$\mathcal{H}$-matrix preconditioners: Storage and runtime reduction via sparse matrices

Sven Christophersen

Department of Computer Science
Christian-Albrechts-Universität Kiel

16th GAMM-ANLA workshop

16.09.2016

1. Motivation
2. $\mathcal{H}$-matrix basics
3. Implementation
4. Compression
5. Outlook and conclusion

Motivation

- Local support of FEM basis function results in sparse matrices.

- Local support of FEM basis function results in sparse matrices.
- Inverses usually not sparse anymore, often even dense.

- Local support of FEM basis function results in sparse matrices.
- Inverses usually not sparse anymore, often even dense.
- Direct solvers unattractive due to the increase in storage and time complexity particularly for 3D problems.

- Local support of FEM basis function results in sparse matrices.
- Inverses usually not sparse anymore, often even dense.
- Direct solvers unattractive due to the increase in storage and time complexity particularly for 3D problems.
- Iterative solvers often suffer from bad condition number of the system matrix. (e.g. $\mathcal{O}(h^{-2})$ for Poisson problem).

- Local support of FEM basis function results in sparse matrices.
- Inverses usually not sparse anymore, often even dense.
- Direct solvers unattractive due to the increase in storage and time complexity particularly for 3D problems.
- Iterative solvers often suffer from bad condition number of the system matrix. (e.g. $\mathcal{O}(h^{-2})$ for Poisson problem).
  - $\rightarrow$ a preconditioner is needed, preferably one that works on a large class of matrices.

- Local support of FEM basis function results in sparse matrices.
- Inverses usually not sparse anymore, often even dense.
- Direct solvers unattractive due to the increase in storage and time complexity particularly for 3D problems.
- Iterative solvers often suffer from bad condition number of the system matrix. (e.g. $\mathcal{O}(h^{-2})$ for Poisson problem).
  - → a preconditioner is needed, preferably one that works on a large class of matrices.
  - → Approximative matrix factorizations with $\mathcal{H}$-matrices are well-suited for this problem.

BUT...

BUT...

$\mathcal{H}$-matrix-Conversion typically blows up memory requirements!

BUT...

$\mathcal{H}$-matrix-Conversion typically blows up memory requirements!

Examples:

System matrix of Poisson's equation with P1-elements and Domain decomposition clustering on the unit square / cube:

	# DoF	16.129	65.025	261.121	1.046.529	4.190.209
2D	A	0,6 MB	2,5 MB	10 MB	40 MB	160 MB
	$\mathcal{H}$-A	34 MB	168 MB	738 MB	3088 MB	12632 MB

	# DoF	3375	29.791	250.047	2.048.383
3D	A	0,2 MB	1,4 MB	12 MB	101 MB
	$\mathcal{H}$-A	11 MB	165 MB	1779 MB	16376 MB

BUT...

$\mathcal{H}$-matrix-Conversion typically blows up memory requirements!

Examples:

System matrix of Darcy's equation with Raviart-Thomas elements and Domain decomposition clustering on the unit square / cube:

	# DoF	12.288	49.152	196.608	786.432	3.145.728
2D	A	0,3 MB	1,1 MB	4,5 MB	18 MB	72 MB
	$\mathcal{H}$-A	20 MB	119 MB	554 MB	2388 MB	10127 MB

	# DoF	6.016	48.640	391.168	3.137.536
3D	A	0,3 MB	2,3 MB	19 MB	155 MB
	$\mathcal{H}$-A	14 MB	289 MB	3275 MB	30628 MB

BUT...

$\mathcal{H}$-matrix-Conversion typically blows up memory requirements!

Examples:

System matrix of Darcy's equation with Raviart-Thomas elements and Domain decomposition clustering on the unit square / cube:

	# DoF	12.288	49.152	196.608	786.432	3.145.728
2D	A	0,3 MB	1,1 MB	4,5 MB	18 MB	72 MB
	$\mathcal{H}$-A	20 MB	119 MB	554 MB	2388 MB	10127 MB

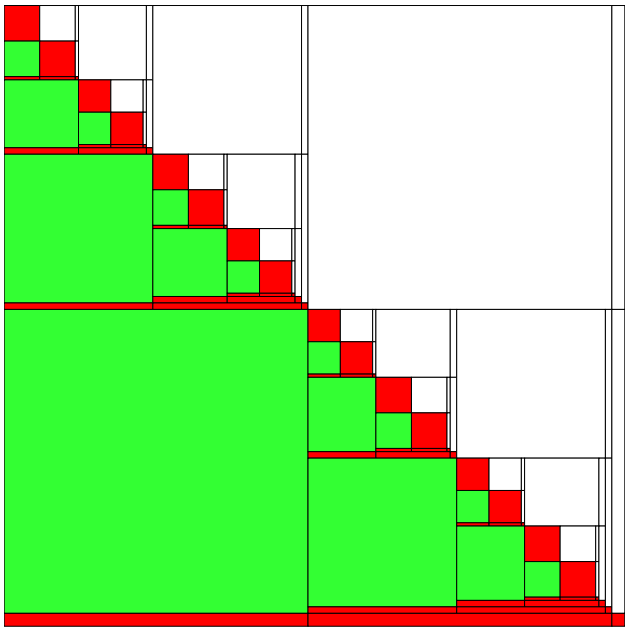
	# DoF	6.016	48.640	391.168	3.137.536
3D	A	0,3 MB	2,3 MB	19 MB	155 MB
	$\mathcal{H}$-A	14 MB	289 MB	3275 MB	30628 MB

Storage increase by a factor of 50 – 200 is unacceptable!

$\mathcal{H}$-matrix basics

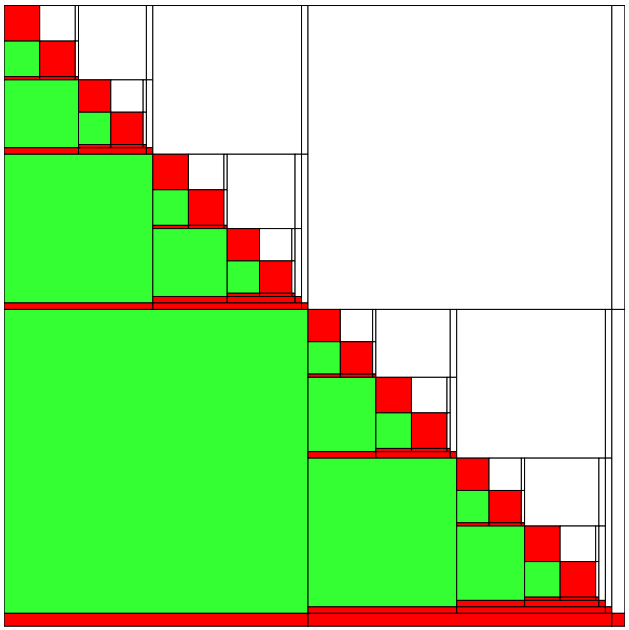
Idea:

Split index set of a matrix into subblocks which can either be approximated very well (green) or are small and can be stored directly (red).



Idea:

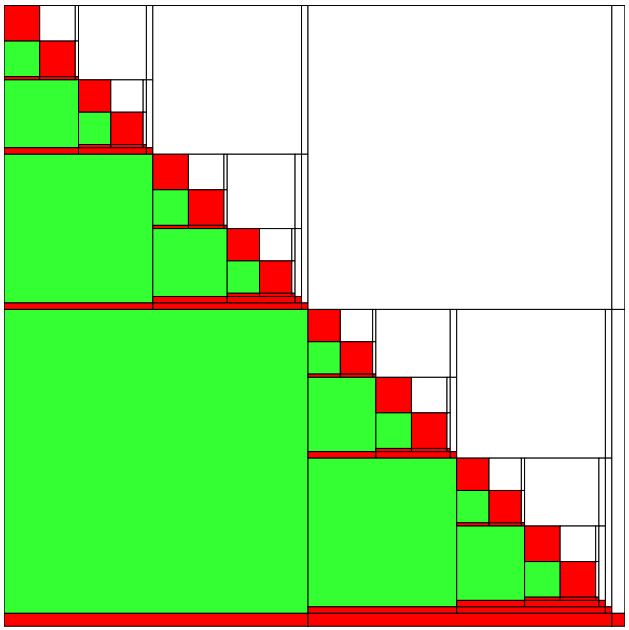
Split index set of a matrix into subblocks which can either be approximated very well (green) or are small and can be stored directly (red).



- In FEM-discretization admissible blocks have rank zero.

Idea:

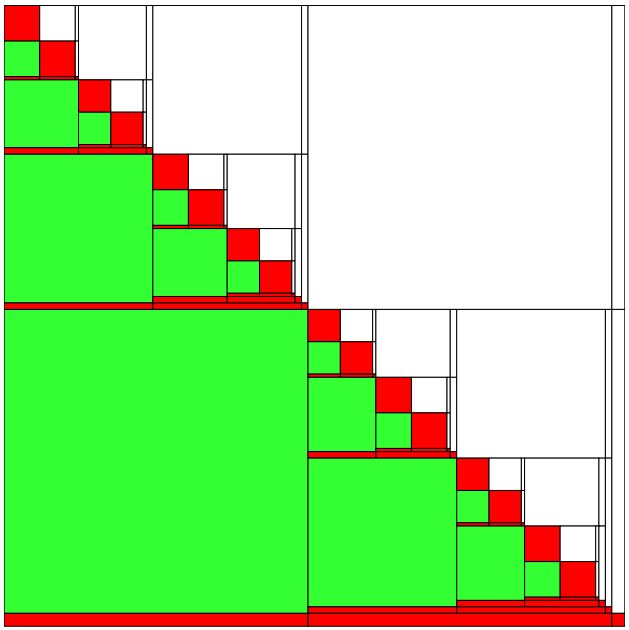
Split index set of a matrix into subblocks which can either be approximated very well (green) or are small and can be stored directly (red).



- In FEM-discretization admissible blocks have rank zero.
- Within $\mathcal{H}$-arithmetics rank zero matrices can fill up with non-zeros entries
  $\rightarrow$ low rank matrices.

Idea:

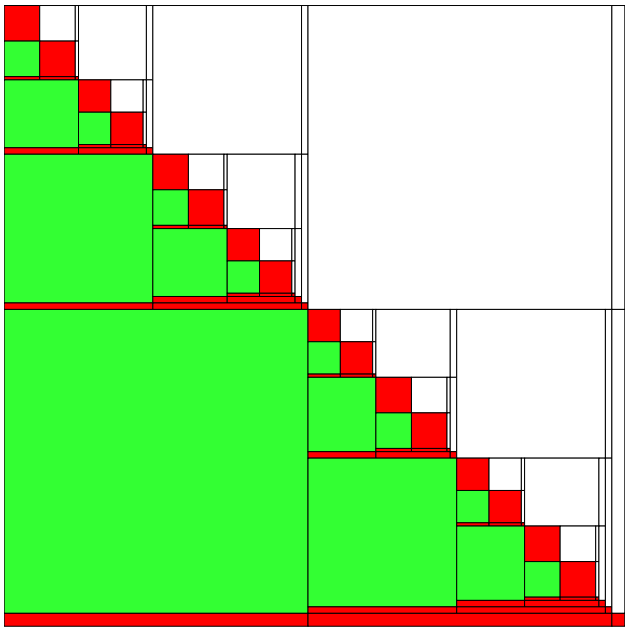
Split index set of a matrix into subblocks which can either be approximated very well (green) or are small and can be stored directly (red).



- In FEM-discretization admissible blocks have rank zero.
- Within $\mathcal{H}$-arithmetics rank zero matrices can fill up with non-zeros entries
  $\rightarrow$ low rank matrices.
- Matrix $X \in \mathbb{K}^{n \times m}$ has rank k, store as *factorized* low rank matrix $X = AB^*$ with $A \in \mathbb{K}^{n \times k}$ and $B \in \mathbb{K}^{m \times k}$, $k \ll n, m$.
  Storage: $k(n+m)$ vs. $n \cdot m$.

Idea:

Split index set of a matrix into subblocks which can either be approximated very well (green) or are small and can be stored directly (red).



- In FEM-discretization admissible blocks have rank zero.
- Within $\mathcal{H}$-arithmetics rank zero matrices can fill up with non-zeros entries
  $\rightarrow$ low rank matrices.
- Matrix $X \in \mathbb{K}^{n \times m}$ has rank k, store as *factorized* low rank matrix $X = AB^*$ with $A \in \mathbb{K}^{n \times k}$ and $B \in \mathbb{K}^{m \times k}$, $k \ll n, m$.
  Storage: $k(n+m)$ vs. $n \cdot m$.

Question:

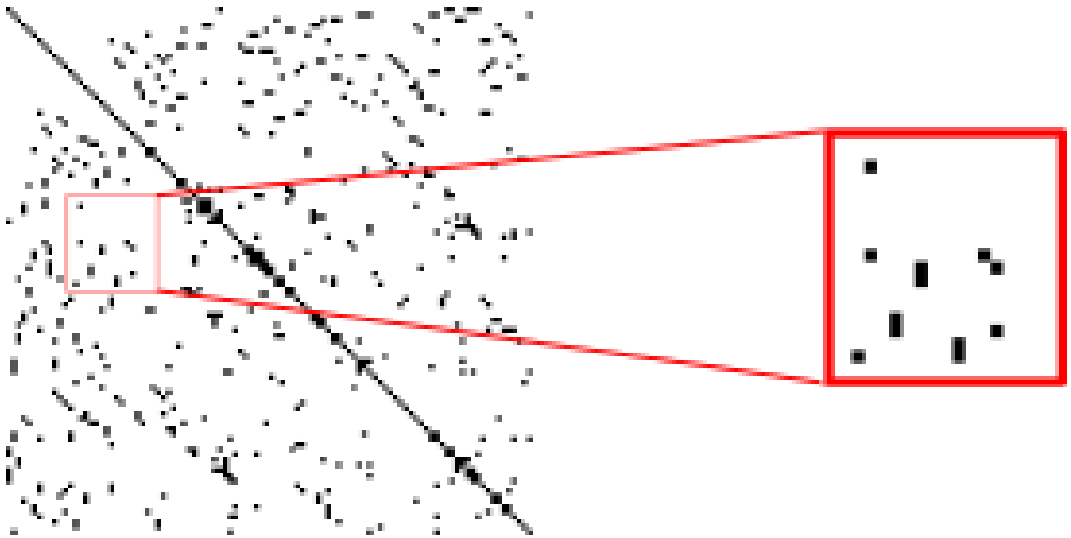
What about the *red* blocks?

Inadmissible blocks:

- For the dimension of an inadmissible block $X \in \mathbb{K}^{n \times m}$ it holds $n \leq c_{lf} \vee m \leq c_{lf}$ for some small constant $c_{lf} \in \mathbb{N}$. In most applications $c_{lf} \in \{10, \ldots, 100\}$ is a good choice.
  If we consider a *strict* block partition we even have $\min(n, m) \leq c_{lf}$.

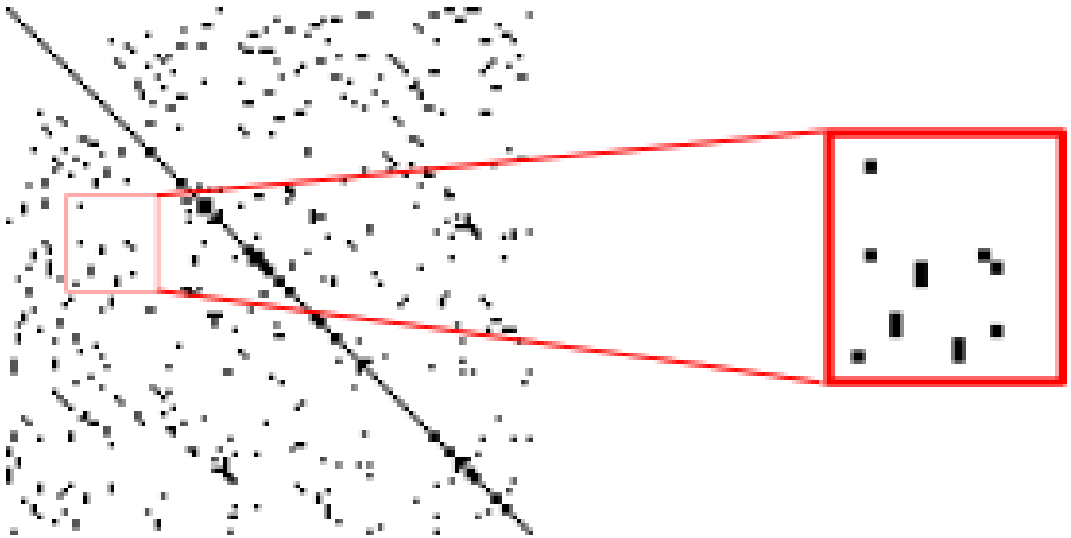
Inadmissible blocks:

- For the dimension of an inadmissible block $X \in \mathbb{K}^{n \times m}$ it holds $n \leq c_{lf} \vee m \leq c_{lf}$ for some small constant $c_{lf} \in \mathbb{N}$. In most applications $c_{lf} \in \{10, \ldots, 100\}$ is a good choice.
  If we consider a *strict* block partition we even have $\min(n, m) \leq c_{lf}$.
- It is also known that due to the local support of FEM basis functions there are at most $c_{sp} \in \mathbb{N}$ many non-zeros per row / column.



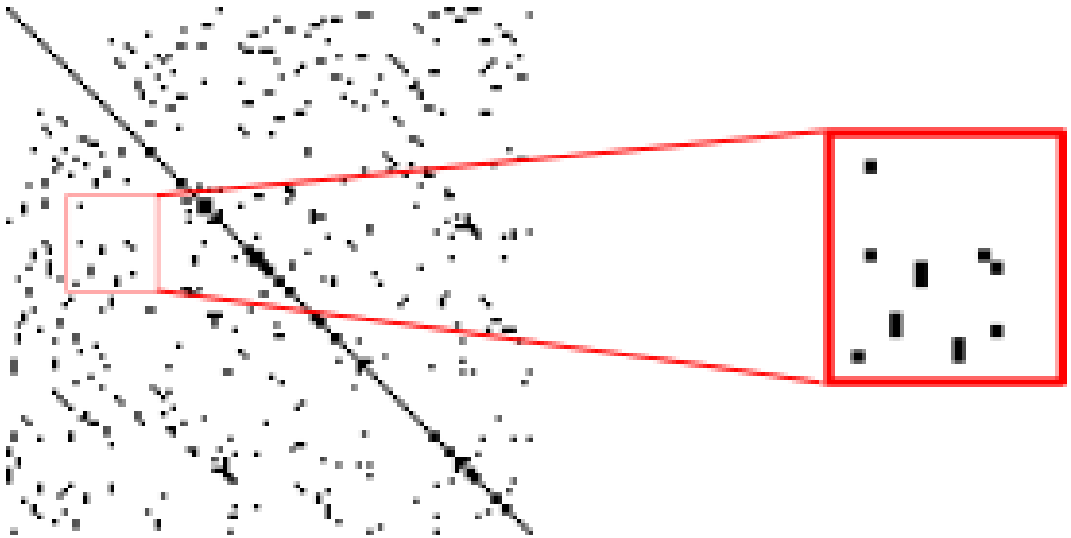
Inadmissible blocks:

- For the dimension of an inadmissible block $X \in \mathbb{K}^{n \times m}$ it holds $n \leq c_{lf} \vee m \leq c_{lf}$ for some small constant $c_{lf} \in \mathbb{N}$. In most applications $c_{lf} \in \{10, \ldots, 100\}$ is a good choice.
  If we consider a *strict* block partition we even have $\min(n, m) \leq c_{lf}$.
- It is also known that due to the local support of FEM basis functions there are at most $c_{sp} \in \mathbb{N}$ many non-zeros per row / column.
  → even less non-zeros per row / column of an inadmissible subblock X.



Inadmissible blocks:

- For the dimension of an inadmissible block $X \in \mathbb{K}^{n \times m}$ it holds $n \leq c_{lf} \vee m \leq c_{lf}$ for some small constant $c_{lf} \in \mathbb{N}$. In most applications $c_{lf} \in \{10, \ldots, 100\}$ is a good choice.
  If we consider a *strict* block partition we even have $\min(n, m) \leq c_{lf}$.
- It is also known that due to the local support of FEM basis functions there are at most $c_{sp} \in \mathbb{N}$ many non-zeros per row / column.
  → even less non-zeros per row / column of an inadmissible subblock X.
  → X is not dense but sparse!



Implementation

$\mathcal{H}$-matrix data structure in H2Lib

- The $\mathcal{H}$-matrix data structure contains either a pointer to
  - a dense matrix `f` or
  - a low rank matrix `r` or
  - an array of succeeding $\mathcal{H}$-matrices `son`.

```
struct hmatrix {
  amatrix       *f;
  rkmatrix      *r;

  hmatrix       *son;
};
```

$\mathcal{H}$-matrix data structure in H2Lib

- The $\mathcal{H}$-matrix data structure contains either a pointer to
  - a dense matrix `f` or
  - a low rank matrix `r` or
  - an array of succeeding $\mathcal{H}$-matrices `son`.
- Now introduce additional pointer to a sparse matrix `s`.

```
struct hmatrix {
  amatrix      *f;
  rkmatrix     *r;
  sparsematrix *s;
  hmatrix      *son;
};
```

Memory savings

How much memory can we save by this approach?

Memory savings

How much memory can we save by this approach?

Poisson equation

	# DoF	16.129	65.025	261.121	1.046.529	4.190.209
	A	0,6 MB	2,5 MB	10 MB	40 MB	160 MB
2D	$\mathcal{H}$-A	34 MB	168 MB	738 MB	3088 MB	12632 MB

	# DoF	3375	29.791	250.047	2.048.383
	A	0,2 MB	1,4 MB	12 MB	101 MB
3D	$\mathcal{H}$-A	11 MB	165 MB	1779 MB	16376 MB

Memory savings

How much memory can we save by this approach?

Poisson equation

	# DoF	16.129	65.025	261.121	1.046.529	4.190.209
2D	A	0,6 MB	2,5 MB	10 MB	40 MB	160 MB
	$\mathcal{H}$-A	34 MB	168 MB	738 MB	3088 MB	12632 MB
	$\mathcal{H}$-A strict	9,5 MB	40 MB	164 MB	666 MB	2685 MB

	# DoF	3375	29.791	250.047	2.048.383
3D	A	0,2 MB	1,4 MB	12 MB	101 MB
	$\mathcal{H}$-A	11 MB	165 MB	1779 MB	16376 MB
	$\mathcal{H}$-A strict	8 MB	109 MB	1113 MB	9984 MB

Memory savings

How much memory can we save by this approach?

Poisson equation

	# DoF	16.129	65.025	261.121	1.046.529	4.190.209
	A	0,6 MB	2,5 MB	10 MB	40 MB	160 MB
2D	$\mathcal{H}$-A	34 MB	168 MB	738 MB	3088 MB	12632 MB
	$\mathcal{H}$-A strict	9,5 MB	40 MB	164 MB	666 MB	2685 MB
	$\mathcal{H}_{sp}$-A	1,1 MB	4,6 MB	19 MB	77 MB	308 MB

	# DoF	3375	29.791	250.047	2.048.383
	A	0,2 MB	1,4 MB	12 MB	101 MB
3D	$\mathcal{H}$-A	11 MB	165 MB	1779 MB	16376 MB
	$\mathcal{H}$-A strict	8 MB	109 MB	1113 MB	9984 MB
	$\mathcal{H}_{sp}$-A	0,4 MB	5,8 MB	63 MB	581 MB

Memory savings

How much memory can we save by this approach?

Darcy's equation

	# DoF	12.288	49.152	196.608	786.432	3.145.728
	A	0,3 MB	1,1 MB	4,5 MB	18 MB	72 MB
2D	$\mathcal{H}$-A	20 MB	119 MB	554 MB	2388 MB	10127 MB

	# DoF	6.016	48.640	391.168	3.137.536
	A	0,3 MB	2,3 MB	19 MB	155 MB
3D	$\mathcal{H}$-A	14 MB	289 MB	3275 MB	30628 MB

Memory savings

How much memory can we save by this approach?

Darcy's equation

	# DoF	12.288	49.152	196.608	786.432	3.145.728
2D	A	0,3 MB	1,1 MB	4,5 MB	18 MB	72 MB
	$\mathcal{H}$-A	20 MB	119 MB	554 MB	2388 MB	10127 MB
	$\mathcal{H}$-A strict	6,6 MB	28 MB	116 MB	473 MB	1905 MB

	# DoF	6.016	48.640	391.168	3.137.536
3D	A	0,3 MB	2,3 MB	19 MB	155 MB
	$\mathcal{H}$-A	14 MB	289 MB	3275 MB	30628 MB
	$\mathcal{H}$-A strict	9,6 MB	110 MB	1051 MB	9145 MB

Memory savings

How much memory can we save by this approach?

Darcy's equation

	# DoF	12.288	49.152	196.608	786.432	3.145.728
2D	A	0,3 MB	1,1 MB	4,5 MB	18 MB	72 MB
	$\mathcal{H}$-A	20 MB	119 MB	554 MB	2388 MB	10127 MB
	$\mathcal{H}$-A strict	6,6 MB	28 MB	116 MB	473 MB	1905 MB
	$\mathcal{H}_{sp}$-A	0,5 MB	2,2 MB	8,8 MB	35 MB	142 MB

	# DoF	6.016	48.640	391.168	3.137.536
3D	A	0,3 MB	2,3 MB	19 MB	155 MB
	$\mathcal{H}$-A	14 MB	289 MB	3275 MB	30628 MB
	$\mathcal{H}$-A strict	9,6 MB	110 MB	1051 MB	9145 MB
	$\mathcal{H}_{sp}$-A	1,3 MB	4 MB	40 MB	358 MB

Arithmetics adaption:

How to extend the $\mathcal{H}$-matrix arithmetics to these new cases?

Arithmetics adaption:

How to extend the $\mathcal{H}$-matrix arithmetics to these new cases?

Answer: Treat sparse matrix blocks as if they were dense matrix blocks:

Arithmetics adaption:

How to extend the $\mathcal{H}$-matrix arithmetics to these new cases?

Answer: Treat sparse matrix blocks as if they were dense matrix blocks:

- Convert a sparse matrix S into a dense matrix F.

Arithmetics adaption:

How to extend the $\mathcal{H}$-matrix arithmetics to these new cases?

Answer: Treat sparse matrix blocks as if they were dense matrix blocks:

- Convert a sparse matrix S into a dense matrix F.
- Perform operation on the dense matrix F.

Arithmetics adaption:

How to extend the $\mathcal{H}$-matrix arithmetics to these new cases?

Answer: Treat sparse matrix blocks as if they were dense matrix blocks:

- Convert a sparse matrix S into a dense matrix F.
- Perform operation on the dense matrix F.
- If the values of F have changed, convert it back to a sparse matrix $\hat{S}$.

Arithmetics adaption:

How to extend the $\mathcal{H}$-matrix arithmetics to these new cases?

Answer: Treat sparse matrix blocks as if they were dense matrix blocks:

- Convert a sparse matrix S into a dense matrix F.
- Perform operation on the dense matrix F.
- If the values of F have changed, convert it back to a sparse matrix $\hat{S}$.
- Delete the dense matrix F.

Arithmetics adaption:

How to extend the $\mathcal{H}$-matrix arithmetics to these new cases?

Answer: Treat sparse matrix blocks as if they were dense matrix blocks:

- Convert a sparse matrix S into a dense matrix F.
- Perform operation on the dense matrix F.
- If the values of F have changed, convert it back to a sparse matrix $\hat{S}$.
- Delete the dense matrix F.

Possible approach, but inefficient! ($\sim$ 3 times slower!)

Adapting arithmetics

- Most $\mathcal{H}$-matrix arithmetic function follow a similar pattern:

```
void op(hmatrix *G, ...) {
  if(G->son != NULL)
    for(i = 0; i < G->sons; i++)
      op(G->son[i], ...);
  else if (G->r != NULL)
    op_rk(G->r, ...);
  else if (G->f != NULL)
    op_dense(G->f, ...);



}
```

Adapting arithmetics

- Most $\mathcal{H}$-matrix arithmetic function follow a similar pattern:
- Append another case for an $\mathcal{H}$-matrix being a sparse matrix.

```
void op(hmatrix *G, ...) {
  if(G->son != NULL)
    for(i = 0; i < G->sons; i++)
      op(G->son[i], ...);
  else if (G->r != NULL)
    op_rk(G->r, ...);
  else if (G->f != NULL)
    op_dense(G->f, ...);
  else if (G->s != NULL)
    F = Convert_sparse_dense(G->s);
    op_dense(F, ...);
    G->s = Convert_dense_sparse(F);
    del_dense(F);
}
```

Adapting arithmetics

- Most $\mathcal{H}$-matrix arithmetic function follow a similar pattern:
- Append another case for an $\mathcal{H}$-matrix being a sparse matrix.
- Implement specialized function to deal with the new situation

```
void op(hmatrix *G, ...) {
  if(G->son != NULL)
    for(i = 0; i < G->sons; i++)
      op(G->son[i], ...);
  else if (G->r != NULL)
    op_rk(G->r, ...);
  else if (G->f != NULL)
    op_dense(G->f, ...);
  else if (G->s != NULL)
    op_sparse(G->s, ...);



}
```

Example: Cholesky factorization of a s.p.d matrix

Example: Cholesky factorization of a s.p.d matrix

Operations to be adapted:

- Matrix solves with lower triangular matrices L: $X \leftarrow L^{-1}X$.
- Matrix multiplications: $Z \leftarrow Z + \alpha XY^*$.
- Cholesky factorizations: $X \leftarrow L$ with $X \approx LL^*$.

Example: Cholesky factorization of a s.p.d matrix

Operations to be adapted:

- Matrix solves with lower triangular matrices L: $X \leftarrow L^{-1}X$.
- Matrix multiplications: $Z \leftarrow Z + \alpha XY^*$.
- Cholesky factorizations: $X \leftarrow L$ with $X \approx LL^*$.

Remarks:

- Easy to implement when sparse matrices are only being read.

Example: Cholesky factorization of a s.p.d matrix

Operations to be adapted:

- Matrix solves with lower triangular matrices L: $X \leftarrow L^{-1}X$.
- Matrix multiplications: $Z \leftarrow Z + \alpha XY^*$.
- Cholesky factorizations: $X \leftarrow L$ with $X \approx LL^*$.

Remarks:

- Easy to implement when sparse matrices are only being read.
- More challenging when sparse matrices are being modified in an operation.

Numerical experiments on Cholesky factorization:

Choosing $\epsilon = 1 \times 10^{-3}$ with $\|A - LL^*\|_2 / \|A\|_2 \leq \epsilon$

Numerical experiments on Cholesky factorization:

Choosing $\epsilon = 1 \times 10^{-3}$ with $\|A - LL^*\|_2 / \|A\|_2 \leq \epsilon$

Poisson equation

2D

# DoF	16.129	65.025	261.121	1.046.529	4.190.209
A	0,6 MB	2,5 MB	10 MB	40 MB	160 MB
$\mathcal{H}$-L strict	9,5 MB	40 MB	116 MB	666 MB	2685 MB
	0,07 s	0,3 s	1,2 s	4,7 s	19 s
$\mathcal{H}_{sp}$-L	1,3 MB	5,1 MB	21 MB	84 MB	339 MB
	0,02 s	0,1 s	0,5 s	1,9 s	8 s

3D

# DoF	3375	29.791	250.047	2.048.383
A	0,2 MB	1,4 MB	12 MB	101 MB
$\mathcal{H}$-L strict	8 MB	109 MB	1113 MB	9984 MB
	0,13 s	5,3 s	71 s	699 s
$\mathcal{H}_{sp}$-L	0,8 MB	9 MB	88 MB	786 MB
	0,01 s	0,4 s	4,5 s	46 s

Numerical experiments on Cholesky factorization:

Choosing $\epsilon = 1 \times 10^{-3}$ with $\|A - LL^*\|_2 / \|A\|_2 \leq \epsilon$

Darcy's equation

2D

# DoF	12.288	49.152	196.608	786.432	3.145.728
A	0,3 MB	1,1 MB	4,5 MB	18 MB	72 MB
$\mathcal{H}$-L strict	6,6 MB 0,02 s	28 MB 0,08 s	116 MB 0,4 s	473 MB 1,5 s	1905 MB 6,6 s
$\mathcal{H}_{sp}$-L	0,5 MB < 0,01 s	2,2 MB 0,04 s	8,8 MB 0,15 s	36 MB 0,6 s	143 MB 2,5 s

3D

# DoF	6.016	48.640	391.168	3.137.536
A	0,3 MB	2,3 MB	19 MB	155 MB
$\mathcal{H}$-L strict	9,6 MB 0,15 s (0,03 s)	110 MB 4,7 s (0,9 s)	1051 MB 59 s (16 s)	9145 MB 610 s (184 s)
$\mathcal{H}_{sp}$-L	0,7 MB 0,01 s	7,4 MB 0,3 s	69 MB 4 s	592 MB 38 s

Remarks on performance:

Remarks on performance:

- Dense matrix algebra is done via Intel MKL, while sparse matrix algebra is currently done with our own implementation. (possibly not optimal)

Remarks on performance:

- Dense matrix algebra is done via Intel MKL, while sparse matrix algebra is currently done with our own implementation. (possibly not optimal)
- Before performing any matrix operation with low rank or sparse matrices, check if they are zero. (single if-statement)
  Example: 3D Poisson equation, 2m DoFs:

Remarks on performance:

- Dense matrix algebra is done via Intel MKL, while sparse matrix algebra is currently done with our own implementation. (possibly not optimal)
- Before performing any matrix operation with low rank or sparse matrices, check if they are zero. (single if-statement)
  Example: 3D Poisson equation, 2m DoFs:
  - with zero testing: 46 s

Remarks on performance:

- Dense matrix algebra is done via Intel MKL, while sparse matrix algebra is currently done with our own implementation. (possibly not optimal)
- Before performing any matrix operation with low rank or sparse matrices, check if they are zero. (single if-statement)
  Example: 3D Poisson equation, 2m DoFs:
  - with zero testing: 46 s
  - witout zero testing: 399 s

Remarks on performance:

- Dense matrix algebra is done via Intel MKL, while sparse matrix algebra is currently done with our own implementation. (possibly not optimal)
- Before performing any matrix operation with low rank or sparse matrices, check if they are zero. (single if-statement)
  Example: 3D Poisson equation, 2m DoFs:
  - with zero testing: 46 s
  - witout zero testing: 399 s
- Minimal leafsize C_{lf} has a huge impact on storage requirements as well as on runtime.

Compression

Additional compression strategies:

Additional compression strategies:

- While computing matrix factorizations, naturally *fill-ins* occur.
  $\rightarrow$ at some point storing as a dense matrix becomes attractive again.

Additional compression strategies:

- While computing matrix factorizations, naturally *fill-ins* occur.
  $\rightarrow$ at some point storing as a dense matrix becomes attractive again.
- In some cases a dense matrix might be stored more efficiently as low rank matrix.

Additional compression strategies:

- While computing matrix factorizations, naturally *fill-ins* occur.
  $\rightarrow$ at some point storing as a dense matrix becomes attractive again.
- In some cases a dense matrix might be stored more efficiently as low rank matrix.
- The other way around is usually not very attractive.

Additional compression strategies:

- While computing matrix factorizations, naturally *fill-ins* occur.
  $\rightarrow$ at some point storing as a dense matrix becomes attractive again.
- In some cases a dense matrix might be stored more efficiently as low rank matrix.
- The other way around is usually not very attractive.
- Similar to incomplete LU-factorization it might be useful to kick out some entries within the sparse matrices. (Problem: error control?)

Outlook and conclusion

Outlook

Outlook

- The same idea could be applied to $\mathcal{H}^2$-matrices as well.

Outlook

- The same idea could be applied to $\mathcal{H}^2$-matrices as well.
  - $\rightarrow$ Hopefully the same potential memory savings and performance gains.

Outlook

- The same idea could be applied to $\mathcal{H}^2$-matrices as well.
  - $\rightarrow$ Hopefully the same potential memory savings and performance gains.
- Sparse matrices might also help improving coupling matrices or cluster bases for $\mathcal{H}^2$-matrices.

Outlook

- The same idea could be applied to $\mathcal{H}^2$-matrices as well.
  - $\rightarrow$ Hopefully the same potential memory savings and performance gains.
- Sparse matrices might also help improving coupling matrices or cluster bases for $\mathcal{H}^2$-matrices.
- Different sparse matrix formats (we used CSR) and optimized sparse matrix algebra libraries might further speed up the computations.

Outlook

- The same idea could be applied to $\mathcal{H}^2$-matrices as well.
  - $\rightarrow$ Hopefully the same potential memory savings and performance gains.
- Sparse matrices might also help improving coupling matrices or cluster bases for $\mathcal{H}^2$-matrices.
- Different sparse matrix formats (we used CSR) and optimized sparse matrix algebra libraries might further speed up the computations.
- Problem-specific clustering strategies and admissibility conditions might reduce fill-ins and help decrease memory footprint.

Conclusions

Conclusions

- Using sparse matrices instead of dense matrices for inadmissible blocks reduces memory consumption of $\mathcal{H}$-matrices by a factor of 50 - 200.

Conclusions

- Using sparse matrices instead of dense matrices for inadmissible blocks reduces memory consumption of $\mathcal{H}$-matrices by a factor of 50 - 200.
- Performance gains lie between 2 and 20.

Conclusions

- Using sparse matrices instead of dense matrices for inadmissible blocks reduces memory consumption of $\mathcal{H}$-matrices by a factor of 50 - 200.
- Performance gains lie between 2 and 20.
- There are a few pitfalls when implementing new sparse-$\mathcal{H}$-arithmetics, but actually straight forward, when dense-$\mathcal{H}$-arithmetics is available.

Conclusions

- Using sparse matrices instead of dense matrices for inadmissible blocks reduces memory consumption of $\mathcal{H}$-matrices by a factor of 50 - 200.
- Performance gains lie between 2 and 20.
- There are a few pitfalls when implementing new sparse-$\mathcal{H}$-arithmetics, but actually straight forward, when dense-$\mathcal{H}$-arithmetics is available.
- Will be available in *H2Lib* (`www.h2lib.org`) package early in 2017.