$\mathcal{H}$-Arithmetic for Many-Core Systems

Ronald Kriemann
MPI MIS

Universität Kiel

2015-10-22

Outline

1. $\mathcal{H}$-Matrices
2. Classical $\mathcal{H}$-Matrix Arithmetic
3. Dense LU Factorization
4. Task based $\mathcal{H}$-Arithmetic

$\mathcal{H}$-Matrices

Singular Value Decomposition

Let $M \in \mathbb{R}^{n \times m}$ and $\ell := \min(n, m)$. Then there exist orthogonal matrices $U \in \mathbb{R}^{n \times \ell}$, $V \in \mathbb{R}^{m \times \ell}$ and diagonal $S \in \mathbb{R}^{\ell \times \ell}$ such that

$$M = USV^T = \sum_{i=0}^{\ell-1} s_i u_i v_i^T$$

with *singular values* $s_i := S_{ii}, s_0 \geq s_1 \geq \ldots \geq s_\ell \geq 0$, and *singular vectors* $u_i := U(:, i)$, $v_i := V(:, i)$.

Singular Value Decomposition

Let $M \in \mathbb{R}^{n \times m}$ and $\ell := \min(n, m)$. Then there exist orthogonal matrices $U \in \mathbb{R}^{n \times \ell}$, $V \in \mathbb{R}^{m \times \ell}$ and diagonal $S \in \mathbb{R}^{\ell \times \ell}$ such that

$$M = USV^T = \sum_{i=0}^{\ell-1} s_i u_i v_i^T$$

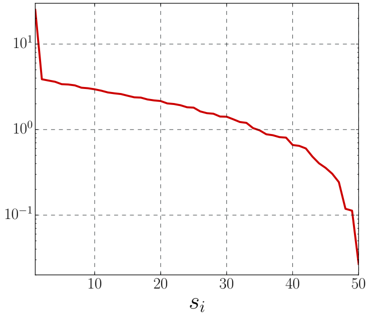
with *singular values* $s_i := S_{ii}, s_0 \geq s_1 \geq \ldots \geq s_\ell \geq 0$, and *singular vectors* $u_i := U(:, i)$, $v_i := V(:, i)$.

The *best approximation* of M with *rank* k w.r.t. $\|\cdot\|_2$ is defined by

$$M_k := \sum_{i=0}^{k-1} s_i u_i v_i^T \quad \text{with} \quad \|M - M_k\|_2 = s_k.$$

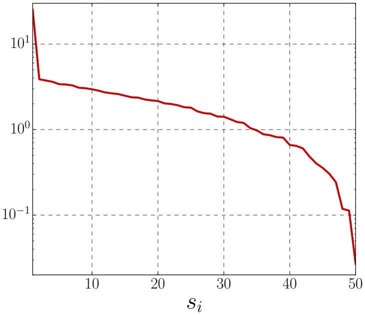
M_k is called a *low-rank* approximation of M if $\|M - M_k\|_2 < \varepsilon$, $\varepsilon \geq 0$, and $k \ll \ell$.

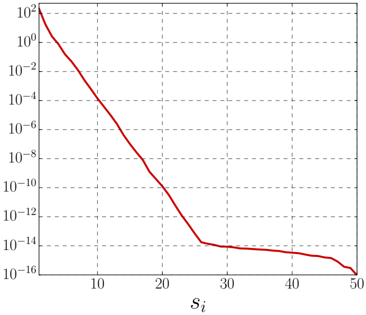
Examples



- most singular values of comparable size,
- *not* approximable by low-rank matrix.

Examples





- most singular values of comparable size,
- *not* approximable by low-rank matrix.

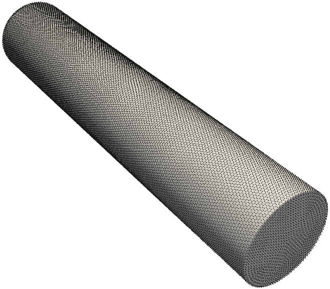
- singular values decay exponentially,
- *good* approximable by low-rank matrix.

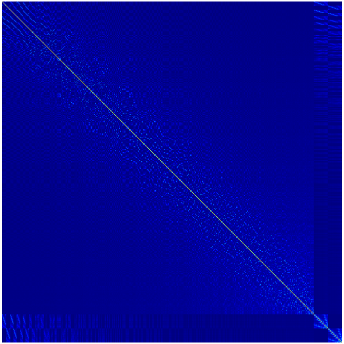
Model Problem

Let A be a matrix defined by the discretization of the integral equation

$$\int_\Gamma \frac{1}{\|x-y\|} u(x) dy = f(x), \quad x \in \Gamma$$

over a domain $\Gamma = \partial\Omega \subset \mathbb{R}^3$.





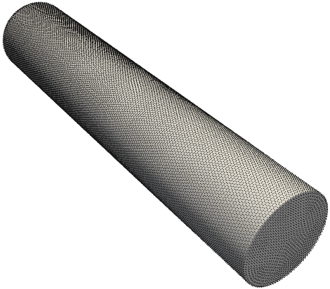
$n = 652, \varepsilon = 10^{-8}$

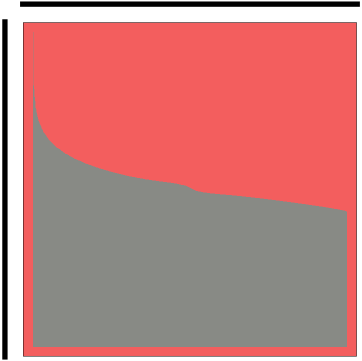
Model Problem

Let A be a matrix defined by the discretization of the integral equation

$$\int_\Gamma \frac{1}{\|x-y\|} u(x) dy = f(x), \quad x \in \Gamma$$

over a domain $\Gamma = \partial\Omega \subset \mathbb{R}^3$.





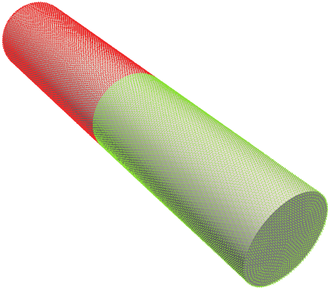
$n = 652, \varepsilon = 10^{-8}$

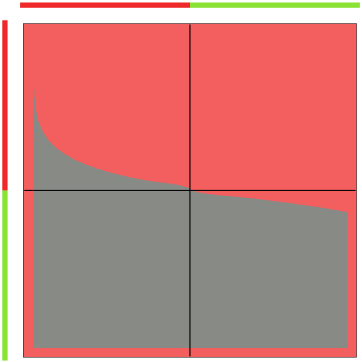
Model Problem

Let A be a matrix defined by the discretization of the integral equation

$$\int_\Gamma \frac{1}{\|x-y\|} u(x) dy = f(x), \quad x \in \Gamma$$

over a domain $\Gamma = \partial\Omega \subset \mathbb{R}^3$.





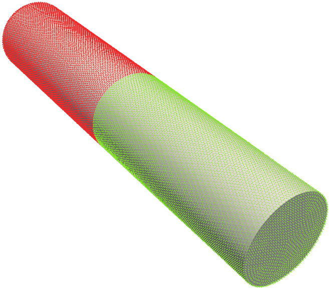
$n = 652, \varepsilon = 10^{-8}$

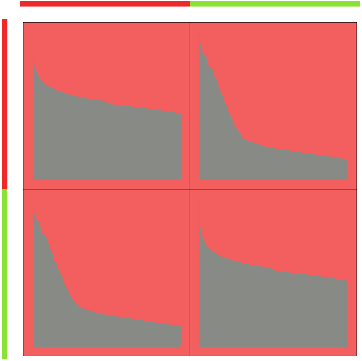
Model Problem

Let A be a matrix defined by the discretization of the integral equation

$$\int_\Gamma \frac{1}{\|x-y\|} u(x)dy = f(x), \quad x \in \Gamma$$

over a domain $\Gamma = \partial\Omega \subset \mathbb{R}^3$.





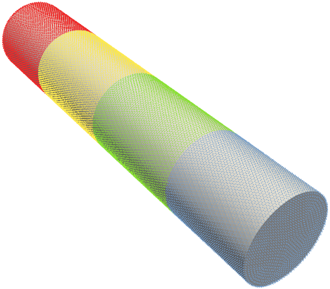
$n = 652, \varepsilon = 10^{-8}$

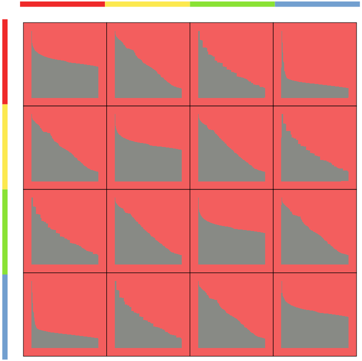
Model Problem

Let A be a matrix defined by the discretization of the integral equation

$$\int_\Gamma \frac{1}{\|x-y\|} u(x) dy = f(x), \quad x \in \Gamma$$

over a domain $\Gamma = \partial\Omega \subset \mathbb{R}^3$.





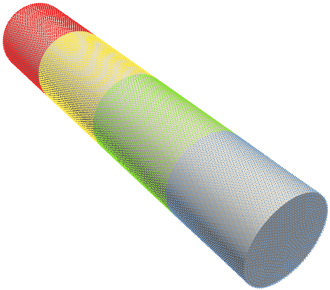
$n = 652, \varepsilon = 10^{-8}$

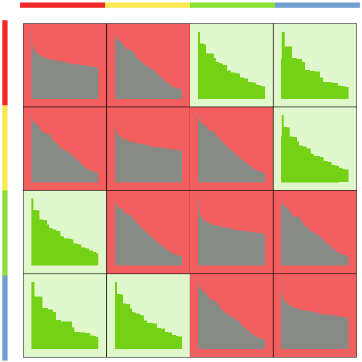
Model Problem

Let A be a matrix defined by the discretization of the integral equation

$$\int_\Gamma \frac{1}{\|x-y\|} u(x) dy = f(x), \quad x \in \Gamma$$

over a domain $\Gamma = \partial\Omega \subset \mathbb{R}^3$.





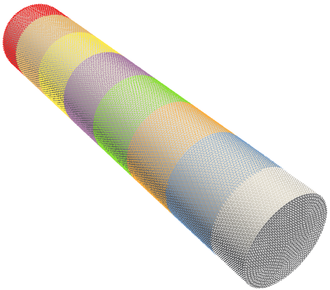
$n = 652, \varepsilon = 10^{-8}$

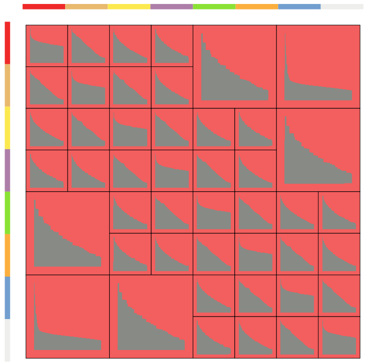
Model Problem

Let A be a matrix defined by the discretization of the integral equation

$$\int_\Gamma \frac{1}{\|x-y\|} u(x) dy = f(x), \quad x \in \Gamma$$

over a domain $\Gamma = \partial\Omega \subset \mathbb{R}^3$.





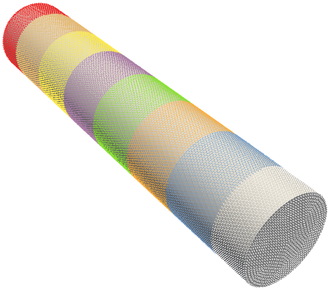
$n = 652, \varepsilon = 10^{-8}$

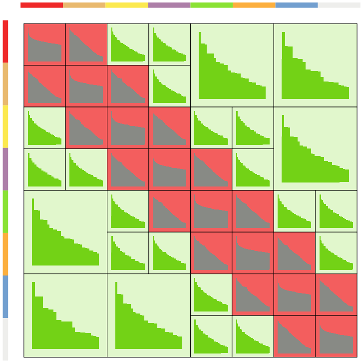
Model Problem

Let A be a matrix defined by the discretization of the integral equation

$$\int_\Gamma \frac{1}{\|x-y\|} u(x) dy = f(x), \quad x \in \Gamma$$

over a domain $\Gamma = \partial\Omega \subset \mathbb{R}^3$.



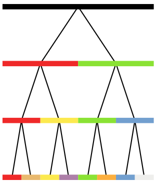


$n = 652, \varepsilon = 10^{-8}$

Definitions

Let $I := \{0, \ldots, n-1\}$. The *hierarchical* partitioning of the index set I forms the (binary) *cluster tree* $T(I)$.

For each node $I \supseteq t \in T(I)$ let $\mathcal{S}(t)$ be the set of sons of t (either $\mathcal{S}(t) = \emptyset$ or $\mathcal{S}(t) = \{t_0, t_1\}$).

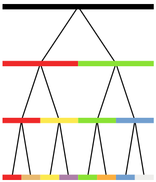


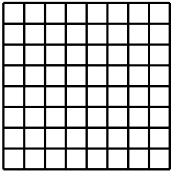
Definitions

Let $I := \{0, \ldots, n-1\}$. The *hierarchical* partitioning of the index set I forms the (binary) *cluster tree* $T(I)$.

For each node $I \supseteq t \in T(I)$ let $\mathcal{S}(t)$ be the set of sons of t (either $\mathcal{S}(t) = \emptyset$ or $\mathcal{S}(t) = \{t_0, t_1\}$).

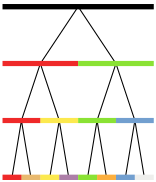
The hierarchical partitioning of $I \times I$ based on $T(I)$ forms the *block cluster tree* $T(I \times I)$. Let $\mathcal{L}(T(I \times I))$ be the set of leaves of $T(I \times I)$.





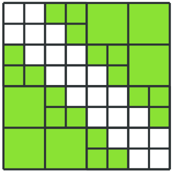
Definitions

Let $I := \{0, \ldots, n-1\}$. The *hierarchical* partitioning of the index set I forms the (binary) *cluster tree* $T(I)$.



For each node $I \supseteq t \in T(I)$ let $\mathcal{S}(t)$ be the set of sons of t (either $\mathcal{S}(t) = \emptyset$ or $\mathcal{S}(t) = \{t_0, t_1\}$).

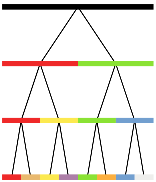
The hierarchical partitioning of $I \times I$ based on $T(I)$ forms the *block cluster tree* $T(I \times I)$. Let $\mathcal{L}(T(I \times I))$ be the set of leaves of $T(I \times I)$.



Block clusters $t \times s \in T(I \times I)$ with low-rank approximations are determined by an *admissibility condition*:

$$\min(\operatorname{diam}(t), \operatorname{diam}(s)) \leq \eta \operatorname{dist}(t, s), \;\; \eta \geq 0.$$

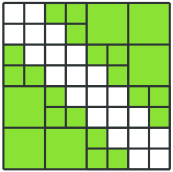
Definitions



Let $I := \{0, \ldots, n-1\}$. The *hierarchical* partitioning of the index set I forms the (binary) *cluster tree* $T(I)$.

For each node $I \supseteq t \in T(I)$ let $\mathcal{S}(t)$ be the set of sons of t (either $\mathcal{S}(t) = \emptyset$ or $\mathcal{S}(t) = \{t_0, t_1\}$).

The hierarchical partitioning of $I \times I$ based on $T(I)$ forms the *block cluster tree* $T(I \times I)$. Let $\mathcal{L}(T(I \times I))$ be the set of leaves of $T(I \times I)$.



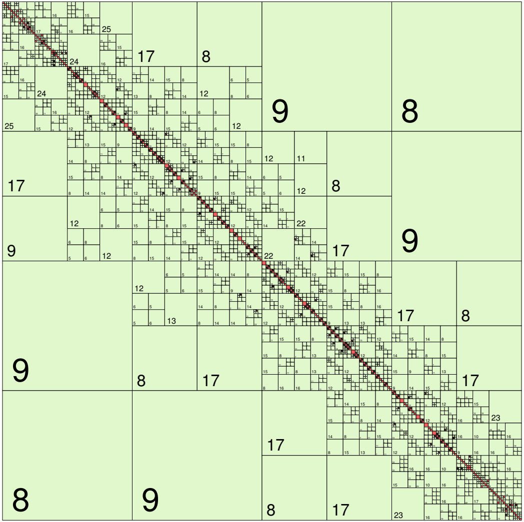
Block clusters $t \times s \in T(I \times I)$ with low-rank approximations are determined by an *admissibility condition*:

$$\min(\operatorname{diam}(t), \operatorname{diam}(s)) \leq \eta \operatorname{dist}(t, s), \;\; \eta \geq 0.$$

Finally, let

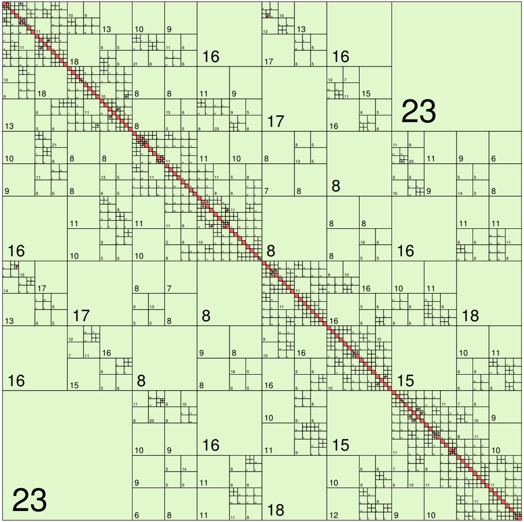
$$\mathcal{H}(T(I \times I), k) := \{M \in \mathbb{R}^{I \times I} \text{ with } \operatorname{rank}(M|_{t \times s}) \leq k \text{ for all admissible } t \times s \in T(I \times I)\}$$

Examples



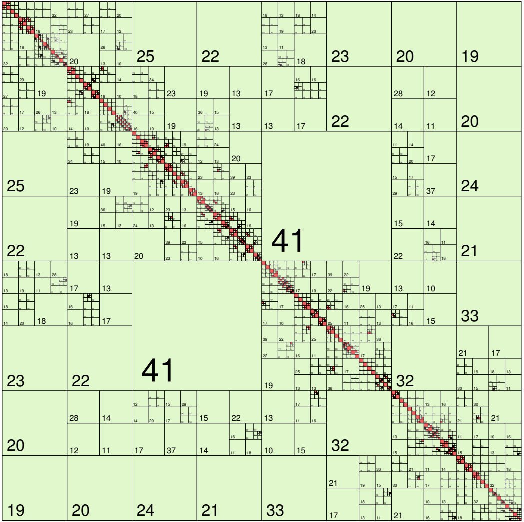
$n = 10432, \varepsilon = 10^{-4}$

Examples



$n = 7808, \varepsilon = 10^{-4}$

Examples



Helmholtz, $n = 9344, \varepsilon = 10^{-4}$

Complexity

The complexity for storing an $\mathcal{H}$-matrix in $\mathcal{H}(T(I \times I), k)$ is

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

For $\mathcal{H}$-matrix arithmetic, one obtains the following complexity:

- Matrix-Vector Mult., Addition: $\mathcal{O}(n \log n)$,
- Multiplication, Inversion, $\mathcal{H}$-LU: $\mathcal{O}(n \log^2 n)$,

Complexity

The complexity for storing an $\mathcal{H}$-matrix in $\mathcal{H}(T(I \times I), k)$ is

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

For $\mathcal{H}$-matrix arithmetic, one obtains the following complexity:

- Matrix-Vector Mult., Addition: $\mathcal{O}(n \log n)$,
- Multiplication, Inversion, $\mathcal{H}$-LU: $\mathcal{O}(n \log^2 n)$,

Truncated Addition

Let $A, B \in \mathcal{H}(T(I \times I), k)$. Then, in general for $A + B$ we have

$$A + B \in \mathcal{H}(T(I \times I), 2 \cdot k)$$

Complexity

The complexity for storing an $\mathcal{H}$-matrix in $\mathcal{H}(T(I \times I), k)$ is

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

For $\mathcal{H}$-matrix arithmetic, one obtains the following complexity:

- Matrix-Vector Mult., Addition: $\mathcal{O}(n \log n)$,
- Multiplication, Inversion, $\mathcal{H}$-LU: $\mathcal{O}(n \log^2 n)$,

Truncated Addition

Let $A, B \in \mathcal{H}(T(I \times I), k)$. Then, in general for $A + B$ we have

$$A + B \in \mathcal{H}(T(I \times I), 2 \cdot k)$$

Therefore, always a *truncated* addition is used: for all low-rank sub blocks of $A + B$ compute the best approximation with rank k.
For this, a *low-rank SVD* algorithm with complexity $\mathcal{O}(nk^2 + k^3)$ is used.

Classical $\mathcal{H}$-Matrix Arithmetic

Matrix Multiplication

Let $A, B, C \in \mathcal{H}(T(I \times I), k)$. Then each sub block $A|_{t \times s}$, $t \times s \in T(I \times I)$, of A is either a *low-rank matrix*, a *dense matrix* or a *block matrix* with subblocks

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} := \begin{pmatrix} A_{t_0 \times s_0} & A_{t_0 \times s_1} \\ A_{t_1 \times s_0} & A_{t_1 \times s_1} \end{pmatrix}$$

Matrix Multiplication

Let $A, B, C \in \mathcal{H}(T(I \times I), k)$. Then each sub block $A|_{t \times s}$, $t \times s \in T(I \times I)$, of A is either a *low-rank matrix*, a *dense matrix* or a *block matrix* with subblocks

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} := \begin{pmatrix} A_{t_0 \times s_0} & A_{t_0 \times s_1} \\ A_{t_1 \times s_0} & A_{t_1 \times s_1} \end{pmatrix}$$

The product $C := C + \alpha A \cdot B$ is computed by:

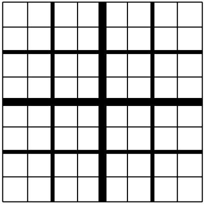
procedure $\textsc{Multiply}(\alpha, A, B, C)$
  if A, B, C are block matrices **then**
    for $i \in \{0, 1\}$ **do**
      for $j \in \{0, 1\}$ **do**
        for $\ell \in \{0, 1\}$ **do**
          $\textsc{Multiply}(\alpha, A_{ij}, B_{i\ell}, C_{\ell j});$
  else
    $C := C + \alpha AB;$ *//specialized functions*

Properties

procedure MULTIPLY(α, A, B, C)
  if A, B, C are block matrices **then**
    for $i, j, \ell \in \{0, 1\}$ **do**
      MULTIPLY($\alpha, A_{ij}, B_{i\ell}, C_{\ell j}$);
  else
    $C := C + \alpha AB$;

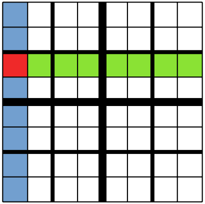
Algorithm has same structure as for (block-wise) dense matrices.

Special "$\mathcal{H}$-matrix" algorithms only used for low-rank sub blocks.



Differences to Dense Multiplication

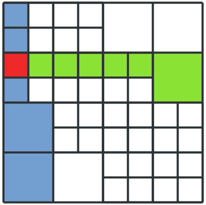
procedure MULTIPLY(α, A, B, C)
  if A, B, C are block matrices **then**
    for $i, j, \ell \in \{0, 1\}$ **do**
      MULTIPLY($\alpha, A_{ij}, B_{i\ell}, C_{\ell j}$);
  else
    $C := C + \alpha AB$;



Costs per sub block:

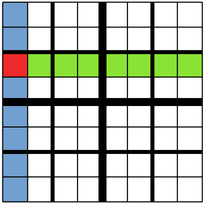
Dense: equal costs,

$\mathcal{H}$: depends on structure and rank



Differences to Dense Multiplication

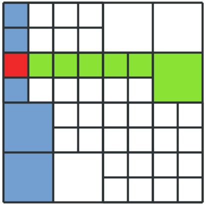
procedure MULTIPLY(α, A, B, C)
  if A, B, C are block matrices **then**
    for $i, j, \ell \in \{0, 1\}$ **do**
      MULTIPLY$(\ \alpha, A_{ij}, B_{i\ell}, C_{\ell j}\ )$;
  else
    $C := C + \alpha AB$;



Costs per sub block:

Dense: equal costs,

$\mathcal{H}$: depends on structure and rank



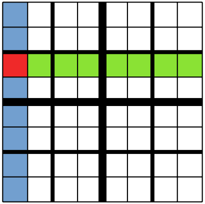
Collisions:

Dense: only for operations on same sub block,

$\mathcal{H}$: possible for all non-disjoint sub blocks.

Differences to Dense Multiplication

procedure MULTIPLY(α, A, B, C)
  if A, B, C are block matrices **then**
    for $i, j, \ell \in \{0, 1\}$ **do**
      MULTIPLY$(\alpha, A_{ij}, B_{i\ell}, C_{\ell j})$;
  else
    $C := C + \alpha AB$;

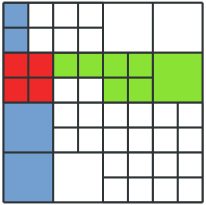


Costs per sub block:

- Dense: equal costs,
- $\mathcal{H}$: depends on structure and rank

Collisions:

- Dense: only for operations on same sub block,
- $\mathcal{H}$: possible for all non-disjoint sub blocks.



Parallelization

Static top-down block-to-processor distribution (as in dense case) not sufficient due to different cost per block.

The number of destination blocks (up to *all* blocks in $T(I \times I)$) is high compared to number of processors.

Hence, no parallelization of per-sub block operations needed. Only handle collisions from non-disjoint blocks.

procedure $\textsc{Multiply}(\alpha, A, B, C)$
  if A, B, C are block matrices **then**
    for $i, j, \ell \in \{0, 1\}$ **do**
      $\textsc{Multiply}(\ \alpha, A_{ij}, B_{i\ell}, C_{\ell j}\ );$
  else
    $\mathcal{M}_C = \mathcal{M}_C \cup \{(A, B)\};$ *//Collect factors*

for all sub blocks $C|_{t \times s}, t \times s \in T(I \times I)$ **do** *//parallel for*
  for $(A, B) \in \mathcal{M}_{C|_{t \times s}}$ **do**
    $C|_{t \times s} := C|_{t \times s} + \alpha AB;$

Numerical Results

Parallel speedup for model problem with $n = 32.768$:

	Speedup
Intel Xeon E5-2670 (2x8 cores, 2 hyperthreads)	17.84
Intel XeonPhi 5110P (60 cores, 4 hyperthreads)	90.27

$\mathcal{H}$-LU Factorization

Again, let $A \in \mathcal{H}(T(I \times I), k)$. The LU factorisation $A = LU$ is defined by the block structure of A. If $A|_{t \times t}, t \in T(I)$ is a block matrix, then we have:

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

which leads to the following equations:

$$\begin{aligned}
A_{00} &= L_{00}U_{00} && \text{(Recursion)} \\
A_{01} &= L_{00}U_{01} && \text{(Matrix Solve)} \\
A_{10} &= L_{10}U_{00} && \text{(Matrix Solve)} \\
A_{11} &= A_{11} - L_{10}U_{01} && \text{(Multiplication)} \\
A_{11} &= L_{11}U_{11} && \text{(Recursion)}
\end{aligned}$$

$\mathcal{H}$-LU Factorization

The above equations directly translate into the following algorithm for the $\mathcal{H}$-LU factorisation:

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
    $U := A^{-1}; L := I;$

$\mathcal{H}$-LU Factorization

The above equations directly translate into the following algorithm for the $\mathcal{H}$-LU factorisation and matrix solves:

```
procedure LU(A, L, U)
   if A is block matrix then
      LU( A_00, L_00, U_00 );
      SolveLL( A_01, L_00, U_01 );
      SolveUR( A_10, L_10, U_00 );
      Multiply( −1, L_10, U_01, A_11 );
      LU( A_11, L_11, U_11 );
   else
      U := A^{-1}; L := I;
```

```
procedure SolveLL(A, L, B)
   if A, L, B are block matrices then
      SolveLL( A_00, L_00, B_00 );
      SolveLL( A_01, L_00, B_01 );
      Multiply( −1, L_10, B_00, A_11 );
      Multiply( −1, L_10, B_01, A_11 );
      SolveLL( A_10, L_11, B_10 );
      SolveLL( A_11, L_11, B_11 );
   else
      B := L^{-1} A;
```

$\mathcal{H}$-LU Factorization

The above equations directly translate into the following algorithm for the $\mathcal{H}$-LU factorisation and matrix solves:

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

procedure $\mathrm{SolveLL}(A, L, B)$
  if A, L, B are block matrices **then**
    $\mathrm{SolveLL}(\ A_{00}, L_{00}, B_{00}\ );$
    $\mathrm{SolveLL}(\ A_{01}, L_{00}, B_{01}\ );$
    $\mathrm{Multiply}(\ -1, L_{10}, B_{00}, A_{11}\ );$
    $\mathrm{Multiply}(\ -1, L_{10}, B_{01}, A_{11}\ );$
    $\mathrm{SolveLL}(\ A_{10}, L_{11}, B_{10}\ );$
    $\mathrm{SolveLL}(\ A_{11}, L_{11}, B_{11}\ );$
  else
    $B := L^{-1}A;$

Both procedures only consist of *recursion* and *matrix multiplication*.

Remark

Again: algorithm structure as for dense matrices with specialised algorithms only for low-rank sub blocks.

Parallelization

The recursive algorithm is inherently *sequential*. Only the matrix solves may be performed in parallel:

procedure $\text{LU}(A, L, U)$
$\quad \text{LU}(\ A_{00}, L_{00}, U_{00}\ );$
$\quad \{\ \textsc{SolveLL}(\ A_{01}, L_{00}, U_{01}\ )\ |\ \textsc{SolveUR}(\ A_{10}, L_{10}, U_{00}\ );\ \}$
$\quad \textsc{Multiply}(\ -1, L_{10}, U_{01}, A_{11}\ );$
$\quad \text{LU}(\ A_{11}, L_{11}, U_{11}\ );$

Parallelization

The recursive algorithm is inherently *sequential*. Only the matrix solves may be performed in parallel:

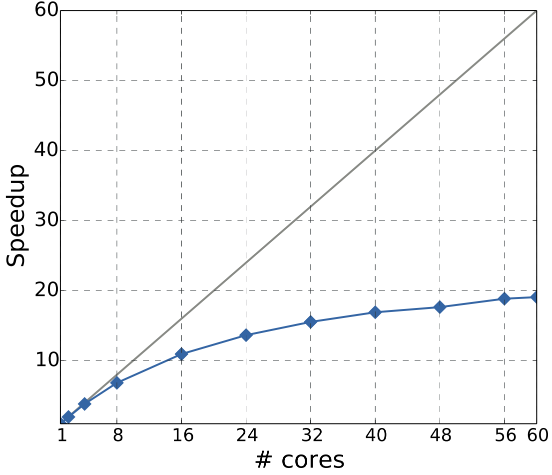
procedure $\text{LU}(A, L, U)$
$\quad \text{LU}(\ A_{00}, L_{00}, U_{00}\ );$
$\quad \{\ \text{SolveLL}(\ A_{01}, L_{00}, U_{01}\ )\ \|\ \text{SolveUR}(\ A_{10}, L_{10}, U_{00}\ );\ \}$
$\quad \text{Multiply}(\ -1, L_{10}, U_{01}, A_{11}\ );$
$\quad \text{LU}(\ A_{11}, L_{11}, U_{11}\ );$

Matrix solve algorithm can be parallelised only slightly better:

procedure $\text{SolveLL}(A, L, B)$
$\quad \{\ \text{SolveLL}(\ A_{00}, L_{00}, B_{00}\ );\ \|\ \text{SolveLL}(\ A_{01}, L_{00}, B_{01}\ );\ \}$
$\quad \{\ \text{Multiply}(\ -1, L_{10}, B_{00}, A_{10}\ );\ \|\ \text{Multiply}(\ -1, L_{10}, B_{01}, A_{11}\ );\ \}$
$\quad \{\ \text{SolveLL}(\ A_{10}, L_{11}, B_{10}\ );\ \|\ \text{SolveLL}(\ A_{11}, L_{11}, B_{11}\ );\ \}$

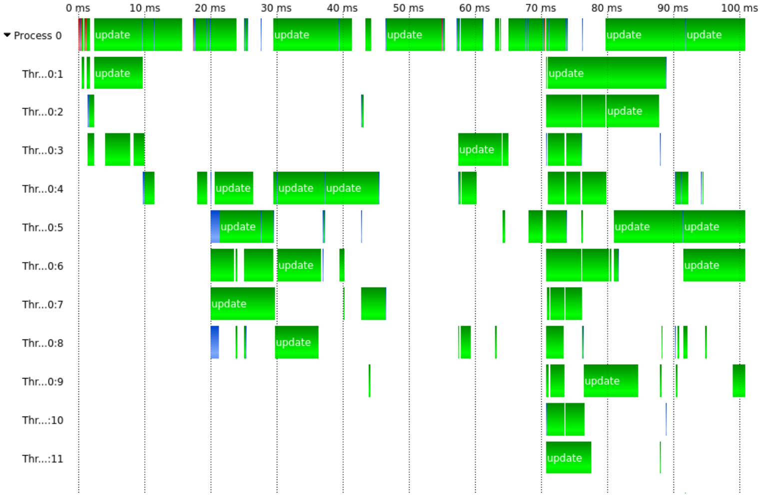
Numerical Results

Parallel speedup for model problem with $n = 32.768$:



(XeonPhi 5110P)

Function trace of $\mathcal{H}$-LU factorisation



(Xeon E5-2640)

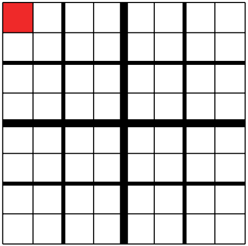
Dense LU Factorization

Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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
    $A := LU$;

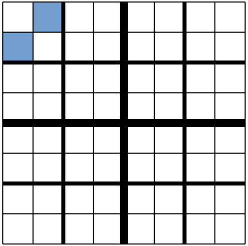


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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

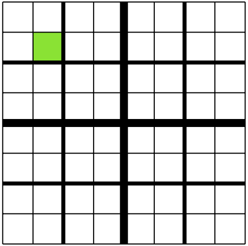


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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

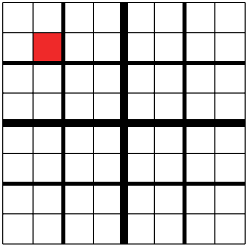


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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

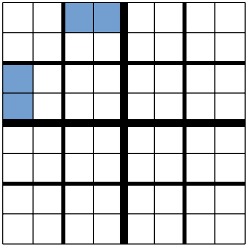


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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

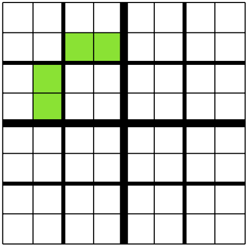


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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
    $A := LU$;

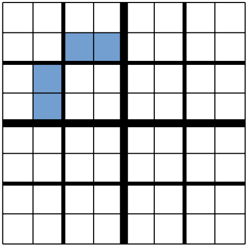


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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
    $A := LU$;

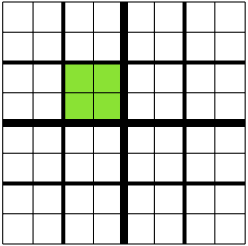


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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

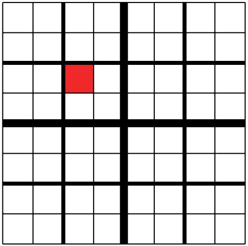


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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
    $A := LU$;

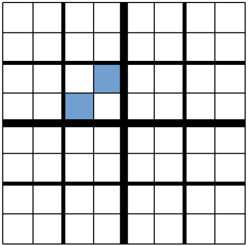


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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
    $A := LU$;

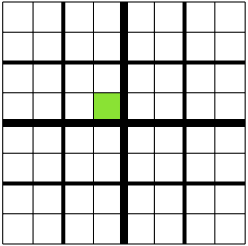


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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
    $A := LU$;

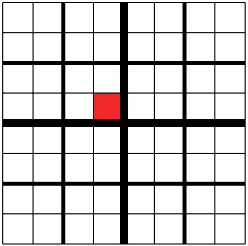


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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

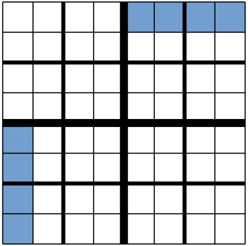


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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

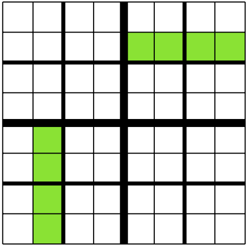


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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
    $A := LU$;

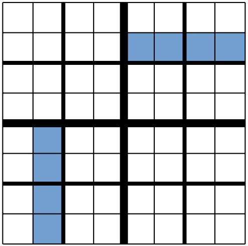


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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

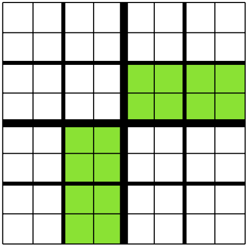


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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
    $A := LU$;

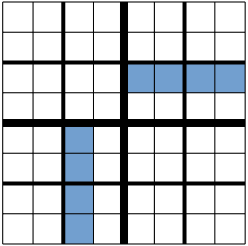


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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
    $A := LU$;

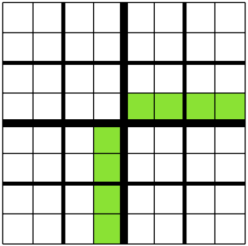


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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

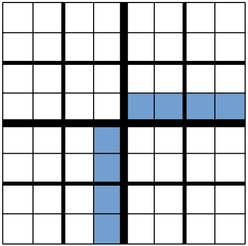


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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
    $A := LU$;

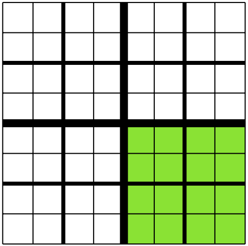


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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
    $A := LU$;

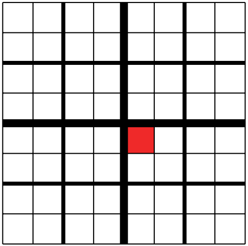


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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

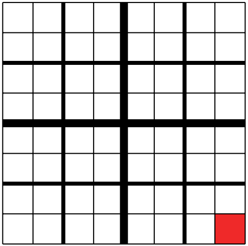


Algorithm Scope

Let $A \in \mathbb{R}^{n \times n}$ be a dense matrix with block size $0 < N < n$.

Using the recursive LU algorithm, the matrix blocks are processed in a *localised* execution order:

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
    $A := LU$;

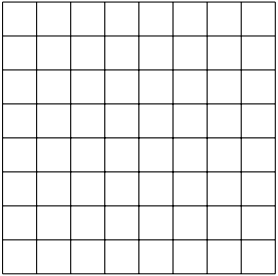


- Only small number of processors may be used at once.
- Local processor sets are synchronised during local computations.
- Global synchronisation for each diagonal block.

Algorithm Scope

Execution order for *global* dense LU factorisation:

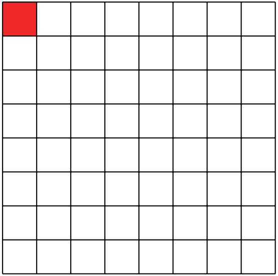
for $0 \leq i < n/N$ **do**
  $A_{ii} = L_{ii}U_{ii}$;
  for $i < j < n/N$ **do**
    SolveLL(A_{ij}, L_{ii}, U_{ij});
    SolveUR(A_{ji}, L_{ji}, U_{ii});
  for $i < j < n/N$ **do**
    for $i < \ell < n/N$ **do**
      $A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}$;



Algorithm Scope

Execution order for *global* dense LU factorisation:

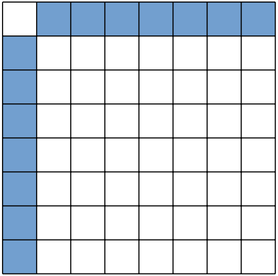
for $0 \leq i < n/N$ **do**
  $A_{ii} = L_{ii} U_{ii}$;
  for $i < j < n/N$ **do**
    $\text{SOLVELL}(\ A_{ij}, L_{ii}, U_{ij}\ )$;
    $\text{SOLVEUR}(\ A_{ji}, L_{ji}, U_{ii}\ )$;
  for $i < j < n/N$ **do**
    for $i < \ell < n/N$ **do**
      $A_{j\ell} := A_{j\ell} - L_{ji} U_{i\ell}$;



Algorithm Scope

Execution order for *global* dense LU factorisation:

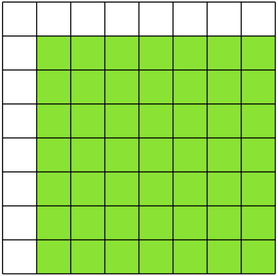
for $0 \leq i < n/N$ **do**
  $A_{ii} = L_{ii} U_{ii}$;
  for $i < j < n/N$ **do**
    $\text{SolveLL}(\ A_{ij}, L_{ii}, U_{ij}\ )$;
    $\text{SolveUR}(\ A_{ji}, L_{ji}, U_{ii}\ )$;
  for $i < j < n/N$ **do**
    for $i < \ell < n/N$ **do**
      $A_{j\ell} := A_{j\ell} - L_{ji} U_{i\ell}$;



Algorithm Scope

Execution order for *global* dense LU factorisation:

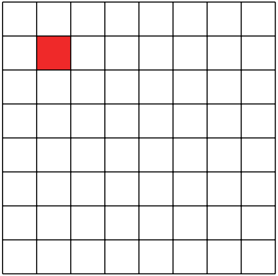
for $0 \leq i < n/N$ **do**
  $A_{ii} = L_{ii}U_{ii}$;
  for $i < j < n/N$ **do**
    $\text{SolveLL}(\ A_{ij}, L_{ii}, U_{ij}\ )$;
    $\text{SolveUR}(\ A_{ji}, L_{ji}, U_{ii}\ )$;
  for $i < j < n/N$ **do**
    for $i < \ell < n/N$ **do**
      $A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}$;



Algorithm Scope

Execution order for *global* dense LU factorisation:

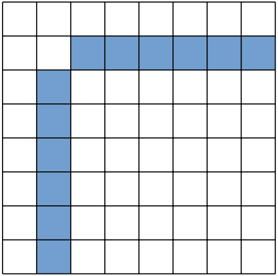
for $0 \le i < n/N$ **do**
  $A_{ii} = L_{ii} U_{ii}$;
  for $i < j < n/N$ **do**
    SolveLL(A_{ij}, L_{ii}, U_{ij});
    SolveUR(A_{ji}, L_{ji}, U_{ii});
  for $i < j < n/N$ **do**
    for $i < \ell < n/N$ **do**
      $A_{j\ell} := A_{j\ell} - L_{ji} U_{i\ell}$;



Algorithm Scope

Execution order for *global* dense LU factorisation:

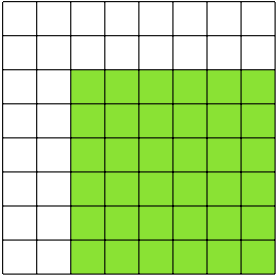
for $0 \le i < n/N$ **do**
  $A_{ii} = L_{ii}U_{ii}$;
  for $i < j < n/N$ **do**
    $\text{SolveLL}(\ A_{ij}, L_{ii}, U_{ij}\ )$;
    $\text{SolveUR}(\ A_{ji}, L_{ji}, U_{ii}\ )$;
  for $i < j < n/N$ **do**
    for $i < \ell < n/N$ **do**
      $A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}$;



Algorithm Scope

Execution order for *global* dense LU factorisation:

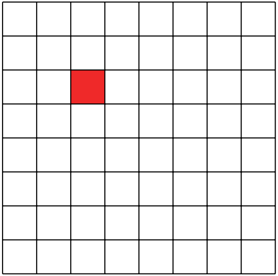
for $0 \leq i < n/N$ **do**
$A_{ii} = L_{ii} U_{ii}$;
for $i < j < n/N$ **do**
$\textsc{SolveLL}(\ A_{ij}, L_{ii}, U_{ij}\ )$;
$\textsc{SolveUR}(\ A_{ji}, L_{ji}, U_{ii}\ )$;
for $i < j < n/N$ **do**
for $i < \ell < n/N$ **do**
$A_{j\ell} := A_{j\ell} - L_{ji} U_{i\ell}$;



Algorithm Scope

Execution order for *global* dense LU factorisation:

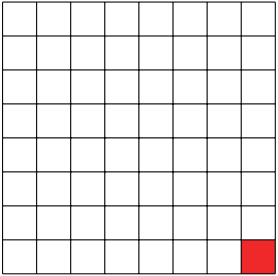
for $0 \leq i < n/N$ **do**
  $A_{ii} = L_{ii}U_{ii}$;
  for $i < j < n/N$ **do**
    $\textsc{SolveLL}(\ A_{ij}, L_{ii}, U_{ij}\ )$;
    $\textsc{SolveUR}(\ A_{ji}, L_{ji}, U_{ii}\ )$;
  for $i < j < n/N$ **do**
    for $i < \ell < n/N$ **do**
      $A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}$;



Algorithm Scope

Execution order for *global* dense LU factorisation:

for $0 \leq i < n/N$ **do**
  $A_{ii} = L_{ii}U_{ii}$;
  for $i < j < n/N$ **do**
    $\text{SolveLL}(\ A_{ij}, L_{ii}, U_{ij}\ )$;
    $\text{SolveUR}(\ A_{ji}, L_{ji}, U_{ii}\ )$;
  for $i < j < n/N$ **do**
    for $i < \ell < n/N$ **do**
      $A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}$;



- At update (and solve) stages, much more processors may be used due to *global* scope.
- Still:
  - synchronisation after solve and update stages and
  - while diagonal factorization only single processor active.

DAG Computation

Every atomic set of operations, executed by a single processor is called a *task*.

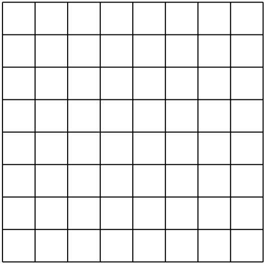
for $0 \leq i < n/N$ **do**
  task($A_{ii} = L_{ii} U_{ii}$);
  for $i < j < n/N$ **do**
    task($\text{SolveLL}(A_{ij}, L_{ii}, U_{ij})$);
    task($\text{SolveUR}(A_{ji}, L_{ji}, U_{ii})$);
  for $i < j < n/N$ **do**
    for $i < \ell < n/N$ **do**
      task($A_{j\ell} := A_{j\ell} - L_{ji} U_{i\ell}$);

Instead of computing the LU factorization, the algorithm only produces the corresponding set of tasks.

DAG Computation

Each task of the LU factorization has input *dependencies*, which have to be fullfilled to permit execution:

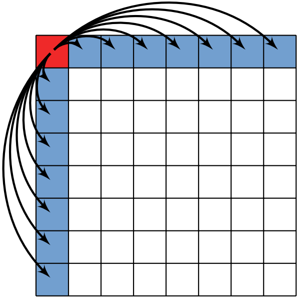
for $0 \leq i < n/N$ **do**
  task($A_{ii} = L_{ii}U_{ii}$);
  for $i < j < n/N$ **do**
    task($\textsc{SolveLL}(\ A_{ij}, L_{ii}, U_{ij}\ )$);
    task($\textsc{SolveUR}(\ A_{ji}, L_{ji}, U_{ii}\ )$);
  for $i < j < n/N$ **do**
    for $i < \ell < n/N$ **do**
      task($A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}$);



DAG Computation

Each task of the LU factorization has input *dependencies*, which have to be fullfilled to permit execution:

for $0 \le i < n/N$ **do**
  task($A_{ii} = L_{ii} U_{ii}$);
  for $i < j < n/N$ **do**
    task($\textsc{SolveLL}(A_{ij}, L_{ii}, U_{ij})$);
    task($\textsc{SolveUR}(A_{ji}, L_{ji}, U_{ii})$);
  for $i < j < n/N$ **do**
    for $i < \ell < n/N$ **do**
      task($A_{j\ell} := A_{j\ell} - L_{ji} U_{i\ell}$);

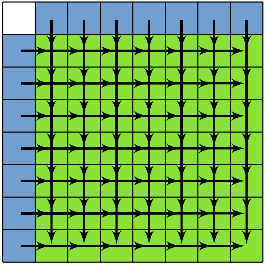


task($A_{ii} = L_{ii} U_{ii}$) ➔ **task**($\textsc{SolveLL}(A_{ij}, L_{ii}, U_{ij})$)

DAG Computation

Each task of the LU factorization has input *dependencies*, which have to be fullfilled to permit execution:

for $0 \leq i < n/N$ **do**
  task($A_{ii} = L_{ii}U_{ii}$);
  for $i < j < n/N$ **do**
    task($\textsc{SolveLL}(\ A_{ij}, L_{ii}, U_{ij}\ )$);
    task($\textsc{SolveUR}(\ A_{ji}, L_{ji}, U_{ii}\ )$);
  for $i < j < n/N$ **do**
    for $i < \ell < n/N$ **do**
      task($A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}$);



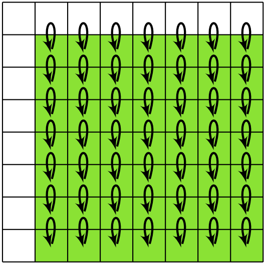
task($A_{ii} = L_{ii}U_{ii}$) → task($\textsc{SolveLL}(A_{ij}, L_{ii}, U_{ij})$)

task($\textsc{SolveLL}(\ A_{i\ell}, L_{ii}, U_{i\ell}\ )$) → task($A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}$)

DAG Computation

Each task of the LU factorization has input *dependencies*, which have to be fullfilled to permit execution:

for $0 \leq i < n/N$ **do**
  task($A_{ii} = L_{ii}U_{ii}$);
  for $i < j < n/N$ **do**
    task($\text{SolveLL}(\ A_{ij}, L_{ii}, U_{ij}\ )$);
    task($\text{SolveUR}(\ A_{ji}, L_{ji}, U_{ii}\ )$);
  for $i < j < n/N$ **do**
    for $i < \ell < n/N$ **do**
      task($A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}$);



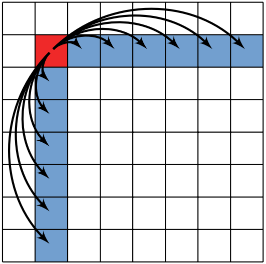
task($A_{ii} = L_{ii}U_{ii}$) ➔ **task**($\text{SolveLL}(A_{ij}, L_{ii}, U_{ij})$)

task($\text{SolveLL}(\ A_{i\ell}, L_{ii}, U_{i\ell}\ )$) ➔ **task**($A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}$)

DAG Computation

Each task of the LU factorization has input *dependencies*, which have to be fullfilled to permit execution:

for $0 \le i < n/N$ **do**
  task$(A_{ii} = L_{ii}U_{ii})$;
  for $i < j < n/N$ **do**
    task$(\textsc{SolveLL}(\ A_{ij}, L_{ii}, U_{ij}\ ))$;
    task$(\textsc{SolveUR}(\ A_{ji}, L_{ji}, U_{ii}\ ))$;
  for $i < j < n/N$ **do**
    for $i < \ell < n/N$ **do**
      task$(A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell})$;



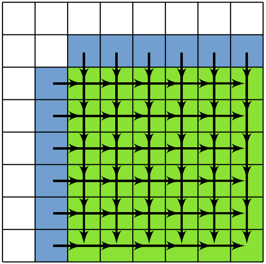
task$(A_{ii} = L_{ii}U_{ii})$ ➜ **task**$(\textsc{SolveLL}(A_{ij}, L_{ii}, U_{ij}))$

task$(\textsc{SolveLL}(\ A_{i\ell}, L_{ii}, U_{i\ell}\ ))$ ➜ **task**$(A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}\ ))$

DAG Computation

Each task of the LU factorization has input *dependencies*, which have to be fullfilled to permit execution:

for $0 \leq i < n/N$ **do**
  task$(A_{ii} = L_{ii}U_{ii})$;
  for $i < j < n/N$ **do**
    task$(\text{SolveLL}(\ A_{ij}, L_{ii}, U_{ij}\ ))$;
    task$(\text{SolveUR}(\ A_{ji}, L_{ji}, U_{ii}\ ))$;
  for $i < j < n/N$ **do**
    for $i < \ell < n/N$ **do**
      task$(A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell})$;



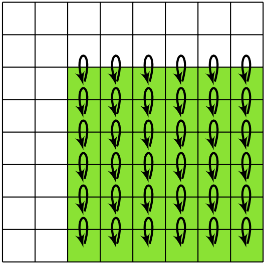
task$(A_{ii} = L_{ii}U_{ii})$ ➞ **task**$(\text{SolveLL}(A_{ij}, L_{ii}, U_{ij}))$

task$(\text{SolveLL}(\ A_{i\ell}, L_{ii}, U_{i\ell}\ ))$ ➞ **task**$(A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}\ ))$

DAG Computation

Each task of the LU factorization has input *dependencies*, which have to be fullfilled to permit execution:

for $0 \leq i < n/N$ **do**
  task($A_{ii} = L_{ii}U_{ii}$);
  for $i < j < n/N$ **do**
    task($\text{SolveLL}(\ A_{ij}, L_{ii}, U_{ij}\ )$);
    task($\text{SolveUR}(\ A_{ji}, L_{ji}, U_{ii}\ )$);
  for $i < j < n/N$ **do**
    for $i < \ell < n/N$ **do**
      task($A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}$);



task($A_{ii} = L_{ii}U_{ii}$) → **task**($\text{SolveLL}(A_{ij}, L_{ii}, U_{ij})$)

task($\text{SolveLL}(\ A_{i\ell}, L_{ii}, U_{i\ell}\ )$) → **task**($A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}$)

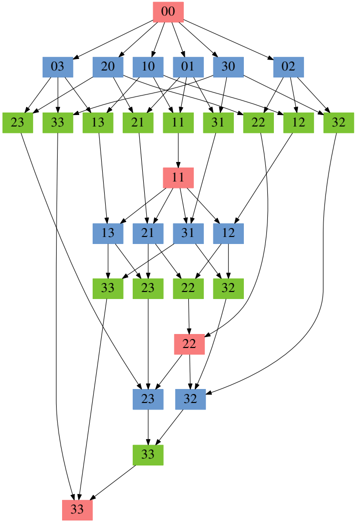
DAG Computation

The previous algorithm can be extended to compute both tasks and their dependencies:

for $0 \leq i < n/N$ **do**
  task($A_{ii} = L_{ii}U_{ii}$);
  for $i < j < n/N$ **do**
    task($\text{SolveLL}(\ A_{ij}, L_{ii}, U_{ij}\ )$);
    task($A_{ii} = L_{ii}U_{ii}$) ➔ **task**($\text{SolveLL}(A_{ij}, L_{ii}, U_{ij})$);
    task($\text{SolveUR}(\ A_{ji}, L_{ji}, U_{ii}\ )$);
    task($A_{ii} = L_{ii}U_{ii}$) ➔ **task**($\text{SolveUR}(A_{ji}, L_{ji}, U_{ii})$);
  for $i < j < n/N$ **do**
    for $i < \ell < n/N$ **do**
      task($A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}$);
      task($\text{SolveUR}(A_{ji}, L_{ji}, U_{ii})$) ➔ **task**($A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}$);
      task($\text{SolveLL}(A_{i\ell}, L_{ii}, U_{i\ell})$) ➔ **task**($A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}$);
      if $j = \ell$ **then**
        task($A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}$) ➔ **task**($A_{jj} = L_{jj}U_{jj}$);
      else if $j > \ell$ **then**
        task($A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}$) ➔ **task**($\text{SolveUR}(A_{j\ell}, L_{j\ell}, U_{\ell\ell})$);
      else
        task($A_{j\ell} := A_{j\ell} - L_{ji}U_{i\ell}$) ➔ **task**($\text{SolveLL}(A_{j\ell}, L_{jj}, U_{j\ell})$);

DAG Computation

Tasks and dependencies form a *directed acyclic graph* (DAG).



(DAG for a 4×4 matrix)

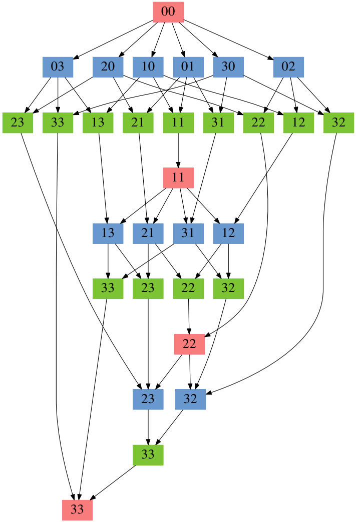
DAG Computation

Tasks and dependencies form a *directed acyclic graph* (DAG).

DAG execution

As soon as all dependencies for a task are met, it is scheduled for execution.

- avoids redundant synchronisations,
- execution of different types of tasks may *overlap*.



(DAG for a 4×4 matrix)

DAG Computation

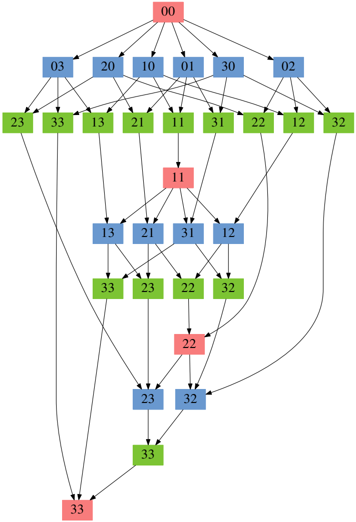
Tasks and dependencies form a *directed acyclic graph* (DAG).

DAG execution

As soon as all dependencies for a task are met, it is scheduled for execution.

- avoids redundant synchronisations,
- execution of different types of tasks may *overlap*.

DAG definition is hardware *independent*. DAG execution (task scheduling) may be optimised for specific systems.



(DAG for a 4×4 matrix)

Task based $\mathcal{H}$-Arithmetic

Definitions

For $t \in T(I)$ let $\mathrm{level}(t)$ denote the distance of t from the root of $T(I)$. Furthermore, let

$$T^\ell(I) := \{s \in T(I) \ : \ \mathrm{level}(s) = \ell\}$$

be the set of clusters on level ℓ.

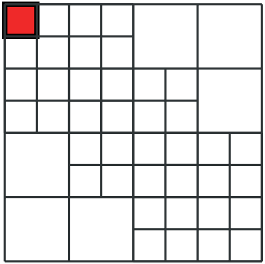
For $t, s \subset I$ we define $>_I$ by:

$$s >_I t \text{ iff } \forall i \in t, i' \in s : i' > i$$

Global Algorithm

The $\mathcal{H}$-LU algorithm with global scope computes matrix solves and updates for *all* blocks accessible by the current block row/column on the *same* level as the diagonal block.

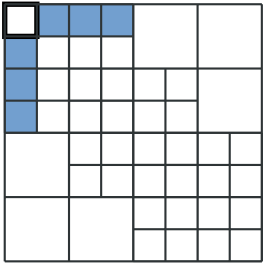
procedure $\mathrm{LU}(\ A|_{t\times t},\ L|_{t\times t}, U|_{t\times t}\ )$
  if A is block matrix **then**
    for $i \in \{0,1\}$ **do**
      $\mathrm{LU}(\ A|_{t_i\times t_i}\ );\ \ell := \mathrm{level}(t_i);$
      for $s \in T^{\ell}(I), s >_I t_i$ **do**
        $\textsc{SolveUR}(\ A|_{s\times t_i}, L|_{s\times t_i}, U|_{t_i\times t_i}\ );$
        $\textsc{SolveLL}(\ A|_{t_i\times s}, L|_{t_i\times t_i}, U|_{t_i\times s}\ );$
      for $s, r \in T^{\ell}(I), s, r >_I t_i$ **do**
        $\textsc{Multiply}(-1, L_{r\times t_i}, U_{t_i\times s}, A|_{r\times s});$
  else
    $U := A^{-1}; L := I;$



Global Algorithm

The $\mathcal{H}$-LU algorithm with global scope computes matrix solves and updates for *all* blocks accessible by the current block row/column on the *same* level as the diagonal block.

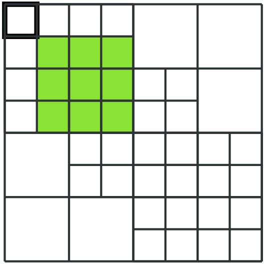
procedure $\mathrm{LU}(\ A|_{t\times t},\ L|_{t\times t}, U|_{t\times t}\ )$
  if A is block matrix **then**
    for $i \in \{0,1\}$ **do**
      $\mathrm{LU}(\ A|_{t_i\times t_i}\ )$; $\ell := \mathrm{level}(t_i)$;
      for $s \in T^{\ell}(I), s >_I t_i$ **do**
        $\textsc{SolveUR}(\ A|_{s\times t_i}, L|_{s\times t_i}, U|_{t_i\times t_i}\ )$;
        $\textsc{SolveLL}(\ A|_{t_i\times s}, L|_{t_i\times t_i}, U|_{t_i\times s}\ )$;
      for $s, r \in T^{\ell}(I), s, r >_I t_i$ **do**
        $\textsc{Multiply}(-1, L_{r\times t_i}, U_{t_i\times s}, A|_{r\times s})$;
  else
    $U := A^{-1}; L := I$;



Global Algorithm

The $\mathcal{H}$-LU algorithm with global scope computes matrix solves and updates for *all* blocks accessible by the current block row/column on the *same* level as the diagonal block.

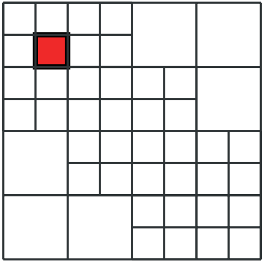
procedure $\mathrm{LU}(\ A|_{t \times t},\ L|_{t \times t}, U|_{t \times t}\ )$
  if A is block matrix **then**
    for $i \in \{0, 1\}$ **do**
      $\mathrm{LU}(\ A|_{t_i \times t_i}\ );\ \ell := \mathrm{level}(t_i);$
      for $s \in T^{\ell}(I), s >_I t_i$ **do**
        $\textsc{SolveUR}(\ A|_{s \times t_i}, L|_{s \times t_i}, U|_{t_i \times t_i}\ );$
        $\textsc{SolveLL}(\ A|_{t_i \times s}, L|_{t_i \times t_i}, U|_{t_i \times s}\ );$
      for $s, r \in T^{\ell}(I), s, r >_I t_i$ **do**
        $\textsc{Multiply}(-1, L_{r \times t_i}, U_{t_i \times s}, A|_{r \times s});$
  else
    $U := A^{-1}; L := I;$



Global Algorithm

The $\mathcal{H}$-LU algorithm with global scope computes matrix solves and updates for *all* blocks accessible by the current block row/column on the *same* level as the diagonal block.

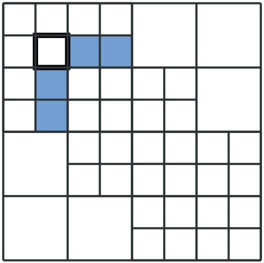
procedure $\mathrm{LU}(\ A|_{t\times t},\ L|_{t\times t}, U|_{t\times t}\ )$
  if A is block matrix **then**
    for $i \in \{0,1\}$ **do**
      $\mathrm{LU}(\ A|_{t_i\times t_i}\ );\ \ell := \mathrm{level}(t_i);$
      for $s \in T^\ell(I), s >_I t_i$ **do**
        $\textsc{SolveUR}(\ A|_{s\times t_i}, L|_{s\times t_i}, U|_{t_i\times t_i}\ );$
        $\textsc{SolveLL}(\ A|_{t_i\times s}, L|_{t_i\times t_i}, U|_{t_i\times s}\ );$
      for $s, r \in T^\ell(I), s, r >_I t_i$ **do**
        $\textsc{Multiply}(-1, L_{r\times t_i}, U_{t_i\times s}, A|_{r\times s});$
  else
    $U := A^{-1}; L := I;$



Global Algorithm

The $\mathcal{H}$-LU algorithm with global scope computes matrix solves and updates for *all* blocks accessible by the current block row/column on the *same* level as the diagonal block.

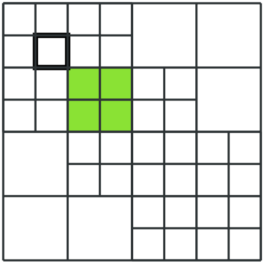
procedure $\mathrm{LU}(\ A|_{t\times t},\ L|_{t\times t}, U|_{t\times t}\ )$
  if A is block matrix **then**
    for $i \in \{0,1\}$ **do**
      $\mathrm{LU}(\ A|_{t_i\times t_i}\ );\ \ell := \mathrm{level}(t_i);$
      for $s \in T^{\ell}(I), s >_I t_i$ **do**
        $\textsc{SolveUR}(\ A|_{s\times t_i}, L|_{s\times t_i}, U|_{t_i\times t_i}\ );$
        $\textsc{SolveLL}(\ A|_{t_i\times s}, L|_{t_i\times t_i}, U|_{t_i\times s}\ );$
      for $s, r \in T^{\ell}(I), s, r >_I t_i$ **do**
        $\textsc{Multiply}(-1, L_{r\times t_i}, U_{t_i\times s}, A|_{r\times s});$
  else
    $U := A^{-1}; L := I;$



Global Algorithm

The $\mathcal{H}$-LU algorithm with global scope computes matrix solves and updates for *all* blocks accessible by the current block row/column on the *same* level as the diagonal block.

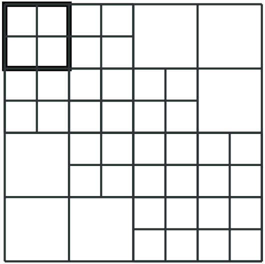
procedure $\mathrm{LU}(\ A|_{t\times t},\ L|_{t\times t}, U|_{t\times t}\ )$
  if A is block matrix **then**
    for $i \in \{0,1\}$ **do**
      $\mathrm{LU}(\ A|_{t_i\times t_i}\ );\ \ell := \mathrm{level}(t_i);$
      for $s \in T^\ell(I), s >_I t_i$ **do**
        $\textsc{SolveUR}(\ A|_{s\times t_i}, L|_{s\times t_i}, U|_{t_i\times t_i}\ );$
        $\textsc{SolveLL}(\ A|_{t_i\times s}, L|_{t_i\times t_i}, U|_{t_i\times s}\ );$
      for $s, r \in T^\ell(I), s, r >_I t_i$ **do**
        $\textsc{Multiply}(-1, L_{r\times t_i}, U_{t_i\times s}, A|_{r\times s});$
  else
    $U := A^{-1}; L := I;$



Global Algorithm

The $\mathcal{H}$-LU algorithm with global scope computes matrix solves and updates for *all* blocks accessible by the current block row/column on the *same* level as the diagonal block.

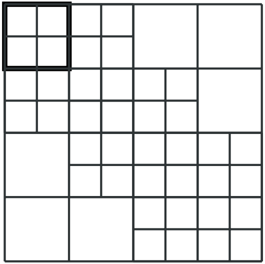
procedure $\mathrm{LU}(A|_{t \times t}, L|_{t \times t}, U|_{t \times t})$
  if A is block matrix **then**
    for $i \in \{0, 1\}$ **do**
      $\mathrm{LU}(A|_{t_i \times t_i})$; $\ell := \mathrm{level}(t_i)$;
      for $s \in T^\ell(I), s >_I t_i$ **do**
        $\textsc{SolveUR}(A|_{s \times t_i}, L|_{s \times t_i}, U|_{t_i \times t_i})$;
        $\textsc{SolveLL}(A|_{t_i \times s}, L|_{t_i \times t_i}, U|_{t_i \times s})$;
      for $s, r \in T^\ell(I), s, r >_I t_i$ **do**
        $\textsc{Multiply}(-1, L_{r \times t_i}, U_{t_i \times s}, A|_{r \times s})$;
  else
    $U := A^{-1}; L := I$;



Global Algorithm

The $\mathcal{H}$-LU algorithm with global scope computes matrix solves and updates for *all* blocks accessible by the current block row/column on the *same* level as the diagonal block.

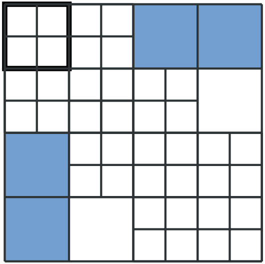
procedure $\mathrm{LU}(\ A|_{t\times t},\ L|_{t\times t}, U|_{t\times t}\ )$
  if A is block matrix **then**
    for $i \in \{0,1\}$ **do**
      $\mathrm{LU}(\ A|_{t_i\times t_i}\ );\ \ell := \mathrm{level}(t_i);$
      for $s \in T^{\ell}(I), s >_I t_i$ **do**
        if $A|_{s\times t_i}$ is not blocked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



Global Algorithm

The $\mathcal{H}$-LU algorithm with global scope computes matrix solves and updates for *all* blocks accessible by the current block row/column on the *same* level as the diagonal block.

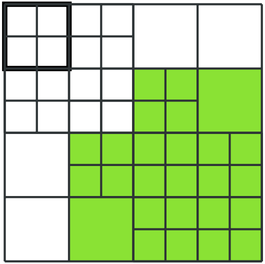
procedure $\mathrm{LU}(\ A|_{t\times t},\ L|_{t\times t}, U|_{t\times t}\ )$
  if A is block matrix **then**
    for $i \in \{0,1\}$ **do**
      $\mathrm{LU}(\ A|_{t_i\times t_i}\ );\ \ell := \mathrm{level}(t_i);$
      for $s \in T^\ell(I), s >_I t_i$ **do**
        if $A|_{s\times t_i}$ is not blocked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



Global Algorithm

The $\mathcal{H}$-LU algorithm with global scope computes matrix solves and updates for *all* blocks accessible by the current block row/column on the *same* level as the diagonal block.

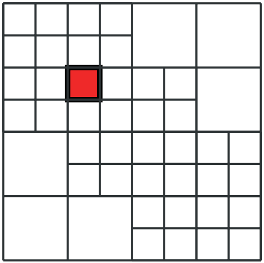
procedure $\mathrm{LU}(\ A|_{t\times t},\ L|_{t\times t}, U|_{t\times t}\ )$
  if A is block matrix **then**
    for $i \in \{0,1\}$ **do**
      $\mathrm{LU}(\ A|_{t_i\times t_i}\ )$; $\ell := \mathrm{level}(t_i)$;
      for $s \in T^{\ell}(I), s >_I t_i$ **do**
        if $A|_{s\times t_i}$ is not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



Global Algorithm

The $\mathcal{H}$-LU algorithm with global scope computes matrix solves and updates for *all* blocks accessible by the current block row/column on the *same* level as the diagonal block.

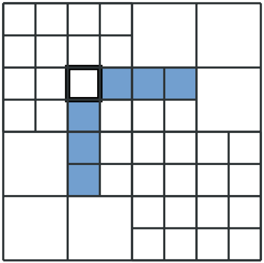
procedure $\mathrm{LU}(\ A|_{t\times t},\ L|_{t\times t}, U|_{t\times t}\ )$
  if A is block matrix **then**
    for $i \in \{0,1\}$ **do**
      $\mathrm{LU}(\ A|_{t_i\times t_i}\ );\ \ell := \mathrm{level}(t_i);$
      for $s \in T^\ell(I), s >_I t_i$ **do**
        if $A|_{s\times t_i}$ is not blocked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



Global Algorithm

The $\mathcal{H}$-LU algorithm with global scope computes matrix solves and updates for *all* blocks accessible by the current block row/column on the *same* level as the diagonal block.

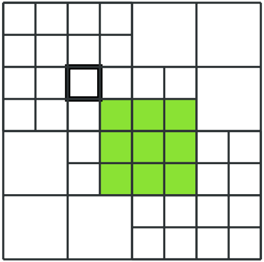
procedure $\mathrm{LU}(\ A|_{t \times t},\ L|_{t \times t}, U|_{t \times t}\ )$
  if A is block matrix **then**
    for $i \in \{0, 1\}$ **do**
      $\mathrm{LU}(\ A|_{t_i \times t_i}\ );\ \ell := \mathrm{level}(t_i);$
      for $s \in T^{\ell}(I), s >_I t_i$ **do**
        if $A|_{s \times t_i}$ is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



Global Algorithm

The $\mathcal{H}$-LU algorithm with global scope computes matrix solves and updates for *all* blocks accessible by the current block row/column on the *same* level as the diagonal block.

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



Task Dependencies

Factorize → Solve

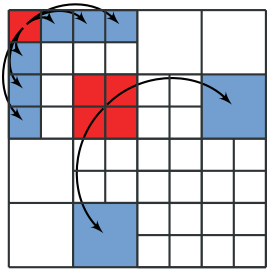
Let $t \in T(I)$ and $\ell := \mathrm{level}(t)$. Then $\forall s \in T^{\ell}(I), s >_I t$:

$s \times t \in \mathcal{L}(T(I \times I)) \implies$

task$(\mathrm{LU}(A|_{t \times t}))$ ➔ **task**$(\textsc{SolveUR}(A|_{s \times t}, L|_{s \times t}, U|_{t \times t}))$,

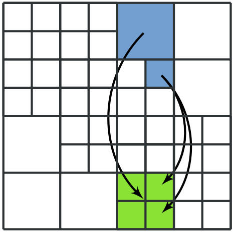
$t \times s \in \mathcal{L}(T(I \times I)) \implies$

task$(\mathrm{LU}(A|_{t \times t}))$ ➔ **task**$(\textsc{SolveLL}(A|_{t \times s}, L|_{t \times t}, U|_{t \times s}))$,



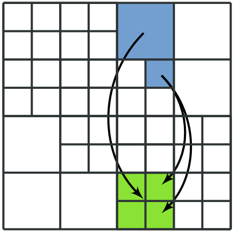
Task Dependencies

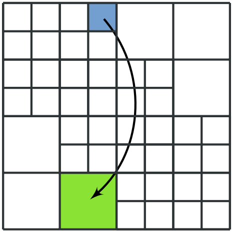
Solve ➞ Update



Task Dependencies

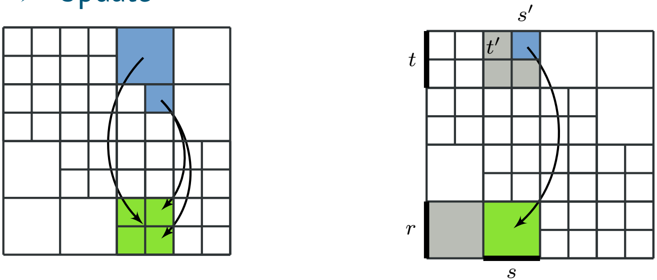
Solve ➔ Update





Task Dependencies

Solve ➔ Update



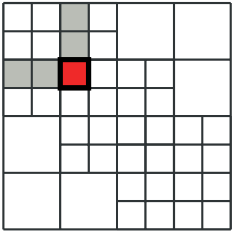
Let $t \in T(I)$ and $r, s \in T^\ell(I), r, s >_I t$ such that a $\textsc{Multiply}(-1, L|_{r\times t}, U|_{t\times s}, A|_{r\times s})$ task exists.

Then for all $t' \times s' \subseteq t \times s$ with a solve task $\textsc{SolveLL}(A|_{t'\times s'}, L|_{t'\times t'}, U|_{t'\times s'})$ we have:

$$\textbf{task}(\textsc{SolveLL}(A|_{t'\times s'}, L|_{t'\times t'}, U|_{t'\times s'})) \rightarrow \textbf{task}(\textsc{Multiply}(-1, L|_{r\times t}, U|_{t\times s}, A|_{r\times s})).$$

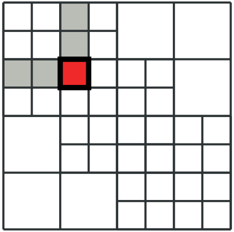
Task Dependencies

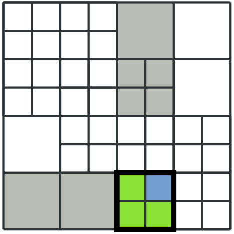
Update ➛ Factorize/Solve



Task Dependencies

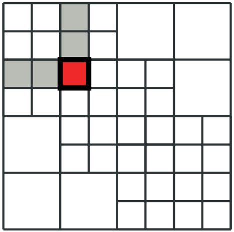
Update → Factorize/Solve

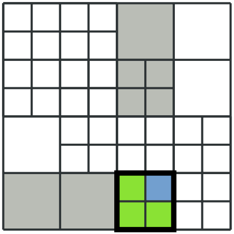




Task Dependencies

Update ➛ Factorize/Solve





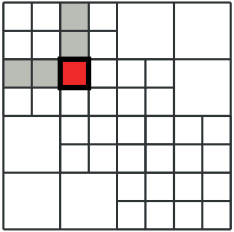
Destination blocks of update tasks may be

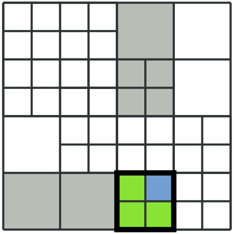
- identical to,
- on levels *above*

blocks for factorise/solve tasks

Task Dependencies

Update ➛ Factorize/Solve



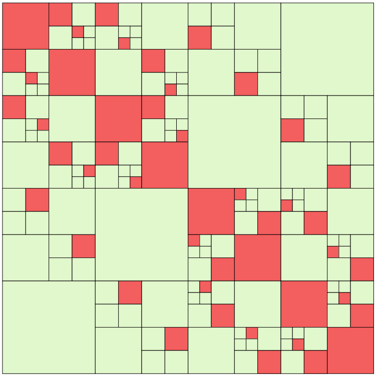


Destination blocks of update tasks may be

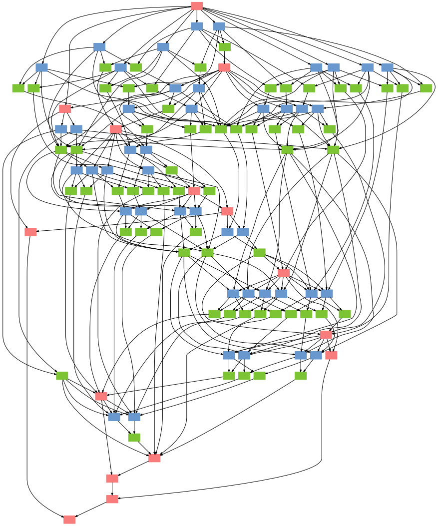
- identical to,
- on levels *above* or
- on levels *below*

blocks for factorise/solve tasks

Example



$\mathcal{H}$-matrix



$\mathcal{H}$-LU DAG

Numerical Results

Parallel speedup for model problem with $n = 32.768$:

	Speedup
Intel Xeon E5-2670 (2x8 cores, 2 hyperthreads)	18.04
Intel XeonPhi 5110P (60 cores, 4 hyperthreads)	106.18

Function trace

(Xeon E5-2640)

$\mathcal{H}$-Inversion

Inversion of an $\mathcal{H}$-Matrix A may be computed using *Gaussian elimination*.

However, this requires additional storage (an extra $\mathcal{H}$-matrix for temporary computations).

Alternatively, an $\mathcal{H}$-LU based algorithm is available:

```
procedure INVERT( A )
  LU( A, L, U );                 // A → LU
  INVERTLL( L );                 // L → L^{-1}
  INVERTUR( U );                 // U → U^{-1}
  MULTIPLYURLL( U, L, A );       // U^{-1} L^{-1} → A^{-1}
```

Here, all operations are computed *in-place*, i.e., A is used to store L and U.

Triangular $\mathcal{H}$-Inversion

Let U be an upper triangular $\mathcal{H}$-matrix:

$$U = \begin{pmatrix} U_{00} & U_{01} \\ & U_{11} \end{pmatrix}$$

For the inverse $B = U^{-1}$ we have:

$$\begin{pmatrix} U_{00} & U_{01} \\ & U_{11} \end{pmatrix} \begin{pmatrix} B_{00} & B_{01} \\ & B_{11} \end{pmatrix} = \begin{pmatrix} I & 0 \\ 0 & I \end{pmatrix}$$

which results in the equations

$$U_{00} B_{00} = I \qquad B_{01} = -B_{00} U_{01} B_{11}$$
$$U_{11} B_{11} = I$$

Triangular $\mathcal{H}$-Inversion

Properties of the DAG

- *all* dense diagonal blocks are start nodes,
- off-diagonal blocks have dependencies to row *and* column diagonal blocks.

Triangular $\mathcal{H}$-Inversion

Properties of the DAG

- *all* dense diagonal blocks are start nodes,
- off-diagonal blocks have dependencies to row *and* column diagonal blocks.

Numerical Results

Parallel speedup for model problem with $n = 32.768$:

	Speedup
Intel Xeon E5-2670 (2x8 cores, 2 hyperthreads)	16.52
Intel XeonPhi 5110P (60 cores, 4 hyperthreads)	69.20

$\mathcal{H}$-Matrix-Vector Multiplication

Let $A = LU$ be an $\mathcal{H}$-LU factorization of an $\mathcal{H}$-matrix A. Fhe following parallel speedup for solving $Ax = y$ for some RHS y is:

	Speedup
Intel Xeon E5-2670 (2x8 cores, 2 hyperthreads)	1.46
Intel XeonPhi 5110P (60 cores, 4 hyperthreads)	2.64

$\mathcal{H}$-Matrix-Vector Multiplication

Let $A = LU$ be an $\mathcal{H}$-LU factorization of an $\mathcal{H}$-matrix A. Fhe following parallel speedup for solving $Ax = y$ for some RHS y is:

	Speedup
Intel Xeon E5-2670 (2x8 cores, 2 hyperthreads)	1.46
Intel XeonPhi 5110P (60 cores, 4 hyperthreads)	2.64

For solving $x = A^{-1}y = U^{-1}L^{-1}y$ one obtains:

	Speedup
Intel Xeon E5-2670 (2x8 cores, 2 hyperthreads)	9.36
Intel XeonPhi 5110P (60 cores, 4 hyperthreads)	101.23

Triangular Matrix Multiplication

Let L be a lower triangular $\mathcal{H}$-matrix and U an upper triangular $\mathcal{H}$-matrix. For the product $U \cdot L$ we have:

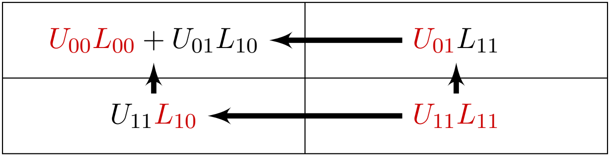
$$\begin{pmatrix} U_{00} & U_{01} \\ & U_{11} \end{pmatrix} \begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} = \begin{pmatrix} U_{00}L_{00} + U_{01}L_{10} & U_{01}L_{11} \\ U_{11}L_{10} & U_{11}L_{11} \end{pmatrix}$$

Triangular Matrix Multiplication

Let L be a lower triangular $\mathcal{H}$-matrix and U an upper triangular $\mathcal{H}$-matrix. For the product $U \cdot L$ we have:

$$\begin{pmatrix} U_{00} & U_{01} \\ & U_{11} \end{pmatrix} \begin{pmatrix} L_{00} & \\ L_{10} & L_{11} \end{pmatrix} = \begin{pmatrix} U_{00}L_{00} + U_{01}L_{10} & U_{01}L_{11} \\ U_{11}L_{10} & U_{11}L_{11} \end{pmatrix}$$

The multiplication should be performed *in-place*, which results in the following dependencies:



Triangular Matrix Multiplication

Parallel speedup for triangular matrix multiplication:

	Speedup
Intel Xeon E5-2670 (2x8 cores, 2 hyperthreads)	16.54
Intel XeonPhi 5110P (60 cores, 4 hyperthreads)	72.83

Triangular Matrix Multiplication

Parallel speedup for triangular matrix multiplication:

	Speedup
Intel Xeon E5-2670 (2x8 cores, 2 hyperthreads)	16.54
Intel XeonPhi 5110P (60 cores, 4 hyperthreads)	72.83

Parallel speedup for full $\mathcal{H}$-inversion:

	Speedup
Intel Xeon E5-2670 (2x8 cores, 2 hyperthreads)	16.46
Intel XeonPhi 5110P (60 cores, 4 hyperthreads)	66.04

Conclusion

Conclusion

What was discussed

- Identified problems of classical recursive $\mathcal{H}$-arithmetic for parallelization on many-core systems.
- Reformulated $\mathcal{H}$-algorithms to have global scope.
- Defined tasks and dependencies between tasks for DAG computations.

Conclusion

What was discussed

- Identified problems of classical recursive $\mathcal{H}$-arithmetic for parallelization on many-core systems.
- Reformulated $\mathcal{H}$-algorithms to have global scope.
- Defined tasks and dependencies between tasks for DAG computations.

Outlook

- Apply technique to $\mathcal{H}^2$-arithmetic.
- Combine shared and distributed memory with DAG formulation.
- Automatic DAG-generation based on recursive algorithm.

Literature

R. Kriemann, S. Le Borne,
H-FAINV: Hierarchically factored approximate inverse preconditioners,
Computing and Visualization in Science, to appear.

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

R. Kriemann,
Parallel $\mathcal{H}$-Matrix Arithmetics on Shared Memory Systems,
Computing, 74:273–297, 2005.