A Heterogeneous Hierarchical Semi-Separable Library for Heterogeneous Clusters

Isuru Fernando Sanath Jayasena
University of Moratuwa
{isuru.11, sanath}@cse.mrt.ac.lk

Milinda Fernando Hari Sundar
University of Utah
{milinda, hari}@cs.utah.edu

***Abstract*—We present a scalable distributed memory library for generating and computing using structured dense matrices, such as those produced by the boundary integral equation formulations. Such matrices are dense, but has special structure that can be exploited to obtain efficient storage and matrix-vector product evaluations and consequently the fast solution of linear systems. At the core of these methods we use is the observation that off-diagonal matrix blocks of such matrices have a low numerical rank, and that this property can be exploited in a multi-level fashion. In this work we focus on the Hierarchically Semi-Separable representation (HSS). We present algorithms for building and using HSS representations that are parallelized using MPI and CUDA to leverage state-of-the-art heterogeneous clusters. The efficiency of our methods and implementation is demonstrated on large dense matrices obtained from an boundary integral equation formulation of the Laplace equation with Dirichlet boundary conditions. We demonstrate excellent (linear) scalability on up to 128 GPUs on 128 nodes. Our codes will be made available publicly and lay the foundation for a fast direct solver for elliptic problems.**

I. Introduction

As we scale up to exascale machines, developing software that has optimal runtime–$\mathcal{O}(n)$–and is scalable–$\mathcal{O}(n/p + \log p)$–is critical for furthering scientific discovery. In addition, the ability for the software to be used in a blackbox fashion (robustness) is essential for widespread adoption and use by non-experts. We focus on parallel solvers for discrete linear systems that arise when solving elliptic partial differential equations (PDEs) numerically, primarily for two reasons: Firstly, elliptic PDEs are ubiquitous not only in engineering but several data systems as well. Secondly, algorithms based on integral equation formulations offer optimal storage and work per digit of accuracy, especially those involving complex geometries. They commonly provide favorable conditioning and spectral clustering, and the resulting linear systems can be solved in a few iterations when classical iterative methods are used. These problems in their discretized form can be represented by a linear system, $Au = f$, where A is a structured dense matrix. For discretizations based on PDEs, say using the finite element method, this matrix is similar to the inverse elliptic operator, that is again dense. The dense nature of such operators leads to a storage and matrix-vector product (`matvec`) evaluation complexity of $O(n^2)$ making it impractical for large systems. The integral equations community have tackled this problem by using efficient methods such as the Fast Multipole Method (FMM) [1] to evaluate the `matvec` in $O(n)$ time.

The complexity of the FMM algorithm and its kernel-dependent nature have limited its widespread use, in spite of its sophistication. Research research has focused on developing algebraic equivalents of the FMM by exploiting the structure of the matrices, by building hierarchical low-rank representations that reduces the complexity of storage as well as `matvec` evaluation. Instead of relying on analytic expressions for the low-rank representation of well separated blocks, hierarchical matrix based approaches rely on linear algebra techniques to estimate low-rank representations for off-diagonal blocks. This makes it fairly expensive to build such representations, but it is usually worthwhile if multiple matvecs need to be evaluated, say while solving a linear system or while solving PDE constrained optimization problems where multiple solves might need to be performed. Additionally, an approximate representation can be built more efficiently that can be used as an preconditioner to improve the convergence of iterative solvers. Therefore these methods are highly relevant and the availability of scalable and efficient implementation can have significant impact. Our primary goal is to develop a scalable library to build and use such hierarchical representation of matrices. We will release out code via `github` under an MIT license.

Our work targets large heterogeneous clusters. Meth-

ods such as hierarchical matrices are primarily needed for problems where n is extremely large, necessitating the need for large supercomputers. As our supercomputers are becoming increasingly heterogeneous, it is important for large-scale software libraries to target heterogeneity. Out hierarchical matrix library supports multi-node CPUs as well as NVidia GPUs. Will demonstrate good strong and weak scalability on the BigRed-II cluster at Indiana University across 128 nodes, each with one NVIDIA Tesla K20. To the best of our knowledge ours is the first heterogeneous implementation for hierarchical matrices.

II. Related Work

A significant basis for the work on Hierarchical matrices stems from research done within the integral equations and FMM community. The initial work focused on developing a kernel-independent variant [2] that does not require the Green's function. More recent work has focused on developing the inverse FMM operators [3]. Recent work has also focused on developing algebraic variants of the FMM, such as the $\mathcal{H}$ and $\mathcal{H}^2$ matrices [4], [5], hierarchically semi-separable (HSS) [6], and hierarchically off-diagonal low-rank (HODLR) matrices [7]. These algebraic generalizations of the FMM can perform addition, multiplication, and even factorization of dense matrices with near linear complexity. A significant work in this area has focused on developing the numerical methods and consequently limited work has been done on developing scalable software libraries. There has been significant work in parallelizing FMM and its kernel independent variants [2], [8]–[11]. There has also been more recent work on parallelization of H-matrices [12] as well as a substantial effort at developing multifrontal solvers using similar concepts [13]–[15]. The closest work to ours is [16], where they develop MPI parallel methods for computing HSS representations and computing `matvecs`. Our work extends this work by supporting heterogeneous clusters by parallelizing across multiple GPUs. The basic mathematical background of our work derives from [6] and additional details on the core algorithms for building HSS representations can be found in this paper. In the following section, we will give a brief overview of Hierarchical matrix representations, focusing on HSS.

III. Hierarchical Representations

Systems arising from the discretization of integral equations or the inverse operators of discretized elliptic differential equations, are dense but contain special structure that can be exploited to compress their representation and usage. At the core of these methods is the observation that off-diagonal matrix blocks of such matrices have a low numerical rank, and that this property can be exploited in a multi-level fashion. While the cost of storing and evaluating the matrix-vector product (`matvec`) can be $O(n^2)$ for a $n \times n$ dense matrix, such matrices can be represented in a hierarchical fashion enabling $O(n)$ or $O(n \log n)$ storage and `matvec` computation. There is considerable computational cost in building such representations, but these are appealing when dealing with multiple right-hand sides or with ill-conditioned problems. In this section, we summarize the general approach to building hierarchical representations of matrices and efficient computation of the `matvec` using the HSS representation. Since these representations rely on computing a low-rank approximation for sub-blocks of the matrix, we first present a brief overview of the approach we use for computing the low-rank decomposition of matrices.

Low rank decomposition: An essential building block for algorithms exploiting low-rank structure is the method of choice to produce approximate low rank decompositions of a matrix block B, given a desired target accuracy. The common choice in many hierarchical matrix methods is the interpolative decomposition (ID) [17]. The ID factors an $m \times n$ matrix B into a narrower *skeleton* matrix $B_c = B(:, I_c)$ of size $m \times k$, consisting of a subset of the columns of B indexed by I_c—the so called column skeleton of B—and the interpolation matrix T of size $k \times (n-k)$, representing the remaining columns of B as a linear combination of columns of B_s. Using P_s to represent a permutation matrix ordering the indices I_c first followed by the remaining, we get

$$B = B_c \, [I_k \quad T] \, P_s + E = B_c R + E \tag{1}$$

where $||E||_2 \sim \sigma_{k+1}$ vanishes as we increase k [17], I_k is the $k \times k$ identity matrix, σ_{k+1} is the $(k+1)^{th}$ eigenvalue of B and $R = [I_k \quad T] P_s$ is the *restriction* or downsampling matrix. This is essentially a compression of the matrix B controlled by some parameter ε that controls the norm of E. We can represent this compression by $[T, I_c] = \mathrm{ID}(B, \varepsilon)$. A similar compression can be performed for rows by applying the same operation to B^T to obtain $B = L B_r + E_r$, where L is an upsampling interpolation matrix. Compression in both row and columns can be obtained by a linear combination of the submatrix of B corresponding to I_r and I_c, to obtain

$$B \simeq L \ B(I_r, I_c) \ R.$$

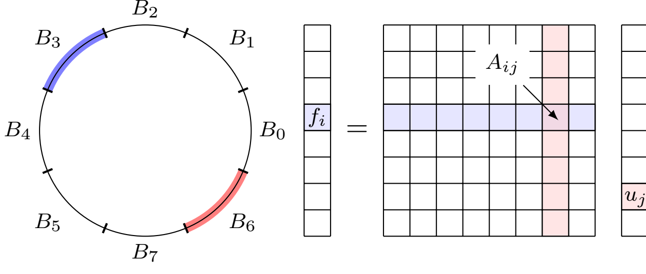

Fig. 1. Blocking of the linear system $Au = f$ into blocks (2) based on spatial bisection of the domain. For this example, we consider a circular domain (left) that is divided into 8 blocks (level $l = 3$). Consider the source samples from block B_6 shown in red and target samples from block B_3. Submatrix A_{ij} represents the interaction between these blocks.

Such a compression can be applied in a hierarchical fashion, to produce a hierarchically semi-separable matrix.

Hierarchically Semi Separable structure: The rows and columns of our matrices correspond to spatial discretization points from either the differential or integral equation formulation. These can be recursively bisected into a hierarchy of regions B_i corresponding to the nodes of a tree $\mathcal{T}$. Several algorithms exist for doing these efficiently in parallel [18]–[21]. Let $\mathcal{L}$ be the set of boxes B at a given level of this tree. This lends to a blocking of our discretized system, $Au = f$, as

$$\sum_{j \in \mathcal{L}} A_{ij} u_j = f_i, \qquad i \in \mathcal{L}, \tag{2}$$

where A_{ij} is the sub-block corresponding to source samples in B_j and target samples in B_i, and the vectors u and f are partitioned accordingly. for clarity this is illustrated in Figure 1

For a given target accuracy ε, a semi separable representation of A can be obtained by building an approximation for every off-diagonal block A_{ij} at a given level, $(i \neq j)$ of size $m_i \times m_j$ of the form:

$$A_{ij} = \underbrace{L_i}_{m_i \times k_i} \underbrace{M_{ij}}_{k_i \times k_j} \underbrace{R_j}_{k_j \times m_j}, \tag{3}$$

This is obtained by computing the ID for the set of interactions of all boxes B_i with all other boxes B_j except itself, i.e., all but the diagonal blocks A_{ii} of A. Here L_i and R_j are interpolation operators, and M_{ij} is a sub-block of A_{ij} corresponding to row and column skeletons. Representing the diagonal block using $D_i = A_{ii}$, we can write the block factorization of A for level d as,

$$A = D^{(d)} + L^{(d)} M^{(d-1)} R^{(d)} \tag{4}$$

Multi level structure: The key property that allows dense matrix operations to be performed with less than $O(N^2)$ complexity is that the low-rank structure in (3) can be exploited recursively in the sense that the matrix $A^{(d-1)} := M^{(d-1)}$ in (4) itself is block-separable. Specifically, we re-block the matrix $A^{(d-1)}$ by merging 2×2 sets of blocks from children nodes on the tree $\mathcal{T}$ to form new larger blocks. The resulting matrix with larger blocks is then itself semi-separable and admits a factorization and compression. This is illustrated in Figure 2 and can be recursively applied.

We say A is hierarchically semi-separable (HSS) if the process of blocking and factorization can be continued through all levels of the tree. In other words, we assume that $A^{(i)} = D^{(i)} + L^{(i)} A^{(i-1)} R^{(i)}$ for $i = d \ldots 1$, or, more explicitly,

$$A^{(d)} = D^{(d)} + L^{(d)} \overbrace{D^{(d-1)} + L^{(d-1)} \overbrace{D^{(d-2)} + L^{(d-2)} \overbrace{\ddots \overbrace{D^{(1)} + L^{(1)} A^{(0)} R^{(1)}}}^{} R^{(d-2)}}^{} R^{(d-1)}}^{} R^{(d)}, \tag{5}$$

where d is the level to which the matrix was initially blocked. This representation is called a telescoping factorization and is is common to other hierarchical matrix formats, such as those produced by the Fast Multipole Method [1] (FMM) and $\mathcal{H}$ matrices [4], and may be used to obtain and analyze fast arithmetic algorithms.

Storage Cost of HSS: The dense matrix A requires $O(n^2)$ storage. The telescoping representation presented requires only $O(n)$ storage. This is easy to see as at each level, we only need to store D, L and R. Assuming a block size of k, we can see that the storage for D is $nk = O(n)$. Similarly, L and R are basically block matrices requiring $O(n)$ storage. At the lower levels, we have smaller problems and without loss of generality, if we assume that M is $n/2 \times n/2$, then the total storage $S(n)$ is

$$S(n) = O(n) + S(n/2) = O(n).$$

Matrix-Vector product using HSS: Recall that a `matvec` with the original matrix A would require $O(n^2)$ operations making it infeasible for large n. In addition to reducing the storage costs to $O(n)$, the HSS representation also allows us to compute a `matvec` with $O(n)$ complexity. Given the telescoping matrix factorization (5), we can perform the `matvec` with $O(n)$ complexity

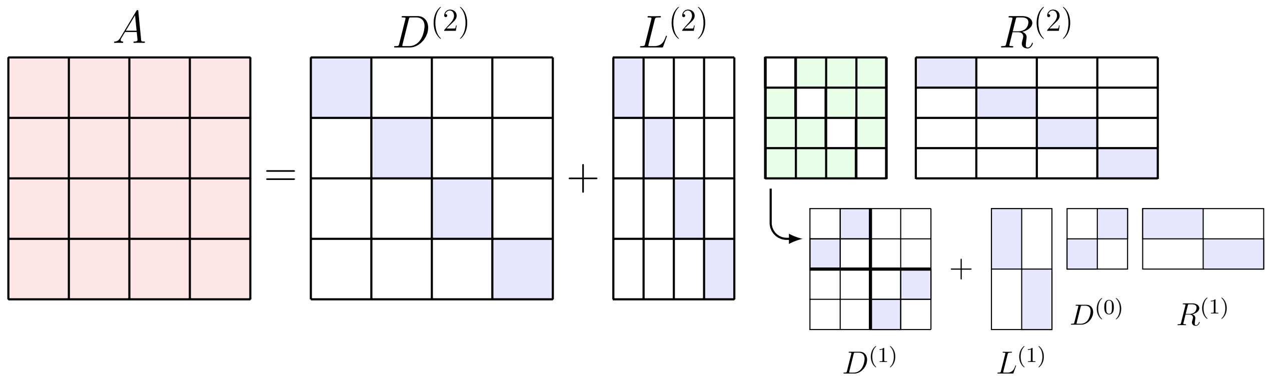

Fig. 2. *Two levels of block-separable compression: blocks of M corresponding to children are merged and then off-diagonal interactions are further compressed. The original dense matrix A is shown in red. The telescoping decomposition via HSS is shown to the right. Note that only the blocks in blue are stored.*

by interpreting the application of the block-diagonal factors in two stages. These stages correspond to an upward and downward traversal of the tree $\mathcal{T}$.

1) *Upward pass:* In the upward pass, we start at the leaf blocks of $\mathcal{T}$ and move up the tree, computing equivalent source terms $\hat{u}_i$ using the downsampling interpolation operators R_i at the skeleton sets and by merging the data between sibling boxes

$$\hat{u}^{(l)} = R^{(l+1)}\hat{u}^{(l+1)}, \tag{6}$$

where $\hat{u}^{(d)} = u$ is the input vector with which we wish to compute the `matvec`. Note that the overall complexity of this step is $O(n)$ as this involves multiplication with a block diagonal matrix, R and merging of a constant number of blocks.

2) *Downward pass:* In the downward pass, we start at the root of the tree $\mathcal{T}$ and proceed to deeper levels. We apply the dense diagonal blocks D_i (basically a block `matvec`) to obtain the results $\hat{f}_i^l$ followed by interpolation to the deeper level using L_i that maps data from the parent to the child index sets.

$$\hat{f}^{(l)} = D^{(l)}\hat{u}^{(l)} + L^{(l)}\hat{f}^{(l-1)}, \tag{7}$$

where $\hat{f}^{(d)} = f$ is the desired output vector. Note that the complexity of this step is again $O(n)$ requiring multiplication with two block-diagonal matrices, D and L and one addition of length n vectors.

IV. Parallel HSS

A. Parallelization across a GPU

We first describe our parallelization of the HSS factorization and `matvec` computation on a single GPU. Our single GPU experiments were performed on the latest Nvidia Pascal architecture; *e.g.* Titan X GPU mentioned in V-A has 3096 cores running at 1.08 GHz and 12 GB of memory. NVIDIA GPU cards support Single instruction, multiple threads (SIMT) execution model [22] and also supports CUDA framework [22] for general purpose computing on the GPU. Since NVIDIA GPUs support SIMT, it's important that all of the threads do the same work (same intruction) on different data at the same time. CUDA kernel functions give control to the programmer on how many threads to use, how to use shared memory and how the computation should be done.

1) Factorization: We divide the input matrix tree into diagonal and off-diagonal blocks using the tree $\mathcal{T}$ [6]. This approach can be generalized into non-complete binary trees by ignoring nodes in a level where the node has no sibling. Since the computations done on each node of the same level are independent of each other, this is relatively simple to parallelize across a GPU. The fact that these computations are the same on different data fits the SIMT execution model of NVIDIA GPUs.

To compute the HSS factorization on the GPU, first the matrix is randomized by multiplying it with a random matrix generated using CURAND [23] and multiplied using the CUBLAS library. Then an upward pass on the tree $\mathcal{T}$ creates the HSS representation. On each node, matrix-multiplications and interpolative decompositions (ID) are computed. In our ID implementation, we first randomize the matrix, compute a pivoted QR factorization using householder transformations and subsequently perform a triangular solve to obtain the components of the interpolative decomposition.

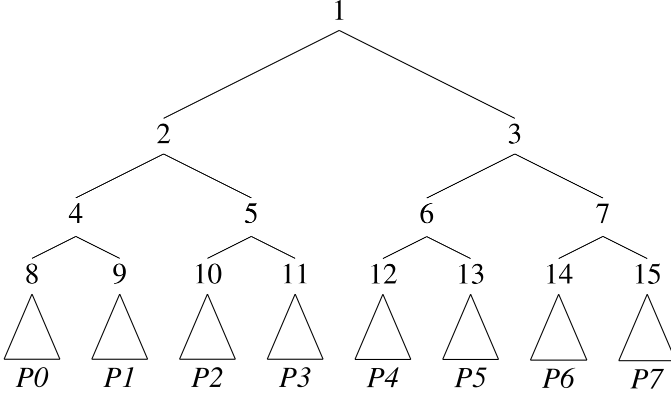

Fig. 3. HSS tree with the subtrees given to 8 processors

2) Matrix-vector multiplication: Matrix-vector multiplication using a HSS matrix is performed using an up and a down sweep as explained earlier. During the upward pass (6), we perform a set of blockwise `matvecs` and in the downward pass (7), we perform a set of blockwise `matvecs` and additions. These matrices are of the size of the upper bound k of the low rank of the off-diagonal block. The CUBLAS library [24] provides routines to perform BLAS operations on the GPU in a batched mode. Based on our experiments, for matrices smaller than 64, a specialized CUDA kernel function performs better than CUBLAS and was used for small matrix sizes. For smaller matrices, CUDA kernel functions assigned each block on the grid to a node from $\mathcal{T}$ for computation and each block utilized k CUDA threads to compute the products. For matrix-matrix multiplication, where only the first matrix is a HSS matrix, each block is assigned a vector from the second matrix and a node in the HSS tree $\mathcal{T}$.

B. Parallelization across nodes

We now descibe our strategy for parallizing the HSS computation and matvec across nodes. For a distributed memory system, data needs to be communicated amongst nodes to do the computation. CUDA aware MPI implementations, such as MPICH [25], are efficient when communication needs to happen between GPU buffers. For intra-node communication, GPU buffers are copied directly from one GPU to another using NVIDIA's GPUDirect for P2P technology. For inter-node communication, GPU buffers can be directly sent from the GPU to the network adaptor without going through host memory.

1) Factorization: Using the observation that a subtree of the HSS tree can be computed with only the data

Algorithm 1: Factorization of HSS matrix

input : Matrix A of size $N \times N$ divided into chunks of rows and distributed among $p = 2^q$ processors
An upper bound for the HSS-rank k of A.
A tree T on the index vector $[1, 2, ..., N]$.
output: Matrices $L_\tau, R_\tau, D_{\nu_1,\nu_2}$ that form an HSS factorization of A.

Let I_p be the index vector for the pth node on level q of the tree and A_p be the chunk of rows of A in processor p
Generate a random seed on rank 0 and broadcast it.
Generate $N \times (k + 10)$ Gaussian random matrix (Ωr) on all processors using the seed;
Generate an $(N/p) \times (k + 10)$ Gaussian random matrix (Ωc_p) on each processor p
begin
  $Sr_p = A_p \cdot \Omega r$
  $Sc_p = A_p \cdot \Omega c_p$
  Reduce Sc_p using sum operator across all processors and scatter
  $\Omega r_p = \Omega r(I_p, :)$
  $\widetilde{A} = A_p(:, I_p)$
  $h = I_p(0)$
  `/* Calculation on the GPU */`
  p^{th} processor will process all nodes in the subtree rooted at the pth node of level q
  for $l = L, L - 1, ..., q$ **do**
    Calculate $L_\tau^{row}, L_\tau^{col}, D_{\nu_1,\nu_2}, D_{\nu_2,\nu_1}$ as in original algorithm using $Sr_p, Sc_p, \Omega r_p, \Omega c_p, \widetilde{A}$ which contains the parts of $Sr, Sc, \Omega r, \Omega c, A$ needed.
  end
  `/* Calculation in the CPU */`
  for $l = q - 1, q - 2, ..., 1$ **do**
    ν_2 sends $\widetilde{I}_{\nu_2}, \Omega_{\nu_2}^{row}, S_{\nu_2}^{row}, \Omega_{\nu_2}^{col}, S_{\nu_2}^{col}$ to ν_1
    ν_1 receives D_{ν_1,ν_2} and D_{ν_2,ν_1} from the other processors.
    Calculate $L_\tau^{row}, L_\tau^{col}, D_{\nu_1,\nu_2}, D_{\nu_2,\nu_1}$
    Processor processing ν_1 will process τ in next level
  end
end

corresponding to the diagonal block of the subtree, the computation can be done in a distributed system with each node having a GPU. In other words all diagonal blocks are initially distributed across the GPUs. This is illustrated in Figure IV-B1.

When distributing the matrix, it is important to distribute $A(I_r, I_r)$ to node r to reduce communication. Therefore, the matrix is partitioned by rows, each node getting a block of rows. Recall that we need to compute the ID for the matrix A as well as A^T, to obtain I_r and I_c. This involves computing random projections with the matrix A. Different blocks need to be multiplied by the same set of random matrices (for both the rows and columns). In order to avoid exchanging data, we only exchange the seed for the pseudorandom number generator and generate (the same) random matrix on all nodes. Then each node computes the random projection by multiplying its matrix block with the generated random matrix. As mentioned in the single GPU case, we compute a pivoted QR factorization on this randomized projection to get the indices $[I_r, I_c]$ and the interpolation matrices L and R according to (1).

Once the factorization is done for the subtree (that is stored on a single GPU), the rest of the computations in the upward pass is performed on the CPU. While it would be faster to do these on the GPUs, it is not worth it due to the data movement costs. The main task is to merge the blocks, therefore the left child of a node is promoted as the parent and the right child sends the data for the computation. This proceeds in a reduction-tree like fashion. The overall algorithm for computing the HSS factorization on a heterogeneous cluster is given in Algorithm 1.

2) Matrix-vector multiplication: For matrix multiplication a similar approach was used. For the subtrees corresponding to the blocks given to each node, local computation was done on the GPU. During the upward pass, merging of the blocks were done on the left child and sent up the HSS tree. During the downward pass, vectors generated by the parent are passed to the children until the subtrees corresponding to the blocks of the nodes are reached. The remainder of the downward sweep is performed on the GPU and the resulting vector is a distributed vector across all nodes, just as the input vector. Note that the overall algorithm involves minimal communication, primarily up and down the tree and is therefore very efficient. The overall algorithm for computing the `matvec`, given the HSS factorization, on a heterogeneous cluster is given in Algorithm 2.

Algorithm 2: Distributed HSS matrix-vector multiplication

input : Matrices $L_\tau, R_\tau, D_{\nu_1,\nu_2}$ that form an HSS factorization of A.
Vector x divided into same sized chunks as A was and distributed among the processors
output: Vector $y = A \cdot x$ distributed among the processors

```
begin
    /* Calculation on the GPU     */
    p^th processor will process all nodes in the
      subtree rooted at the p^th node of level q
    for l = L, L-1, ..., q do
        Calculate x̃_τ
    end
    /* Calculation in the CPU     */
    for l = q-1, q-2, ..., 1 do
        ν2 sends x̃_τ to ν1
        Calculate x̃_τ on ν1
        Processor processing ν1 will process τ in
          next level
    end
    /* Calculation on the CPU     */
    for l = 1, 2, ..., q-1 do
        Calculate b̃_ν1
        Calculate b̃_ν2 and send it to ν2
    end
    /* Calculation in the GPU     */
    for l = q, q+1, ..., L-1 do
        Calculate b̃_ν1 and b̃_ν2
    end
    Calculate b(I_τ)
end
```

V. Results

A. Experimental Setup

In this section, we present the experimental setup that we used to carry out experiments described in section V-B. We have used 3 different setups to perform our experiments.

- **single node** : All the single node computations (i.e. Using only 1 GPU), were performed on a Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz with a NVIDIA GeForce GTX TITAN X with 3584 CUDA cores.
- **Indiana's BIGRED2** : Big Red II features a hybrid architecture based on two Cray, Inc., supercom-

puter platforms. Big Red II comprised 344 XE6 (CPU-only) compute nodes and 676 XK7 "GPU-accelerated" compute nodes, all connected through Cray's Gemini scalable interconnect, providing a total of 1,020 compute nodes, 21,824 processor cores (with combined CPU and GPU cores), and 43,648 GB of RAM. Each XE6 node has two AMD Opteron 16-core Abu Dhabi x86_64 CPUs and 64 GB of RAM; each XK7 node has one AMD Opteron 16-core Interlagos x86_64 CPU, 32 GB of RAM, and one NVIDIA Tesla K20 GPU accelerator.
- **Utah's KINGSPEAK**: KINGSPEAK consists of 48 dual socket Intel Xeon (Sandybridge/Ivybridge E5-2670 and Haswell) processors, and addition for CPU nodes consists of 8 Nvidia Tesla P100 GPUs with 4.7T flops each.

B. Results

In this section, we present a detailed description of experiments that we have performed and the results presented in this paper. We mainly focus on the execution time of HSS factorization, performing MATVEC & MATMAT multiplication operations based on computed HSS factorizations using single as well as multiple GPUs across multiple nodes. We also present a comparison of distributed MATVEC & MATMAT multiplication operations for the same input matrices performed without HSS factorization.

1) Input matrices for HSS *Factorization:* In general, there are two main constraints that needed to be satisfied by a given matrix A in order for A to be hierarchically semi-separable. Those constraints can be listed as, off-diagonal blocks of matrix A needs to be rank deficient, and we should be able to compute the interpolative decomposition of off-diagonal blocks hierarchically. These constraints need to be considered when we generate input matrices to test the scalability of heterogeneous implementation of HSS factorization. We generated our input matrices A based on the integral equation formulation for Laplace equation on a unit sphere with Dirichlet boundary conditions.

We present factorization results using a single GPU (see Figure 4), and weak scaling in two GPU architectures (i.e Indiana's BIGRED2& Utah's KINGSPEAK) in Figures 6 & 9.

2) MATVEC *&* MATMAT *operations:* Numerical schemes such as finite difference & finite element methods used to solve Partial Differential Equations (PDEs) will result in large linear systems, which are solved by

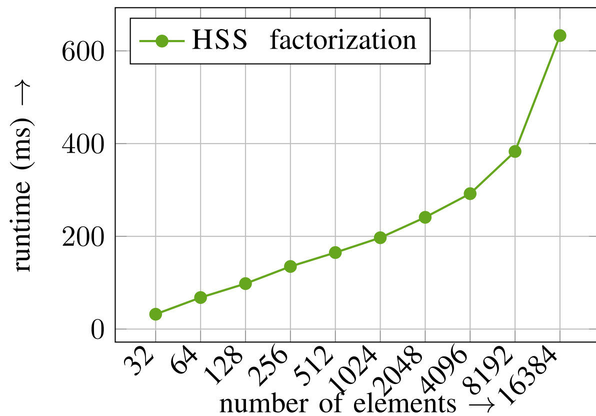

Fig. 4. Runtime to perform the HSS factorization using a 1 GPU (Nvidia Titan X), with varying matrix sizes(see Table I).

iterative schemes such as CG or GMRES. When using iterative methods to solve linear systems we need to compute a large number of MATVEC operations until the convergence. Since the overall runtime of iterative solvers, largely dominated by the time complexity of the MATVEC or MATMAT operations we present results to perform above-mentioned operations, using HSS factors of the input matrix. The key point it since we need to perform large number of MATVEC or MATMAT operations, it might be efficient to perform the HSS factorization once, and use it to perform matrix operations such as MATVEC in $\mathcal{O}(n)$ time.

Table I shows the runtime for HSS factorization & MATVEC multiplication. HSS factorization is linear and HSS based MATVEC is sub-linear. As expected, dense format MATVEC is quadratic as the algorithm in CUBLAS is $O(N^2)$. HSS based MATVEC is better than a linear algorithm which means that the GPU is not utilized fully for smaller matrix sizes and as matrix sizes grow it is utilized better leading to increase in performance. This can be confirmed by looking at timings for MATMAT multiplication in table I. Since the matrix has 100 columns, parallelism is multiplied 100-fold and makes the multiplication routine utilize the GPU more leading to a linear graph.

We present results for performing MATVEC & MATMAToperations using both HSS based, and dense format base for both single GPU (see Figure 5), and weak scaling in two different GPU architectures similar to factorization results in Figures 7 , 8 & 10. Note the we had to limit weak scalability results for Nvidia Tesla p100 at 8 GPUs due lack of availability.

matrix size	HSS time(ms)	MATVEC (HSS) time(ms)	MATVEC (dense) time (ms)	MATMAT (HSS) time(ms)	MATMAT (dense) time (ms)
32	32	0.065	0.161	0.066	0.164
64	68	0.066	0.055	0.092	0.171
128	98	0.127	0.184	0.177	0.308
256	135	0.128	0.315	0.227	0.393
512	165	0.152	0.368	0.507	0.68
1024	197	0.178	0.644	0.76	2.124
2048	241	0.218	0.798	1.214	7.635
4096	292	0.257	2.278	2.06	32.888
8192	383	0.291	8.004	3.825	91.844
16384	633	0.352	25.139	7.534	365.964

TABLE I

RUNTIME FOR HSS FACTORIZATION FOR A GIVEN INPUT MATRIX RANK 16 APPROXIMATION FOR OFF-DIAGONAL BLOCKS AND PERFORMING, MATVEC & MATMAT OPERATIONS BASED ON HSS FACTORS AND USING THE DENSE REPRESENTATION OF THE INPUT MATRICES, USING A SINGLE GPU(NVIDIA GEFORCE GTX TITAN X).

matrix size	# GPUs	HSS time(ms)	MATVEC (HSS) time(ms)	MATVEC (dense) time (ms)
1024	1	181.189	0.313	0.894
2048	2	187.564	0.335	0.824
4096	4	195.951	0.336	0.719
8192	8	214.312	0.343	1.012
16384	16	245.603	0.358	1.818
32768	32	308.729	0.362	2.932
65536	64	439.41	0.366	5.176
131072	128	700.293	0.382	9.574

TABLE II

WEAK SCALING RESULTS FOR HSS FACTORIZATION, MATVEC MATMAT OPERATIONS, FOR BOTH HSS BASED AND DENSE REPRESENTATION BASED, IN INDIANA'S BIGRED2WITH A GRAIN SIZE OF $1024 = 32 \times 32$ PER EACH PROCESS.

matrix size	# GPUs	HSS time(ms)	MATVEC (HSS) time(ms)	MATVEC (dense) time (ms)
1024	1	40.98	0.164	0.75
2048	2	43.2	0.185	1.313
4096	4	49.997	0.253	0.839
8192	8	58.11	0.227	0.895

TABLE III

WEAK SCALING RESULTS FOR HSS FACTORIZATION, MATVEC MATMAT OPERATIONS, FOR BOTH HSS BASED AND DENSE REPRESENTATION BASED, IN UTAH'S KINGSPEAK NVIDIA TESLA P100 GPUS WITH A GRAIN SIZE OF $1024 = 32 \times 32$ PER EACH PROCESS.

VI. CONCLUSION

We presented a scalable heterogeneous library for computing hierarchical semi-separable factorization of dense matrices arising from integral equation formulation of elliptic operators. The use of HSS representation allows us to reduce the complexity of storage as well as the evaluation of the matrix-vector product from $O(n^2)$ to $O(n)$. We demonstrated good weak scalability of our implementation. We will make our code available freely so that other researchers can benefit from our implementation. For future work, we are currently working on reducing the cost of building the HSS factorization and integrating with FMM methods so that the HSS representation can be built directly using FMM. We would also like to use this work to develop a fast direct solver, that takes a sparse forward operator, such as those produced by the finite element method and generate the HSS representation for the inverse operator.

VII. ACKOWLEDGEMENTS

This work was supported in part by the National Science Foundation grants ACI-1464244 and CCF-1643056.

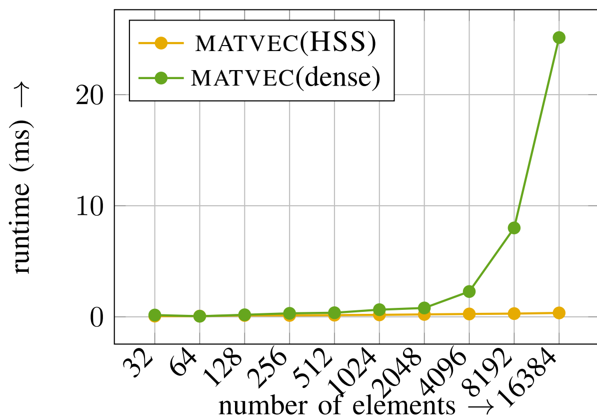

Fig. 5. Runtime to perform a single MATVEC operation using HSS factorized version, compared against dense MATVEC in Indiana's BIGRED2, using 1 GPU with varying matrix sizes (see Table I).

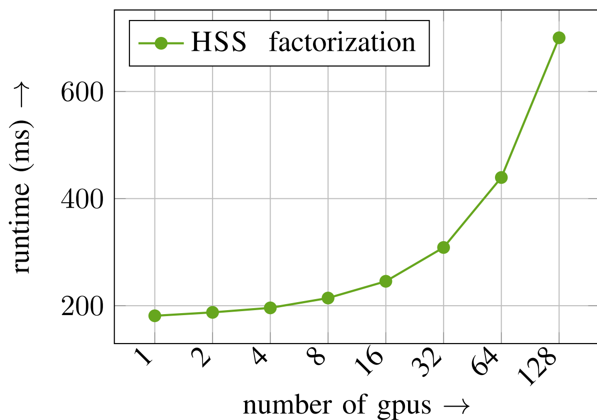

Fig. 6. Weak scaling results for the HSS factorization, using up to 128 GPUs in Indiana's BIGRED2 with a grain size of $1024 (32 \times 32)$ block size, per GPU (node) for a full rank matrix, with off diagonal block ranks are bounded by 16.Note that, in the largest run, we compute HSS factorization of a matrix with 131072 elements in $0.7s$ using 128 GPUs (see Table II).

This work is supported in part by the Senate Research Committee (SRC) Grant from University of Moratuwa. This research was supported in part by Lilly Endowment, Inc., through its support for the Indiana University Pervasive Technology Institute, and in part by the Indiana METACyt Initiative. The Indiana METACyt Initiative at IU is also supported in part by Lilly Endowment, Inc.

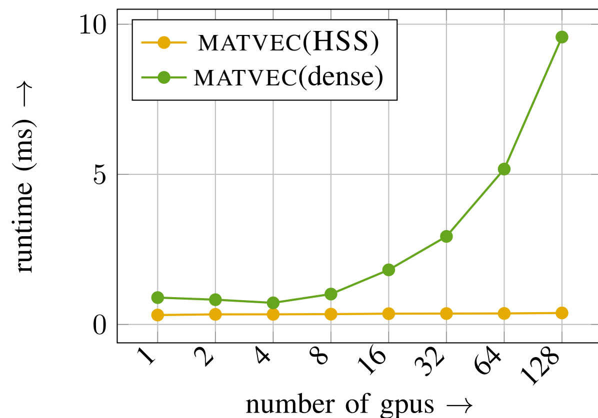

Fig. 7. Weak scaling results, for the MATVEC operation, with computed HSS factorizations, compared against distributed GPU version of dense MATVEC operation using up to 128 GPUs in Indiana's BIGRED2 . Note that the MATVEC operation performed using HSS factorizations shows good weak scaling compared to the distributed dense MATVEC operation (see Table II).

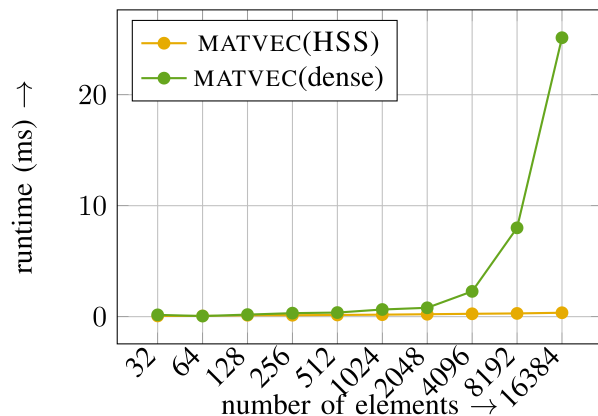

Fig. 8. Runtime to perform a matrix-matrix multiplication using HSS factorized version, compared against dense matrix-matrix multiplications in Indiana's BIGRED2 , using 1 GPU with varying matrix sizes. The second operand for the matrix-matrix multiplication has dimensions of $m \times 100$ where m is the number of columns in the first operand i.e. the matrix that we use to compute the HSS factorization (see Table II).

REFERENCES

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

[2] L. Ying, G. Biros, and D. Zorin, "A kernel independent adaptive fast multipole algorithm in two and three dimensions," *Journal of Computational Physics*, vol. 196, no. 2, pp. 591–626, 2004.

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

[4] W. Hackbusch, B. Khoromskij, and E. Tyrtyshnikov, "Hierarchical kronecker tensor-product approximations," *Journal of Numerical Mathematics jnma*, vol. 13, no. 2, pp. 119–156, 2005.

[5] S. Börm, "Construction of data-sparse $\mathcal{H}^2$ -matrices by hierarchical compression," *SIAM Journal on Scientific Computing*, vol. 31, no. 3, pp. 1820–1839, 2009.

[6] P. G. Martinsson, "A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix," *SIAM J. Matrix Anal. Appl.*, vol. 32, no. 4, pp. 1251–1274, Nov. 2011. [Online]. Available: http://dx.doi.org/10.1137/100786617

[7] S. Ambikasaran and E. Darve, "An\ mathcal o (n\ log n) fast direct solver for partial hierarchically semi-separable matrices,"

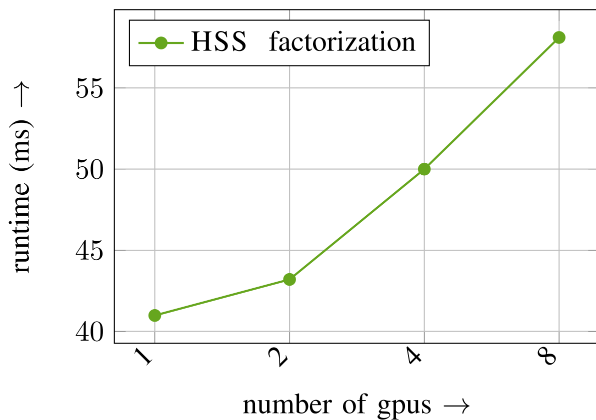

Fig. 9. Weak scaling results for the HSS factorization, using up to 8 GPUs in Utah's KINGSPEAK with a grain size of $1024(32 \times 32)$ block size, per GPU (node) for a full rank matrix, with off diagonal block ranks are bounded by 16.Note that, for this we have used Nvidia Tesla p100 GPUs (see Table III).

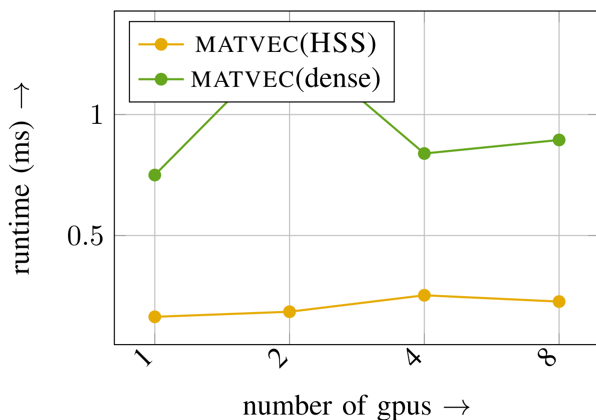

Fig. 10. Weak scaling results, for the MATVEC operation, with computed HSS factorizations, compared against distributed GPU version of dense MATVEC operation using up to 8 GPUs in Utah's KINGSPEAK with Nvidia Tesla p100 GPUs. Note that the MATVEC operation performed using HSS factorizations shows good weak scaling compared to the distributed dense MATVEC operation (see Table III).

Journal of Scientific Computing, vol. 57, no. 3, pp. 477–501, 2013.

[8] F. A. Cruz, M. G. Knepley, and L. A. Barba, "Petfmm?a dynamically load-balancing parallel fast multipole library," *International Journal for Numerical Methods in Engineering*, vol. 85, no. 4, pp. 403–428, 2011.

[9] R. Yokota and L. A. Barba, "A tuned and scalable fast multipole method as a preeminent algorithm for exascale systems," *International Journal of High Performance Computing Applications*, vol. 26, no. 4, pp. 337–346, 2012.

[10] D. Malhotra and G. Biros, "Pvfmm: A parallel kernel independent fmm for particle and volume potentials," *Communications in Computational Physics*, vol. 18, no. 03, pp. 808–830, 2015.

[11] B. Pierre, B. Bramas, O. Coulaud, E. Darve, L. Dupuy, A. Etcheverry, and G. SYLVAND, "Scalfmm: A generic parallel fast multipole library," in *Computational Science and Engineering (CSE)*, 2015.

[12] M. Bebendorf and R. Kriemann, "Fast parallel solution of boundary integral equations and related problems," *Computing and Visualization in Science*, vol. 8, no. 3-4, pp. 121–135, 2005.

[13] S. Wang, X. S. Li, J. Xia, Y. Situ, and M. V. De Hoop, "Efficient scalable algorithms for solving dense linear systems with hierarchically semiseparable structures," *SIAM Journal on Scientific Computing*, vol. 35, no. 6, pp. C519–C544, 2013.

[14] S. Wang, V. Maarten, J. Xia, and X. S. Li, "Massively parallel structured multifrontal solver for time-harmonic elastic waves in 3-d anisotropic media," *Geophysical Journal International*, vol. 191, no. 1, pp. 346–366, 2012.

[15] P. Ghysels, X. S. Li, F.-H. Rouet, S. Williams, and A. Napov, "An efficient multi-core implementation of a novel hss-structured multifrontal solver using randomized sampling," *arXiv preprint arXiv:1502.07405*, 2015.

[16] F.-H. Rouet, X. S. Li, P. Ghysels, and A. Napov, "A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization," *arXiv preprint arXiv:1503.05464*, 2015.

[17] N. Halko, P. Martinsson, and J. Tropp, "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions," *SIAM review*, vol. 53, no. 2, pp. 217–288, 2011.

[18] H. Sundar, R. S. Sampath, S. S. Adavani, C. Davatzikos, and G. Biros, "Low-constant parallel algorithms for finite element simulations using linear octrees," in *SC'07: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. ACM/IEEE, 2007.

[19] H. Sundar, R. Sampath, and G. Biros, "Bottom-up construction and 2:1 balance refinement of linear octrees in parallel," *SIAM Journal on Scientific Computing*, vol. 30, no. 5, pp. 2675–2708, 2008.

[20] R. Sampath, H. Sundar, S. S. Adavani, I. Lashuk, and G. Biros, "Dendro home page," 2008, http://www.seas.upenn.edu/csela/dendro.

[21] C. Burstedde, L. C. Wilcox, and O. Ghattas, "p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees," *SIAM Journal on Scientific Computing*, vol. 33, no. 3, pp. 1103–1133, 2011.

[22] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[23] NVIDIA, *CUDA CURAND Library*, NVIDIA Corporation, Santa Clara, CA, USA, Aug. 2010.

[24] nVidia, *CUBLAS Library User Guide*, v5.0 ed., http://docs.nvidia.com/cublas/index.html, nVidia, Oct. 2012. [Online]. Available: http://docs.nvidia.com/cublas/index.html

[25] W. Gropp, "Mpich2: A new start for mpi implementations," in *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. London, UK, UK: Springer-Verlag, 2002, pp. 7–. [Online]. Available: http://dl.acm.org/citation.cfm?id=648139.749473