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 $M$. 

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 $B$. 

Kriemann, »Parallel $\mathcal{H}$-Arithmetic«
Hierarchical Matrices

In hierarchical matrices ($\mathcal{H}$-matrices) the index set $I$ 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 (\textit{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 (\textit{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 (\textit{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 (\textit{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 (\textit{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 $B$. 

Kriemann, »Parallel \textit{H}-Arithmetic«
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 $H$-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 $\varepsilon$) after each (sub-) operation.

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

No pivoting possible due to fixed block structure.

Kriemann, »Parallel $\mathcal{H}$-Arithmetic«
Hierarchical Matrices

Structure depends on Geometry
Hierarchical Matrices

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 seamless integration with C++11 via lambda functions.
OpenMP?

- Tasks not available in OpenMP 2.5 when task based 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 mutexes in inner loop

```plaintext
task APPLY_UPDATE(U, M)
lock mutex(M);
spawn sub task applying U to M;
unlock mutex(M);
```

Kriemann, »Parallel $\mathcal{H}$-Arithmetic«
OpenCL/CUDA?

• $H$-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 ($O(n)$), SVD ($O(k)$), gemm ($O(n)$) 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.
Algorithm

All subblocks can be built independently.

\begin{verbatim}
procedure BUILD(t x s)
    if t x s is leaf then
        build dense/low-rank block
    else
        parallel for all sub blocks t' x s' do
            build(t' x s');
\end{verbatim}
Algorithm

All subblocks can be built independently.

```c
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.
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)

<table>
<thead>
<tr>
<th>$n$</th>
<th>$t$</th>
<th>$\frac{t}{n \log n}$</th>
<th>Mem</th>
<th>$\frac{\text{Mem}}{n \log n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,720</td>
<td>46.4</td>
<td>3.24</td>
<td>186</td>
<td>1.30</td>
</tr>
<tr>
<td>42,880</td>
<td>207.8</td>
<td>3.15</td>
<td>904</td>
<td>1.37</td>
</tr>
<tr>
<td>171,520</td>
<td>872.6</td>
<td>2.93</td>
<td>4,290</td>
<td>1.44</td>
</tr>
<tr>
<td>686,080</td>
<td>3689.4</td>
<td>2.77</td>
<td>19,810</td>
<td>1.49</td>
</tr>
</tbody>
</table>

(E7-8857)
Algorithm

All subblocks can be built independently.

```c
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)

<table>
<thead>
<tr>
<th></th>
<th>Cores</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>E7-8857</td>
<td>12</td>
<td>69.4s</td>
<td>10.23</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>18.0s</td>
<td>39.36</td>
</tr>
<tr>
<td>KNL 7210</td>
<td>64</td>
<td>24.0s</td>
<td>87.89</td>
</tr>
</tbody>
</table>
The $H$-LU factorisation $A = LU$ is defined by:

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

which leads to the following equations and recursive algorithm

\[ A_{00} = L_{00}U_{00} \]
\[ A_{01} = L_{00}U_{01} \]
\[ A_{10} = L_{10}U_{00} \]
\[ A_{11} = A_{11} - L_{10}U_{01} \]
\[ A_{11} = L_{11}U_{11} \]

**procedure** $LU(A, L, U)$

- **if** $A$ is block matrix **then**
  
  $LU( A_{00}, L_{00}, U_{00} );$
  $SOLVE LL( A_{01}, L_{00}, U_{01} );$
  $SOLVE UR( A_{10}, L_{11}, U_{00} );$
  $MULT IPLY( -1, L_{10}, U_{01}, A_{11} );$
  $LU( A_{11}, L_{11}, U_{11} );$

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

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

which leads to the following equations and recursive algorithm

$$A_{00} = L_{00}U_{00},$$

$$A_{01} = L_{00}U_{01},$$

$$A_{10} = L_{10}U_{00},$$

$$A_{11} = A_{11} - L_{10}U_{01},$$

$$A_{11} = L_{11}U_{11}.$$

Procedure $\text{LU}(A, L, U)$

\[\text{if } A \text{ is block matrix then}\]

\[\text{LU}(A_{00}, L_{00}, U_{00});\]

\[\text{SOLVE}\_\text{LL}(A_{01}, L_{00}, U_{01});\]

\[\text{SOLVE}\_\text{UR}(A_{10}, L_{10}, U_{00});\]

\[\text{MULTIPLY}(-1, L_{10}, U_{01}, A_{11});\]

\[\text{LU}(A_{11}, L_{11}, U_{11});\]

\[\text{else}\]

\[A = LU;\]

Recursive algorithm is not optimal for parallelisation.
Parallel $\mathcal{H}$-LU

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

\begin{verbatim}
procedure LU( A|t×t, L|t×t, U|t×t )
  if A is block matrix then
    for i ∈ {0, 1} do
      task(LU( A|t_i×t_i )); ℓ := level(t_i);
    for s ∈ T^ℓ( I ), s > I t_i do
      if A|s×t_i is not blocked then
        task(SOLVEUR( A|s×t_i, L|s×t_i, U|t_i×t_i ));
      if A|t_i×s is not blocked then
        task(SOLVELL( A|t_i×s, L|t_i×t_i, U|t_i×s ));
    for s, r ∈ T^ℓ( I ), s, r > I t_i do
      if L|r×t_i , U_t_i×s or A|r×s is not blocked then
        task(MULTIPLY(-1, L|r×t_i, U_t_i×s, A|r×s ));
  else
    task(A := LU);
\end{verbatim}
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) );
    }
};
**H-LU Factorization**

### Numerical Results (Sequential)

<table>
<thead>
<tr>
<th>( n )</th>
<th>( t ) (in sec)</th>
<th>( \frac{t}{n \log^3 n} )</th>
<th>Mem (in MB)</th>
<th>( \frac{\text{Mem}}{n \log n} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,680</td>
<td>5.9</td>
<td>1.49</td>
<td>30</td>
<td>0.98</td>
</tr>
<tr>
<td>10,720</td>
<td>48.4</td>
<td>1.88</td>
<td>182</td>
<td>1.27</td>
</tr>
<tr>
<td>42,880</td>
<td>266.9</td>
<td>1.71</td>
<td>887</td>
<td>1.34</td>
</tr>
<tr>
<td>171,520</td>
<td>1636.2</td>
<td>1.81</td>
<td>4,220</td>
<td>1.41</td>
</tr>
<tr>
<td>686,080</td>
<td>8835.4</td>
<td>1.77</td>
<td>20,010</td>
<td>1.50</td>
</tr>
</tbody>
</table>

\((E7-8857)\)
### Numerical Results (Sequential)

<table>
<thead>
<tr>
<th>( n )</th>
<th>( t ) in sec</th>
<th>( \frac{t}{n \log^3 n} )</th>
<th>( \text{Mem} ) in MB</th>
<th>( \frac{\text{Mem}}{n \log n} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2,680</td>
<td>5.9</td>
<td>1.49</td>
<td>30</td>
<td>0.98</td>
</tr>
<tr>
<td>10,720</td>
<td>48.4</td>
<td>1.88</td>
<td>182</td>
<td>1.27</td>
</tr>
<tr>
<td>42,880</td>
<td>266.9</td>
<td>1.71</td>
<td>887</td>
<td>1.34</td>
</tr>
<tr>
<td>171,520</td>
<td>1636.2</td>
<td>1.81</td>
<td>4,220</td>
<td>1.41</td>
</tr>
<tr>
<td>686,080</td>
<td>8835.4</td>
<td>1.77</td>
<td>20,010</td>
<td>1.50</td>
</tr>
</tbody>
</table>

(E7-8857)

### Numerical Results (Parallel)

<table>
<thead>
<tr>
<th>Parallel</th>
<th>#Cores</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>E7-8857</td>
<td>12</td>
<td>132.6s</td>
<td>10.89</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>38.8s</td>
<td>37.24</td>
</tr>
<tr>
<td>KNL 7210</td>
<td>64</td>
<td>144.2s</td>
<td>59.60</td>
</tr>
</tbody>
</table>
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).
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).

$H$-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).
And Beyond: Distributed Memory

Handling of Large Blocks

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

procedure \textsc{truncate}(A, B)

\[
\begin{align*}
[Q_A, R_A] &= \text{qr}(A); \\
[Q_B, R_B] &= \text{qr}(B); \\
[U, S, V] &= \text{svd}(R_AR_B^T); \\
k' &:= \text{new\_rank}(S); \\
A' &:= (Q_AUS')(1:k',:); \\
B' &:= (Q_BV)(1:k',:); \\
\end{align*}
\]
And Beyond: Distributed Memory

Handling of Large Blocks

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

**procedure** TRUNCATE($A$, $B$)

\[
[Q_A, R_A] = qr(A);
[Q_B, R_B] = qr(B);
[U, S, V] = svd(R_A R_B^T);
\]

\[
k' := \text{new\_rank}(S);
A' := (Q_A U S)(1 : k', :);
B' := (Q_B V)(1 : k', :);
\]
Handling of Large Blocks

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

procedure TRUNCATE($A$, $B$)

\[
\begin{align*}
[Q_A, R_A] &= \text{qr}(A); \\
[Q_B, R_B] &= \text{qr}(B); \\
[U, S, V] &= \text{svd}(R_A R_B^T); \\
k' &:= \text{new\_rank}(S); \\
A' &:= (Q_A U S)(1 : k', :); \\
B' &:= (Q_B V)(1 : k', :);
\end{align*}
\]

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