Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures
Final Report for JHPCN Joint Research of FY 2018, May 2019

JH180012-NAHI

Hierarchical Low-Rank Approximation Methods on Distributed Memory and GPUs

Rio Yokota（Tokyo Institute of Technology）

Hierarchical low-rank approximation methods such as H-matrix, H^2-matrix, and HSS can compress a dense matrix with $O(N^2)$ elements into a hierarchical matrix with $O(N)$ elements. By using such compressed matrices it is possible to perform matrix-matrix multiplication, LU decomposition, and eigenvalue computation in near-linear time. They are most commonly used in boundary integral problems where the matrix to be solved is dense. Hierarchical matrices can also be applied to Schur complements that arise in sparse direct solvers, so their applicability extends to fluid, structure, and electromagnetic simulations. However, these hierarchical algorithms are rather new and highly optimized implementations do not exist at the moment. A highly optimized distributed memory GPU implementation is needed to extract the potential parallelism of these methods.

1. Basic Information

(1) Collaborating JHPCN Centers

The University of Tokyo
Information Technology Center

Tokyo Institute of Technology
Global Scientific Information and Computing Center

Hokkaido University
Information Initiative Center

Kyoto University
Academic Center for Computing and Media Studies

(2) Research Areas

- ☑ Very large-scale numerical computation
- ☐ Very large-scale data processing
- ☐ Very large capacity network technology
- ☐ Very large-scale information systems

(3) Roles of Project Members

Rio Yokota (Tokyo Institute of Technology)
Low-rank approximation using FMM and its GPU-MPI implementation

Ichitaro Yamazaki (University of Tennessee)
Development of distributed memory runtime –ParSEC and blocked BLAS library for GPU –block MAGMA

Akihiro Ida (The University of Tokyo)
Feature extension of hybrid MPI/OpenMP H-matrix code –HACApK, and its integration with ParSEC and block MAGMA

Takeshi Iwashita (Hokkaido University)
Application of HACApK to boundary integral solvers for electromagnetics, and optimization of H-matrix-vector product

Takayuki Aoki (Tokyo Institute of Technology)
Application of HACApK to Poisson solvers for multiphase flows

Satoshi Oshima (Kyushu University)
GPU implementation of HACApK and integration with MAGMA

Taku Hiraishi (Kyoto University)
Dynamic load-balancing of HACApK

Kengo Nakajima (University of Tokyo)
Extend capability of HACApK within the ppOpen-HPC framework.

Jack Dongarra (University of Tennessee)
Development of distributed memory runtime –ParSEC and blocked BLAS library for GPU –block MAGMA

2. Purpose and Significance of the Research

H-matrices can reduce the arithmetic complexity of dense matrix-multiplication and factorization from $O(N^3)$ to $O(N\log^2 N)$ but still attain high Flop/s on GPUs by making use of batched BLAS operations. Conventional fast algorithms with low arithmetic complexity such as FFT, sparse linear algebra, and multigrid methods have

low arithmetic intensity and are memory-bound on most modern architectures. Conversely, methods with high arithmetic intensity like dense linear algebra and N-body methods can remain compute-bound on modern architectures, but tend to have a high arithmetic complexity and waste many Flop/s. H-matrices have a rare combination of high arithmetic intensity and low arithmetic complexity, which makes them an interesting alternative to many existing algorithms on future architectures.

H-matrices were initially applied to boundary integral problems in electromagnetics, seismic, and fluid simulations. However, H-matrices have recently been growing in popularity in new fields that can benefit from approximate dense linear algebra operations such as machine learning and statistics/big data. The broad applicability of H-matrices makes it a worthwhile algorithm to heavily optimize on many-core and accelerator architectures.

One of the goals of this project is to facilitate the transition to the algorithm of the future, by providing a highly optimized H-matrix library that users can simply call from their existing framework. An important aspect of this approach is that our code will be optimized on CPU, GPU, and Xeon Phi, which represents the range of architectures for JHPCN platforms during the coming years.

3. Significance as a JHPCN Joint Research Project

Each member of this project has different expertise, all of which are essential for the development and verification of a high performance H-matrix library. R. Yokota is the developer of exaFMM, which is a highly scalable and GPU equipped FMM code have the same data structure as an HLRA code. A. Ida and T. Iwashita are developers of HACApK – a hybrid MPI-OpenMP implementation of the HLRA. T. Hiraishi has experience in load-balancing for distributed memory H-matrix codes. I. Yamazaki and J. Dongarra are developers of dense linear algebra libraries such as MAGMA and PLASMA. T. Aoki has expertise in large scale fluid dynamics simulations that make use of distributed memory and GPUs. S. Oshima has expertise in tuning solvers for GPUs and Xeon Phi. K. Nakajima has expertise in parallel preconditioned iterative solvers and their application in CFD with AMR. The combination of these expertise is necessary for achieving the goals mentioned above.

Furthermore, each member already has highly optimized code for each component, which gives us an advantage over other groups that are writing an H-matrix code from scratch. There are a few existing H-matrix implementations, but they are limited to shared memory and have not been ported to GPUs. To our knowledge, HACApK is the only multi-GPU H-matrix code available at the moment. This could only have been done through a JHPCN international collaboration between the experts in each area.

4. Outline of the Research Achievements up to FY2017

4.1 H-matrix-vector multiplication on multi-GPUs

The present JHPCN project “Hierarchical Low-Rank Approximation Methods on Distributed Memory and GPUs” is currently in its third year. The first year (FY2016) focused on porting the HACApK code to GPU and using it to perform the matrix-vector multiplication inside a GMRES solver.

The GPU implementation of HACApK was achieved through the use of MAGMA. Since one of the developers of MAGMA – Ichitaro Yamazaki was a collaborator in this project, the integration of MAGMA with HACApK was done in a very short amount of time. We were therefore able to have a multi-GPU version by the end of FY2016.

The performance of HACApK on multi-GPU is shown in Figure 1, where ``Comp” is the computational kernels, ``Copy” is the CUDA memory copy time, ``Comm” is the MPI communication. The MPI communication time eventually becomes the bottleneck because the mat-vec computation part takes very little time on the GPU.

During FY2017, we have made significant progress in the use of batched MAGMA with HACApK, and its multi-GPU implementation. We have validated its performance by applying it to the matrix-vector multiplication in a BiCGSTAB solver. Though the low-rank compression reduces the cost of the matrix multiply, in many cases, the BiCG’s iteration time is still dominated by this H-Mat-Vec.

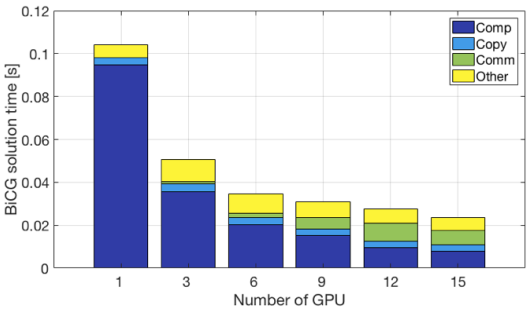


Figure 1. Performance of BiCGSTAB using HACApK for the mat-vec on multiple GPUs.

To reduce the iteration time using a distributed-memory computer, HACApK distributes the contiguous, but not disjoint, rows of the matrix among the processes. Then, each process performs HMVM with its local submatrix. With this parallelization scheme, the only required inter-process communication is the all-gather needed after HMVM to form the global vector on each process (using MPI_Allgatherv).

This parallelization scheme is motivated by two performance properties of the solver: 1) the BiCG's computation time is dominated by HMVM, while the time needed for the remaining vector operations is insignificant in the computation time and 2) the redundant computation of the vector operations avoids the global all-reduces needed to compute the six dot-products for each BICG iteration, which can be much more expensive compared with the arithmetic operations. Hence, by redundantly performing the vector operations, this parallelization scheme aims to balance out two conflicting performance factors: distributing the computation with a minimum inter-process communication.

We conducted all the experiments in double precision, and used the matrices from electrostatic field simulations with perfect conductors of two particular shapes:

- Sphere: pairs of perfect conductors with the shape of a sphere. For each pair, one sphere has its electric potential set to be 1 Volt, while the other has the electric potential of –1 Volt. We use the boundary value of 0 Volt at infinity and analyze the induced electrical charge on the surface of the spheres.
- Human: perfect conductors with the shape of a humanoid who is standing on a uniform 2D grid on a uniform electric field. The surface of the humanoid is divided into 2, 359, 680 triangular elements and the induced electrical charge on the humanoid's surface was calculated using an indirect BEM with a single layer potential formulation and step functions as the base function for the BEM.

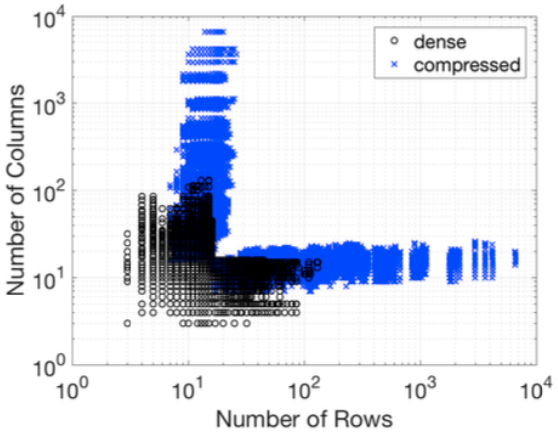


Figure 2. Block sizes in test matrix ``100ts"

Table 1 lists our test matrices. The large-scale matrices 8ms and 20ms were used for the inner-iteration to precondition the linear system. All of the compressed blocks have rank one and we fixed the number of inner iterations to be 20 for our experiments. Figure 2 shows the sizes of the blocks in the matrix 100ts. We see a wide range of the block sizes where all the blocks on the diagonal are square and dense, while off-diagonal blocks can be either dense or compressed and are either tall-skinny or wide-short. To utilize the GPU, each process divides its local dgemv tasks into several batches (e.g., a batch with a fixed number of dgemv's), and then calls dgemv_batched for each batch. dgemv_batched can execute dgemv's with variable matrix sizes in a single kernel launch. However, the performance of dgemv_batched can be much lower than its fixed-size counterpart.

Table 1. Test matrices where "compression%" is the ratio of the total number of numerical values in the compressed matrix over $n\log_2(n)$.

	Sphere objects			
name	100ts	288ts	338ts	1ms
size, n	101,250	288,000	338,000	1,004,400
compress%	16.0	16.7	16.9	17.6

	Sphere objects (precond)		Human objects		
name	8ms	20ms	hum2	hum4	hum6
size, n	7,996,800	20,736,000	78,656	314,624	707,904
compress%	1.7	1.7	17.3	22.9	31.7

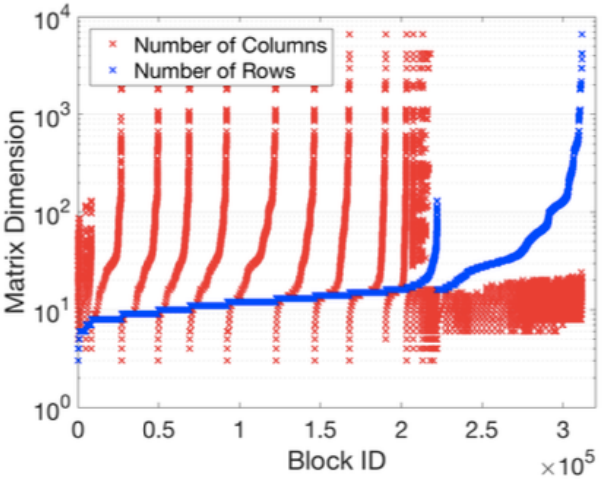


Figure 3. Matrix 100ts sorted by the number of rows

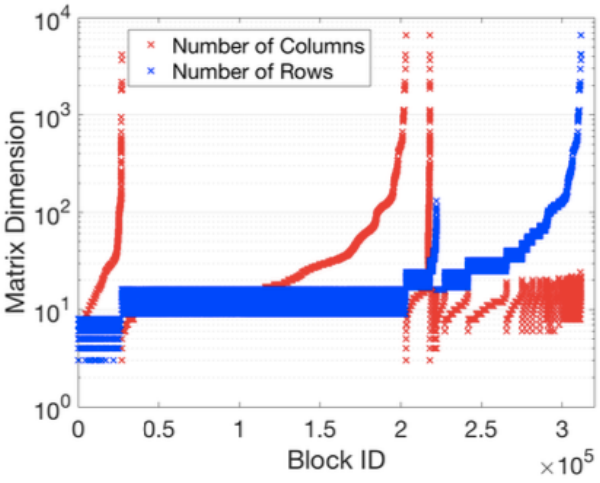


Figure 4. Matrix 100ts sorted by number of rows, and then by number of columns within group.

This is especially true when there is a wide range of matrix sizes in the single batch. In order to improve the performance of dgemv_batched, we examined several schemes to sort the blocks of A, on which dgemv's operate. In Figure 3, the blocks were sorted in the ascending order of their numbers of rows, and in Figure 4, we first grouped the blocks according to the number of rows (the k-th group contains the block with the number of rows in the range between 8(k – 1) + 1 and 8k), and then we order the blocks in the same group according to their numbers of columns.

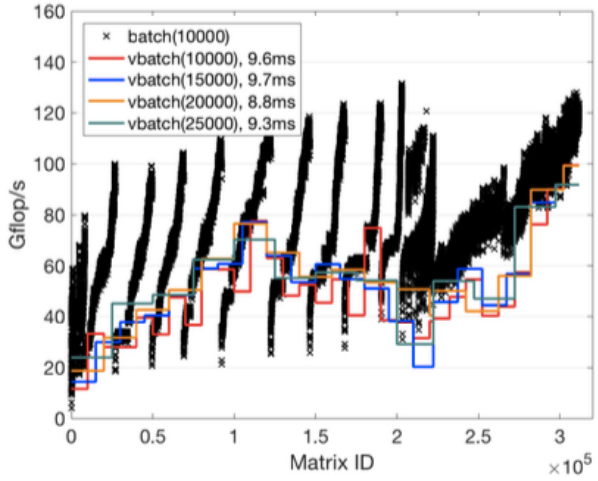


Figure 5. Sorting scheme of Figure 3 with fixed batch count.

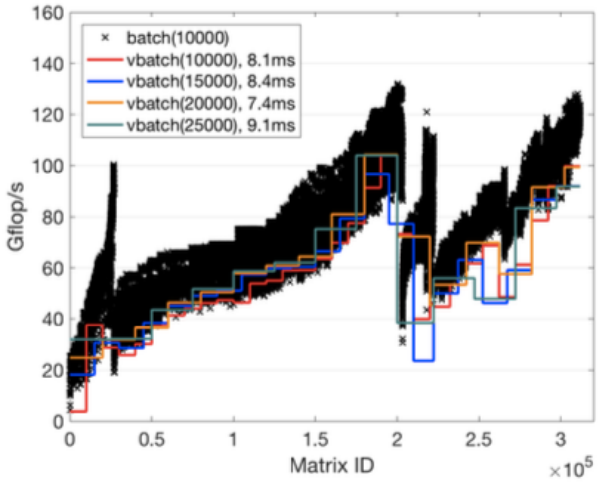
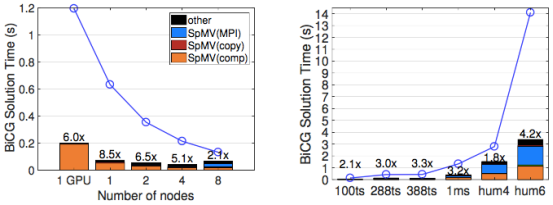


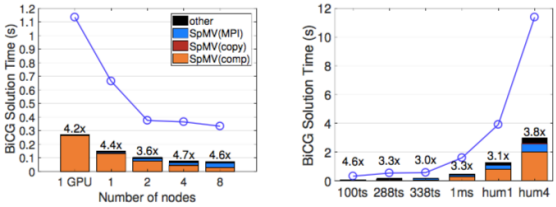
Figure 6. Sorting scheme of Figure 4 with fixed batch count.

Figures 5 and 6 show the effects of these two sorting schemes on the kernel performance for the matrix 100ts. The figure also shows the performance of the fixed-size batched kernel for each block size, which we consider as the upper bound on the performance of the variable-size kernel. We clearly see that the performance can be significantly improved by properly sorting the blocks (speedups of up to 2.5×) and the variable size kernel may obtain the performance closer to that of the fixed-size kernel.



(a) Strong scaling 100ts (b) On 8 nodes

Figure 7. Performance on Tsubame-3. The blue markers show the solution time with original HACApK without GPUs, while the bars are with the GPUs.



(a) Strong scaling 100ts (b) On 8 nodes

Figure 8. Performance on Reedbush-H. The blue markers show the solution time with original HACApK without GPUs, while the bars are with the GPUs.

Figures 7 and 8 show the effects of the GPU kernels on the BiCG performance on Tsubame-3 and Reedbush-H, respectively. For the performance without the GPUs, we bind each process to a socket and launch one OpenMP thread on each of the available cores of the socket. We found this process/thread configuration typically gives the best performance of the hybrid MPI/OpenMP implementation. With the GPUs, we launch one process per GPU (i.e., four or two processes per node on Tsubame-3 or Reedbush-H). The figures clearly show that the GPUs have reduced the iteration time significantly, obtaining the speedups of about 4.2× and 4.5× on eight nodes of Tsubame-3 and Reedbush-H, respectively (in Figures 7(b) and 8(b)).

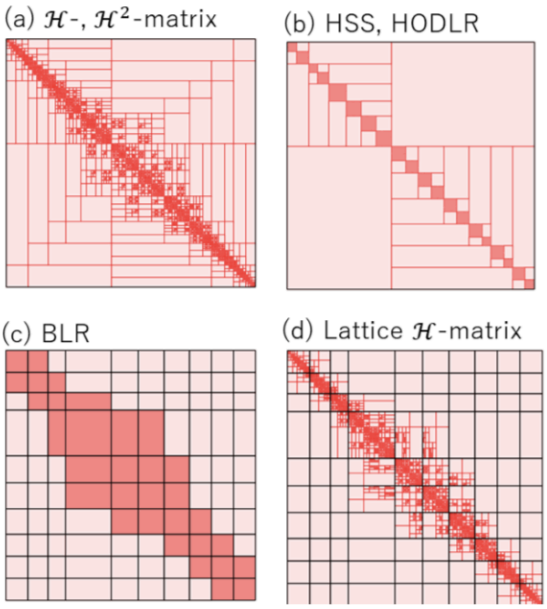


Figure 9 Illustration of popular low-rank structures generated for the same problem: (a) general H-matrix and its conversion to (b) Hierarchically Off-Diagonal Low-Rank (HODLR) layout, (c) Block Low-Rank (BLR) layout, and (d) lattice H-matrix layout. Blocks painted in deep red show dense submatrices, and blocks in light red indicate low-rank submatrices.

4.2 H-matrix LU-decomposition on distributed memory

We propose a novel method to parallelize the factorization of a hierarchical low-rank matrix (H-matrix) on distributed memory computers. H-matrix factorization is much more difficult than the parallelization of a dense matrix factorization due to the hierarchical block structure of the matrix, and it is much more difficult than the H-matrix vector multiplication due to the dataflow of the factorization. Getting rid of the hierarchy, the Block Low Rank (BLR) format not only simplifies the parallelization, but also increases concurrency. However, this comes at a price of loosing the near linear complexity of the H-matrix factorization. In the present work, we propose a "lattice H-matrix" factorization that combines the parallel scalability of BLR with the near linear complexity of H-matrix. To the extent of our knowledge, there have been no such attempts to balance the complexity and concurrency of two structured low-rank approximation methods.

We can generate any of the low-rank structures shown in Figure 9 by controlling the admissibility condition and the branch truncation of the block cluster tree during the construction of the H-matrix. Hence, all structured low-rank matrices can be regarded as special types of the H-matrices, as illustrated in Figure 9 for a given problem.

Each partitioning structure has different pros and cons. For instance, the H-matrix is the most general low-rank matrix structure, and it is an effective way of compressing the matrix with small numerical ranks. However, it usually has a complicated partitioning structure as shown in Figure 9(a), which makes it challenging to perform some matrix operations on a distributed-memory computer (e.g. LU factorization).

To improve the parallel scalability of the matrix operations, simpler partitioning structures have been proposed. For instance, by setting a weak admissibility condition, we can construct the partitioning structure shown in Figure 9 (b), which can be seen in the Hierarchical Semi-Separable (HSS) matrix or in the Hierarchically Off-Diagonal Low-Rank (HODLR) matrix. Compared with the H-matrix structure, this structure is simpler and more convenient for performing certain matrix operations on distributed-memory. 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 3D or higher dimensional problem, this structure leads to a higher asymptotic complexity due to the larger ranks of off-diagonal blocks.

Block Low Rank (BLR), shown in Figure 9(c), is another simpler matrix structure. Though the construction of the BLR matrix does not require the block cluster tree used to construct the H-matrix, it can be constructed by truncating all branches of the cluster tree at a certain depth level. The partitioning structure of BLR is then given by the induced blocks in the block cluster tree. The admissible condition is verified on each block after the structure is defined. The memory complexity of the BLR matrix is $O(n^{1.5})$ and is higher than O(nlogn) of the H-matrix. However, the BLR matrix is a simple, nonhierarchical, and effective low-rank layout, especially for the distributed-memory. In particular, the BLR structure has the block layout similar to the 2D block layout used in many dense matrix operations, and it can use many of the high-performance optimization techniques developed for the dense matrix operations.

Figure 9 (d) shows the partitioning structures of the lattice H-matrix that we use in this paper. It combines the structures of BLR with the H-matrix by introducing the lattice structure on top of the H-matrix. In other words, the lattice H-matrix utilizes the H-matrix format for each block of the BLR matrix. These lattices are then distributed among the processes in a 2D block cyclic fashion. It is designed to balance the advantages of the H and BLR matrices: the high compressibility of the H- matrix, which reduces the memory and computational costs, and the parallel scalability of the BLR matrix. Using the lattice H-matrix, the complex matrix operations originating from the H-matrix structure are performed using the threaded computational kernels.

Taken a large enough lattice size, the lattice H-matrix can reduce the memory complexity from $O(n^{1.5})$ of the BLR matrix to O(nlogn) of the H-matrix. At the same time, the lattice matrix has the same communication pattern as the BLR matrix, and it can utilize the high-performance parallel algorithms and the efficient communication schemes developed for the dense matrix operations.

We conducted our experiments on the on the Reedbush-L supercomputer at the University of Tokyo, and the TSUBAME3.0 supercomputer at the Tokyo Institute of Technology, and also the Edison supercomputer at the National Energy Research Scientific Computing Center (NERSC). We used Intel MPI compiler mpiifort and mpiicpc of version 18.1.163. On all systems, the code was compiled using the -O3 optimization flag, and linked to sequential MKL version 2018.1.163.

We first study the performance of the threaded BLR LU (BLU) and threaded H-matrix LU (HLU) factorization on the shared-memory CPUs. One of the critical parameters that affect the factorization performance is the leaf size. Hence, we first seek for an optimal leaf size that obtains the shortest factorization time.

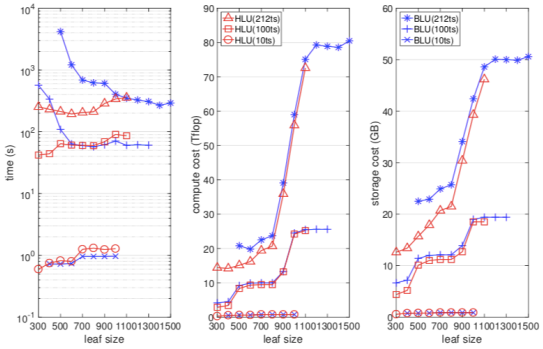
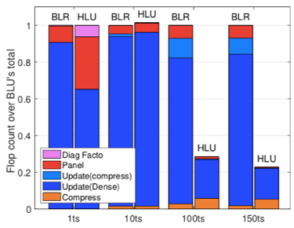


Figure 10 Effects of the leaf sizes on the factorization time, and the computational and storage costs of the factorization.

Figure 10 shows 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 factorization time using the leaf size that is larger than that obtained the minimum storage or computation. The optimal leaf size also tends to increase with the increase in the matrix dimension. In contrast, HLU's factorization time was less sensitive to the leaf size. Overall, for our test matrices, the leaf size of $O(\sqrt{kn})$ with k = 5 was a good choice for BLU, and that is what we use for the rest of the experiments.

For HLU, we use the fixed leaf size of 300. With these choices of the leaf sizes that obtain the near-optimal performance of each algorithm, BLU needed a much larger leaf size and had much higher storage and computational costs than HLU, especially for a large matrix.

Figure 11(a) 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. Figure 11(b) and 11(c) show the resulting thread scalability of the two factorization algorithms. Since we used a fixed leaf size for HLU, its scalability was lower than BLU's for a small matrix. 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.



(a) Breakdown of flop count.

# of thread	1ts	10ts	100ts
1	0.06	16.2	892
6	0.03	3.0	172
12	0.02	1.6	94
24	0.02	0.9	63

(b) Thread scalability of BLU.

# of thread	1ts	10ts	100ts
1	0.07	14.4	426
6	0.05	2.8	82
12	0.06	1.5	45
24	0.06	1.0	32

(c) Thread scalability of HLU.

Figure 11 Performance of the shared-memory factorization. Left graph shows the breakdown of the flop counts needed for different phases of factorization (diagonal factorization, computation of off-diagonal tiles in the panel, updating either compressed or dense blocks, and ACA), relative to the total flop count for BLR. Right tables shows the factorization time in seconds using different numbers of threads on one node.

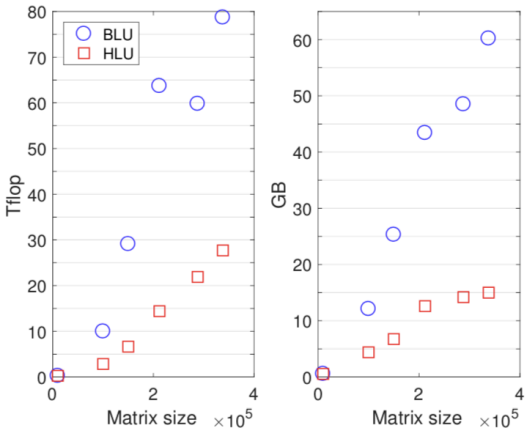


Figure 12 Computational and storage costs with varying matrix sizes. The low-rank compression greatly reduces the costs of the factorization, e.g., for the matrices in this figure, the dense factorization would require the computational costs of 0.7, 667, 2250, 6352, and 25743 Tflops, and the storage costs of 0.8, 80, 180, 360, and 914 GB.

Figure 12 visualizes 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 lattice LU is between those of BLU and HLU depending on the lattice size used.

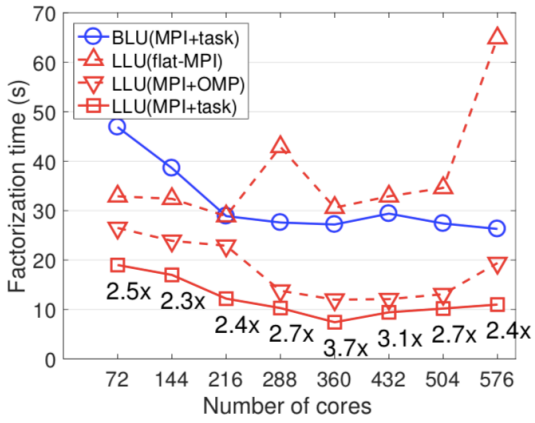


Figure 13 Effects of MPI/OpenMP configurations on the factorization time for the matrix 100ts with 18 threads per process on Reedbush-L.

We now compare the performance of BLU and the new LLU (lattice LU) on the distributed-memory computer. 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 used the fixed leaf size used for HLU (i.e., 300). As we reduce the lattice size, the H-matrix is split into smaller lattices, becoming closer to BLR. The factorization costs did not significantly change using different leaf sizes.

Figure 13 shows the performance of LLU using three different MPI/OpenMP configurations: i.e., 1) flat-MPI with one process per core, 2) MPI+OpenMP with one process per socket and one thread 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 the artificial synchronizations and obtain a better core utilization.

To accommodate the large storage costs of BLU, we conducted the remaining experiments on Reedbush-L. Figure 12(a) shows the load imbalance among the processes for computing BLU and HLU. Since HLU's lattice size is larger than BLU's tile size, HLU had a greater load imbalance, especially with a larger process count.

5. Details of FY2018 Research Achievements

During FY2018 we have tackled multiple objectives, including the use of FMM to compress H-matrices and the hybrid BLR / H-matrix format for distributed H-LU decomposition.

5.1 Using FMM for H-matrix compression

The fast multipole method (FMM) was originally developed as an algorithm to bring down the $O(N^2)$ complexity of the direct N-body problem to $O(N)$ by approximating the hierarchically decomposed far field with multipole/local expansions. In its original form, the applicability of FMM is limited to problems that have a Green's function solution, for which the multipole/local expansions can be calculated analytically. Their function is also limited to matrix-vector multiplications, in contrast to the algebraic variants that can perform matrix-matrix multiplication and factorizations. However, these restrictions no longer apply to the FMM since the kernel independent FMM does not require a Green's function. All it requires is a functional form of the operator that generates the matrix. Using this function, the KIFMM is able to form a hierarchical low-rank matrix, because the FMM is nothing but a matrix-free version of the H-matrix-vector multiplication.

A large part of the calculation time of FMM is spent on the translation of multipole expansions to local expansions (or their equivalent charges). Therefore, much work has focused on developing fast translation operators to accelerate this part of the FMM. Rotation of spherical harmonics, Block FFT, Planewaves are analytic options for fast translation operators.

These translation operators are applied to a pair of boxes in the FMM tree structure that satisfy a certain proximity threshold. This proximity is usually defined as the parent's neighbors' children that are non-neighbors. This produces a list of boxes that are far enough that the multipole/local expansion converges, but are close enough that the expansion does not converge for the their parents. Such an interaction list can contain up to $6^3 - 3^3 = 189$ source boxes for each target box. Out of these 189 boxes, the ones that are further from the target box can perform the translation operation using their parent box as the source without loss of

accuracy. There are a few variants for these techniques that reduce the interaction list size such as the level-skip M2L method and 8,4,2-box method. There are also methods that use the dual tree traversal along with the multipole acceptance criterion to construct optimal interaction lists, which automates the process of finding the optimal interaction list size.

Another technique to accelerate the translation operators is the use of variable expansion order, as proposed in the very fast multipole method (VFMM), Gaussian VFMM, optimal parameter FMM, and error controlled FMM. There are two main reasons why spatially varying the expansion order in the translation operators is beneficial. One is because not all boxes in the interaction list are of equal distance, and the boxes that are further from each other can afford to use lower expansion order, while retaining the accuracy. The other reason is because some parts of the domain may have smaller values, and the contribution from that part can afford to use lower expansion order without sacrificing the overall accuracy.

The translation operators can be stored as matrices that operate on the vector of expansion coefficients. Therefore, singular value decomposition (SVD) can be used to compress this matrix and BLAS can be used to maximize the cache utilization. Some methods use a combination of these techniques like Chebychev with SVD and planewave with adaptive cross approximation (ACA) and SVD. The use of SVD is a systematic and optimal way of achieving what the variable expansion order techniques in the previous paragraph were trying to do manually. Precomputing these translation matrices and storing them is a typical optimization technique in many FMM implementations.

One important connection to make here is that these matrices for the translation operators are precisely what H^2-matrices and HSS matrices store in the off-diagonal blocks after compression. One can think of FMM as a method that has the analytical form to generate these small matrices in the off-diagonal blocks, without relying on numerical low-rank approximation methods. To complete this analogy, we point out that the dense diagonal blocks in H^2-matrices and HSS matrices are simply storing the direct operator in FMM.

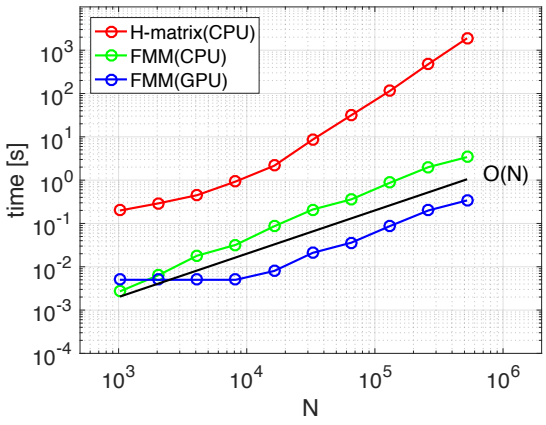


Figure 14 Calculation time for a single matrix-vector multiplication including setup time for the Green's function of a 3-D Laplace equation for H-matrix and FMM on the CPU and FMM on the GPU.

Noticing this equivalence leads to many possibilities of hybridization among the analytic and algebraic variants. Possibly the most profound is the following. Those that are familiar with FMM know that translation operators for boxes with the same relative positioning are identical. This suggests that many of the entries in the off-diagonal blocks of H^2-matrices and HSS matrices are identical. For matrices that are generated from a mesh that has a regular structure even the diagonal blocks would be identical, which is what happens in FMMs for continuous volume integrals. This leads to O(1) storage for the matrix entries at every level of the hierarchy, so the total storage cost of these hierarchical matrices could be reduced to O(logN) if the identical entries are not stored redundantly. This aspect is currently underutilized in the algebraic variants, but seems obvious from the analytic side. By making use of the translational invariance and rotational symmetry of the interaction list one can reduce the amount of storage even further. This also results in blocking techniques for better cache utilization.

In Figure 14 we show the results of the FMM compression by comparing it with a H-matrix. The FMM is calculated on both the CPU and GPU. The CPU is a 12 core Ivy Bridge (E5-2695v2) and the GPU is a P100. GPU kernel launch has a constant overhead so for small N it does not show O(N) behavior.

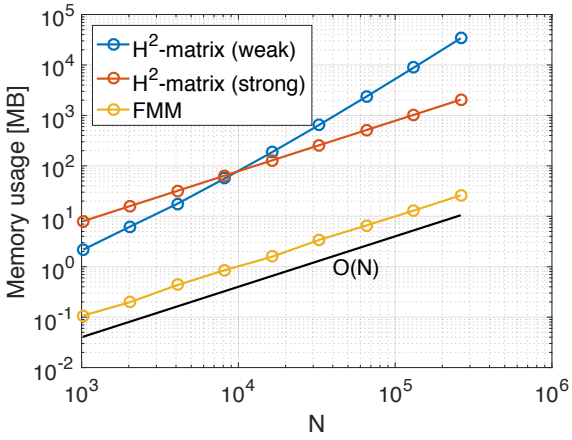


Figure 15 Memory usage of H²-matrix vs FMM

Another important aspect of using FMM for H-matrix compression is the memory usage as shown in Figure 15. The original FMM is matrix-free and therefore, consumes the least amount of memory. If the FMM naively stores all it's M2L translation matrices, it will consume the same amount of memory as the HSS. However, rotational symmetry and translational invariance of the FMM node structure can be exploited to make the M2L translation matrix have O(1) memory usages. Furthermore, if the points are positioned with fine grain homogeneity the P2P translation matrix can also be stored using O(1) memory, which means the dense blocks in the resulting H-matrix can also be stored with constant memory usage. The extension of the FMM compression to distributed memory is straightforward since we already have a MPI implementation of the FMM code.

5.2 Hybrid H-matrix-BLR format (lattice H-matrix) for LU decomposition

We propose a novel method to parallelize the factorization of a hierarchical low-rank matrix (H-matrix) on the distributed-memory computers. By getting rid of the hierarchy, the Block Low Rank (BLR) format not only simplifies the parallelization, but also increases concurrency. However, this comes at a price of losing the near linear complexity of the H-matrix factorization. In the present work, we propose a "lattice H-matrix" format that generalizes the BLR format by storing each of the blocks,

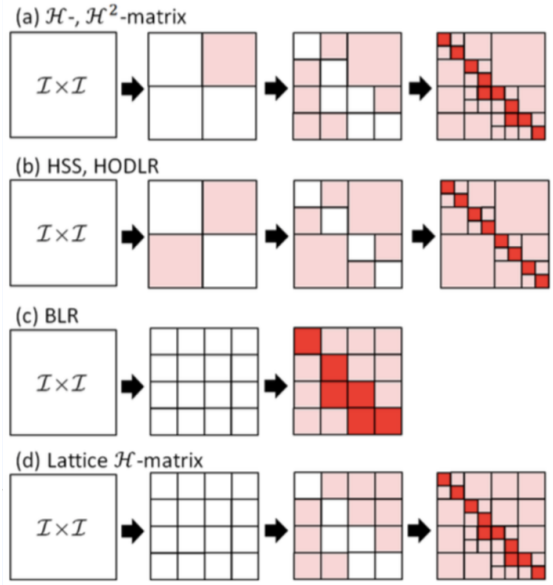


Figure 16 Various hierarchical low-rank structures and the lattice H-matrix approach

referred to as lattices, in the H-matrix format. Hence, this new format aims to combine the parallel scalability of BLR with the near linear complexity of H-matrices.

Figure 16 illustrates how a different low-rank structured matrix can be generated from a different partition structure for a given problem. Each partition structure has its own pros and cons. For instance, the H-matrix is the most general low-rank matrix structure, and it is an effective way of compressing the matrix with small numerical ranks. However, the H-matrix partition structure usually has the irregular structure as shown in Figure 16(a), which makes it challenging to perform some matrix operations on a distributed-memory computer (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, we can construct the partition structure, which is used to generate the Hierarchical Semi-Separable (HSS) matrix or the Hierarchically Off-Diagonal Low-Rank (HODLR) matrix, and is shown in Figure 16(b). Compared with the H-matrix structure, the structure is simpler and more convenient for performing certain matrix operations on the 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 3D or higher dimensional problem, this structure leads to a higher asymptotic complexity due to the larger ranks of off-diagonal blocks.

Block Low Rank (BLR), shown in Figure 16(c), is another simpler matrix structure, which has the lattice partition structure. Though the construction of the BLR matrix does not require the block cluster tree used to construct the H-matrix, it can be constructed by truncating all branches of the cluster tree at a certain depth level. The partitioning structure of BLR is then given by the induced blocks. After the structure is defined, the admissible condition is verified on each block. The memory complexity of the BLR matrix is $O(n^{1.5})$ and is higher than O(n log n) of the H-matrix. However, the BLR matrix is a simple, nonhierarchical, and effective low-rank format, especially for the distributed-memory. For instance, the lattice structure observed in the BLR matrix is similar to the 2D block layout used in many dense matrix operations (e.g., ScaLAPACK), and it can use many of the high-performance optimization techniques developed for the dense matrix operations.

Finally, Figure 16(d) shows the hybrid partitioning structure of the new lattice H-matrix that we use in this paper. It combines the BLR's lattice structure with the H-matrix partition structure by introducing the lattice structure on top of the H-matrix structure. In other words, the lattice H-matrix utilizes the H-matrix format for each lattice block. These lattice blocks are then distributed among the processes in a 2D block cyclic fashion. It is designed to balance the advantages of the H-matrix and BLR formats: the high compressibility of the H-matrix, which reduces the memory and computational costs, and the parallel scalability of the BLR matrix. Using the lattice H-matrix, the complex matrix operations originating from the H-matrix structure are performed using the threaded computational kernels. Hence, taken a large enough lattice size, the lattice H-matrix can reduce the memory complexity from $O(n^{1.5})$ of the BLR matrix to O(n log n) of the H-matrix. At the same time, the lattice matrix has the same communication pattern as the BLR matrix, and it can utilize the high-performance parallel algorithms and the efficient communication schemes developed for the dense matrix operations.

Our current contributions can be summarized as follows:

- The new "lattice H-matrix" format combines the scalability of BLR with the near linear complexity of the H-matrix. To the best of our knowledge, the proposed format is the first attempt to balance the complexity and concurrency of these two structured low-rank formats for LU factorization. The hierarchical structure was previously embedded in a flat structure, but the previous format did not consider the balance between the complexity and concurrency.
- We compare the factorization performance using the H-matrix, BLR, and new lattice H-matrix formats under various conditions on shared and distributed-memory computers. This helps quantify the benefit of the formats and determines under what conditions which format benefits the performance. Such studies are of interests to a wide range of audiences including the solver developers and users.
- Our performance comparison includes different combinations of MPI and OpenMP thread or task configurations. For instance, the dynamic task scheduling avoids the artificial synchronization and remedies the load imbalance from having a large lattice. In contrast, without tasking, the synchronization point exposes the load imbalance at the end of each factorization phase.

6. Progress of FY2018 and Future Prospects

The four goals for FY2018 were; "Implement the FMM compression on GPUs", "Extend the FMM compression to distributed memory", "Use the hybrid H-matrix-BLR format for LU decomposition", and "Optimize the distributed memory LU decomposition using the ParSEC runtime".

We were able to achieve 4 out of the 4 objectives, although the 4th objective required us to change our strategy and the code has just been completed at the time of writing this final report. We expect to submit the results for the 4th objective to IEEE Cluster and are currently preparing

the new results.

For "Implementing the FMM compression on GPUs", we were able to finish the GPU part during the latter half of FY2018. We are currently discussing with Timo Betcke to integrate the FMM code with BEM++ — one of the most widely used open source BEM codes. BEM++ currently uses H-matrices to accelerate the computation, but we have convinced the developers of BEM++ that FMM can be much faster. This is also related to the second objective "Extend the FMM compression to distributed memory". We are discussing how to interface the distributed BEM++ with the distributed FMM code. We have decided to pass the octree between the two codes, which only requires passing the distributed (hashed) Morton key.

For "Using the hybrid H-matrix-BLR format for LU decomposition", we have achieved our goals with satisfactory performance, where the corresponding papers have been accepted to IPDPS'18 [3,4]. Further optimization of the HACApK code has been performed with advancements in the CPU implementation [5], distributed memory implementation [6], and GPU implementation [7,8].

For "Optimizing the distributed memory LU decomposition using the ParSEC runtime", we have had multiple discussions with the ParSEC developers and have decided that ParSEC cannot provide the features we need to perform LU decomposition of H-matrices. We have changed our strategy to use StarPU, which is much more user friendly and better maintained. Our StarPU implementation of LU decomposition of H-matrices is ready and we plan to submit our initial results to IEEE Cluster.

We not only developed the H-matrix library in FY2018 but also applied it to various scientific problems including electro-magnetics [1] and micro-magnetics [2]. A new and exciting application that we have extended H-matrices to is second order optimization methods in deep learning [13]. Though, we have found that exploiting the Kronecker structure and not the hierarchical low-rank structure is a more natural fit [9,10].

7. List of Publications and Presentations

(1) Journal Papers

1. N. Tominaga, T. Mifune, A. Ida, Y. Sogabe, T. Iwashita, N. Amemiya, ``Application of hierarchical matrices to large-scale electromagnetic field analyses of coils wound with coated conductors", IEEE Transactions on Applied Superconductivity, Vol. 28, No. 3, pp.1-5 (2018).
2. A. Ida, T. Ataka, T. Mifune, Y. Takahashi, T. Iwashita, A. Furuya,``Application of Improved H-matrices in Micromagnetic Simulations", IEEE Transactions on Magnetics, Vol. 54, No. 3, 2018.

(2) Conference Papers

3. I. Yamazaki, A. Abdelfattah, A. Ida, S. Ohshima, S. Tomov, R. Yokota, J. Dongarra, ``Analyzing Performance of BiCGStab with Hierarchical Matrix on GPU clusters," 32nd IEEE International Parallel & Distributed Processing Symposium, IPDPS2018, Vancouer, Canada, 21-25 May (2018).
4. A. Ida, ``Lattice H-Matrices on Distributed-Memory Systems", 32nd IEEE International Parallel & Distributed Processing Symposium (IPDPS 2018), Vancouver, Canada, 21-25 May (2018).
5. *T. Hoshino*, Akihiro Ida, *T. Hanawa*, K. Nakajima, ``Design of Parallel BEM Analyses Framework for SIMD Processors", The International Conference on Computational Science 2018 (ICCS 2018), Wuxi, China, 11-13 Jun. (2018).
6. A. Ida, ``Introduction to Lattice H-matcies on Distributed Memory Computer Systems", Proceedings of the Conference on Computational Engineering and Science, Nagoya, Japan, 6-8 Jun. (2018)
7. S. Ohshima, I. Yamazaki, A. Ida, R. Yokota, ``Software Auto-Tuning for Hierarchical Matrix Computation", Proceedings of the Conference on Computational Engineering and Science, Nagoya, Japan, 6-8 Jun. (2018)
8. S. Oshima, I. Yamazaki, A. Ida, R. Yokota, `` Optimization of Hierarchical Matrix Computation on GPU", Proceedings of the 4th Asian Conference, Lecture Notes in Computer Science, Vol. 10776, pp. 274-292, Singapore, 26-29 Mar. (2018).

(3) Oral Presentations

9. R. Yokota, ``Optimization Methods for Large Scale Distributed Deep Learning", IPAM Workshop I: Big Data Meets Large-Scale Computing, Los Angeles, USA, 24-28, Sept. (2018).
10. R. Yokota, Scaling ``Deep Learning to

Thousands of GPUs", HPC 2018, Cetraro, Italy, 2-6 Jul. (2018).

11. R. Yokota, ``Energy Conserving Fast Multipole Methods for the Calculation of Long-range Interactions", Mathematics in Action: Modeling and analysis in molecular biology and electro- physiology, Suzhou, China, 16-18 Jun. (2018).
12. A. Ida, ``Low Rank Approximation Methods Used in Hierarchical Matrices", ATAT in HPC 2018, National Cheng Kung University, Tainan, Taiwan, 27 Mar. (2018)
13. R. Yokota, Can we use Hierarchical Low-Rank Approximation for Deep Learning?, HPC Saudi 2018, Jeddah, Saudi Arabia, 12-13 Mar. (2018).

(4) Others