$\mathcal{H}$-Matrices and $\mathcal{H}$-Arithmetic on Many-Core Systems

Ronald Kriemann
MPI MIS

TC/PC^2 Kolloquium

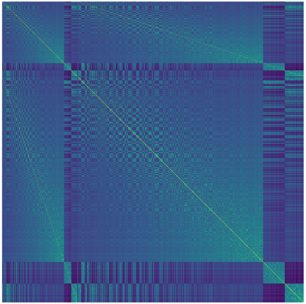
Uni Paderborn

2018-09-10

Hierarchical Matrices

Motivation

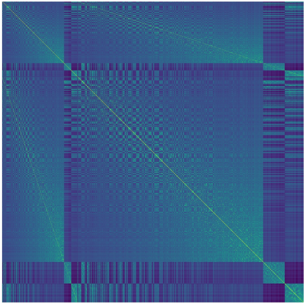
In *$\mathcal{H}$-matrices* the rows and columns of a given dense $n \times n$ matrix M are reordered to expose the (numerical) *low-rank structure* of subblocks of M.



(Example: Helmholtz Integral Equation)

Motivation

In *$\mathcal{H}$-matrices* the rows and columns of a given dense $n \times n$ matrix M are reordered to expose the (numerical) *low-rank structure* of subblocks of M.



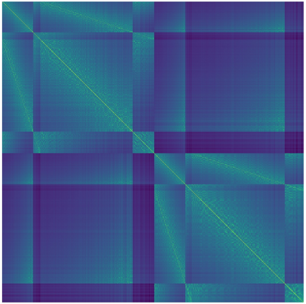
(Example: Helmholtz Integral Equation)

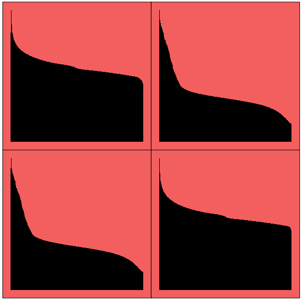
Singular Value Decomposition (SVD)

For any $n \times n$ matrix M exist orthogonal $n \times n$ matrices U, V and $S = \text{diag}(s_0, \ldots, s_{n-1})$ such that $M = USV^T = \sum_{i=0}^{n-1} s_i U(:, i) V(:, i)^T$. The s_i are called *singular values* and are descending: $s_0 \geq s_1 \geq \ldots \geq s_{n-1} \geq 0$.

Motivation

In *$\mathcal{H}$-matrices* the rows and columns of a given dense $n \times n$ matrix M are reordered to expose the (numerical) *low-rank structure* of subblocks of M.





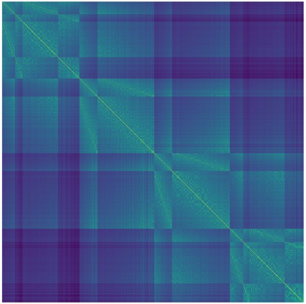
(Example: Helmholtz Integral Equation)

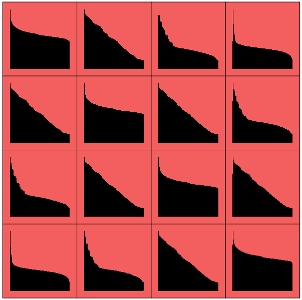
Singular Value Decomposition (SVD)

For any $n \times n$ matrix M exist orthogonal $n \times n$ matrices U, V and $S = \text{diag}(s_0, \ldots, s_{n-1})$ such that $M = USV^T = \sum_{i=0}^{n-1} s_i U(:, i) V(:, i)^T$. The s_i are called *singular values* and are descending: $s_0 \geq s_1 \geq \ldots \geq s_{n-1} \geq 0$.

Motivation

In *$\mathcal{H}$-matrices* the rows and columns of a given dense $n \times n$ matrix M are reordered to expose the (numerical) *low-rank structure* of subblocks of M.





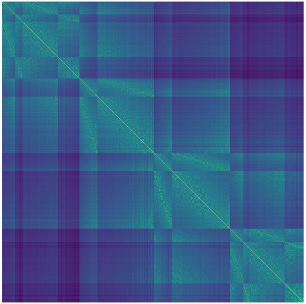
(Example: Helmholtz Integral Equation)

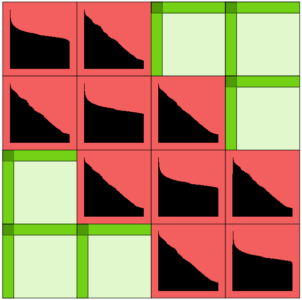
Singular Value Decomposition (SVD)

For any $n \times n$ matrix M exist orthogonal $n \times n$ matrices U, V and $S = \operatorname{diag}(s_0, \ldots, s_{n-1})$ such that $M = USV^T = \sum_{i=0}^{n-1} s_i U(:,i)V(:,i)^T$. The s_i are called *singular values* and are descending: $s_0 \geq s_1 \geq \ldots \geq s_{n-1} \geq 0$.

Motivation

In *$\mathcal{H}$-matrices* the rows and columns of a given dense $n \times n$ matrix M are reordered to expose the (numerical) *low-rank structure* of subblocks of M.





(Example: Helmholtz Integral Equation)

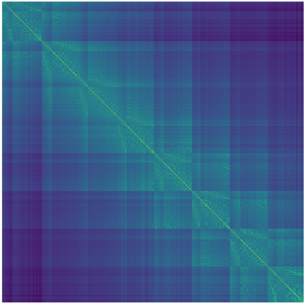
Singular Value Decomposition (SVD)

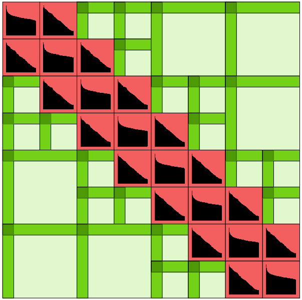
For any $n \times n$ matrix M exist orthogonal $n \times n$ matrices U, V and $S = \text{diag}(s_0, \dots, s_{n-1})$ such that $M = USV^T = \sum_{i=0}^{n-1} s_i U(:,i) V(:,i)^T$. The s_i are called *singular values* and are descending: $s_0 \geq s_1 \geq \dots \geq s_{n-1} \geq 0$.

Low-rank approximable $n' \times m'$ subblocks M' are represented in factorised form $M' \approx A \cdot B^T$, with $n' \times k$ matrix A and $m' \times k$ matrix B.

Motivation

In *$\mathcal{H}$-matrices* the rows and columns of a given dense $n \times n$ matrix M are reordered to expose the (numerical) *low-rank structure* of subblocks of M.





(Example: Helmholtz Integral Equation)

Singular Value Decomposition (SVD)

For any $n \times n$ matrix M exist orthogonal $n \times n$ matrices U, V and $S = \mathrm{diag}(s_0, \dots, s_{n-1})$ such that $M = USV^T = \sum_{i=0}^{n-1} s_i U(:,i)V(:,i)^T$. The s_i are called *singular values* and are descending: $s_0 \geq s_1 \geq \dots \geq s_{n-1} \geq 0$.

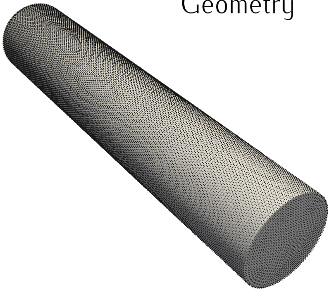
Low-rank approximable $n' \times m'$ subblocks M' are represented in factorised form $M' \approx A \cdot B^T$, with $n' \times k$ matrix A and $m' \times k$ matrix B.

Clustering

(Recursive) Block Structure

The *clustering* (reordering) defines a *hierarchical* partitioning for block index set $I \times I$, $I = \{0, \ldots, n-1\}$.

Geometry

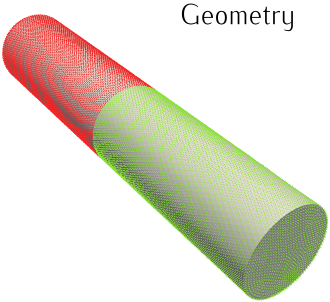


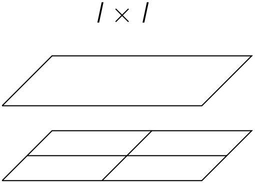
$I \times I$

Clustering

(Recursive) Block Structure

The *clustering* (reordering) defines a *hierarchical* partitioning for block index set $I \times I$, $I = \{0, \ldots, n-1\}$.

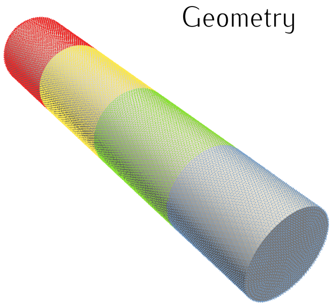


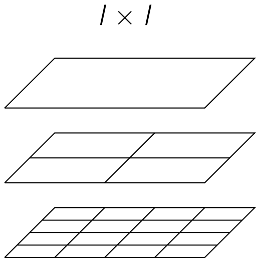


Clustering

(Recursive) Block Structure

The *clustering* (reordering) defines a *hierarchical* partitioning for block index set $I \times I$, $I = \{0, \ldots, n-1\}$.



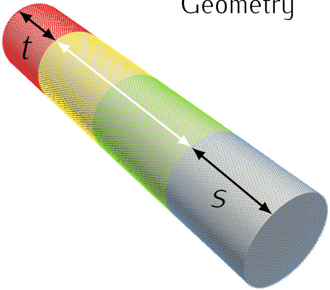


Clustering

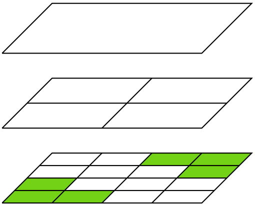
(Recursive) Block Structure

The *clustering* (reordering) defines a *hierarchical* partitioning for block index set $I \times I$, $I = \{0, \dots, n-1\}$.

Geometry



$I \times I$



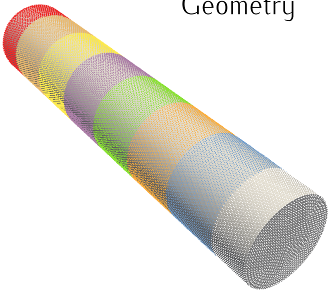
Low-rank approximable blocks are identified with an *admissibility condition*:

$$\max\{\text{diam}(t), \text{diam}(s)\} \leq \eta \, \text{dist}(t, s), \quad \eta > 0$$

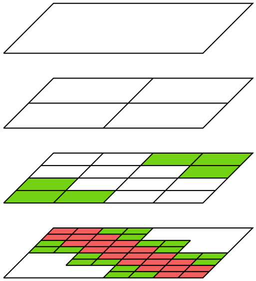
(Recursive) Block Structure

The *clustering* (reordering) defines a *hierarchical* partitioning for block index set $I \times I$, $I = \{0, \ldots, n-1\}$.

Geometry



$I \times I$

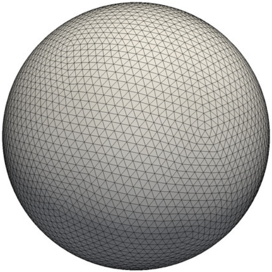


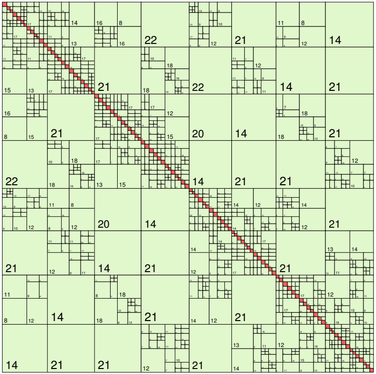
Low-rank approximable blocks are identified with an *admissibility condition*:

$$\max\{\operatorname{diam}(t), \operatorname{diam}(s)\} \leq \eta \operatorname{dist}(t, s), \quad \eta > 0$$

Clustering

Structure depends on Geometry



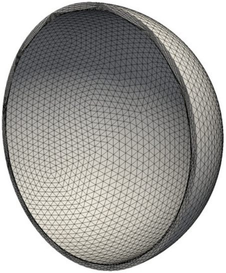


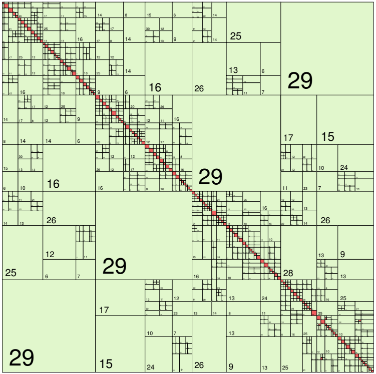
$n = 124.928, \quad \text{compression} = 98.78\%$

(Example: Helmholtz Integral Equation)

Clustering

Structure depends on Geometry



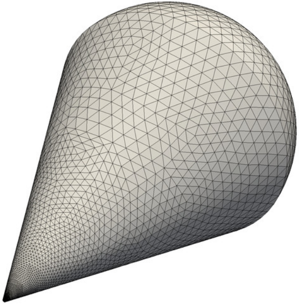


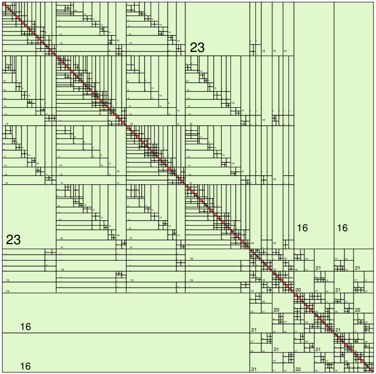
$n = 149.504, \quad \text{compression} = 98.75\%$

(Example: Helmholtz Integral Equation)

Clustering

Structure depends on Geometry



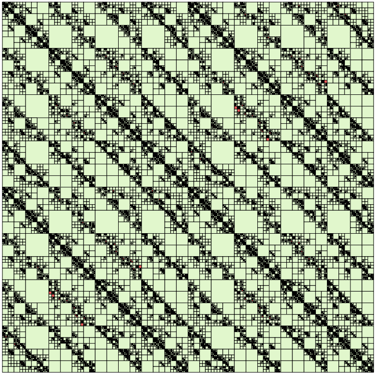


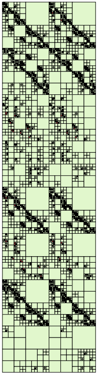
$n = 175.616, \quad \text{compression} = 99.09\%$

(Example: Helmholtz Integral Equation)

Clustering

Structure depends on Geometry/Problem



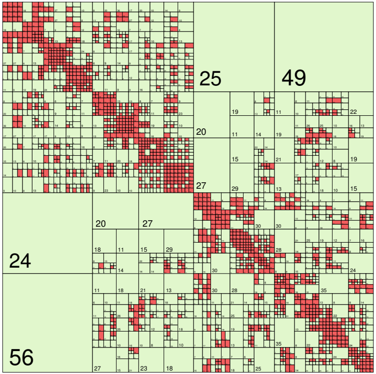


$n = 75.440, \#RHS = 15.088,$ compression $= 92.55\%/93.39\%$

(Example: AO Tomography for E-ELT)

Clustering

Structure depends on Geometry/Problem



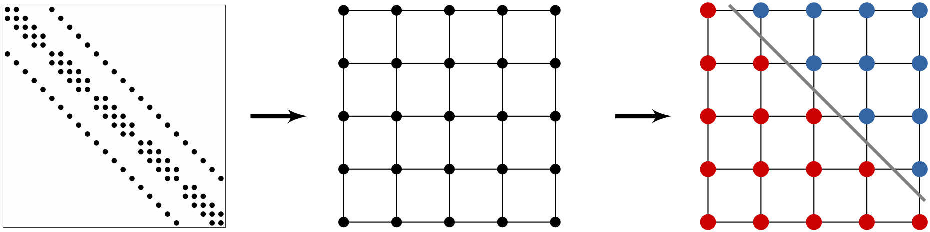
$$n = 70.785, \quad \text{compression} = 92.22\%$$

(Example: Inverse of Sparse Matrix)

Clustering

Sparse Matrices

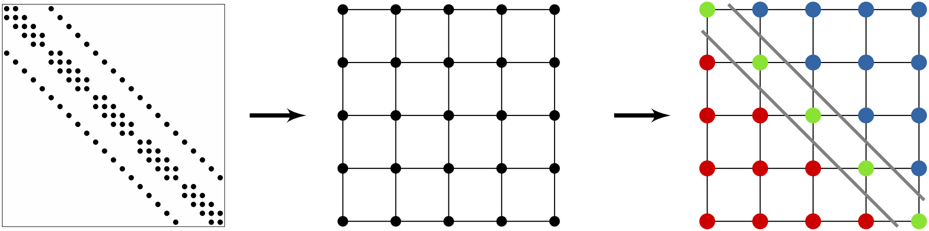
For sparse matrices, if no geometry data is available, also *graph partitioning* applied to the matrix graph can be used to compute the $\mathcal{H}$-matrix partition.



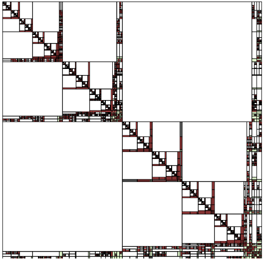
Clustering

Sparse Matrices

For sparse matrices, if no geometry data is available, also *graph partitioning* applied to the matrix graph can be used to compute the $\mathcal{H}$-matrix partition.



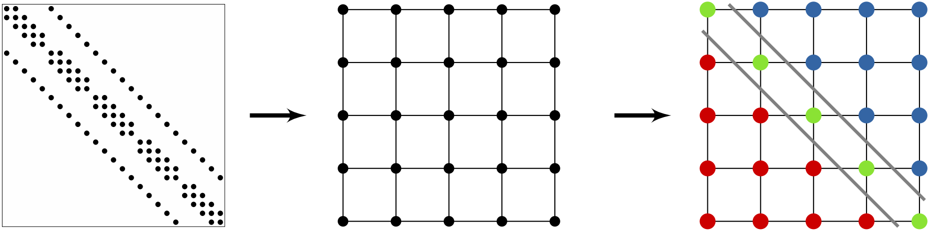
Combined with *nested dissection*, this yields efficient partitionings for the $\mathcal{H}$-LU of sparse matrices.



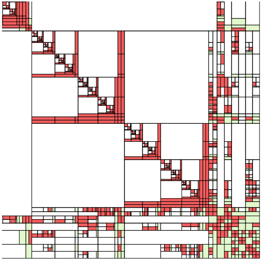
Clustering

Sparse Matrices

For sparse matrices, if no geometry data is available, also *graph partitioning* applied to the matrix graph can be used to compute the $\mathcal{H}$–matrix partition.



Combined with *nested dissection*, this yields efficient partitionings for the $\mathcal{H}$–LU of sparse matrices.



Low-Rank Approximation

Different algorithms are available for computing low-rank approximations for dense matrix blocks, e.g.,

> *SVD, interpolation, adaptive cross approximation, hybrid cross approximation, RRQR, Rand-SVD, . . .*

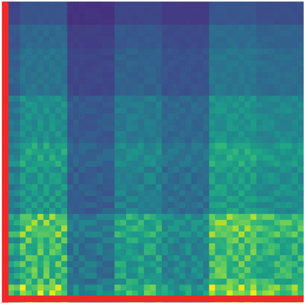
Low-Rank Approximation

Different algorithms are available for computing low-rank approximations for dense matrix blocks, e.g.,

SVD, interpolation, adaptive cross approximation, hybrid cross approximation, RRQR, Rand-SVD, . . .

Adaptive Cross Approximation

procedure ACA(in: M, k, out: A, B)
  $A := []; B := []; k := 0$
  for $i = 0, \ldots, k-1$ **do**
    $u := M(:, i) - A \cdot B(i, :)'$;
    $[u_{\max}, j] = \max(\text{abs}(u))$;
    $v := M(j, :) - A(j, :) \cdot B'$;
    $u := u / M(i, j)$;
    $A := [A, u]$;
    $B := [B, v']$;



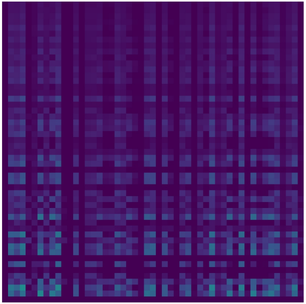
Low-Rank Approximation

Different algorithms are available for computing low-rank approximations for dense matrix blocks, e.g.,

SVD, interpolation, adaptive cross approximation, hybrid cross approximation, RRQR, Rand-SVD, ...

Adaptive Cross Approximation

procedure ACA(in: M, k, out: A, B)
  $A := []; B := []; k := 0$
  for $i = 0, \dots, k-1$ **do**
    $u := M(:, i) - A \cdot B(i, :)'$;
    $[u_{\max}, j] = \max(\text{abs}(u))$;
    $v := M(j, :) - A(j, :) \cdot B'$;
    $u := u / M(i, j)$;
    $A := [A, u]$;
    $B := [B, v']$;



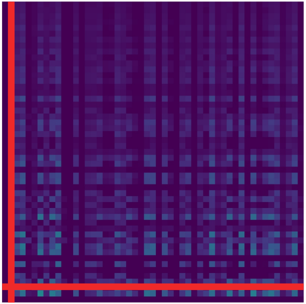
Low-Rank Approximation

Different algorithms are available for computing low-rank approximations for dense matrix blocks, e.g.,

SVD, interpolation, adaptive cross approximation, hybrid cross approximation, RRQR, Rand-SVD, . . .

Adaptive Cross Approximation

procedure ACA(in: M, k, out: A, B)
  $A := []; B := []; k := 0$
  for $i = 0, \ldots, k-1$ **do**
    $u := M(:, i) - A \cdot B(i, :)'$;
    $[u_{\max}, j] = \max(\text{abs}(u))$;
    $v := M(j, :) - A(j, :) \cdot B'$;
    $u := u / M(i, j)$;
    $A := [A, u]$;
    $B := [B, v']$;



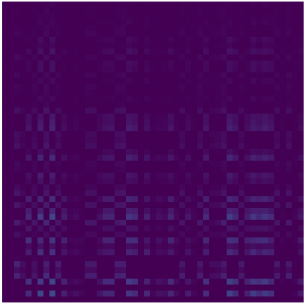
Low-Rank Approximation

Different algorithms are available for computing low-rank approximations for dense matrix blocks, e.g.,

> *SVD, interpolation, adaptive cross approximation, hybrid cross approximation, RRQR, Rand-SVD, ...*

Adaptive Cross Approximation

procedure ACA(in: M, k, out: A, B)
  $A := []; B := []; k := 0$
  for $i = 0, \dots, k-1$ **do**
    $u := M(:, i) - A \cdot B(i, :)'$;
    $[u_{\max}, j] = \max(\text{abs}(u))$;
    $v := M(j, :) - A(j, :) \cdot B'$;
    $u := u / M(i, j)$;
    $A := [A, u]$;
    $B := [B, v']$;



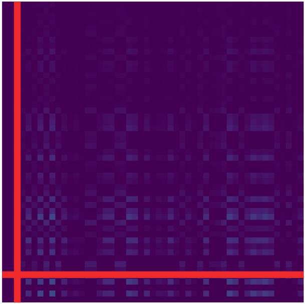
Low-Rank Approximation

Different algorithms are available for computing low-rank approximations for dense matrix blocks, e.g.,

SVD, interpolation, adaptive cross approximation, hybrid cross approximation, RRQR, Rand-SVD, ...

Adaptive Cross Approximation

procedure ACA(in: M, k, out: A, B)
  $A := []; B := []; k := 0$
  for $i = 0, \ldots, k-1$ **do**
    $u := M(:, i) - A \cdot B(i, :)'$;
    $[u_{\max}, j] = \max(\text{abs}(u))$;
    $v := M(j, :) - A(j, :) \cdot B'$;
    $u := u / M(i, j)$;
    $A := [A, u]$;
    $B := [B, v']$;



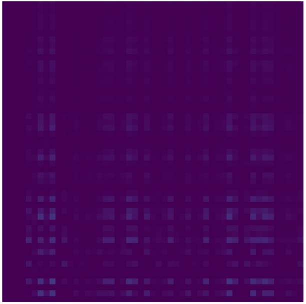
Low-Rank Approximation

Different algorithms are available for computing low-rank approximations for dense matrix blocks, e.g.,

> *SVD, interpolation, adaptive cross approximation, hybrid cross approximation, RRQR, Rand-SVD, . . .*

Adaptive Cross Approximation

```
procedure ACA(in: M, k, out: A, B)
    A := []; B := []; k := 0
    for i = 0, ..., k − 1 do
        u := M(:, i) − A · B(i, :)';
        [u_max, j] = max(abs(u));
        v := M(j, :) − A(j, :) · B';
        u := u/M(i, j);
        A := [A, u];
        B := [B, v'];
```



The resulting $\mathcal{H}$-matrix has storage complexity of $\mathcal{O}(n \log n)$.

Arithmetic

Low-Rank Arithmetic

Low-rank matrices $M \in \mathbb{C}^{n \times m}$ are stored in factorized form

$$M = A \cdot B^T$$

Matrix multiplication with a low-rank matrix preserves the rank.

However, matrix addition will increase the rank, e.g., for two rank-k matrices M_1 and M_2, the sum

$$M_1 + M_2 = A_1 \cdot B_1^T + A_2 \cdot B_2^T = [A_1, A_2] \cdot [B_1, B_2]^T$$

has rank $2k$.

Low-Rank Arithmetic

Low-rank matrices $M \in \mathbb{C}^{n \times m}$ are stored in factorized form

$$M = A \cdot B^T$$

Matrix multiplication with a low-rank matrix preserves the rank.

However, matrix addition will increase the rank, e.g., for two rank-k matrices M_1 and M_2, the sum

$$M_1 + M_2 = A_1 \cdot B_1^T + A_2 \cdot B_2^T = [A_1, A_2] \cdot [B_1, B_2]^T$$

has rank $2k$.

In $\mathcal{H}$-arithmetic all sums of low-rank matrices are *truncated* back to rank k.

$\mathcal{H}$-matrix arithmetic is not exact but approximative.

Instead of a fixed rank k, this can also be performed with a given precision $\varepsilon > 0$.

Arithmetic

$\mathcal{H}$-Arithmetic is based on *recursive* block algorithms and (truncated) *low-rank* arithmetic.

For an $\mathcal{H}$-Matrix A with a 2×2 block structure, e.g.,

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix},$$

we have the following algorithms for matrix multiplication and LU factorization:

```
procedure MULTIPLY(α, A, B, C)
  if A, B, C are block matrices then
    for i ∈ {0,1} do
      for j ∈ {0,1} do
        for ℓ ∈ {0,1} do
          MULTIPLY( α, A_ij, B_iℓ, C_ℓj );
  else
    C := C + αAB;
```

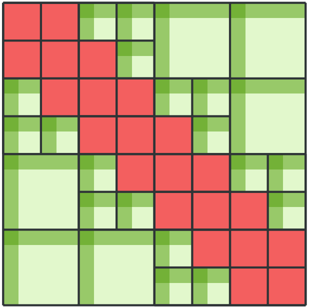
```
procedure LU(A, L, U)
  if A is block matrix then
    LU( A_00, L_00, U_00 );
    SOLVELL( A_01, L_00, U_01 );
    SOLVEUR( A_10, L_10, U_00 );
    MULTIPLY( −1, L_10, U_01, A_11 );
    LU( A_11, L_11, U_11 );
  else
    A = LU;
```

All $\mathcal{H}$-matrix arithmetic functions have computational complexity of $\mathcal{O}(n \log^\alpha n)$.

$\mathcal{H}$-Matrix Variants

$\mathcal{H}^2$-Matrices

In $\mathcal{H}$-matrices all low-rank blocks have individual row/column bases.

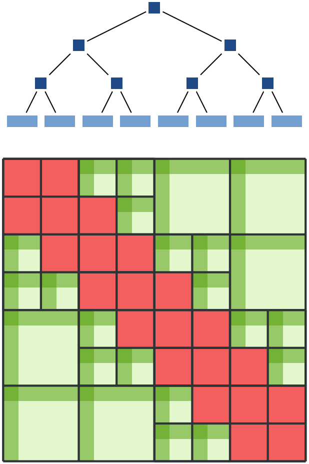


$\mathcal{H}$-Matrix Variants

$\mathcal{H}^2$-Matrices

In $\mathcal{H}$-matrices all low-rank blocks have individual row/column bases.

In $\mathcal{H}^2$-matrices, a single row/column basis for all blocks with the same row/column cluster is used instead. Furthermore, these row/column bases are nested.



$\mathcal{H}$-Matrix Variants

$\mathcal{H}^2$-Matrices

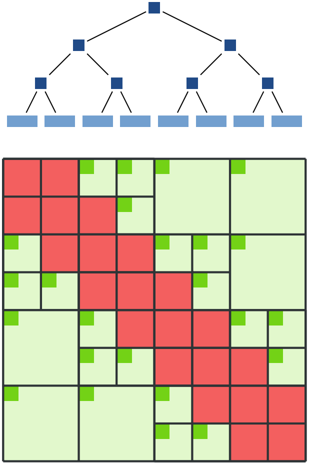
In $\mathcal{H}$-matrices all low-rank blocks have individual row/column bases.

In $\mathcal{H}^2$-matrices, a single row/column basis for all blocks with the same row/column cluster is used instead. Furthermore, these row/column bases are nested.

With this, matrix coefficients in the $\mathcal{H}^2$-matrix are stored with $k \times k$ matrices per low-rank block.

Storage complexity is reduced to $\mathcal{O}(n)$ and computational complexity to $\mathcal{O}(n \log n)$.

However, $\mathcal{H}^2$-arithmetic is more complicated.

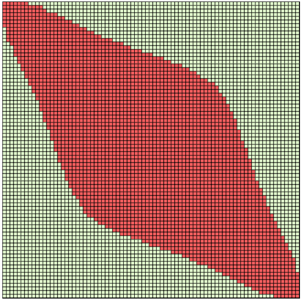


$\mathcal{H}$-Matrix Variants

Block Low-Rank (BLR)

No hierarchy is used, e.g., dense and low-rank blocks are on a single level.

Simplified arithmetic, e.g., also on distributed systems, but $\mathcal{O}\left(n^2\right)$ storage and computational complexity.



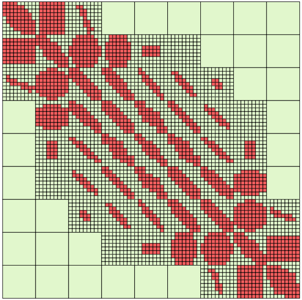
$\mathcal{H}$-Matrix Variants

Block Low-Rank (BLR)

No hierarchy is used, e.g., dense and low-rank blocks are on a single level.

Simplified arithmetic, e.g., also on distributed systems, but $\mathcal{O}\left(n^2\right)$ storage and computational complexity.

A generalisation of BLR is *Multi-Level BLR* which introduces a predefined number of hierarchy levels independent on the problem dimension.



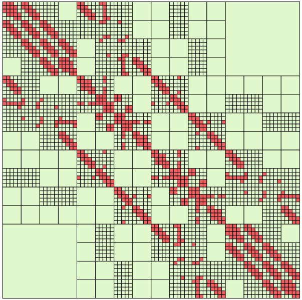
$\mathcal{H}$-Matrix Variants

Block Low-Rank (BLR)

No hierarchy is used, e.g., dense and low-rank blocks are on a single level.

Simplified arithmetic, e.g., also on distributed systems, but $\mathcal{O}\left(n^2\right)$ storage and computational complexity.

A generalisation of BLR is *Multi-Level BLR* which introduces a predefined number of hierarchy levels independent on the problem dimension.

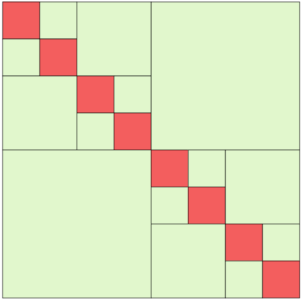


$\mathcal{H}$-Matrix Variants

HODLR

In the HODLR format, all off-diagonal blocks are handled as low-rank matrices.

Simplified arithmetic, but rank is dependent on n.

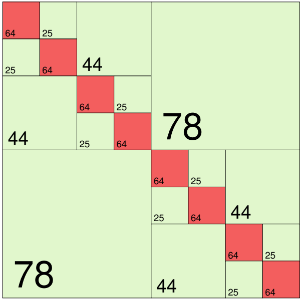


$\mathcal{H}$-Matrix Variants

HODLR

In the HODLR format, all off-diagonal blocks are handled as low-rank matrices.

Simplified arithmetic, but rank is dependent on n.



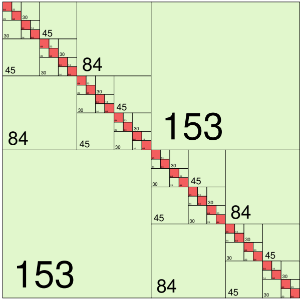
Laplace SLP, $n = 512$

$\mathcal{H}$-Matrix Variants

HODLR

In the HODLR format, all off-diagonal blocks are handled as low-rank matrices.

Simplified arithmetic, but rank is dependent on n.



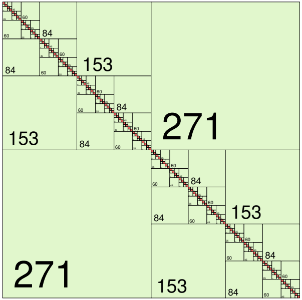
Laplace SLP, $n = 2048$

$\mathcal{H}$-Matrix Variants

HODLR

In the HODLR format, all off-diagonal blocks are handled as low-rank matrices.

Simplified arithmetic, but rank is dependent on n.



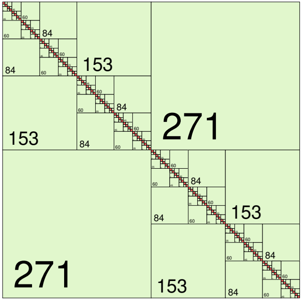
Laplace SLP, $n = 8192$

$\mathcal{H}$-Matrix Variants

HODLR

In the HODLR format, all off-diagonal blocks are handled as low-rank matrices.

Simplified arithmetic, but rank is dependent on n.



Laplace SLP, $n = 8192$

HSS

Same block layout as HODLR but based on $\mathcal{H}^2$-matrices.

Enables efficient $\mathcal{H}^2$-arithmetic but same rank problems as HODLR format.

Parallel $\mathcal{H}$-Arithmetic

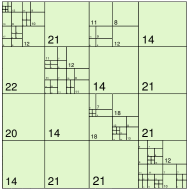
Hardware Architecture

Todays computing landscape consists of two implementations of a *many core* architecture: CPUs with up to 32 (72) cores or GPUs with $\mathcal{O}\left(10^3\right)$ cores.

$\mathcal{H}$-matrices on GPUs

General $\mathcal{H}$-matrices and $\mathcal{H}$-arithmetic have properties not best suited for GPUs:

- many different sized memory blocks (different rank, block sizes),
- not a priori known data sizes (rank after truncation unknown),
- updates to global data of different sizes, e.g., $\mathcal{H}$-LU,
- more involved algorithms, e.g. SVD.



So, either inefficient $\mathcal{H}$-matrix properties (constant rank, equal block sizes, BLR format) or inefficient GPU algorithms can be used.

In the following, we consider only (multiple) many-core CPUs.

Programming Model

Classical $\mathcal{H}$-matrix algorithms are formulated based on their block structure, which leads to *recursive* algorithms.

```
procedure LU(A, L, U)
  if A is block matrix then
    LU( A00, L00, U00 );
    SolveLL( A01, L00, U01 );
    SolveUR( A10, L10, U00 );
    Multiply( −1, L10, U01, A11 );
    LU( A11, L11, U11 );
  else
    A = LU;
```

```
procedure SolveLL(A, L, B)
  if A, L, B are block matrices then
    SolveLL( A0,0, L0,0, B0,0 );
    SolveLL( A0,1, L0,0, B0,1 );
    Multiply( −1, L1,0, B0,0, A1,0 );
    Multiply( −1, L1,0, B0,1, A1,1 );
    SolveLL( A1,0, L1,1, B1,0 );
    SolveLL( A1,1, L1,1, B1,1 );
  else
    LB = A;
```

While making programming very simple, it is inefficient on many core CPUs due to artificial *synchronisations* during runtime.

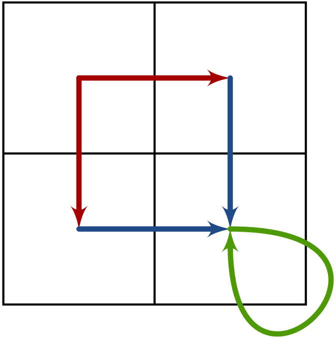
Only relies on matrix multiplication for efficient parallelization.

Programming Model

Instead, these algorithms are used to identify the basic computational *tasks* and their *dependencies*, which form a *directed acyclic graph* (DAG).

The DAG is *refined* based on the block-wise dependencies.

procedure LU(A, L, U)
  if A is a block matrix **then**
    task(LU(A_{00}, L_{00}, U_{00}));
    task(SolveLL(A_{01}, L_{00}, U_{01}));
    task(SolveUR(A_{10}, L_{10}, U_{00}));
    task(Multiply($-1, L_{10}, U_{01}, A_{11}$));
    task(LU(A_{11}, L_{11}, U_{11}));
  else
    $L \cdot U = A$;

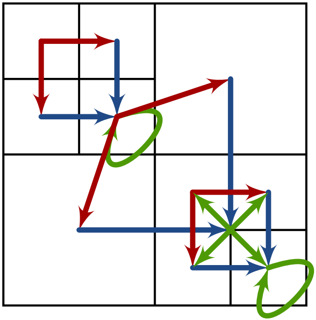


Programming Model

Instead, these algorithms are used to identify the basic computational *tasks* and their *dependencies*, which form a *directed acyclic graph* (DAG).

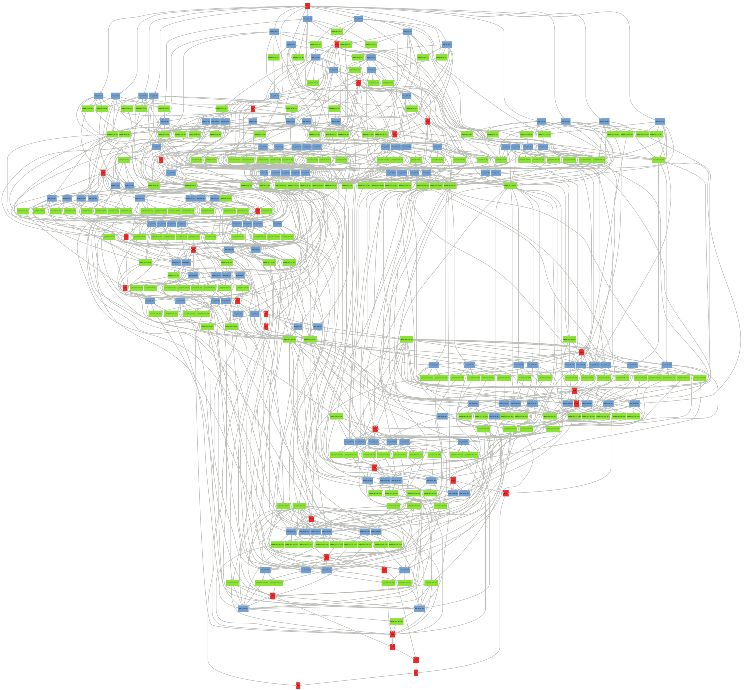
The DAG is *refined* based on the block-wise dependencies.

procedure LU(A, L, U)
  if A is a block matrix **then**
    task(LU(A_{00}, L_{00}, U_{00}));
    task(SolveLL(A_{01}, L_{00}, U_{01}));
    task(SolveUR(A_{10}, L_{10}, U_{00}));
    task(Multiply($-1, L_{10}, U_{01}, A_{11}$));
    task(LU(A_{11}, L_{11}, U_{11}));
  else
    $L \cdot U = A$;



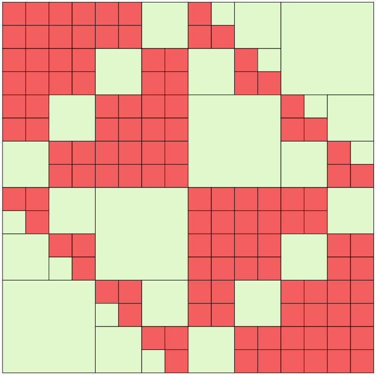
Using this DAG for a task runtime system, $\mathcal{H}$-arithmetic can efficiently be scheduled to many core systems.

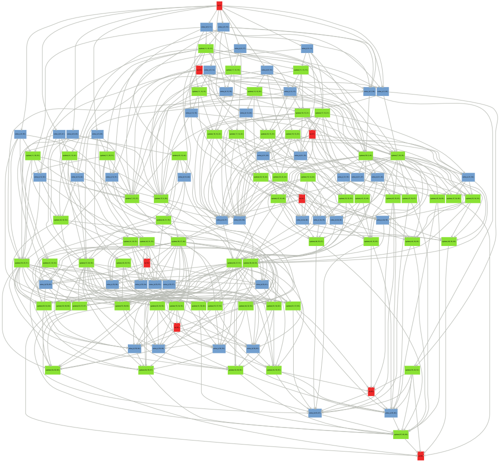
Programming Model



Programming Model

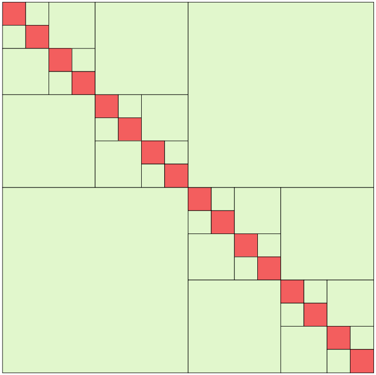
The parallel degree of this DAG strongly depends on the structure of the $\mathcal{H}$-matrix.

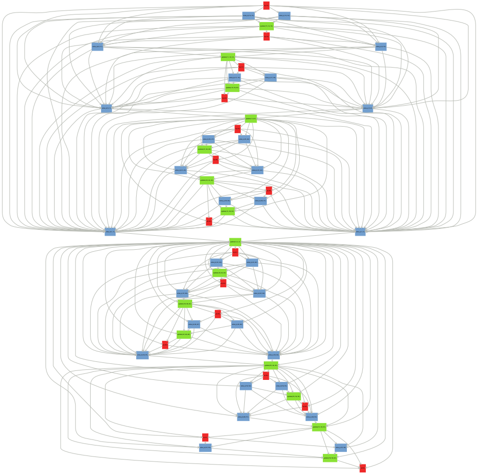




Programming Model

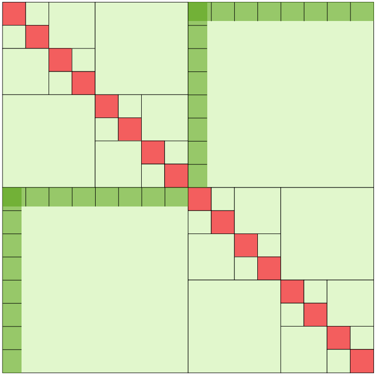
The parallel degree of this DAG strongly depends on the structure of the $\mathcal{H}$-matrix.

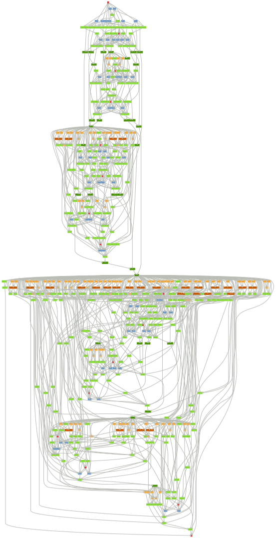




Programming Model

The parallel degree of this DAG strongly depends on the structure of the $\mathcal{H}$-matrix.





$\mathcal{H}$-LU Factorization

Numerical Results ($n = 131.072$)

Xeon 8176

# Cores	t in sec	Speedup	Reference
28	30.68	17.22	10.29
56	17.78	29.72	17.00

Epyc 7601

# Cores	t in sec	Speedup	Reference
32	37.01	28.27	25.74
64	24.91	42.01	43.27

KNL 7210

# Cores	t in sec	Speedup	Reference
64	86.09	36.8	24.02

(Reference: Dense LU factorization with Intel MKL)

Compression

$\mathcal{H}$-matrix construction can be performed *independently* for *all* matrix blocks of the $\mathcal{H}$-matrix, e.g., trivially parallelizable.

for all blocks $t \times s$ **do**
  if $t \times s$ is low-rank **then**
    task(compute compression);
  else
    task(compute dense);

Furthermore, depending on the low-rank approximation scheme, further vectorization and parallelization is possible *within* a matrix block.

Compression

$\mathcal{H}$-matrix construction can be performed *independently* for *all* matrix blocks of the $\mathcal{H}$-matrix, e.g., trivially parallelizable.

```
for all blocks t × s do
    if t × s is low-rank then
        task(compute compression);
    else
        task(compute dense);
```

Furthermore, depending on the low-rank approximation scheme, further vectorization and parallelization is possible *within* a matrix block.

Numerical Results ($n = 131.072$)

Xeon 8176

# Cores	t in sec	Speedup
28	47.45	18.88
56	24.29	36.89
112	16.86	53.15

Epyc 7601

# Cores	t in sec	Speedup
32	17.86	31.43
64	9.28	60.48
128	7.46	75.24

KNL 7210

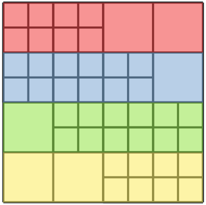
# Cores	t in sec	Speedup
64	22.67	59.22
128	18.09	74.21

Matrix-Vector Multiplication

For $Mx = y$ per-block computations can also be performed independently. Only the update of y requires synchronisation.

for all blocks $t \times s$ of M **do**
  if $t \times s$ is low-rank **then**
    task($t := B^T x|_s; y' = At;$);
  else
    task($y' = M|_{t\times s} x|_s;$);
  task($y|_t := y|_t + y';$);

To minimize this, the operations per CPU core can be scheduled based on the row indices.

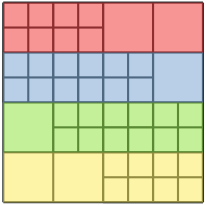


Matrix-Vector Multiplication

For $Mx = y$ per-block computations can also be performed independently. Only the update of y requires synchronisation.

for all blocks $t \times s$ of M **do**
  if $t \times s$ is low-rank **then**
    task($t := B^T x|_s; y' = At$;);
  else
    task($y' = M|_{t \times s} x|_s$;);
  task($y|_t := y|_t + y'$;);

To minimize this, the operations per CPU core can be scheduled based on the row indices.



Numerical Results ($n = 131.072$)

Xeon 8176

# Cores	t in sec	Speedup
28	4.99_{10}-2	9.01
56	4.85_{10}-2	9.23

Epyc 7601

# Cores	t in sec	Speedup
32	6.97_{10}-2	9.31
64	6.89_{10}-2	9.41

KNL 7210

# Cores	t in sec	Speedup
64	2.90_{10}-2	44.61
128	2.65_{10}-2	48.77

KNC 5110

# Cores	t in sec	Speedup
120	1.72_{10}-2	113.55

Literature

W. Hackbusch,
A sparse matrix arithmetic based on $\mathcal{H}$-matrices. I. Introduction to $\mathcal{H}$-matrices,
Computing, 62(2), pp. 89–108, 1999.

W. Hackbusch, B. Khoromskij, S. Sauter,
On $\mathcal{H}^2$-matrices,
Lecture Notes on Applied Mathematics, Springer, 2000.

M. Bebendorf,
Approximation of boundary element matrices,
Numerisch Mathematik, 86, pp. 565–589, 2000.

Z. Sheng, P. Dewilde, S. Chandrasekaran,
Algorithms to solve Hierarchically Semi-separable Systems,
Operator Theory: Advances and Applications, 176, pp. 255–294, 2007.

C. Weisbecker,
Improving multifrontal solvers by means of algebraic Block Low-Rank representations,
PhD thesis, 2013.

S. Ambikasaran
Fast algorithms for dense numerical linear algebra and applications,
PhD thesis, 2013.

R. Kriemann,
$\mathcal{H}$-LU Factorization on Many-Core Systems,
Computing and Visualization in Science, 16, pp. 105-117, 2013.

S. Börm, S. Christophersen,
Approximation of BEM matrices using GPGPUs,
CVS, accepted.

HLib pro

hlibpro.com