in the low rank and $\mathcal{H}$-matrix formats, respectively; and (4) and (5) `gemm`(low-rank) and `gemm`($\mathcal{H}$) for updating the trailing lattices.[a]

Function	Number of calls	Complexity	Total complexity
`getrf` ($\mathcal{H}$)	$\mathcal{O}(n/l)$	$\mathcal{O}(l\log^2 l)$	$\mathcal{O}((n/l)*(l\log^2 l))$
`trsm` (low-rank)	$\mathcal{O}((n/l)^2)$	$\mathcal{O}(l)$	$\mathcal{O}((n/l)^2*l)$
`trsm` ($\mathcal{H}$)	$\mathcal{O}(n/l)$	$\mathcal{O}(l\log l)$	$\mathcal{O}((n/l)*(l\log l))$
`gemm` (low-rank)	$\mathcal{O}((n/l)^3)$	$\mathcal{O}(l)$	$\mathcal{O}((n/l)^3*l)$
`gemm` ($\mathcal{H}$)	$\mathcal{O}((n/l)^2)$	$\mathcal{O}(l\log^2 l)$	$\mathcal{O}((n/l)^2*(l\log^2 l))$

FLOP: floating-point operation.
[a]In the table, n is the size of the matrix A and l is the lattice size, while "# of calls" is the number of times the given function is called, "Complexity" is the arithmetic complexity of each function call, and "Total complexity" is the product of these two. With the strong admissibility of the lattice $\mathcal{H}$-matrix format, the rank r is assumed to be constant, and it does not appear in the complexity estimates. The cost of recompressing the low-rank blocks is included in `gemm` update (e.g. $\mathcal{O}(\ell r)$ FLOPs for recompression, compared with $\mathcal{O}(\ell r^2)$ FLOPs for update).

Figure 8). The surface of the conductor was divided into triangular elements, and the induced electrical charge on the surface was calculated using an indirect BEM with a single layer potential formulation and step functions as the base function. The matrix name represents its dimension. For instance, `100ts` is a matrix of dimension 10^5.

In our attempt to make a fair comparison between BLU, HLU, and LLU performance, we generated the lattice partition of the matrix from the $\mathcal{H}$-matrix partition generated by HACApK. For instance, since the $\mathcal{H}$-matrix does not use nested bases, we generate a BLR matrix from an $\mathcal{H}$-matrix by simply subdividing the off-diagonal blocks of the $\mathcal{H}$-matrix along the diagonal blocks such that they match the sizes of the corresponding diagonal blocks. For a low-rank block given by $B \approx VY^T$, we first duplicate the corresponding parts of V and Y among the blocks, and then recompress each block using ACA, as illustrated in Figure 9. After the recompression, we store these matrices, V and Y^T, in one contiguous memory for each block.

Similarly, to generate the lattice $\mathcal{H}$-matrix for LLU, starting from the top left block of the $\mathcal{H}$-matrix, we group the diagonal blocks into the diagonal lattice such that the size of each diagonal lattice is the smallest integer that is at least the specified lattice size (the diagonal blocks are not subdivided). The off-diagonal blocks of the $\mathcal{H}$-matrix are then grouped into the off-diagonal lattices by splitting them along the corresponding diagonal lattices.

All experiments were in double precision. During the factorization (so as to not introduce additional errors), we used the same threshold for ACA as that used to generate the matrix, i.e., approximately $\| B - VY^T \| \leq \tau \| B \|$. We set the threshold to be $\tau = 10^{-3}$ unless otherwise specified

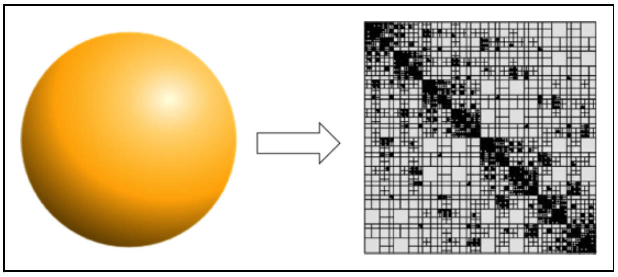


Figure 8. $\mathcal{H}$-matrix partition for `100ts`.

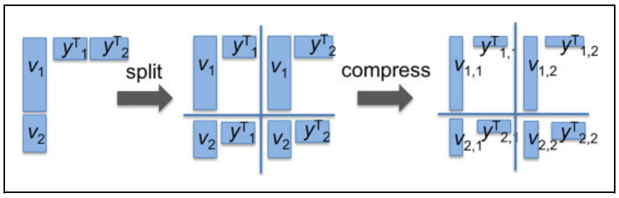


Figure 9. $\mathcal{H}$-matrix block to BLR block conversion. BLR: block low-rank.

(this threshold is used and validated for the electrostatic field simulations) (Ida et al., 2014; Tominaga et al., 2017; Ida et al., 2018a;). The right-hand side vector $\mathbf{b}$ is set such that $\mathbf{b} := A\bar{\mathbf{x}}$ with $\bar{\mathbf{x}} := \mathbf{1}$.

We conducted our experiments either on the Edison supercomputer at the National Energy Research Scientific Computing Center or on the Reedbush-L supercomputer at the University of Tokyo. Each node of Edison has two 12-core Intel Ivy Bridge CPUs at 2.4 GHz and 64 GB of main memory. These compute nodes are connected by the Cray Aries interconnect with Dragonfly topology with 23.7 TB/s of global bandwidth. Each node of Reedbush has two 18-core Intel Xeon Broadwell-EP CPUs at 2.1 GHz and 256 GB of main memory. These nodes are connected by the 4× InfiniBand EDR with 2× 100 Gbit/s.

On Edison, we complied HACApK and our solver using the Cray Fortran and C++ compilers `ftn` and `CC`, respectively, linking to Cray MPICH version 7.7.0. On Reedbush, we used Intel's MPI compiler `mpiifort` and `mpiicpc` version 18.1.163. On both systems, the code was compiled using the -O3 optimization flag and linked to sequential MKL version 2018.1.163.

7.2. Parallel performance with OpenMP

We first study the performance of the threaded BLU and HLU factorization on the shared-memory CPUs. One of the critical parameters that affects the factorization performance is the leaf size that specifies the minimum size of the diagonal blocks in the matrix (see Section 2.2).

Figure 10 illustrates the effects of the leaf size on the factorization performance. A larger leaf size tends to increase both the computational and storage costs of the BLU factorization, but it also improves the performance of the BLAS and LAPACK subroutines used for the local computation. In many cases, BLU obtained the fastest

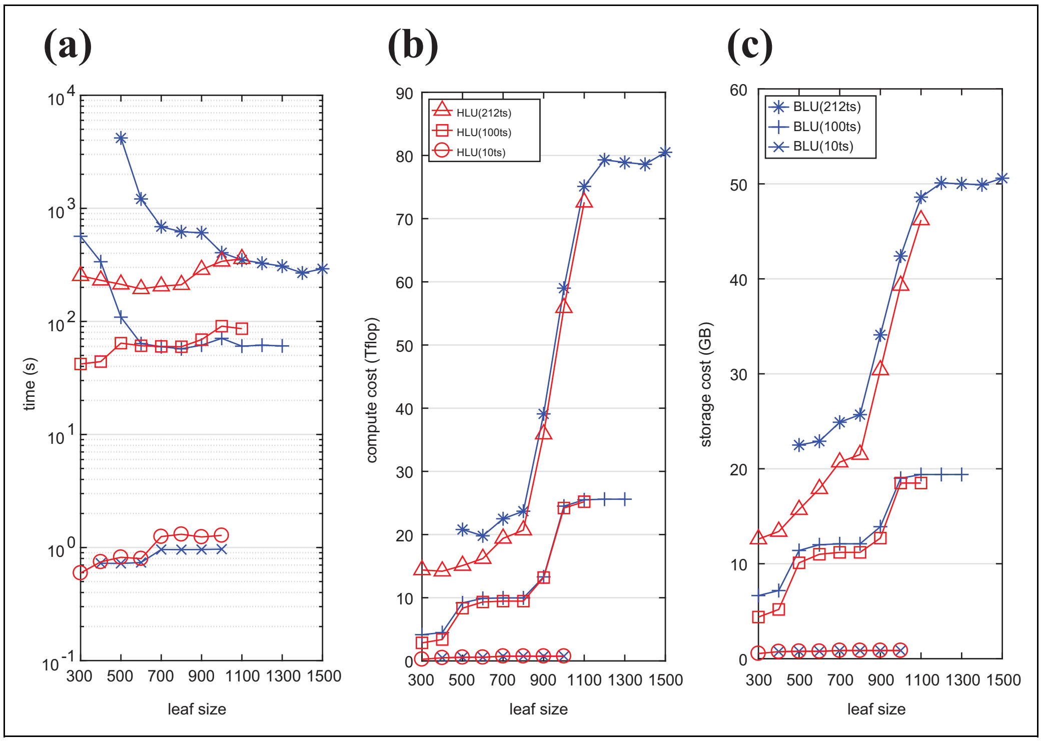


Figure 10. Effects of the leaf sizes on the factorization time, and the computational and storage costs of the factorization (on one node of Edison). Legends are shared among all three figures. (a) Time (s). (b) Compute (tetraFLOPs). (c) Storage (GB).

factorization time using a leaf size that is larger than the leaf size that obtained the minimum storage or computation. The optimal leaf size also tends to increase with the increase in the matrix dimension. Overall, for our test matrices, the leaf size of $\sqrt{5n}$ was a good choice for BLU, and that is what we use for the rest of the experiments (it has been shown that the block size of $O(\sqrt{c \cdot n})$ obtains the optimal complexity bound for BLU when c is set to be the maximum rank of the blocks) (Amestoy et al., 2018).

In contrast, HLU's factorization time was less sensitive to the leaf size. Hence, we use the fixed leaf size of 300 for the rest of the experiments. With these choices of leaf sizes (that obtain the near-optimal performance of each algorithm), BLU's leaf sizes were often larger than that of HLU. Overall, though many of the BLR blocks had lower ranks than the original $\mathscr{H}$-matrix blocks, BLR's storage cost was still higher than that of the $\mathscr{H}$-matrix.

Figure 11 shows the relative FLOP counts for different phases of the factorization. We see that the FLOP count is dominated by the trailing submatrix update that is well suited for the parallelization. Table 3 then compares the thread scalabilities of the BLU and HLU factorizations. Since we used a fixed leaf size for HLU, its scalability was lower than BLU's for small matrices. For instance, for the matrix `1ts`, there were only three diagonal blocks for HLU. On the other hand, for a large matrix, HLU was often faster than BLU due to HLU's lower computational cost. As a reference, Table 3 also shows the performance of the dense

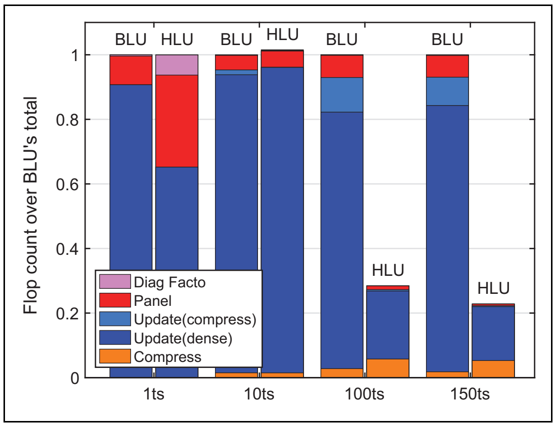


Figure 11. Breakdown of the FLOP counts needed for different phases of factorization (diagonal factorization, computation of off-diagonal blocks in the panel, updating either compressed or dense blocks, and ACA), relative to the total FLOP count for BLR. FLOP: floating-point operation; BLR: block low-rank; ACA: adaptive cross approximation.

LU factorization subroutine `dgetrf` from the threaded MKL. Although `dgetrf` scaled slightly better than BLU or HLU, BLU still obtained the speedups of 17.4× and 14.3× over `dgetrf` on 6 and 24 cores, respectively, while HLU's respective speedups were 36.5× and 28.2×.

Tables 4 and 5 compare the performance of threaded BLU and HLU for different test matrices using all available cores on the node. BLU required much greater storage, and it failed to factorize a large matrix because its storage cost

Table 3. Performance of the shared-memory factorization on Edison.[a]

	BLU			HLU			`dgetrf`		
Number of threads	`1ts`	`10ts`	`100ts`	`1ts`	`10ts`	`100ts`	`1ts`	`10ts`	`100ts`
1	0.06	16.2	892	0.07	14.4	426	34.8 (21)	—	—
6	0.03	3.0	172	0.05	2.8	82	6.8 (110)	2990 (114)	—
12	0.02	1.6	94	0.06	1.5	45	3.8 (195)	1595 (214)	—
24	0.02	0.9	63	0.06	1.0	32	2.7 (280)	902 (378)	—

BLU: block low-rank LU.
[a]The factorization time in seconds using different numbers of threads on one node of Edison. For the results with MKL's `dgetrf`, we also show the obtained gigaFLOP/s in parenthesis.

Table 4. Shared-memory BLU performance on Edison using 24 threads: "rel. res." and "rel. err." show the relative residual and error norms $10^9 \cdot ||\mathbf{b} - A\mathbf{x}||/(n||\mathbf{b}||)$ and $10^2 \cdot ||\mathbf{x} - \bar{\mathbf{x}}||/||\mathbf{x}||$, respectively, while GB (A) and GB (LU) are the respective storage costs for the original $\mathcal{H}$-matrix and its LU factors in gigabytes.[a]

	$\tau = 10^{-3}$				
	`10t`	`100t`	`150t`	`212t`	`338t`
GB (A)	0.55	12.1	25.3	—	—
GB (LU)	0.55	12.1	25.3	—	—
teraFLOPs	0.26	10.0	29.1	—	—
Time (s)	0.93	63.3	157.0	—	—
Gflop/s	279.6	158.0	185.4	—	—
rel. res.	3.3	0.35	0.23	—	—
rel. err.	0.15	0.35	0.40	—	—
	$\tau = 10^{-5}$				
	`10t`	`100t`	`150t`	`212t`	`338t`
GB (A)	0.57	12.8	26.5	—	—
GB (LU)	0.57	12.9	26.5	—	—
teraFLOPs	0.27	11.8	32.9	—	—
Time (s)	1.04	71.1	174.0	—	—
Gflop/s	259.6	166.0	189.1	—	—
rel. res.	0.06	0.02	0.01	—	—
rel. err.	.003	0.02	0.02	—	—

BLU: block low-rank LU.
[a]BLU failed for `212t` and `338t` to excessive memory.

Table 5. Shared-memory HLU performance on Edison using 24 threads (similarly to the BLU results in Table 4).

	$\tau = 10^{-3}$				
	`10t`	`100t`	`150t`	`212t`	`338t`
GB (A)	0.55	4.4	6.7	12.6	14.9
GB (LU)	0.55	4.4	6.7	12.6	15.0
teraFLOPs	0.26	2.8	6.6	14.4	27.7
Time (s)	1.02	32.5	98.0	196.0	504.0
Gflop/s	254.9	86.2	67.3	73.5	55.0
rel. res.	10.0	3.1	3.1	1.9	1.5
rel. err.	0.2	1.7	3.4	3.0	5.3
	$\tau = 10^{-5}$				
	`10t`	`100t`	`150t`	`212t`	`338t`
GB (A)	0.56	4.8	7.4	13.6	16.8
GB (LU)	0.56	4.8	7.4	13.6	16.8
teraFLOPs	0.28	4.8	12.1	25.2	56.1
Time (s)	0.92	41.3	117.0	252.0	722.0
Gflop/s	304.3	116.2	103.4	100.0	77.7
rel. res.	0.04	0.03	0.17	0.24	0.35
rel. err.	0.01	0.01	0.57	1.29	4.0

BLU: block low-rank LU.

exceeded the memory available on the node. Overall, HLU was often faster than BLU—especially for a large matrix—while using less memory.

7.3. Accuracy tests

Tables 4 and 5 also show the relative residual norms and the error norms of BLU and HLU, respectively. The residual norm tends to increase with a smaller value of the leaf size. Since for the best performance, BLU prefers a larger leaf size than HLU, BLU's residual norms were often smaller than that of HLU.

Figure 12 shows the effects of the leaf and lattice sizes on the residual norms of LLU. The residual norm tends to increase with a smaller value of the leaf size or with a larger value of lattice size. As we increase the lattice size, LLU becomes closer to HLU; LLU becomes closer to BLU as we decrease the lattice size. Overall, LLU's accuracy is between those of HLU and BLU.

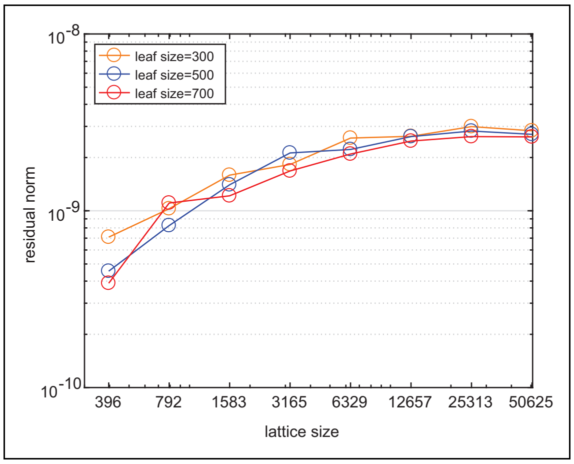


Figure 12. Effects of leaf and lattice sizes on LLU's residual norm for `100ts` matrix.

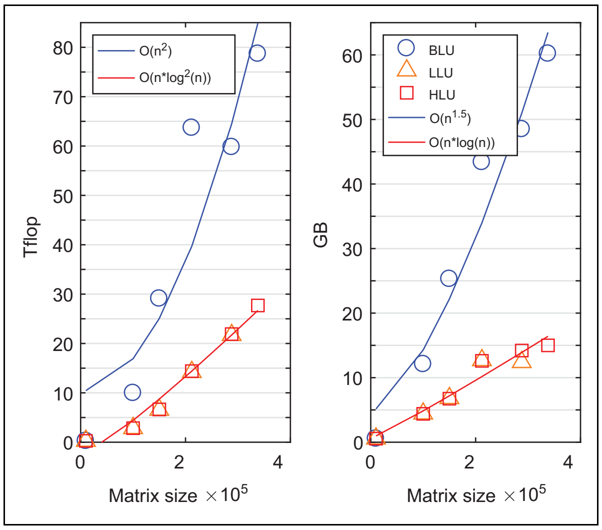


Figure 13. Computational and storage costs with varying matrix sizes (the leaf size is set to be $\sqrt{5n}$ and 300 for BLU and HLU, respectively, while for LLU, we used the leaf size of 300 and the lattice size of $n/(10\sqrt{n_p})$. The low-rank compression greatly reduces the costs of the factorization, for example, for the matrices in this figure, the dense factorization would require the computational costs of 0.7, 667, 2250, 6352, and 25, 743 tera-FLOPs, and the storage costs of 0.8, 80, 180, 360, and 914 GB. BLU: block low-rank LU.

In Section 7.5, we will show that a single Krylov iteration can reduce the residual norms of BLU, HLU, and LLU below the accuracy required by the underlying physical problems.

7.4. Computational and storage costs

Figure 13 shows the computational and storage costs of BLU and HLU for varying matrix dimension. The difference between the costs of the two algorithms tends to increase with the increase in the matrix dimension. The cost of the new LLU is between those of BLU and HLU, depending on the lattice size used. For these experiments, we used the lattice size of $n/10$. We discuss the effect of the lattice size in Section 7.5.

7.5. Parallel performance with MPI

We now compare the performance of BLU and the new lattice LU (LLU) on the distributed-memory computers. In Tables 6 and 7, we first study the effects of the lattice size with the increasing number of processes (e.g. the storage or computational cost per process). For LLU, we fixed the leaf size for the $\mathscr{H}$-matrix format used to store each of the lattices (i.e., 300). Though the factorization costs did not significantly change using different lattice sizes, the shortest factorization time was obtained using a smaller lattice size on a greater number of processes. For the remaining experiments, we use the lattice size of $\frac{n}{c\sqrt{n_p}}$ with $c = 10$, reducing the lattice size with the increasing number of processes to maintain the parallel scalability (using a smaller lattice size, the lattice $\mathscr{H}$-matrix becomes closer to BLR).

To accommodate the large storage costs of BLU, we conducted the remaining experiments on Reedbush. Figure 14 shows the load imbalance among the processes for computing BLU and LLU. Since LLU's lattice size is larger than BLU's block size, LLU had a greater load imbalance, especially with a larger process count.

One of our motivations for using OpenMP tasks is to reduce the effects of the load imbalance on LLU's or BLU's parallel scalability. Figure 15 shows the performance of LLU using three different MPI/OpenMP configurations, that is, (1) flat-MPI with one process per core, (2) MPI + OpenMP with one process per socket and one thread

Table 6. Effects of lattice size for `338ts` using 12 threads per process on Edison (block size is fixed at 300).

	Lattice size/100														
	(a) Factorization time (s)					(b) Computational cost (teraFLOPs)					(c) Storage cost (GB)				
Number of process	13	26	52	78	104	13	26	52	78	104	13	26	52	78	104
4	—	375	330	340	348	—	8.0	8.7	8.8	8.8	—	5.1	5.4	5.6	5.6
8	—	361	286	260	247	—	4.0	4.1	4.0	3.8	—	2.6	2.6	2.6	2.5
16	274	196	177	172	181	2.5	2.8	3.0	3.4	3.2	1.6	1.9	2.1	2.2	2.3
32	167	142	135	140	140	1.3	1.5	1.8	1.6	1.6	0.9	1.1	1.2	1.2	1.2

Table 7. Effects of lattice size for human4 using 12 threads per process on Edison (block size is fixed at 300).

	Lattice size/100														
	(a) Factorization time (s)					(b) Computational cost (teraFLOPs)					(c) Storage cost (GB)				
Number of process	12	24	48	72	96	12	24	48	72	96	12	24	48	72	96
4	—	358	309	310	313	—	8.1	8.2	9.3	8.8	—	4.4	4.4	4.8	4.6
8	409	269	215	207	205	4.4	4.1	4.0	4.5	4.1	2.4	2.2	2.2	2.4	2.3
16	260	187	155	147	152	2.5	2.8	2.9	3.3	3.1	1.4	1.6	1.7	1.8	1.8
32	228	160	126	117	120	1.3	1.4	1.4	1.8	1.5	0.7	0.8	0.8	1.0	0.9

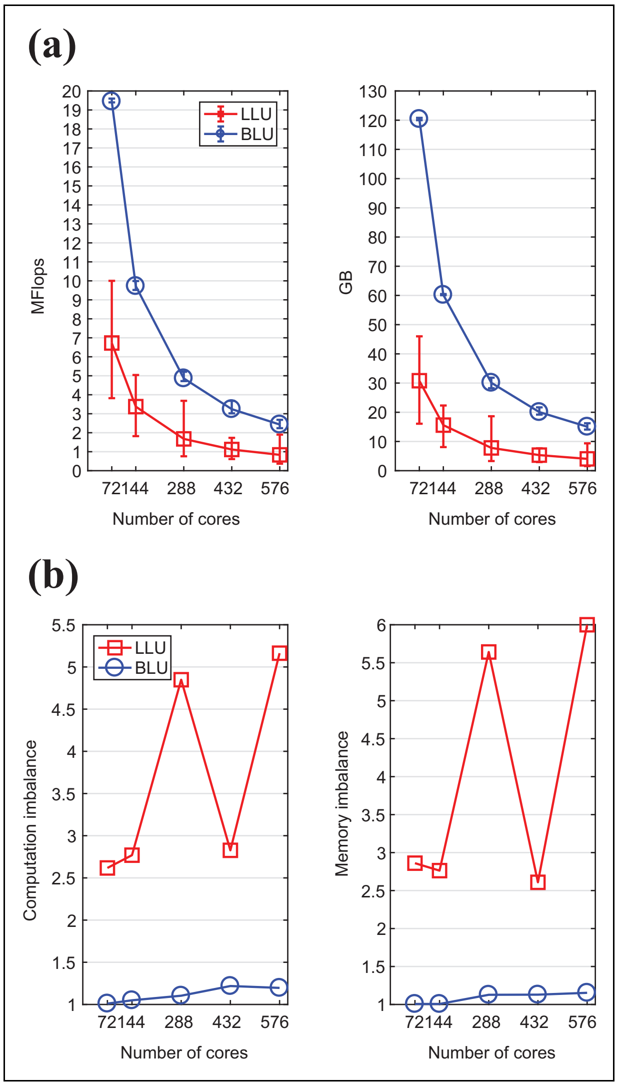


Figure 14. Strong parallel scaling for `338ts` with 18 threads per process on Reedbush. The imbalance is the ratio of the maximum and the minimum loads among the processes. (a) Average computational and storage costs. The error bars show the minimum and maximum costs among the processes. (b) Computational and storage imbalances.

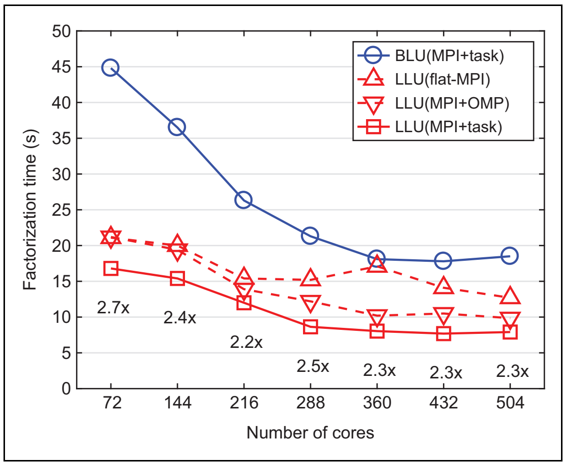


Figure 15. Effects of MPI/OpenMP configurations on the factorization time for the matrix `100ts` with 18 threads per process on Reedbush. MPI: message passing interface.

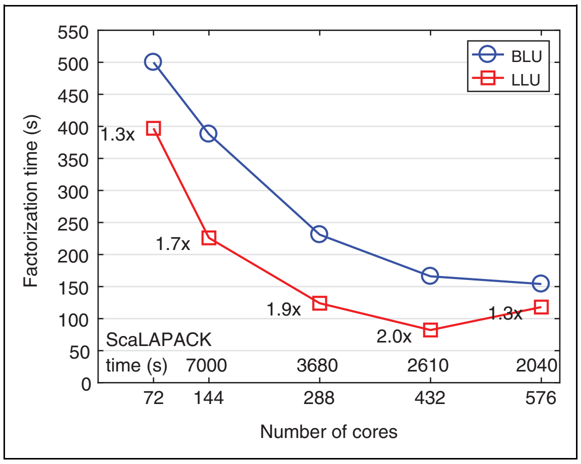


Figure 16. Strong parallel scaling for `338ts` with 18 threads per process on Reedbush. For the ScaLAPACK's `pdgetrf` runs, we launched one process per core. ScaLAPACK: Scalable Linear Algebra PACKage.

per core but with a synchronization among the local threads before each phase of factorization, and (3) MPI + OpenMP tasks. We see that the hybrid MPI/OpenMP programming often reduces the cost of inter-process communication, performing better than the flat-MPI. The performance can be further improved using tasks that avoid artificial synchronization points (obtain better core utilization by increasing the parallelism and overlapping communication with computation). For the remaining experiments, we use OpenMP tasks for both BLU and LLU. Overall, in

Table 8. Factorization performance on Reedbush (4 processes with 18 threads per process).[a]

	BLU					LLU				
	100ts	150ts	212ts	288ts	338ts	100ts	150ts	212ts	288ts	338ts
Time (s)	45	112	263	417	505	17	70	115	334	384
Memory, total (GB)	12.1	25.3	43.4	48.5	60.2	4.5	6.9	12.9	12.6	15.4
Memory, min–max	3.0–3.0	6.3–6.4	11–11	12–12	15–15	0.8–1.4	1–2	2–5	2–5	2–6
teraFLOPs, total	10.0	29.1	61.8	59.8	77.8	2.9	6.6	14.2	21.4	26.9
teraFLOPs, min–max	2.4–2.6	7.2–7.5	15–16	14–16	19–20	0.5–0.9	1–2	2–5	3–8	4–10
# iter, time (s)	1, 2.0	1, 3.9	1, 6.4	1, 8.2	1, 10.2	1, 1.1	1, 2.8	1, 4.5	1, 8.0	1, 10.0
Relative residual norm	0.23	0.15	0.24	4.0	4.6	53.4	18.7	16.2	20.7	13.9
Relative error norm	0.04	0.04	0.11	2.8	3.9	5.8	3.1	5.2	10.2	84.5
Memory reduction/BLU	—	—	—	—	—	3.3×	3.2×	2.2×	1.5×	1.5×
Time speedup/BLU	—	—	—	—	—	2.6×	1.6×	2.2×	1.2×	1.3×

BLU: block low-rank LU; FLOP: floating-point operation.
[a]For the memory and FLOP costs (GB and teraFLOP/s), we show the minimum and maximum among the processes. The relative residual and error norms were computed as $10^{15} \cdot ||\mathbf{b} - A\mathbf{x}||/(n||\mathbf{b}||)$ and $10^7 \cdot ||\mathbf{x} - \bar{\mathbf{x}}||/||\mathbf{x}||$, respectively, after the BiCG iterations.

Figures 15 and 16, LLU's respective speedups over BLU were up to 2.7× and 2.0×.

Table 8 summarizes our findings by comparing the BLU and LLU performance for different test matrices. Compared to BLU, LLU reduces both the factorization time and the memory requirement. If BLU used the same amount of storage as LLU or used the leaf size that obtains the minimum storage or computation, then BLU's factorization time would increase significantly (see Figure 10).

Figure 10 also shows that after one iteration of the BiConjugate Gradient Stabilized (van der Vorst, 1992), the residual and error norms with BLU and LLU are both below those required by the underlying physical problems. In addition, the triangular solve with BLU is often more scalable but is more expensive (due to its higher storage cost) than that with LLU.

8. Conclusion

To factorize an $\mathscr{H}$-matrix on a distributed-memory computer, we introduced a novel *lattice* LU (LLU) factorization algorithm. The algorithm partitions the matrix into 2-D $\mathscr{H}$-submatrices called lattices and distributes the lattices in a 2-D cyclic pattern. Compared with the standard $\mathscr{H}$-matrix factorization, the lattice simplifies both the communication pattern and the parallel computation, while compared with the BLR layout, the lattice reduces both the storage and computational costs. Our experimental results demonstrate that the LLU can reduce the factorization time over BLU while using less storage.

We are working to improve the parallel performance of LLU (e.g. accumulating multiple updates before compression, using task priority for reducing idling time, and examining the potential of other runtime systems). Although the low-rank compression reduces the communication volume, communication can still be the parallel performance bottleneck. We are looking to reduce this bottleneck by integrating other techniques (e.g. 2.5-D factorization) (Solomonik and Demmel, 2011). To improve the scalability, we are also investigating the techniques to reduce load imbalance (e.g., adaptively adjusting the lattice size based on the trailing submatrix size, or selecting the appropriate processor grid to reduce the load imbalance).

To maintain the strong-scale parallel performance of LLU, we reduced the lattice size on a larger number of processes (such that LLU is never slower than BLU). However, this increases the total factorization costs as we increase the process count, and thus the strong scaling is still a challenge. Our complexity analysis shows that to obtain linear complexity, the lattice size needs to be proportional to the problem size. It may be possible to improve the strong scaling by adopting the techniques used in FMM (Yokota et al., 2014) and multigrid (Gahvari et al., 2013) such as redundantly computing the coarse blocks. Another critical parameter is the leaf size. For our experiment with LLU, we used the leaf size that obtained a good HLU performance. We are examining the effects of the lattice size on the optimal leaf size for LLU's performance.

We would like to extend our study by comparing our solvers with other existing software packages (including the distributed-memory HLU). For our performance comparison, we used the matrix partition generated by HACApK, which the existing algebraic linear solver packages may not be able to directly use. However, our layered interface allows any partition structure to be imported into our solver, and our solver can be used as an algebraic solver.

Other future work includes using the factors as a preconditioner, accelerating the factorization process using GPUs, and experiments with larger matrices, for which compared with BLU, LLU is expected to obtain the lower costs and the shorter time of factorization.

Authors' note

This work was done while the author is at the University of Tennessee, Knoxville, TN, USA. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy National Nuclear Security Administration under contract de-na0003525. This article describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the article do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

Declaration of Conflicting Interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported in part by Joint Usage/Research Center for Interdisciplinary Large-Scale Information Infrastructures and High Performance Computing Infrastructure in Japan (Project ID: jh170057); JSPS KAKENHI under grant numbers #17K19962, #17H01749, and #18H03248; the National Science Foundation under grant no 1740250; the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, under prime contract #DE-AC05-00OR22725; and UT Battelle subaward #4000152412.

ORCID iD

Ichitaro Yamazaki https://orcid.org/0000-0002-6196-2508
Rio Yokota https://orcid.org/0000-0001-7573-7873

Notes

1. We are investigating the other LAPACK SVD solvers `dgesdd` and `dgesvdx` that can compute a subset of singular values
2. The assumption about the constant rank holds for many applications of our interests, but may not hold for other applications (e.g., [Engquist and Ying, 2011]).
3. Recently, the complexity of the weak-admissibility lattice $\mathscr{H}$-matrix factorization has been independently analyzed (Amestoy et al., 2018), while we use the strong admissibility in this article.

References

Akbudak K, Ltaief H, Mikhalev A, et al. (2017) Tile low rank Cholesky factorization for climate/weather modeling applications on manycore architectures. In: *Proceedings of High Performance Computing - 32nd International Conference, ISC High Performance 2017* (eds Kunkel JM, Yokota R, Balaji P and Keyes DE), Frankfurt, Germany, 18–22 June 2017, vol 10266. pp. 22–40.

Aliaga JI, Carratalá-Sáez R, Kriemann R, et al. (2017) Task-parallel LU factorization of hierarchical matrices using OmpSs. In: *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops*, IPDPSW'17.

Amestoy P, Ashcraft C, Boiteau O, et al. (2015) Improving multifrontal methods by means of block low-rank representations. *SIAM Journal on Scientific Computing* 37: A1451–A1474.

Amestoy P, Buttari A, L 'excellent JY, et al. (2018) Bridging the gap between flat and hierarchical low-rank matrix formats: the multilevel BLR format. Technical Report hal-01774642, University of Manchester.

Amestoy P, Buttari A, L'Excellent J, et al. (2017) On the complexity of the block low-rank multifrontal factorization. *SIAM Journal on Scientific Computing* 39(4): A1710–A1740.

Bebendorf M and Rjasanow S (2003) Adaptive low-rank approximation of collocation matrices. *Computing* 70: 1–24.

Buttari A, Langou J, Kurzak J, et al. (2009) A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing* 35: 38–53.

Chandrasekaran S, Gu M and Pals T (2006) A fast ULV decomposition solver for hierarchically semiseparable representations. *SIAM Journal on Matrix Analysis and Applications* 28(3): 603–622.

Chavez G, Turkiyyah G, Zampini S, et al. (2018) Parallel accelerated cyclic reduction preconditioner for three-dimensional elliptic PDEs with variable coefficients. *Journal of Computational and Applied Mathematics* 344: 760–781.

Engquist B and Ying L (2011) Sweeping preconditioner for the Helmholtz equation: Hierarchical matrix representation. *Communications on Pure and Applied Mathematics* 64: 697–735.

Gahvari H, Gropp W, Jordan KE, et al. (2013) Systematic reduction of data movement in algebraic multigrid solvers. In: *In Proceedings of IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum.*

Greengard L and Rokhlin V (1987) A fast algorithm for particle simulations. *Journal of Computational Physics* 73(2): 325–348.

Hackbusch W (1999) A sparse matrix arithmetic based on $\mathscr{H}$-matrices, part I: introduction to $\mathscr{H}$-matrices. *Computing* 62: 89–108.

Hackbusch W, Khoromskij B and Sauter SA (2000) On $\mathscr{H}^2$-matrices. In: Bungartz H, Hoppe R and Zenger C (eds.) *Lectures on Applied Mathematics*. Berlin, Heidelberg: Springer.

Ida A (2018) Lattice $\mathscr{H}$-matrices on distributed-memory systems. In: *Proceedings of the International Parallel and Distributed Processing Symposium.*

Ida A, Ataka T, Takahashi Y, et al. (2018a) Application of improved $\mathscr{H}$-matrices in micromagnetic simulations of spin torque oscillator. *IEEE Transactions on Magnetics* 54.

Ida A, Iwashita T, Mifune T, et al. (2014) Parallel hierarchical matrices with adaptive cross approximation on symmetric multiprocessing clusters. *Journal of Information Processing* 22: 642–650.

Ida A, Nakashima H and Kawai M (2018b) Parallel hierarchical matrices with block low-rank representation on distributed memory computer systems. In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. ACM, pp. 232–240.

Iwashita T, Ida A, Mifune T, et al. (2017) Software framework for parallel BEM analyses with $\mathscr{H}$-matrices using MPI and OpenMP. In: *Proceedings of the International Conference on Computational Science*, pp. 12–14.

Kriemann R (2014) H-LU factorization on many-core systems. In: *Computing and Visualization in Science*. Berlin Heidelberg: Springer.

Kurtz S, Rain O and Rjasanow S (2002) The adaptive cross-approximation technique for the 3-D boundary-element method. *IEEE Transactions on Magnetics* 38: 421–424.

Li Y and Ying L (2016) Distributed-memory hierarchical interpolative factorization. *arXiv:1607.00346v1*.

Li Y, Yang H, Martin ER, et al. (2015) Butterfly factorization *Multiscale Modeling and Simulation* 13(2): 714–732

Liu X, Xia J and de Hoop MV (2016) Parallel randomized and matrix-free direct solvers for large structured dense linear systems. *SIAM Journal on Scientific Computing* 38(5): S508–S538.

Minden V, Ho KL, Damle A, et al. (2017) A recursive skeletonization factorization based on strong admissiblity. *Multiscale Modeling and Simulation* 15(2): 768–796.

Poulson J, Demanet L, Maxwell N, et al. (2014) A parallel butterfly algorithm. *SIAM Journal on Scientific Computing* 36(1): C49–C65.

Pouransari H, Coulier P and Darve E (2017) Fast hierarchical solvers for sparse matrices using extended sparsification and low-rank approximation. *SIAM Journal on Scientific Computing* 39(3): A797–A830.

Rouet FH, Li XS, Ghysels P, et al. (2016) A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. *ACM Transactions on Mathematical Software* 42(4): Article 27.

Solomonik E and Demmel J (2011) Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In: *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pp. 90–109.

Tominaga N, Mifune T, Ida A, et al. (2017) Application of hierarchical matrices to large-scale electromagnetic field analyses of coils wound with coated conductors. *IEEE Transactions on Applied Superconductivity* 28.

van der Vorst HA (1992) Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on Scientific and Statistical Computing* 13: 631–644.

Wang S, Li XS, Xia J, et al. (2013) Efficient scalable algorithms for solving dense linear systems with hierarchically semiseparable structures. *SIAM Journal on Scientific Computing* 35(6): C519–C544.

YarKhan A, Kurzak J, Luszczek P, et al. (2017) Porting the PLASMA numerical library to the OpenMP standard. *International Journal of Parallel Programming* 45(3): 612–633.

Yokota R, Turkiyyah G and Keyes D (2014) Communication complexity of the fast multipole method and its algebraic variants. *Supercomputing Frontiers and Innovations* 1(1): 63–84.

Yu CD, Levitt J, Reiz S, et al. (2017) Geometry-oblivious FMM for compressing dense SPD matrices. In: *Proceedings of the 2017 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*.

Yu CD, March WB, Xiao B, et al. (2016) INV-ASKIT: a parallel fast direct solver for kernel matrices. In: *Proceedings of the 2016 IEEE International Parallel and Distributed Processing Symposium*.

Author biographies

Ichitaro Yamazaki received his PhD degree in computer science from the University of California at Davis in 2008. He is currently a research scientist at the Sandia National Laboratories, where his interests lie in high-performance computing, especially for linear algebra and scientific computing. Before joining the Sandia National Laboratories, he has also worked in the Innovative Computing Laboratory at the University of Tennessee at Knoxville as a research scientist from 2011 to 2019, and in Scientific Computing Group at Lawrence Berkeley National Laboratory from 2008 to 2011, as a postdoctoral researcher.

Akihiro Ida was born in Japan in 1971. He received B.Math and M.E. degrees from Nagoya University in 1994 and 1996, respectively. In 2008, he awarded him a Ph.D. degree in mathematics. In 2000–2012, he researched and developed linear solvers at VINAS Co., Ltd. In 2012–2015, he worked as an assistant professor in the Academic Center for Computing and Media Studies, Kyoto University. He currently works as an associate professor in the Information Technology Center, The University of Tokyo. His research interests include discretization methods for integrodifferential equations, numerical linear algebra, and high-performance computing.

Rio Yokota is an associate professor at the Global Scientific Information and Computing Center, Tokyo Institute of Technology. His work focuses on hierarchical low-rank approximations and scalable deep learning. He was one of the early adapters of GPU computing, and was one of the first people to receive the ACM Gordon Bell prize using a GPU-based system in 2009.

Jack Dongarra holds an appointment at the University of Tennessee, Oak Ridge National Laboratory, and the University of Manchester. He specializes in numerical algorithms in linear algebra, parallel computing, use of advanced-computer architectures, programming methodology, and tools for parallel computers. He was awarded the IEEE Sid Fernbach Award in 2004; in 2008, he was the recipient of the first IEEE Medal of Excellence in Scalable Computing; in 2010, he was the first recipient of the SIAM Special Interest Group on Supercomputing's award for Career Achievement; in 2011, he was the recipient of the IEEE IPDPS Charles Babbage Award; and in 2013, he received the ACM/IEEE Ken Kennedy Award. He is a Fellow of the AAAS, ACM, IEEE, and SIAM and a member of the National Academy of Engineering.