Parallel $\mathcal{H}$-Arithmetic

On Many-Core Systems and Beyond

Ronald Kriemann
MPI MIS

European ExaScale Applications Workshop

School of Mathematics,
University of Manchester

2016-10-11/12

Hierarchical Matrices

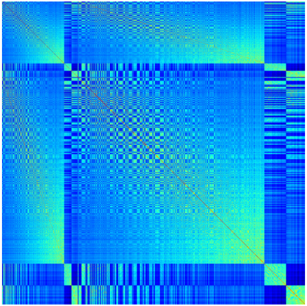
Hierarchical Matrices

In hierarchical matrices (*$\mathcal{H}$-matrices*) the indexset I of a given dense matrix $M^{I \times I}$ is reordered to expose the (numerical) low-rank structure of subblocks of M.

Hierarchical Matrices

In hierarchical matrices (*$\mathcal{H}$-matrices*) the indexset I of a given dense matrix $M^{I \times I}$ is reordered to expose the (numerical) low-rank structure of subblocks of M.

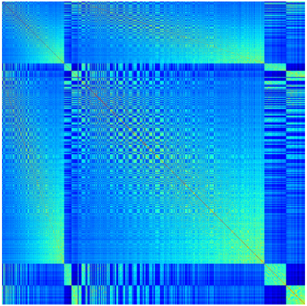
Example: Helmholtz Integral Equation



Hierarchical Matrices

In hierarchical matrices (*$\mathcal{H}$-matrices*) the indexset I of a given dense matrix $M^{I\times I}$ is reordered to expose the (numerical) low-rank structure of subblocks of M.

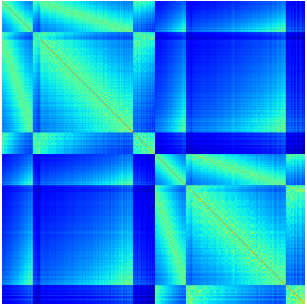
Example: Helmholtz Integral Equation

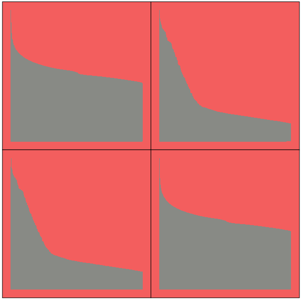


Hierarchical Matrices

In hierarchical matrices (*$\mathcal{H}$-matrices*) the indexset I of a given dense matrix $M^{I \times I}$ is reordered to expose the (numerical) low-rank structure of subblocks of M.

Example: Helmholtz Integral Equation

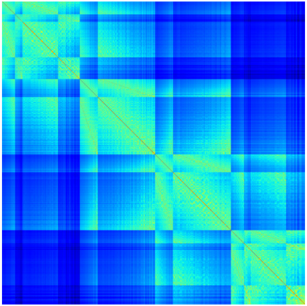


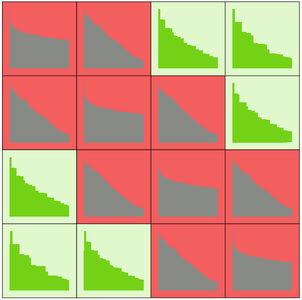


Hierarchical Matrices

In hierarchical matrices (*$\mathcal{H}$-matrices*) the indexset I of a given dense matrix $M^{I \times I}$ is reordered to expose the (numerical) low-rank structure of subblocks of M.

Example: Helmholtz Integral Equation



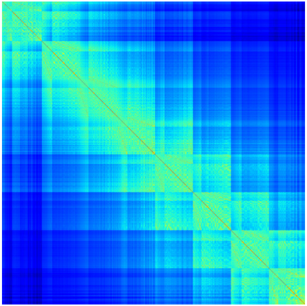


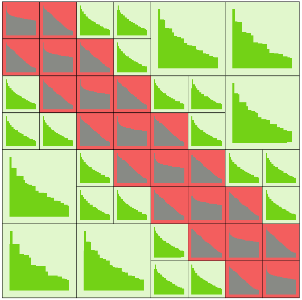
Subblocks $t \times s$ of M with rank k approximations are represented by $M|_{t \times s} = A \cdot B^T$, with $\#t \times k$-matrix A and $\#s \times k$-matrix B.

Hierarchical Matrices

In hierarchical matrices (*$\mathcal{H}$-matrices*) the indexset I of a given dense matrix $M^{I \times I}$ is reordered to expose the (numerical) low-rank structure of subblocks of M.

Example: Helmholtz Integral Equation



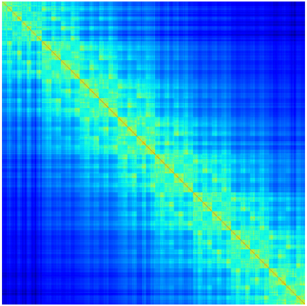


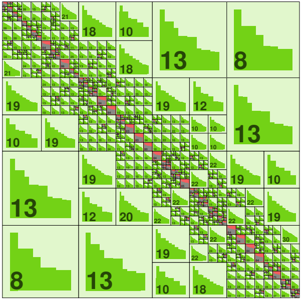
Subblocks $t \times s$ of M with rank k approximations are represented by $M|_{t \times s} = A \cdot B^T$, with $\#t \times k$-matrix A and $\#s \times k$-matrix B.

Hierarchical Matrices

In hierarchical matrices (*$\mathcal{H}$-matrices*) the indexset I of a given dense matrix $M^{I \times I}$ is reordered to expose the (numerical) low-rank structure of subblocks of M.

Example: Helmholtz Integral Equation





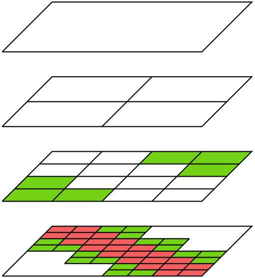
Subblocks $t \times s$ of M with rank k approximations are represented by $M|_{t \times s} = A \cdot B^T$, with $\#t \times k$-matrix A and $\#s \times k$-matrix B.

Hierarchical Matrices

(Recursive) Block Structure

The *clustering* (reordering) defines a hierarchical partitioning for $I \times I$.

Only blocks of the partition are represented in the $\mathcal{H}$-matrix, either as a dense matrix, a low-rank matrix or a block matrix (with further subblocks).

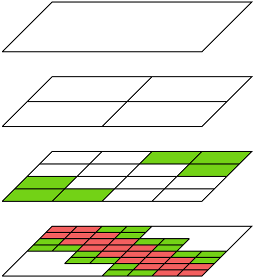


Hierarchical Matrices

(Recursive) Block Structure

The *clustering* (reordering) defines a hierarchical partitioning for $I \times I$.

Only blocks of the partition are represented in the $\mathcal{H}$-matrix, either as a dense matrix, a low-rank matrix or a block matrix (with further subblocks).



$\mathcal{H}$-Arithmetic

Complete matrix arithmetic is possible, e.g., addition, multiplication, inversion, LU factorization (recursive, block-wise operations)

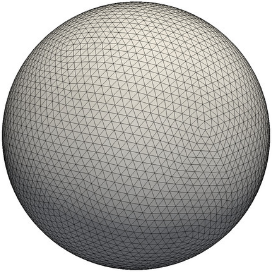
$\mathcal{H}$-arithmetic is *approximative*. Low-rank subblocks are *truncated* to rank k (precision ε) after each (sub-) operation.

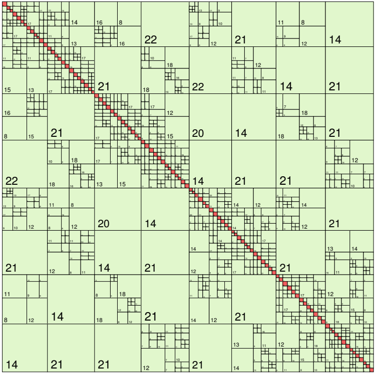
$\mathcal{H}$-arithmetic has $\mathcal{O}\left(n \log^\alpha n\right)$ complexity.

No pivoting possible due to fixed block structure.

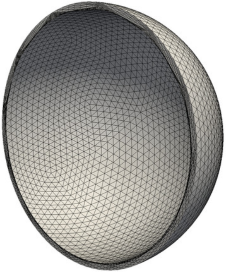
Hierarchical Matrices

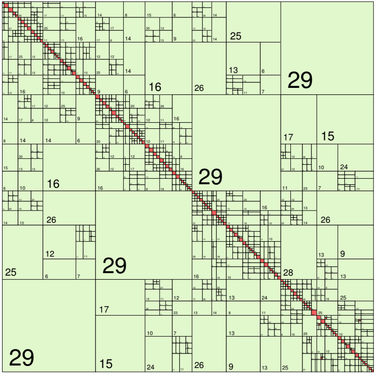
Structure depends on Geometry





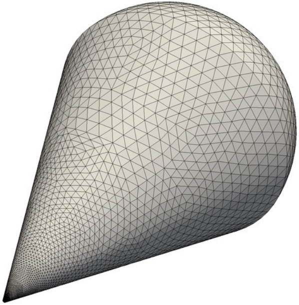
Structure depends on Geometry

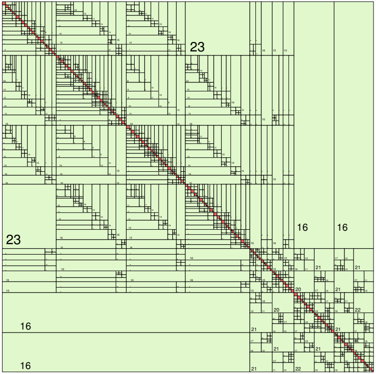




Hierarchical Matrices

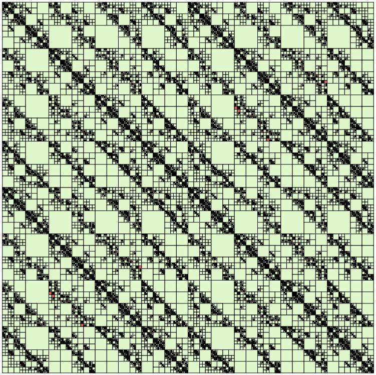
Structure depends on Geometry

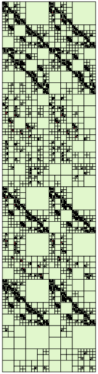




Hierarchical Matrices

Structure depends on Geometry/Problem





Implementation

Implementation

All $\mathcal{H}$-matrix algorithms are implemented in the library HLIBpro.

HLIBpro

- HLIBpro implements an extensive set of $\mathcal{H}$-matrix algorithms,
- was developed using C++ since the beginning,
- various parallel APIs used in the past (Pthreads, MPI, OpenMP).

On multi-/many-core CPUs *Threading Building Blocks* (TBB) is used for parallelisation.

TBB

- open source software library for C++
- implements various forms of loop-parallelisation
- is based on tasks and exposes this for task based computations,
- permits seemless integration with C++11 via lambda functions.

Implementation

OpenMP?

- Tasks not available in OpenMP 2.5 when task based $\mathcal{H}$-arithmetic was developed,
- not all C++ compilers fully support(-ed?) OpenMP,
- Tasks and task dependencies are fixed at compile time at source code level (TBB: at runtime).

Implementation

OpenMP?

- Tasks not available in OpenMP 2.5 when task based $\mathcal{H}$-arithmetic was developed,
- not all C++ compilers fully support(-ed?) OpenMP,
- Tasks and task dependencies are fixed at compile time at source code level (TBB: at runtime).

Problems

- Known deadlock issue in TBB with recursive parallelisation and mutices in inner loop

task $\text{APPLY\_UPDATE}(U, M)$
  lock mutex(M);
  spawn sub task applying U to M;
  unlock mutex(M);

Implementation

OpenCL/CUDA?

- $\mathcal{H}$-matrix algorithms work on an extremely heterogenous data
  - up to several million sub blocks
  - block sizes from $10..10^6$,
  - different rank per block
- low-rank truncation involves QR ($\mathcal{O}(n)$), SVD ($\mathcal{O}(k)$), gemm ($\mathcal{O}(n)$) up to several thousand times per block,
- for batch operations: need to fix rank/block sizes, loose memory eff./accuracy,
- can efficiently be used for evaluation of quadrature rules during construction.

$\mathcal{H}$-Matrix Construction

Algorithm

All subblocks can be built independently.

procedure BUILD($t \times s$)
  if $t \times s$ is leaf **then**
    build dense/low-rank block
  else
    parallel for all sub blocks $t' \times s'$ **do**
      build($t' \times s'$);

$\mathcal{H}$-Matrix Construction

Algorithm

All subblocks can be built independently.

```
mat_build ( Block * b ) {
  parallel_for( blocked_range2d( 0, nbrows, 0, nbcols ),
    [...] ( const blocked_range2d &  r ) {
      for ( auto  i = r.rows().begin(); i != r.rows().end(); ++i )
        for ( auto  j = r.cols().begin(); j != r.cols().end(); ++j )
          mat_build( b->son( i, j ) ); } ); }
```

Scheduling by TBB respects CPU core locality.

$\mathcal{H}$-Matrix Construction

Algorithm

All subblocks can be built independently.

```
mat_build ( Block * b ) {
  parallel_for( blocked_range2d( 0, nbrows, 0, nbcols ),
    [...] ( const blocked_range2d &  r ) {
      for ( auto  i = r.rows().begin(); i != r.rows().end(); ++i )
        for ( auto  j = r.cols().begin(); j != r.cols().end(); ++j )
          mat_build( b->son( i, j ) ); } ); }
```

Scheduling by TBB respects CPU core locality.

Numerical Results (Sequential)

n	t in sec	$\frac{t}{n \log n}$	Mem in MB	$\frac{\text{Mem}}{n \log n}$
10,720	46.4	3.24	186	1.30
42,880	207.8	3.15	904	1.37
171,520	872.6	2.93	4,290	1.44
686,080	3689.4	2.77	19,810	1.49

(E7-8857)

$\mathcal{H}$-Matrix Construction

Algorithm

All subblocks can be built independently.

```
mat_build ( Block * b ) {
  parallel_for( blocked_range2d( 0, nbrows, 0, nbcols ),
    [...] ( const blocked_range2d &  r ) {
      for ( auto  i = r.rows().begin(); i != r.rows().end(); ++i )
        for ( auto  j = r.cols().begin(); j != r.cols().end(); ++j )
          mat_build( b->son( i, j ) ); } ); }
```

Scheduling by TBB respects CPU core locality.

Numerical Results (Parallel)

	Cores	Time	Speedup
E7-8857	12	69.4s	10.23
	48	18.0s	39.36
KNL 7210	64	24.0s	87.89

$\mathcal{H}$-LU Factorization

The $\mathcal{H}$-LU factorisation $A = LU$ is defined by:

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

which leads to the following equations and *recursive* algorithm

$$\begin{aligned} A_{00} &= L_{00} U_{00} \\ A_{01} &= L_{00} U_{01} \\ A_{10} &= L_{10} U_{00} \\ A_{11} &= A_{11} - L_{10} U_{01} \\ A_{11} &= L_{11} U_{11} \end{aligned}$$

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

$\mathcal{H}$-LU Factorization

The $\mathcal{H}$-LU factorisation $A = LU$ is defined by:

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

which leads to the following equations and *recursive* algorithm

$$\begin{aligned} A_{00} &= L_{00}U_{00} \\ A_{01} &= L_{00}U_{01} \\ A_{10} &= L_{10}U_{00} \\ A_{11} &= A_{11} - L_{10}U_{01} \\ A_{11} &= L_{11}U_{11} \end{aligned}$$

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

Recursive algorithm is not optimal for parallelisation.

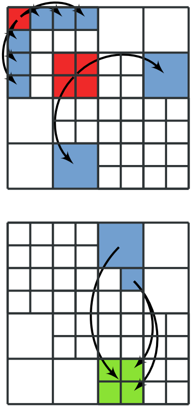
$\mathcal{H}$-LU Factorization

Parallel $\mathcal{H}$-LU

Tasks for sub-operations together with dependencies between them are defined, yielding a DAG:

procedure $\mathrm{LU}(\ A|_{t\times t},\ L|_{t\times t}, U|_{t\times t}\ )$
  if A is block matrix **then**
    for $i \in \{0,1\}$ **do**
      task$(\mathrm{LU}(\ A|_{t_i\times t_i}\ ));\ \ell := \mathrm{level}(t_i);$
      for $s \in T^\ell(I), s >_I t_i$ **do**
        if $A|_{s\times t_i}$ is not blocked **then**
          task$(\textsc{SolveUR}(\ A|_{s\times t_i}, L|_{s\times t_i}, U|_{t_i\times t_i}\ ));$
        if $A|_{t_i\times s}$ is not blocked **then**
          task$(\textsc{SolveLL}(\ A|_{t_i\times s}, L|_{t_i\times t_i}, U|_{t_i\times s}\ ));$
      for $s, r \in T^\ell(I), s, r >_I t_i$ **do**
        if $L|_{r\times t_i}$, $U_{t_i\times s}$ or $A|_{r\times s}$ is not blocked **then**
          task$(\textsc{Multiply}(-1, L_{r\times t_i}, U_{t_i\times s}, A|_{r\times s}));$
  else
    task$(A := LU);$

Dependencies:



$\mathcal{H}$-LU Factorization

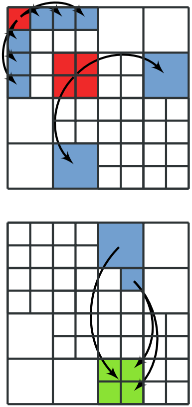
Parallel $\mathcal{H}$-LU

Tasks for sub-operations together with dependencies between them are defined, yielding a DAG:

```
class LU : public tbb::task {
  task * execute () {
    factorize( A );
    for ( auto M : matrices_right_of( A ) )
      if ( solve_task(M)->dec_ref_count() == 0 )
        spawn( solve_task(M) );
} };

class SolveLL : public tbb::task {
  task * execute () {
    solve( L, X );
    for ( auto M : matrices_below( X ) )
      if ( update_task(M)->dec_ref_count() == 0 )
        spawn( update_task(M) );
} };
```

Dependencies:



$\mathcal{H}$-LU Factorization

Numerical Results (Sequential)

n	t in sec	$\frac{t}{n \log^3 n}$	Mem in MB	$\frac{\text{Mem}}{n \log n}$
2,680	5.9	1.49	30	0.98
10,720	48.4	1.88	182	1.27
42,880	266.9	1.71	887	1.34
171,520	1636.2	1.81	4,220	1.41
686,080	8835.4	1.77	20,010	1.50

(E7-8857)

$\mathcal{H}$-LU Factorization

Numerical Results (Sequential)

n	t in sec	$\frac{t}{n \log^3 n}$	Mem in MB	$\frac{\text{Mem}}{n \log n}$
2,680	5.9	1.49	30	0.98
10,720	48.4	1.88	182	1.27
42,880	266.9	1.71	887	1.34
171,520	1636.2	1.81	4,220	1.41
686,080	8835.4	1.77	20,010	1.50

(E7-8857)

Numerical Results (Parallel)

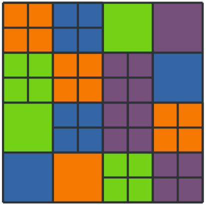
	Parallel		
	#Cores	Time	Speedup
E7-8857	12	132.6s	10.89
	48	38.8s	37.24
KNL 7210	64	144.2s	59.60

And Beyond: Distributed Memory

And Beyond: Distributed Memory

Simple Arithmetic

Algorithms with (mostly) independent operations are implemented using MPI (construction, MVM, addition).



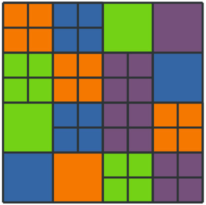
Problem: load balancing. Cost per block is only roughly known (depends on rank).

And Beyond: Distributed Memory

Simple Arithmetic

Algorithms with (mostly) independent operations are implemented using MPI (construction, MVM, addition).

Problem: load balancing. Cost per block is only roughly known (depends on rank).

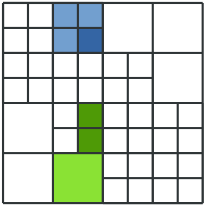


$\mathcal{H}$-LU factorization

Communication pattern similar to dense LU.

However, subblocks are used on different levels of the hierarchy.

Wanted: bring task approach to distributed memory with efficient task scheduling (handling communication).



Handling of Large Blocks

For large low-rank blocks $M|_{t \times s} = A \cdot B^T, \min\{\#t, \#s\} \geq n_{large}$, $n_{large} > n/p$ need further parallelization of A and B.

procedure TRUNCATE(A, B)

$[Q_A, R_A] = \mathrm{qr}(A);$

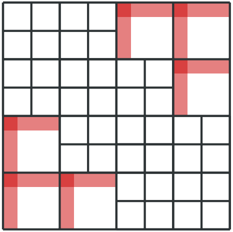
$[Q_B, R_B] = \mathrm{qr}(B);$

$[U, S, V] = \mathrm{svd}(R_A R_B^T);$

$k' := \mathrm{new\_rank}(S);$

$A' := (Q_A U S)(1 : k', :);$

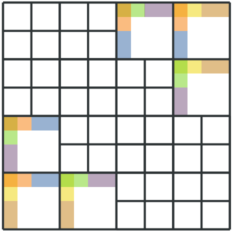
$B' := (Q_B V)(1 : k', :);$



Handling of Large Blocks

For large low-rank blocks $M|_{t \times s} = A \cdot B^T, \min\{\#t, \#s\} \geq n_{large}$, $n_{large} > n/p$ need further parallelization of A and B.

procedure $\textsc{Truncate}(A, B)$
- $[Q_A, R_A] = \mathrm{qr}(A);$
- $[Q_B, R_B] = \mathrm{qr}(B);$
- $[U, S, V] = \mathrm{svd}(R_A R_B^T);$
- $k' := \mathrm{new\_rank}(S);$
- $A' := (Q_A U S)(1 : k', :);$
- $B' := (Q_B V)(1 : k', :);$

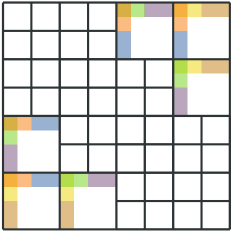


And Beyond: Distributed Memory

Handling of Large Blocks

For large low-rank blocks $M|_{t \times s} = A \cdot B^T, \min\{\#t, \#s\} \geq n_{large}$, $n_{large} > n/p$ need further parallelization of A and B.

procedure $\textsc{Truncate}(A, B)$
$[Q_A, R_A] = \mathrm{qr}(A);$
$[Q_B, R_B] = \mathrm{qr}(B);$
$[U, S, V] = \mathrm{svd}(R_A R_B^T);$
$k' := \mathrm{new\_rank}(S);$
$A' := (Q_A U S)(1 : k', :);$
$B' := (Q_B V)(1 : k', :);$



Introduces additional *synchronization* (e.g., during QR).