Hierarchically Semi Separable Matrices on Heterogeneous GPU Clusters

Isuru Fernando and Sanath Jayasena, (University of Moratuwa),
Milinda Fernando and Hari Sundar (University of Utah)

Introduction

Matrix-vector multiplication (matvec) is at the heart of other linear algebraic operations like matrix-matrix multiplication, solving linear systems, inversion, factorization.

Doing matvec efficiently will lead to efficient algorithms for other linear algebra routines. Some ways to optimize matvec,

1. Exploit special structure of the matrix
2. Use GPU parallelism
3. Use a distributed memory system

Introduction

Matrix - vector multiplication time-complexity

Dense matrix of size N * N	$O(N^2)$
Sparse matrix with d data per row	$O(N \times d)$
Hierarchically semi-separable with rank k	$O(N \times k)$

Problem

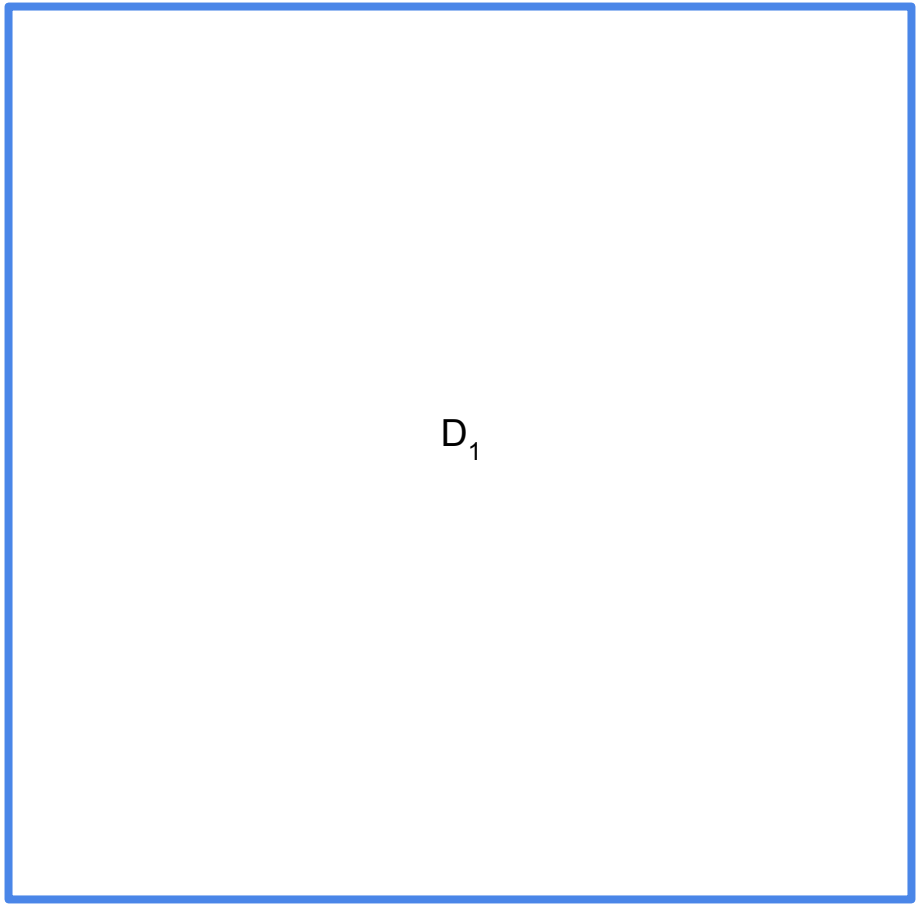
Factorizing hierarchically semi-separable matrices and performing matrix-vector multiplication as efficiently as possible.

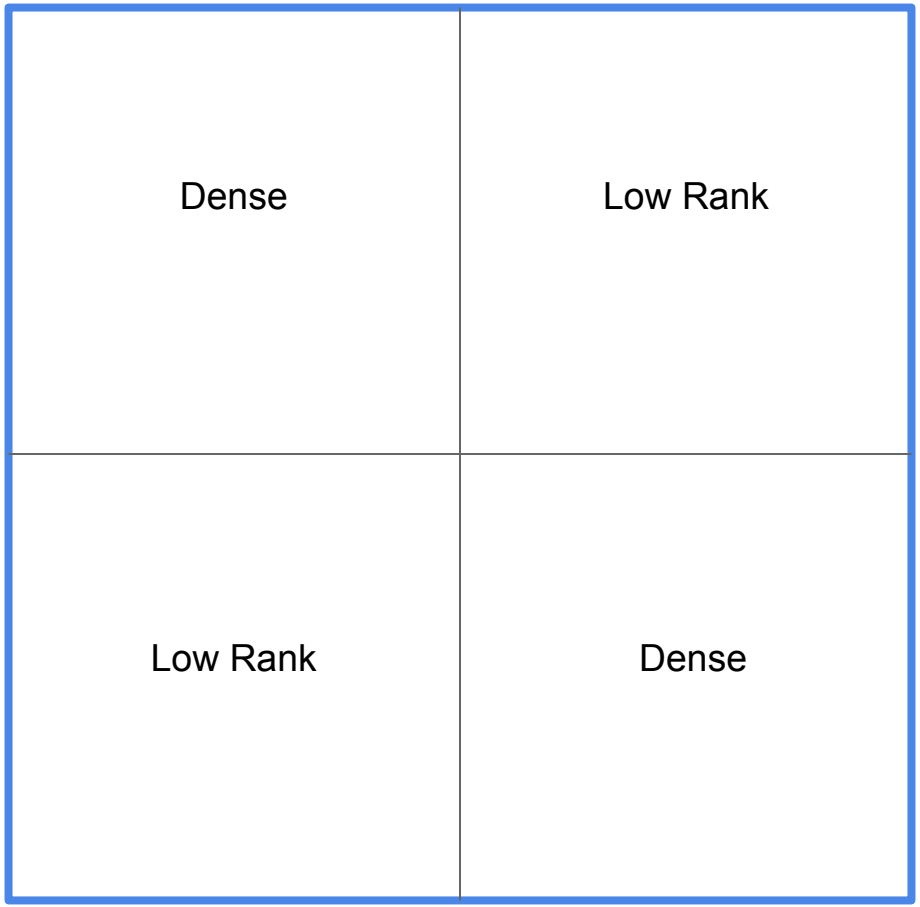
We'll look at using a distributed set of nodes with GPUs to do the computation.

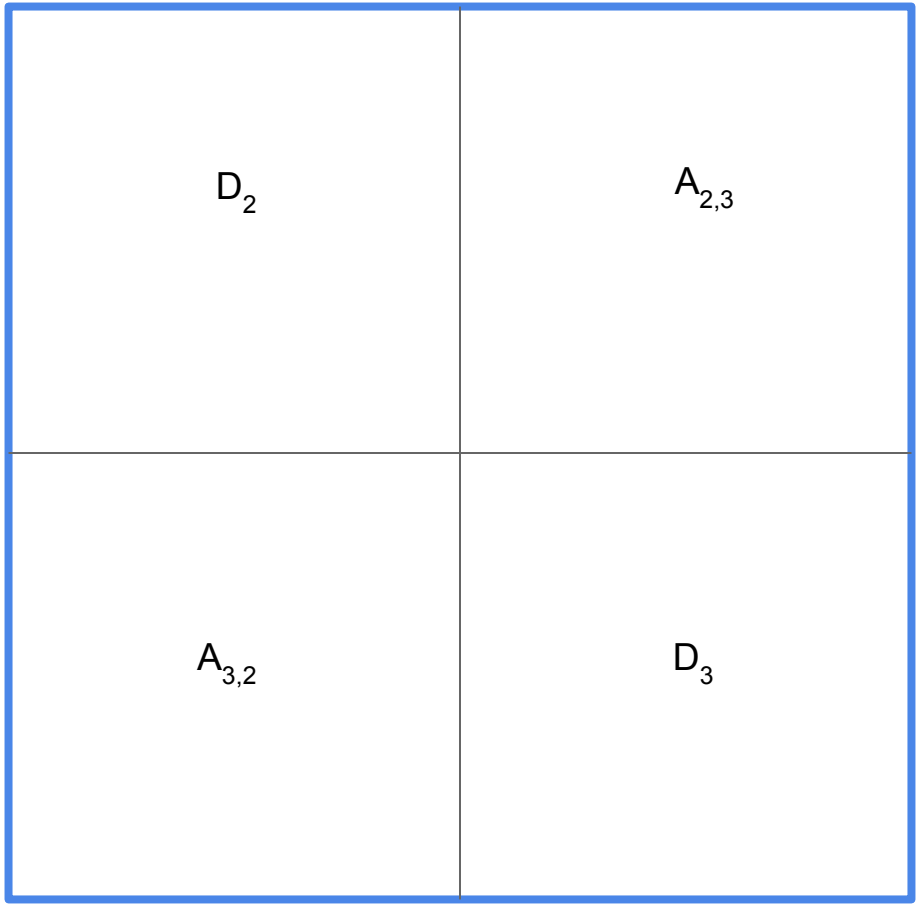
HSS Structure

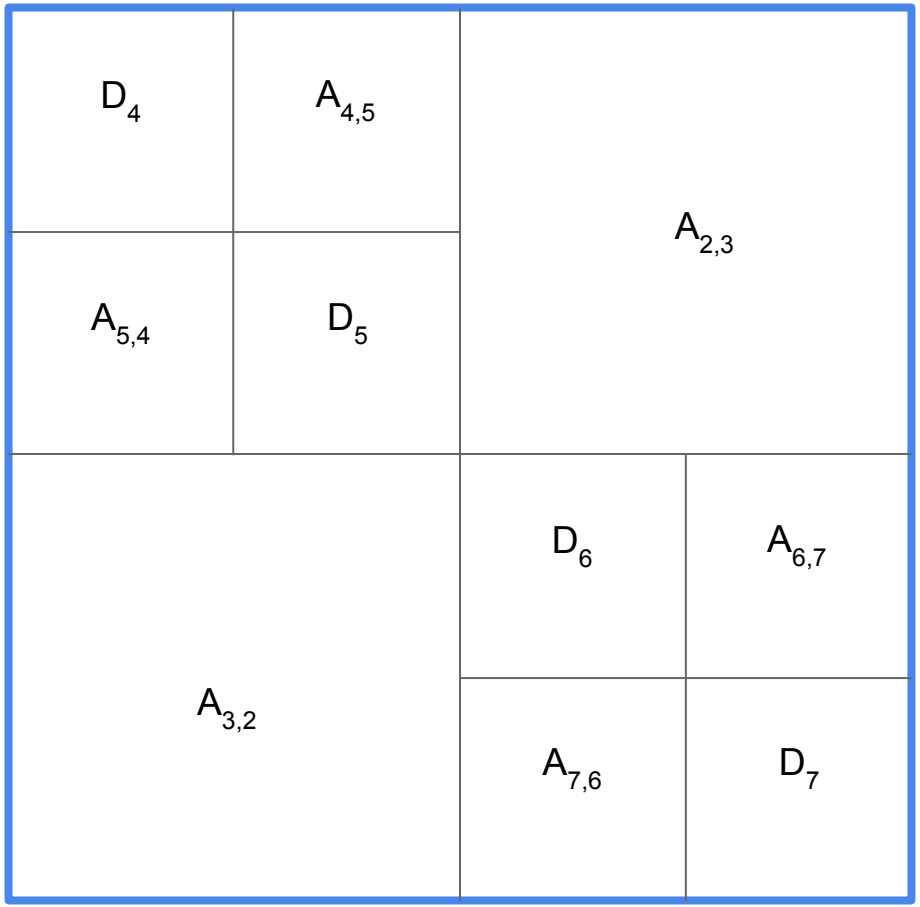
HSS matrices have off-diagonal blocks with low rank (upper bound is named k) and the off-diagonal blocks satisfy recursive relations mentioned later

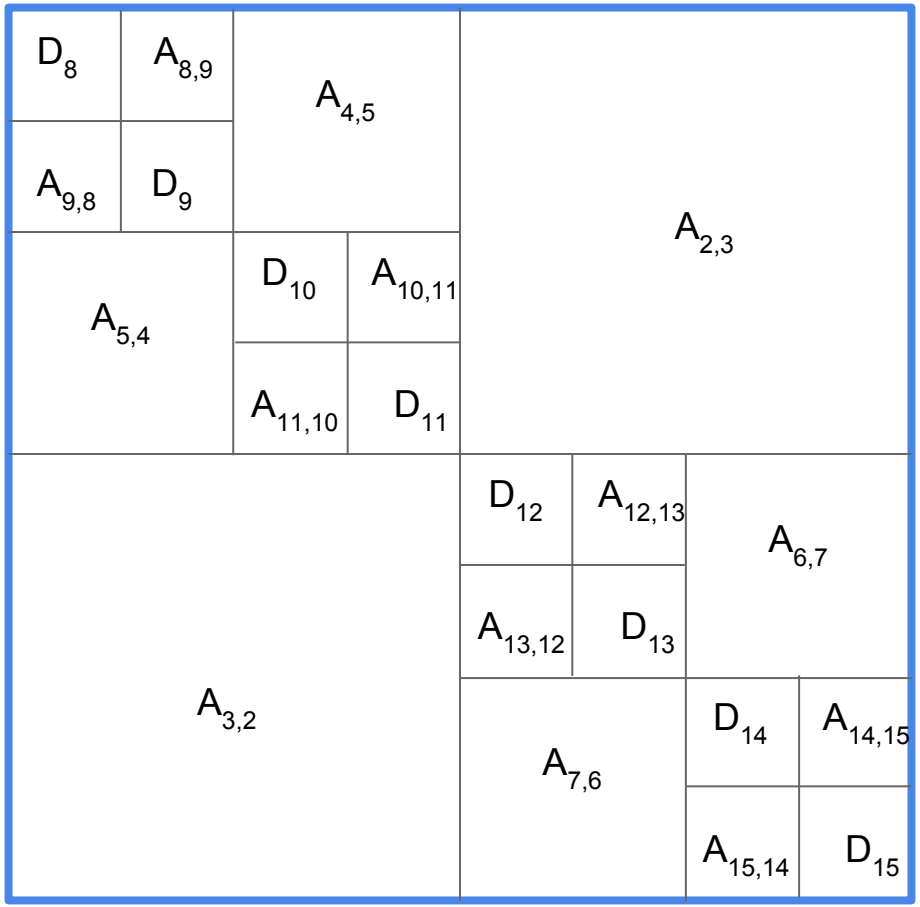
Divide the matrix into a 2 x 2 block matrix and then do the same recursively to the two diagonal blocks until the matrix is small enough.


D_1

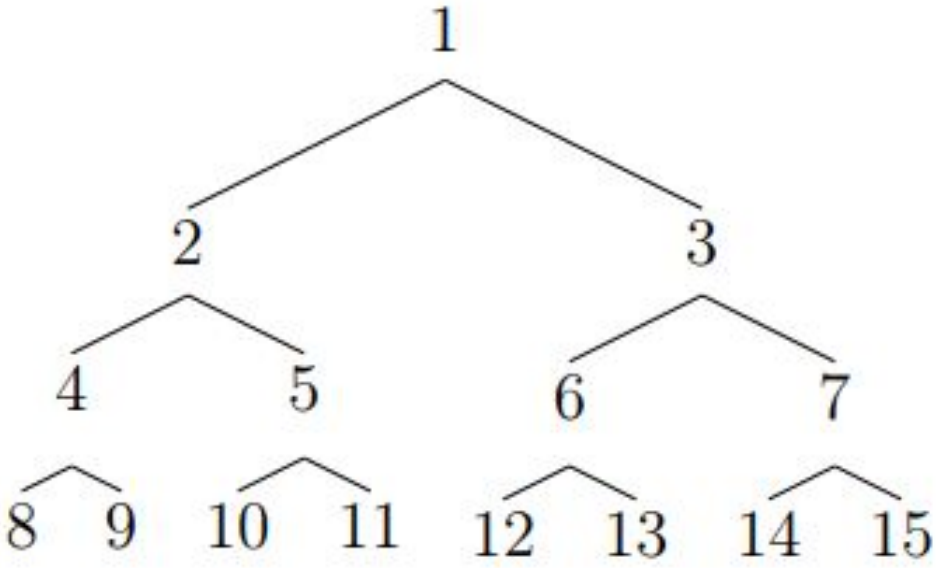

Dense
Low Rank
Low Rank
Dense


D_2
$A_{2,3}$
$A_{3,2}$
D_3


D_4
$A_{4,5}$
$A_{2,3}$
$A_{5,4}$
D_5
$A_{3,2}$
D_6
$A_{6,7}$
$A_{7,6}$
D_7


D_8
$A_{8,9}$
$A_{4,5}$
$A_{9,8}$
D_9
$A_{2,3}$
$A_{5,4}$
D_{10}
$A_{10,11}$
$A_{11,10}$
D_{11}
D_{12}
$A_{12,13}$
$A_{6,7}$
$A_{13,12}$
D_{13}
$A_{3,2}$
$A_{7,6}$
D_{14}
$A_{14,15}$
$A_{15,14}$
D_{15}

HSS Tree Representation



Interpolative Decomposition

For a matrix B, find X and J such that

$$B = B[:J] * X$$

where J is a set of k columns where k is a upper bound for the rank of the matrix B

Randomized algorithm which does a Rank Revealing QR Factorization and then a triangular solve

Recursive structure of off-diagonal blocks

For a matrix B with rank k we can write (via ID)

$$B = B[:J^{row}] * V^T$$

$$B[:J^{row}]^T = B[J^{col}, J^{row}]^T * U^T$$

$$B = U * B[J^{col}, J^{row}] * V^T$$

$$A_{v1,v2} = U_{v1}^{big} * B_{v1,v2} * V_{v2}^{big}$$

where $\dim(B_{v1,v2}) = (k, k)$

$$U_{\tau}^{big} = \begin{bmatrix} U_{v1}^{big} & 0 \\ 0 & U_{v2}^{big} \end{bmatrix} U_{\tau}$$

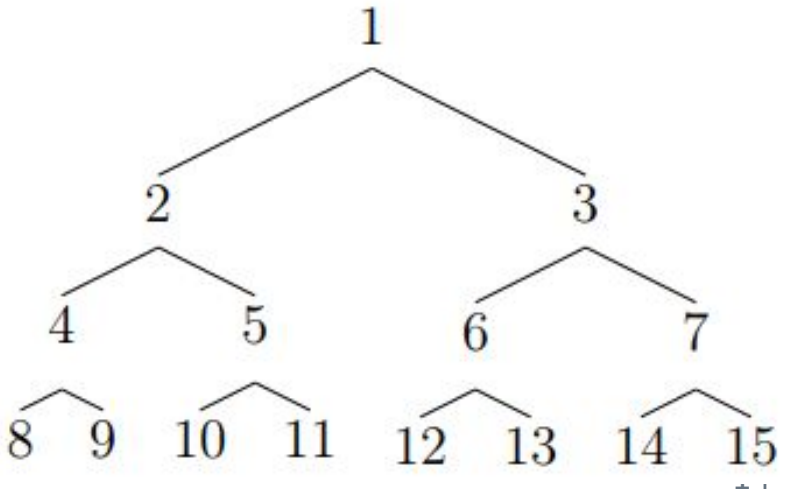
where $\dim(U_{\tau}) = (2*k, k)$

and v1, v2 are children of τ

Storage Cost of HSS

Assume that the hierarchical factorization stops when M is of size (2*k , 2*k), then there are $O(n/k)$ number of nodes.

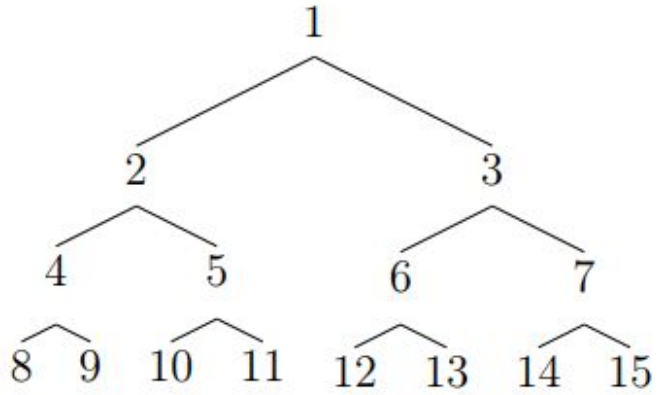
O(k*n) total storage cost which is O(n) when k is constant.



HSS Factorization

1. Generate 2 random matrices and multiply

2. Bottom up pass with ID and multiplications at each node



Algorithm 4: Computing the HSS factorization of a non-symmetric matrix.

Input: A fast means of computing matrix-vector products $x \mapsto A\,x$ and $x \mapsto A^*\,x$.
A method for computing individual entries of A.
An upper bound for the HSS-rank k of A.
A tree $\mathcal{T}$ on the index vector $[1, 2, \ldots, N]$.

Output: Matrices U_τ, V_τ, B_{ν_1,ν_2}, D_τ that form an HSS factorization of A.

Generate two $N \times (k+10)$ Gaussian random matrices R^{row} and R^{col}.
Evaluate $S^{\text{row}} = A^*\,R^{\text{row}}$ and $S^{\text{col}} = A\,R^{\text{col}}$ using the fast matrix-vector multiplier.
loop over levels, finer to coarser, $\ell = L, L-1, \ldots, 1$
  loop over all nodes τ on level ℓ
    if τ is a leaf node then

$$I^{\text{row}}_{\text{loc}} = I_\tau \qquad I^{\text{col}}_{\text{loc}} = I_\tau$$
$$R^{\text{row}}_{\text{loc}} = R(I_\tau, :) \qquad R^{\text{col}}_{\text{loc}} = R(I_\tau, :)$$
$$S^{\text{row}}_{\text{loc}} = S^{\text{row}}(I_\tau, :) - A(I_\tau, I_\tau)\,R^{\text{row}}_{\text{loc}} \qquad S^{\text{col}}_{\text{loc}} = S^{\text{col}}(I_\tau, :) - A(I_\tau, I_\tau)^*\,R^{\text{col}}_{\text{loc}}$$

    else
      Let ν_1 and ν_2 be the two children of τ.

$$I^{\text{row}}_{\text{loc}} = [\tilde{I}^{\text{row}}_{\nu_1}, \tilde{I}^{\text{row}}_{\nu_2}] \qquad I^{\text{col}}_{\text{loc}} = [\tilde{I}^{\text{col}}_{\nu_1}, \tilde{I}^{\text{col}}_{\nu_2}]$$
$$R^{\text{row}}_{\text{loc}} = \begin{bmatrix} R^{\text{row}}_{\nu_1} \\ R^{\text{row}}_{\nu_2} \end{bmatrix} \qquad R^{\text{col}}_{\text{loc}} = \begin{bmatrix} R^{\text{col}}_{\nu_1} \\ R^{\text{col}}_{\nu_2} \end{bmatrix}$$
$$S^{\text{row}}_{\text{loc}} = \begin{bmatrix} S^{\text{row}}_{\nu_1} - A(\tilde{I}^{\text{row}}_{\nu_1}, \tilde{I}^{\text{col}}_{\nu_2})\,R^{\text{row}}_{\nu_2} \\ S^{\text{row}}_{\nu_2} - A(\tilde{I}^{\text{row}}_{\nu_2}, \tilde{I}^{\text{col}}_{\nu_1})\,R^{\text{row}}_{\nu_1} \end{bmatrix} \qquad S^{\text{col}}_{\text{loc}} = \begin{bmatrix} S^{\text{col}}_{\nu_1} - A(\tilde{I}^{\text{row}}_{\nu_1}, \tilde{I}^{\text{col}}_{\nu_2})\,R^{\text{col}}_{\nu_2} \\ S^{\text{col}}_{\nu_2} - A(\tilde{I}^{\text{row}}_{\nu_2}, \tilde{I}^{\text{col}}_{\nu_1})\,R^{\text{col}}_{\nu_1} \end{bmatrix}$$

    end if

$$[U^{\text{row}}_\tau, J^{\text{row}}_\tau] = \texttt{interpolate}((S^{\text{row}}_{\text{loc}})^*) \qquad [U^{\text{col}}_\tau, J^{\text{col}}_\tau] = \texttt{interpolate}((S^{\text{col}}_{\text{loc}})^*)$$
$$R^{\text{row}}_\tau = (U^{\text{col}}_\tau)^*\,R^{\text{row}}_{\text{loc}} \qquad R^{\text{col}}_\tau = (U^{\text{row}}_\tau)^*\,R^{\text{col}}_{\text{loc}}$$
$$S^{\text{row}}_\tau = S^{\text{row}}_{\text{loc}}(J^{\text{row}}_\tau, :) \qquad S^{\text{col}}_\tau = S^{\text{col}}_{\text{loc}}(J^{\text{col}}_\tau, :)$$
$$\tilde{I}^{\text{row}}_\tau = I^{\text{row}}_{\text{loc}}(J^{\text{row}}_\tau) \qquad \tilde{I}^{\text{col}}_\tau = I^{\text{col}}_{\text{loc}}(J^{\text{col}}_\tau)$$

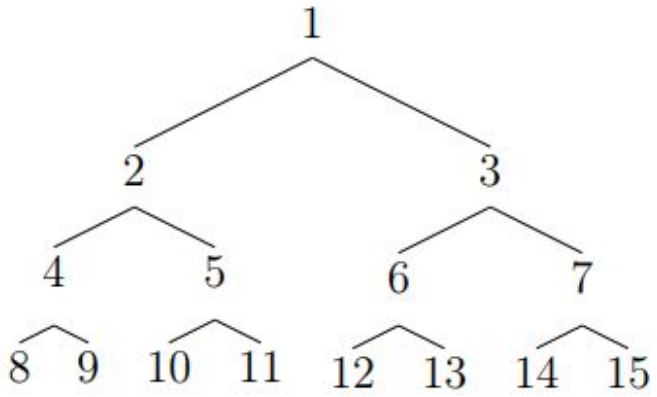
  end loop
end loop
For all leaf nodes τ, set $D_\tau = A(I_\tau, I_\tau)$.
For all sibling pairs $\{\nu_1, \nu_2\}$ set $B_{\nu_1,\nu_2} = A(\tilde{I}^{\text{row}}_{\nu_1}, \tilde{I}^{\text{col}}_{\nu_2})$.

Methodology

Single GPU implementation

- Transfer the full matrix to GPU
- Use CURAND and CUBLAS for randomized sampling and projection

- CUDA kernels for each level of the HSS tree for factorization
- Assumes that all of the small matrices have rank the same as the upper bound of the rank. When it is not, some rows in Q of QR factorization become zero.

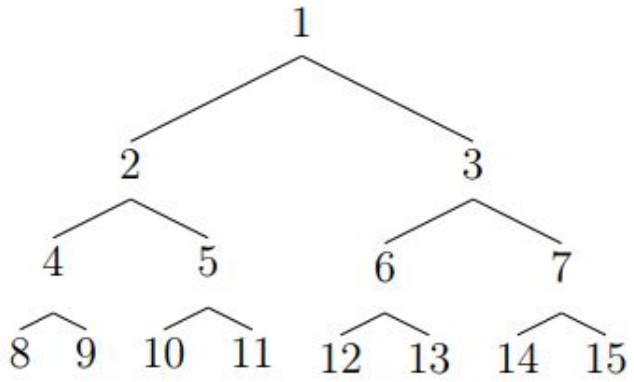


Matrix-Vector product using HSS

At each node a dense matrix-multiplication of a matrix $O(k^2)$ and a vector $O(k)$ is done.

Number of nodes = $O(N/k)$

Time complexity = $O(N^*k)$



Algorithm 1

Given all factors U_τ, V_τ, B_{ν_1,ν_2} and D_τ of an HSS matrix A, and a vector x, this scheme computes the product $b = A\,x$.

(1) For every leaf node τ, calculate $\tilde{x}_\tau = V_\tau^*\, x(I_\tau)$.

(2) Looping over all non-leaf nodes τ, from finer to coarser, calculate

$$\tilde{x}_\tau = V_\tau^* \begin{bmatrix} \tilde{x}_{\nu_1} \\ \tilde{x}_{\nu_2} \end{bmatrix},$$

where ν_1 and ν_2 are the children of τ.

(3) Set $\tilde{b}_\tau = 0$ for the root node τ.

(4) Looping over all non-leaf nodes τ, from coarser to finer, calculate

$$\begin{bmatrix} \tilde{b}_{\nu_1} \\ \tilde{b}_{\nu_2} \end{bmatrix} = \begin{bmatrix} 0 & B_{\nu_1,\nu_2} \\ B_{\nu_2,\nu_1} & 0 \end{bmatrix} \begin{bmatrix} \tilde{x}_{\nu_1} \\ \tilde{x}_{\nu_2} \end{bmatrix} + U_\tau\, \tilde{b}_\tau$$

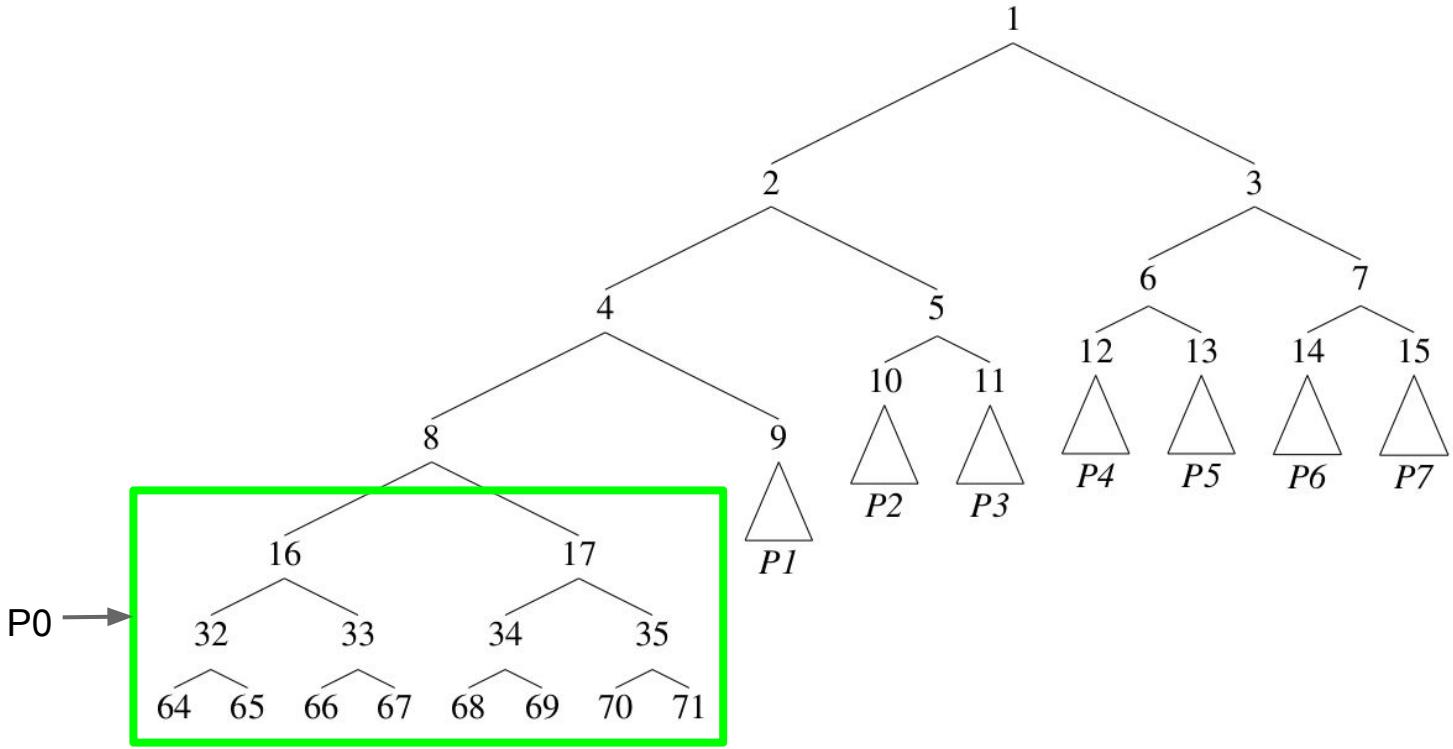
where ν_1 and ν_2 are the children of τ.

(5) For every leaf node τ, calculate $b(I_\tau) = U_\tau\, \tilde{b}_\tau + D_\tau\, x(I_\tau)$.

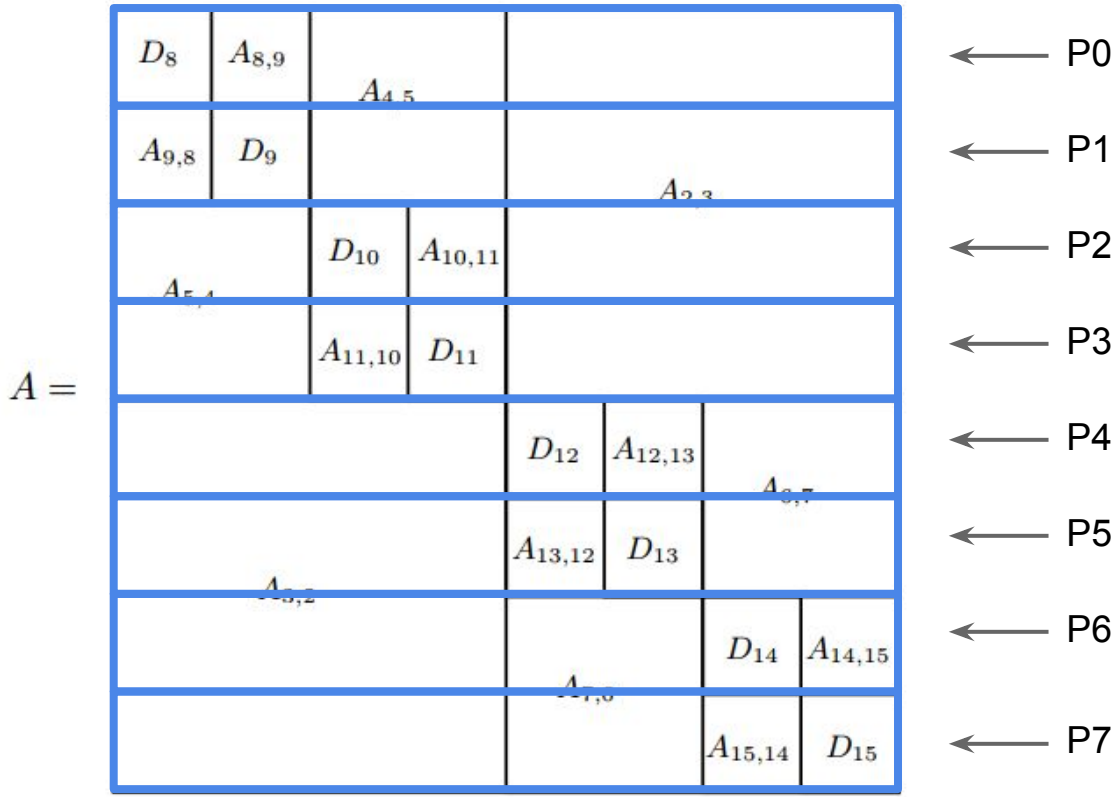
Parallelization across nodes

- Matrix factorization and matvec for a distributed memory system using MPI

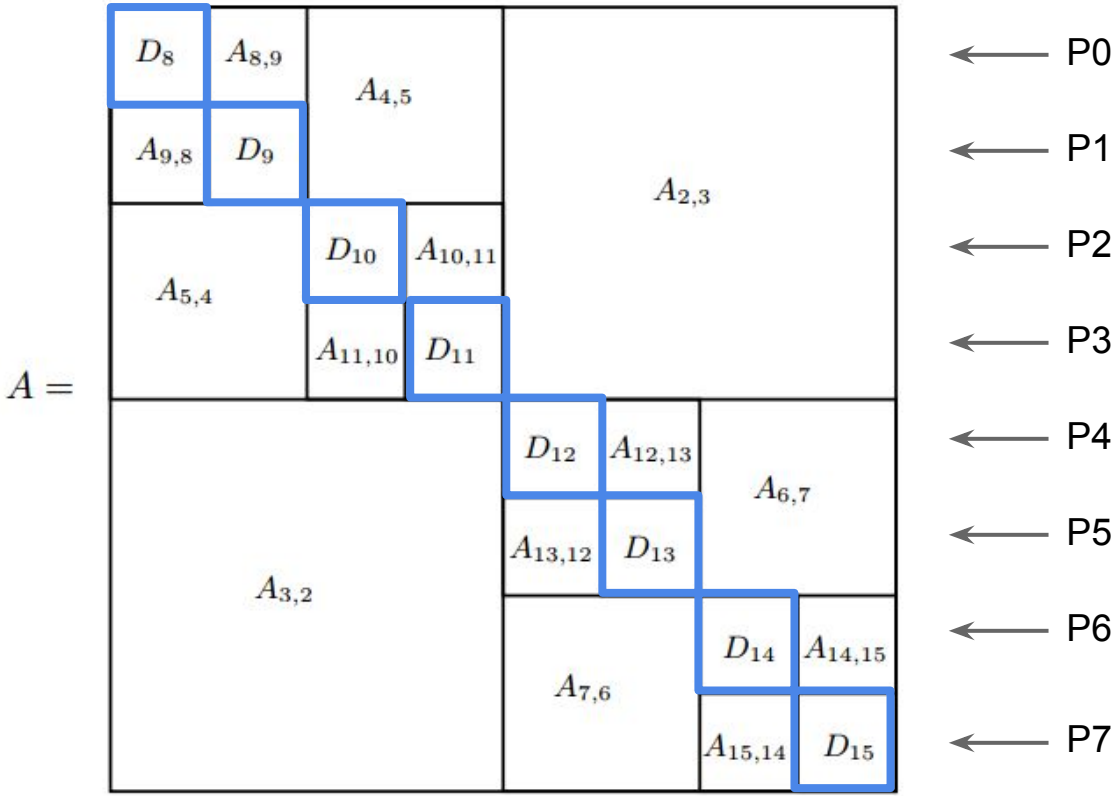
How computation is distributed



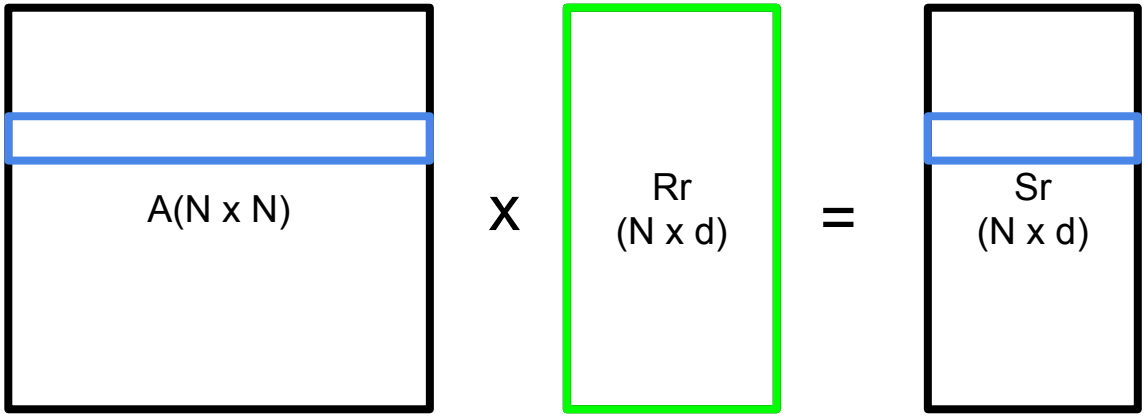
How data is distributed on the CPU



How data is distributed on the GPU



Distributed randomize



Distributed randomize

Option 1

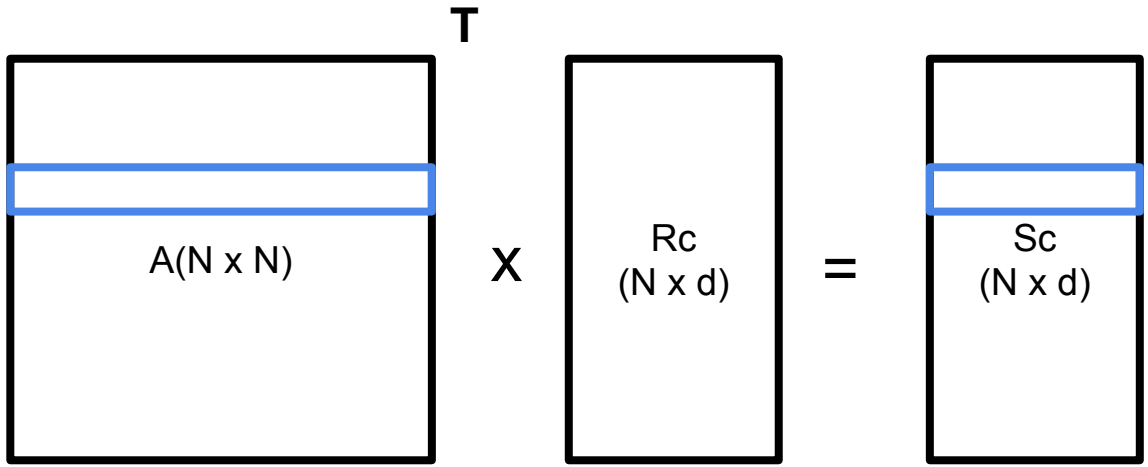
- Distribute the seed and compute Nxd matrix in each GPU

Option 2

- Compute *(N/p) x d* on each node on the GPU, bring it back to the CPU
- Do a MPI reduce scatter
- Transfer *(N/p) x d* data to the GPU.

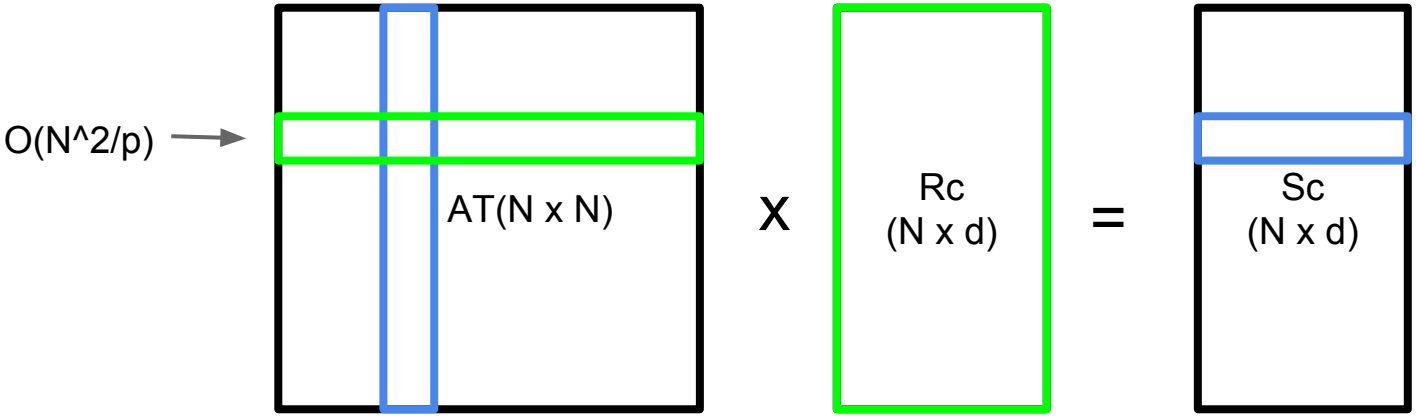
Distributed randomize

2. Generate Rc and compute Sc



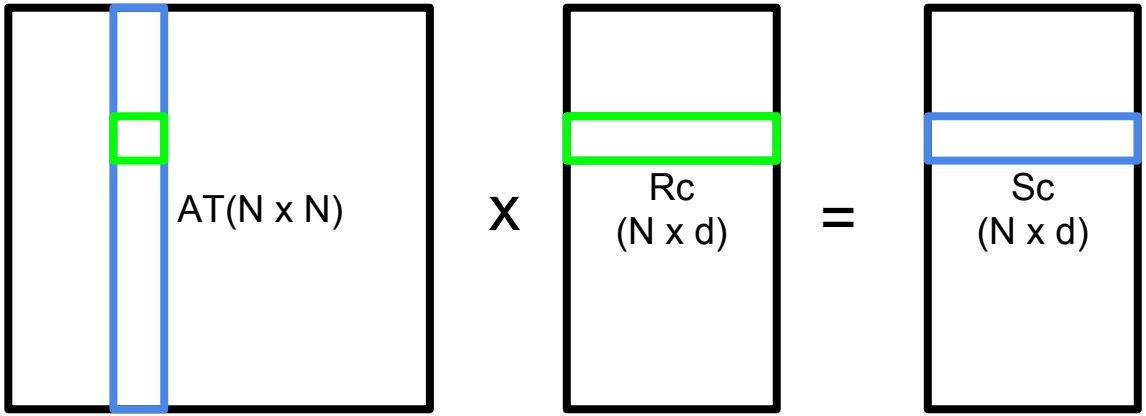
Distributed randomize

2. Generate Rc and compute Sc



Distributed randomize

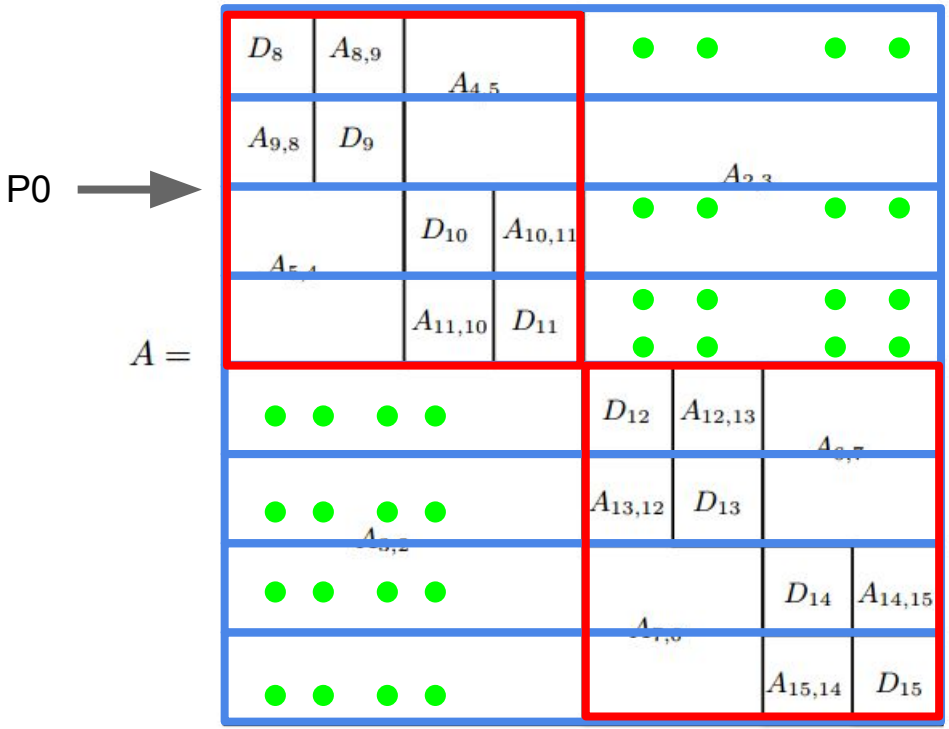
2. Generate Rc and compute Sc

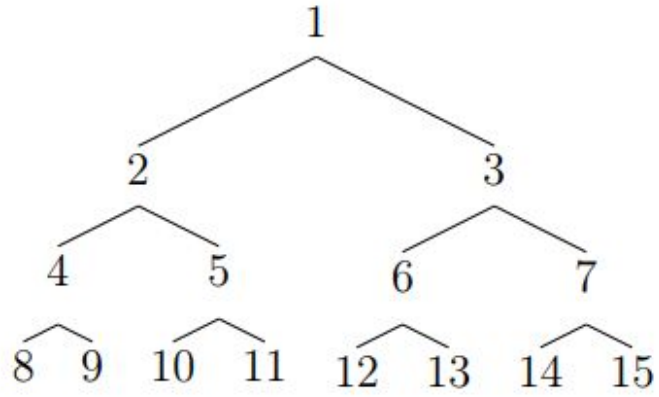


Do the multiplication and a MPI_Allreduce

O(N x d) communication better than $O(N^2)$

Distributed Merging





Distributed Matvec

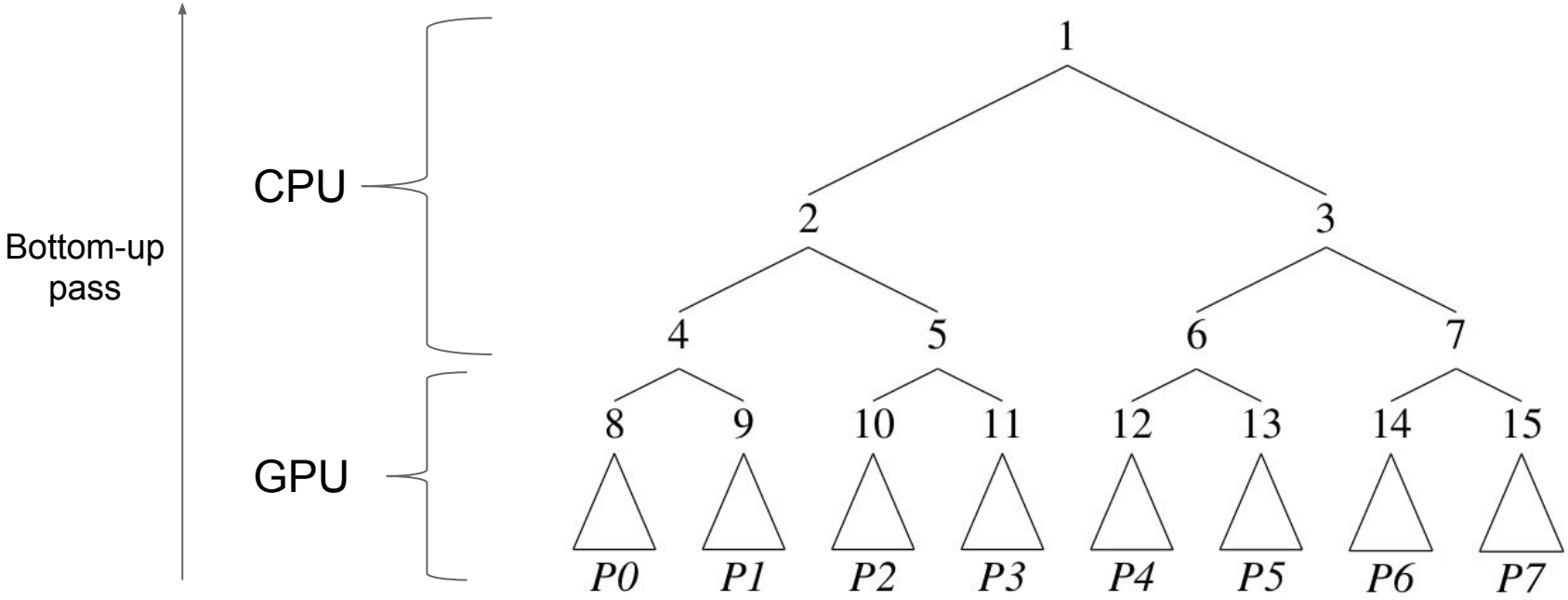


Fig. 3. HSS tree with the subtrees given to 8 processors

Distributed Matvec

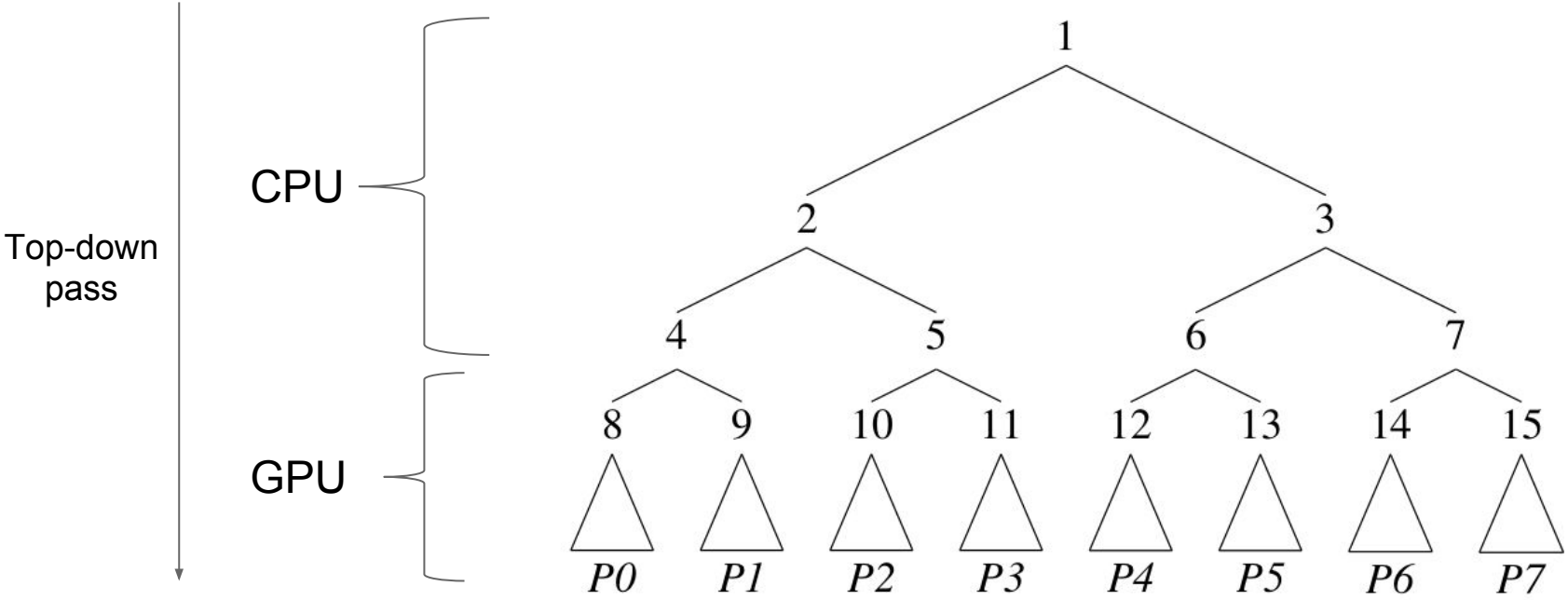


Fig. 3. HSS tree with the subtrees given to 8 processors

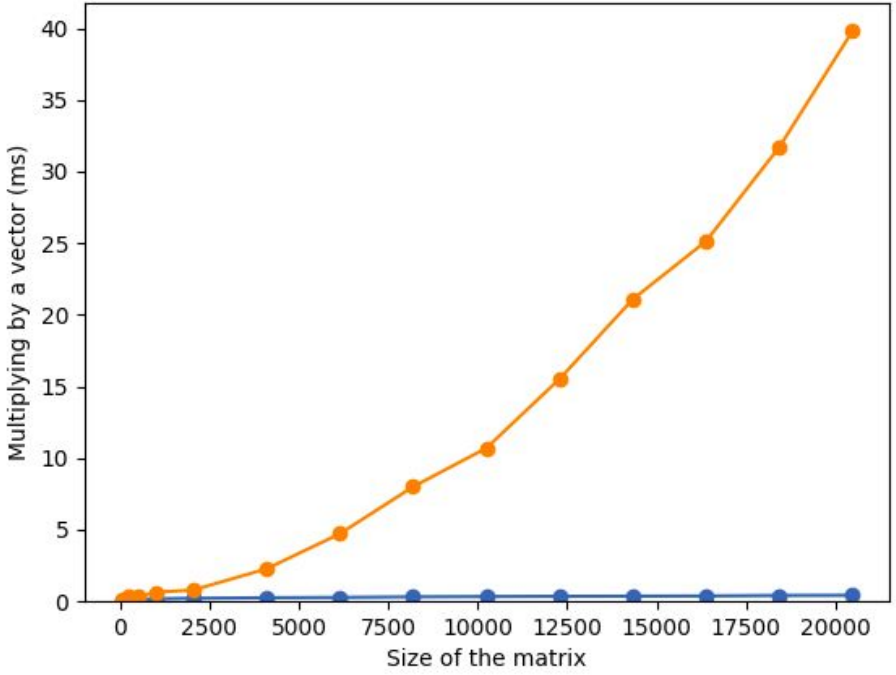
Results - Single Node

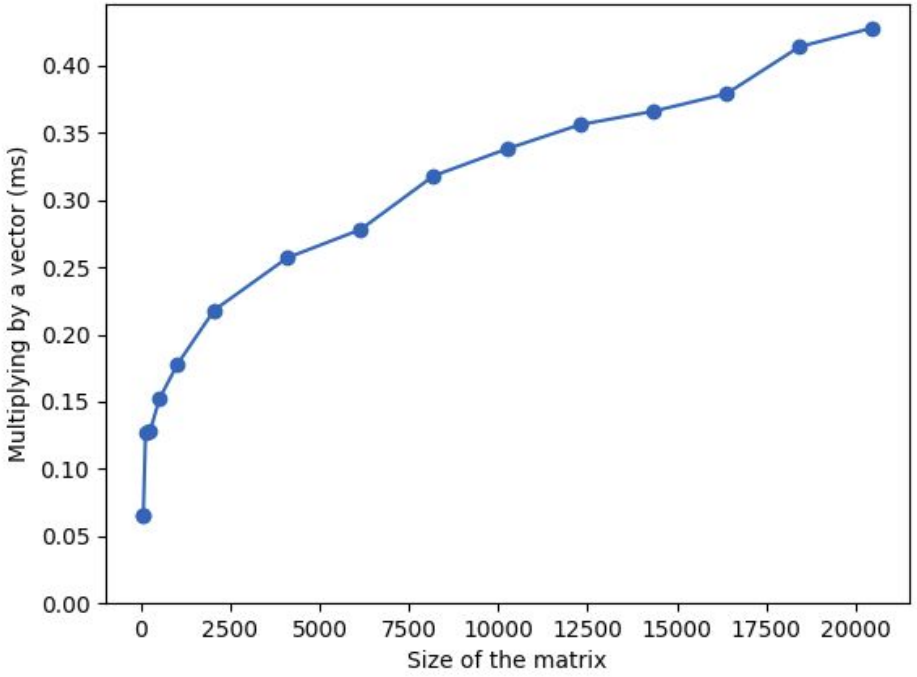
Matrix Size	HSS time (ms)	MATVEC (HSS) time (ms)	MATVEC (dense) time (ms)	MATMAT (HSS) time (ms)	MATMAT (dense) time (ms)	Error of norm of MATVEC
32	32	0.065	0.161	0.066	0.164	0.0
64	68	0.066	0.055	0.092	0.171	0.0
128	98	0.127	0.184	0.177	0.308	0.0
256	135	0.128	0.315	0.227	0.393	1.183e-16
512	165	0.152	0.368	0.507	0.68	1.070e-15
1024	197	0.178	0.644	0.76	2.124	1.545e-15
2048	241	0.218	0.798	1.214	7.635	-2.147e-15
4096	292	0.257	2.278	2.06	32.888	-1.778e-14
8192	383	0.291	8.004	3.825	91.844	-5.290e-14
16384	633	0.352	25.139	7.534	365.964	-7.360e-14

Single node GPU results

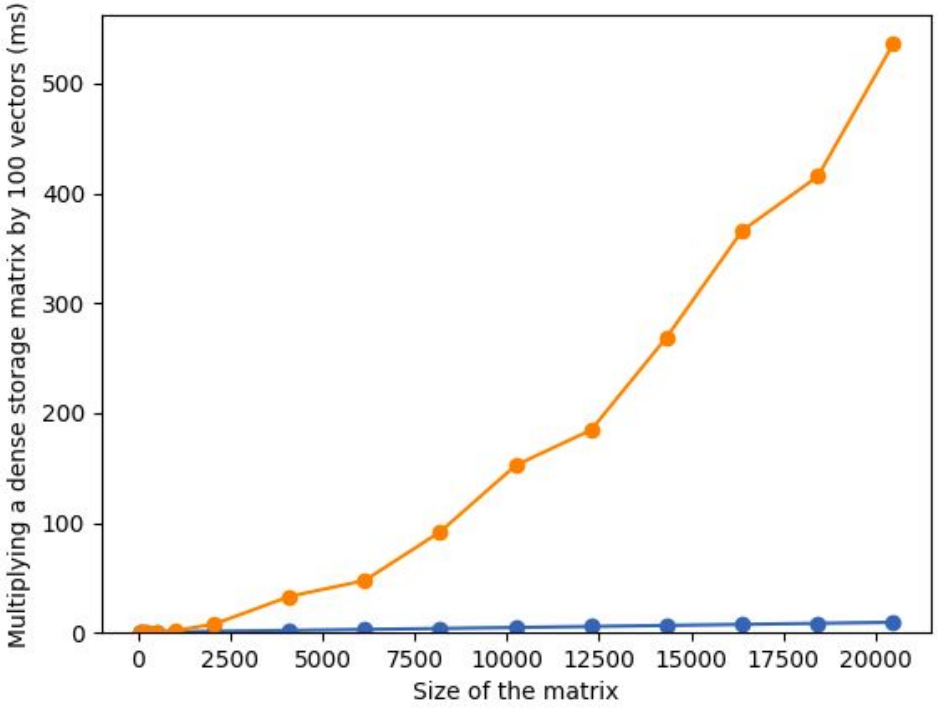
- Intel 6700U processor with 1 GeForce Titan X GPU

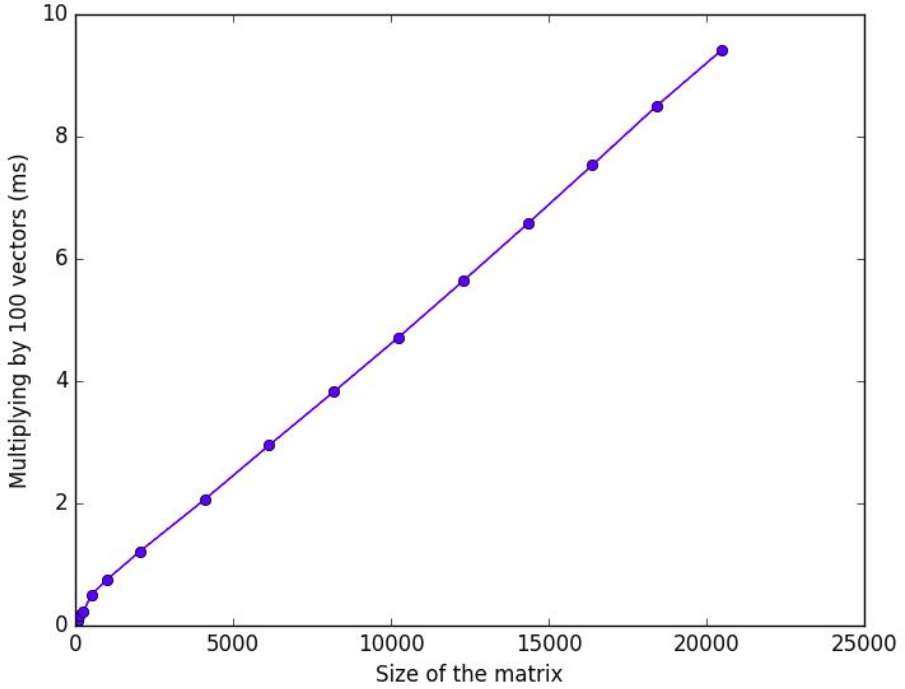
Results - Single Node





Results - Single Node





Results - Multi-node

Weak Scaling

Matrix size / Number of GPU nodes = constant

Results - Multi-node

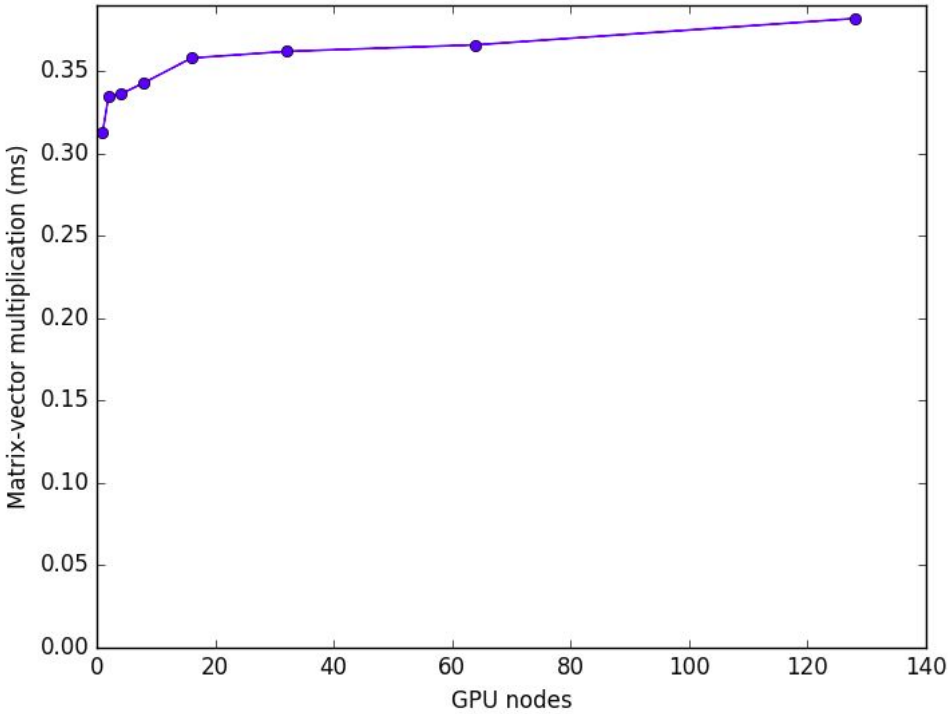
matrix size	# GPUs	HSS time(ms)	MATVEC (HSS) time(ms)	MATVEC (dense) time (ms)
1024	1	181.189	0.313	0.894
2048	2	187.564	0.335	0.824
4096	4	195.951	0.336	0.719
8192	8	214.312	0.343	1.012
16384	16	245.603	0.358	1.818
32768	32	308.729	0.362	2.932
65536	64	439.41	0.366	5.176
131072	128	700.293	0.382	9.574

Weak scaling results run on BigRed2 at Indiana

128 nodes with 1 K20 GPU each

Results - Multi-node

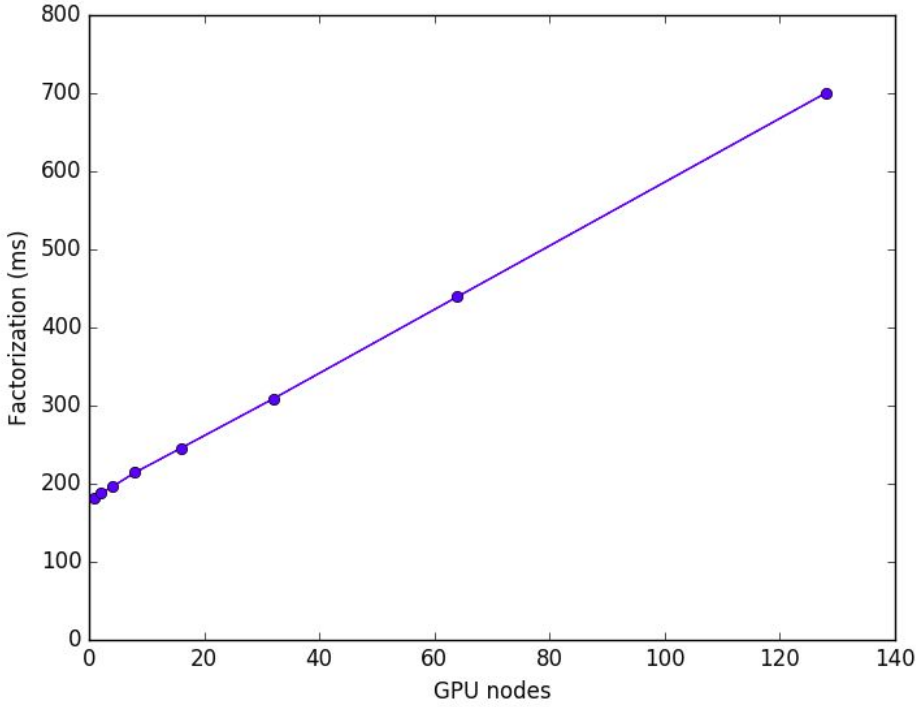
Matvec = O(N/p) = O(1)



Results - Multi-node

Factorization = $O(N^2/p)$

= $O(N)$



Conclusion

Matrix-vector multiplication can be done efficiently for HSS matrices on GPUs.

It has a high setup cost of factorization as in other factorization algorithms, but if there are many matvecs, then the setup cost can be negligible.

THANK YOU

Experimental setup

- BigRed2 at Indiana University
  - 132 AMD Opteron 16-core Interlagos x86 64 CPUs
  - 1 x K20 GPU

- KINGSPEAK at University of Utah
  - 4 Intel Xeon E5-2670 processors
  - 2 x P100 GPUs each

- Single Node at University of Moratuwa
  - 1 Intel 6700U processor
  - 1 GeForce Titan X GPU

HSS Factorization - summary

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 p^{th} 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 p^{th} 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

Benchmarks

Multiplication of (N, N) * (N, E)					
	N	GPU HSS Time (s)	GPU Cublas Time (s)	CPU HSS Time (s)	CPU OpenBLAS Time (s)
E = 1	4225	0.453	6.767	0.802	28.207
	16641	0.583	100.717	3.075	441.287
E = 100	4225	2.788	29.843	48.793	52.695
	16,641	10.160	366.082	198.407	1019.200

Methodology

Parallelization across a GPU