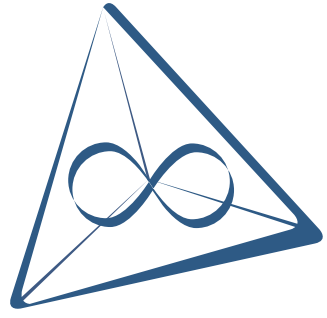
Numerische Behandlung von $\mathcal{H}$-Matrizen mit der HLIBpro

Ronald Kriemann
MPI MIS Leipzig

TU Bergakademie Freiberg

2011-11-09



Überblick

Entwicklung der HLIBpro

Entwicklung der HLIBpro ⟩ Geschichte und Ziele

1999 Start der Enwicklung einer C++-Bibliothek für $\mathcal{H}$-Matrizen mit Fokus auf parallelen Algorithmen.

2004 Fertigstellung der Bibliothek in zwei Versionen für gemeinsamen und verteilten Speicher.

2005 Fortsetzung der Softwareentwicklung hin zu einer industrietauglichen Bibliothek: HLIBpro

- Robustheit und Stabilität (fast) unabhängig von Eingabedaten,
- einfache Bedienung bei Vielzahl von möglichen Algorithmen,
- sehr gute Skalierung auf Standard-Rechnern/kleinen Clustern,
- Unterstützung von üblicher (Industrie-) Software, z.B. über verschiedenste Dateiformate,

2006 Kooperation mit Fraunhofer SCAI zur Vermarktung von HLIBpro.

Entwicklung der HLIBpro ⟩ Geschichte und Ziele

2006 HLIBpro v0.10 - v0.12

- komplett neue Fehlerbehandlung (Exceptions),
- einfache algebraische Clusterung,
- hierarchische Gebietszerlegung („nested dissection“),
- Start der komplexwertige Arithmetik,
- komplexwertiges ACA, reellwertiges HCA,
- interne Routinen für Laplace SLP/DLP,
- Einführung einer C-Schnittstelle,
- Unterstützung für MS-Windows

2007 HLIBpro v0.13

- Thread-parallele Arithmetik wiedereingeführt,
- Arithmetik mit einfacher Genauigkeit,
- vollständige, komplexwertige $\mathcal{H}$-Arithmetik,
- LU-/LDL-basierte $\mathcal{H}$-Invertierung,
- Unterstüzung für periodische Koordinaten bei Clusterung,
- interne Routinen für Helmholtz SLP/DLP und akustische Streuung,

2009 HLIBpro v0.13.6

- Verwendung von OpenMP zum Starten von Threads,
- Diagonalskalierung zur Stabilierung der $\mathcal{H}$-LU,
- Optimierung der algebraischen Clusterung

Entwicklung der HLIBpro ⟩ Geschichte und Ziele

2011 HLIBpro v1.0

- MPI-parallele Arithmetik für Matrix-Aufbau und $\mathcal{H}$-LU/LDL,
- $\mathcal{H}^2$-Matrizen,
- eigene Multilevel-Graphpartitionierung für algebraische Clusterung,
- Unterstützung für lineare Basisfunktionen und Maxwell-EFIE/MFIE,
- komplett neue Schnittstelle zu BLAS/LAPACK (ohne Zeiger),
- Routinen für Stabilisierung der $\mathcal{H}$-Faktorisierung,
- bessere Skalierung bei Thread-paralleler $\mathcal{H}$-Faktorisierung

Entwicklung der HLIBpro 〉HLib und HLIBpro

HLib und HLIBpro waren stets verschiedene Projekte mit unterschiedlichem Fokus:

HLib: Forschungscode von Lars Grasedyck und Steffen Börm zum Testen der jeweils aktuellen Algorithmen, d.h. enthält Vielzahl von Einzelverfahren in nicht zwingend konsistenter Form für spezielle Einsatzgebiete,

HLIBpro: Implementierung von Algorithmen im Kontext der $\mathcal{H}$-Matrizen in einheitlicher und robuster Form. Daher im Einzelnen nicht alle Verfahren aus der HLib verfügbar.

Aber: aufgrund der Entwicklung „unter einem Dach“ gab es natürlich regen Austausch.

Unterschiede zwischen HLib und HLIBpro

HLib:
- entwickelt in C,
+ Vielzahl von Approximationsverfahren, z.B. Interpolation, etc.,
+ komplette $\mathcal{H}^2$-Arithmetik,
– keine komplexwertige oder parallele Arithmetik,
– nur rudimentäre, algebraische Clusterung
– nur eingeschränkte I/O-Funktionen,

HLIBpro:
- entwickelt in C++,
+ reell- und komplexwertige $\mathcal{H}$-Arithmetik in einfacher und doppelter Genauigkeit,
+ Thread- und MPI-parallele Arithmetik,
+ hochentwickelte algebraische Clusterung,
+ Unterstützung einer Vielzahl von Dateiformaten: Matlab, MatrixMarket, Harwell-Boeing, Ply, Gmsh, etc.,
– nur Standardapproximation, d.h. ACA, HCA,
– keine $\mathcal{H}^2$-Arithmetik

Akademische Nutzung

- HLIBpro ist für akademische Zwecke in Binärform frei verfügbar.
- Jeder Lizenzschlüssel ist an einen Benutzer und einen Rechner gebunden.
- Pro Lizenznehmer können aber mehrere Lizenzschlüssel ausgegeben werden.
- Zugang zum Quelltext ist nur möglich im Rahmen eines offiziellen Projektes, z.B. DFG, BMBF.

Kommerzielle Nutzung

- HLIBpro ist nicht frei und eine Lizenz stets mit Kosten verbunden.
- Das betrifft auch „außeruniversitäre“ Forschungseinrichtungen.

Beispiel Integralgleichung

Beispiel Integralgleichung ⟩ Problemdefinition

Gegeben sei Integralgleichung

$$\int_\Gamma k(x,y)u(y)dy = f(x) \quad \forall x \in \Gamma$$

mit Kern $k(\cdot,\cdot)$, $\Gamma = \partial\Omega \in \mathbb{R}^3$, rechter Seite f und der gesuchten Funktion u.

Der übliche Galerkinansatz mit geeigneten Basisfunktionen $V := \{\varphi_1, \ldots, \varphi_n\}$ führt dann zu dem Gleichungssystem

$$A\mathbf{u} = \mathbf{f}$$

mit

$$a_{ij} = \int_\Gamma \int_\Gamma \varphi_i(x)k(x,y)\varphi_j(y)dxdy$$
$$f_i = \int_\Gamma f(x)\varphi_i(x)dx$$

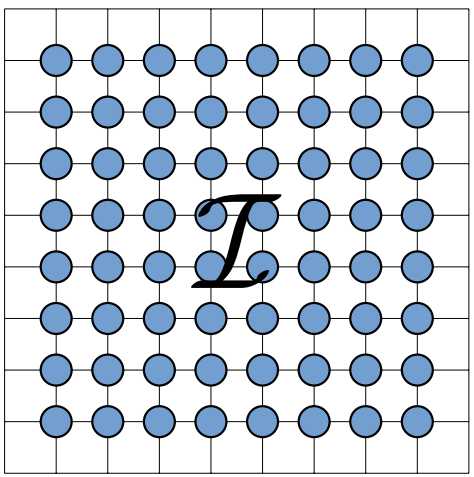
Beispiel Integralgleichung ⟩ $\mathcal{H}$-Matrizen

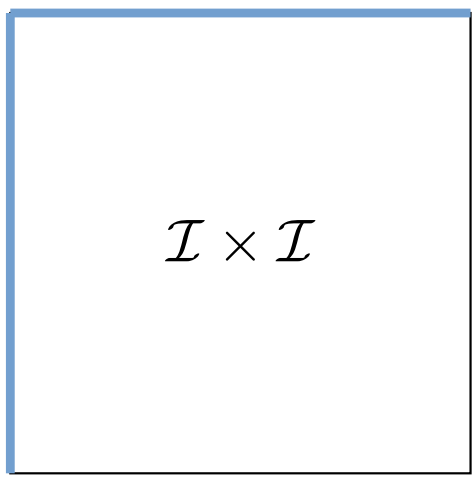
Sei $\mathcal{I}$ eine Indexmenge mit $\#\mathcal{I} = n$, d.h. den Basisfunktionen φ_i zugeordnet. Die Aufgaben bei der Anwendung von $\mathcal{H}$-Matrizen zur Lösung von $A\mathbf{u} = \mathbf{f}$ sind:

Beispiel Integralgleichung ⟩ $\mathcal{H}$-Matrizen

Sei $\mathcal{I}$ eine Indexmenge mit $\#\mathcal{I} = n$, d.h. den Basisfunktionen φ_i zugeordnet. Die Aufgaben bei der Anwendung von $\mathcal{H}$-Matrizen zur Lösung von $A\mathbf{u} = \mathbf{f}$ sind:

Clusterung Identifikation geeigneter Teilbereiche $t \times s \in \mathcal{I} \times \mathcal{I}$ in welchen sich $A|_{t\times s}$ datenschwach approximieren läßt.

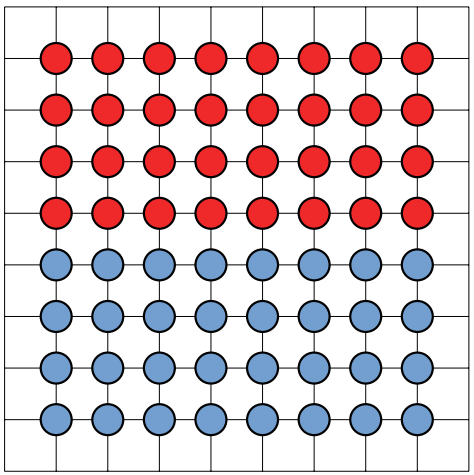


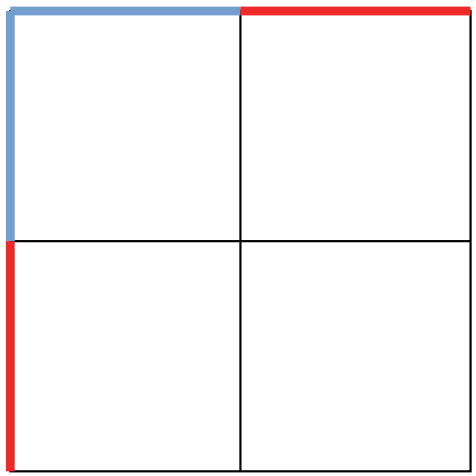


Beispiel Integralgleichung ⟩ $\mathcal{H}$-Matrizen

Sei $\mathcal{I}$ eine Indexmenge mit $\#\mathcal{I} = n$, d.h. den Basisfunktionen φ_i zugeordnet. Die Aufgaben bei der Anwendung von $\mathcal{H}$-Matrizen zur Lösung von $A\mathbf{u} = \mathbf{f}$ sind:

Clusterung Identifikation geeigneter Teilbereiche $t \times s \in \mathcal{I} \times \mathcal{I}$ in welchen sich $A|_{t\times s}$ datenschwach approximieren läßt.

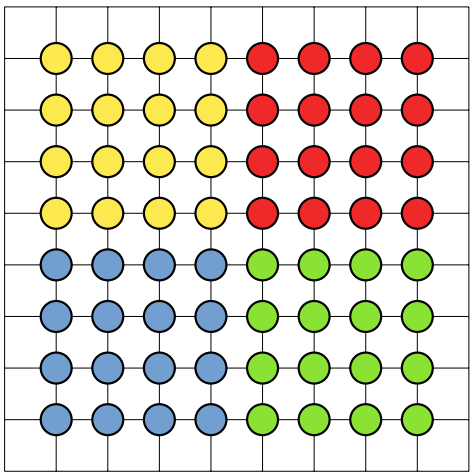


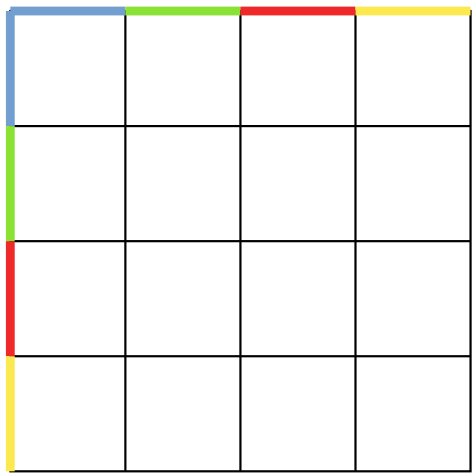


Beispiel Integralgleichung ⟩ $\mathcal{H}$-Matrizen

Sei $\mathcal{I}$ eine Indexmenge mit $\#\mathcal{I} = n$, d.h. den Basisfunktionen φ_i zugeordnet. Die Aufgaben bei der Anwendung von $\mathcal{H}$-Matrizen zur Lösung von $A\mathbf{u} = \mathbf{f}$ sind:

Clusterung Identifikation geeigneter Teilbereiche $t \times s \in \mathcal{I} \times \mathcal{I}$ in welchen sich $A|_{t\times s}$ datenschwach approximieren läßt.

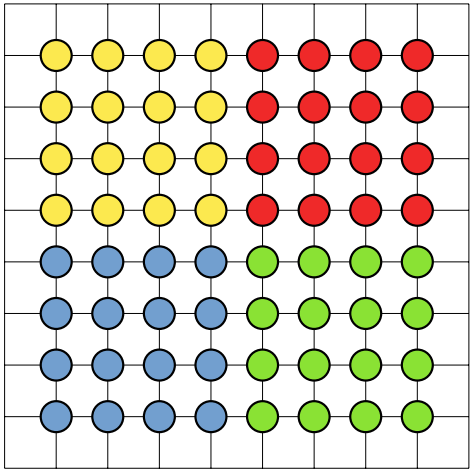


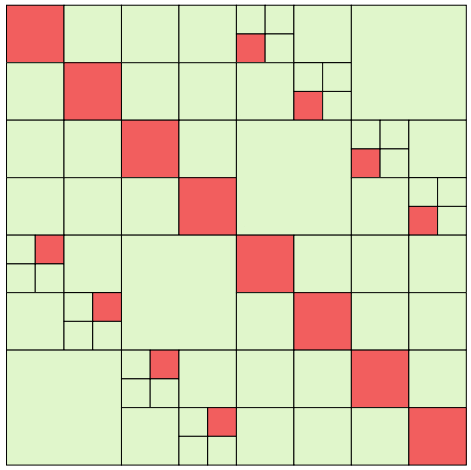


Beispiel Integralgleichung ⟩ $\mathcal{H}$-Matrizen

Sei $\mathcal{I}$ eine Indexmenge mit $\#\mathcal{I} = n$, d.h. den Basisfunktionen φ_i zugeordnet. Die Aufgaben bei der Anwendung von $\mathcal{H}$-Matrizen zur Lösung von $A\mathbf{u} = \mathbf{f}$ sind:

Clusterung Identifikation geeigneter Teilbereiche $t \times s \in \mathcal{I} \times \mathcal{I}$ in welchen sich $A|_{t\times s}$ datenschwach approximieren läßt.



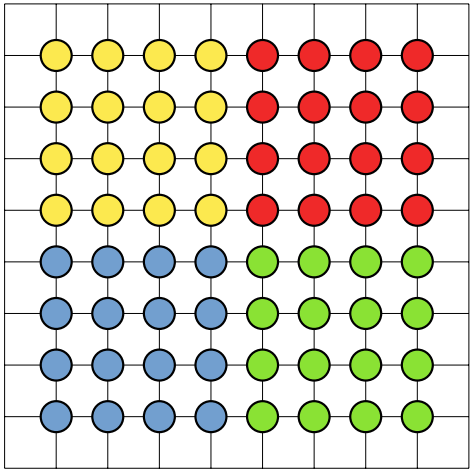


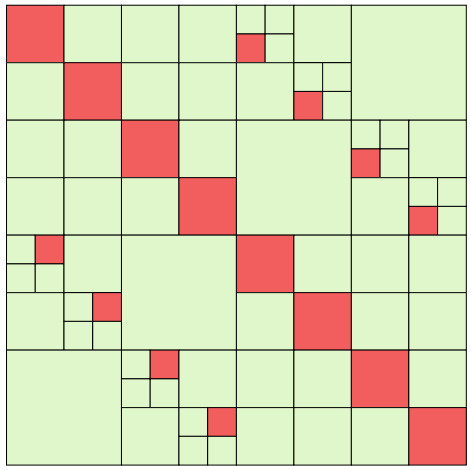
Beispiel Integralgleichung ⟩ $\mathcal{H}$-Matrizen

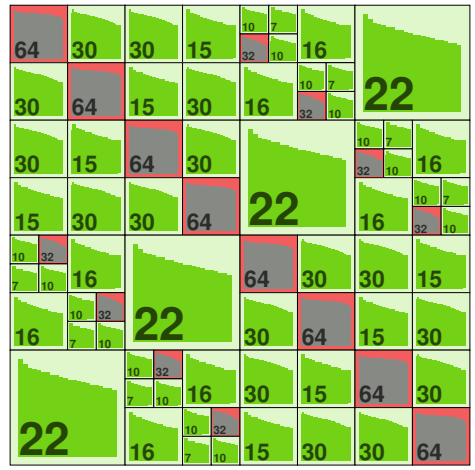
Sei $\mathcal{I}$ eine Indexmenge mit $\#\mathcal{I} = n$, d.h. den Basisfunktionen φ_i zugeordnet. Die Aufgaben bei der Anwendung von $\mathcal{H}$-Matrizen zur Lösung von $A\mathbf{u} = \mathbf{f}$ sind:

Clusterung Identifikation geeigneter Teilbereiche $t \times s \in \mathcal{I} \times \mathcal{I}$ in welchen sich $A|_{t\times s}$ datenschwach approximieren läßt.

Aufbau Berechnung der blockweisen Approximation von A (als Niedrigrangmatrizen),







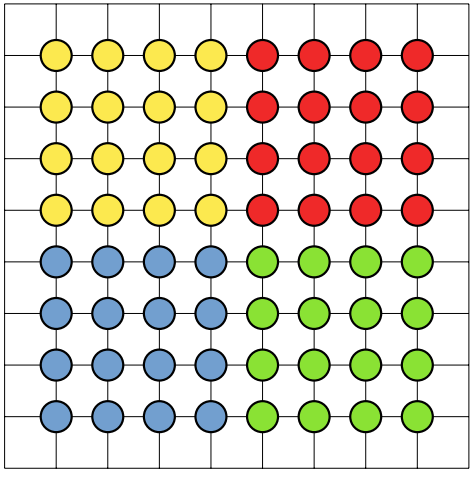
Beispiel Integralgleichung ⟩ $\mathcal{H}$-Matrizen

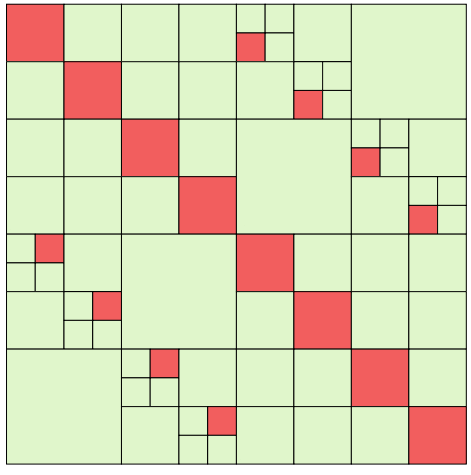
Sei $\mathcal{I}$ eine Indexmenge mit $\#\mathcal{I} = n$, d.h. den Basisfunktionen φ_i zugeordnet. Die Aufgaben bei der Anwendung von $\mathcal{H}$-Matrizen zur Lösung von $A\mathbf{u} = \mathbf{f}$ sind:

Clusterung Identifikation geeigneter Teilbereiche $t \times s \in \mathcal{I} \times \mathcal{I}$ in welchen sich $A|_{t\times s}$ datenschwach approximieren läßt.

Aufbau Berechnung der blockweisen Approximation von A (als Niedrigrangmatrizen),

Lösung Definition einer Arithmetik für $\mathcal{H}$-Matrizen für die Lösung des Gleichungssystems, d.h. Matrix-Vektor-Mult. und u.U. $\mathcal{H}$-LU-Zerlegung.

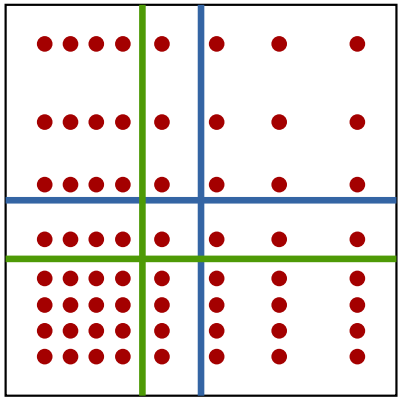




Clusterbaum

HLIBpro bietet verschiedene (BSP-) Clusterungsalgorithmen:

- *volumen*- oder *kardinalitäts*- balanciert,
- optional mit *regulärer* (zyklischer) Achsenwahl,
- *Hauptkomponentenanalyse* (PCA)



Standard ist automatische Clusterung mit Kombination aus Volumen- und Kardinalitätsbalancierung:

```
TCoordinate         coord( vertices, 3, bbmin, bbmax );
TAutoBSPPartStrat   part_strat( & coord );
TBSPCTBuilder       ct_builder( & part_strat );
TClusterTree *      ct = ct_builder.build( & coord );
```

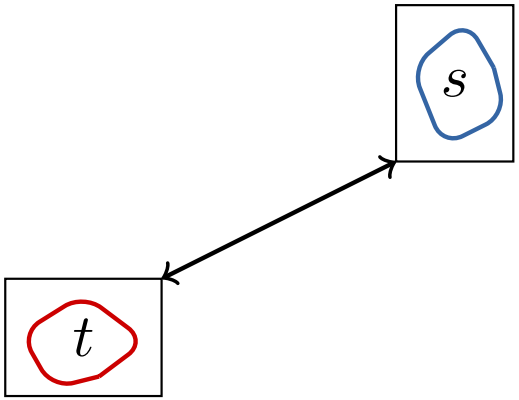
Sei $T(\mathcal{I})$ der so berechnete Clusterbaum über $\mathcal{I}$.

Blockclusterbaum

Entscheidend für den Blockclusterbaum ist die Zulässigkeitsbedingung, üblicherweise

$$\min\{\operatorname{diam}(t), \operatorname{diam}(s)\} \leq \eta \operatorname{dist}(t, s)$$

für Cluster $t, s \in T(\mathcal{I})$.



Die Berechnung von diam bzw. dist erfolgt in HLIBpro standardmäßig über die „bounding box“.

```
TStdAdmCond           adm( 2.0, false ); // true → max{diam(t), diam(s)}
TBCBuilder            bc_builder;
TBlockClusterTree *   bct = bct_builder.build( ct, ct, & adm );
```

$T(\mathcal{I} \times \mathcal{I})$ bezeichne den entsprechend konstruierten Blockclusterbaum über $\mathcal{I} \times \mathcal{I}$.

Beispiel Integralgleichung ⟩ Matrix-Aufbau

Für den $\mathcal{H}$-Matrixaufbau, hier vereinfacht nur für die Blätter $\mathcal{L}$ von $T(\mathcal{I} \times \mathcal{I})$, genügt folgender Algorithmus:

for $(t, s) \in \mathcal{L}(T(\mathcal{I} \times \mathcal{I}))$ **do**
  if (t, s) ist zulässig **then**
    erstelle Niedrigrangmatrix;
  else
    erstelle volle Matrix;

Entsprechend notwendig ist ein Verfahren zur Approximation von $A|_{(t,s)}$ für zulässige Blöcke $(t, s) \in \mathcal{T}(I \times I)$.

HLIBpro bietet hierfür

- ACA Adaptive Kreuzapproximation (ACA, ACA+, ACA-Full),
- HCA Hybride Kreuzapproximation,
- SVD Singulärwertzerlegung.

Beispiel Integralgleichung ⟩ Matrix-Aufbau

Für diese Verfahren müssen die Matrixkoeffizienten

$$a_{ij} = \int_\Gamma \int_\Gamma \varphi_i(x) k(x,y) \varphi_j(y) dx dy$$

in Form einer *Koeffizientenfunktion* bereitgestellt werden:

```
TMyCoeffFn      my_fn( ..., rowct->perm_i2e(), colct->perm_i2e() );
```

mit zu implementierender `eval`-Funktion:

```
void TMyCoeffFn::eval ( const vector< idx_t > &  rowidxs,
                        const vector< idx_t > &  colidxs,
                        real *                   matrix ) const;
```

Anschließend wird das Approximationsverfahren und die Genauigkeit definiert:

```
TACAPlus        lrapx( & my_fn );
TDenseMBuilder  h_ctor( & my_fn, & lrapx );
TTruncAcc       acc( 1e-4, 0.0 );
TMatrix *       A = h_ctor.build( nthreads, bct, MATFORM_NONSYM, acc );
```

Alternativ können die Matrixblöcke auch direkt mit eigenen Verfahren konstruiert werden, z.B. für $\mathcal{H}^2$-Matrizen.

Paralleler Matrix-Aufbau

Der (vereinfachte) Algorithmus

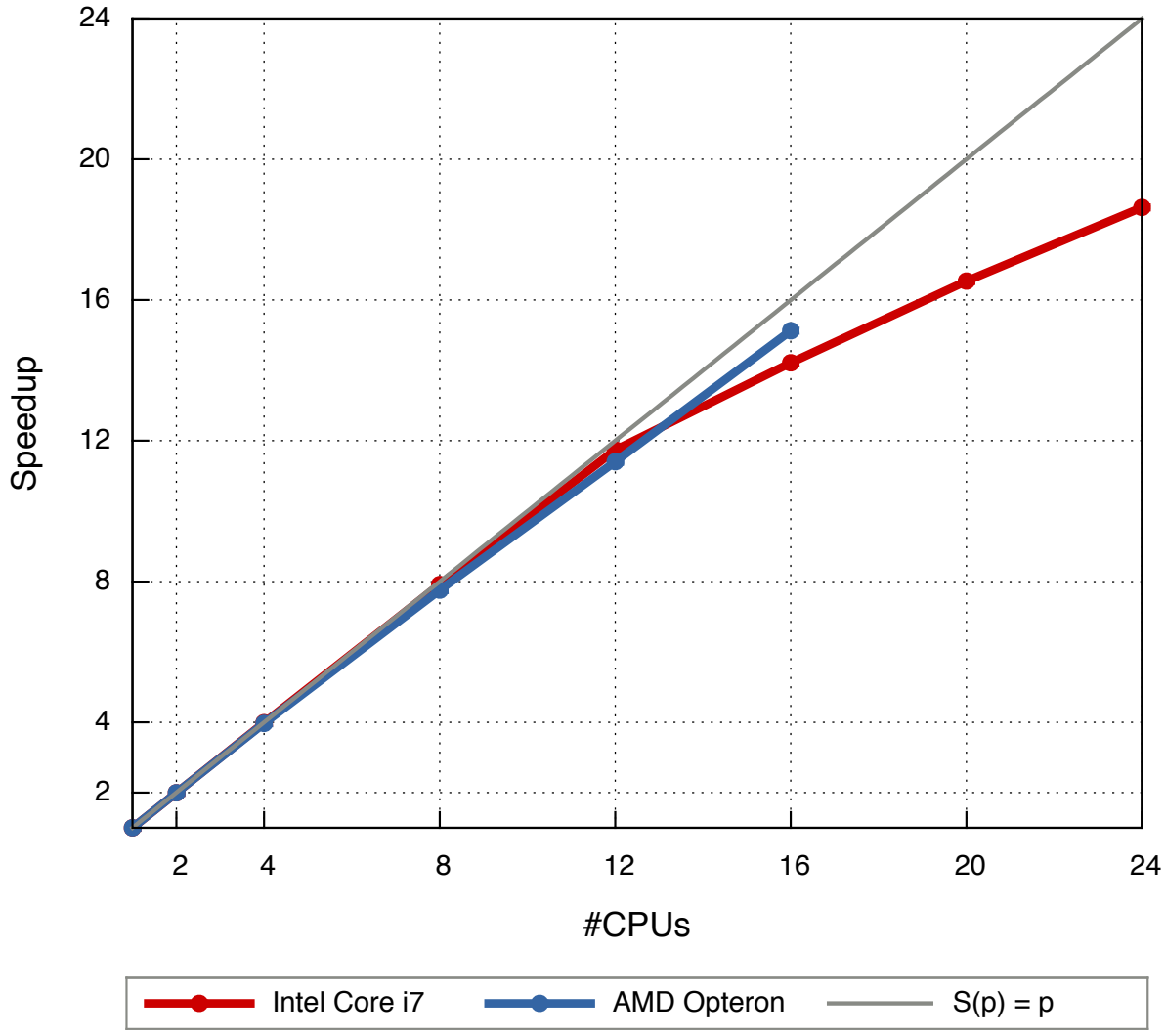
for $(t, s) \in \mathcal{L}(T(\mathcal{I} \times \mathcal{I}))$ **do**
konstruiere Matrixblock für (t, s)

für den $\mathcal{H}$-Matrixaufbau hat folgende, für die Parallelisierung wichtige Eigenschaften:

- alle Matrixblöcke können unabhängig voneinander konstruiert werden,
- $\#\mathcal{L}(T(\mathcal{I} \times \mathcal{I})) \gg \#$procs, d.h. genügend Arbeit für jeden Prozessor und damit keine explizite Lastverteilung notwendig („list/online scheduling“ genügt).

Beispiel Integralgleichung ⟩ Matrix-Aufbau

Helmholtz, Doppelschicht-Potential im $\mathbb{R}^3$:



Beispiel Integralgleichung ⟩ Lösen des GLS

Zur iterativen Lösung von Gleichungssystemen stehen in HLIBpro mehrere Verfahren zur Verfügung, z.B. CG, BiCG-Stab, MINRES und GMRES.

```
TGMRES   gmres( 20, 1000 ); // Neustart und maximale Iterationsanzahl
TVector  x, b;

b = ...; // rechte Seite
x = ...; // Lösungsvektor

gmres.solve( A, x, b );
```

Beschleunigung ist mit Präkonditionierer, z.B. per $\mathcal{H}$-LU, möglich:

```
TMatrix *  B     = A->copy();
TMatrix *  A_inv = factorise_inv( nthreads, B, acc );

gmres.solve( A, x, b, A_inv );
```

Wesentliches Element in beiden Fällen ist die $\mathcal{H}$-Matrix-Vektor-Multiplikation (bzw. $\mathcal{H}$-Dreiecksauflösen).

Parallele Matrix-Vektor-Multiplikation

Analog zum Matrix-Aufbau kann die Vektor-Mult. vereinfacht für die Blätter betrachtet werden:

$$\begin{aligned}&\textbf{for}\ \ (t,s) \in \mathcal{L}(T(I \times I))\ \ \textbf{do}\\ &\qquad y|_t := y|_t + A|_{(t,s)} \cdot x|_s;\end{aligned}$$

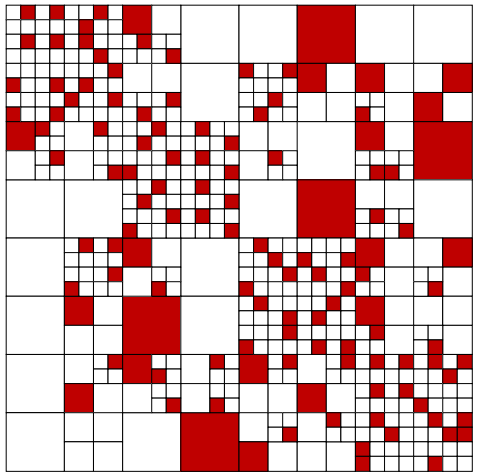
Im parallelen Fall wird hierfür noch ein lokaler Vektor y^p für Prozessor p benötigt:

$$\begin{aligned}&y^p \in \mathbb{R}^m;\\ &\textbf{for}\ \ (t,s) \in \mathcal{L}(T(I \times I))\ \ \textbf{do}\\ &\qquad y^p|_t := y^p|_t + A|_{(t,s)} \cdot x|_s;\\ &\textit{\{paralleles Aufsummieren\}}\\ &y := \textstyle\sum_i y^i;\end{aligned}$$

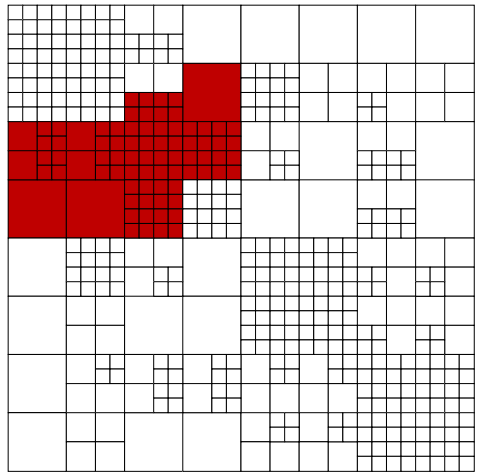
Bei angenommener gleicher Last (analog zum Matrix-Aufbau) hängt die parallele Komplexität nur von der Größe m von y^p ab.

Parallele Matrix-Vektor-Multiplikation

Die Größe m des lokalen Vektors wird bestimmt durch die Verteilung der Matrixblöcke:



$m \in \mathcal{O}(n)$



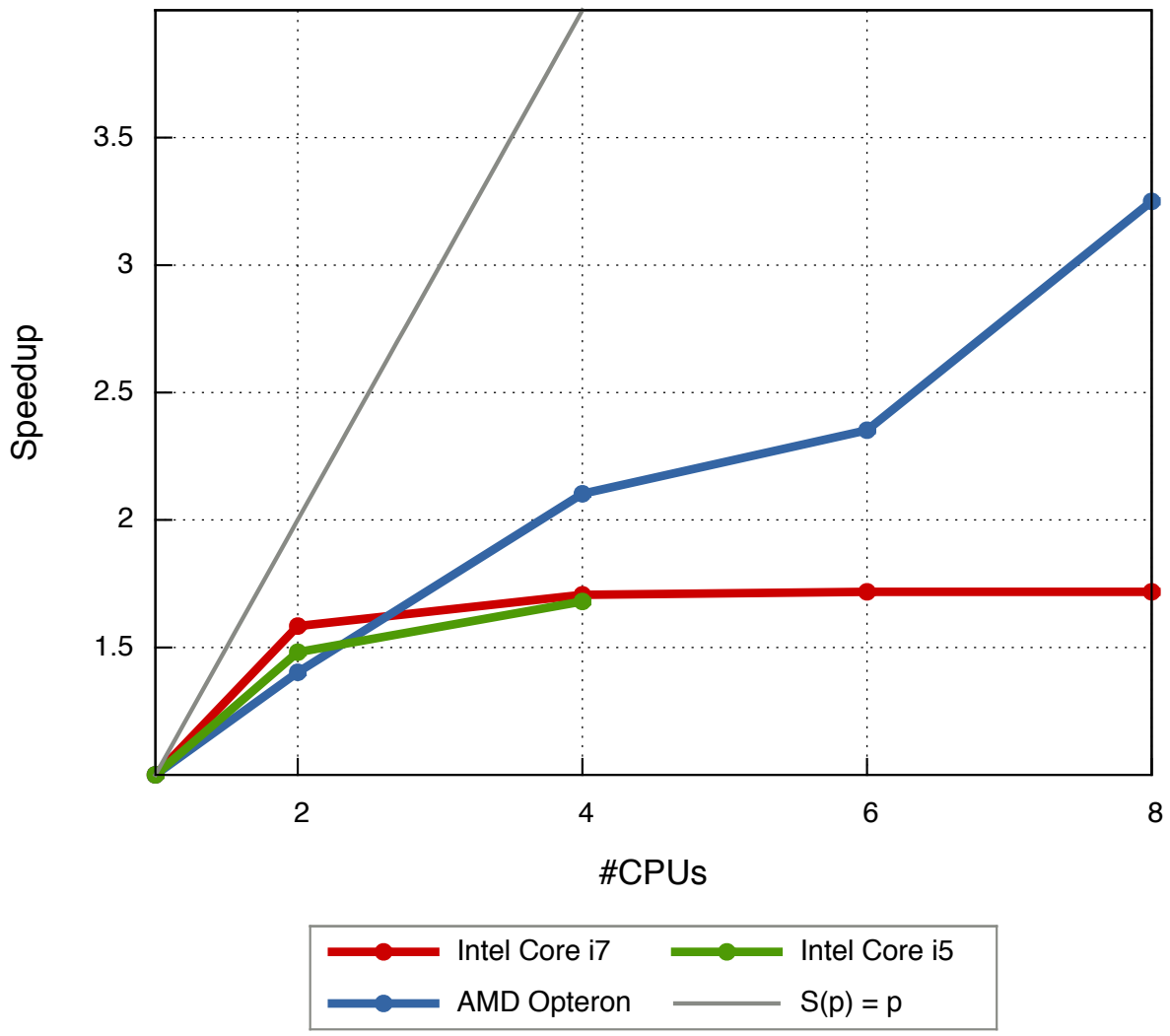
$m \in \mathcal{O}\left(\frac{n}{\sqrt{p}}\right)$

Bei *lokaler* Verteilung ist die Gesamtkomplexität damit

$$\mathcal{O}\left(\frac{n \log n}{p} + \frac{n}{\sqrt{p}}\right)$$

Beispiel Integralgleichung ⟩ Lösen des GLS

Matrix-Vektor-Mult., Helmholtz, DLP im $\mathbb{R}^3$:



Parallele $\mathcal{H}$-LU-Zerlegung

Bei einer angenommenen 2×2-Blockstruktur von A:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \begin{pmatrix} L_{11} & \\ L_{21} & L_{22} \end{pmatrix} \cdot \begin{pmatrix} U_{11} & U_{12} \\ & U_{22} \end{pmatrix}$$

erhält man

$$\begin{aligned} L_{11}U_{11} &= A_{11}, & L_{11}U_{12} &= A_{12}, \\ L_{21}U_{11} &= A_{21}, & L_{22}U_{22} &= A'_{22} \end{aligned}$$

mit

$$A'_{22} = A_{22} - L_{21}U_{12}.$$

Damit besteht die $\mathcal{H}$-LU-Zerlegung aus Rekursionen und Matrix-Multiplikationen (analog für Dreiecksauflösen).

Thread-Parallele $\mathcal{H}$-LU-Zerlegung

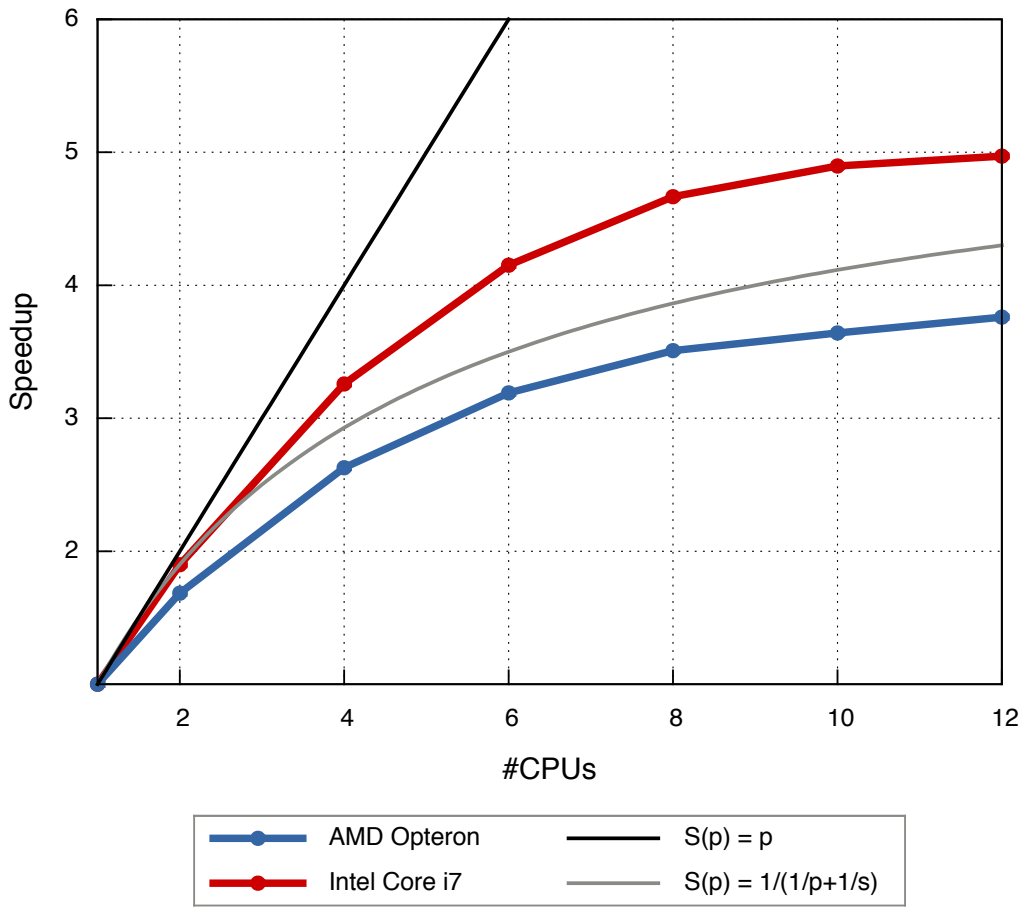
Bei Systemen mit *gemeinsamem* Speicher kann die $\mathcal{H}$-Matrix-Multiplikation mit optimaler Skalierung durchgeführt werden.

Thread-Parallele $\mathcal{H}$-LU-Zerlegung

Bei Systemen mit *gemeinsamem* Speicher kann die $\mathcal{H}$-Matrix-Multiplikation mit optimaler Skalierung durchgeführt werden.

Trotzdem ergibt sich durch die Rekursionen ein sequentieller Anteil und damit eine Komplexität von

$$\mathcal{O}\left(\frac{n \log^2 n}{p} + n \log n\right).$$

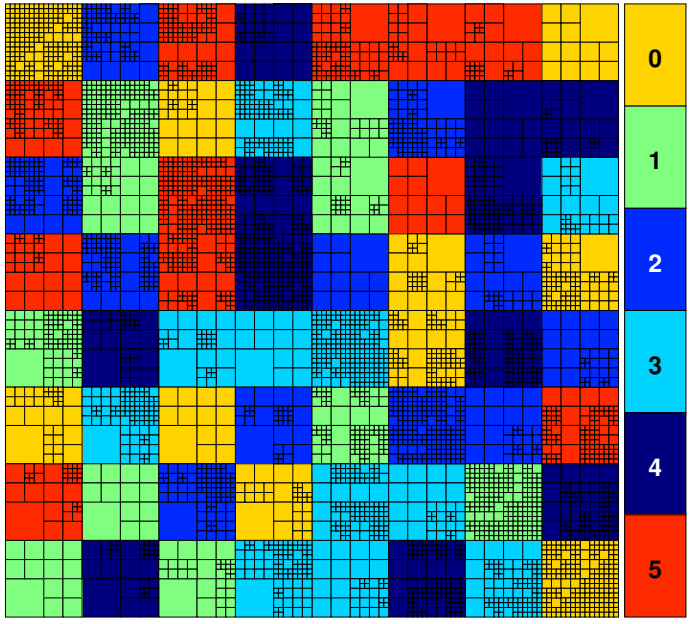


MPI-Parallele $\mathcal{H}$-LU-Zerlegung

Im Falle eines *verteilten* Speichers kann die $\mathcal{H}$-Matrix-Multiplikation nicht parallel ausgeführt werden. Es müssen die jeweils benötigten Teilblöcke der Matrix komplett über das Netzwerk gesendet werden.

Parallelität ergibt sich aber aus dem gleichzeitigen Auflösen innerhalb einer Blockspalte bzw. -zeile. Außerdem ist der Speicher optimal verteilt (*Hauptanwendung* momentan!).

Bei schnellem Netzwerk ergibt sich ein Speedup von etwa 2.



MPI-Parallele $\mathcal{H}$-LU-Zerlegung

Die Definition der Lastverteilung erfolgt in HLIBpro über den Blockclusterbaum, vor dem Matrix-Aufbau:

```
TBlockDistrBC  bc_distr( min_lvl, false );  // true → sym. Verteilung
TMyCostFn      cost_fn;

bc_distr.distribute( NET::nprocs(), bct, & cost_fn );
```

Hierfür ist eine entsprechende Kostenfunktion zu definieren.

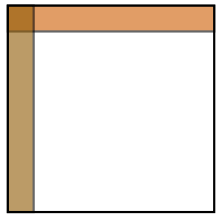
Für eine lokale Verteilung in Hinblick auf Matrix-Vektor-Multiplikation:

```
TSFCDistrBC    bc_distr;
```

$\mathcal{H}^2$-Matrizen

$\mathcal{H}^2$-Matrizen ⟩ Einführung

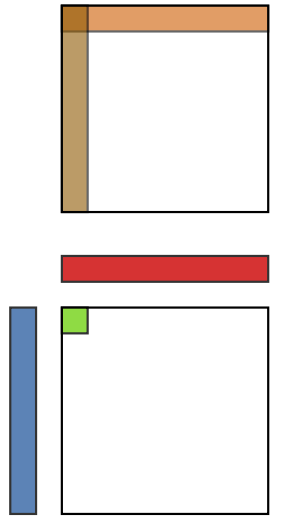
In einer $\mathcal{H}$-Matrix werden Rang-k-Matrizen $M \in \mathbb{K}^{t \times s}$ in faktorisierter Form $M = A \cdot B^H$ mit $A \in \mathbb{K}^{t \times k}, B \in \mathbb{K}^{s \times k}$ dargestellt, d.h. mit Aufwand $k \cdot (\#t + \#s)$.



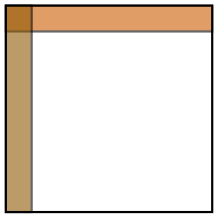
$\mathcal{H}^2$-Matrizen ⟩ Einführung

In einer $\mathcal{H}$-Matrix werden Rang-k-Matrizen $M \in \mathbb{K}^{t \times s}$ in faktorisierter Form $M = A \cdot B^H$ mit $A \in \mathbb{K}^{t \times k}, B \in \mathbb{K}^{s \times k}$ dargestellt, d.h. mit Aufwand $k \cdot (\#t + \#s)$.

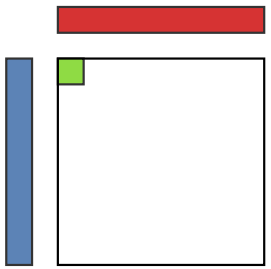
Sei V eine Basis für $\mathbb{K}^t$ und W für $\mathbb{K}^s$. Dann ergibt sich $M = VSW^T$ mit *Basiskoeffizienten* $S \in \mathbb{K}^{k \times k}$.



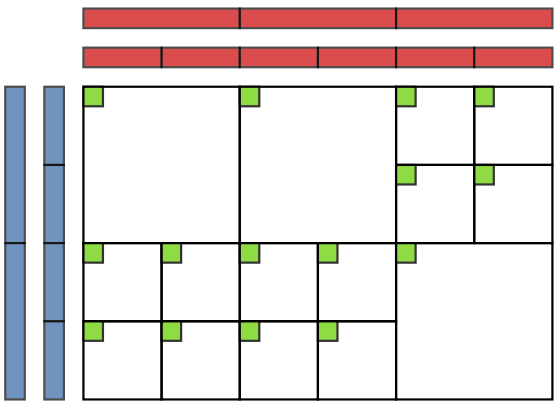
In einer $\mathcal{H}$-Matrix werden Rang-k-Matrizen $M \in \mathbb{K}^{t \times s}$ in faktorisierter Form $M = A \cdot B^H$ mit $A \in \mathbb{K}^{t \times k}, B \in \mathbb{K}^{s \times k}$ dargestellt, d.h. mit Aufwand $k \cdot (\#t + \#s)$.



Sei V eine Basis für $\mathbb{K}^t$ und W für $\mathbb{K}^s$. Dann ergibt sich $M = VSW^T$ mit *Basiskoeffizienten* $S \in \mathbb{K}^{k \times k}$.

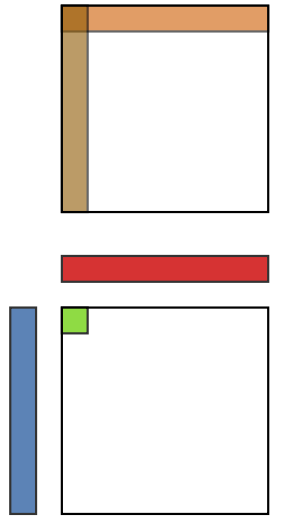


Wählt man geeignete Basen V^t, W^s für alle $t, s \in \mathcal{T}$, so lassen sich alle Matrizen der $\mathcal{H}$-Matrix allein durch die Koeffizientenmatrizen $S^{t,s}$ darstellen. Dadurch sinkt der Speicheraufwand für die $\mathcal{H}$-Matrix auf $\mathcal{O}(n)$.



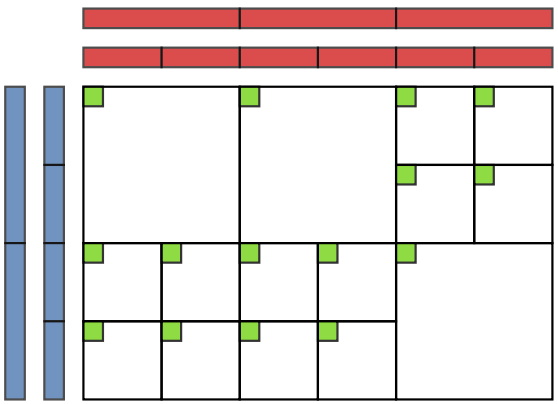
$\mathcal{H}^2$-Matrizen ⟩ Einführung

In einer $\mathcal{H}$-Matrix werden Rang-k-Matrizen $M \in \mathbb{K}^{t \times s}$ in faktorisierter Form $M = A \cdot B^H$ mit $A \in \mathbb{K}^{t \times k}, B \in \mathbb{K}^{s \times k}$ dargestellt, d.h. mit Aufwand $k \cdot (\#t + \#s)$.



Sei V eine Basis für $\mathbb{K}^t$ und W für $\mathbb{K}^s$. Dann ergibt sich $M = VSW^T$ mit *Basiskoeffizienten* $S \in \mathbb{K}^{k \times k}$.

Wählt man geeignete Basen V^t, W^s für alle $t, s \in \mathcal{T}$, so lassen sich alle Matrizen der $\mathcal{H}$-Matrix allein durch die Koeffizientenmatrizen $S^{t,s}$ darstellen. Dadurch sinkt der Speicheraufwand für die $\mathcal{H}$-Matrix auf $\mathcal{O}(n)$.



Die Basen $\{V^t : t \in \mathcal{T}\}$ und $\{W^t : t \in \mathcal{T}\}$ heißen Clusterbasen und besitzen die gleiche (hierarchische) Struktur wie der Clusterbaum. Clusterbasen sind üblicherweise geschachtelt.

$\mathcal{H}^2$-Matrizen sind in der HLIBpro erst seit kurzem implementiert, dafür aber in einer modernen Form vorhanden. Aktuell verfügbar sind folgende Datenstrukturen:

- Clusterbasen: sowohl reell- als auch komplexwertig,
- uniforme Vektoren über einzelnen Clustern und Clusterbäumen,
- uniforme Matrizen,

und Algorithmen:

- Konstruktoren für Clusterbasen aus vollbesetzen Matrizen und $\mathcal{H}$-Matrizen,
- $\mathcal{H}^2$-Matrix-Vektor-Multiplikation

Eine $\mathcal{H}^2$-Arithmetik steht nicht zur Verfügung.

Im Gegensatz zu den üblichen Datenstrukturen in der HLIBpro ist der Wertetyp bei den Clusterbasen generisch. Außerdem wurde die neue BLAS/LAPACK-Schnittstelle voll in den Entwurf integriert.

Manuelle Konstruktion

Bei vorhandenem Wissen über die entsprechenden Koeffizienten einer geeigneten Clusterbasis für eine $\mathcal{H}^2$-Matrix, kann diese wie folgt manuell konstruiert werden:

- Clusterbasis für ein Blatt:

```
BLAS::Matrix< T >              V = ... // Basis für Blatt
TIndexSet                      is( first, last );
TClusterBasis< T >             cb( is, V );
```

- Clusterbasis für inneren Knoten:

```
TClusterBasis< T >             cb0( is0, V1 ), cb1( is1, V1 );
vector< TClusterBasis< T > >   sons = { cb0, cb1 };
BLAS::Matrix< T >              E0 =..., E1 =...; // Transfermatrizen
vector< BLAS::Matrix< T > >    E = { E0, E1 };
TClusterBasis< T >             cb( is, sons, E );
```

Konstruktion aus $\mathcal{H}$-Matrix

Desweiteren kann eine bestehende $\mathcal{H}$-Matrix in eine $\mathcal{H}^2$-Matrix mit einer gewünschten Genauigkeit ε konvertiert werden:

```
THClusterBasisBuilder< T >  bbuilder;
TClusterBasis< T > *        rowcb;
TClusterBasis< T > *        colcb;

tie( rowcb, colcb ) = bbuilder.build( A, TTruncAcc( eps ) );

TMatrix *                   A2 = to_h2( A, rowcb, colcb );
```

Die Konvertierung von $\mathcal{H}$ nach $\mathcal{H}^2$ während der Konstruktion der $\mathcal{H}$-Matrix ist derzeit noch nicht möglich.

Die $\mathcal{H}^2$-Matrix-Vektor-Multiplikation $Ax = y$ besteht aus drei Schritten:

1. Vorwärtstransformation: Projektion von $x|_s$ in den von W^s aufgespannten Raum: $S_x^s := W^{H,s} x|_s$,
2. Multiplikation für alle Matrizen: $S_y^t := S^{t,s} \cdot S_x^s$
3. Rückwärtstransformation: Rückprojektion $V^t \cdot S_y^t$ und Aufsummieren der einzelnen Anteile

Der Aufwand für die $\mathcal{H}^2$-Matrix-Vektor-Multiplikation reduziert sich im Vergleich zu den $\mathcal{H}$-Matrizen auf

$$\mathcal{O}(n)$$

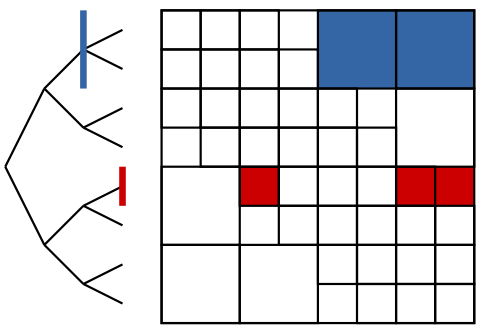
Die $\mathcal{H}^2$-Matrix-Vektor-Multiplikation $Ax = y$ besteht aus drei Schritten:

1. Vorwärtstransformation: Projektion von $x|_s$ in den von W^s aufgespannten Raum: $S_x^s := W^{H,s} x|_s$,
2. Multiplikation für alle Matrizen: $S_y^t := S^{t,s} \cdot S_x^s$
3. Rückwärtstransformation: Rückprojektion $V^t \cdot S_y^t$ und Aufsummieren der einzelnen Anteile

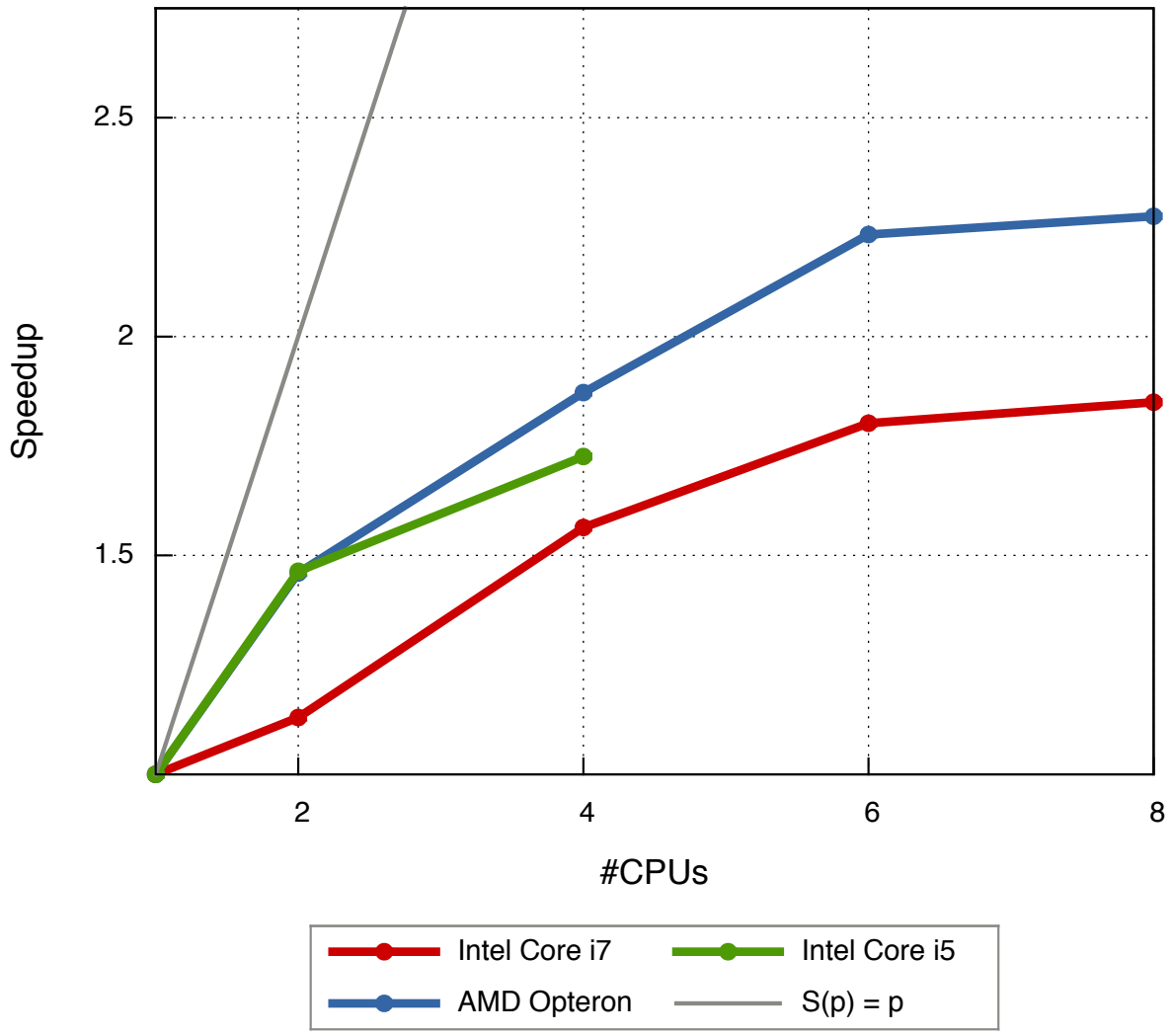
Der Aufwand für die $\mathcal{H}^2$-Matrix-Vektor-Multiplikation reduziert sich im Vergleich zu den $\mathcal{H}$-Matrizen auf

$$\mathcal{O}(n)$$

In der HLIBpro ist die Multiplikation aktuell für threadparallele Systeme in einer *einfachen* Version implementiert, d.h. paralleles Abarbeiten für Knoten des Clusterbaumes.



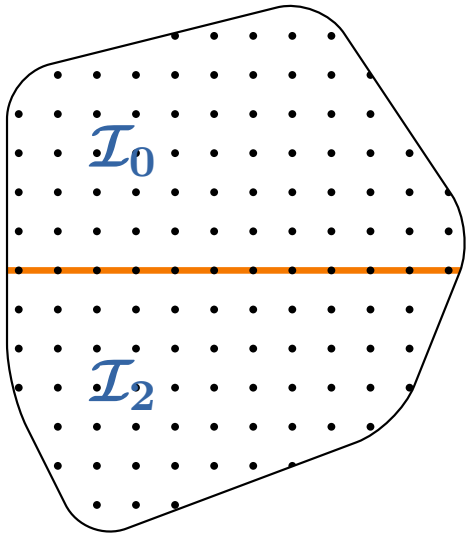
Num. Resultate, Helmholtz-DLP im $\mathbb{R}^3$:

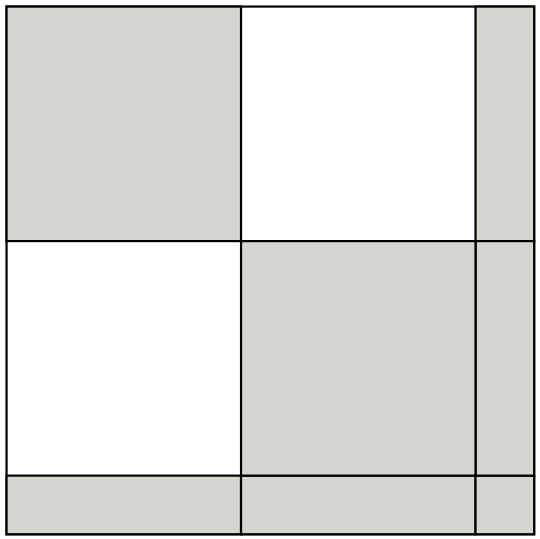


Und sonst?

Und sonst? ⟩ Nested Dissection

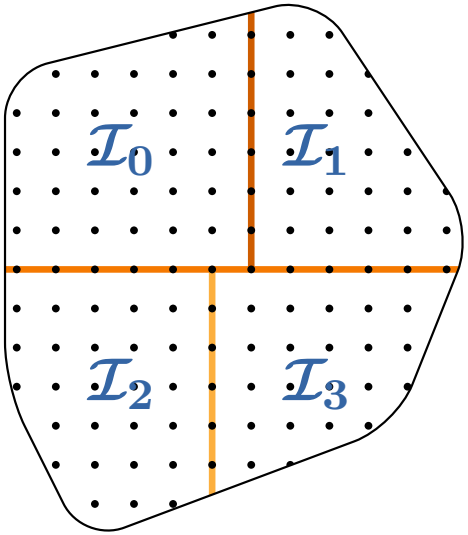
Für schwachbesetzte Systeme läßt sich eine spezielle Clusterung vornehmen, welche die Indizes durch einen inneren Rand rekursiv entkoppelt:

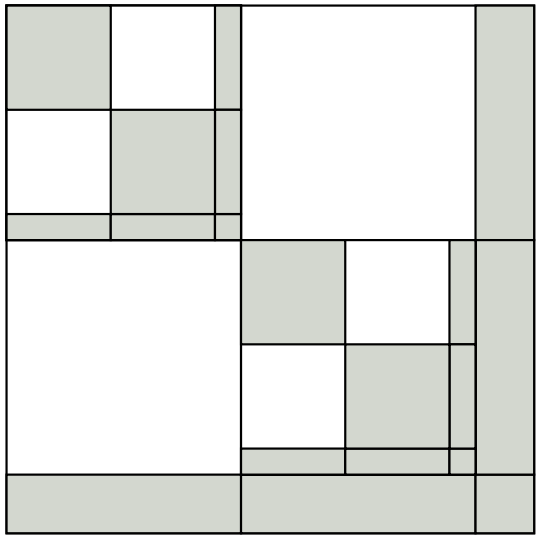




Und sonst? ⟩ Nested Dissection

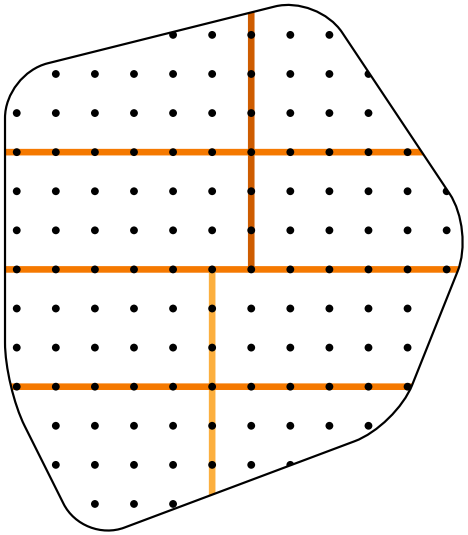
Für schwachbesetzte Systeme läßt sich eine spezielle Clusterung vornehmen, welche die Indizes durch einen inneren Rand rekursiv entkoppelt:

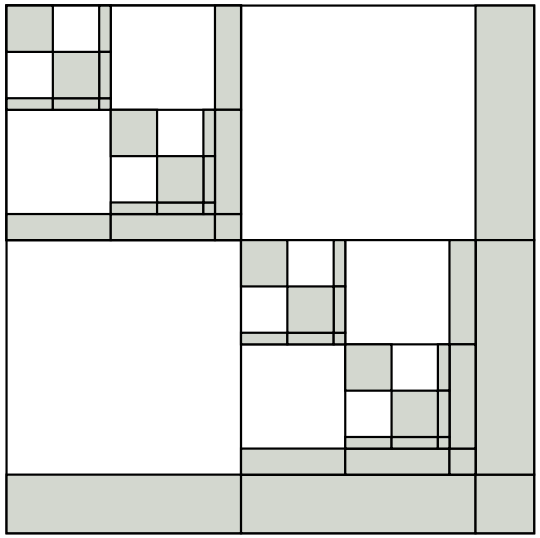




Und sonst? ⟩ Nested Dissection

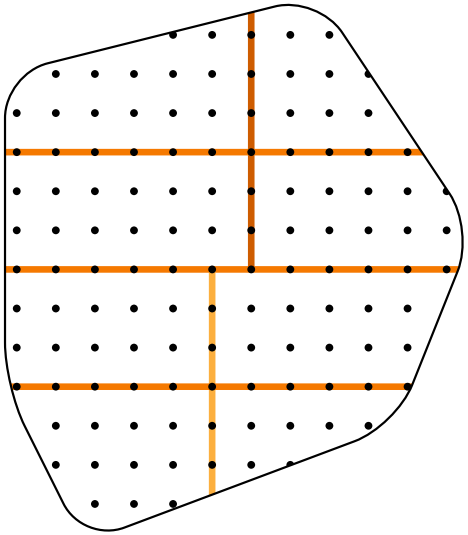
Für schwachbesetzte Systeme läßt sich eine spezielle Clusterung vornehmen, welche die Indizes durch einen inneren Rand rekursiv entkoppelt:

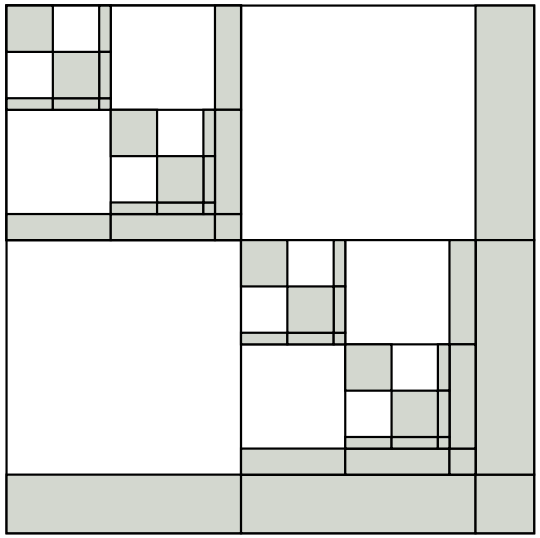




Und sonst? ⟩ Nested Dissection

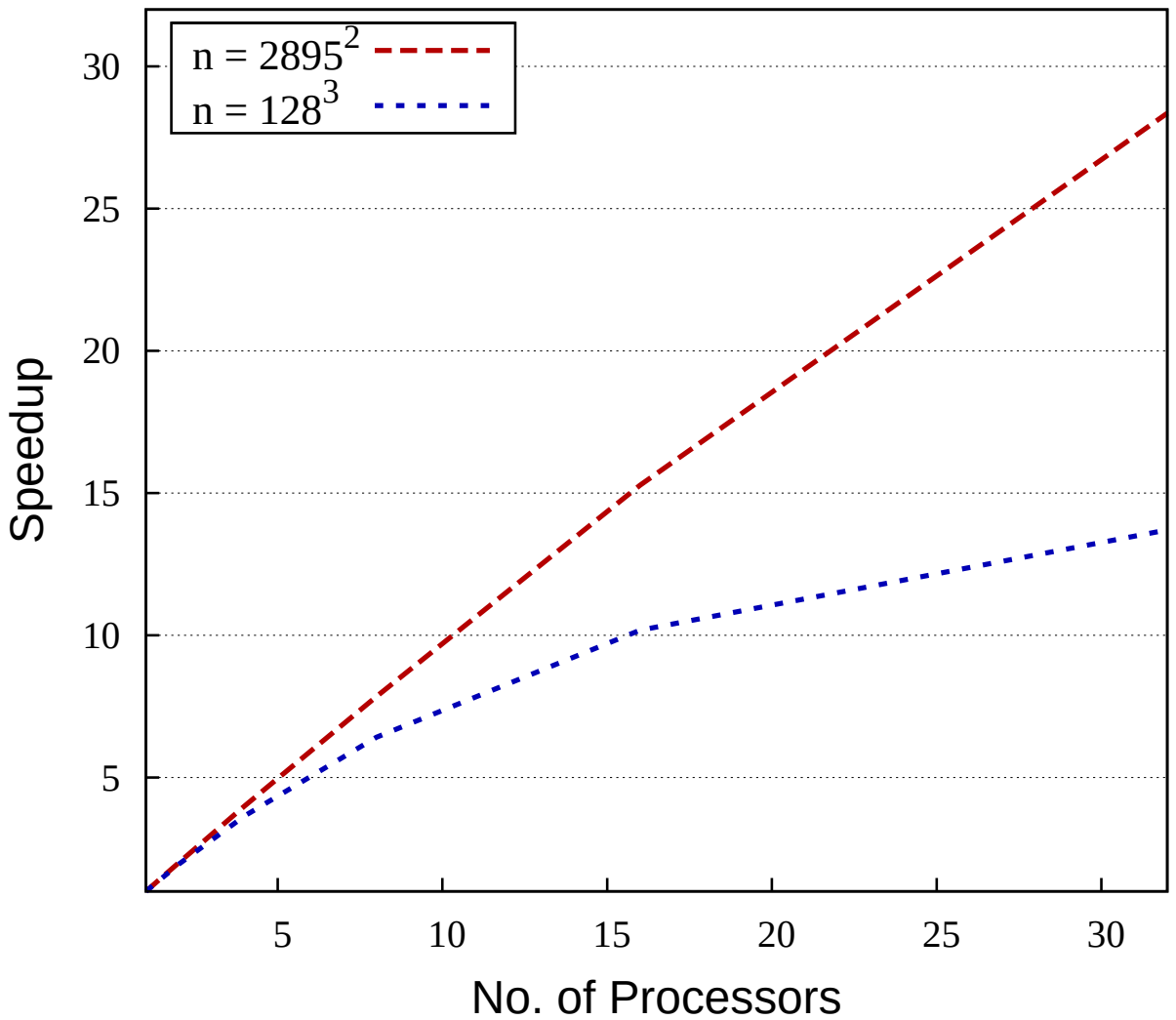
Für schwachbesetzte Systeme läßt sich eine spezielle Clusterung vornehmen, welche die Indizes durch einen inneren Rand rekursiv entkoppelt:





Hierdurch ergeben sich große Nullblöcke, die sich auch während der $\mathcal{H}$-LU-Zerlegung nicht auffüllen. Das Ergebnis ist eine wesentliche Beschleunigung der $\mathcal{H}$-Arithmetik. Desweiteren ist die $\mathcal{H}$-LU-Zerlegung inherent parallel.

Num. Resultate, Poisson-Problem im $\mathbb{R}^2$ und $\mathbb{R}^3$:

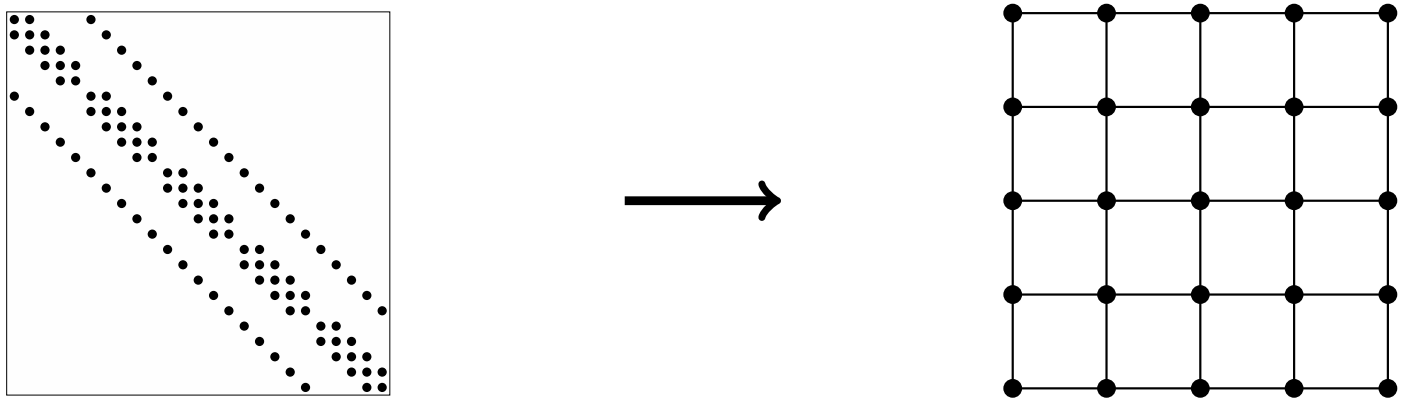


Oftmals steht ausschließlich die schwachbesetzte Matrix für ein Gleichunssystem zur Verfügung, d.h. ohne Geometrieinformationen. Unter Verwendung des *Matrixgraphen* und *Graphenpartitionierung* kann dann trotzdem ein Cluster- und Blockclusterbaum konstruiert werden.

Matrixgraph

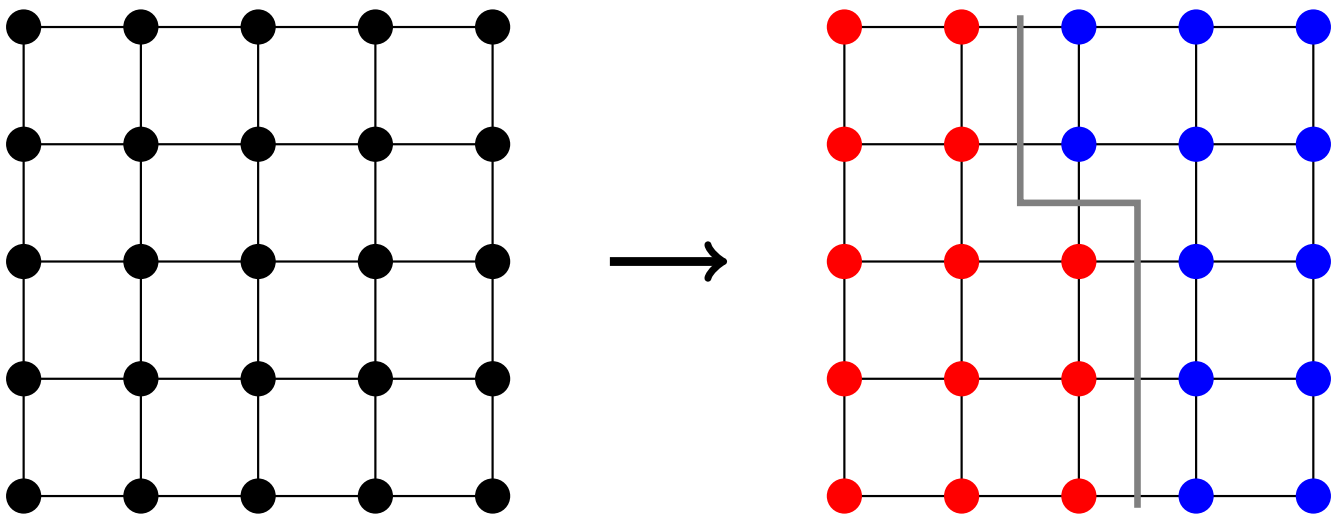
Der Matrixgraph $G(A) = (V, E)$ von A ist def. durch $V := \mathcal{I}$ und

$$E := \{(i,j) \in I \times I \ : \ i \neq j \wedge (a_{ij} \neq 0 \vee a_{ji} \neq 0)\},$$



Graphenpartitionierung

Zum Graphen $G = (V, E)$ ist eine Partitionierung $V = P_1 \cup P_2$ mit $\#P1 \sim \#P2$ und minimalem Kantenschnitt gesucht.

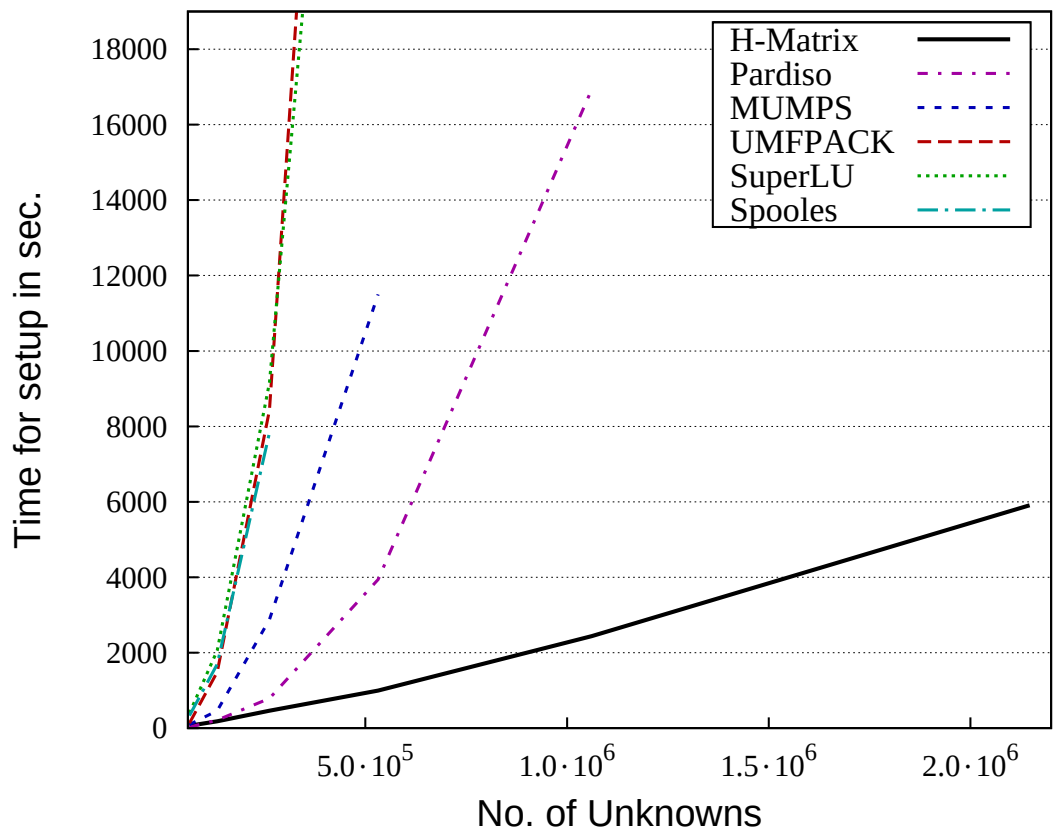


Das Problem ist zwar NP-vollständig, es existieren aber gute Approximationsverfahren (z.B. METIS).

Black-Box-$\mathcal{H}$-LU-Zerlegung

In HLIBpro werden eigene Verfahren zur Graphenpartitionierung genutzt als auch z.B. METIS unterstützt. Das Verfahren läßt sich außerdem mit „nested dissection“ kombinieren.

Die so definierte *Blackbox-$\mathcal{H}$-LU-Zerlegung* steht z.B. in Konkurrenz mit direkten Lösern oder algebraischen Mehrgitterverfahren.



Und sonst? ⟩ BEM-Kern

HLIBpro implementiert verschiedenste Routinen für die numerische Behandlung von Integralgleichungen:

- Integralkerne:
  - Laplace SLP und DLP,
  - Helmholtz SLP und DLP (plus akustische Streuung),
  - Maxwell EFIE und MFIE,
  - Massematrix
- Basisfunktionen: stückweise konstant und linear,
- volle SSE2/AVX-Beschleunigung der Berechnung der Matrixkoeffizienten (Sauter-Schwab-Quadratur),
- HCA-Kollokationsintegrale für Laplace und Helmholtz,
- Darstellung der Oberflächen als Dreiecksgitter; I/O für verschiedenste Dateiformate.

Beispiel für Helmholtz SLP

```
typedef  TConstFnSpace                                ansatzsp_t;
typedef  TLinearFnSpace                               testsp_t;
typedef  THelmholtzSLPBF< ansatzsp_t, testsp_t >      bf_t;
typedef  TBFCoeffFn< slpbf_t >                        coefffn_t;

TGrid *          grid           = grid_io.read( grid_filename );
ansatzsp_t       ansatz_fnspace = new ansatzsp_t( grid.get() );
testsp_t         test_fnspace   = new testsp_t( grid.get() );
TCoordinate *    coord1         = test_fnspace->build_coord();
TCoordinate *    coord2         = ansatz_fnspace->build_coord();

...

bf_t             bf( kappa, ansatz_fnspace, test_fnspace, 4 );
coefffn_t        coeff_fn( & bf, row_ct->perm_i2e(), col_ct->perm_i2e() );
TACAPlus         aca( & coeff_fn );
TDenseMBuilder   h_builder( & coeff_fn, & aca );
```

HLIBpro.com